Missouri S&T Library and Learning Resources

Scholars' Mine

Masters Theses

Student Theses and Dissertations

Summer 2005

Bio-inspired approaches for critical infrastructure protection: Application of clonal selection principle for intrusion detection and FACTS placement

Kasthurirangan Parthasarathy

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

Part of the Computer Sciences Commons

Department:

Recommended Citation

Parthasarathy, Kasthurirangan, "Bio-inspired approaches for critical infrastructure protection: Application of clonal selection principle for intrusion detection and FACTS placement" (2005). *Masters Theses*. 3714.
https://scholarsmine.mst.edu/masters_theses/3714

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

BIO-INSPIRED APPROACHES FOR CRITICAL INFRASTRUCTURE PROTECTION: APPLICATION OF CLONAL SELECTION PRINCIPLE FOR INTRUSION DETECTION AND FACTS PLACEMENT

by

KASTHURIRANGAN PARTHASARATHY

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI–ROLLA

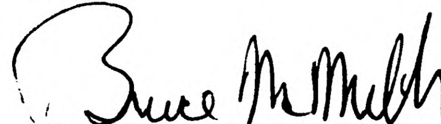
in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

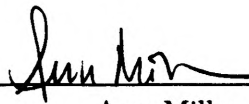
2005

Approved by

Daniel R. Tauritz, Advisor



Bruce McMillin



Ann Miller

Copyright 2005
KASTHURIRANGAN PARTHASARATHY
All Rights Reserved

ABSTRACT

In this research, Clonal Selection, an immune system inspired approach, is utilized along with Evolutionary Algorithms to solve complex engineering problems such as Intrusion Detection and optimization of Flexible AC Transmission System (FACTS) device placement in a power grid. The clonal selection principle increases the strength of good solutions and alters their properties to find better solutions in a problem space. A special class of evolutionary algorithms that utilizes the clonal selection principle to guide its heuristic search process is termed Clonal EA. Clonal EAs can be used to solve complex pattern recognition and function optimization problems, which involve searching an enormous problem space for a solution. Intrusion Detection is modeled, in this research, as a pattern recognition problem wherein efficient detectors are to be designed to detect intrusive behavior. Optimization of FACTS device placement in a power grid is modeled as a function optimization problem wherein optimal placement positions for FACTS devices are to be determined, in order to balance load across power lines. Clonal EAs are designed to implement the solution models. The benefits and limitations of using Clonal EAs to solve the above mentioned problems are discussed and the performance of Clonal EAs is compared with that of traditional evolutionary algorithms and greedy algorithms.

ACKNOWLEDGMENTS

I would like to thank my research advisors, Dr. Daniel Tauritz, Dr. Bruce McMillin and Dr. Ann Miller, for their continuous support and guidance throughout this research project. Dr. Daniel Tauritz was a great source of motivation and was primarily responsible for transforming me from being a naive student to a competitive professional. I would like to acknowledge his exceptional attention to detail and passion for perfection which made this research a success.

I am indebted to my parents, Mr. and Mrs. Parthasarathy, my uncle Mr. Venugopal and my grandfather Mr. Srinivasan for their invaluable support, constant guidance and inspirational mentoring without which this research would not have been possible.

A special thanks to Archana Vasudevan, my soon to be wife, who helped cope with a difficult phase in my life and complete this research through her compassionate presence, constant encouragement and words of wisdom!

I would also like to thank William Atkins, John C. Mulder, Scott A. Miller and many other friends who have immensely contributed to making this report as correct and comprehensive as possible.

I acknowledge the support and company of my friends, especially that of Pradeep and Karthikeyan, which helped me unwind and focus on my research after several busy days at work.

Finally, I would like to thank the University of Missouri-Rolla for providing me the opportunity and facilities to complete this exciting research.

TABLE OF CONTENTS

Page

ABSTRACT . . . iii

ACKNOWLEDGMENTS . . . iv

LIST OF ILLUSTRATIONS . . . vii

LIST OF TABLES . . . viii

NOMENCLATURE . . . ix

SECTION

1. INTRODUCTION . . . 1
2. CLONAL SELECTION AND EVOLUTIONARY ALGORITHMS . . . 3
3. INTRUSION DETECTION . . . 11
   - 3.1. DEFINITION OF SELF . . . 13
   - 3.2. NEGATIVE DETECTOR SET GENERATION . . . 15
     - 3.2.1. Exponential Algorithm. . . . 17
     - 3.2.2. Greedy Algorithm. . . . 18
   - 3.3. POSITIVE DETECTOR SET GENERATION . . . 25
   - 3.4. POSITIVE VS. NEGATIVE DETECTION . . . 31
   - 3.5. EXPERIMENTS . . . 34
     - 3.5.1. Experiments with Exponential Algorithm. . . . 35
     - 3.5.2. Experiments with Greedy Algorithm. . . . 36
   - 3.6. RESULTS AND DISCUSSION . . . 37
   - 3.7. APPLICATION OF CLONAL EAS AND SELF/NON-SELF PRINCIPLES . . . 39
     - 3.7.1. Data Preparation. . . . 40
       - 3.7.1.1. Data Separation. . . . 42
       - 3.7.1.2. Principal Component Analysis. . . . 43
     - 3.7.2. IDS Design. . . . 43
       - 3.7.2.1. Generator. . . . 43
       - 3.7.2.2. Evaluator. . . . 48
     - 3.7.3. Results. . . . 49
     - 3.7.4. Conclusion. . . . 50
4. FACTS PLACEMENT IN A POWER GRID . . . 55
   - 4.1. THE POWER GRID MODEL . . . 55
     - 4.1.1. Power Grid. . . . 55
     - 4.1.2. Component Representation. . . . 57
     - 4.1.3. Identification of Overloaded Lines. . . . 58
     - 4.1.4. Metrics for Evaluation of FACTS Device Configuration. . . . 59

4.1.5. Placement of a FACTS Device on the Grid. 59
4.2. CLONALG DESIGN 60
4.3. EA DESIGN 63
4.4. GREEDY ALGORITHMS 65
4.4.1. Greedy Algorithm based on Number of Overloaded Lines. . . . 65
4.4.2. Greedy Algorithm based on Amount of Line Overload. 65
4.5. EXPERIMENTS 66
4.6. RESULTS AND DISCUSSION 66
5. CONCLUSION 79
6. FUTURE WORK 80
BIBLIOGRAPHY 82
VITA 84

LIST OF ILLUSTRATIONS

Figure Page

2.1 Comparative Features of a Traditional EA and a Clonal EA 10

3.1 A Set Theoretical Representation of the System Definition 12

3.2 An Example to Illustrate PD and ND Schemes 33

3.3 Performance Comparison of ND and PD for FPs - ND: Exponential Algorithm (Algorithm 3), PD: Random Algorithm (Algorithm 12) 38

3.4 Performance Comparison of ND and PD for FNs - ND: Exponential Algorithm (Algorithm 3), PD: Random Algorithm (Algorithm 12) 39

3.5 Performance Comparison of ND and PD for P_fs - ND: Exponential Algorithm (Algorithm 3), PD: Random algorithm (Algorithm 12) 40

3.6 Performance Comparison of ND and PD for P_fs - ND: Greedy Algorithm (Algorithm 4), PD: Random Algorithm (Algorithm 12) and $l = 16$ 41

3.7 Performance Comparison of ND and PD for P_fs - ND: Greedy Algorithm (Algorithm 4), PD: Random Algorithm (Algorithm 12) and $l = 32$ 42

3.8 A Sample Data Schema with Critical Attributes 42

4.1 The IEEE 118-bus test system 56

4.2 Comparison of Results obtained using Various Algorithms 78

LIST OF TABLES

Table Page

2.1 Differences between Clonal EA and Traditional EA 9

3.1 Mapping of Labels to Activities 14

3.2 Mapping of Labels to Activities 14

3.3 Exhaustive List of Templates for $r = 3$ 23

3.4 Input Parameters for Experiments using Algorithm 3 36

3.5 Performance of the Exponential Non-self Algorithm (Algorithm 3) and Random Positive Algorithm (Algorithm 12) 36

3.6 Performance of the Greedy Non-self Algorithm (Algorithm 4) and Random Positive Algorithm (Algorithm 12). The values within parenthesis in Column 4 are the standard deviations for 100 sample runs. 37

3.7 IDSClonalg Results for Self, Non-self Detectors. (* Percentage values of results averaged over 50 trials, Values in parenthesis represent standard deviations) 50

4.1 Parameter Values for Grid Configuration 66

4.2 Parameter Values for Clonalg 67

4.3 Parameter Values for EA 67

4.4 Labels for FACTS Device Placements on the Grid 68

4.5 FACTS Device Placements obtained using Clonalg 69

4.6 FACTS Device Placements obtained using the Evolutionary Algorithm 69

4.7 FACTS Device Placements obtained using Greedy Algorithm based on Number of Overloaded Lines 74

4.8 FACTS Device Placements obtained using Greedy Algorithm based on Amount of Overload 75

NOMENCLATURE

Symbol	Description	Page
EAs	Evolutionary Algorithms	1
FACTS	Flexible AC Transmission System	1
IDS	Intrusion Detection Systems	1
Clonal EA	Evolutionary Algorithm using clonal selection operators	2
WBCs	White Blood Cells	3
PD	Positive Detection	11
ND	Negative Detection	11
AIS	Artificial Immune System	11
Self	Activities that are considered normal	12
Non-self	Activities that are considered undesirable/abnormal	12
FP	False Positives	32
ES	Estimated Self	32
FN	False Negatives	33
P_f	Number of Incorrect Classifications	34
Power Grid	A system for distribution of power/electricity	55
Lines	Power transmission lines in a power grid	55
SLC	single line contingency	58

1. INTRODUCTION

The application of bio-inspired principles to solve complex engineering problems has been gaining significance of late. Evolutionary Algorithms (EAs), inspired by the theory of evolution, have been applied, with resounding success, in the search for solutions in an enormous problem space with no known deterministic solutions. The Clonal Selection principle, inspired by the human immune system, has also been studied extensively of late. This principle involves taking a set of solutions to a problem and subjecting good solutions to minute alterations in an effort to discover better solutions and subjecting bad solutions to relatively larger alterations which might result in the discovery of better solutions. The objective of this research is to determine how an evolutionary algorithm designed using the clonal selection principle performs in searching a problem space, as the clonal selection principle seems to possess the ability to refine and direct a heuristic search process efficiently. In this research, Clonal EAs, combining the clonal selection principle with evolutionary algorithms, are designed to solve two complex engineering problems, namely intrusion detection and optimization of Flexible AC Transmission System (FACTS) devices placement in a power grid.

Intrusion Detection Systems (IDS) are responsible for protecting sensitive data and information. One type of IDS involves defining normal behavior in a computer network, henceforth termed system, and identifying behavior that is deviant from normal behavior. Typically, unauthorized activities, errors and irregularities in a system can be considered as deviant behaviors. These deviant behaviors are termed intrusions. Intrusion detectors, further simply referred to as detectors, which represent either normal or abnormal behavior in a system, monitor the system in order to detect intrusions. They do so by comparing themselves against the current activities in that system. If the detectors represent normal behavior, then the activities that do not match the detectors are termed intrusions. If the detectors represent abnormal behavior, then the activities that match the detectors are termed intrusions. Generating a set of detectors that are efficient enough to detect all intrusions in a system, based on a limited definition of normal behavior, is a challenging task. In this research, the task of creating efficient detectors is modeled as a pattern matching problem and Clonal EAs are employed to solve this pattern matching problem.

A power grid is a critical infrastructure for a nation. Any disruption in the operation of a power grid is bound to have a significant impact on the lives of a large number of people. Faults in power transmission lines or a transformer in a grid can result in

unbalanced flow of electricity and may cause severe damage to the grid. FACTS devices are placed on power transmission lines to stabilize the flow of electricity across the grid, when the transmission lines or transformers malfunction. These FACTS devices are highly expensive and if placed on ideal locations, can help maintain the grid in normal operating conditions, even when abnormal conditions are encountered. It is a challenging task to find optimal placement positions for FACTS devices in a grid, as there are a large number of such positions. Even if we consider a grid with only a 100 power lines, given five FACTS devices, we would have 75, 287, 520 possible combinations for the placement of FACTS devices on that grid. In a practical case, the number of power lines in a power grid could well be in the range of millions or more. Hence, a large search space needs to be explored to determine optimal placements for a given number of FACTS devices. Clonal EAs are used in this research to solve this problem.

In Section 2, the clonal selection principle is explained and a general framework for designing a Clonal EA is provided. It is then compared with a traditional EA to illustrate the differences between the two algorithms. In Section 3, intrusion detection principles are explained and methods for defining detectors are explored. A Clonal EA design to generate detectors for a particular data set is proposed and the results are analyzed. Finally, a Clonal EA design is proposed for generation of rules in Snort, a rule based IDS. In Section 4, a brief introduction to power grids and FACTS devices is provided. Then, four algorithms including a Clonal EA are proposed for determining optimal FACTS device placements and the results obtained using the algorithms are compared, and analyzed. In Section 5, a summary of the performance of a Clonal EA in solving the intrusion detection and FACTS device placement problems is provided. Also, some important considerations that are to be made when a Clonal EA is applied for pattern matching or function optimization problems are discussed. Section 6 details possible future research work.

2. CLONAL SELECTION AND EVOLUTIONARY ALGORITHMS

The clonal selection principle is inspired by the human immune system [1]. The human body generates White Blood Cells (WBCs) and compares them against body cells during a phase termed Negative Selection. During this phase, the WBCs that match the body cells are destroyed and only those that do not match the body cells are released into the blood stream. These WBCs are called antibodies. When a foreign cell (antigen) enters the body, the WBCs are compared against "epitopes", which are small parts of the antigen. Those WBCs that closely match the epitopes are replicated to form copies (clones), thereby increasing their concentration in the blood stream during a process referred to as Clonal Expansion. The clones are then subjected to a Hyper-Mutation process during which their characteristics are altered proportional to the proximity of their match to the antigen. Alteration of a clone's characteristics may either result in a better match with the antigen or a poorer one. The clonal expansion and hyper-mutation operations result in the generation of mature clones and the whole process is referred to as Affinity Maturation. These mature clones have diverse characteristics though they may not match the antigen better than the antibodies they were created from.

The mature clones of some antibodies that exhibit a high affinity towards the antigen are retained as memory antibodies. The memory antibodies serve to efficiently detect the antigen and destroy it, if it is encountered again. A certain number of newly generated mature clones are arbitrarily selected to replace antibodies that have a poor match with the antigen. This facilitates improving the efficiency of the antibodies in matching the antigens, while maintaining diversity in the population. This process is termed Repertoire Diversity and it helps in maintaining reserves of antibodies that can efficiently match variants of antigens previously encountered. Thus, the antibodies are improved as well as fine-tuned to efficiently detect and destroy previously encountered as well as novel antigens [2]. This is a dynamic process which is abstracted into the clonal selection principle for solving engineering problems.

Evolutionary Algorithms (EAs) are inspired by the process of biological evolution [3]. EAs operate on a population of potential solutions to a problem. These potential solutions are refined to produce better approximations or offspring to a solution using the *survival of the fittest* principle, borrowed from the process of evolution. This process involves selecting a set of approximations termed parents, based on some problem-specific criteria, and breeding them together using genetic operators to form the offspring. Thus, this process leads to the generation of a population of individuals that have, on average,

superior fitness levels as compared to the original approximations. This process is repeated over several generations in order to refine the approximations. The genetic operators that EAs borrow from the process of biological evolution are:

1. Selection: Selecting a set of individuals from a population for reproduction based on some criteria.
2. Reproduction: The selected members reproduce by transferring part of their genetic content to the offspring. This involves two operations, namely:
   - Cross-Over: The process of creating new genetic content based on the parents. In a typical Cross-Over between two parents, one part of the genetic content of the offspring comes from one parent and the remaining part comes from the other parent to form new genetic content. The selection of the parts of genetic content received from the parents is based on some stochastic process.
   - Mutation: This is a process involving subtle alterations in the genetic content of an offspring
3. Evaluation: In the case of static environments, the newly created offspring are evaluated to determine how fit they are to survive in their environment. In the case of dynamic environments, the entire population along with the newly created offspring is evaluated.
4. Competition: The offspring and the parent population compete among themselves to enter the population pool for the next generation.

The following notations are used in the description of an EA:

1. M_p represents a population set.
2. M_o represents an offspring set.
3. M_s represents a parent set which is a subset of the population set.
4. n_p represents the cardinality of M_p.
5. n_o represents the cardinality of M_o.
6. n_s represents the cardinality of M_s. Usually $n_s = 2 \cdot n_o$, if the cross-over mechanism uses pairs of members to create offspring. If $n_p \leq 2 \cdot n_o$, then M_s can be a multiset.

7. $F(.)$ represents the fitness function which evaluates the fitness of a member in M_p or M_o. The objective is to maximize the fitness value of a member which corrsponds to the optimization of some problem dependent objective function.

One type of EA using the genetic operators previously described can be designed as:

1. Initialize M_p with uniform randomly created members.
2. Evaluate the fitness of all the members of M_p using $F(.)$.
3. Select n_s members of the population pool based on a selection criterion that is influenced by the fitness of the members and add them to M_s. If $n_p \leq n_s$, then it is possible that a member M_p^x of M_p gets selected into M_s more than once.
4. Create n_o offspring by selecting pairs of members from M_s, using a specific selection criterion and subjecting each of those pairs to cross-over and mutation wherein the offspring obtain unique properties based off the properties they acquired from the parents during Cross-Over.
5. Evaluate the offspring to determine their fitness based on $F(.)$.
6. Set up a competition between the members of M_p and M_o and insert the n_p highest fitness members among them into the next generation population set, which happens to be M_p again. (*survival of the fittest*).
7. Repeat steps 2 to 6 until a specified number of generations is reached, or the members obtain a specified fitness level.

In a typical search algorithm, the properties of EAs serve to broadly direct the search process and the properties of the clonal selection principle can be used to fine tune the search mechanism. This is emphasized by the fact that clonal selection multiplies the population members proportional to their fitness values and examines the high-fitness members more closely using the *Hyper-Mutation* process. As a result, a large number of members with high fitness values are subjected to subtle variations, facilitating small steps in a number of directions, and a small group of members with low-fitness values are subjected to large variations thereby enhancing the diversity of the solution variants. Thus, EAs can be used as a framework wherein the clonal selection principle is employed to improve the efficiency of the search mechanism [1]. Such algorithms are termed Clonal EAs in this research.

Clonal EAs refine a set of solutions, over several generations, to determine an optimal set of approximations for a certain number of problem scenarios. The solutions are modeled as antibodies and a collection of such antibodies is termed a solution set. The problem scenarios are represented as antigens and a collection of problem scenarios is presented in an antigen set for which solutions need to be determined. Clonal EAs use two sets of solutions, namely a normal set and a memory set, which together constitute a population pool set. The normal set contains a set of solutions that are improved over several generations. The memory set contains a set of solutions that represent the best approximations found, until a particular generation, for each antigen in the antigen set. The memory set is only altered if an approximation, for a particular antigen, obtained in a particular generation, out-performs an approximation, for the same antigen, in the memory set. Generally, the cardinality of the memory set is equal to the cardinality of the antigen set.

The following notations are used in the description of a Clonal EA:

1. M_n represents a normal solution set.
2. M_m represents a memory solution set.
3. M_p represents a population set which is the union of the normal and memory sets ($M_p = M_n \bigcup M_m$).
4. M_c represents a clone set.
5. M_{ag} represents an antigen set comprising the problem scenarios.
6. n_n represents the cardinality of M_n.
7. n_m represents the cardinality of M_m.
8. n_p represents the cardinality of M_p.
9. n_c represents the cardinality of M_c.
10. n_c^i represents the number of clones of a member i in M_p, that are to be generated in M_c.
11. n_{ag} represents the cardinality of M_{ag}.
12. $F(.)$ represents the fitness function which evaluates the fitness of a member in M_p or M_c against an antigen in M_{ag}. F(.) uses an evaluation criterion, *Eval* to compare the members against an antigen.

13. M_{ag}^{x} represents a member x in the antigen set.

14. M_{c}^{x} represents a member x in the clone set.

15. f_{c}^{x} represents the fitness value of member x in M_c.

One type of Clonal EA for solving static problems can be described as follows:

1. Initialize M_n with uniform randomly created members. (The members of this set are analogous to the antibodies in the immune system).

2. Initialize M_m with uniform randomly created members. (The members of this set are analogous to the memory antibodies in the immune system).

3. Populate M_{ag} with the problem scenarios for which solutions are to be devised.

4. Repeat steps 4 to 11 for each antigen M_{ag}^{i}, where $i = 1$ to n_{ag}, in M_{ag}.

5. Populate M_p with members from M_n and M_m using the set union operation.

6. Compute the fitness of all the n_p members of M_p with respect to M_{ag}^{i} using $F(.)$. Assign a fitness value to each member in the population pool based on *Eval*.

7. Select the n highest fitness members from M_p, and create multiple clones of each selected member (Clonal Expansion), with the number of clones generated for each member being proportional to the fitness of the member. Insert the clones thus generated into M_c. For example, if the members of M_p are sorted in descending order of fitness such that the position of each member represents its fitness-based rank, then the number of clones to be generated for each one of the n_p members is given by the formula:

$$n_c^i = \left\lceil \frac{\gamma \cdot n_p}{i} \right\rceil \qquad (2.1)$$

where γ is a multiplicative constant that determines the proportion of number of clones of each member, to be generated with respect to the total number of members, n_p, and i represents the fitness-based rank of each one of the n_p members and takes values in the range $[1, n_p]$. The total number of clones to be generated is then computed as:

$$n_c = \sum_{i=1}^{n_p} n_c^i \tag{2.2}$$

If $\gamma = 0.5$, and $n_p = 100$, then for $i = 1$, the number clones to be generated is 50 and for $i = 5$, the number of clones to be generated is 10. Thus, the higher the fitness of a member, the larger the number of clones to be generated for that member.

8. Subject all the members of M_c to hyper-mutation, wherein the high fitness clones undergo minute variations in their properties, and the low fitness clones undergo large variations. (These clones are akin to offspring discussed in the description of EAs). For example, a *mutate* function can be defined such that it takes as input arguments the population member to be mutated, M_c^x, and an upper bound on the maximum amount by which the member's properties can be mutated, *ceil*. Assuming that the fitness value of each clone is in the range $[0, 1]$, the upper bound can be calculated as:

$$ceil = \omega \cdot (1 - f_c^x) \tag{2.3}$$

where ω is a multiplicative constant. Then, the hyper-mutation operation can be performed such that the mutated member replaces the original member:

$$M_c^x = mutate(M_c^x, ceil) \tag{2.4}$$

If $\omega = 10$, then for $f_p^x = 0.7$, $ceil = 3$ and for $f_p^x = 0.2$, $ceil = 8$. Thus, the lower the fitness, the higher the range for the properties of a member to be mutated.

9. Compute the fitness of all the clones in M_c with respect to M_{ag}^i using $F(.)$.

10. Select the highest fitness member M_c^h from the set of newly created clones and compare it with a member M_m^x in M_m that corresponds to antigen M_{ag}^i. If the fitness of M_c^h is greater than that of M_m^x, then replace M_m^x with M_c^h.

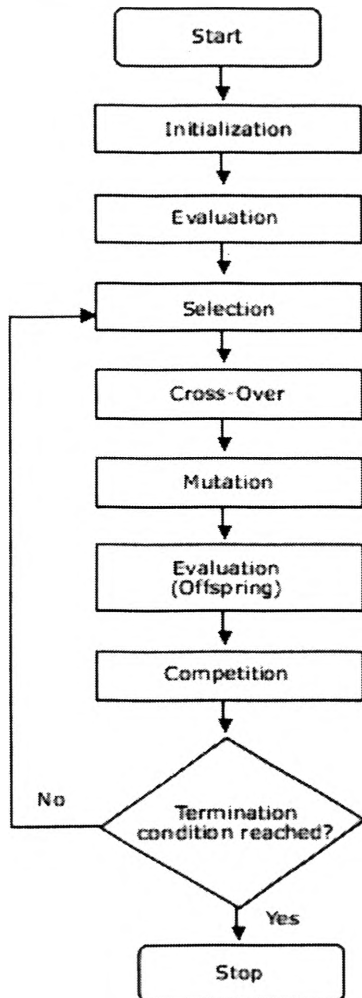
11. Select the m lowest fitness members from M_n and replace them with the m lowest fitness members from M_c (Repertoire diversity).

12. Repeat steps 3 to 11 until the specified number of generations is reached, or the members obtain a specified fitness level.

A comparison of the features of EAs and Clonal EAs is provided in Table 2.1. Figure 2.1 shows an EA and a Clonal EA side by side, illustrating the operations involved in each of them and the differences between the two algorithms. The shaded portions in the Clonal EA flowchart represent the additional steps that are absent in a traditional EA. The process flow depicted in the flowchart is based on the description of EAs and Clonal EAs provided previously.

Table 2.1. Differences between Clonal EA and Traditional EA

Clonal EA	*Traditional EA*
Maintains a memory solution set along with the normal solution set	Maintains a single solution set
Uses clonal expansion for reproduction	Uses cross-over for reproduction
Mutation is inversely proportional to fitness	Mutation is typically the same for all members irrespective of their fitness
Uses repertoire diversity for maintaining diversity in the population	There is no specific method for maintaining diversity in the population



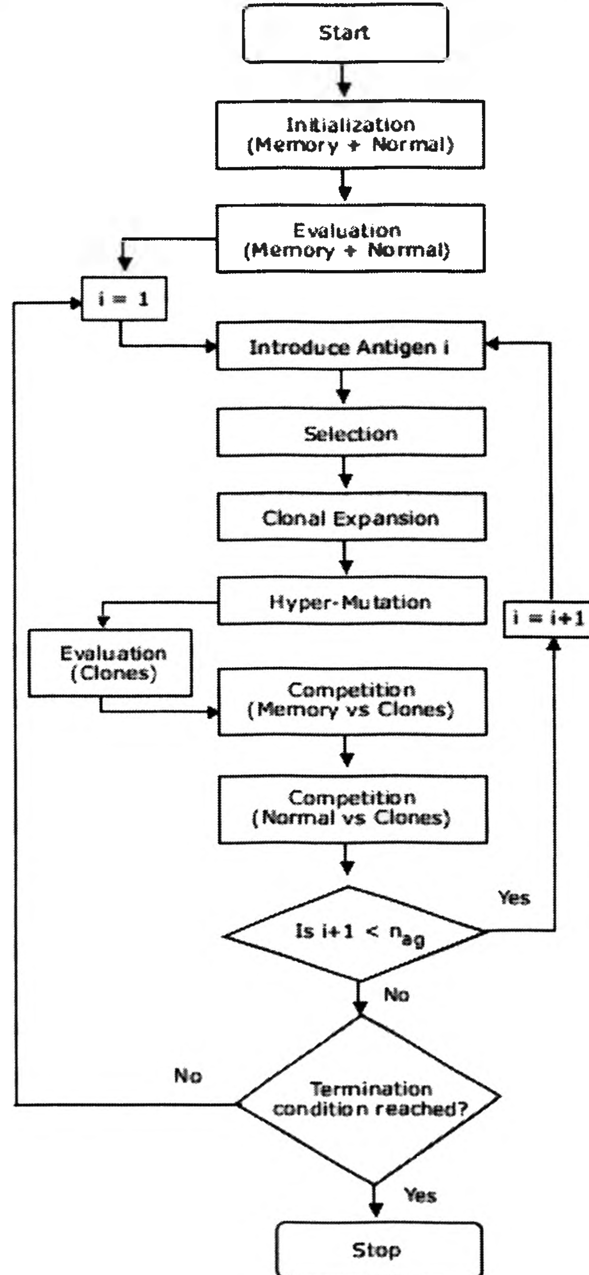


Figure 2.1. Comparative Features of a Traditional EA and a Clonal EA

3. INTRUSION DETECTION

Intrusion Detection Systems (IDS) provide a layer of security below the identification and authentication systems. They can be classified into two categories, namely Positive Detection (PD) and Negative Detection (ND) schemes, based on the detection mechanism employed. The PD scheme creates detectors, termed positive detectors, that represent normal activities or behavior in a system and uses them to evaluate the current activities in that system. If any of the current activities do not match the detectors, then they are termed intrusions (they may also be normal activities classified incorrectly). Checksum methods, many neural network based methods, and pattern classifiers are examples of IDS based on the PD scheme [4]. IDS employing the ND scheme derive their inspiration from the immune system that protects the human body against pathogens (cells not part of the body), and hence are termed Artificial Immune Systems (AIS) [5, 6]. The ND scheme creates detectors, termed negative detectors, which represent intrusive activities in a system. If any of the current activities match the detectors, then they are termed intrusions.

In order to detect intrusions in a system, it is important to have a clear definition of what types of activities would be considered as intrusions. The activities in a system are classified in this research as:

1. Normal: represents a known set of routine activities executed by normal users in a system.

2. Misuse: represents a known set of all the undesirable activities in a system.

    - Type I: Activities that are normal, but affect the performance and efficiency of a system such as the denial of service attack.

    - Type II: Activities that are abnormal by definition of the behavior of a system. These can be further classified into intended malicious and unintentional abnormal activities.

3. Anomaly: represents the types of activities in a system that are not classified as Normal or Misuse and they comprise unknown normal activities and unknown misuses.

Misuses and anomalies are usually considered as intrusions to be detected, though there might be some anomalies which are not intrusions. Activities that are considered

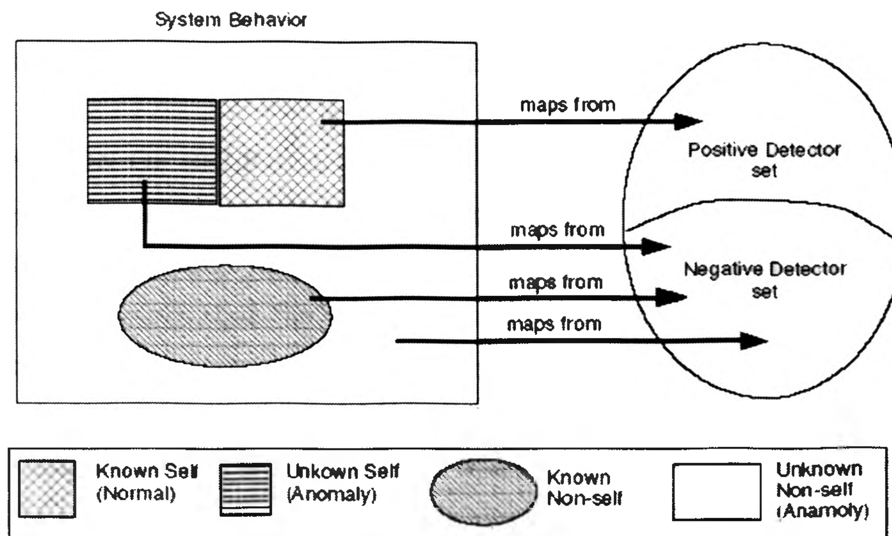


Figure 3.1. A Set Theoretical Representation of the System Definition

normal in a system are termed as self and activities that are considered misuses are termed non-self. A self set is comprised of activities that have been previously identified as normal activities (known self) and anomalies that are yet to be identified as such (unknown self). A non-self set is comprised of misuses (known non-self) and anomalies that are yet to be identified as misuses (unknown non-self). Figure 3.1 gives a set theoretical representation of system behavior or activities. The detectors for the PD scheme represent the known self and detectors for the ND scheme represent known non-self, unknown non-self and unknown self.

IDS, using the PD or ND scheme, can be classified into two broad categories based on the types of systems they protect:

1. Host-based IDS: These IDS reside in a single computer, protecting it against attacks and anomalies.

2. Network IDS: These IDS are distributed across various critical nodes of a system and they protect the network resources against attacks and anomalies.

In this section, the PD and ND schemes are explained along with the experiments performed to determine the circumstances under which they perform well. The PD and

ND schemes are then employed in designing a Clonal EA termed IDSClonalg that detects intrusions in a static data set obtained from KDD Cup 1999[1] data.

3.1. DEFINITION OF SELF

Artificial Immune Systems (AIS) depend upon a comprehensive definition of activities that are considered safe and normal for a particular system. A comprehensive definition of self ensures that the detection mechanism can effectively distinguish abnormal activities from the normal activities. The more comprehensive the definition of self, the smaller the number of unknown self activities. Therefore, given a sufficiently comprehensive definition of self, it is reasonable to consider any activity that does not belong to the self set as an intrusion without incurring significant false alarms. However, a comprehensive definition of self would result in a lot of time being spent on creating the detectors. This is due to the fact that there will be a large number of self activities for the detectors to be compared with and evaluated. Also, a considerable amount of time may need to be spent on perfecting the definition of self. Thus, there is a trade-off between the accuracy of the detection system and the time taken to generate the detectors.

The self set can be constructed as a statistical estimate of the activities of normal users. For this purpose, all the processes executed and resources utilized by the users must be tracked and logged. The patterns associated with activities the users perform must also be monitored to define the normal state of a system. Consider a system which is monitored to collect statistical data for defining the self set of that system. The monitoring process would record the activities and processes executed by each user. Each user's profile is represented as s_i, and their corresponding activities , a_x^i, occurring in a temporal sequence during a time span, t, are to be recorded. The representation would then be a vector of vectors:

$$S = \{s_1, s_2, s_3, \ldots, s_n\}, \tag{3.1}$$

where

$$s_i = \{a_1^i a_2^i \ldots a_t^i\}. \tag{3.2}$$

A large collection of such vectors must be constructed out of activities that are considered normal, with respect to a particular system, to obtain a reasonably comprehensive

[1]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

definition of self. A set of these vectors, therefore, represents an approximation of the self set.

The representation scheme also has an impact on the process of detection. A simple representation scheme, such as one using a string representation, will facilitate easy and swift comparison as well as compact storage. In this research, the string representation scheme is used to represent the self and non-self set members. In this scheme, the activities are represented by a single or a collection of symbols defined for a particular alphabet. The mapping from the observable universe, formed by the activities that can be performed in a system, to the alphabet, depends on the number of activities defined. For example, considering a binary alphabet for the representation scheme, it would contain symbols $\{0,1\}$. If there are only two activities, namely A and B, which can be performed in a system, then they can be mapped to labels formed by the symbols as shown in Table 3.1.

Table 3.1. Mapping of Labels to Activities

Activity	Label
A	0
B	1

If four activities, namely A, B, C and D have to be mapped onto the representation scheme, then they can be mapped to labels formed by a collection of symbols as shown in Table 3.2.

The strings, s_i, can then be represented as a sequence of labeled activities (containing a single symbol or a collection of symbols). Thus, a string s_i can be constructed for a

Table 3.2. Mapping of Labels to Activities

Activity	Label
A	00
B	01
C	10
D	11

sequence of activities A, B, D, C as:

$$s_i = \{00011110\} \tag{3.3}$$

A collection of such strings would form the self set, S.

3.2. NEGATIVE DETECTOR SET GENERATION

The ND scheme employs negative detectors representing the non-self set to monitor a system for misuses and anomalies. If any activity in the system matches with one or more negative detectors, then the activity is considered an intrusion. The negative detectors are created using the negative selection process. Algorithm 1 provides a general description of the negative selection process applied to intrusion detection. The size of the negative detector set can be set to an arbitrary value, N_r.

Algorithm 1 A General Negative Selection Algorithm

```
//INPUTS: N_r
//OUTPUTS: ND
Construct set S
//The method of constructing this set is implementation specific
Initialize the detector set ND ← φ
i ← 0
while i < N_r do
  Generate a string DetectorString
  //The method of generating the string is implementation specific
  if match(DetectorString, S) = true then
    //match is a function that is defined based on the matching
    //scheme used and it returns true or false
    Reject DetectorString
  else
    ND ← ND ∪ DetectorString
    i ← i + 1
  end if
end while
```

The detector set is a subset of the non-self set and optimally contains those vectors that can represent an effective approximation of the non-self set. Initially, all these vectors can be randomly generated and compared against a set of self vectors. The comparison can be performed using a matching process which checks for similarity in the individual attributes of the two entities being compared, termed a matching scheme, and this scheme

is in turn dependent on the representation scheme. A detailed discussion of the different types of matching schemes can be found in [7]. Those detector vectors, d_i, that do not match (depending upon the matching scheme) the self vectors can be retained, and those that do can be eliminated. The surviving detectors form the set, ND.

$$ND = \{d_1, d_2, d_3, \ldots, d_n\}, \tag{3.4}$$

where

$$d_i = \{a_1^i a_2^i \ldots a_t^i\}. \tag{3.5}$$

with $a_1^i a_2^i \cdots a_t^i$ representing a sequence of non-self activities. The current set of activities in a system can be represented by a set Q as:

$$Q = \{q_1, q_2, \ldots, q_t\}, \tag{3.6}$$

where

$$q_i = \{b_1^i b_2^i \ldots b_t^i\}. \tag{3.7}$$

with $b_1^i b_2^i \cdots b_t^i$ representing a sequence of activities observed in a system. The members of Q can then be compared against the members of the detector set, ND, using the matching scheme, and if a reasonable match occurs, it will indicate the occurrence of an abnormal activity. The IDS must determine the amount of deviation from normalcy that can be tolerated. Activities that deviate more than the specified amount should be termed intrusions.

A prototype system that uses binary strings to represent self and non-self activities is used in this research to study the ND scheme. The self set of the prototype system is constructed as a set of random binary strings with a specified length, l. Algorithm 2 describes the process of creating the self set. The prototype system uses an exact matching scheme that contains the operators "$\in$" and "$\notin$".

Various methods for construction of the detector sets based on the negative selection principle have been proposed in [8], [9] and [10]. The exponential algorithm proposed in

Algorithm 2 An algorithm for generating the self set of the prototype system

//INPUTS: N_s, l
//OUTPUTS: S
$i \leftarrow 0$
$S \leftarrow \phi$
//Loop for constructing the self set
while $i < N_s$ **do**
  $selfstring \leftarrow randomstring(l)$
  //$randomstring(l)$ generates a binary string of length l, uniform randomly
  if $selfstring \notin S$ **then**
    //Ensure that the same string is not included more than once
    $S \leftarrow S \bigcup selfstring$
    $i \leftarrow i + 1$
  end if
end while

[8] and the greedy algorithm proposed in [9], for generating negative detectors, are implemented for the prototype binary system in this research. Some of the common terminologies associated with the exponential and greedy negative detector generation algorithms are:

N_s	-	Number of members in the self set S
N_r	-	Number of detectors in the detector set ND
P_m	-	Probability that two random strings match
P_f	-	Probability that N_r detectors fail to detect an intrusion
m	-	Number of symbols in the alphabet used to define the strings
l	-	Number of symbols in a string
r	-	Number of contiguous matches required for a match

3.2.1. Exponential Algorithm. The exponential algorithm is an implementation of the negative selection algorithm (Algorithm 1). It constructs the self set using Algorithm 2. The prospective detector strings are generated randomly and tested against the self set. Those strings that do not match any of the self set members are added to the detector set ND and those strings that match the self set members are discarded. The exact matching scheme is used to implement the function $match(d, S)$ discussed in Algorithm 1. This process is repeated until the required number of detectors (N_r) is obtained. The value N_r is computed stochastically based on P_f and P_m as in [8]:

Algorithm 3 Exponential Negative Detector Generation Algorithm

//INPUTS: S, N_s, N_r, l
//OUTPUTS: ND
$i \leftarrow 0$
$ND \leftarrow \phi$
//Loop for generating random strings that might be included in the detector set
while $i < N_r$ **do**
  $DetectorString \leftarrow randomstring(l)$
  if $DetectorString \notin S$ **then**
    $ND \leftarrow ND \bigcup DetectorString$
    $i \leftarrow i + 1$
  end if
end while

$$N_r = \frac{-lnP_f}{P_m} \tag{3.8}$$

A detailed explanation of the computations involved in stochastically determining the value of P_f and P_m can be found in [8]. The main objective of computing N_r using Equation 3.8 is to ensure that a sufficient number of detectors are available for detecting intrusions. A pseudocode for the exponential algorithm is described in Algorithm 3.

The exponential algorithm is a primitive method for the generation of detectors using the negative selection method. Since it involves generating strings randomly, it may take a long time for the generation of valid detectors and in some cases it may result in an unreasonably large number of iterations. For example, if $m = 2$ (binary) and $l = 10$, then there are 1024 (2^{10}) strings in the universe. For $N_s = 24$ and $N_r = 1000$, assuming that 999 detectors have already been generated by Algorithm 3, the probability that the final detector string would be generated by the function *randomstring*(l) is 0.00098 (1/1024). The detector generation process is expected to be much faster and more accurate, when the greedy algorithm, proposed in [9], is used.

3.2.2. Greedy Algorithm. The greedy algorithm is another negative selection algorithm implemented for the prototype system. The self set for this algorithm is also constructed using Algorithm 2. The greedy algorithm efficiently computes all possible combinations of bit positions that would result in non-self strings (based on the existing definition of self, created as a set of random binary strings) and picks only those strings that would cover a maximum number of non-self strings *not added* to the detector set,

a distinctive weakness of the exponential algorithm. The algorithm thus avoids the exhaustive generation of random strings and ensures a fair distribution of the representative detectors from the non-self set, thus providing better coverage. Some of the notations used for developing a pseudocode for the greedy algorithm proposed in [9] are:

1. S is a set of binary strings of length l.

2. $S[i \ldots j]$ denotes a subset of strings in S $\{s|s$ is the restriction of positions $i \ldots j$ of some s' in $S\}$.

3. s denotes a bit string. The bits in s are numbered left to right.

4. $\hat{s}$ denotes s stripped of its leftmost (most significant) bit.

5. $s \cdot b$, where $b \in \{0, 1\}$, denotes s appended with b. $\hat{s} \cdot b$ represents s stripped of its first bit and appended with b at its right end. $b \cdot \hat{s}$ represents s stripped of its last (rightmost) bit and appended with b at its left end.

6. Two strings *match* each other if they have identical bits in at least r contiguous positions.

7. A *template* of order r is a string of length l consisting of $l - r$ "blank" symbols (represented here by an asterisk) and r fully specified contiguous bits. In particular, a template, $t_{i,s}$, in which s is an r-bit string, is that template in which the r contiguous bits start at bit position i and are given by s. For example, if $l = 64$, $r = 3$ and $s = 010$, then $t_{2,s} = \star 010 \star \star$.

8. A template *matches* a string if they have identical bits (no blanks) in at least r contiguous positions.

9. A right or left *completion* of a template, t, is that template with all the blanks to its right or left replaced by bits. For example, $\star 01011$ is a right completion for $\star 010 \star \star$.

A set of pseudocodes is devised in this research based on a series of algorithms proposed in [9], for implementing a greedy algorithm. Generally, in a string of length l, a template of length r can be positioned in $l - r + 1$ ways, hence the data structures employed by the algorithms have to account for 2^r templates and $l - r + 1$ possibilities. Some of the data structures used in the algorithms are:

1. $C_1[1 \ldots 2^r][1 \ldots (l - r + 1)]$ represents the number of right completions $t'_{i,s}$ of $t_{i,s}$, such that $t'_{i,s}$ is unmatched by any string in S.

2. $C'_1[1 \ldots 2^r][1 \ldots (l-r+1)]$ represents the number of left completions $t'_{i,s}$ of $t_{i,s}$, such that $t'_{i,s}$ is unmatched by any string in S.

3. $D_s[1 \ldots 2^r][1 \ldots (l-r+1)]$ represents the product of the C_1 and C'_1 array values.

4. $D_r[1 \ldots 2^r][1 \ldots (l-r+1)]$ represents the current state of the detector set, D.

5. $C_2[1 \ldots 2^r][1 \ldots (l-r+1)]$ and $C'_2[1 \ldots 2^r][1 \ldots (l-r+1)]$ are arrays that keep track of the number of non-self strings covered by a particular template.

6. $Match[1 \ldots 2^r][1 \ldots (l-r+1)]$ represents the number of possible and valid non-self strings that can be generated.

7. $T[1 \ldots 2^r]$ stores all the combinations of templates.

The greedy algorithm (Algorithm 4) accepts a complete template list, T, as input along with other parameters. The self set, S, is then constructed by introducing random strings of a particular length, l, into the set, allowing redundancies that result in a multi-set. The algorithm initializes the above mentioned data structures using the InitStructures algorithm (Algorithm 5). Once the initialization of data structures is complete, it is possible to determine the number of valid non-self strings that can be generated and this information is provided by the data structure, D_s.

The next step is to generate a new non-self string. This is performed by selecting a template that has the maximum value in the D_r array. The D_r array maintains for each template of size r, the number of non-self strings that are yet to be matched by the detectors already generated and populated in ND. So, if a template is chosen for the construction of a detector, then the number of non-self strings matched by that detector is deducted from D_r for that template. Therefore, D_r is constantly updated as and when a detector is created. Specific computations involved in updating D_r can be found in Algorithm 4, Algorithm 8 and Algorithm 9. A detailed explanation of the computations involved in updating D_r can be found in [11]. The template that matches the largest number of non-self strings that are yet to be matched by the detectors in ND, is thus selected.

The selected template is inserted into a particular position in the non-self string and the remaining bit positions are filled using the RightFill (Algorithm 8) and LeftFill (Algorithm 9) algorithms. If there are no valid bits that can fill a bit position in the new string, then the $flag$ variable is set to "0", indicating that no new strings can be formed based on the selected template and its position. However, if all the l bits in the new string are filled, then it would represent a valid string and hence can be added to the detector

Algorithm 4 Greedy Negative Detector Generation Algorithm

//INPUTS: S, N_s, N_r, T, l, r
//OUTPUTS: ND
$ND \leftarrow \phi$
Call InitStructures($S, N_s, C_1, C_1', C_2, C_2', D_r, Match$)
//Initialize all the data structures involved
//Create a candidate detector
$flag \leftarrow 1$
while $flag = 1$ **do**
  //Select a template that covers the maximum number of
  //non-self strings for a particular bit position
  $[maxi, maxj] \leftarrow Maxindex(D_r)$
  //Maxindex takes an array as input, determines the maximum value
  //in the array and returns its index
  $DetectorString \leftarrow NULL$
  //Insert the selected template into the new string
  $DetectorString(maxj) \leftarrow T[maxi]$
  RightFill($DetectorString, S, C_2, C_2', D_r, Match, N_s, l, r, maxi, maxj, flag$)
  //RightFill fills *DetectorString* with binary bits on the right side of the
  //inserted template, sets the $flag$ to "0" if valid strings cannot be
  //generated and updates the book-keeping arrays (C_2, C_2', D_r)
  if $flag \neq 0$ **then**
    LeftFill($DetectorString, S, C_2, C_2', D_r, Match, N_s, l, r, maxi, maxj, flag$)
    //LeftFill fills *DetectorString* with binary bits on the left side of the
    //inserted template, sets the $flag$ to "0" if valid strings cannot be
    //generated and updates the book-keeping arrays (C_2, C_2', D_r)
  end if
  if $flag \neq 0$ **then**
    $ND \leftarrow ND \bigcup DetectorString$
  end if
  $flag \leftarrow 1$
  call ColumnCheck($Match, l, r, flag$)
  //ColumnCheck determines the termination point by checking if a particular
  //bit position is exhausted for all the templates in which case no further
  //valid strings can be produced and the $flag$ is set to "0"
  if $flag \neq 0$ **then**
    UpdateArray(C_2, C_2', l, r)
    //UpdateArray updates the book-keeping arrays to reflect the non-self
    //string coverage status of the templates of size r
    $D_r \leftarrow C_2 \cdot C_2'$
  end if
end while

set ND. Once this is done, the book-keeping arrays, namely C_2, C_2', $Match$ and D_r, which keep track of the number of templates that have already been used to construct non-self strings, have to be updated to indicate the templates that are exhausted due to

Algorithm 5 InitStructures: Algorithm for initializing the data structures

//INPUTS: S, N_s, C_1, C_1', C_2, C_2', D_s, D_r, $Match$
//Initialize C_1, C_1', D_s arrays
Call InitC1(S, C_1, N_s, l, r)
Call InitC1D(S, C_1', N_s, l, r)
$D_s \leftarrow C_1 \cdot C_1'$
//Initialization of C_2, C_2', D_r and $Match$ arrays
$i \leftarrow 1$
while $i \leq 2^r$ **do**
  $j \leftarrow 1$
  while $j \leq l - r + 1$ **do**
    //$l - r + 1$ represents the number of different ways a template of length r
    //can be positioned within a string of length l
    $C_2[i][j] \leftarrow 2^{l-r-j}$
    $C_2'[i][j] \leftarrow 2^j$
    $D_r[i][j] \leftarrow 2^{l-r}$
    if $D_s[i][j] \neq 0$ **then**
      //$Match$ positions are initialized to "1" only if the D_s array has
      //non-zero entries in those positions
      $Match[i][j] \leftarrow 1$
    end if
  end while
end while

addition of the new string. But, if the $Match$ array has a zero entry for a particular bit position for all the templates, then valid non-self strings can no longer be generated and hence the process must stop. This is remedied by reinitializing the $flag$ and updating its value using the ColumnCheck algorithm (Algorithm 10). If the $flag$ is set to "0" by the ColumnCheck algorithm, then it is not necessary to update the book-keeping arrays. The greedy algorithm repeats this process until all the valid detectors are generated (i.e., until the $flag$ is reset by the ColumnCheck algorithm).

The C_1 array is constructed by comparing the strings in the self set from right to left with all the possible templates of length r. If from all the self strings none of the substrings extracted from a particular bit position match a particular template, then the score for that particular template at that bit position is updated in C_1. Otherwise, the score is set to "0" for the self string s that matched the template. If the template is inserted at position $l - r + 1$, then there are no right completions possible. This case has to be addressed separately, hence two formulas (Equation 3.9 and Equation 3.10) for updating the score have been suggested in [9]. If right completions are not possible:

Table 3.3. Exhaustive List of Templates for $r = 3$

Label	Template
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

$$C_{l-r+1}[s] = \begin{cases} 0 & \text{if } t_{l-r+1,s} \text{ is matched in } S \\ 1 & \text{otherwise} \end{cases} \tag{3.9}$$

and if right completions are possible:

$$C_{l-r+1}[s] = \begin{cases} 0 & \text{if } t_{l-r+1,s} \text{ is matched in } S \\ C_{i+1}[\hat{s}.0] + C_{i+1}[\hat{s}.1] & \text{otherwise} \end{cases} \tag{3.10}$$

The implementation details related to these two formulas are illustrated in the InitC1 algorithm (Algorithm 6). The $\hat{s} \cdot b$ operation is implemented in the algorithm using a slightly different method. Consider templates of length $r = 3$, then the exhaustive list of templates can be numbered as in Table 3.3.

Consider $s = 001$ and $b = 0$. $\hat{s}.b$ results in 010. Since 010 can be indexed by "1", and 011 can be indexed by "2", the formula, $(2 \cdot j)mod(2^r)$, where j is the index for $\hat{s}$, is used to determine the indices. In this example, $j = 1$ and $r = 3$, which would result in the new index $(2 \cdot 1)mod(2^3) = 2$. Thus, the formula results in the correct index. If $b = 1$, then the index is incremented by 1 and the resulting template is extracted.

The array C'_1 is constructed in almost the same way as the array C_1, except that it examines the self strings from left to right, and the reverse procedure is illustrated by the InitC1D algorithm (Algorithm 7).

The RightFill algorithm (Algorithm 8) is responsible for filling a new string, pre-inserted with a template, with bits towards the right side. While doing so, it must ensure that the bits appended in positions to the right side of the template are the ones that would cover the maximum number of non-self strings. For example, a particular bit position, b, of the string is filled with "1", if "0" has already been used at b in another

string using the same template or if "0" covers a lesser number of non-self strings than "1". If "0" has already been used, then the *Match* cell would contain a value "0" for bit "0" at position b. If "0" covers a lesser number of non-self strings than "1", then it is indicated in the score for "0" in D_r. Once a particular bit (0 or 1) is inserted at a particular position in the string, the bits appended until that point form a new template. The book-keeping arrays, C_1 and C_1', are re-evaluated, wherein the bit position for the inserted template is set to 0 and the other cells are updated using the methods illustrated in Equation 3.9 and Equation 3.10.

The LeftFill algorithm (Algorithm 9) fills bits to the left side of the inserted template and reverses the method followed in the RightFill algorithm. The ColumnCheck algorithm (Algorithm 10) ensures that the greedy algorithm stops at a particular point. This happens when a particular bit position for all the templates is exhausted, meaning that no more strings can be constructed with values in that particular position. It takes the $flag$ as input and resets it if the termination condition is reached.

The C_2 and C_2' arrays are primarily responsible for tracking the number of valid non-self strings that are not yet inserted into the detector set ND. These arrays need to be updated after each newly constructed string that is added to the detector set ND. Algorithm 11 implements the procedure to perform the update.

3.3. POSITIVE DETECTOR SET GENERATION

The PD scheme employs detectors representing the self set to detect intrusions in the set of activities being observed in a system. The detector set PD thus contains the self set members that are obtained from an exhaustive definition of the self set. This set can be constructed by randomly picking members from the self set or by selecting those members from the self set that cover a maximum number of self set members. The second

Algorithm 6 InitC1: Algorithm for initializing the C_1 array

```
//INPUTS: S, C1, Ns, l, r
i ← l − r + 1
//Loop for checking the strings in self set, from right to left, with the templates
//in the corresponding positions
while i ≥ 1 do
  j ← 1
  while j ≤ 2^r do
    //Loop through all the templates
    //2^r represents the number of possible templates of length r
    k ← 1
    flag ← 1
    while k ≤ Ns do
      //Loop through all the strings in S
      if S[k][i...i + r] = T[j] then
        //If the template matches any self substring at position i reset the flag
        flag ← 0
      end if
      k ← k + 1
    end while
    //If the template is not matched by any of the self strings at position i,
    //then update the C1 array
    if flag = 1 then
      if i = l − r + 1 then
        //First update
        C1[j][i] ← 1
      else
        //Successive updates
        C1[j][i] ← C1[(2 · j)mod(2^r)][i + 1] + C1[(2 · j)mod(2^r) + 1][i + 1]
      end if
    end if
    j ← j + 1
  end while
  i ← i − 1
end while
```

Algorithm 7 InitC1D: Algorithm for initializing the C_1' array

```
//INPUTS: S, C'_1, N_s, l, r
i ← 1
//Loop for checking self strings, left to right, with the templates
while i ≤ l − r + 1 do
  j ← 1
  while j ≤ 2^r do
    k ← 1
    flag ← 1
    while k ≤ N_s do
      //Check if the template matches any self substring at position i
      if S[k][i...i + r] = T[j] then
        flag ← 0
      end if
      k ← k + 1
    end while
    //If template is unmatched by any self string at i, update the C'_1
    if flag = 1 then
      if i = 1 then
        //First update
        C'_1[j][i] ← 1
      else
        n ← r
        zero ← one ← 0
        x ← 0
        //Determine the templates formed when the rightmost bit in the
        //self substring is removed and a new bit is added to the left
        while n > 1 do
          if T[j][n − 1] > 0 then
            zero ← zero + 2^x
          end if
          x ← x + 1
          n ← n − 1
        end while
        one ← one + 2^x
        //Successive updates
        C'_1[j][i] ← C'_1[zero][i − 1] + C'_1[one][i − 1]
      end if
    end if
    j ← j + 1
  end while
  i ← i + 1
end while
```

method will provide a better coverage, while keeping the detector set size smaller. The detector set generated in this case would be represented as:

Algorithm 8 RightFill: Algorithm for filling up the new string on the right side

```
//INPUTS: S, C2, C2'', Dr, Match, Ns, l, r, DetectorString, maxi, maxj, flag
//Initialize the right pointer for the new string to the appropriate position
//after the last bit of the copied template
right ← maxj + r
Indi ← maxi
Indj ← maxj + 1
while right ≤ l and flag = 1 do
  //Determine the templates formed when the leftmost bit in the self substring is
  //removed and a new bit is added to the right
  zero ← (2 · Indi)mod(2^r)
  one ← (2 · Indi)mod(2^r) + 1
  if Match[zero][Indj] = 0 and Match[one][indj] = 0 then
    //Check to verify that the particular position for the template has not already
    //been assigned to any string
    flag ← 0
  else if Match[zero][Indj] = 0 then
    //Bit "0" bit has already been used
    DetectorString[right] ← 1
    Dr[one][Indj] ← Match[one][Indj] ← 0
    C2[one][Indj] ← C2''[one][Indj] ← 0
    Indi ← one
  else if Match[one][Indj] = 0 then
    //Bit "1" has already been used
    DetectorString[right] ← 0
    Dr[zero][Indj] ← Match[zero][Indj] ← 0
    C2[zero][Indj] ← C2''[zero][Indj] ← 0
    Indi ← zero
  else if Dr[zero][Indj] > Dr[one][Indj] then
    //Select a bit that covers the highest number of non-self strings
    DetectorString[right] ← 0
    Dr[zero][Indj] ← Match[zero][Indj] ← 0
    C2[zero][Indj] ← C2''[zero][Indj] ← 0
    Indi ← zero
  else
    DetectorString[right] ← 1
    Dr[one][Indj] ← Match[one][Indj] ← 0
    C2[one][Indj] ← C2''[one][Indj] ← 0
    Indi ← one
  end if
  right ← right + 1
end while
```

$$PD = \{d_1, d_2, d_3, \ldots, d_n\}, \tag{3.11}$$

Algorithm 9 LeftFill: Algorithm for filling up the new string on the left side

//INPUTS: $S, C_2, C_2', D_r, Match, N_s, l, r, DetectorString, maxi, maxj, flag$
$left \leftarrow indj \leftarrow maxj - 1$
$Indi \leftarrow maxi$
//Loop for filling up the new string till the first bit
while $left \geq 1$ and $flag = 1$ **do**
  $n \leftarrow r$
  $zero \leftarrow one \leftarrow x \leftarrow 0$
  //In a self substring, add a left bit and remove the rightmost bit
  for $i \leftarrow n$ to 1 **do**
    if $T[maxj][i-1] > 0$ **then**
      $zero \leftarrow zero + 2^x$
    end if
    $x \leftarrow x + 1$
  end for
  $one \leftarrow one + 2^x$
  if $Match[zero][Indj] = 0$ and $Match[one][indj] = 0$ **then**
    //Verify that a particular position for the template has not been used
    $flag \leftarrow 0$
  else if $Match[zero][Indj] = 0$ **then**
    //Bit "0" bit has already been used
    $DetectorString[left] \leftarrow 1$
    $D_r[one][Indj] \leftarrow Match[one][Indj] \leftarrow 0$
    $C_2[one][Indj] \leftarrow C_2'[one][Indj] \leftarrow 0$
    $Indi \leftarrow one$
  else if $Match[one][Indj] = 0$ **then**
    //Bit "1" has already been used
    $DetectorString[left] \leftarrow 0$
    $D_r[zero][Indj] \leftarrow Match[zero][Indj] \leftarrow 0$
    $C_2[zero][Indj] \leftarrow C_2'[zero][Indj] \leftarrow 0$
    $Indi \leftarrow zero$
  else if $D_r[zero][Indj] > D_r[one][Indj]$ **then**
    //Select a bit that covers the highest number of non-self strings
    $DetectorString[left] \leftarrow 0$
    $D_r[zero][Indj] \leftarrow Match[zero][Indj] \leftarrow 0$
    $C_2[zero][Indj] \leftarrow C_2'[zero][Indj] \leftarrow 0$
    $Indi \leftarrow zero$
  else
    $DetectorString[left] \leftarrow 1$
    $D_r[one][Indj] \leftarrow Match[one][Indj] \leftarrow 0$
    $C_2[one][Indj] \leftarrow C_2'[one][Indj] \leftarrow 0$
    $Indi \leftarrow one$
  end if
  $left \leftarrow left - 1$
end while

Algorithm 10 ColumnCheck: Algorithm for checking if a position in the string has been invalidated

```
//INPUTS: Match, l, r, flag
i ← 1
while i ≤ l − r + 1 do
  j ← 1
  //Determine if a particular position in the string (for each template)
  //has been exhausted
  columncount ← 0
  while j ≤ 2^r do
    columncount ← columncount + Match[j][i]
    j ← j + 1
  end while
  if columncount = 0 then
    flag ← 0
  end if
  i ← i + 1
end while
```

where

$$d_i = \{a_1^i a_2^i \dots a_t^i\}. \tag{3.12}$$

with $a_1^i a_2^i \dots a_t^i$ representing a sequence of activities. This detector set is then compared against the set of activities being observed and if the activities do not match any of the detector components, then those activities can be categorized as intrusions. Algorithm 12 and Algorithm 14 are exclusively designed in this research for the generation of positive detector sets in the prototype binary system. These algorithms also use the operators "$\in$" and "$\notin$" to implement the exact matching scheme used by the prototype system. The algorithms use the self set constructed using Algorithm 2.

Algorithm 12 constructs a detector set based on the description of a self set. Random strings are selected from the previously constructed self set and are validated against the detector set to eliminate redundancies. If the newly generated strings are indeed unique, then they are added to the detector set.

Algorithm 14 also constructs the detector set based on the description of the self set, but instead of selecting the self strings at random, it prioritizes the strings in the self set on the basis of the number of other self strings that the particular self string covers or matches. Thus, the detector set would consist of those self strings that cover most of their counterparts in the self set. This approach leads to a better representation of the self set while keeping the size of detector set small. In order to determine the coverage

Algorithm 11 UpdateArray: Algorithm for updating the arrays

//INPUTS: C_2, C_2', l, r
//Update the C_2 array
if $flag = 1$ **then**
  $i \leftarrow l - r$
  while $i \geq 1$ **do**
    $j \leftarrow 1$
    while $j \leq 2^r$ **do**
      $zero \leftarrow (2 \cdot j) mod(2^r)$
      $one \leftarrow (2 \cdot j) mod(2^r) + 1$
      if $C_2'[j][i] \neq 0$ **then**
        $C_2[j][i] \leftarrow C_2[zero][i+1] + C_2[zero][i+1]$
      end if
      $j \leftarrow j + 1$
    end while
    $i \leftarrow i - 1$
  end while
  //Update the C_2' array
  $i \leftarrow 2$
  while $i \leq l - r + 1$ **do**
    $j \leftarrow 1$
    while $j \leq 2^r$ **do**
      $n \leftarrow r$
      $zero \leftarrow 0$
      $one \leftarrow 0$
      $x \leftarrow 0$
      while $n > 1$ **do**
        if $T[j][n-1] > 0$ **then**
          $zero \leftarrow zero + 2^x$
        end if
        $x \leftarrow x + 1$
        $n \leftarrow n - 1$
      end while
      $one \leftarrow one + 2^x$
      if $C_2'[j][i] \neq 0$ **then**
        $C_2'[j][i] \leftarrow C_2'[zero][i-1] + C_2'[one][i-1]$
      end if
      $j \leftarrow j + 1$
    end while
    $i \leftarrow i + 1$
  end while
end if

of a self string, it is compared against the remaining strings in S using the function $MatchString(string1, string2, l)$ described in Algorithm 13.

Algorithm 12 Random Positive Selection Algorithm

```
//INPUTS: S, N_s, N_r
//OUTPUTS: PD
PD ← φ
i ← 0
while i < N_r do
  Pick a string from S randomly, say s
  if s ∈ PD then
    Reject s
  else
    PD ← PD ∪ s
    i ← i + 1
  end if
end while
```

Algorithm 13 MatchString: A function for comparing strings

```
//INPUTS: string1, string2, l
//OUTPUTS: result
count ← 0
for i = 1 to l do
  if string1[i] = string2[i] then
    count ← count + 1
  end if
end for
if count > (l/2) then
  result ← true
else
  result ← false
end if
```

The following sections explore some of the practical issues that are confronted when using the PD scheme for identifying intrusions, as compared to the ND scheme.

3.4. POSITIVE VS. NEGATIVE DETECTION

The literature reviewed has shown that the ND scheme has been explored extensively, but the effectiveness of the PD scheme has not. The objective of this research is to analyze the advantages of using the two schemes under different situations and to evaluate their performance in specific environments. The PD scheme seems to have certain distinct advantages over the ND scheme, both of which are illustrated in Section 3.6.

The definition of self, in most cases, is not complete. Hence, if a negative detector set is to be constructed on the basis of this self set, it is bound to be erroneous. The reason

Algorithm 14 Greedy Positive Selection Algorithm

```
//INPUTS: S, N_s, N_r, l
//OUTPUTS: PD
PD ← φ
i ← 0
j ← 0
while i < N_s do
  count[i] ← 0
  while j < N_s do
    if MatchString(S[i], S[j], l) then
      count[i] ← count[i] + 1
    end if
    j ← j + 1
  end while
  i ← i + 1
end while
Sort S
//The elements in S are sorted in descending order according to their count
i ← 0
while i < N_r do
  //Select the first N_r elements from S and include them in Ds
  PD ← S[i]
end while
```

is that the detector set may have members that actually belong to the self set, but are not included in the self set used in the construction of the detector set. This may result in false positives (FP), i.e., the self activities may be classified as non-self. When the PD scheme is used, the detectors are generated directly from the self set. The number of FPs in this case may exceed those obtained using the ND scheme as the detector set is merely a subset of the self set, which has an incomplete definition (there are unknown self strings). Thus, chances for a self activity to be identified correctly are lower for the PD scheme compared to the ND scheme. This can be illustrated using an example based on the data set depicted in Figure 3.2, where the length of the strings, l, in the binary universe is 3. The self set, S, consists of five strings and it is assumed that two of those strings are not recognized (unknown self). The remaining strings (known self) represent a subset of S, called the estimated self set (ES). The non-self set, NS, consists of the remaining three strings in the universe. A negative detector set, ND, with three members is constructed by picking random strings from S and NS that are not members of ES:

$$ND = \{000, 101, 111\} \tag{3.13}$$

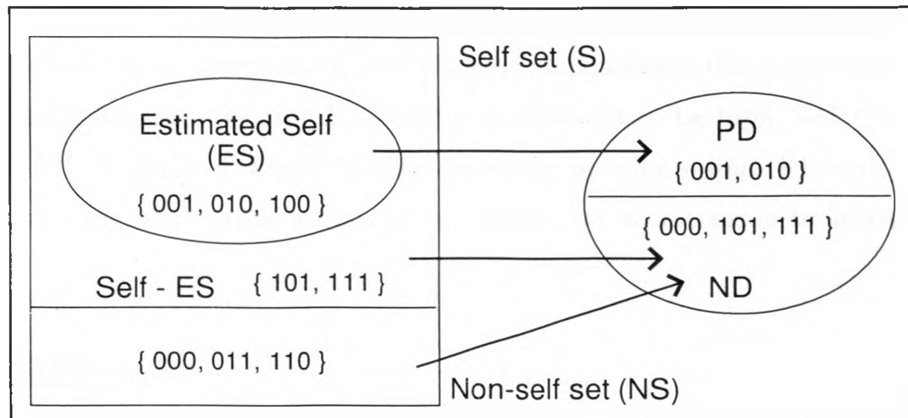


Figure 3.2. An Example to Illustrate PD and ND Schemes

Consider a system characterized by the universe described in Figure 3.2, which contains eight possible activities represented by the eight strings. If an activity represented by the string 101 is observed in that system, it would be classified as non-self because the string 101 is in ND, despite its being a self string. This results in an FP. If string 110 is observed, it would be classified as self given it is not in ND. However, 110 happens to be a non-self string; hence is a false negative (FN).

Further, if a positive detector set, PD, with two members is constructed using ES, it results in the set:

$$PD = \{001, 010\} \tag{3.14}$$

If the strings $100, 101$, and 111 are observed, they would result in FPs. However, the chance of FNs is very low and depends on the matching scheme used and its efficiency.

When the ND scheme is used, an observed self activity would be compared against all the detectors unless an FP detection (i.e., the self being classified as non-self) occurs. An observed non-self activity would be compared against the detector set only until a match is found, unless an FN detection (i.e., the non-self being classified as self) occurs. In the case of the PD scheme, an observed self activity would be compared against the detector set only until a match is found, unless FPs are detected. A non-self activity observed would be compared against all the detectors, unless FNs are detected. But the chances for occurrence of FNs are significantly less in the case of the PD scheme, unless the definition of self is incorrect. If the number of self activities is, on average, greater than the number of non-self activities in a system, the PD scheme involves less

comparison than the ND scheme. This could prove to be a significant factor in real-time systems wherein the number of self activities far outnumbers the abnormal ones.

In the PD scheme, the number of FPs is expected to be high, but it would be rare to have an FN. Though it would be disconcerting to have a high FP rate, it should be satisfactory to note that there is only a rare chance for an intrusion to succeed in the PD scheme.

3.5. EXPERIMENTS

Experiments were conducted using Algorithm 15 designed in this research to verify the implications discussed in Section 3.4. The detector set for the ND scheme was constructed using the exponential algorithm (Algorithm 3) and the greedy algorithm (Algorithm 4). The detector set for the PD scheme was constructed based on Algorithm 12 and Algorithm 14. The same self set was used by the PD and ND algorithms. After the construction of the detector sets, they were tested against one thousand non-self strings and the procedure was repeated one hundred times. The average number of detectors that were generated and the average number of incorrect classifications, P_f, for the detector sets to classify the non-self activities incorrectly, were recorded.

The r-contiguous match rule [5] was used for all experiments. In this method, the strings are supposed to match each other, if and only if they have identical symbols at r continuous locations. For example, if two strings, 1001 and 1000, are to be compared and r is set to "3", the r-contiguous match rule will produce a *true* result since both the strings match in the first three bit positions (if the least significant bit is expected to be the leftmost). A detailed explanation of the working methodologies of the r-contiguous match rule can be found in [5].

Algorithm 15 Procedure for Conducting Experiments

```
//INPUTS: N_s, N_r, l, m, r
count ← 0
P_f^PD ← FP^PD ← FN^PD ← 0
P_f^ND ← FP^ND ← FN^ND ← 0
while count < 100 do
  Construct S using Algorithm 2
  Construct ND using a negative detector generation algorithm
  //Use either exponential or greedy negative detector generation algorithm
  Construct PD using a positive detector generation algorithm
  //Use either exponential or greedy positive detector generation algorithm
  i ← 0
  while i < 1000 do
    DetectorString ← randomstring(l)
    //randomstring() generates a random string of length l
    if DetectorString ∈ S then
      //A case when the DetectorString belongs to self
      if DetectorString ∉ PD then
        FP_PD ← FP_PD + 1
      end if
      if DetectorString ∈ ND then
        FP_ND ← FP_ND + 1
      end if
    else
      //A case when the DetectorString belongs to non-self
      if DetectorString ∈ PD then
        FN_PD ← FN_PD + 1
      end if
      if DetectorString ∉ ND then
        FN_ND ← FN_ND + 1
      end if
    end if
    i ← i + 1
  end while
  count ← count + 1
end while
//Determine average values of P_f, FP and FN for PD and ND schemes
P_f^PD ← (FP^PD + FN^PD)/100
P_f^ND ← (FP^ND + FN^ND)/100
FP^PD ← FP^PD/100
FN^PD ← FN^PD/100
FP^ND ← FP^ND/100
FN^ND ← FN^ND/100
```

3.5.1. Experiments with Exponential Algorithm. Algorithm 3 generates detectors exhaustively, checking them against the self set to ensure that the generated detectors

do not match the self components. Experiments were conducted using the same values for the parameters that were used for the experiments in [8] to recreate the results. Table 3.4 lists those parameters and their corresponding values. Algorithm 12 was also run simultaneously and the results were recorded for both the algorithms.

Experiments were conducted for Algorithm 3 by varying the size of the self set, N_s and using parameter values suggested in [8]. The P_f, FPs, and FNs were recorded for the same. Experiments were conducted for Algorithm 12 by varying the number of detectors and the size of the self set. The P_fs, FPs, and FNs were recorded for the same. The results are shown in Table 3.5.

Table 3.4. Input Parameters for Experiments using Algorithm 3

N_r	46 (Non-self)
P_M	0.0502
m	2 (binary)
l	32
r	8

Table 3.5. Performance of the Exponential Non-self Algorithm (Algorithm 3) and Random Positive Algorithm (Algorithm 12)

N_s	ND (Algorithm 3)				PD (Algorithm 12)			
	N_r	FP	FN	P_f	N_r	FP	FN	P_f
8	46	0.867	0.077	0.346	4	0.442	0.0	0.151
16	46	0.856	0.071	0.516	8	0.397	0.0	0.224
24	46	0.862	0.061	0.640	12	0.343	0.0	0.248
32	46	0.854	0.056	0.700	16	0.296	0.0	0.239
48	46	0.834	0.023	0.769	24	0.216	0.0	0.180
64	46	0.824	0.001	0.794	32	0.158	0.0	0.152
80	46	0.806	0.003	0.794	40	0.109	0.0	0.107
96	46	0.801	0.004	0.796	48	0.076	0.0	0.076

3.5.2. Experiments with Greedy Algorithm. The greedy algorithm, discussed in [9], is aimed at generating non-self detectors that are far apart from one another, thus

providing better coverage with a minimal number of detectors. Also, it is possible to generate all the possible sets of valid detectors for a given self set in linear time with this particular algorithm. Experiments were conducted for different string lengths l and matching lengths r, as suggested in [10]. The size of the self set, N_s, was limited to 250 members. The greedy algorithm was set up to generate all the possible negative detectors. Random strings were generated and tested against the negative detectors to evaluate the performance of the detector set. The P_fs, FPs, and FNs were recorded in each case. The PD algorithm was also run simultaneously using the same self set, S, but the size of detector sets, N_r, were varied so as to determine the impact of the detector set size on performance. The P_fs, FPs, and FNs were recorded for the different values of N_r. The results are shown in Table 3.6, where N_r^* represents the number of detectors generated, as per the results published in [9].

Table 3.6. Performance of the Greedy Non-self Algorithm (Algorithm 4) and Random Positive Algorithm (Algorithm 12). The values within parenthesis in Column 4 are the standard deviations for 100 sample runs.

N_s	l	r	ND (Algorithm 4)					PD (Algorithm 12)			
			N_r	N_r^*	FP	FN	P_f	N_r	FP	FN	P_f
250	16	10	741 (2.03)	793	0.95	0.0003	0.60	125	0.34	0.0	0.22
250	16	9	230 (3.85)	320	0.89	0.0005	0.79	125	0.21	0.0	0.19
250	16	8	34 (6.47)	88	0.50	0.0003	0.49	125	0.55	0.0	0.06
250	32	11	1685 (2.19)	1796	0.99	0.0000	0.76	125	0.31	0.0	0.24
250	32	10	615 (3.10)	821	0.99	0.0000	0.95	125	0.17	0.0	0.16
250	32	9	143 (5.08)	378	0.98	0.0000	0.98	125	0.03	0.0	0.03
250	32	8	14 (7.72)	89	0.53	0.0000	0.53	125	0.00	0.0	0.00

3.6. RESULTS AND DISCUSSION

Table 3.5 and Table 3.6 clearly show that the P_f values for the detector sets generated using Algorithm 12 are low. It is observed that Algorithm 12 is much faster when compared to the ND scheme algorithms, since the size of the detector sets in the case of the PD scheme is very small compared to the ND scheme, for the same P_f.

It is also interesting to note that the number of FPs and FNs decrease with an increase in the size of the self set N_s. The inference made here is that the size of the self set defines the accuracy with which the normal activities in a system are defined. The

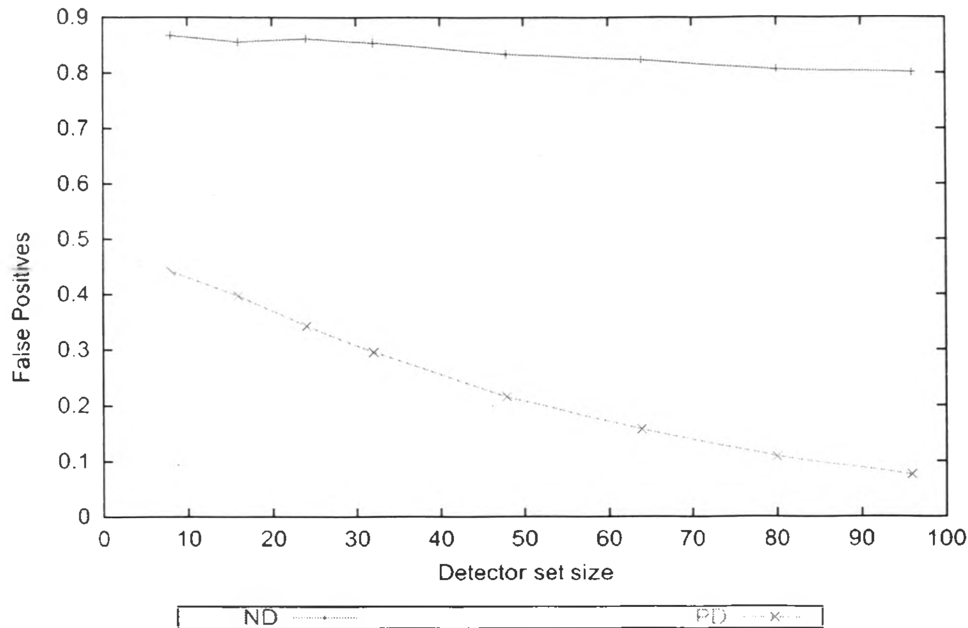


Figure 3.3. Performance Comparison of ND and PD for FPs - ND: Exponential Algorithm (Algorithm 3), PD: Random Algorithm (Algorithm 12)

matching length, r, also plays a significant role in the performance of IDS. It can be seen from Table 3.6 that with the reduction in the matching lengths, the FPs and FNs are also reduced. This is because the detector set tends to become more general with broader coverage.

Figure 3.3, Figure 3.4 and Figure 3.5 show the performance of detector sets in terms of FPs, FNs, and P_fs, respectively, based on the variation in the number of self set members. It can be seen that the FPs and FNs are reduced with the increase in the size of the self set. This is because the clearer the definition of the self set, the fewer the chances for misclassification. Figure 3.6 and Figure 3.7 show the effect of string lengths and matching lengths on the performance of the system. Generally, with the reduction in matching length, the failure rates are also minimized. This is again due to the fact that detector sets tend to behave like generalists with smaller matching lengths. It is also observed that an increase in the length of the strings generally results in higher failure rates. This is because the larger strings have more information encoded and comparison operations do not effectively resolve the encoded information.

Results show that the non-self detectors result in more FPs than FNs. This is mainly due to the matching rule used. Since the matching rule used is unsophisticated, the non-self strings in the detector set match the self strings that are generated at random during

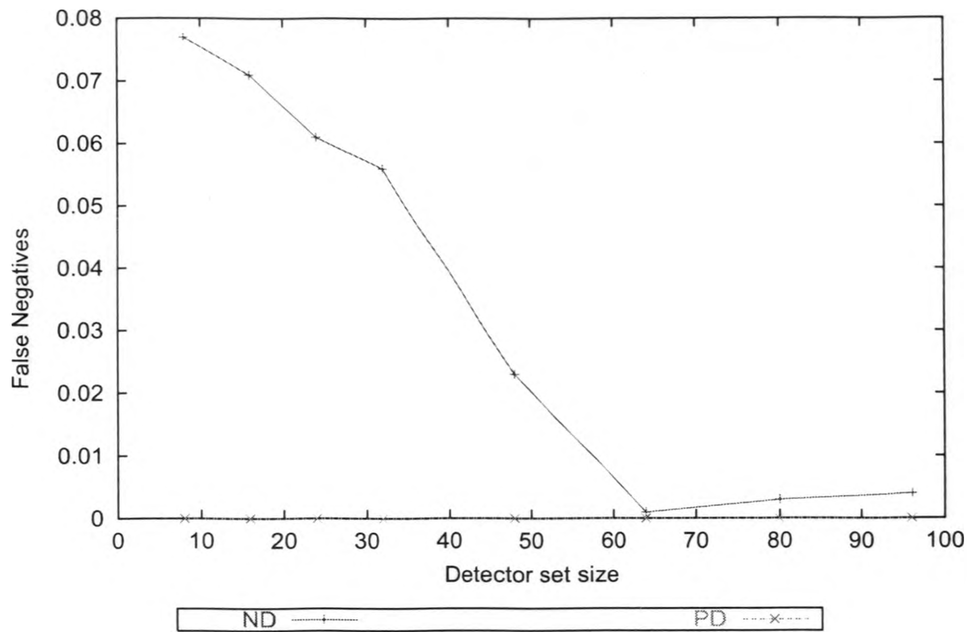


Figure 3.4. Performance Comparison of ND and PD for FNs - ND: Exponential Algorithm (Algorithm 3), PD: Random Algorithm (Algorithm 12)

the tests conducted. The random strings used for testing are generated using the self set and are labeled as either self or non-self. Since the matching process is performed using the r-contiguous match rule, the test strings, labeled non-self because they do not match any member of the self set, would not match the strings in the detector set (a subset of the self set) as well. This explains why the FNs for the positive detector set are always zero. Also, for the same reason, the FPs obtained may not be a true reflection of the performance of the detector sets. But it is evident that the PD scheme performs better in cases where the self strings are fewer in number.

When the size of the self set exceeds the non-self set size, the performance of the detectors is expected to be better. No explicit experiments were done to investigate this since the non-self and the self strings are complementary.

3.7. APPLICATION OF CLONAL EAS AND SELF/NON-SELF PRINCIPLES

The Clonal EAs and the PD/ND schemes discussed previously have been implemented in a prototype IDS described in this section. The IDS uses a pattern recognition algorithm termed IDSClonalg, a Clonal EA, and implements PD/ND schemes in IDSClonalg to classify a data set into two classes, namely *normal* and *intrusive*. Many intrusion

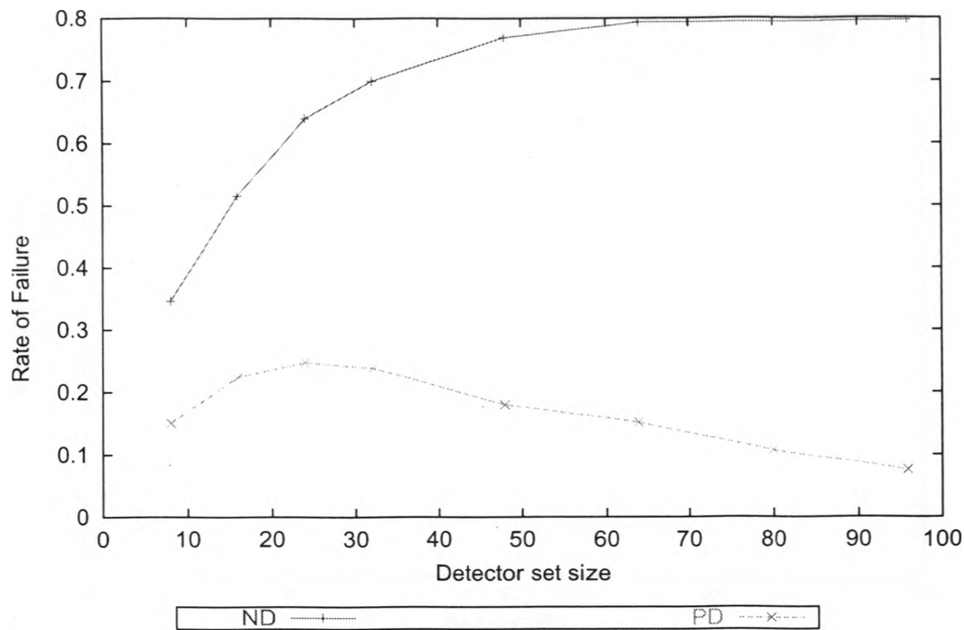


Figure 3.5. Performance Comparison of ND and PD for P_fs - ND: Exponential Algorithm (Algorithm 3), PD: Random algorithm (Algorithm 12)

detection algorithms based on Clonal EAs have been proposed in [1, 12, 13, 14] for detecting intrusions in a system. However, all these algorithms use the ND scheme to generate detectors. In this research, both the PD and ND schemes are implemented in a Clonal EA framework and their performance is compared for a static data set.

The KDD Cup 1999[2] data set was used for this purpose. The data was processed using various data cleaning and transformation techniques before being separated into *TrainData* and *TestData* sets. The *TrainData* set was provided as input to the IDS and detectors were generated. The detectors were then used to evaluate the *TestData* set members to perform the classification. It must be noted that the members of the original data set were labeled as belonging to the *normal* or *intrusive* class, thus facilitating comparison of the performance of the IDS when used with the PD and ND schemes.

3.7.1. Data Preparation. The KDD Cup 1999 data set contains snapshots of a system's state recorded at regular time intervals. Activities simulating attacks had been performed on the monitored system to facilitate generation of data that reflected abnormal/intrusive behavior. The data set is composed of several rows of such snapshots, each possessing forty one heterogeneous data type attributes representing various system

[2]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

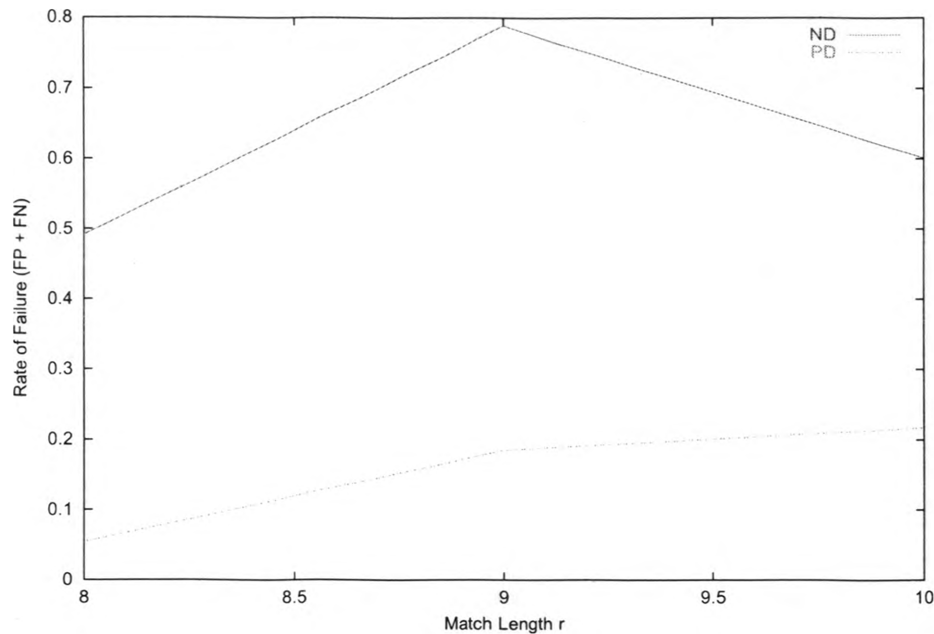


Figure 3.6. Performance Comparison of ND and PD for P_fs - ND: Greedy Algorithm (Algorithm 4), PD: Random Algorithm (Algorithm 12) and $l = 16$

characteristics. Each row of data is henceforth referred to as a data member or vector and a collection of all such rows of data is referred to as a data set.

Detailed information on the methods used to prepare the data set can be found at `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`. In order for the IDS to perform efficient classification, it is essential that all the attributes possess values belonging to the same data type and that their values are within a uniform range. An example set of attributes that constitute a data member is provided in Figure 3.8. Note that the figure shows only a partial list of attributes to facilitate understanding the nature of the data involved.

The original data set possessed forty one attributes excluding a *label* attribute. The *label* attribute represented the class which the data member belonged to. It was removed from the data set to avoid its values being tampered with during the data processing phase. The data set was then processed to ensure that all the forty one attributes belonged to the same data type (numeric). This was done by creating look-up tables for attributes that possessed text data type values. For example, if there was an attribute whose value set was $\{A, B, C\}$ then a look-up table was created as $\{A - 0, B - 1, C - 2\}$. The numeric values from the look-up table were used to replace the text data value for that attribute for each data member. The data set was then normalized such that all the

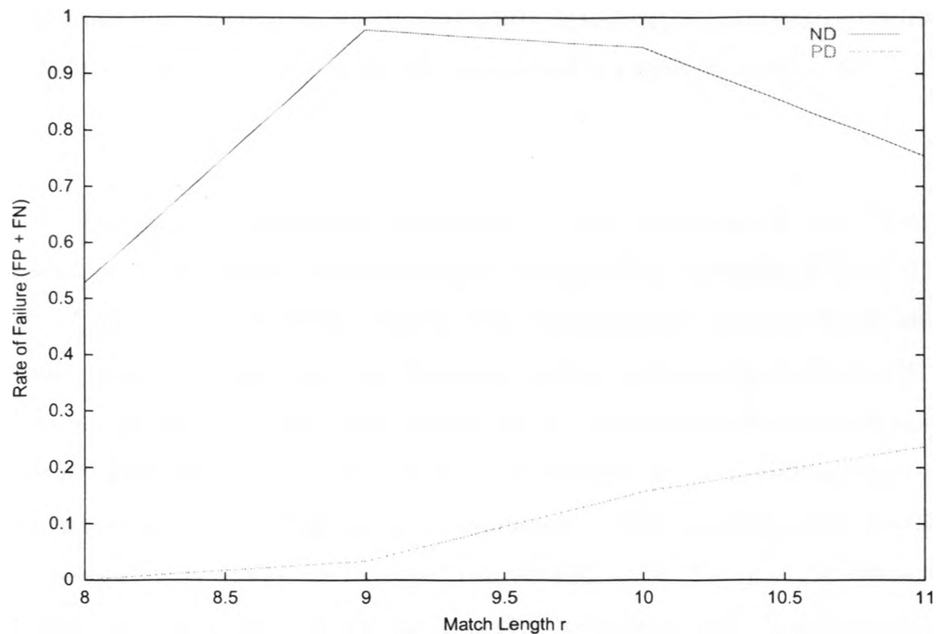


Figure 3.7. Performance Comparison of ND and PD for P_fs - ND: Greedy Algorithm (Algorithm 4), PD: Random Algorithm (Algorithm 12) and $l = 32$

Protocol Type	Service	Num_failed logins	Src_bytes	Dst_bytes	urgent	Serror rate	Class

Figure 3.8. A Sample Data Schema with Critical Attributes

attributes possessed values in the range $[0, 1]$. For example, if there was an attribute with values in the range $[0, 10000]$, then the maximum value for that attribute was identified and the values of that attribute for all the data members were divided by the maximum value. After processing the data, the *label* attribute was added back to the data set.

3.7.1.1. Data Separation. The processed data set was condensed into a smaller set by using class-based proportional sampling. Data members belonging to different labeled classes were selected based on the number of members in the original data set. For example, if there were 50 members belonging to class C in the original set containing 100 members and if the size of the condensed set was set to 50 members, then 25 members belonging to class C were uniform randomly selected into the condensed set. Two subsets

were created from the condensed set by randomly inserting 60 percent of its members into the *TrainData* set and the remaining 40 percent of its members into the *TestData* data set.

3.7.1.2. Principal Component Analysis. The members of the *TrainData* and *TestData* sets were processed using Principal Component Analysis (PCA) [15] to reduce the number of attributes, thereby making the classification process more efficient. The *label* attribute was removed from the data sets before subjecting them to PCA. The important features of the data sets were extracted by discarding those attributes that were correlated with each other. Correlated data attributes lead to redundancy in the data set without necessarily providing more information. PCA extracts only the uncorrelated attributes. Hence, only those attributes that significantly impact the detection process were identified and extracted. Thus processed *TrainData* and *TestData* sets possessed only ten attributes instead of the original set of forty one. These ten attributes were most significant in terms of differentiating normal activities from intrusions. Finally, the *label* attribute was reattached to the data sets.

3.7.2. IDS Design. The IDS was comprised of two main components, namely Generator, the detector generation part, and Evaluator, the data evaluation part.

3.7.2.1. Generator. The Generator is responsible for creating positive and negative detectors based on the input data provided to it. Previously, deterministic algorithms such as the exponential algorithm, the greedy algorithm and the random positive selection algorithm were used to generate detectors for a binary system. But, considering the fact that the *KDDCup*1999 data set was huge and that there were thousands of rows of data to be analyzed, a stochastic algorithm such as a Clonal EA was required. Thus, the generator incorporated the IDSClonalg algorithm designed for generating detectors based on the *TrainData* set.

IDSClonalg: The IDSClonalg algorithm was designed as a pattern recognition algorithm. The *TrainData* set members were considered as patterns associated with a particular labeled class, namely *normal* or *intrusive*. IDSClonalg was designed to learn the patterns from the *TrainData* set and express them as detectors. The algorithm design involved modeling the detector generation problem as Clonal EA components. The following notations are used in the description of IDSClonalg:

1. M_{ag} represents an antigen set comprising the problem scenarios. This set contains a number of patterns which need to be approximated and learned. In order to generate detectors, this set needs to be populated with $TrainData$ set members along with their *label* attribute.

2. M_n represents a normal solution set. The members of this set represent the detectors that are being developed and possess all the attributes that the members of $TrainData$ and $TestData$ sets do, including the *label* attribute. Members of this set also possess an additional attribute termed $fitness$. The $fitness$ attribute determines how efficient a detector is.

3. M_m represents a memory solution set. The members of this set usually represent a specific pattern that was learned. So this set should contain only two detectors, one belonging to the *normal* class and the other belonging to the *intrusive* class. But, considering the fact that there were a large number of members in the antigen set that belonged to the *normal* and *intrusive* classes, it would not be sufficient to have a single detector represent each class. Thus, the cardinality of the detector set was set to ten and of these, five were positive detectors that represent the *normal* class and the rest were negative detectors that represent the *intrusive* class. The members of this set possess all the attributes that the members of $TrainData$ and $TestData$ sets do, including the label attribute. Members of this set also possess an additional attribute termed $fitness$.

   The attributes and fitness of the members in M_m are initialized to zero. These members are replaced with members from M_n, which possess the highest fitness when evaluated against an antigen. The replacement is performed only between members of the same class represented by their labels.

4. M_p represents a population set which is the union of the normal and memory sets ($M_p = M_n \bigcup M_m$).

5. M_c represents a clone set.

6. n_{ag} represents the cardinality of M_{ag}.

7. n_n represents the cardinality of M_n.

8. n_m represents the cardinality of M_m.

9. n_p represents the cardinality of M_p.

10. n_c represents the cardinality of M_c.

11. n_c^i represents the number of clones of member i in M_p that are to be generated in M_c.

12. M_{ag}^x represents a member x in the antigen set as:

$$M_{ag}^x = \{a_x^1, a_x^2, \cdots, a_x^{10}, label_{ag}^x\} \tag{3.15}$$

13. M_n^x represents a member x in the normal set as:

$$M_n^x = \{a_1^x, a_2^x, \cdots, a_{10}^x, label_{ag}^x, f_n^x\} \tag{3.16}$$

14. M_c^x represents a member x in the clone set.

15. f_c^x represents the fitness value of a member x in M_c.

16. $IDSFitness(Detector, Antigen)$ represents a function that assigns a fitness value to a detector in M_p or M_c when compared with an antigen in M_{ag}. It uses the Manhattan distance metric to determine the distance, $d_{Detector}$, between a *Detector* and an *Antigen*. Since the *Detector* and *Antigen* possess ten attributes each, with values in the range $[0, 1]$, the distance value should be in the range $[0, 10]$. The fitness of a *Detector* is then expressed as $f_{Detector} = 10 - d_{Detector}$. For example, if the fitness function is executed as $Fitness(M_p^x, M_{ag}^y)$, then:

$$Detector = M_p^x = [0.5, 0.5, 0.5, 0.5, 0.0, 0.4, 0.6, 0.8, 0.9, 1.0] \tag{3.17}$$

$$Antigen = M_{ag}^y = [0.4, 0.4, 0.4, 0.4, 1.0, 0.4, 0.6, 0.8, 0.9, 1.0] \tag{3.18}$$

The Manhattan distance between *Detector* and *Antigen* is computed as:

$$d_{Detector} = \sum_{i=1}^{10} |Detector_i - Antigen_i| \tag{3.19}$$

Hence,

$$d_{Antigen} = 1.4 \tag{3.20}$$

Then, the fitness $f_{Detector}$ of *Detector* is computed as:

$$f_{Detector} = 10 - d_{Detector} \quad (3.21)$$

$$f_{Detector} = 8.6 \quad (3.22)$$

Finally,

$$f_p^x = f_{Detector} \quad (3.23)$$

A fitness value close to "10" is considered as a high fitness value and a fitness value close to "0" is considered as a low fitness value. Algorithm 16 summarizes the fitness computation process.

Algorithm 16 IDSFitness: A function for computing the fitness of members in M_n

//INPUTS: *Detector*, *Antigen*
for $i \leftarrow 1$ to 10 **do**
  $d_{Detector} \leftarrow d_{Detector} + |Detector_i - Antigen_i|$
end for
$f_{Detector} \leftarrow 10 - d_{Detector}$

17. $IDSCreateClones(n, \gamma, label)$ represents a function that populates M_c with clones created from the first n members of M_p. The number of clones created for each of the first n members is computed using Equation 2.1. Algorithm 17 explains the process for creating the clones.

18. $IDSHyperMutate(Clone)$ represents a function that performs hyper-mutation. The upper bound on the maximum amount by which a clone's attribute values can be altered is termed *ceil*. The value of *ceil* is calculated as:

$$ceil = 1 - (f_{Clone})/10 \quad (3.24)$$

For example, if $f_{Clone} = 8.6$, then $ceil = 0.14$. Thus, the higher the fitness of a member, the smaller the range of values by which the attributes of the member can be altered. The $IDSHyperMutate(clone)$ function is defined in Algorithm 18.

Algorithm 17 IDSCreateClones: A function for creating multiple clones of members in M_n

//INPUTS: n, γ, $label$
$k \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $numclones \leftarrow \lceil(\gamma \cdot n_p)/i\rceil$
  for $j \leftarrow 1$ to $numclones$ **do**
    $k \leftarrow k+1$
    $M_c^k \leftarrow M_p^i$
    $label_c^k \leftarrow label$
  end for
end for
$n_c \leftarrow k$

Algorithm 18 IDSHyperMutate: A hyper-mutation function

//INPUTS: $Clone$
$ceil \leftarrow 1 - (f_{Clone})/10$
for $i \leftarrow 1$ to 10 **do**
  $disp \leftarrow random(0, ceil)$
  // The random() function generates a value between 0 and $ceil$
  if $random() \leq 0.5$ **then**
    // The random() function generates a value in the range $(0, 1)$
    if $Clone[i] + disp \leq 1.0$ **then**
      $Clone[i] \leftarrow Clone[i] + disp$
    end if
  else
    if $Clone[i] - disp \geq 0$ **then**
      $Clone[i] \leftarrow Clone[i] - disp$
    end if
  end if
end for

19. $InitNormal()$ represents a function that assigns uniform random values to the attributes in members of M_n and sets their label values to null. The $InitNormal()$ function is defined in Algorithm 19.

20. $InitMemory()$ represents a function that initializes the attributes in members of M_m to zero and sets the label values for the first five members to *normal* and the last five members to *intrusive*. The $InitMemory()$ function is defined in Algorithm 20.

The IDSClonalg design based on the notations described previously is provided in Algorithm 21. The algorithm first initializes the members of M_m, M_n and M_{ag}. The

Algorithm 19 InitNormal: A function for intializing members of M_n

for $i \leftarrow 1$ to n_n **do**
  $fitness_n^i \leftarrow 0$
  $label_n^i \leftarrow null$
  for $j \leftarrow 1$ to 10 **do**
    $a_j^i \leftarrow random(0, 1)$
  end for
end for

Algorithm 20 InitMemory: A function for intializing members of M_m

for $i \leftarrow 1$ to 5 **do**
  $fitness_n^i \leftarrow 0$
  $label_n^i \leftarrow$ "*normal*"
  for $j \leftarrow 1$ to 10 **do**
    $a_j^i \leftarrow 0$
  end for
end for
for $i \leftarrow 6$ to 10 **do**
  $fitness_n^i \leftarrow 0$
  $label_n^i \leftarrow$ "*intrusive*"
  for $j \leftarrow 1$ to 10 **do**
    $a_j^i \leftarrow 0$
  end for
end for

members of M_m and M_n are then combined to form M_p and the members of M_p are evaluated against each antigen in M_{ag}. Based on their fitness, members of M_p are subjected to clonal expansion and the clones are populated in M_c. The clones then undergo hypermutation and their fitness is evaluated. Based on the fitness of the clones, the memory set and the normal set are altered. The whole process is repeated for a specified number of generations, *numgenerations*. *numparents* represents the number of members from M_p that are to be selected for cloning.

3.7.2.2. Evaluator. This component is responsible for using the detectors generated by the IDS to classify the *TestData* set members as *normal* or *intrusive*. The evaluator extracts members of M_m which belong to the *normal* class and inserts them into a new set of positive detectors, M_{PD} and extracts members of M_m which belong to the *intrusive* class and inserts them into a new set of negative detectors, M_{ND}. The cardinality of M_{PD} is $n_{PD} = 5$ and the cardinality of M_{ND} is $n_{ND} = 5$. The cardinality of *TestData* set is

$n_{TestData}$. These two detector sets are then used to implement the PD and ND schemes to classify members of the *TestData* set into *normal* or *intrusive* classes.

PD Scheme: In this scheme, members of the *TestData* set are compared against the detectors in M_{PD}. A member, $TestData^i$, of the *TestData* set is compared against all the members of M_{PD} to determine the Manhattan distance between them and $TestData^i$. Then the minimum of those distances is identified and compared with a tolerance value, Tol_{PD}. If the minimum Manhattan distance is less than Tol_{PD}, then $TestData^i$ is classified as *normal* since $TestData^i$ is considered to closely match the patterns represented by the detectors in M_{PD}. But, if the minimum Manhattan distance is greater than Tol_{PD}, the $TestData^i$ is classified as *intrusive*. The tolerance value Tol_{PD} sets the delineation between the *normal* and *intrusive* classes and is computed as in Algorithm 22.

The purpose is to determine a Tol_{PD} value based on the maximum average Manhattan distance between the *TestData* set and M_{PD}. This would enable the determination of the extent of tolerance that could be allowed for declaring a match. The classification function for the PD scheme is defined in Algorithm 23.

The objective is to determine the closest possible match between a member of M_{PD} and the data member under consideration, $TestData^i$, and compare it with the Tol_{PD} value. The variable $computedlabel^i_{TestData}$ records the class that the evaluator assigns to $TestData^i$. The performance of the evaluator can be determined by comparing $computedlabel^i_{TestData}$ against $label^i_{TestData}$, which provides the actual classification for $TestData^i$. Algorithm 24 demonstrates the procedure for evaluting the performance of the PD scheme.

ND Scheme: In this scheme, members of the *TestData* set are compared against the detectors in M_{ND}. A member, $TestData^i$, of the *TestData* set is compared against all the members of M_{ND} to determine the Manhattan distance between them and $TestData^i$. Then the minimum of those distances is identified and compared with a tolerance value, Tol_{ND}. If the minimum Manhattan distance is less than Tol_{ND}, then $TestData^i$ is classified as *intrusive* since $TestData^i$ is considered to closely match the patterns represented by the detectors in M_{ND}. But, if the minimum Manhattan distance is greater than Tol_{ND}, $TestData^i$ is classified as *normal*. The tolerance value Tol_{ND} sets the delineation between the *normal* and *intrusive* classes. The computation of Tol_{ND}, the classification function and the performance of the evaluator for the *ND* scheme, is exactly the same as for the *PD* scheme. The only modification would be that the set M_{PD} would be replaced by M_{ND}.

3.7.3. Results. The accuracy of the IDS is evaluated on the basis of the number of correct classifications, the number of FPs and the number of FNs. Table 3.7 shows the best results obtained using positive and negative detectors. It is evident from the results that the positive detectors significantly outperformed the negative detectors. It was noted from the data that there were certain unique "positive patterns" that could be identified based on the *normal* class data; whereas, the *intrusive* class data was inconsistent in terms of its characteristics so that no such patterns could be identified.

Table 3.7. IDSClonalg Results for Self, Non-self Detectors. (* Percentage values of results averaged over 50 trials, Values in parenthesis represent standard deviations)

Scheme	No. of Detectors	*Correct Classifications	*False Positives	*False Negatives
PD	5	99.07 (0.21)	0.37 (0.08)	0.56 (0.13)
ND	5	37.36 (2.46)	0.0 (0.00)	62.64 (4.12)

This resulted in an inability to set the Tol_{ND} value, based on M_{ND}, along a line that delineates the *intrusive* class from the *normal* class. Thus, a Tol_{ND} value was set such that it was always below the $dmin^{i}_{TestData}$ values of $TestData$ members, regardless of whether $TestData^{i}$ belonged to the *normal* or *intrusive* class. A high percentage of *intrusive* class data was classified incorrectly mainly because the Tol_{ND} value could not be set such that it provided an accurate upper bound for the distance metric. Hence, the evaluator for the ND scheme classified only 37.36 percent of $TestData$ members correctly. It can be seen that all the self activities were correctly classified ($FPs = 0\%$) and that most of the non-self activities were classified incorrectly. The reason might be that *intrusive* data used for training ($TrainData$) differed from that used for testing ($TestData$), which may have resulted in the detectors being closer to the *intrusive* class members in $TrainData$ and not to the *intrusive* class members in $TestData$.

In the case of the PD scheme, a Tol_{PD} value was identified along the boundaries of *normal* data, which greatly improved the classification process. This underscores the importance of using the PD scheme when intrusions are sporadically distributed in terms of their characteristics. In such cases, the classification process can restrict itself to checking if a datum belongs to the *normal* class, using positive detectors that can be generated based on a consistent pattern associated with data belonging to the *normal* class.

3.7.4. Conclusion. IDSClonalg uses labeled classes of data to identify intrusions. Though IDSClonalg is used with labeled classes of data, it is also possible for the algorithm to be provided with unlabeled data to generate detectors. In such cases, its performance is expected to decrease. But, the simplicity of the solution with a relatively low number of detectors, compared to the overhead involved in detection systems such as those that involve neural networks, makes it a better choice for real-time IDS.

Algorithm 21 IDSClonalg: An algorithm for generating positive and negative detectors

```
//INPUTS: numgenerations, numparents, γ, ℓ
Call InitRandom()
Call InitMemory()
M_ag ← TrainData
for iter ← 1 to numgenerations do
  //Loop for specified number of generations
  for i ← 1 to n_ag do
    //Loop through all the antigens in M_ag
    label ← label^i_ag
    //Record the label of the antigen, which is to be assigned to the clones
    M_p ← M_n ∪ M_m
    for j ← 1 to n_p do
      Call IDSFitness(M^j_p, M^i_ag)
    end for
    Call Sort(M_p)
    //Sorts members of a set in descending order of fitness
    Call IDSCreateClones(numparents, γ, label)
    for j ← 1 to n_c do
      Call IDSHyperMutate(M^j_c)
    end for
    for j ← 1 to n_c do
      Call IDSFitness(M^j_c, M^i_ag)
    end for
    Call Sort(M_c)
    if label = "normal" then
      startindex ← 1
      endindex ← 5
    else
      startindex ← 6
      endindex ← 10
    end if
    for j ← startindex to endindex do
      if f^1_c > f^j_m then
        M^j_m ← M^1_c //Insert the highest fitness member into M_m
      end if
    end for
    nindex ← n_n − ℓ
    for cindex ← n_c − ℓ to n_c do
      M^nindex_n ← M^cindex_c //Repertoire Diversity
      nindex ← nindex + 1
    end for
  end for
end for
```

Algorithm 22 Computation of Tolerance Value

$sum_{TestData} \leftarrow 0$
for $i \leftarrow 1$ to $n_{TestData}$ **do**
  $dmax^{i}_{TestData} \leftarrow 0$
  for $j \leftarrow 1$ to n_{PD} **do**
    Determine the Manhattan distance, $d^{ij}_{TestData}$ between $TestData^{i}$ and M^{j}_{PD}
    if $d^{ij}_{TestData} > dmax^{i}_{TestData}$ **then**
      //Determine the maximum of the Manhattan distances between a member
      //in $TestData$ and those of M_{PD}
      $dmax^{i}_{TestData} \leftarrow d^{ij}_{TestData}$
    end if
  end for
  $sum_{TestData} \leftarrow sum_{TestData} + dmax^{i}_{TestData}$
end for
$Tol_{PD} \leftarrow sum_{TestData}/n_{TestData}$
//Determine the average of the maximum distances of all the members of $TestData$ from the members of M_{PD}

Algorithm 23 Data Classification Function

for $i \leftarrow 1$ to $n_{TestData}$ **do**
  $dmin^{i}_{TestData} \leftarrow 100$
  for $j \leftarrow 1$ to n_{PD} **do**
    Determine the Manhattan distance, $d^{ij}_{TestData}$ between $TestData^{i}$ and M^{j}_{PD}
    if $d^{ij}_{TestData} < dmin^{i}_{TestData}$ **then**
      //Determine the minimum of the Manhattan distances between a member
      //in $TestData$ and those of M_{PD}
      $dmin^{i}_{TestData} \leftarrow d^{ij}_{TestData}$
    end if
  end for
  if $dmin^{i}_{TestData} < T_{PD}$ **then**
    $PDcomputedlabel^{i}_{TestData} \leftarrow normal$
  else
    $PDcomputedlabel^{i}_{TestData} \leftarrow intrusive$
  end if
end for

Algorithm 24 Performance Evaluation for PD Scheme

$CorrectClassifications_{PD} \leftarrow 0$
$IncorrectClassifications_{PD} \leftarrow 0$
$FalsePositives_{PD} \leftarrow 0$
$FalseNegatives_{PD} \leftarrow 0$
for $i \leftarrow 1$ to $n_{TestData}$ **do**
  if $PDcomputedlabel^{i}_{TestData} = label^{i}_{TestData}$ **then**
    $CorrectClassifications_{PD} \leftarrow CorrectClassifications_{PD} + 1$
  else
    if $PDcomputedlabel^{i}_{TestData} = normal$ AND $label^{i}_{TestData} =$ "$intrusive$" **then**
      $FalseNegatives_{PD} \leftarrow FalseNegatives_{PD} + 1$
    end if
    if $PDcomputatedlabel^{i}_{TestData} = intrusive$ AND $label^{i}_{TestData} =$ "$normal$" **then**
      $FalsePositives_{PD} \leftarrow FalsePositives_{PD} + 1$
    end if
  end if
end for
$IncorrectClassfications_{PD} \leftarrow n_{TestData} - CorrectClassifications_{PD}$

4. FACTS PLACEMENT IN A POWER GRID

A power grid is an enormous electrical power transmission network. It is made up of power transmission lines, alternatively referred to as lines, that carry power across the network and buses which act as hubs relaying power delivered to them by transmission lines to other transmission lines and consumer loads. A power grid is thus a collection of buses interconnected by transmission lines. Failure of transmission lines, termed line contingencies, can trigger cascading failures in the power grid [16]. A cascading failure occurs when the failure of one transmission line causes other power lines to carry excess power. If such a condition is not immediately rectified, it might lead to failure of a large number of transmission lines.

The placement of FACTS devices on ideal locations could help distribute the excess load on overloaded lines across other transmission lines in the grid [16]. The FACTS devices are very expensive and it is important to use as few devices as possible. Hence, it is necessary to find ideal placements for a limited number of FACTS devices such that the contingencies are handled efficiently, allowing the grid to operate under optimal conditions. Given the fact that there are an extremely large number of possible locations to place the FACTS devices in a power grid, the complexity of the problem is significant. In this section, Clonalg, a Clonal EA designed for function optimization, is used to determine optimal placement positions for FACTS devices. To evaluate its performance, a standard evolutionary algorithm and two greedy algorithms were implemented to determine comparative FACTS device placements.

4.1. THE POWER GRID MODEL

In this section, a simulation model developed to represent a power grid, the power flows, contingencies and the placement of FACTS devices, is explained.

4.1.1. Power Grid.

A power grid system termed "IEEE 118-bus test system" [17] was considered for performing the experiments. The grid system included 118 buses and 179 lines and several consumer loads. In this grid system, there can only be one line connecting any two buses. Figure 4.1 shows the layout of the buses and the lines connecting them along with the loads represented as circles.

In order to determine the ideal placement positions for FACTS devices on a grid, it is necessary to establish a set of metrics for evaluating the placements. FACTS devices are placed on lines and are configured to regulate power across those lines. A set of FACTS

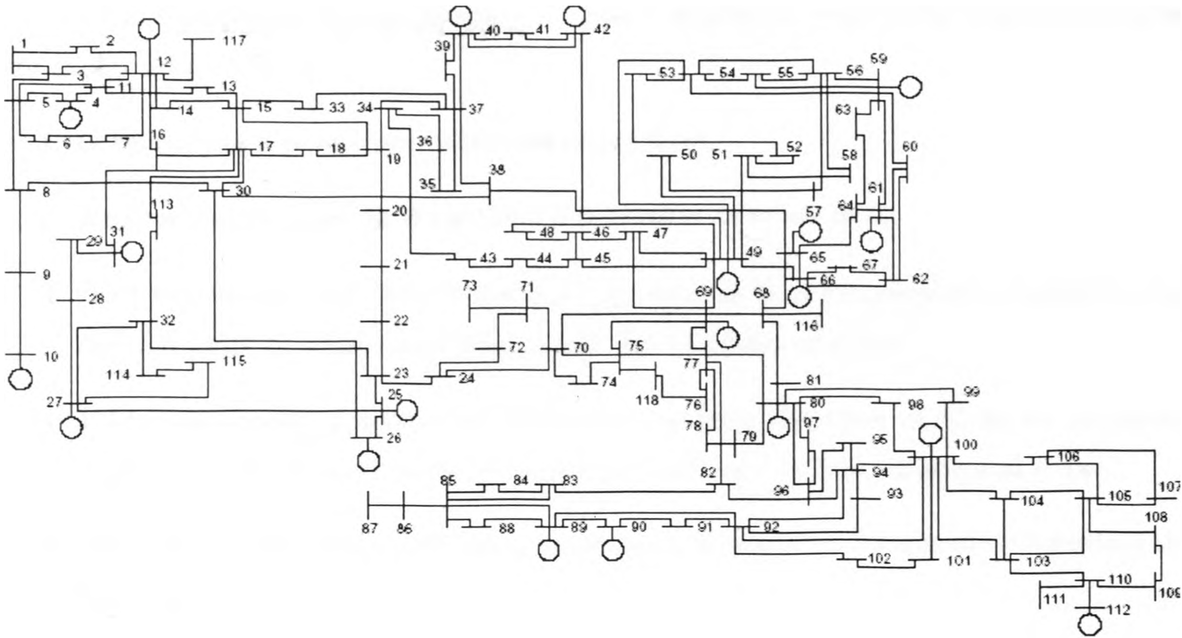


Figure 4.1. The IEEE 118-bus test system

devices placed on certain lines is henceforth referred to as a "FACTS device configuration". It is important to place the FACTS devices on critical lines such that when the power flow across those lines is regulated, the power flow across the whole grid is regulated. Thus, it is necessary to determine the power flow across the whole grid so that the power flow across individual lines can be computed. Those lines that carry power more than their specified capacities are considered to be overloaded. The overloaded lines that cause a large number of lines in the grid to be overloaded are considered critical lines. The main aim of this research is to develop algorithms to identify such critical lines and place the FACTS devices on them, thus regulating the power flow across a large number of lines.

4.1.2. Component Representation. The components used in the simulation model are explained as follows:

1. G represents a power grid with buses and lines.
2. Bus represents a set that contains a collection of buses in G.
3. $Line$ represents a set that contains all the lines in G as its members, represented as ordered pairs of source and destination bus numbers of a line.
4. $VLine$ represents a subset of $Line$ that contains the lines in G as its members, represented as ordered pairs of source and destination bus numbers of a line.
5. $BusConfig$ represents a set that maintains configuration information regarding the buses in G.
6. $LineConfig$ represents a set that maintains configuration information related to all the lines in G.
7. Bus_i represents a bus i in G.
8. $Line_{ij}$ represents a line between Bus_i and Bus_j.
9. cap_{ij} represents the power carrying capacity of a line, $Line_{ij}$.
10. $flow_{ij}$ represents the amount of power flowing across $Line_{ij}$ with the direction of power flow represented by the sign (positive or negative) assigned to $flow_{ij}$.
11. fac_{ij} represents a FACTS device placed on a line, $Line_{ij}$.
12. $numoverload_{ij}$ represents the number of times $Line_{ij}$ is overloaded for a certain number of single line contingencies (SLCs), further explained in Section 4.1.3..
13. $overload_{ij}$ represents the sum of the amounts $(flow_{ij} - cap_{ij})$ by which $Line_{ij}$ is overloaded during each SLC.
14. n_{Bus} represents the cardinality of Bus.
15. n_{VLine} represents the cardinality of $Vline$.
16. $nfacts$ represents the total number of FACTS devices available for placement in the grid.

17. $Fconfig_k^{nfacts}$ represents a configuration k wherein $nfacts$ FACTS devices are placed on certain lines:

$$Fconfig_k^{nfacts} = \{fac_{i_1j_1}, fac_{i_2j_2}, \cdots, fac_{i_{nfacts}j_{nfacts}}\} \tag{4.1}$$

There cannot be multiple configurations such as $Fconfig_5^6$ and $Fconfig_5^7$ because each configuration is unique and adding a facts device to $Fconfig_5^6$ will only create a new configuration $Fconfig_6^7$. But, there can be multiple configurations such as $Fconfig_5^6$ and $Fconfig_6^6$ which indicate two different configurations for the same number of FACTS devices.

18. $sumoverload_k$ represents the total number of overloaded lines in the grid for configuration $Fconfig_k^{nfacts}$, considering all possible contingencies.

4.1.3. Identification of Overloaded Lines. The process of identifying the overloaded lines involves performing a contingency analysis by iteratively deactivating a single line (causing an SLC) and for each such SLC determining which lines are overloaded. In order to simulate the SLCs, the lines in *Line* are first populated into *Vline* and alterations are done in *Vline*. This preserves the original grid configuration for the lines in *Line*. The process of creating an SLC is explained in Algorithm 25.

Algorithm 25 Deactivate: A function for deactivating a line in the grid

```
Remove Vline_ij from Vline
Alter BusConfig and LineConfig to reflect the change in line configuration
//Detailed explanation of the procedure for altering the configurations
//is provided in [17]
```

In order to determine the lines in the grid that are usually the most overloaded, it is necessary to cause one SLC at a time and determine the power flow across the grid. Thus, by deactivating one line at a time and determining the power flow across the grid it is possible to identify those lines that are overloaded during that contingency. Based on this information, the number of times a line is overloaded for all SLCs and the total amount by which a line is overloaded for all SLCs can be determined. Computation of power flow across the lines in a grid is accomplished using a MaxFlow algorithm discussed in [17].

The MaxFlow algorithm uses the pre-computed grid configuration represented by Bus, $VLine$, $BusConfig$ and $LineConfig$. A detailed explanation of the MaxFlow algorithm can be found in [17]. The procedure for determining the number of times each line in the grid is overloaded and the maximum amount by which they are loaded is provided in Algorithm 26.

Algorithm 26 ComputeOverloads: An algorithm to determine overloaded lines

```
for i ← 1 to n_Bus do
  for j ← 1 to n_Bus do
    numoverload_ij ← 0
    overload_ij ← 0
  end for
end for
for i ← 1 to n_Bus do
  for j ← 1 to n_Bus do
    if Vline_ij ∈ Vline then
      Simulate a contingency for the failure of Vline_ij
      Deactivate( i, j )
      Use MaxFlow algorithm to compute the power flow across G using Bus, Vline,
      BusConfig and LineConfig
      for k ← 1 to n_Bus do
        for m ← 1 to n_Bus do
          if |flow_km| > cap_km then
            //Increment the number of times Vline_km is overloaded
            numoverload_km ← numoverload_km + 1
            //Aggregate the amount by which Vline_km is overloaded
            overload_km ← overload_km + (|flow_km| − cap_km)
          end if
        end for
      end for
    end if
  end for
end for
```

4.1.4. Metrics for Evaluation of FACTS Device Configuration. The quality of FACTS device configurations is evaluated on the basis of the reduction in the number of overloaded lines in the grid. When FACTS devices are placed on the grid, the number of overloaded lines is expected to be minimized, although that might not always be the case. The benchmark value for this criterion is set as the total number of overloaded lines, n_{base}, determined without the placement of any FACTS devices on the grid. The procedure for computing n_{base} is given in Algorithm 27.

Algorithm 27 ComputeNbase: An algorithm to determine the number of overloaded lines for the initial configuration

$n_{base} \leftarrow 0$
Call ComputeOverloads
for $i \leftarrow 1$ to n_{Bus} **do**
  for $j \leftarrow 1$ to n_{Bus} **do**
    $n_{base} \leftarrow n_{base} + numoverload_{ij}$
  end for
end for

4.1.5. Placement of a FACTS Device on the Grid. The placement of a FACTS device on a line involves efficiently controlling the FACTS device settings. In order to simulate the configuration of a FACTS device's settings, $BusConfig$ and $LineConfig$ should be altered accordingly. Extensive information on the process of configuring FACTS device settings is provided in [17]. This research uses a ConfigureFACTS function that is designed based on the process for configuring FACTS devices explained in [17]. Algorithm 28 describes the procedure for placing a FACTS device on a line.

Algorithm 28 ConfigureFACTS: An algorithm for placing a FACTS device on a line

//INPUTS: $BusConfig$, $LineConfig$, fac_{ij}
Place a FACTS device on $Vline_{ij}$
Alter $BusConfig$ and $LineConfig$ to reflect the change in grid configuration

4.2. CLONALG DESIGN

Clonalg is a Clonal EA designed for the determination of optimal FACTS device configurations. In this context, Clonalg was specially designed to optimize a fitness function that assigns a fitness value to any FACTS device configuration, $Fconfig_k^{nfacts}$. The main differences between Clonalg and IDSClonalg (Algorithm 21) are:

1. *It does not use an antigen set:* This is because there are no specific antigens to be recognized and remembered. Rather, the requirement is to evaluate a given set of FACTS device configurations using a fitness function and retain the best ones.

2. *It does not use a memory antibody set:* Since there are no specific antigens to be recognized there is not a need for remembering antibodies. Rather, the FACTS

device configurations with superior fitness are preserved through a fitness-based competition.

The following notations are used in the description of IDSClonalg:

1. M_p represents a solution set. The members of this set represent FACTS device configurations. Each member of this set is represented as:

$$M_p^i = \{Fconfig_i^{nfacts}, sumoverload_i, f_i\} \quad (4.2)$$

   where f_i represents the fitness value assigned to M_p^i by a fitness function. All members of this set must possess the same number of FACTS devices, $nfacts$.

2. M_c represents a clone set, which is a multi-set containing multiple copies of members in M_p.

3. n_p represents the cardinality of M_p.

4. n_c represents the cardinality of M_c.

5. n_c^i represents the number of clones of member i in M_p that are to be generated in M_c.

6. M_c^x represents a member x in the clone set.

7. $FACTSFitness(M, n_{facts})$ represents a function that assigns a fitness value to all the members of M. The fitness value is computed by subtracting $sumoverload_i$ from n_{base}. The procedure for computing the fitness value is provided in Algorithm 29.

8. $FACTSCreateClones(n, \gamma)$ represents a function that populates M_c with clones created from the first n members of M_p. The number of clones created for each of the first n members is computed using Equation 2.1. Algorithm 30 explains the process for creating the clones.

9. $FACTSHyperMutate(\tau)$ represents a hyper-mutation function. It alters the FACTS device configuration by changing the position of the FACTS devices i.e., by changing the lines on which the FACTS devices are placed, and the process is fitness-based. The higher the fitness of a member M_c^i, the higher the probability that the FACTS devices will be placed on lines closer to their location (i.e., on lines that connect buses in the immediate neighborhood). The neighborhood is defined on the basis of the grid layout shown in Figure 4.1. For example, considering the line that connects

Algorithm 29 FACTSFitness: A function for computing the fitness value of a FACTS device configuration

//INPUTS: M, n_{facts}
for $i \leftarrow 1$ to n_M **do**
  //Iterate through all the members of M
  $sumoverload_i \leftarrow 0$
  for $j \leftarrow 1$ to n_{facts} **do**
    Call ConfigureFACTS($BusConfig$, $LineConfig$, $Fconfig_i^n(j)$)
  end for
  Call ComputeOverloads
  for $j \leftarrow 1$ to n_{Bus} **do**
    for $k \leftarrow 1$ to n_{Bus} **do**
      $sumoverload_i \leftarrow sumoverload_i + numoverload_{jk}$
    end for
  end for
  $f_i \leftarrow n_{base} - sumoverload_i$
end for

Algorithm 30 FACTSCreateClones: A function for creating multiple clones of members in M_n

//INPUTS: γ
$k \leftarrow 0$
for $i \leftarrow 1$ to n_p **do**
  $numclones \leftarrow \lceil(\gamma \cdot n_p)/i\rceil$
  for $j \leftarrow 1$ to $numclones$ **do**
    $k \leftarrow k + 1$
    $M_c^k \leftarrow M_p^i$
  end for
end for
$n_c \leftarrow k$

buses numbered "44" and "45", a line that connects buses "46" and "47", and a line that connects buses "47" and "49" may be among those that are considered to be in the immediate neighborhood. Also, a line that connects buses "17" and "18" may be among those lines that are considered to be far away. The lower the fitness of a member M_p^i, the higher the probability that the FACTS devices will be placed on lines farther (lines that connect buses that are far away) from their current location. For this purpose, the members of M_c are classified into different groups based on their fitness. An upper limit that constrains the farthest line to which a FACTS device can be moved from its current location is assigned to each group. In order to create such groups, the members of M_c^i are sorted in descending order of their

fitness. They are then classified into groups of τ members each. The groups are represented as $\{G_1, \cdots, G_\eta\}$ where:

$$\eta = \left\lceil \frac{n_c}{\tau} \right\rceil \tag{4.3}$$

For example, if $n_c = 53$ and $\tau = 10$, then $\eta = 6$. Group G_1 holds τ members with the highest fitness and group G_η holds τ or fewer members (not zero) with the lowest fitness. The upper bound, U_i for each group is then computed as:

$$U_i = \left\lfloor \frac{n_{Vline}}{2 * \tau} \cdot i \right\rfloor \tag{4.4}$$

where $i = 1$ to η. For example, if $n_{Vline} = 118$ and $\tau = 10$, then $U_1 = 6$, $U_2 = 12$ and so on. Thus, the upper bound increases with the group number. This ensures that the high fitness members are altered such that their FACTS devices are moved to relatively closer locations, if mutated, compared to the low fitness ones. Algorithm 31 precisely describes the hyper-mutation process and it uses Algorithm 32 and Algorithm 33 for its sub-computations.

The value of τ combined with γ, the multiplication constant used in determining the number of clones of each member in M_p, provides for interesting variations to experiment. For example, if $\gamma = 0.5$, $\tau = 20$, and $n_p = 50$, then group G_1 would have twenty members that are clones of M_p^1 that use U_1, and G_2 would have five members that are clones of M_p^1 that use U_2 and five members of M_p^2 which use U_2. Thus, it is possible to expose the clones to different ranges of mutations.

10. $RandInit(M_p)$ represents a function that uniform randomly initializes the members of M_p. Algorithm 34 explains the process.

A description of Clonalg using the components discussed previously is provided in Algorithm 35.

4.3. EA DESIGN

An EA was designed to determine optimal placements for a given number of FACTS devices, $nfacts$. The primary purpose of designing this algorithm was to compare its

performance with that of Clonalg. The following notations are used in the description of the EA:

1. M_p represents a parent solution set. The members of this set represent FACTS device configurations. Each member of this set is represented using the format described in Equation 4.2
2. M_o represents an offspring set that contains offspring created from members in M_p. The members of this set possess the same format as that of M_p.
3. n_p represents the cardinality of M_p.
4. n_o represents the cardinality of M_o.
5. $Select(parent1, parent2)$ represents a function that performs proportional selection of parents for reproduction. The parents are selected from M_p whose members are sorted in descending order of their fitness. M_p is partitioned into four bins with the first 40 percent of the members allocated to bin 1, the next 30 percent to bin 2, the next 20 percent to bin 3 and the last 10 percent to bin 4. Algorithm 36 explains the process of proportional selection of the parents using the four bins.
6. $Fitness(M, nfacts)$ represents the fitness function. Algorithm 29 describes the function in detail.
7. $CrossOver(parent1, parent2, nfacts)$ represents a function that implements the cross-over operation. Algorithm 37 explains the cross-over procedure designed for use in the EA.
8. $EAmutate(\rho)$ represents a function that implements the mutation operation. Unlike hyper-mutation in Clonalg, the mutation operation of an EA is much simpler in the sense that all the members of M_o are mutated similarly, and there is only one upper bound value *upper* which takes a user-defined value in the range $[1, 118]$. Algorithm 38 explains the mutation process.

The EA design based on the notations previously described is provided in Algorithm 39.

4.4. GREEDY ALGORITHMS

Two greedy algorithms were also designed to evaluate the performance of Clonalg. These greedy algorithms try to break-down a problem into a series of sub-problems and try to find the best solution for each of those sub-problems. This method may result in sub-optimal solutions because they suffer from the limitation that the best solutions to the sub-problems may not yield an optimal solution for the problem.

4.4.1. Greedy Algorithm based on Number of Overloaded Lines.

This greedy algorithm, alternatively referred to as count greedy algorithm, breaks down the problem of placing $nfacts$ FACTS devices in the grid into $nfacts$ sub-problems of placing one FACTS device on an ideal location in the grid. The ideal location for placing the FACTS device is determined by identifying a line in the grid that is overloaded the most number of times, for all the SLCs. Algorithm 40 explains the greedy algorithm design. The algorithm computes the value n_{base} for the initial configuration (without any FACTS devices). It also determines the line that is overloaded the most number of times, aggregated over all the SLCs, for the initial configuration and places the first FACTS device on that line. The power flow in the grid is again calculated and the line that is overloaded the most number of times is identified. The Second FACTS device is then placed on that line. This process is repeated until all the FACTS devices are placed on the grid.

4.4.2. Greedy Algorithm based on Amount of Line Overload.

This greedy algorithm, alternatively referred to as amount greedy algorithm, breaks down the problem of placing $nfacts$ FACTS devices in the grid into $nfacts$ sub-problems of placing one FACTS device on an ideal location in the grid. The ideal location for the placement of the FACTS device is determined by identifying a line in the grid that is overloaded by a maximum amount for all the SLCs. The maximum amount of overload is determined by aggregating the difference between the actual power flow ($flow_{ij}$) and rated capacity (cap_{ij}) of each line over all the SLCs. Algorithm 41 explains the greedy algorithm design. The algorithm computes the value n_{base} for the initial configuration (without any FACTS devices). It also determines the line that is overloaded by the maximum amount, aggregated over all the SLCs, for the initial configuration and places the first FACTS device on that line. The power flow in the grid is again calculated and the line that is overloaded by the maximum amount is identified. The second FACTS device is then placed on that line. This process is repeated until all the FACTS devices are placed on the grid.

4.5. EXPERIMENTS

The grid as already explained in Section 4.1.1., possesses 118 buses with at most one line connecting any two buses. Table 4.1 lists the parameters and their values corresponding to the grid setting. The total number of lines that were considered feasible for performing the actual experiments, namely causing SLCs and placing FACTS devices, is provided by n_{Vline}.

Table 4.1. Parameter Values for Grid Configuration

Parameters	Values
n_{Bus}	118
n_{Line}	179
n_{Vline}	169

The four algorithms, namely Clonalg, EA, count greedy algorithm and amount greedy algorithm were executed to determine the best possible configurations for the number of FACTS devices varying from one to ten. Clonalg was executed for $nfacts$ values varying from one to ten, along with the parameter values listed in Table 4.2. From the values of n_p and γ, the value of n_c can be computed to be 194 using Equation 2.2. It was important for comparison purposes, that the EA was also run with an equivalent number of members. Table 4.3 lists the parameters and their values that were used in the execution of the EA. It can be seen that the number of offspring to be generated was specified as 194 to compare the performance of EA with Clonalg. There were no specific paramenters associated with count greedy algorithm and amount greedy algorithm apart from $nfacts$. These two algorithms were also executed for $nfacts$ values varying from one to ten.

4.6. RESULTS AND DISCUSSION

The algorithms discussed previously were executed against a grid configuration model based on the grid shown in Figure 4.1. The algorithms were executed to determine the optimal placement positions for a number of FACTS devices ranging from one to ten. For simplicity of representation, Table 4.4 associates labels with various lines

Table 4.2. Parameter Values for Clonalg

Parameters	Values
n_p	100
γ	0.5
τ	2
numgenerations	50

Table 4.3. Parameter Values for EA

Parameters	Values
n_p	100
n_o	194
ρ	2
numgenerations	50

that were identified by the optimization algorithms as optimal locations for placing the FACTS devices. First, the value of n_{base} was computed using Algorithm 27 and it was determined to be 463.

Table 4.5, Table 4.6, Table 4.7 and Table 4.8 show the optimal FACTS device configurations obtained using Clonalg, EA, greedy algorithm based on number of line overloads and greedy algorithm based on amount of line overload, respectively. The $sumoverload_i$ values are also listed for each of those configurations.

Figure 4.2 shows the best results obtained for different number of FACTS devices placed on the power grid by using the algorithms discussed above. It can be seen that Clonalg performs the best in determining optimal FACTS device configurations. The greedy algorithms are inconsistent, since they attempt to find the best possible placement for a particular instance and hence tend to generate worse placements when additional FACTS devices are introduced. The evolutionary algorithm provides the second best performance and actually outperforms Clonalg for a smaller number of FACTS devices. However, as the number of FACTS devices increase the performance drops. Thus, it is inferred that the evolutionary algorithm is not as scalable as Clonalg.

The Clonalg algorithm and EA determine the critical lines for placing the FACTS devices by first placing the FACTS devices on the lines and evaluating their quality. This is a "passive" method for determining optimal FACTS device configurations. A unique

Table 4.4. Labels for FACTS Device Placements on the Grid

FACTS Placement		Label	FACTS Placement		Label
17	113	A	76	118	O
56	58	B	23	25	P
114	115	C	17	30	Q
60	62	D	49	51	R
61	64	E	75	118	S
54	56	F	45	46	T
54	55	G	25	26	U
35	36	H	75	77	V
105	106	I	19	20	W
91	92	J	54	59	X
32	113	K	114	115	Y
23	24	L	56	58	Z
24	72	M	90	91	AA
17	31	N	3	5	AB
89	90	AC	49	66	AD
74	75	AE	42	49	AF
52	53	AG	92	94	AH
37	39	AI	63	64	AJ
12	14	AK	64	65	AL
59	60	AM	17	18	AN
21	22	AO	45	46	AP
5	8	AQ	75	118	AR
78	79	AS	15	17	AT

feature of this method is that not a lot of grid specific information or methods to identify critical lines are needed to determine optimal FACTS device configurations. On the contrary, the greedy algorithms use an "active" method for determining optimal FACTS device configurations. In this method, the critical lines are identified first by using some specific criteria and a FACTS device configuration is then evaluated by placing FACTS devices on those lines. Therefore, this method requires careful consideration of the criteria to be used for identifying critical lines.

Table 4.5. FACTS Device Placements obtained using Clonalg

No. of FACTS	FACTS Placement										Fitness	Over-loads
	1	2	3	4	5	6	7	8	9	10		
0											0.4463	143
1	P										0.40885	131
2	A	J									0.38388	123
3	P	J	Q								0.35579	114
4	R	A	S	J							0.34019	109
5	T	A	S	U	P						0.31834	102
6	V	Q	W	J	F	R					0.29025	93
7	X	Y	Z	W	V	J	Q				0.26216	84
8	X	Y	Z	A	S	AA	Q	W			0.24344	78
9	X	Y	Z	A	P	AA	Q	W	S		0.2278	73
10	X	Y	Z	A	P	AA	Q	W	S	AB	0.2122	68

Table 4.6. FACTS Device Placements obtained using the Evolutionary Algorithm

No. of FACTS	FACTS Placement										Fit-ness	Over-loads
	1	2	3	4	5	6	7	8	9	10		
0											0.446	143
1	P										0.409	131
2	A	W									0.378	121
3	B	Q	W								0.346	111
4	A	W	AC	O							0.337	108
5	AD	P	AE	J	AF						0.331	106
6	V	Q	W	J	AD	AG					0.299	96
7	X	C	Z	P	V	AH	Q				0.287	92
8	AA	U	AI	A	O	AJ	AK	AL			0.259	83
9	S	AM	AA	AL	AN	A	D	AO	AP		0.259	83
10	AQ	AO	S	X	I	AS	AT	A	J	AP	0.247	79

Algorithm 31 FACTSHyperMutate: A function for performing hyper-mutation on a FACTS device configuration

```
//INPUTS: τ
η ← ⌈n_c/τ⌉
for i ← 1 to η do
  U_i ← ⌊(n_Vline/(2 * τ)) · i⌋
end for
Sort the members of M_c in descending order of their fitness values
k ← 1
for i ← 1 to η do
  for j ← 1 to τ do
    if k ≤ n_c then
      //Organize the members of M_p into fitness-based groups
      G_i ← G_i ∪ {M_c^k}
      k ← k + 1
    end if
  end for
end for
for i ← 1 to η do
  for j ← 1 to τ do
    for k ← 1 to nfacts do
      if random() > 0.5 then
        //The random() function returns a uniform random value between "0"
        //and "1"
        //Hyper-mutate a FACTS device position with 50 percent probability
        Remove fac_{α_k β_k} from Fconfig_j^{nfacts}
        if random() > 0.5 then
          {α, β} ← ForwardDisplace(U_i, α_k, β_k)
        else
          {α, β} ← BackwardDisplace(U_i, α_k, β_k)
        end if
        if Vline_{αβ} ∈ Vline then
          //Check if hyper-mutation yields a valid line for the FACTS device
          //to be placed
          Insert fac_{αβ} into Fconfig_j^{nfacts}
        else
          Insert fac_{α_k β_k} into Fconfig_j^{nfacts}
        end if
      end if
    end for
  end for
end for
```

Algorithm 32 ForwardDisplace: A function for moving the bus numbers of a FACTS placement forward

//INPUTS: U_i, α_k, β_k
//OUTPUTS: α, β
$temp1 \leftarrow random(1, U_i)$
//The function random(a,b) returns a value in the range $[a, b]$
//Displace the FACTS device to a random position within a range specified by U_i
if $\alpha_k + temp \leq n_{Bus}$ **then**
  $\alpha \leftarrow \alpha_k + temp1$
else
  $\alpha \leftarrow \alpha_k - temp1$
end if
$temp2 \leftarrow random(1, U_i)$
if $\beta + temp \leq n_{Bus}$ **then**
  $\beta \leftarrow \beta_k + temp2$
else
  $\beta \leftarrow \beta_k - temp2$
end if

Algorithm 33 BackwardDisplace: A function for moving the bus numbers of a FACTS placement backward

//INPUTS: U_i, α_k, β_k
//OUTPUTS: α, β
$temp1 \leftarrow random(1, U_i)$
if $\alpha_k - temp > 0$ **then**
  $\alpha \leftarrow \alpha_k - temp1$
else
  $\alpha \leftarrow \alpha_k + temp1$
end if
$temp2 \leftarrow random(1, U_i)$
if $\beta_k - temp > 0$ **then**
  $\beta \leftarrow \beta_k - temp2$
else
  $\beta \leftarrow \beta_k + temp2$
end if

Algorithm 34 RandInit: A function for uniform randomly initializing the members of M_p

for $i \leftarrow 1$ to n_p **do**
  $Fconfig_i^n facts \leftarrow \{\}$
  $sumoverload \leftarrow 0$
  $f_i \leftarrow 0$
  for $j \leftarrow 1$ to $nfacts$ **do**
    $found \leftarrow false$
    while $found = false$ **do**
      $\alpha \leftarrow random(1, n_{Bus})$
      $\alpha \leftarrow random(1, n_{Bus})$
      if $Vline_{\alpha\beta} \in Vline$ **then**
        if $fac_{\alpha\beta} \notin Fconfig_i^{nfacts}$ **then**
          $Fconfig_i^{nfacts} \leftarrow Fconfig_i^{nfacts} \cup fac_{\alpha\beta}$
          $found \leftarrow true$
        end if
      end if
    end while
  end for
end for

Algorithm 35 Clonalg: An algorithm for determining optimal FACTS device configuration

//INPUTS: $numgenerations$, $nfacts$, n_p, τ, γ
Call $RandInit(M_p)$
Call $FACTSFitness(M_p, nfacts)$
//Evaluate the fitness of all the randomly intialized members of M_p
Call $Sort(M_p)$
//Arrange the members of M_p in descending order of fitness
for $iter \leftarrow 1$ to $numgenerations$ **do**
  //Loop for a specified number of generations
  Call $FACTSCreateClones(\gamma)$
  //Perform clonal expansion
  Call $FACTSHyperMutate(\tau)$
  //Perform hyper-mutation
  Call $FACTSFitness(M_c, nfacts)$
  //Evaluate the fitness of all newly created clones
  $M_{temp} \leftarrow M_p \cup M_c$
  Call $Sort(M_{temp})$
  //Set up a competition between the members of M_p and M_c
  //and insert n_p of the highest fitness members in to M_p
  for $j = 1$ to n_p **do**
    $M_p \leftarrow M_{temp}$
  end for
end for

Algorithm 36 Select: A function for performing proportional selection

```
//OUTPUTS: parent1, parent2
Part ← {40, 30, 20, 10}
for i ← 1 to 4 do
  bin_i ← n_p · (Part_i/10)
end for
chance ← random()
if chance ≤ 0.4 then
  x ← random(1, bin_1)
else if chance ≤ 0.7 then
  x ← random(bin_1, bin_2)
else if chance ≤ 0.9 then
  x ← random(bin_2, bin_3)
else
  x ← random(bin_3, bin_4)
end if
found ← false
while found = false do
  chance ← random()
  if chance ≤ 0.4 then
    y ← random(1, bin_1)
  else if chance ≤ 0.7 then
    y ← random(bin_1, bin_2)
  else if chance ≤ 0.9 then
    y ← random(bin_2, bin_3)
  else
    y ← random(bin_3, bin_4)
  end if
  if x ≠ y then
    found ← true
  end if
end while
parent1 ← M_x
parent2 ← M_y
```

Algorithm 37 CrossOver: A function for performing cross-over

```
//INPUTS: parent1,parent2,nfacts
//OUTPUTS: offspring1, offspring2
offspring1 ← parent1
offspring2 ← parent2
midpt ← ⌈nfacts/2⌉
for k ← 1 to midpt do
  Swap fac_{α_k,β_k} in Fconfig^{nfacts}_{offspring1} and fac_{γ_k,δ_k} in Fconfig^{n}_{offspring2}facts
end for
```

Algorithm 38 EAmutate: A function for performing mutation in an EA

//INPUT: ρ
for $i \leftarrow 1$ to n_o **do**
  for $j \leftarrow 1$ to $nfacts$ **do**
    if $random() > 0.5$ **then**
      //Mutate a FACTS device position with 50 percent probability
      Remove $fac_{\alpha_j\beta_j}$ from $Fconfig_i^{nfacts}$
      if $random() > 0.5$ **then**
        $\{\alpha, \beta\} \leftarrow ForwardDisplace(\rho, \alpha_k, \beta_k)$
      else
        $\{\alpha, \beta\} \leftarrow BackwardDisplace(\rho, \alpha_k, \beta_k)$
      end if
      if $Vline_{\alpha\beta} \in Vline$ **then**
        //Check if mutation yields a valid line for the FACTS device
        //to be placed
        Insert $fac_{\alpha\beta}$ into $Fconfig_j^{nfacts}$
      else
        Insert $fac_{\alpha_k\beta_k}$ into $Fconfig_j^{nfacts}$
      end if
    end if
  end for
end for

Table 4.7. FACTS Device Placements obtained using Greedy Algorithm based on Number of Overloaded Lines

No. of	FACTS Placement										*Overloads*
FACTS	1	2	3	4	5	6	7	8	9	10	
0											143
1	J										135
2	J	F									130
3	J	F	K								130
4	J	F	K	A							117
5	J	F	K	A	G						114
6	J	F	K	A	G	Z					106
7	J	F	K	A	G	Z	L				228
8	J	F	K	A	G	Z	L	M			94
9	J	F	K	A	G	Z	L	M	N		86
10	J	F	K	A	G	Z	L	M	N	O	82

Algorithm 39 EA: An algorithm for determining optimal FACTS device configuration

```
//INPUTS: numgenerations, nfacts, n_p, n_o, ρ
Call RandInit(M_p)
Call FACTSFitness(M_p)
//Evaluate the fitness of all the randomly intialized members of M_p
for iter ← 1 to numgenerations do
  //Loop for a specified number of generations
  j ← 1
  for i ← 1 to (n_o/2) do
    //Create n_o offspring
    {parent1, parent2} ← Select()
    {offspring1, offspring2} ← CrossOver(parent1, parent2, nfacts)
    M_o^j ← offspring1
    M_o^{j+1} ← offspring2
    j ← j + 2
  end for
  Call EAmutate(ρ)
  //Perform mutation
  Call FACTSFitness(M_o)
  //Evaluate the fitness of offspring
  M_temp ← M_p ∪ M_o
  Call Sort(M_temp)
  //Set up a competition between the members of M_p and M_c
  //and insert n_p of the highest fitness members in to M_p
  for j ← 1 to n_p do
    M_p ← M_temp
  end for
end for
```

Table 4.8. FACTS Device Placements obtained using Greedy Algorithm based on Amount of Overload

No. of FACTS	FACTS Placement										*Overloads*
	1	2	3	4	5	6	7	8	9	10	
0											143
1	A										131
2	A	B									124
3	A	B	C								119
4	A	B	C	D							114
5	A	B	C	D	E						116
6	A	B	C	D	E	F					114
7	A	B	C	D	E	F	G				109
8	A	B	C	D	E	F	G	H			108
9	A	B	C	D	E	F	G	H	I		104
10	A	B	C	D	E	F	G	H	I	J	96

Algorithm 40 Greedy Algorithm1: An algorithm for determining optimal FACTS device configurations based on number of overloaded lines

```
//INPUTS: nfacts
Call ComputeOverloads()
Call ComputeNbase()
//Determine the value of n_base
for i ← 1 to nfacts do
  Max ← 0
  for j ← 1 to n_Bus do
    for k ← 1 to n_Bus do
      if numoverload_jk > Max then
        //Determine the line that is overload the most number of times
        Max ← numoverload_jk
        α ← j
        β ← k
      end if
    end for
  end for
  Call ConfigureFACTS(BusConfig, LineConfig, fac_α,β)
  //Place a FACTS device on the line that is overload the most number of times
  Call ComputeOverloads()
end for
sumoverload ← 0
for i ← 1 to n_Bus do
  for j ← 1 to n_Bus do
    sumoverload ← sumoverload + numoverload_ij
    //Determine the total number of overloaded lines obtained when using the
    //FACTS configuration created using this algorithm
  end for
end for
```

Algorithm 41 Greedy Algorithm2: An algorithm for determining optimal FACTS device configurations based on amount of line overload

//INPUTS: $nfacts$
Call $ComputeOverloads()$
Call $ComputeNbase()$
//Determine the value of n_{base}
for $i \leftarrow 1$ to $nfacts$ **do**
  $Max \leftarrow 0$
  for $j \leftarrow 1$ to n_{Bus} **do**
    for $k \leftarrow 1$ to n_{Bus} **do**
      if $numoverload_{jk} > Max$ **then**
        //Determine the line that is overload by the maximum amount
        $Max \leftarrow overload_{jk}$
        $\alpha \leftarrow j$
        $\beta \leftarrow k$
      end if
    end for
  end for
  Call $ConfigureFACTS(BusConfig, LineConfig, fac_{\alpha,\beta})$
  //Place a FACTS device on the line that is overload by the maximum amount
  Call $ComputeOverloads()$
end for
$sumoverload \leftarrow 0$
for $i \leftarrow 1$ to n_{Bus} **do**
  for $j \leftarrow 1$ to n_{Bus} **do**
    $sumoverload \leftarrow sumoverload + numoverload_{ij}$
    //Determine the total number of overloaded lines obtained when using the
    //FACTS configuration created using this algorithm
  end for
end for

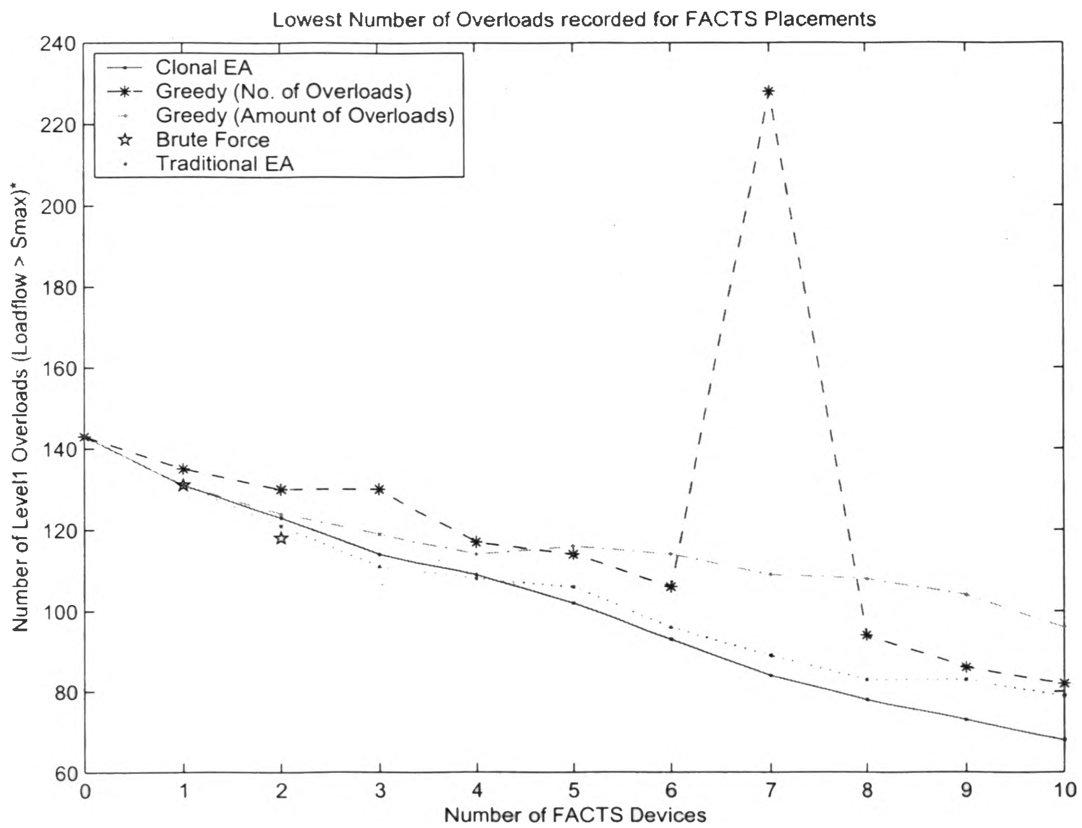


Figure 4.2. Comparison of Results obtained using Various Algorithms

5. CONCLUSION

The main focus of this research was the exploration of various bio-inspired algorithms and their application to different real-world problems. During the course of this research, some theoretical concepts were examined and analyzed. New algorithms were designed for the application of clonal selection principles to solve pattern recognition and function optimization problems.

The IDSClonalg design and a theoretical system for analyzing the methods for generating intrusion detectors were the main features of the research on intrusion detection. The research on the theoretical system for generating detectors revealed certain important aspects of intrusion detection. The literature reviewed has shown that a significant amount of research work has been dedicated to exploring the ND scheme but not the PD scheme. However, it was determined from the experiments conducted that for certain systems, the PD scheme significantly outperforms the ND scheme. This motivated a comparison of the two schemes by utilizing them to detect intrusions in a real-world data set. Thus, the ND and PD schemes were implemented in IDSClonalg for performing intrusion detection in the KDD cup 1999 data set. Again, it was found that for a system with a highly cohesive definition of self, the PD scheme comprehensively outperforms the ND scheme. Further, the IDSClonalg design illustrates a highly simplified approach for generating detectors compared to the complex greedy algorithms discussed in Section 3.2.

The Clonalg algorithm developed for determining the optimal FACTS device configurations was the most successful aspect of this research, as it has enabled researchers at UMR to benchmark their results for developing and evaluating algorithms that compute optimal FACTS device configurations. Previously, only brute force approaches and greedy algorithms were available for such benchmarking, with bruteforcing involving prohibitively long time spans for generating solutions that involved a large number of FACTS devices. The success of Clonalg could well be the beginning of a whole new approach of using clonal selection principles for solving problems in critical infrastructure protection.

6. FUTURE WORK

This research has revealed a lot of avenues for exploration. From the applications considered, to the algorithms used and theoretical concepts discussed, there are many opportunities for further improvement and modification. The self/non-self theoretical model, examined in Section 3., was built on the basis of a binary universe. The model analysis would be more realistic if real data were used and if the matching functions involved tolerances, instead of exact matches. This would require a significant amount of work in preparing the data sets and constructing the AIS model that will be used for performing experiments.

The size of detector sets is to be carefully chosen and it depends on a trade-off between speed, memory and accuracy of the detection system. A significant amount of research would be required to develop broad guidelines for adjusting the detection system parameters based on specific requirements.

Data handling is an important aspect of classification. It is highly recomended that further investigations be made into the pre-processing of data as this will serve to improve the performance of the classification systems. The IDSClonalg implemented to evaluate the *KDD Cup 1999* data set is designed to handle offline labeled data. It could be augmented to work with unlabeled real-time data.

The AIS models built on the basis of the present understanding of the immune system are not comprehensive and the results of application of biological techniques to solve network intrusion detection problems have only been moderately successful. There are several issues that need to be addressed including:

1. Concurrent events involved in an attack are not clearly understood and a considerable amount of research is needed in this direction.

2. There are fundamental differences between the human immune system and computer security systems. A comparative study, distinguishing and discriminating the various aspects of the two systems, needs to be performed. This will provide a better perspective regarding the effectiveness of a mapping from the biological domain to that of computers.

The FACTS placement problem is another area with enormous research opportunities. Parameter optimization, multiple line contingencies, contingencies other than line contingencies, and control of FACTS device settings are some of the interesting areas

for exploration. Further, the use of clonal selection principles for solving some of these problems is an exciting prospect.

BIBLIOGRAPHY

[1] L. N. de Castro and F. J. V. Zuben, "Learning and optimization using the clonal selection principle.," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 239–251, June 2002.

[2] S. Forrest, R. E. Smith, B. Javornik, and A. S. Perelson, "Using genetic algorithms to explore pattern recognition in the immune system," *Evolutionary Computation*, vol. 1, no. 3, pp. 191–211, 1993.

[3] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing.* Springer, 2003.

[4] S. Forrest and S. Hofmeyr, "Engineering an immune sytem," *Graft*, vol. 4:5, pp. 5–9, 2001.

[5] S. Forrest, P. D'haeseleer, and P. Helman, "A distributed approach to anomaly detection." This is listed without publication information on Forrest's website, `http://www.cs.unm.edu/~forrest/papers.html`, 1997.

[6] A. Somayaji, S. Hofmeyr, and S. Forrest, "Principles of a computer immune system," in *Proceedings of New Security Paradigms Workshop*, pp. 75–82, 1997.

[7] P. Harmer, P. Williams, G. Gunsch, and G. Lamont, "An artificial immune system architecture for computer security applications.," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 252–280, June 2002.

[8] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, (Los Alamitos, CA), IEEE Computer Society Press, 1994.

[9] P. D'haeseleer, "Further efficient algorithms for generating antibody strings," Technical Report CS95-03, The University of New Mexico, Albuquerque, NM, 1995.

[10] P. D'haeseleer, "A change detection method inspired by the immune system: Theory, algorithms and techniques," Technical Report CS95-06, The University of New Mexico, Albuquerque, NM, 1995.

[11] P. D'Haeseleer, S. Forrest, and P. Helman, "An immunological approach to change detection: Algorithms, analysis and implications," in *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1996.

[12] J. Kim and P. Bentley, "The artificial immune model for network intrusion detection," in *Proceedings of the 7th European Conference on Intelligent Techniques and Soft Computing (EUFIT'99)*, (Aachen, Germany), 13-19 Sept. 1999.

[13] J. Kim and P. J. Bentley, "Towards an artificial immune system for network intrusion detection: An investigation of clonal selection with a negative selection operator," in *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, (COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea), pp. 1244–1252, IEEE Press, 27-30 May 2001.

[14] J. Kim and P. J. Bentley, "Immune memory in the dynamic clonal selection algorithm," in *Proceedings of the 1st International Conference on Artificial Immune Systems (ICARIS)* (J. Timmis and P. J. Bentley, eds.), (University of Kent at Canterbury), pp. 59–67, University of Kent at Canterbury Printing Unit, Sept. 2002.

[15] F. M. Ham and I. Kostanic, *Principles of Neurocomputing for Science and Engineering.* Tata McGraw-Hill, 2001.

[16] A. Armbruster, B. McMillin, and M. L. Crow, "Controlling power flow using facts devices and the max flow algorithm," in *Proceedings of the International Conference on Power Systems and Control,* (Abuja, Nigeria), 2002.

[17] A. Armbruster, B. McMillin, and M. L. Crow, "The maximum flow algorithm applied to the placement and distributed steady-state control of facts devices." This paper is submitted to the 2005 North American Power Symposium and to the IEEE Transactions on Power Systems, `http://web.umr.edu/~ff/Power/papers.htm`, 2005.

VITA

Kasthurirangan Parthasarathy was born in Chennai, India, on October 19, 1980. In December 2000, he received his Honors Diploma in Network Centered Computing from the National Institute of Information Technology. In May 2002, he received his bachelor's degree in Computer Science and Engineering from Bharathidasan University, Thiruchirapalli, India. Kasthurirangan Parthasarathy worked on his M.S in Computer Science at the University of Missouri - Rolla from Fall 2002 to Summer 2005.

Kasthurirangan Parthasarathy has been a member of the International Engineering Consortium since April 2002.