A Machine Learning Approach to Security Improvement in Mobile Communication

By

Hooshang F. Sharif

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2005

UMI Number: 3187420

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

This manuscript has been read and accepted for the Graduate Facility in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

	Professor Michael Conner
Date	Chair of Examining Committee
	Professor Mumtaz Kassir
Date	Executive Officer

Professor Michael Conner

Professor Ibrahim Habib

Professor Tarek N. Saadawi

Professor M. Ümit Uyar

Professor Jizhong Xiao

Professor Mohammad M. Zahran

Dr. Nidal Khrais

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

A Machine Learning Approach to Security Improvement in Mobile Communication

by

Hooshang Sharif

Advisor: Professor Michael Conner

One of the challenges for today's mobile communications engineers is designing networks that are secure and reliable. Over time, these networks have become increasingly complex. There are situations where a network's functionality and reliability must be maintained under particularly adverse circumstances. For example, in a modern combat environment, high jamming noise would necessitate signal transmission at high energies and under severe bandwidth constraints. Accordingly, there needs to be a significant reduction to the system's

allocation of resources, such as bandwidth, that are dedicated to vulnerability and intrusion detection.

Conventional methods of vulnerability detection are often criticized for their predictability as well as excessive need for resource-allocation. Software agents, however, can be implemented to continuously monitor the network and apply the necessary resources in order to maintain the optimal security status.

In this dissertation, we have addressed the vital role of improving security and reducing vulnerability in a wireless network in the presence of the bandwidth limitation that may exist in a combat environment. We introduce methods that allow learning agents to anticipate vulnerabilities that are likely to occur in the network. Acquiring the knowledge for statistically correct anticipation will result in optimized use of bandwidth as well as other resources that are dedicated to security.

Although the main focus of this work is algorithms developed based on reinforcement learning, we have also developed alternative algorithms based on evolutionary computation methods for comparison.

It is the ability to learn the dynamics of a network environment in a continual and adaptive way that prompted us to utilize machine learning ideas as framework for developing our algorithms. We believe, however, that the methods introduced in this work can be applied to all mobile communication environments in general.

Acknowledgments

I wish to express my sincere gratitude to the many people whose support and guidance made this great achievement possible for me.

My advisor and the committee chair, Professor Michael Conner, provided me with his continuing guidance. Throughout my doctoral education years, whether my research reached high milestones or low moments of uncertainty, his words of knowledge and wisdom instilled in me the confidence and the focus that I needed to continue on a positive and constructive path toward successful completion, and for that I am grateful.

I also wish to acknowledge the contributions made by my examination committee members, professors Ibrahim Habib, Jizhong Xiao, and Umit Uyar, which I greatly benefited from since the time of my thesis proposal. Their words of advice, intelligent questions, valuable comments, and supportive ideas helped establish a strong foundation upon which I could lay out my research efforts.

From the early months of my Ph.D. studies, Professor Tarek Saadawi acknowledged my focus and desire in this great endeavor and provided me with the grants and financial support and I am grateful for it. I also wish to express my most sincere appreciations to Professor Jamal Manhassah for his kind words of encouragement and strength.

I must also recognize the researchers and authors whose names appear in the Bibliography. Their years of hard work will continue to make a positive difference in the lives of all of us, as they have been a source of knowledge and inspiration for me.

And finally, as I complete the final words of this dissertation, I wish to thank the people whose love and support will always be the strength and the courage that I need to reach for the highest goals throughout my life: My family and my friends.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern wireless communication systems share such common characteristics as dynamic end-user participation, mobile hosts, system reconfiguration activities, and a large number of host computers, to name a few. In such systems, the process of assessing and correcting network vulnerabilities is continuous and is achieved through a dedicated allocation of network resources [6][14][1]. A particularly unique requirement for vulnerability detection process has arisen in today's combat environments where mobile communication is an integral part of a military structure. Here an intruder will scan the network looking for vulnerabilities to exploit. Once points of vulnerability are discovered, an intruder can plan an attack, penetrate the network, steal resources or plant malicious code, and withdraw.

There are several characteristics that make vulnerability assessment in such environments particularly challenging. Although network resources dedicated to vulnerability detection are generally limited,

the presence of potentially significant and deliberate levels of noise generated by the enemy requires that all communication be carried out under extremely limited bandwidth availability. A widely popular and simple assessment approach is to apply all vulnerability detection tests to all targets at periodic intervals (e.g., every night; once a year during specific assessment exercises) [6]. This approach has a desirable property in that, once completed, any detectable vulnerability is uncovered. Although simple, this approach can have a drawback in that the number of tests to apply may grow to exceed the resources that can be dedicated at the time assessment is performed. It is insensitive to bandwidth and computational constraints, and more importantly for a combat environment, it is predictable. Current vulnerability and intrusion detection systems have been criticized for numerous shortcomings that include efficiency, upgradability, and a lack of a generic structured building methodology [6].

It is the challenging problem of detecting large number of intrusion and vulnerabilities under the limited bandwidth condition that was the motivation for our work in this dissertation. We approached the detection problem with the assumption that only a small fraction of

the bandwidth at a node can be allotted to vulnerability assessment and that it can be carried out at irregular and therefore unpredictable intervals. This assumption gives rise to the condition that only a subset of known vulnerability tests can be applied to the network at any given time. Furthermore we assumed that no knowledge of the network's vulnerability occurrence is known a priori. A successful vulnerability detection system would need to select a subset of a known vulnerability set and apply it to the network. Our problem was to make the selection process as efficient as possible. An efficient system would select a subset of vulnerabilities that would result in the highest probability of detection. Since no prior knowledge of the network's vulnerability occurrence is available, the successful assessment system would seek to ascertain the occurrence patterns in an adaptive and dynamic way.

We concentrated our work on machine learning methods that, through interaction with the network environment, learn the patterns at which vulnerabilities occur. Obtaining this knowledge would then enable us to optimize the vulnerability assessment process. The essence of our work is the development of an algorithm based on reinforcement

learning [37] techniques. We selected reinforcement learning as the basis of our work because such learning systems require no prior knowledge of the environment they seek to understand.

Throughout this thesis we refer to terminologies such as vulnerability and intrusion as any condition in a network node that would leave the node, or a group of nodes, susceptible to unauthorized access. For any known vulnerability, we assume there is an assessment test designed to detect that particular vulnerability. At desired intervals, a central dispatcher node would transmit a particular assessment test, or a collection of tests, to the network. Figure 1.1 illustrates this process.

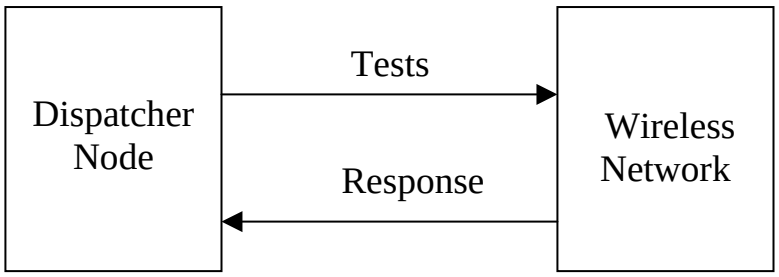

Figure 1.1: Dispatcher-Network Interaction. The dispatcher selects and transmits a vulnerability assessment test, and receives a response from the network. Depending on the prediction, the response is positive or negative, indicating success or failure in vulnerability detection.

Through the use of reinforcement learning methods, the dispatcher is able to ascertain the patterns under which vulnerabilities occur in the network. As depicted in Figure 1.1, the dispatcher obtains this knowledge by interacting with the network.

Although we will not discuss the nature of individual security vulnerabilities in this thesis, we provide a brief description in the following paragraphs for reference. The Security vulnerabilities can be categorized as follows [1]:

- Configuration vulnerabilities constitute incorrect input to a system. They are often addressable in the field by administrative personnel and represent the widest variance in vulnerability since every installation may be configured in subtly different fashion. An example of configuration vulnerability would be a login configured without a password.
- Design vulnerabilities are represented by limitations in design that either facilitate security lapses or do nothing to preclude them. These vulnerabilities can be difficult to remedy since they are addressed through protocol or standard specification

processes. An example of a design vulnerability is the lack of authentication capability in routing protocols, permitting injection of counterfeit protocol packets.

- Implementation vulnerabilities are represented by errors in programming or other implementation that might commonly be termed “bugs”. These are remedied by patching or re-coding the implementation. An example of implementation vulnerability would be a buffer overflow in an application or OS subsystem that permits overflowing auto variables and injecting object code onto the stack to perform unauthorized program execution.

Diverse research in vulnerability and intrusion detection has been conducted where techniques such as agent-based systems, genetic algorithms, and other methods are discussed [30][32][31][7][11]. Although the main focus of this work is algorithms developed based on reinforcement learning, we have also developed alternative algorithms based on evolutionary computation methods for comparison. In Chapter 2, we will establish the principles of

reinforcement learning and genetic algorithms on which our work is based. We also elaborate on the various terminologies used in communication network as pertain to our discussion and the connection between the theoretical concepts in machine learning and their practical application in vulnerability detection. The network security problem will be linked to a general sequence learning and prediction problem in Chapter 3. Various treatments in sequence learning are also elaborated on. In Chapter 4, we provide an in-depth discussion of the reinforcement learning-based algorithm that we developed to learn the vulnerability occurrence patterns in the network, and compare its performance results with a genetic algorithm-based approach. The performance results are discussed under various pattern scenarios. The performance of each approach with respect to communication noise is assessed and discussed in Chapter 5 by showing system S/N performance in the presence of various levels of noise power. Finally, in Chapter 6 we provide a conclusion and summary of our results and present a discussion of future research.

Chapter 2

Reinforcement Learning and Evolutionary Computation Methods

In the following sections, we provide a theoretical background for two methods discussed in this thesis: reinforcement learning and genetic algorithms. The main algorithms developed in this thesis are based on reinforcement learning techniques. However, for comparison we have also discussed the effectiveness and feasibility of evolutionary methods in the vulnerability detection application. In the later chapters, we establish the framework based on which these two approaches are applied to our specific problem.

In a reinforcement learning problem, the learner is given the ability to ascertain its current circumstance or state, select an action, and observe the consequence of its action until it reaches a goal state [37]. After a sufficient number of interactions with the environment, the learner will determine the optimal action to be taken in any state in order to reach its goal state in the most desirable way. It can keep track of all actions and rewards it has taken through its learning process up to reaching a final state. Reinforcement learning is a computational approach to learning from interaction. It is learning what to do – how to map situations to actions _ in order to maximize a numerical reward signal [37]. The learner must discover which of the possible actions yield the most reward by trying all of them a sufficient number of times. In many cases, actions may affect not only the immediate reward but also the next situation and all subsequent rewards.

The two characteristics _ trial-and-error search and delayed reward _ are the most important distinguishing features of reinforcement learning. The learner must be able to sense the state of the environment to some extent and must be able to take actions that

affect the state. The learner, or agent, must also have a goal or goals relating to the state of the environment. There are three aspects to the problem of learning: sensation, action, and goal. By sensation we mean the learner's ability to understand what its current circumstance, or state, is. Action refers to the learner's selection of alternatives in a given state. Clearly, the goal of the learner is to determine, through interaction with the environment, what series of actions from any state it will need to take in order to eventually reach its goal state.

2.1 Elementary Components of Reinforcement Learning

In addition to the two main entities, the learner (agent) and the environment, we need to consider the basic concepts such as *a policy*, *a reward function*, *a value function*, and a *model* of the environment [37].

2.1.1 A Policy

A policy refers to the learning agent's choice of behaving at a given time in the learning process - its action selection. A policy determines which action should be performed in each state; it is therefore a

mapping from states to actions. In some problems the policy may be a simple function or lookup table, and in others it may involve extensive computation such as a search process. In our problem, we employ a lookup table system of policy evaluation.

2.1.2 The Reward Function

A reward function defines the goal in a reinforcement learning problem. It maps each perceived state (or state-action pair) of the environment to a single number, *a reward*, indicating the intrinsic desirability of that state (or state-action pair). A reinforcement learning agent's sole objective is to maximize the total reward it receives in the long run. The reward function defines what the good and bad events are for the agent. The reward function is the basis for altering a policy in order to improve the reward received in a particular state. For example, if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. The reward function indicates what is good in an immediate sense.

2.1.3 Value Functions

Whereas the reward function indicates what action is good in an immediate sense, the value function represents what action is good in the long run. In other words, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state [37]. Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Upon every step in the learning process the value function is improved (i.e., updated) depending on the reward. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward.

2.1.4 A model of the environment

The model refers to the behavior of the environment. That is, how the state of the environment changes in response to a learner's action. We should note that what we mean by a change in state is the change that is perceived by the learner. Models are used for planning, that is, a way of deciding on a course of action by considering possible future situations before they are actually experienced [37]. Reinforcement learning systems were originally explicitly trial-and-error learners.

Then they evolved into state-space planning models. As a learner obtains more information about the behavior of the environment, it can use a model to plan its future actions.

2.1.5 An illustrative example of value function and optimal policy

In order to relate the elements of a reinforcement learning problem, we provide a simple 16-state example. The state-space can be visualized using a 4-by-4 grid [10]. Each square represents a state. The reinforcement function (i.e., the reward) is arbitrarily chosen to be –1 everywhere. Therefore the learner receives a –1 on each transition. The reward received upon reaching a goal state is, by definition, zero. There are 4 possible actions in each state: north, east, south, west. The goal states are the upper left corner and the lower right corner. The value function for the random policy is shown in Figure 2.1. For each state the random policy randomly chooses one of the four possible actions.

0	-14	-20	-22
-14	-18	-22	-20
-20	-22	-18	-14
-22	-20	-14	0

Figure 2.1. State value function for a random policy

The numbers in the states represent the expected values of the states. For example, when starting in the lower left corner and following a random policy, on average there will be 22 transitions to other states before the terminal state is reached. In this example, we consider the terminal states to be the desirable, final goal states and the learning is said to be achieved when the learner has learned the optimal action, or move, from any square that will bring it to a goal state in as few transitions as possible.

The optimal value function is shown in Figure 2.2. Starting in the lower left corner, calculating the sum of the reinforcements when performing the optimal policy (the policy that will maximize the sum of the reinforcements), the value of that state is –3 because it takes only three transitions to reach a terminal state. If we are given the optimal value function, then it becomes a trivial task to extract the optimal policy. For example, one can start in any state in Figure 2.2 and simply choose the action that maximizes the immediate reinforcement received. The optimal policy for the value function shown in Figure 2.2 is depicted in Figure 2.3.

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0

Figure 2.2. State value function for the optimal policy

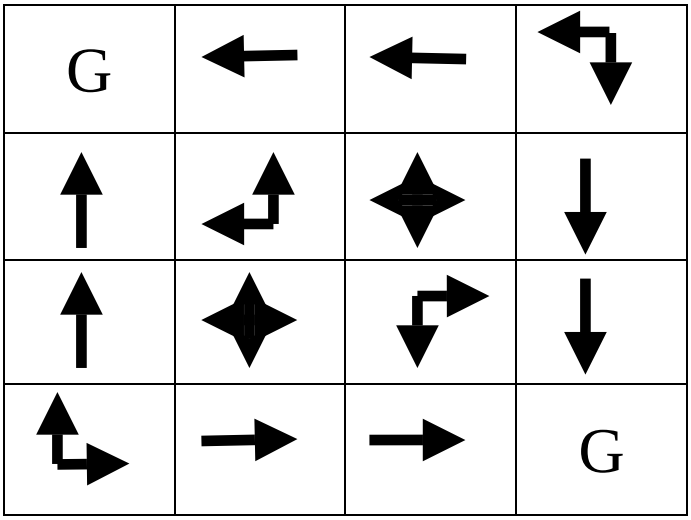

Figure 2.3. Optimal policy corresponding to optimal value function in Figure 2.2.

2.2 Learning by Evaluative Feedback

One of the features of reinforcement learning relevant to our work is that the learner uses training information that evaluates the actions rather than instructs by giving correct actions. The learner needs to actively explore its options in its trail-and-error search for optimal behavior. The simplest problem that can be used to illustrate this evaluative feedback is the n-armed bandit problem [38][9][12][37][4]. We can consider a simple version in which there is only one state or

situation, and the problem is non-associative. A non-associative problem may be viewed as a problem where an action's value does not depend on the situation in which the action is taken, nor does an action change the way the environment responds. The n-armed bandit is analogous to a slot machine, or a one-armed bandit, except that it has *n* levers instead of one. Each action selection is like a play of one of the slot machine's levers, and the rewards are the payoffs for hitting the jackpot. Through repeated plays the goal is to maximize the winnings by concentrating the plays on the best levers. In n-armed bandit problems, each action has an expected or mean reward given that that action is selected. We can call this the *value* of that action. We assume that we do not know the value of each action in the beginning and we will try to determine this optimal action through trial-and-error. If we maintain estimates of the action values, then at any time there is at least one action whose estimated value is greatest. We call this a *greedy* action. If we select a greedy action, we consider this policy an *exploitation* of our current knowledge of the values of the actions. If instead we select one of the non-greedy actions, we call

this *exploration* because this enables us to improve our estimate of the non-greedy action's value.

2.3 ε-Greedy Action-Selection Methods

This method of action selection is based on a learner's attempt to select its next action according to its current knowledge of how good or bad the actions have been in the past [24]. The agent selects the action with the highest average reward most of the time. However, to maintain exploration of its options, the learner occasionally selects an action randomly regardless of what its value may be. This occasional selection is done with a small probability denoted as epsilon and its value needs to be selected based on the particular problem. To illustrate this selection method, consider an n-armed bandit problem [38]. Each arm has a true average value which we call $Q*(a)$, and the estimated value at the t^{th} play as $Q_t(a)$. The true value of an action is the mean reward received when that action is selected. One natural way to estimate this is by averaging the rewards actually received when the action was selected. In other words, if at the t^{th} play action *a*

has been chosen K_a times prior to *t,* resulting in rewards r_1, r_2, r_3, …, $r_{ka,}$ then its value is estimated as shown in Equation 2.1.

$$Q_t(a) = \frac{r_1 + r_2 + r_3 + ... + r_{ka}}{ka}. \qquad (2.1)$$

If $k_a = 0$, then we define $Q_t(a)$ instead as some default value, such as $Q_0(a) = 0$. As $k_a \longrightarrow \infty$, by the law of large numbers $Q_t(a)$ converges to $Q*(a)$. This is called the sample-average method of estimating action values because each estimate is a simple average of the sample of relevant rewards. The learner will select the action with the highest value, that is, action for which $Q_t*(a) = \max_a Q_t(a)$. This method exploits current knowledge to maximize immediate reward. However, in order to maintain exploration, the learner selects the optimum action most of the times but every once in a while, with a small probability ε, it selects as action at random, uniformly, independent of the action-value estimates. This method of action selection is called ε-greedy. The advantage of this method is that, in the limit as the number of plays increases, every action will be

sampled an infinite number of times and thus ensuring that all the $Q_t(a)$ converge to $Q*(a)$. There are numerous variations to the e-greedy action selection method and the best choice of ε has been the subject of much research [42][22].

2.4 Incremental Implementation Method of Computing Action Values

The action-value methods discussed in this chapter estimate values as sample averages of observed rewards. The obvious implementation is to maintain, for each action *a*, a record of all the rewards that have followed the selection of that action. Then, when the estimate of the value of action *a* is needed at time *t*, it can be computed according to Equation (2.1), where r_1, r_2, …, r_{ka} are all the rewards received following a selection of action *a* prior to play *t*. A problem with this implementation is that its memory and computational requirements grow over time without bound. That is, each additional reward following a selection of action *a* requires more memory to store it and results in more computation being required to determine $Q_t(a)$. In

order to prevent this problem, an incremental update formula is used for computing averages with small, constant computation to process each new reward [37]. For some action, let Q_k denote the average of its first *k* rewards (not to be mistaken with $Q_k(a)$, the average for action *a* at the kth play). Given this average and a (k + 1)st reward, r_{k+1}, then the average of all k + 1 rewards can be computed by

$$
\begin{aligned}
Q_{k+1} &= \frac{1}{k+1}\sum_{i=1}^{k+1} ri \\
&= \frac{1}{k+1}(r_{k+1} + \sum_{i=1}^{k} r_i) \\
&= \frac{1}{K+1}(r_{k+1} + k.Q_k + Q_k - Q_k) \\
&= \frac{1}{K+1}(r_{k+1} + (k+1)Q_k - Q_k) \\
&= Q_k + \frac{1}{k+1}[r_{k+1} - Q_k], \qquad (2.2)
\end{aligned}
$$

which holds even for k = 0, obtaining $Q_1 = r_1$ for arbitrary Q_0. This implementation requires memory only for Q_k and k, and only the

small computation (2.2) for each new reward. The general form for the update rule (2.2) is

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate] \tag{2.3}$$

The expression *[Target – OldEstimate]* is the *error* in the estimate. It is reduced by taking a step toward the "Target." The target is presumed to indicate a desirable direction in which to move, though it may be noisy. In the case above, for example, the target is the (k + 1)st reward. Another parameter of interest here is the step-size used in the incremental method. In processing the k^{th} reward for action *a*, this method uses a step-size parameter of 1/*k*. In order to maintain consistency in conventional reinforcement learning terminology, we choose the symbol α to denote step-size parameter. The above incremental implementation of the sample-average method is described by the equation $\alpha_\kappa(a) = 1/k_a$. Accordingly, sometimes the informal shorthand $\alpha = 1/k$ is used to refer to this case, leaving the action dependence implicit [37].

2.5 Associative Search

The n-armed bandit example we mentioned above was a non-associative search for the best action. There was no need to associate actions, i.e., pulling an arm, with different situations. In such tasks, the learner either tries to find a single best action when the task is stationary, or tries to track the best action as it changes over time when the task is non-stationary (the dynamics of the environment, and consequently the choice of best actions, change over time). However, in a general reinforcement learning task, there are more than one situations, and the goal is to learn a policy: a mapping from situations to the actions that are best in those situations. As an example of an associative search [3] problem consider the n-armed bandit problem. Suppose there are several bandit tasks, and on each play you experience one of these tasks chosen at random. Thus the bandit task changes randomly from play to play. This would appear to you as a single, non-stationary n-armed bandit task whose true action values change randomly from play to play. Unless the action values change slowly, the method we considered will not work very well. For

example, the average value of a particular arm may change depending on the value of another arm pulled before it. The successful learner would try to learn by trial-and-error in the form of search for the best actions in association with the situations in which they are best. In this thesis, we will discuss how a search can be associative or non-associative depending on the dynamics of the network environment.

2.6 The Learner – Environment Interaction

The learner is also known as the *agent* in the reinforcement learning literature and we will use the word agent in this thesis as well. The entity that the agent interacts with in order to learn a task is known as the *environment*. The interaction continues as the agent selects actions and the environment responds to those actions and presents new situation to the agent. The interaction gives rise to rewards, special numerical values that the agent tries to maximize over time. A complete specification of an environment defines a *task*, one instance of the reinforcement learning problem. Figure 2.4 depicts the agent-environment interaction.

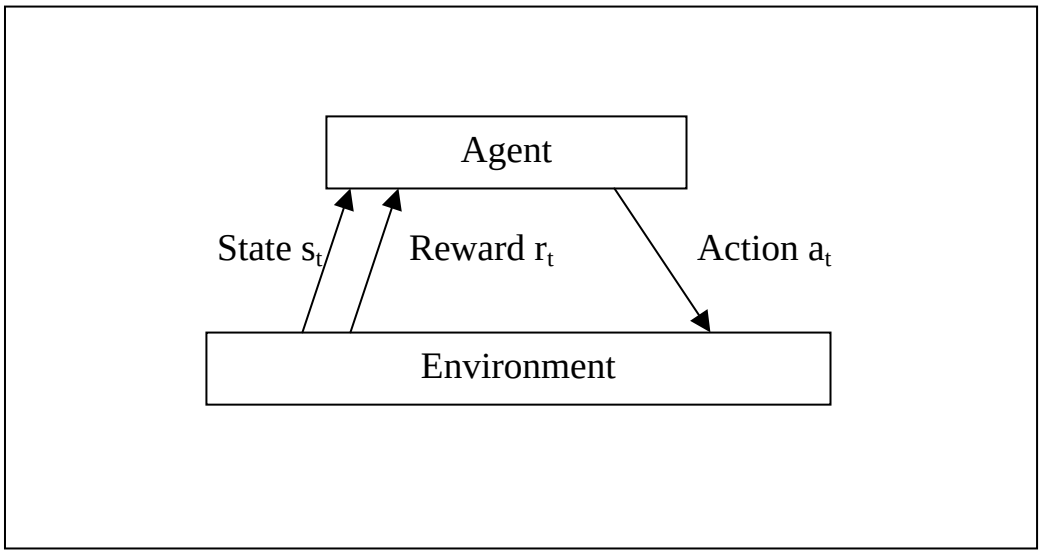

Figure 2.4 The agent-environment Interaction

The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At a time step *t*, the agent receives some representation of the environment's *state* [22], $s_t \in S$, where S is the set of possible states, and on that basis selects an *action*, $a_t \in A(s_t)$, where $A(s_t)$ is the set of possible actions available in state s_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in R$, and finds itself in a new state, s_{t+1}.

2.7 Goals, Rewards, Returns

The propose or goal of a reinforcement learning agent is defined on the basis of the reward signal passing from the environment to the agent. At each time step, the reward is a simple number, $r_t \in R$. The agent's goal in general is to maximize the cumulative reward in the long run. For example, in the n-armed bandit problem the goal of the agent is to determine the arm with the highest average value. Similarly, in the grid problem illustrated in Section 2.1.5, the goal of the agent was to find the sequence of actions (turns) from any square that brought the agent to a goal state with the maximum accumulated reward possible. Clearly, in order for the agent to achieve learning in a task, we need to provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is therefore critical that the rewards we set up truly indicate what we want accomplished.

Another key concept in reinforcement learning problem is the *return*. In order to establish a definition for the idea of maximizing the reward to achieve a goal in the long run, consider that the sequence of rewards after time step *t* is denoted r_{t+1}, $r_{t+2,}$ $r_{t+3,}$ …. In general, we

want to maximize the *expected return*, where the return, R_t, is defined as some specific function of the reward sequence. In the simplest case, the return is the sum of the rewards is expressed as in Equation (2.4).

$$R_t = r_1 + r_2 + r_3 + \ldots r_{T,} \quad (2.4)$$

T in the Equation (2.4) is a final step. This approach makes sense in applications in which there is a natural notion of final time step, when the agent – environment interaction is naturally divided into subsequences called the *episodes*. A complete play of a game, a run through a maze, a sequence of turns from start to goal in the grid problem, etc., are all examples of *episodic* problems. The term episode is known as trial in some literature [37]. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state. There are also non-episodic tasks, or continuing tasks, which form a continuous series of interactions between the agent and the environment. In such tasks, the final time step would be $T = \infty$. In order to maintain a finite value for return

even in infinite series of time steps, a discount rate, γ, can be inserted in the equation for return. The agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. For example, it chooses a_t to maximize the expected *discounted return*:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \qquad (2.5)$$

In Equation (2.5), γ is a discount rate, $0 \leq \gamma \leq 1$.

2.8 State and Action Value Functions

Before we discuss some of the main reinforcement learning methods, namely the Monte Carlo (MC) and the Temporal Difference (TD) learning methods, we will reintroduce the value functions here using the notations in this Section. All reinforcement learning algorithms are based on estimating value functions – functions of states (or of state-action pairs) that estimate how good it is for the agent to be in those given states (or how good it is to perform a given action in a

given state). The concept of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected returns. The rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular policies. A Policy, π, can be defined as a mapping from each state, $s \in$ S, and action $a \in$ A(s), to the probability π (s, a) of taking action a when in a state s. The value of a state s under a policy π, denoted $V^{\pi}(s)$, is the expected return when starting in s and following π thereafter. For a Markov Decision Process, we can define $V^{\pi}(s)$ formally as [5][25] [37][45][43][44]

$$V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\}. \qquad (2.6)$$

We call the function $V^{\pi}(s)$ the state-value function for policy π. An important function that will be used extensively in this thesis is the $Q^{\pi}(s, a)$, or the action-value function for policy π. This function

denotes the expected return a starting from *s*, taking the action *a*, and thereafter following policy π:

$$Q^{\pi}(s, a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\} \quad (2.7)$$

As each episode terminates, the values of all actions in various states encountered in the episode are averaged according to formulas such as above. When each state-action pair is encountered an infinite number of times, the corresponding action-value will converge to an average number. The goal of the agent is then to periodically select an action at random to search and find the state-action values with the highest average values. Learning is accomplished when the agent eventually determines the state-action pairs with the highest value.

2.9 Optimal Value Functions

The learning task of an agent is completed when a policy that achieves the highest rewards in the long run is found by the agent. Value

functions define a partial ordering over policies. A policy π is defined to be better than a policy π' if its expected return is greater than or equal to that of π' for all $s \in S$. There is always at least one policy that is better than, or equal to, all other policies. This is called the *optimal policy*. We denote the optimal policy by π^*. There may be more than one optimal policies but they all have the same state-value function, called the optimal state-value function, denoted as V^*, and defined as

$$V^*(s) = \max_{\pi} V^{\pi}(s), \text{ for all } s \in S \qquad (2.8)$$

Optimal policies also have optimal action-value function in common, denoted Q^*, and defined as

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a), \text{ for all } s \in S \text{ and } a \in A \qquad (2.9)$$

The expressions for the optimal action-value function in terms of the expected return for taking action *a* in state *s* and thereafter following an optimal policy can be written as follows:

$$Q^*(s, a) = E\{r_{t+1} + \gamma V^*(S_{t+1}) \mid s_t = s, a_t = a\} \quad (2.10)$$

2.10 Monte Carlo Methods

Although we did not use Monte Carlo learning methods [18][26] in this work, we briefly mention them since they are similar to Temporal Difference methods, used in our work, in that they do not require any prior knowledge of the environment in order to achieve learning. Furthermore, Monte Carle methods can be considered a variation of Temporal Difference learning methods with some variation in the treatment of the delayed rewards and their updating of state-action values [21]. Monte Carlo methods involve learning in an episode-by-episode sense, but not in a step-by-step sense [2]. They allow the agent to learn action values based on averaging complete returns, as opposed to averaging partial returns used in methods such as Temporal-Difference Learning. In summary, at the end of each

learning episode, Monte Carlo methods use action values encountered in that particular episode to update (and improve) their previous action values. Figure 2.5 shows the back-up diagram for this learning process. The diagram in Figure 2.5 is called a *back-up diagram* because they show relationships that form the basis of the update or *backup* operations fundamental to reinforcement learning [37]. These operations transfer value information *back* to a state (or a state-action pair) from its successor states (or state-action pairs).

In Figure 2.5, hollow circles represent a state and solid circle ovals represent the action taken at a visited state. As shown, upon taking an action, the agent encounters a new state in the environment. This process continues until the agent reaches the Terminal State where the learning episode is completed and a new episode should start. At the end of the episode, the agent will look back and update the values of all the states (or state-action pairs) it visited in that episode according to the rewards it received.

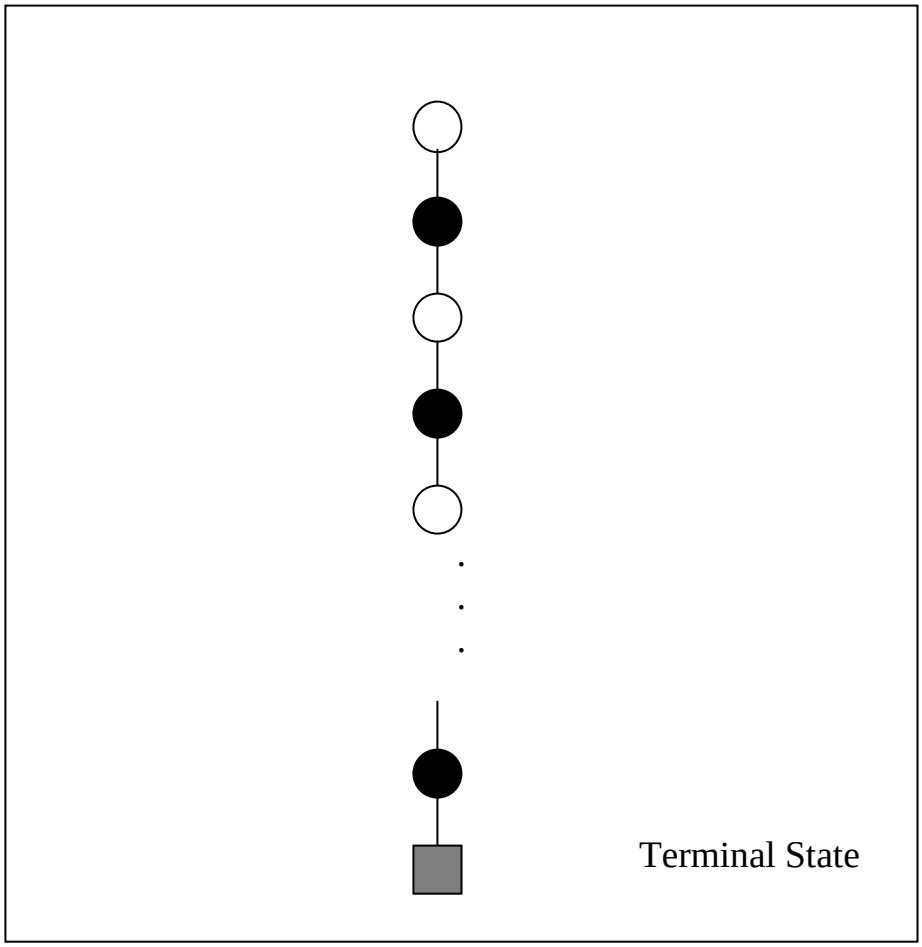

Figure 2.5 The backup Diagram for Monte Carlo Estimation of V^{π}.

2.11 Temporal Difference Learning Methods

Temporal Difference (TD) learning methods [19] are our choice for developing algorithms in our work. There are two variations of the TD learning methods: on-policy or Sarsa [27][36], and off-policy or Q-learning [40][41]. As mentioned previously, TD learning methods

are similar to Monte Carlo methods in that they both are applied to episodic problems and require no prior knowledge of the environment. TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The general case of TD methods is the TD(λ) in which both Monte Carlo and TD learning methods are generalized in a seamless approach. Both methods update their estimates of a state value $V(s_t)$ based on what happens after that visit. Monte Carlo methods wait until the return following the visit to each state in the episode, then use the return as a target for $V(s_t)$. A basic every-visit Monte Carlo method suitable for non-stationary environments is

$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)], \qquad (2.11)$$

where R_t is the actual return following time *t* and α is a constant step-size parameter. The Equation (2.11) is analogous to Equation (2.3) mentioned earlier in the chapter. This method is also called constant-α MC. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(s_t)$ (only then is R_t known),

TD methods need wait only until the next time step. At time t + 1 they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$. The simplest TD method, known as TD(0), is

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]. \quad (2.12)$$

The target for Monte Carlo update is R_t, whereas the target for the TD update is $r_{t+1} + \gamma V_t(s_{t+1})$. Because the TD method's updating is based on an existing estimate, we say that it is a *bootstrapping* method.

The TD(0) learning method is the basic approach for our work. Figure 2.6 describes TD(0) in procedural form, and Figure 2.7 shows its back-up diagram.

Initialize V(s) arbitrarily, π to the policy to be evaluated
Repeat (for each episode):
    Initialize s
    Repeat (for each episode):
    α ← action given by π for s
    Take action a; observe reward, r, and next state, s'
    V(s) ← V(s) + α [r + γV(s') – V(s)]
    s ← s'
until s is terminal

Figure 2.6 TD(0) method of estimating v^{π}

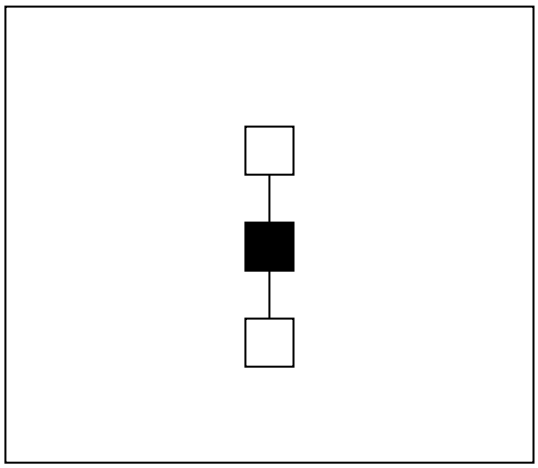

Figure 2.7 The backup diagram for TD(0)

As shown, the value estimate for the sate node at the top of the back-up diagram is updated on the basis of the one sample transition from it to the immediately following state. The TD and Monte Carlo updates are referred to as *sample backups* because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then changing the value of the original state (or state-action pair) accordingly [37].

2.11.1 On-Policy TD Method (Sarsa)

TD control methods fall into two categories of on-policy and off-policy, depending how the states or state-action pairs are evaluated. We start by learning an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $Q^{\pi}(s, a)$ for the current behavior policy π and for all states s and actions a. This is done essentially using the same method for evaluating V^{π}. The episode consists of an alternating sequence of states and state-action pairs. Just as transitions from state to state result in learning the values of states, we can consider moving from

state-action pair to the next state-action pair, and learn the value of state-action pairs. These are basically Markov chains with a reward process. The theorems assuring convergence of state values under TD(0) also apply to the corresponding algorithm for state-action values:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \qquad (2.13)$$

This update is done after every transition from a non-terminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as zero. Note that this rule used every element of the quintuple of events $(s_t, a_t, s_{t+1}, a_{t+1})$ that make up a transition from state-action pair to the next. This gives rise to the terminology *Sarsa* for the algorithm. The control algorithm for this on-policy TD method is shown in Figure 2.8. The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q [27] [28]. For example, one could use ε – greedy or ε – soft policies. Sarsa converges with the probability 1 to an optimal policy and action – value function as long as all state-action pairs are

visited an infinite number of times and the policy converges in the limit to the greedy policy.

Initiate *Q(s,a)* arbitratily
Repeat (for each episode):
    Initialize *s*
    Choose *a* from *s* using policy derived from *Q* (e.g., ε – greedy)
    Repeat (for each step of episode):
        Take action *a*, observe *r, s'*
        Choose *a'* from *s'* using policy derived from *Q* (e.g., ε – greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma\, Q(s',a') - Q(s,a)]$
        $s \leftarrow s';\ a \leftarrow a';$
    until s is terminal

Figure 2.8 Sarsa algorithm: An On-policy TD control method

2.11.2 Q-Learning: Off-Policy TD Method

The Q learning is considered an important breakthrough in reinforcement learning. In the simplest form, *one-step Q-learning*, is defined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\,[\, r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \quad (2.14)$$

In this case, the learned action-value function, Q, directly approximates Q*, the optimal action-value independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. Figure 2.9 shows the Q-learning control algorithm.

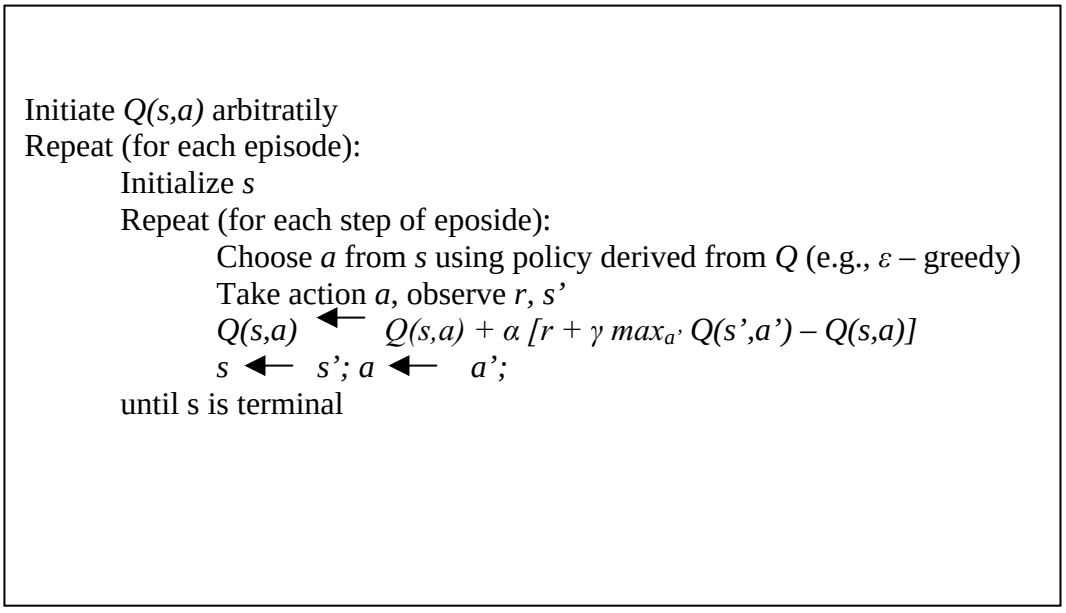

Figure 2.9 Q-learning: An off-policy TD control algorithm.

The back-up diagram for the Q-learning algorithm is shown in Figure 2.10 below.

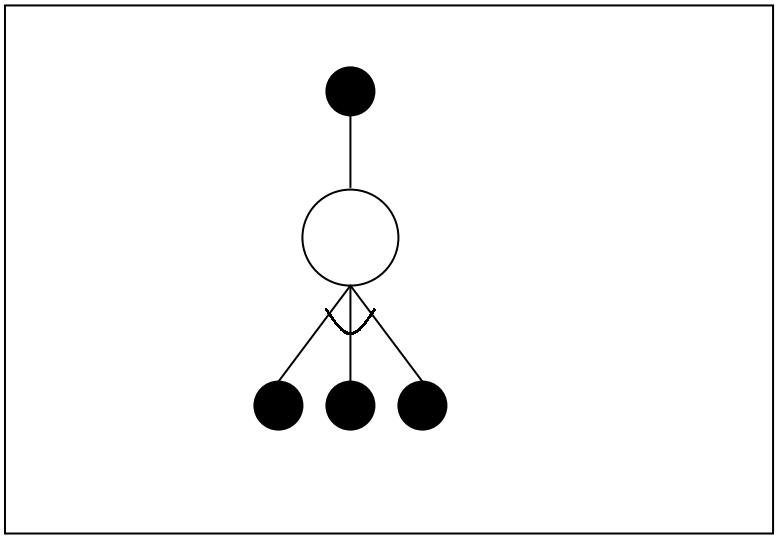

Figure 2.10 The backup algorithm for Q-learning.

2.12 Convergence of One-Step Sarsa

We developed our reinforcement learning algorithm based on one-step Sarsa, or Sarsa(0), whose convergence to optimal policies have been the subject of extensive studies. In this section, we elaborate on one such study [41][28].

A learning policy selects an action at time step *t* as a function of the history of states, actions, and rewards experienced up to this point. Here we consider several learning policies that make decisions based on a summary of history consisting of the current time step *t*, the

current state *s*, the current estimate *Q* of the optimal *Q*-value function, and the number of times state *s* has been visited before time *t*, $n_t(s)$. Such a learning policy can be expressed as the probabilities $\Pr(a/s, t, Q, n_t(s))$, the probability that action *a* is selected given the history. We note that the proof of Sarsa(0)'s convergence based on a learning policy for Markov decision processes that fits into the category of *decaying exploration* learning policies has been discussed in details [15][16]. We have used such policies in our algorithm with favorable results. A decaying exploration, as opposed to a persistent exploration policy, is a policy that becomes more and more greedy over time. The advantage of decaying exploration policies is that the actions taken by the system may converge to the optimal ones eventually, but with the price that their ability to adapt slows down. This category of learning policy is characterized by two properties:

- Each action is executed infinitely often in every state that is visited infinitely often,
- In the limit, the learning policy is greedy with respect to the Q-value function with probability 1.

To ensure the convergence of Sarsa(0), we require a look-up table representation for the Q values and infinite visits to every state-action pair, just as for Q-learning. Sarsa(0) is an on-policy algorithm and in order to achieve its convergence to optimality we have to further assume that the learning policy becomes greedy in the limit.

2.13 A Review of Genetic Algorithms

Genetic algorithms provide an approach to learning that is based on evolutionary processes found in nature [9][13]. The solutions (individuals in the population) are usually described by bit strings with genetic material designed according to the specific optimization problem. One can describe individuals by symbolic expressions or even computer programs. The search for best solution(s) begins with a population of initial individuals. Members of current population give rise to the next generation by means of operations such as random mutation and crossover, which are patterned after processes in biological evolution. Genetic algorithms (GAs) have been used in a variety of problems and they are known to be a successful, robust method for adaptation within biological systems. They can search

spaces of solutions containing complex interacting parts, where the impact of each part on overall individual fitness may be difficult to model. In its simplest form, an individual is represented by a binary string where each bit, or group of bits, signify a gene. The collection of genes forms the chromosome or individual. In each population, the fitness of each member is computed according to a fitness function depending on how successfully the individual solves a particular problem. Genetic algorithms are best understood through simple examples.

2.13.1 Genetic Operators

A new population of individuals is selected through genetic operators. Typical GA operators for manipulating bit strings are *reproduction*, *crossover*, and *mutation*.

The process starts from an initial population consisting of a number of individuals. Individuals are selected according to their fitness and then, offspring from selected individuals are produced. Parents are recombined to produce offspring. Each offspring is then mutated according to a certain mutation probability. The offspring are then

inserted into the population replacing the parents, producing a new generation. This process is repeated until some optimization criterion is reached.

Evolutionary algorithms search a population of points (solutions) in parallel not just a single point. They do not require derivative information of other auxiliary knowledge, only the objective function and corresponding fitness levels influence the directions of search. Perhaps the most powerful feature of GAs is their ability to search a large solution space, introducing a high degree of variety in every search step. The individuals that are chosen for mating (recombination) and how many offspring each individual produces are determined by the *selection* method. There are various selection methods such as roulette-wheel selection, stochastic universal sampling, truncation selection, and tournament selection. The selection method used later in this thesis is roulette wheel selection. The *crossover* operator randomly chooses a crossover point where two parent chromosomes exchange segments of their genes. The new offspring are created from this process, each containing parts of the parent genetic make-up. Single and multi-point crossover methods

can be used. After recombination, every offspring undergoes mutation according to a small mutation probability. In a binary string structure for example, a gene can change value according to the mutation probability from 0 to 1 or vice versa. After producing offspring, a new population must be created by reinsertion of new offspring. There are cases where the new population is not the same size as the previous population. Similarly, not all offspring are always used to form new population. A reinsertion scheme is needed to determine which individuals should be re-inserted into the new population and which individuals are to be replaced with the new offspring.

Chapter 3

Vulnerability Prediction: A Sequence Learning Problem

We previously discussed how the problem of vulnerability detection in face of bandwidth constraint is optimized by learning to predict the next vulnerabilities that might be generated in the network. The problem of prediction can be considered in this way: The mobile network is an environment in which a set of finite, distinct, and recognizable entities (i.e., vulnerabilities) may be created at any time. Such entities or symbols may be generated from time to time in a group or individually. These symbols cannot be *seen* by an observer, or dispatcher, without a cost. The cost is to spend bandwidth and other resources to make observations: For every known symbol, there is a corresponding detection test designed to recognize it. The dispatcher selects a symbol test(s) and transmits it to the network. If

that particular symbol exists in the network, a positive response is transmitted back to the dispatcher and a discovery is accomplished. Otherwise, no positive response is received. When the dispatcher transmits the wrong symbol tests and a negative response is received from the network, the dispatcher does not know what symbols do exist. It only knows that the particular symbol(s) is sought to detect do not exist. This is what we mean by the dispatcher not being able to *see* all the symbols generated and having only partial information about the specific tests it applied at a particular time.

3.1 Discrete Sequence Prediction

The general problem of sequence learning has been studied in numerous literatures. Sequence prediction has been defined as the task of finding statistical regularities in the input data so that the ability to predict the next symbol becomes better than random guessing, when the input consists of an infinite stream of symbols [7]. More generally, we can see the sequence prediction problem as a kind of sequence learning [34][35]. An adaptive approach, one that improves the knowledge of the predictor about the sequence

generator, has been applied using different methods depending on the particular application. A sequence prediction problem may treat the sequence as a train of finite discrete values and attempt to predict the next symbol with the highest probability [7]. In another type of approach, the sequence is assumed to be a pseudorandom series of binary elements generated by a pseudorandom sequence generator [11]. We provide a brief overview of various techniques applies to these problems in the following sections before presenting our assumptions and algorithms in the following chapter.

3.2 A Genetic Algorithm for Discrete Sequence Prediction

In this approach, a hybrid method for sequence prediction is presented that combines a kind of Markovian model and a genetic algorithm [38]. Then a genetic search is applied for finding appropriate weights for the predictions provided by the probability tree.

3.2.1 The Prediction Method

First, a finite (N), but sufficiently large, number of symbols are considered. If N previous values X_1, X_2, …, X_N are recorded, a

prediction for X_{N+1} is made on the basis of the past 1, 2, 3, …, N-1 or N values. Suppose that the symbols that can occur in the sequence are the numbers 0, 1, …, *k*-1, if *k* is the number of different symbols in the sequence. Let $P_i^j(X,N)$ denote the probability that based on the *i* most recent previous values X_{N-i+1}, …, X_N, symbol *j* should follow in the sequence (that is, $X_{N+1} = j$). Since we generally do not know in advance on what number of previous values it is best to base the prediction, we will use a weighted sum of all predictions based on 1, 2, …, N previous values. Then the probability that the next value is *j* can be computed as

$$P^j(X,N) = \sum_{i=1}^{N} W_i \times P_i^j(X,N), \qquad (3.1)$$

where W_i is the weight of the prediction based on *i* values (i.e., X_{N-i+1}, …, X_N) that symbol *j* should follow in the series (that is, $X_{N+1} = j$).

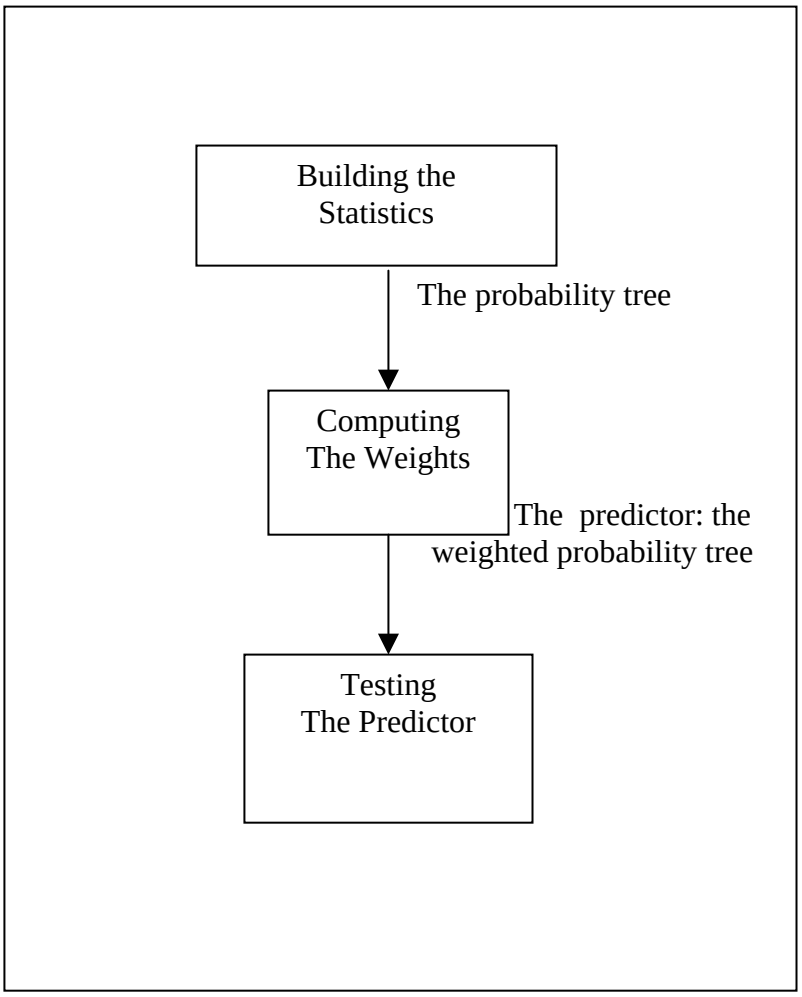

Figure 3.1 The prediction model

The sequence predictor is build in two phases, as shown in Figure 3.1, Accordingly, we divide the available segment of the sequence into training data (further divided into the segment for building statistics, and the segment for computing the weights), and test data.

The predictions based on *i* previous values $P_i^j(X,N)$, *i* = 1, …, N, *j* = 1,…, *k*-1, are computed from the first subset of the training data, for each possible sequence of symbols of length N that appeared in that part of the sequence. The method for computing the probabilities relies on constructing a probability tree for the possible sequences of length up to N as described by Laird and Saul [20]. The genetic algorithm [9] is applied for finding the appropriate weights. The predicted next value, when using a given weight vector, is

$$X_{N+1} = j, \text{ where } P^j(X,N) = \max_{l=0}^{k-1} P^l(X,N) \qquad (3.2)$$

The weights are evolved for accurately predicting the symbols in the second segment of the training data [7]. The performance of the evolved predictors is measured on the last segment of the sequence that has been set aside for testing.

3.2.2 The Data Structure for Maintaining Statistical Information

We construct a probability tree based on the selected part of the sequence. To each subsequence of length i (i = 0, …, N) that occurs in this part of the sequence there is associated a path from the root of the tree to a node at depth i in the tree (by convention, the root is at depth i = 0). The j^{th} child of a node at depth i corresponds to the situation when the i symbols in the corresponding subsequence X_{N-i+1}, …, X_N are followed by $X_{N+1} = j$. This node contains as information the probability value $P_i^j(X, N)$.

For simplicity, this algorithm presents the tree constructed in the case of a binary sequence (k = 2) containing the symbols 0 and 1. Then it is assumed that the part of the sequence for constructing the probability tree is

101100101011. (3.3)

It is shown in Figure 3.2 the probability tree for N – 2 previous values.

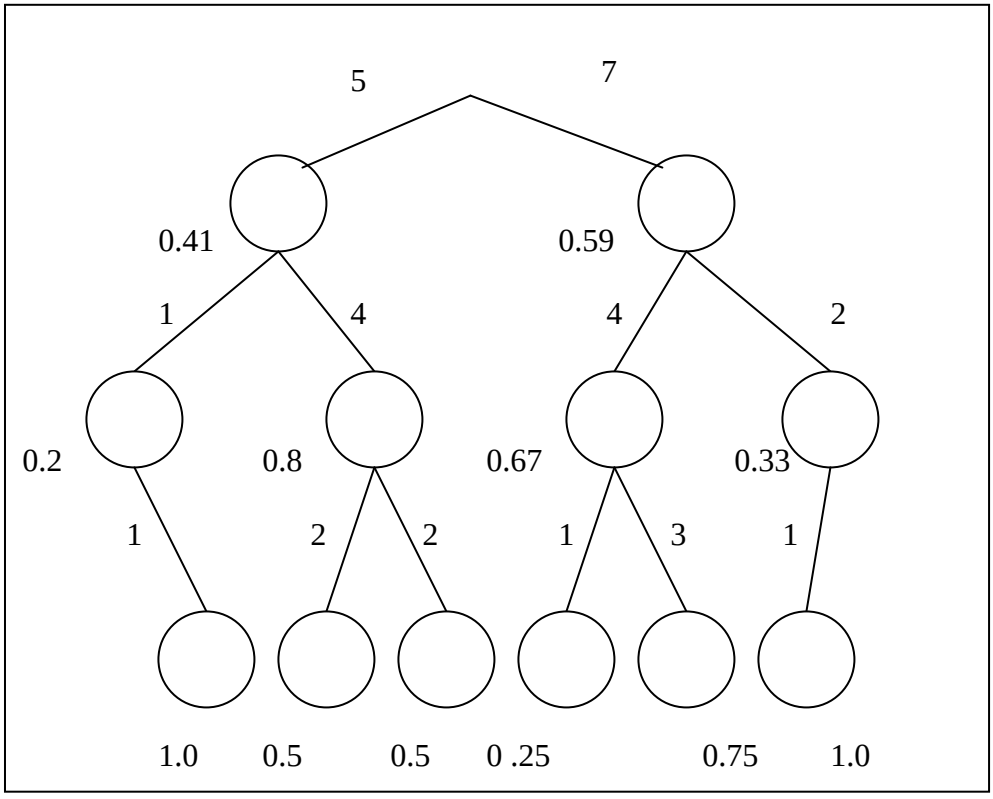

Figure 3.2 An example probability tree.

The left child of each node corresponds to next symbol in the sequence 0 and the right child to next symbol in the sequence 1, respectively. On each arc we mark the number of occurrences of the corresponding subsequence. The marked node containing 0.67 corresponds to the subsequence 1 followed by next symbol 0. In the

given sequence, the subsequence 1 is followed by 0 four times and by 1 two times, as marked on the arcs.

The tree is constructed in two steps. First a tree structure is built on the basis of the given sequence. The new nodes are dynamically created when needed and the information associated to the corresponding arcs is updated. When the tree is completed, i.e., all the data in the sequence have been considered, we compute the probability values. The value for *node* is

$$P_{node} = \frac{in(node)}{\sum_{j=0}^{k-1} in(child_j(parent(node)))}, \qquad (3.4)$$

Where *in*(T) denoted the number on the arc from *parent*(T) to node T and $child_j$(T) – the *j*th child of T in the tree.

We now consider a situation when at some later moment a subsequence 01 occurs in the sequence. Then, considering the prediction based on two previous values the probabilities of having a 0 or a 1 as next symbol are both equal to 0.5. Based on one previous value, the probability of getting a 0 is 0.67, so the next symbol would

be predicted as 0. Based on no previous values, the probability of having a 1 is 0.59, so the next symbol would be predicted as 1. A weighted sum of these probabilities can lead to a more reliable prediction.

3.2.3 The Genetic Algorithm

In this approach, a genetic algorithm [9] was devised for finding the weights of the predictions based on different numbers of previous values. A genetic search is performed in the space of N-dimensional weight vectors. The component W_i of a weight vector is associated to the prediction based on *i* previous values. Starting from a random population of weight vectors, by repeatedly applying the genetic operators in a number of iterations (generations) an improved weight vector is obtained. The fitness evaluation of the individual weight vectors consists of computing their success rate on the selected segment of the sequence.

Evolution is driven by selection of better performing (i.e., fitter) individuals in each generation, the fitter individuals have better chances for survival throughout the generations. The results of the

algorithm here are based on the special case of binary sequence and are provided and discussed in details in Ekart's work [7]. It should be noted that this treatment is applicable to sequences with any finite number of distinct symbols.

3.3 Learning a Pseudorandom Sequence

Our earlier work on sequence prediction included the learning the dynamics of a sequence that consisted of a series of binary digits that occur in a pseudorandom fashion. Noting that a pseudorandom sequence can be generated using a bank of flip-flops arranged with a memory-less feedback configuration, we used genetic algorithms to search in the space of all possible pseudorandom generators to find the one corresponding to the particular sequence of interest. Generally, a pseudorandom generator can be constructed as shown in Figure 3.3.

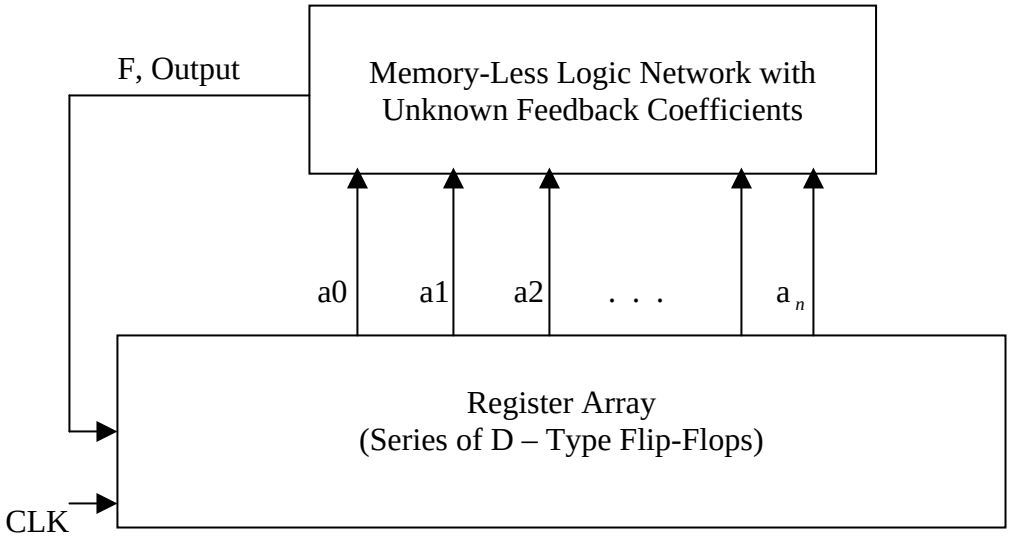

Figure 3.3. Pseudorandom symbol generator with unknown feedback coefficients.

Solving the prediction problem is analogous to finding the linear feedback shift register system that would generate the signal being observed. By finding the pseudorandom signal generator, one is able to deterministically predict all future symbols in the sequence. This prediction is of course possible assuming the sequence if free of noise. In the real world applications, one needs to devise a prediction system that takes noisy symbols in the sequence under consideration. Genetic algorithm methods have shown acceptable degree of robustness in the

presence of noise [11]. In the following sections, we elaborate on some genetic algorithm solutions to pseudorandom sequence prediction problem.

3.3.1 How Pseudorandom Sequence Prediction Works

The symbols in a pseudorandom sequence can be thought of as data generated in a point of observation in a pseudorandom noise (PN) generator. For example, the observation may be done at the output of the feedback logic circuit (output F) in Figure 3.3. Assuming a given initial loading of the flip-flops, the genetic search process is as follows:

- Observe the first bit in the sequence
- Given the initial condition of the flip flop bank, search through all possible feedback configurations that may give rise, in the output observation point, to the observed bit
- Assign high fitness, according to some objective function, to the feedback configuration circuits capable of producing the observed bit; assign low fitness to all others

- Apply the genetic algorithm operators to obtain a new population
- Observe the second bit in the pseudorandom sequence
- Repeat the procedure to find a new fit population
- The search is completed until all members of feedback logic population are high fitness.

To elaborate on the above problem, we provide the following example in which a pseudorandom sequence is generated by a 3-stage PN generator. Also we assume the initial condition, i.e., the initial loading of the flip-flops in the register array is known.

3.3.2 Search for a Pseudorandom Sequence: An Example

Let us consider the PN generator in Figure 3.4, with initial condition for the three flip-flops D1, D2, D3 as [0, 0, 1], respectively. Furthermore, assume the point of bit observation is the output of D1 flip-flop. Let the sequence of interest be:

Pseudorandom Sequence = 1 1 1 0 1 0 0 1 1 1 0 1 0 0 … (3.5)

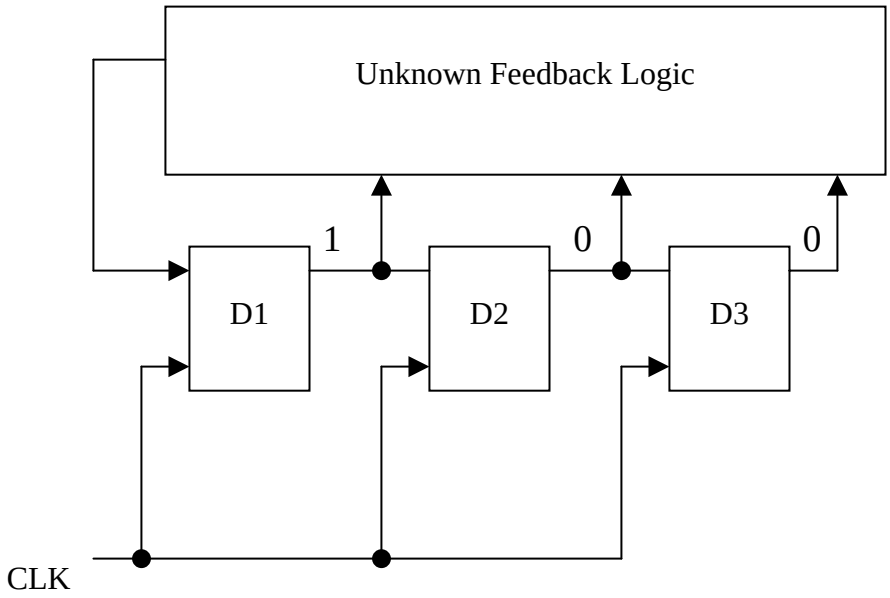

Figure 3.4 Unknown pseudorandom noise generator with initial condition of [1 0 0].

The desired feedback logic can be represented as function F, as shown in Equation (3.6).

$$F(\vec{d}) = \sum_i b_i d_i , \qquad (3.6)$$

Where i represents the number of registers in the array, b_i's are feedback coefficients {0, 1}, and d_i's are outputs of the flip-flops.

Starting with the first clock pulse, we randomly generate an initial population of feedback logic circuits. In the above example, each feedback circuit may be represented by a chromosome with three genes selected from {0,1}. For example, a chromosome [1 0 0] would represent a feedback circuit with only the output of D1 flip-flop connected in the feedback loop. That is,

$b_1 = 1,\ b_2 = 0,\ b_3 = 0;$

F = $b_1.d_1 \oplus b_2.d_2 \oplus b_3.d_3$.

The initial population is shown in Table 3.1.

Initial Population
1 1 0
1 0 0
0 1 0
0 1 1

Table 3.1 Initial population of PN generators

At the first clock pulse, each individual chromosome gives rise to a particular feedback function F. Since we are observing the bit at the output of the first flip-flop (d_1), the resulting F generated by each chromosome is computed and compared to the observed bit. A chromosome whose corresponding PN generator results in

d_1 = observed bit,

receives a high fitness number (in this case, 10), otherwise it receives a low fitness number of 2.

Initial Population	Feedback Output F	d_1 Value Associated with the Individual	Fitness
1 1 0	1	1	10
1 0 0	1	1	10
0 1 0	0	0	2
0 1 1	0	0	2

Table 3.2 Fitness of the first population

This process is continued until only a single chromosome is dominant in the entire population and the search converges to the unique answer. Table 3.2 shows the fitness of the first population. After fitness values are determined, the individuals undergo the genetic processes of reproduction, cross-over, mutation, and the second population is produced. This process is shown in Table 3.3.

Initial Population/ Fitness	Reprod-uction	Cross-over	2nd Populat ion/ Fitness	Reproduct -ion & Cross-over	3rd Population
1 1 0 / 10	1 1 0	1 1 0	2	1 0 0	1 0 0
1 0 0 / 10	1 0 0	1 0 0	10	0 1 0	0 1 0
0 1 0 / 2	0 1 1	0 1 0	10	1 1 0	1 1 0
0 1 1 / 2	1 1 0	1 1 1	2	1 0 1	1 0 1

Table 3.3 Derivation of 2nd and 3rd populations

In the above example, usually in three clock pulses the answer (i.e., the PN generator creating the observed pseudorandom sequence) is found. The algorithm results are shown in Figure 3.5.

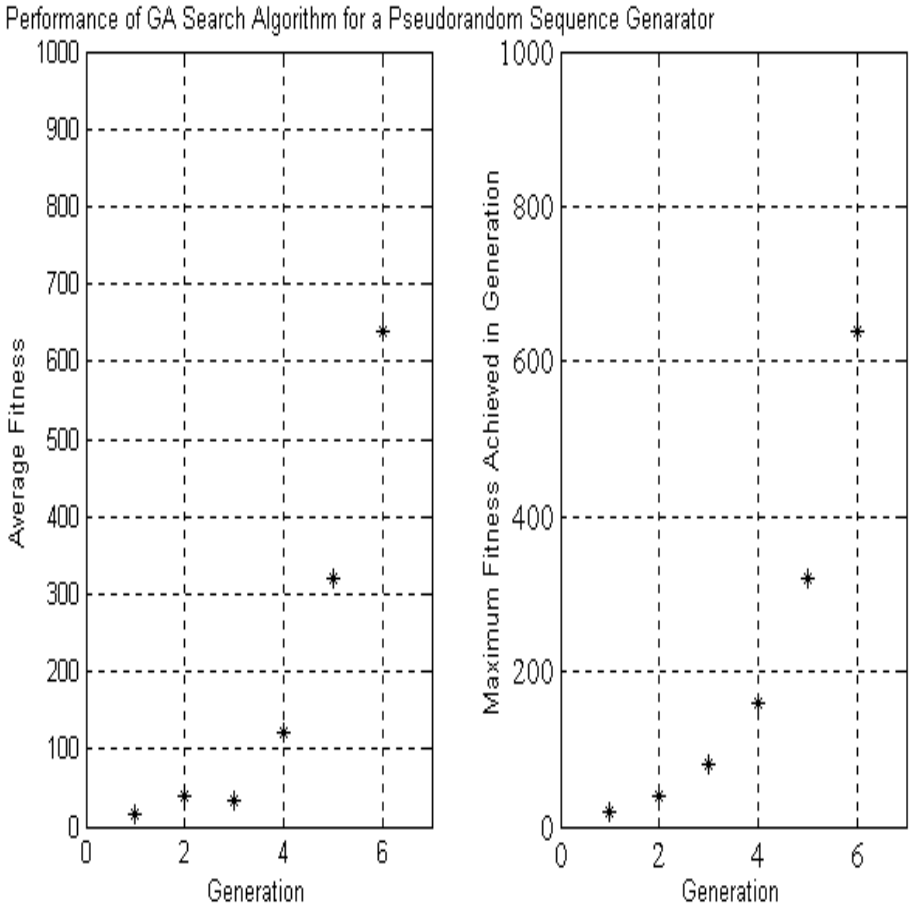

Figure 3.5 Performance of a genetic algorithm search for a 3-stage pseudorandom noise generator

The plot on the left, in Figure 3.5, shows the average fitness of the population of four chromosomes. The maximum fitness plot shows the individual with the maximum fitness in the population. Fitness function selected in this algorithm is $f_{i+1} = 2 * f_i$, where f_i is the fitness a chromosome in the current population being used to obtain the next population. In approximately three clock pulses, the above algorithm found the desired sequence generator on [1 0 1] which corresponds to the linear feedback shift register configuration shown in Figure 3.6 below.

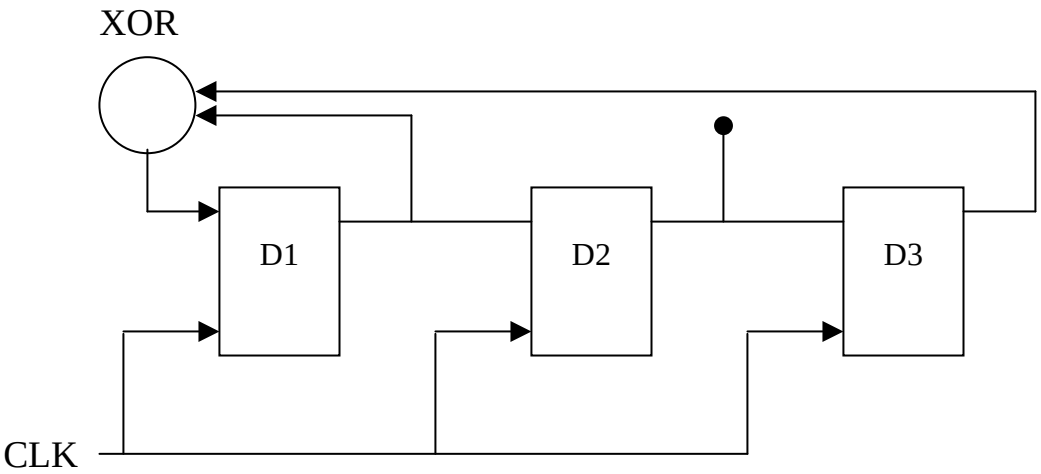

Figure 3.6 Pseudorandom sequence generator obtained by the genetic algorithm in 3 generations corresponding to sequence (3.5).

In the following section, we discuss a case of sequence learning where the frequency of symbols appearing in the sequence is learned through trial-and-error interactions similar to the n-armed bandit problem.

3.4 Learning the Frequency of Symbol Occurrence in a Sequence – The Single-State, N-armed Bandit

This section includes a simple pattern and serves as a basis for more complex sequences that will be discussed in the next chapter. In this problem, the sequence consists of a series of symbols composed of an alphabet of k individual symbols [15]. Symbols may occur in any combinations therefore there are a total of $2^k - 1$ symbols (not including the all-zero element) but only one combination occurs more frequently than all others. The goal is to determine which symbol or symbol combination occurs more frequently in the sequence. This is analogous to a non-associative, single-state, n-armed bandit problem where $n = 2^k - 1$. The non-associative assumption is made to stress the property that the choice of action at any stage does not depend on

the action(s) previously taken. Furthermore, this represents a binary bandit problem since the reward for selecting an arm is either high (correct prediction) or low (incorrect prediction). Selection of an action is analogous to transmitting a particular vulnerability detection test in our work where the cost of making a prediction is the use of bandwidth. Since the system does not know which of the symbols have the highest probability of positive return, it must start the learning process by taking each symbol at random. During the initial steps in our example, one of the choices of symbol will exhibit higher rewards but it will be premature to decide that particular selection as being the best selection to take since the best selection can be determined only after an infinite number of trials. It is known however that the selection with the highest current reward most of the times and selecting other symbols with a small non-zero probability, ε, will yield the best results. This selection policy is termed ε-greedy as previously discussed and in general all action selection policies that assign a non-zero probability of selection to all actions at all times are called ε–soft policies [37]. This is the problem of trade-off between exploitation of the current knowledge of action values and exploration

of other less rewarding actions. We used this method of action selection by sample-average [37] method in our problem to determine the most frequently occurring vulnerability. It should be noted that in this thesis we use terminologies common in reinforcement learning. *Policy* here is the rules of symbol selection, *action* denotes the choice of symbol selection, and *agent* refers to the dispatcher.

To illustrate the approach, we assume the symbol alphabet consists of three symbols (k = 3). We used exploration coefficient $\varepsilon = 0.05$ in 250 interactions. The results of the system's learning performance is shown in Figure 3.7. In the upper plot, the average reward per prediction is shown. The dispatcher (agent) in this case transmits a randomly selected symbol. We designed the simulator (network) such that no matter what the generated symbol is, the next symbol would be a particular symbol 'C' with the probability of 0.8. The agent learns in approximately 50 interactions that 'C' is the most frequently occurring symbol. The selection frequency of symbol 'C' is always slightly less than 0.8 due to the imposed exploration coefficient.

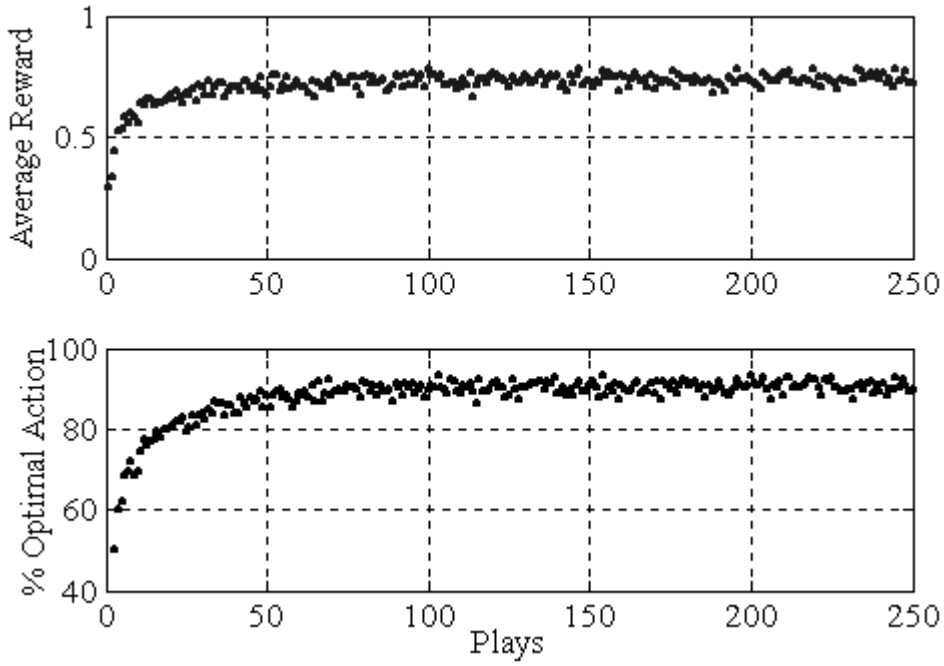

Figure 3.7 Performance of epsilon-greedy algorithm on the binary 3-armed bandit. The results shown are averaged over 400 tasks.

The lower plot shows the percent optimal action taken by the agent. Optimal action here is defined as the symbol selection that results in the highest possible reward. In this case, selection of the symbol 'C' is the best choice. The percent optimal action, however, never reaches 100 due to the exploration. In some problems, it is possible to set the exploration coefficient such that its value decreases with time so that in the limit, the agent's policy will become deterministic. This

example sets the foundation for the next sections in that we use an algorithm based on reinforcement learning methods to solve non-associative sequence problems. The reward function r(a) we assigned to the agent was r(a) = 0, or 1, depending on whether the selection of symbol resulted in a match with the symbol generated by the simulator. The average reward associated with a particular action is Q(a), a sample-average value of the action.

Chapter 4

Reinforcement Learning - Based Algorithm for Vulnerability Assessment

As discussed in the previous chapters, vulnerability detection is a problem that can be represented as a discrete sequence prediction. As such, it is possible to devise various methods based on reinforcement learning that can be applied to the sequence learning problem. In Chapter 3 we discussed straightforward techniques to learn various special cases of sequence prediction.

Here we generalize the sequence learning problem to situations where symbols in the sequence occur is more complex patterns and introduce

our reinforcement learning-based algorithm to learn discrete sequence patterns for various types of patterns [30][32].

Such patterns involve non-associative search algorithms because if a particular symbol occurs with some statistic regularity in relation to another symbol, then the choice of a symbol may depend on what symbols occurred previously. For example if in a sequence of k symbols, symbol s_i occurs- after another symbol s_{i-1} has occurred- with a probability greater than 0.5, then the next time s_{i-1} occurs, the agent's selection will be s_i.

4.1 Agent-Environment Interaction in Sequence Learning

In our approach, the agent interacts with the environment in the following manner: The agent first takes an action a_i (i.e., selects a symbol s_i as its prediction) then observes the network for response. If the response is positive, the prediction was correct and the agent receives a high reward. Also the agent becomes aware of the state of the environment, that is, it knows now that the environment "contains s_i." For simplicity, we call this state as "*state of* s_i", or simply "s_i."

The agent makes a note of this state and selects its next action, a_{i+1}. If the next action gives rise to a positive response (resulting in a high reward), the agent knows that from this particular state, a particular action has returned a high reward. The agent will remember the state-action pair (s_i, a_i) as a desirable, high value situation. If, however, the agent's selection does not give rise to a positive response, the reward received is low and the agent considers the state-action pair as a low value one and will try another action when encountering that state in the future.

In our approach, we have expressed an inherently continuous problem in terms of an episodic problem. Although the agent interacts with the environment in a continuous fashion, the learning process is interrupted when the agent makes an incorrect selection. This will bring the agent to the *terminal state* and an end to that particular learning episode. The following will elaborate further on our discussion:

An important point to consider is that a negative response only means that this particular symbol does not exist and says nothing about the existence of any other symbol. We call this situation *a terminal state*.

The terminal state marks the end of the learning episode. Learning from actions will no longer be possible since the state from which an action is taken is not known to the agent. From the terminal state, the agent must continue selecting actions until one of the actions returns a positive response. Only then the agent has entered a known state and can now select actions and update its state-action values: a new learning episode has started.

As the number of interactions increases and the agent maintains an exploratory policy, the process in the limit will converge to a set of state-action pairs that will be optimal. The agent maintains a look-up table of state-action values that it has accumulated and continually improved (i.e., updated) upon every interaction from a known state. A diagram depicting this interaction process is shown in Figure 4.1.

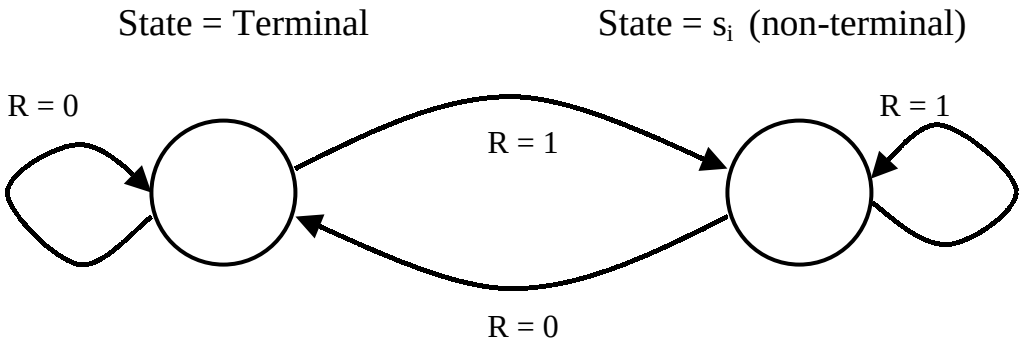

Figure 4.1 Agent's State transitions upon high-reward or low-reward actions

4.2 Simulator Design to Model the Environment

We will discuss the algorithm's performance in learning various types of discrete sequences. In our work, we wrote our simulations in Matlab where the symbols are coded as binary vector (See Appendix A). For a symbol alphabet of n, the symbols and their combinations are n-topple binary vectors as shown in Figure 4.2.

Symbol	Vector Representation of Symbol
a	[1 0 0 0 … 0]
b	[0 1 0 0 … 0]
{a,b}	[1 1 0 0 … 0]

Figure 4.2 Vectors representing symbols in the sequence

For example, we considered a sequence in which all symbols occurred with equal probability except that when a symbol *a* occurs, the next adjacent symbol is *b* with a given probability > 0.5. Examples of such sequences are given in Figure 4.3 (alphabet = 10).

0 1 0 1 0 1 0 1 0 0
0 0 1 0 1 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

Figure 4.3 The first 10 symbols with [a,b] occurring with pr. = 0.9.

4.3 Reinforcement Learning- Based Algorithm to Detect Multiple Patterns

In order to elaborate on the algorithm developed here, we consider a simulation that generates a sequence of symbols made up of an alphabet of choice (five, ten, etc.). It is then assumed that a symbol s_i occurs in the sequence with the probability of Pr > 0.5 and when it does, it is always followed by a symbol s_{i+1}. The learning task is to detect this pattern thereby increasing the likelihood of correct predictions.

The algorithm works in this way:

- Initialize a look-up table, the table of state-action values to zero; start from the *terminal state*
- Select an action from the symbol set with uniform probability
- If action == simulator, mark start of episode:
    - Continue ε-greedy action selection according to Sarsa or Q-learning rules; keep updating the look-up table of state-action values

- If new action ~= simulator, start over from the *terminal state*
- Repeat until end of interaction, or until state-action matrix values have converged.

We performed the algorithm using both Sarsa and Q-learning variations and our results showed a slight improvement in performance using Sarsa method [30]. The Sarsa method allows for more exploration, which is consistent with the observations made from other research [37]. Although Q learning generally results in faster convergence, it explores less than the Sarsa method. The Matlab code for a Sarsa-based algorithm is provided in appendix B.

We discussed the two methods of Q-learning and Sarsa in Chapter 2 and provide the update rule for these approaches again in Equations (4.1), (4.2) respectively.

$$Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max Q(s',a') - Q(s,a)], \quad (4.1)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma Q(s',a') - Q(s,a)], \quad (4.2)$$

In another example, we used a three-symbol simulator with symbols A, B, and C and all combinations. To emphasize on the problem of bandwidth limitation, we restricted our action selection to one symbol at a time. Actions that resulted in high reward obtained +1 and those with low rewards received –3. The simulator produced symbols with the probability shown in Table 4.1 that shows the symbol transition probabilities. It is assumed that the simulator makes one symbol transition at a time. Here, the simulator starts transitions from empty state (no transition, Φ) and moves to state C with the highest probability. Symbol B occurs most often after A. Finally, when B occurs, it tends to be repeated. We have also used a one-step learning approach (TD(0)).

Pr.	Φ	A	B	C	AB	AC	BC	ABC
Φ	0.0	.15	.15	**0.7**	0.0	0.0	0.0	0.0
A	0.05	0.05	0.0	0.0	**.85**	0.05	0.0	0.0
B	0.1	0.0	0.7	0.0	0.1	0.0	0.1	0.0
C	0.1	0.0	0.0	**0.7**	0.0	0.1	0.1	0.0
AB	0.0	0.05	**.85**	0.0	0.05	0.0	0.0	0.05
AC	0.0	0.1	0.0	0.1	0.0	0.1	0.0	**0.7**
BC	0.0	0.0	.45	.45	0.0	0.0	0.05	0.05
ABC	0.0	0.0	0.0	0.0	0.2	0.2	0.5	0.1

Table 4.1 Pattern of Symbols Generated by the Simulator

The step size parameter $\alpha = 0.1$, discount factor $\gamma = 0.9$, and a decaying exploration coefficient $\varepsilon = 1/t$, where t is the number of times a particular action is selected. The look-up table of state-action values obtained by Sarsa and Q-learning approaches are shown in Figures 4.4 and 4.5 below.

Q(s,a)	Action A	Action B	Action C
State ‘T’	0	0	0
State ‘A’	-0.2900	**0.2144**	-0.3544
State ‘B’	-2.1800	1.8888	-2.3223
State ‘C’	-2.2092	-1.0703	**1.8569**

Figure 4.4 State-action Values, Q(s,a), Learned by Q-Learning variation of the algorithm in 1000 steps

Q(s,a)	Action A	Action B	Action C
State ‘T’	0	0	0
State ‘A’	0.0638	**1.0271**	-1.9862
State ‘B’	-2.4215	0.0129	-1.4747
State ‘C’	-2.9194	-2.5794	**1.9574**

Figure 4.5 State-action Values, Q(s,a), Learned by Sarsa Variation of the Algorithm in 1000 Steps

In both approaches, the algorithm successfully learned to select symbol C most of the time as indicated in bold letters. This is shown in agent's look-up table having the largest value for action "C" when in state "C". Furthermore, the agent has also learned that when in state "A", its best action is selecting the symbol "B" as expected. In another variation of the above pattern, we let the simulator produce a series of symbols make up of an alphabet of five.

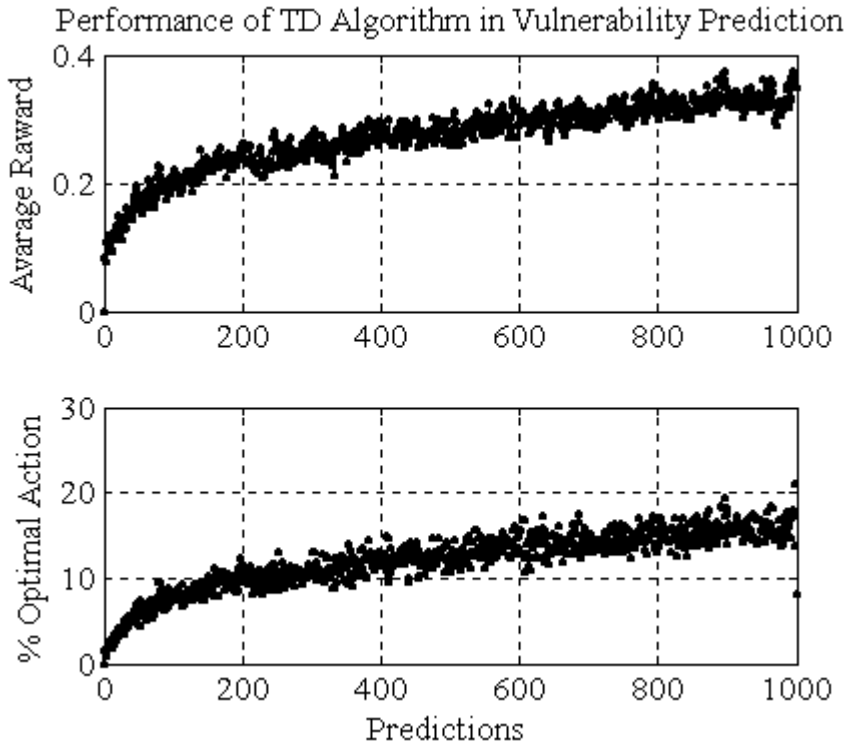

Figure 4.6 Learning a pattern involving a symbol pair.

We let a particular symbol pair (s_i, s_j) occur with the probability Pr = 0.6. The symbol alphabet is five. The number of interactions as shown in Figure 4.6 is 1000. The plots shown are averaged over 400 tasks. The exploration coefficient used is 0.1. The lower plot shows the percent optimal actions. Optimal action in this problem is defined as the action that the agent would take if it had a prior knowledge of the sequence patterns. Alternately, the optimal action here can be considered as follows: If the first action is s_j when the previous action was s_i, then the action is optimal. The plots in Figure 4.6 are interpreted as follows. In the entire sequence the pair (s_i, s_j) occurs with a probability of 0.6. Consequently each the symbol s_i occurred with Pr. = 0.3. As shown in the lower plot, the agent's percent optimal action converges to slightly higher that 15. This represents correct selection of actions slightly more frequently than 30 percent of the time, as expected. It exceeds Pr. = 0.3 because at times, the agent accidentally selects an action that matches the simulator output even though it's not s_i. Similarly, the upper plot shows average reward improving to converge to values slightly better than 0.3. The agent's prediction increases as the number of interactions (predictions and

rewards) with the environment increases. The same experiment is shown in Figure 4.7 but the symbol alphabet is increased to 10.

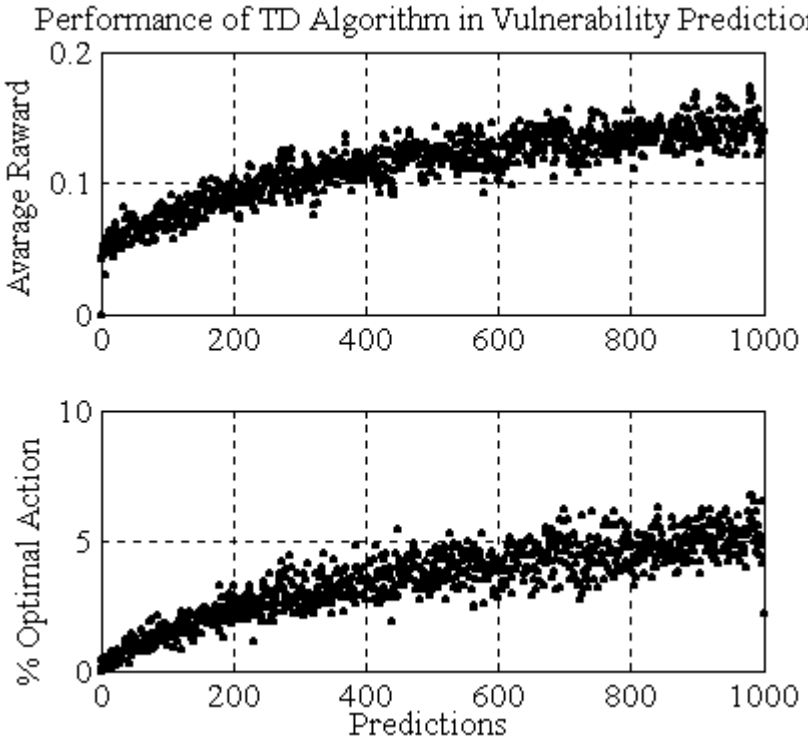

Figure 4.7 Learning a pattern involving a symbol pair with symbol alphabet of 10.

As seen in Figure 4.7, average reward and percent optimal actions are reduced greatly compared to Figure 4.6 since the number of possible

symbols to choose from is increased. The learning rate however improves as the number of predictions increases.

An interesting variation of above symbol-pair pattern is when a symbol s_i occurs randomly, but when it does occur, it is followed by another symbol s_j with a certain probability. This is more difficult than the pattern mentioned previously because the two symbols do not occur in adjacent time deterministically. Such a pattern is harder to detect and as Figure 4.8 shows, our formulas for computing average reward and percent optimal actions did not reveal learning by the agent.

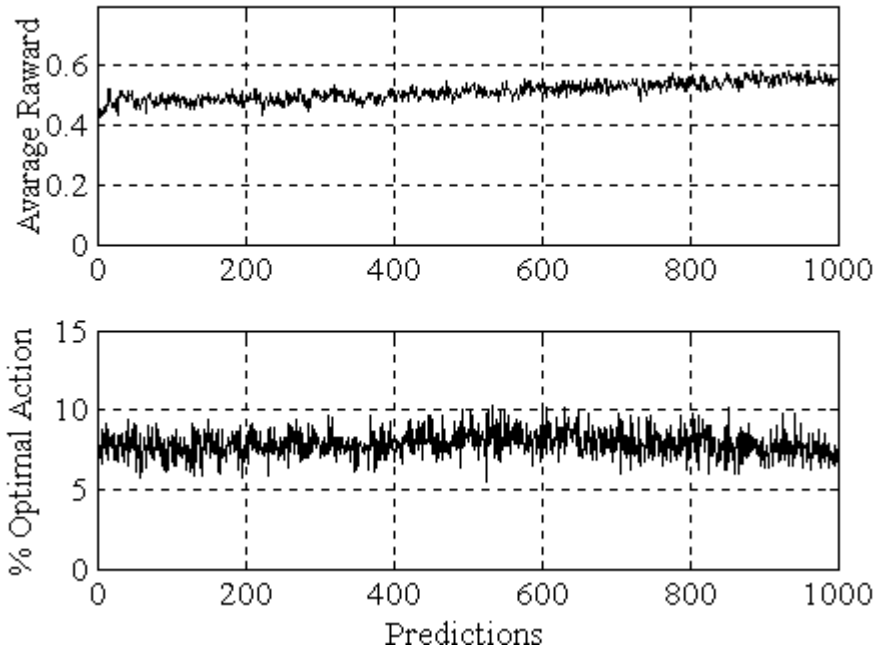

Figure 4.8 Prediction results for a difficult pattern showing an apparent lack of improvement in learning

A closer inspection at the agent's performance, however, shows that in the limit, the agent's state-action values in its look-up table demonstrates that the learning objective is accomplished. This is shown in Figure 4.9.

	S1	S2	S3	S4	S5
State S1	1.4664	1.8812	1.7668	1.5011	2.7597
State S2	1.5851	1.5984	0.9737	1.6029	2.9328
State S3	1.4243	1.9212	1.6616	1.8746	2.5791
State S4	1.1918	1.0634	2.0726	2.4542	**3.2788**
State S5	1.3728	1.8293	1.2956	1.8201	2.7156

Figure 4.9 Matrix of state-action values (look-up table) generated by the agent showing prediction performance: Symbol s_5 achieved highest value when in "State of s_4."

In the experiment shown in Figure 4.9, we let all symbols occur in the sequence with equal probability except that when symbol s_4 occurs, the next symbol is s_5 with probability of 0.6. The agent achieved the matrix values using the Sarsa-based learning in 20,000 interactions. Rows one through five represent the five states in this problem (symbol alphabet is five). The terminal state is not shown since state-action values in the terminal state are by definition zero. The columns represent possible action that can be taken from a state.

4.4 Comparison of Vulnerability Detection with General Sequence Prediction

In this section we show a comparison of our network problem with the general discrete sequence prediction, as mentioned in Chapter 2. The vulnerability prediction differs from general sequence prediction in that the symbols generated by the sequence are not visible to the agent. The agent first has to probe the network by transmitting a vulnerability test. If the particular vulnerability exists in the network, the prediction is correct. Otherwise the agent remains unaware of the state of the network [30]. This is equivalent to a sequence having generated a symbol, but the agent not being able to *see* it. If the prediction is not correct, the agent has no knowledge of what the generated symbol might be. Figure 4.10 shows the advantage of seeing the symbol by the agent in a general sequence prediction problem.

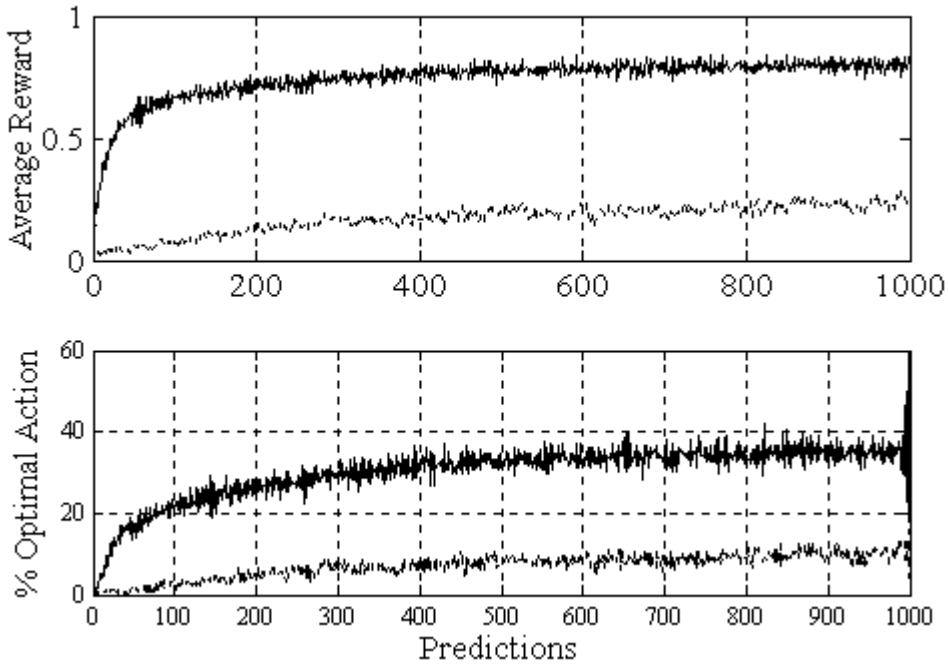

Figure 4.10 Comparison of vulnerability prediction (lower values) with general sequence prediction (higher values)

The initial probing of the network (to find vulnerability) creates additional problem for the agent and greatly reduces the speed of convergence while increasing the overhead cost in the prediction task. This problem is not encountered in a general sequence prediction. As shown in Figure 4.10, both the average reward and percent optimal action obtained by the agent are higher for the case of general sequence prediction. In this example, the frequency of occurrence of

the pattern (s_4, s_5) pair is set to 0.8. For example in the upper plot showing average reward in each action, the general sequence prediction task converges to 0.8 as expected, whereas the average reward obtained in the vulnerability prediction converges only to slightly higher than 0.2. The slower learning rate, as explained above, is due to the fact that the failed vulnerability predictions do not contribute to any learning improvement. The results shown are averages obtained by repeating the prediction task 500 times.

In this chapter, we described our reinforcement learning-based algorithm and showed the performance of this algorithm as applied to various reinforcement learning techniques such as Sarsa and Q-learning. In the next chapter, we will describe, for comparison, another approach to vulnerability prediction problem using evolutionary methods. We will compare the performances of these two methods and show that in our particular problem where the constraints in bandwidth is the most significant resource limitation, the reinforcement learning approach is a more suitable solution to the prediction problem.

Chapter 5

Vulnerability Detection Using Evolutionary Computation Methods

Genetic and evolutionary algorithms have been a popular approach to learning different cases of sequence prediction. In this chapter, we elaborate on a genetic algorithm–based approach to sequence prediction. We show various modifications that needed to be made to the genetic approach in order for it to be applicable to our vulnerability detection problem.

Genetic algorithms can be used in various ways in the sequence prediction problem [20]. For instance, a genetic algorithm can be developed to search for the most effective agents for a given prediction task. In a genetic search problem, a reinforcement learning

agent can be described by its genetic make up: exploration coefficient, step-size parameter, learning rate, the update rule, etc.

Our approach to sequence prediction using genetic algorithms is described and results presented in the following sections.

5.1 GA System Parameters

We considered a population of chromosomes each having a different symbol prediction property. In each time instance, the individuals make a prediction and observe the outcome. Depending on whether the prediction was correct or incorrect, a *partial* fitness value is assigned to the gene responsible for the first prediction. The second prediction is made and a fitness value is generated for the second gene responsible for making this prediction. This process is repeated until all genes in the population have made their predictions and the total fitness for all chromosomes in the population is computed. The next population is then derived using genetic operators of cross-over, mutation, and reproduction. In this approach, the individual's life span is equal to the number of genes which in turn equals the number of predictions. If we are interested in finding a pattern in a sequence

of a certain number of symbols, (say, 5) we would construct a population of chromosomes composed of five genes responsible for making five consecutive prediction. Figure 5.1 shows the genetic structure of such a population.

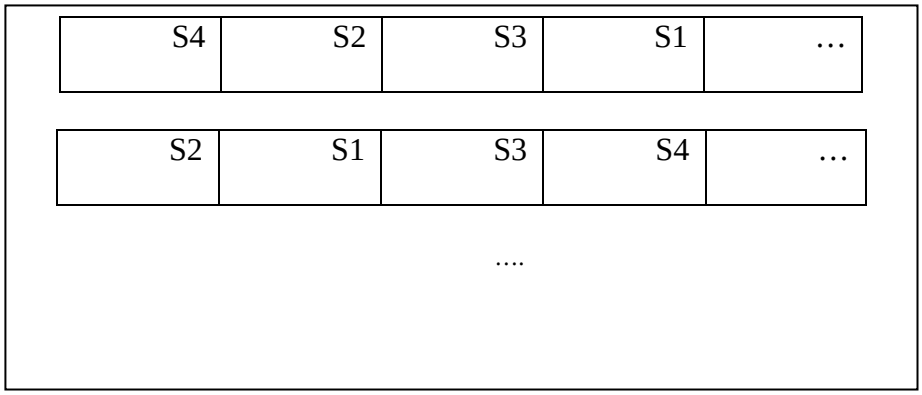

Figure 5.1: A population composed of individuals with various prediction policies according to their genetic make-up.

Here, chromosomes made up of a sequence of genes represent the members of the population. Each gene can be thought of as the individual's genetic disposition to make a particular prediction. In the example below, the prediction tendency is deterministic although this need not be the case. For example, the chromosome on the top of the figure predicts symbols in S4, S1, S2, S3, …. The final population

will be composed of a majority of individuals with the gene arrangement corresponding to the desired decision making policy.

Figure 5.2(a) shows an initial population for at the beginning of a sequence prediction problem, consisting of genetic arrangement uniformly distributed in the chromosome. Here we consider a simplified case by creating populations to learn pattern in two consecutive symbols.

Initial Population of 20 Chromosomes																			
1	3	5	3	1	5	5	5	5	3	3	3	2	3	4	4	3	2	2	3
2	4	2	1	5	1	5	4	5	1	5	1	3	3	2	5	1	2	2	4

Figure 5.2(a) A randomly generated initial population of two-gene chromosomes.

After the learning evolutionary process, a final population is shown in Figure 5.2(b), showing a population predominantly made up of chromosomes with the desired genetic arrangement. Note the first two genes are mostly composed of (b a) symbols, since that was how the sequence generator was designed.

Final Population Chromosomes																			
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	5	1

Figure 5.2(b) An example of the final population after 1000 generations interacting with the simulator generating symbols with (2 1) appearing with the probability of 0.9.

5.2 GA-based Methods and Performance Results

The simulator in this problem generates a sequence of symbols from alphabet of five. A pair of symbols occurs with probability of 0.8 throughout the sequence. The 2-gene population is created to learn patterns involving two consecutive symbols. This example can be expanded for cases where learning patterns involving more symbols are of interest. The population reaches a convergence state when greater than half the population consists of similar chromosomes. The genetic arrangement of this majority will determine the pattern generated by the simulator. The evolutionary code is provided in Appendix C. In Figure 5.3, after approximately ten generations, the

majority chromosomes immerge when the ratio of fit chromosomes reached 0.5. The ratio of fit chromosomes never reaches the ideal 0.8 due to mutation rate. Incorporating a mutation rate is necessary to prevent the population to prematurely reach a sub-optimal population. The plot on the right hand shows average generation fitness where the fitness ranges from zero to 100.

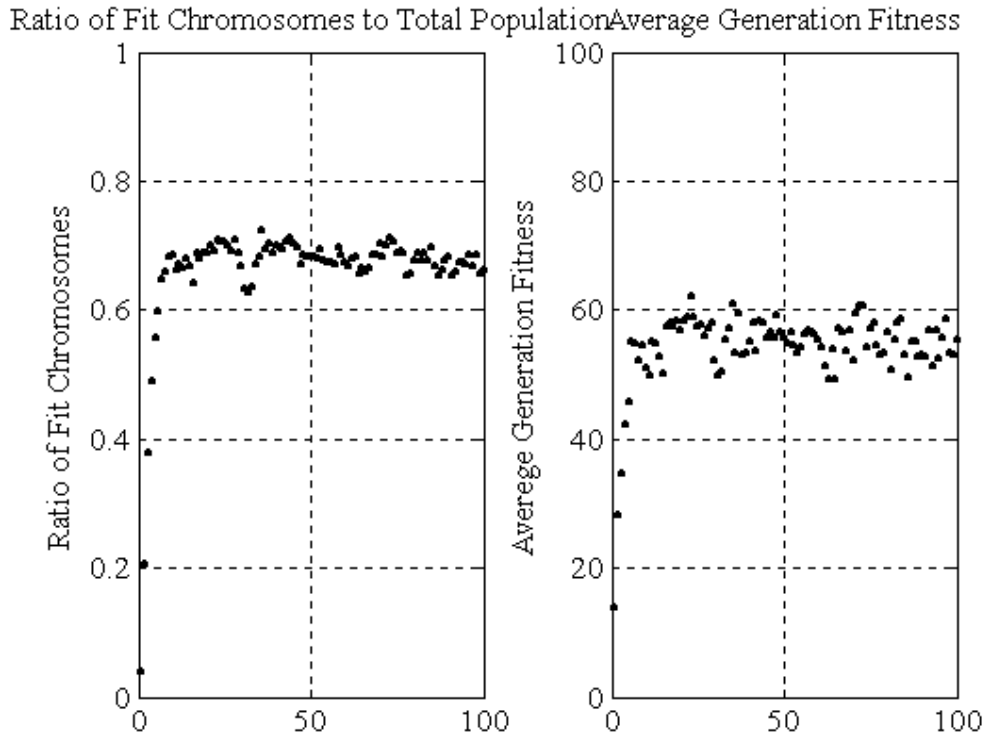

Figure 5.3 Performance of population composed of 2-gene chromosomes. Symbol alphabet = 5, Population = 75, symbol-pair occurrence probability = 0.8.

Increasing the population size of course improves the learning performance of the genetic algorithm. Using the same parameters as above, if we use a population of 10 chromosomes we'll get reduced performance as shown in Figure 5.4.

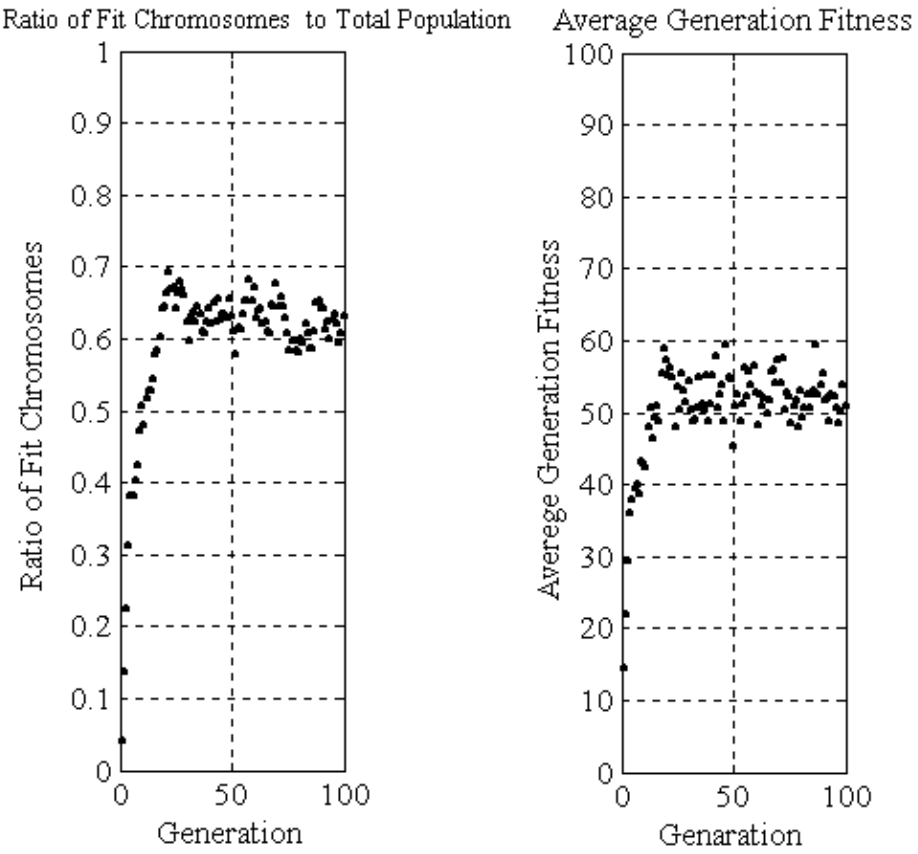

Figure 5.4 Performance of genetic algorithm with population size reduced to 10.

We needed to make modification to the algorithm so that it could be applied to our vulnerability detection problem. In order to consider bandwidth limitation, we had to make sure that genes in a chromosome always represent a single-symbol. This resulted in

special issues in the design. Some of these problems are explained in the following section.

5.3 Observations on Cross-over, Mutation, Gene Representation

We originally implemented a binary vector gene representation. Each gene was represented by the symbol it would predict. In this way, an individual would be represented as shown in Figure 5.5.

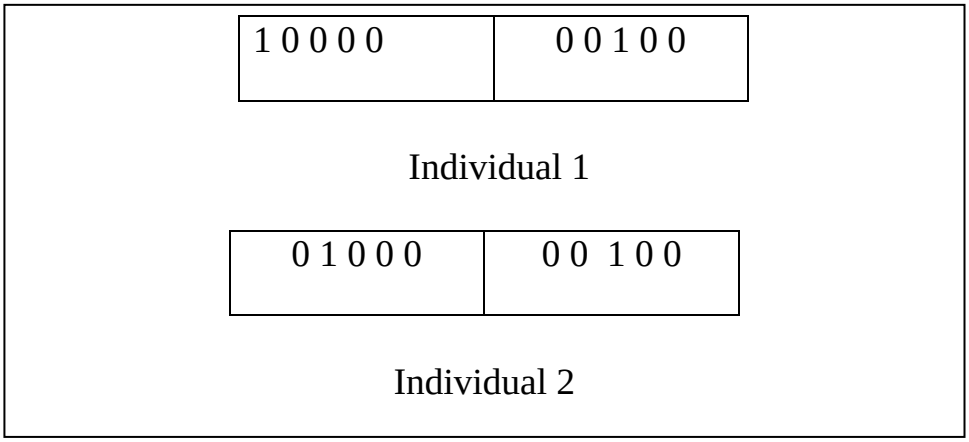

Figure 5.5 Sample individuals from a population.

In the process of cross over, for example, a new gene may be created such as:

0 1 1 0 0

Such a gene represents a prediction of more than one symbol. This violates our limitation of symbol prediction to one since its use of bandwidth will be excessive. To correct this problem, we represented genes as decimals. The operation of *cross-over* was therefore revised to *recombination*. A similar problem was encountered with *mutation*. Another problem created by mutation operator was that a gene that was outside our symbol alphabet might be created. For example, in symbol alphabet of ten, where genes have any value between one and ten, mutation might create a "11" gene. We therefore modified the mutation operation to assure that before going to the next population, all genes will be single-symbol prediction genes. Our mutation code is presented in Appendix D. These modifications were necessary to make a correct comparison between our reinforcement learning-based algorithm and the genetic-based algorithm in this work.

5.4 Advantage of Reinforcement Learning – Based Algorithm over the Genetic Algorithm Approach

The most important advantage of our reinforcement learning – based algorithm in sequence prediction is that the agent achieves learning through single-symbol interactions [23]. This was the desired method of learning because it was most sensitive to our bandwidth constraint. The genetic algorithm – based approach, however, needs to allow all individuals in the population to interact (transmit tests) with the environment in order to determine the fitness of all individual. This extensive interaction must be performed at every instance of network observation. At a minimum, there must be enough individual interactions at each step to allow all individuals with distinct genes to interact. (Therefore the minimum number of interactions at each step would be equal to symbol alphabet.)

In the next chapter, we compare the various algorithms' performance with respect to transmission noise.

Chapter 6

Effects of Transmission Noise on Algorithm's Performance

In order to evaluate the performance of algorithms discussed previously in a practical network situation, we introduced communication noise into the sequence [23]. We created a deterministic and simple sequence of alternating zeros and ones so that the effect of noise on the performance would be more easily observed. The noise introduced is additive white Gaussian with zero average value. We further made observations on the effect of various design parameters in both reinforcement learning (the exploration coefficient) and genetic algorithm (population size, mutation coefficient) on prediction performance. The results are provided in

the following sections. Furthermore, we measured prediction performance under various signal-to-noise values.

6.1 Effects of Noise on the Reinforcement Learning-Based Algorithm

Figures 6.1 and 6.2 show the performance of the reinforcement learning agent under various signal-to-noise values. Since the simulator pattern is perfectly ordered, exploration in higher S/N ranges is unnecessary and in fact it is a disadvantage. The plot for zero exploration ($\varepsilon = 0$) shows the lowest bit error rate value in the S/N > 5 range, as expected. This is expected since for deterministic symbol pattern, there is no need for exploration and the performance will be at its optimum. For lower S/N however, exploration of 0.1 results in slightly better bit error rate. For noisier signals, exploration results in improvement in performance since it leads to finding more optimal action selection.

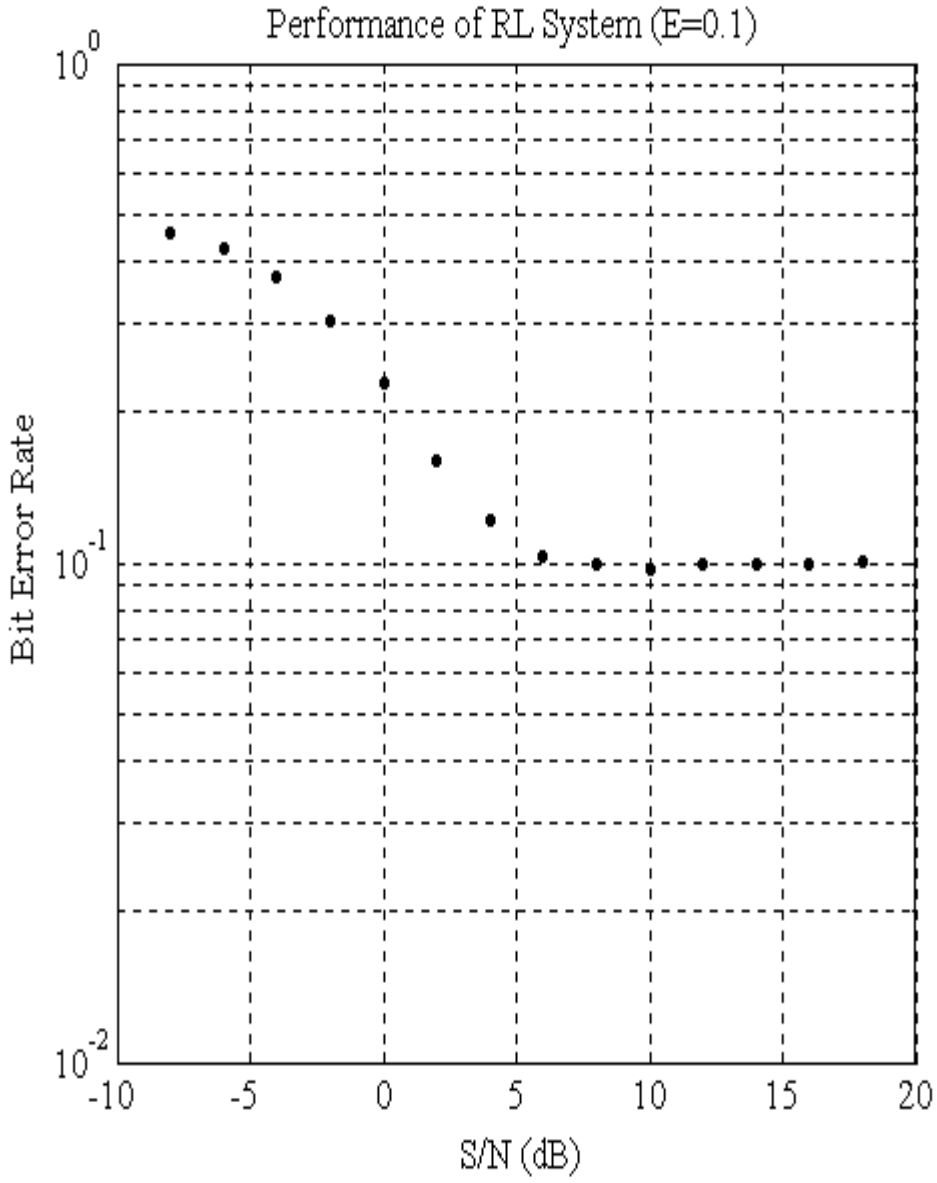

Figure 6.1 Performance of reinforcement learning algorithm in presence of noise: exploration coefficient's effect on prediction error. Exploration coefficient = 0.1.

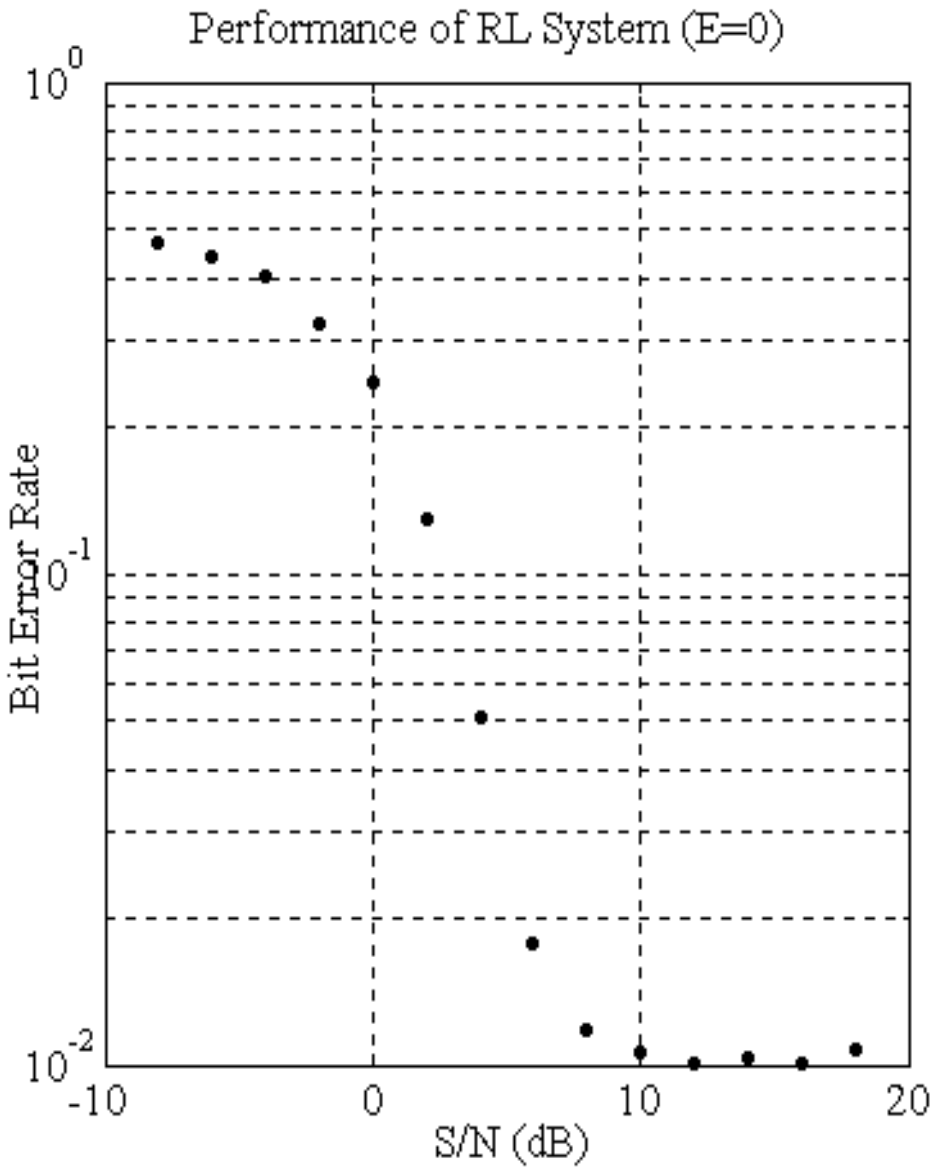

Figure 6.2 Performance of reinforcement learning algorithm in presence of noise: exploration coefficient's effect on prediction error. Exploration coefficient = 0.0.

6.2 Effects of Noise on the Genetic Algorithm – Based Prediction Method

The genetic algorithm-based system's performance is shown in Figures 6.3 and 6.4. As in the above case, the effect of the main parameters of the genetic algorithm system, such as population size, is shown for various signal-to-noise values.

As expected and demonstrated in these examples, the lower mutation rate results in better (lower) error rates particularly when S/N values are high. This is evident by inspection of figures 6.3 and 6.4. Mutation rate in genetic algorithm-based approach in this case is analogous to exploration ratio in reinforcement learning algorithm. Mutation rate's effect on performance with respect to noise levels can be seen in Figures 6.1, 6.2, 6.3, and 6.4.

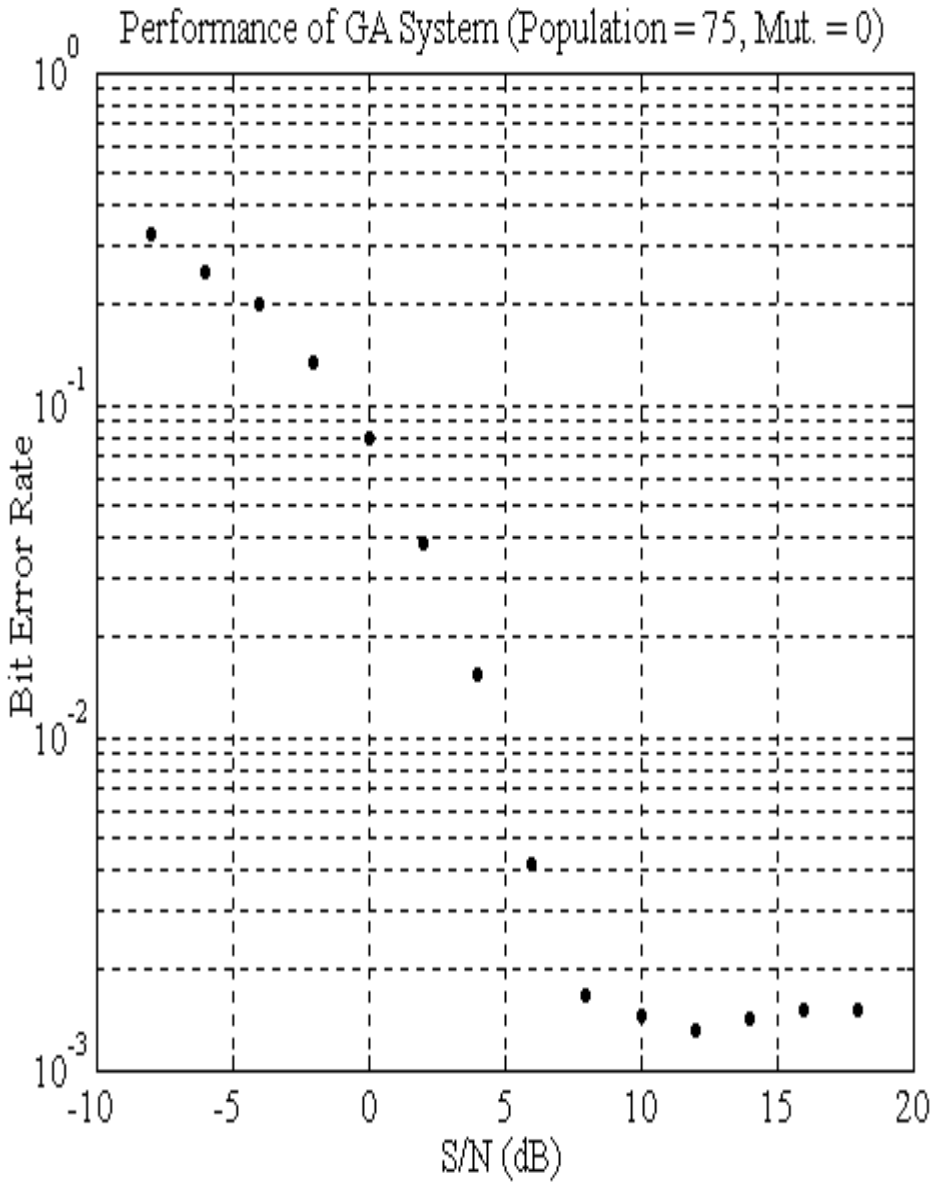

Figure 6.3 Performance of genetic algorithm system in presence of noise: Comparison of Mutation rate's effect on prediction error (population = 75, mutation rate = 0.0).

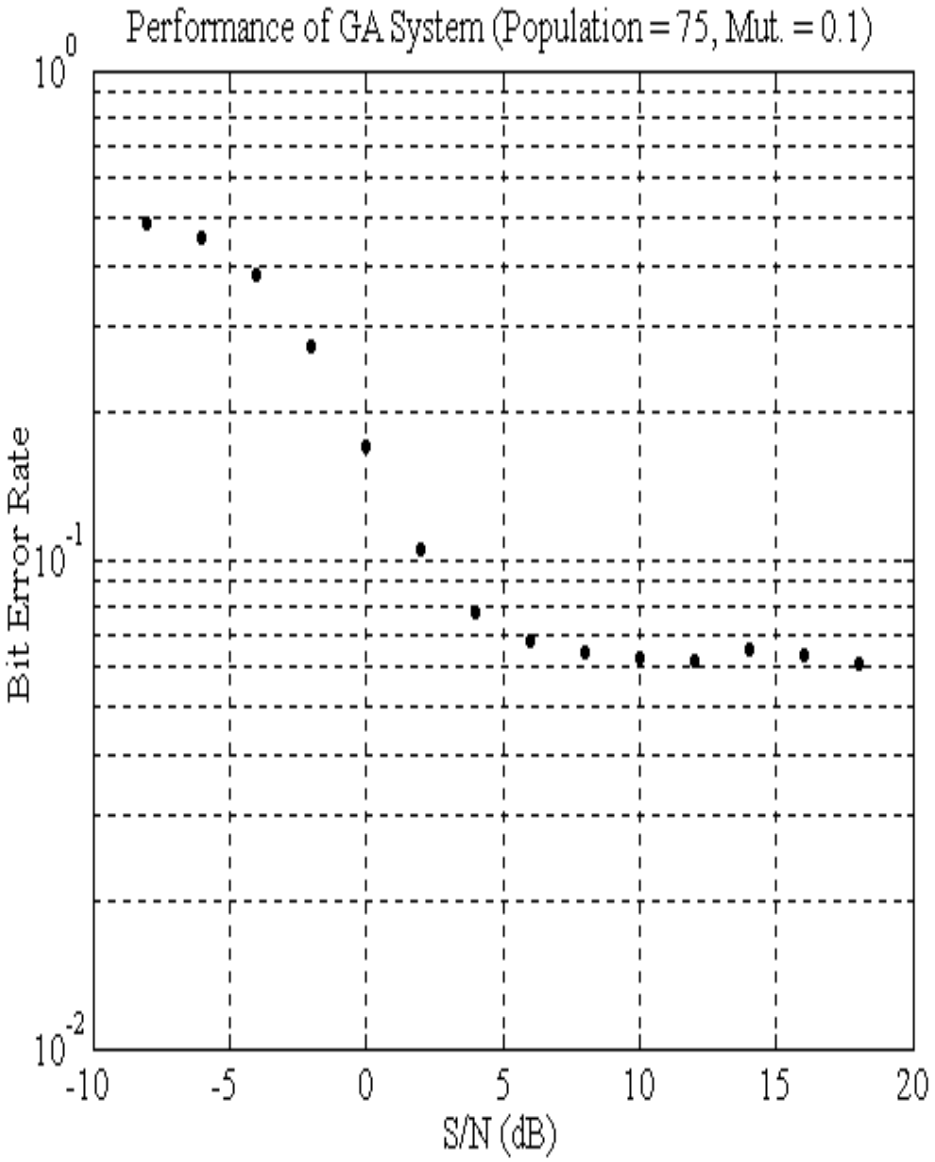

Figure 6.4 Performance of genetic algorithm system in presence of noise: Comparison of Mutation rate's effect on prediction error (population = 75, mutation rate = 0.1).

The above experiment was also performed using a smaller population. The performance results are shown in Figures 6.5 and 6.6 below.

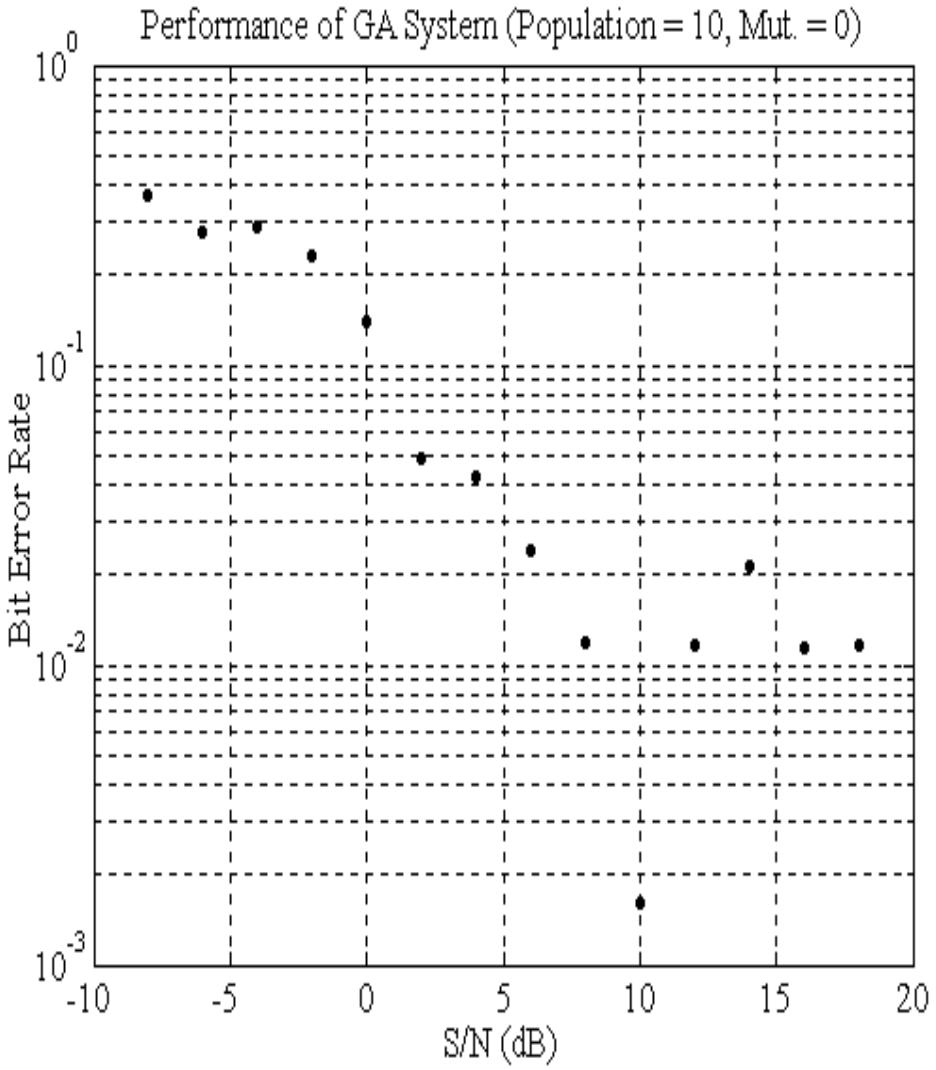

Figure 6.5 Performance of genetic algorithm system in presence of noise: Comparison of mutation rate's effect on prediction error (mutation rate = 0, population = 10).

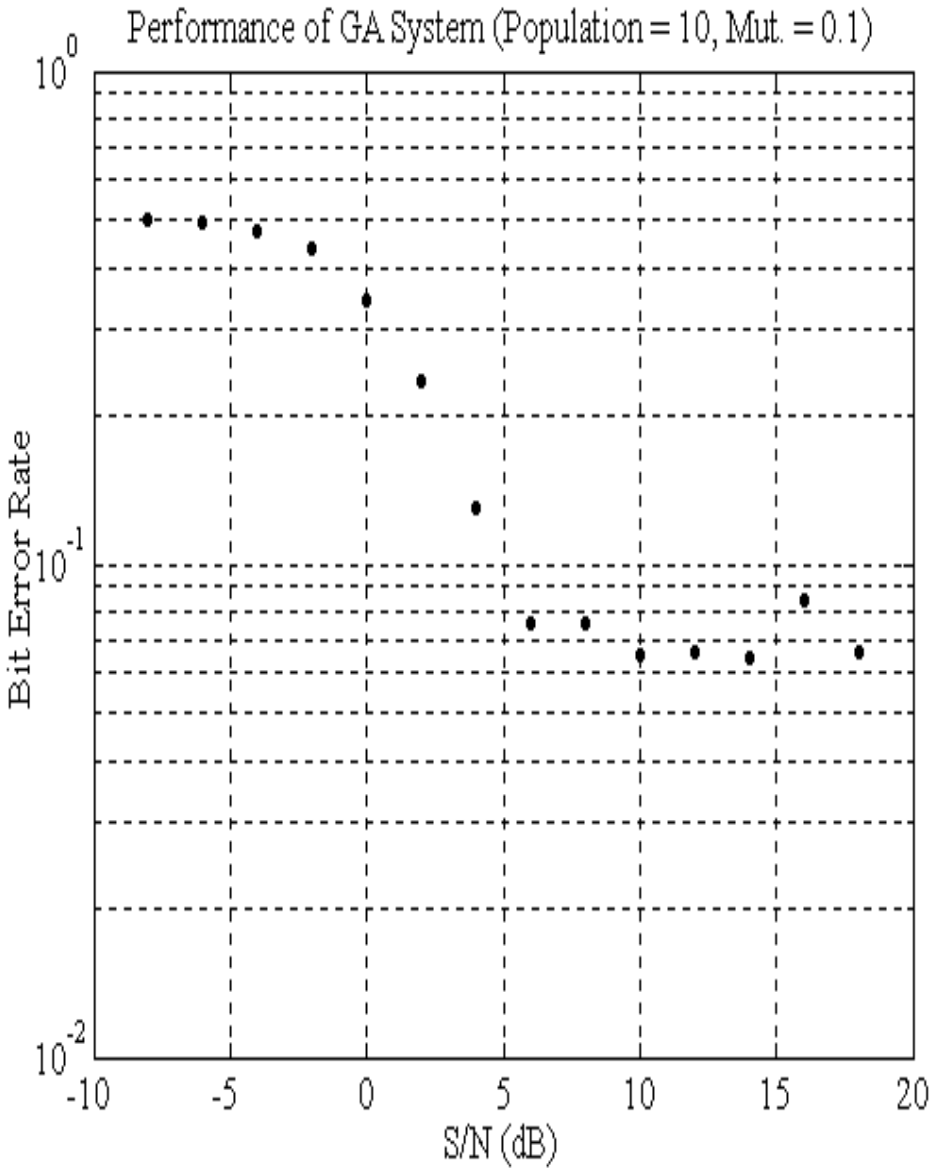

Figure 6.6 Performance of genetic algorithm system in presence of noise: Comparison of mutation rate's effect on prediction error (mutation rate = 0.1, population = 10).

Higher population size generally improves error rates as shown in Figures 6.3 and 6.4. Also as expected, when S/N values are high, the lower mutation rate results in better (lower) error rates.

Chapter 7

Conclusions and Future Research

We showed in Chapter 4 two different approaches to sequence prediction, which exhibit learning through interaction with the network. Although both algorithms can be used to detect sequence patterns, bandwidth constraints in the wireless network application makes the reinforcement learning approach the preferred method. This is because learning is achieved through interaction of one entity, the agent, with the environment at each learning instance. In the genetic algorithm case, however, multiple members of a population may need to interact with the network in order for the fitness of the entire population to be determined. Some areas of improvement to the algorithms presented here are apparent. For example, the fact that vulnerability sequence is invisible to the dispatcher introduces a great reduction in performance efficiency since only correct predictions

contribute to learning. This accounts for low percent optimal actions experienced by the agent [23]. All predictions that return a negative response from the network result in fruitless use of bandwidth as well as other resources. One obvious remedy to this limitation is that after the incorrect prediction has been made, the agent's next selection be based on which symbols have returned positive response, regardless of the state from which the symbol was selected. Alternatively, the terminal state can be redefined to represent a state from which and all actions are evaluated in a look-up table. In this way, the agent's symbol selection is never random. Another area of future work is in developing algorithms that learn patterns that might exist between a large number of consecutive symbols. Such algorithms will undoubtedly require a greater number of interactions with the network and consequently result in a greater cost of learning. Research must be concentrated on ways to reach optimal learning stage in as few interactions as possible.

There remain some open questions at theoretical as well as experimental levels. We need to acknowledge some of these issues in this work.

7.1 Open Theoretical Questions in Reinforcement Learning – Based Methods

The ever present question in reinforcement learning problems is the dilemma of trade-off between exploration and exploitation. How do we devise an algorithm that will efficiently find the optimal value function? The process of learning is described as the process of improving an approximation of the optimal value function by incrementally finding a solution to this set of equations. Exploration is defined as intentionally choosing to perform an action that is not considered the best for the express purpose of acquiring knowledge of unseen states. In order to identify a sub-optimal approximation, state space must be sufficiently explored. More discussions of the issues of efficient exploration can be found in Thrun's work [32].

Another open area of research in reinforcement learning as related to our work is the efficiency of bootstrapping. Bootstrapping is the most important aspect of temporal-difference (TD) learning which our algorithm is based upon. Bootstrapping TD methods have been

shown empirically to learn substantially more efficiently than Monte Carlo methods which have no bootstrapping. (TD methods update estimates of the values of states based on estimates of the values of successor states. That is, they update estimate on the basis of another estimate. This is known generally as Bootstrapping.) In many problems performance of TD methods has been better than the Monte Carlo methods and various analytical results have been shown to support this idea [39], but only for particular tasks and initial settings. Therefore, we have a range of results that suggest that bootstrapping TD methods are generally more efficient than Monte Carlo methods, but no definitive proof. While it remains unclear exactly what should or could be proved, it is clear that this is a key open question at the heart of current and future reinforcement learning research.

7.2 Evolutionary Methods in Searching for Optimal Reinforcement Learning Agents

Genetic algorithms can be used in various ways in the sequence prediction problem as discussed in this thesis. We believe a genetic

algorithm can be developed to search for the most effective agents for a given problem. In a genetic search problem, a reinforcement learning agent can be described by its genetic make up: exploration coefficient, step-size parameter, learning rate, even update rule. Figure 7.1 shows how such a genetic algorithm can be used. The individuals are made to interact with the environment and their learning performance is measured so as to indicate their fitness.

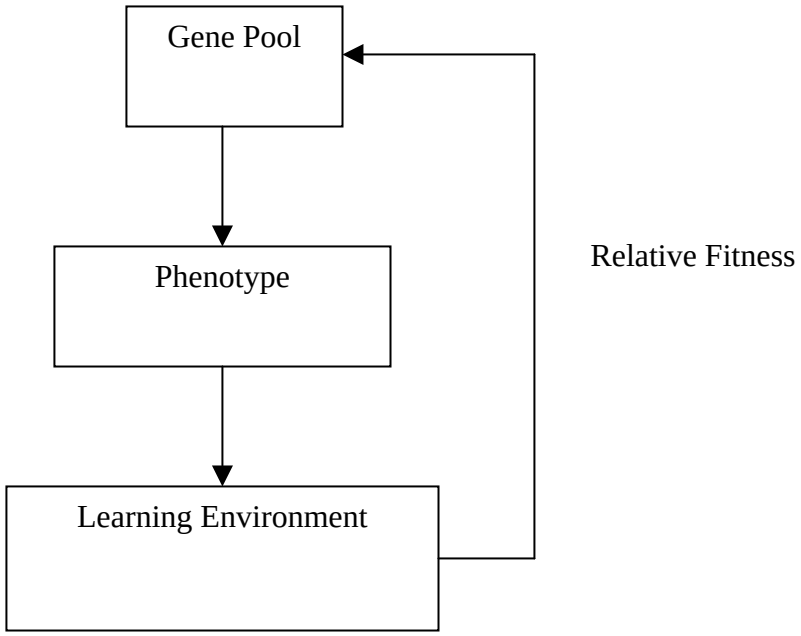

Figure 7.1 Genetic algorithms used in search for a reinforcement learning agent; a gene represents the set of reinforcement learning parameters.

One measure of learning performance can be, for example, how high as average reward in an episode the agent can achieve within a finite number of interactions.

7.3 Evolutionary Methods Effected by Individual Learning

An interesting problem has been suggested by which individual learning can alter the course of evolution. One such mechanism is called the Baldwin effect, after J.M. Baldwin (1896), who first suggested the idea. The Baldwin effect is based on the following observations. First, if a chromosome is evolving in a changing environment, there will be evolutionary pressure to favor individuals with the capability to learn during their lifetime. In fact the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness. In contrast, non-learning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage. Secondly, those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits. As a result, these individuals can support a more

diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code. This more diverse gene pool can, in turn, support more rapid evolutionary adaptation. Thus, the ability of individuals to learn can have an indirect acceleration effect on the rate of evolutionary adaptation for the entire population. We believe this property of learning should be explored in order to improve our sequence prediction problems.

7.4 Large State-Space Problems

So far our research involved finite state-space problems. In cases where the number of known vulnerability becomes exceedingly large, the look-up table method of updating state-action values may no longer be feasible. Although several methods in reinforcement learning area have dealt with this problem using function approximation methods, we believe the key feature in our problem, namely bandwidth limitation, presents new and interesting challenges that need to be researched extensively. As discussed in this thesis, bandwidth limitation forces the agent to limit its exploration of actions from a given state by selecting only a small sub-set of possible

actions. This limitation, and the fact that the agent has a partial ability to *see* the environment are perhaps the single key aspect of vulnerability detection problem that makes it a unique case of general sequence prediction.

Developing algorithms and ideas that can improve our ways to solve problems presented in this work will undoubtedly have a positive impact in all wireless networks in today's complex communication systems. The issues of security and vulnerability in our communication systems are becoming increasing important considering that more diverse wireless systems are being introduced to mobile communication systems.

Appendix A

The following code is a simulator generating a sequence of symbols with the symbol pair [2 1] appearing with a desired frequency. Such a code may represent a pattern in the appearance of vulnerabilities in the network. In this case, alphabet of symbols is set to five. A total of N = 1000 symbols are generated:

```
% generate N symbols from alphabet of 5 (32 - 1 possible non-zero
% combinations);
% Pattern: [b a] occurs with f probability.
% first get output vector; N symbols in integer form
alphabet=5;
actioncombo=1;              % number of symbol combinations in each
ACTION; if 1, actions (i.e., predictions) are single symbol.
numsymbols=2^alphabet-1;
output=zeros(1,N);
outputvector=zeros(alphabet,N);
```

```
 simulator=zeros(alphabet,N);
% choose symbols in the pattern
b1=zeros(alphabet,1);
b1(2)=1;
b=b1;
a1=zeros(alphabet,1);
a1(1)=1;
a=a1;
%*****************************************************
% create matrix of others (symbols other than b and a).
%*****************************************************
% first create the set of all possible symbols
for i=1:N
   symindex=ceil(rand*numsymbols);
   output(i)=symindex;
end
% convert output symbols to binary vector form
for j=1:N
   z=zeros(alphabet,1);
```

```
 cint=output(j);
   cchar=dec2base(cint,2);
   ccharinv=cchar';
   cbin=bin2dec(ccharinv);
   [r c]=size(cbin);
   r1=alphabet-r;
   r2=r1+1;
   z(r2:alphabet)=cbin;
   outputvector(:,j)=z;
end
% now find set of 'other' by eliminating single occurrence b's and a's
from the outputvector
for j=1:N
   if isequal(outputvector(1,j),1) | isequal(outputvector(2,j),1)
   % if xor(outputvector(1,j),outputvector(2,j))
      outputvector(:,j) = zeros(alphabet,1);
   else
   end
end
```

```
 count=N;
while count>=1
    if outputvector(:,count)==zeros(alphabet,1)
        outputvector(:,count)=[];
    else
    end
    count=count-1;
end
other=outputvector;
 % now create a flag to insert pattern
rate=f/100;
p=rand(1,N) < rate;                          % binary vector
% Impose pattern:
pattern=repmat(a,1,N);
for j=1:2:N
    pattern(:,j)=b;
end
for k=1:N
    if ~p(k)
```

```
        [r c]=size(other);
       idx=ceil(rand*c);
       pattern(:,k)=other(:,idx);
   end
end
simulator=pattern;
%*********************************************************
 %  Now determine symbolSet, the set of possible ACTIONS
 % (call it symbolSet to be consistent w/predictEpattern code)
 % In this case, actions can be single or double alphabet set
%*********************************************************
total=1:numsymbols;       % first get integer set of all possible actions
                          % now convert to binary set of all possible
                          actions.
totalbin=zeros(alphabet,numsymbols);
for j=1:numsymbols
   z=zeros(alphabet,1);
   cint=total(j);
   cchar=dec2base(cint,2);
```

```
     ccharinv=cchar';
    cbin=bin2dec(ccharinv);
    [r c]=size(cbin);
    r1=alphabet-r;
    r2=r1+1;
    z(r2:alphabet)=cbin;
    totalbin(:,j)=z;
end
set=zeros(alphabet,numsymbols);
for i=1:numsymbols                % 'set' is all single & double symbol
actions
    col=totalbin(:,i);
    if     sum(col)==actioncombo
 % Let's generate only single actions .
        set(:,i)=col;
    end
end
set;
lastSet=set;
```

```
 count=numsymbols;                % eliminate all-zero columns out of
'set' to get final answer
while count >= 1
    if lastSet(:,count)==zeros(alphabet,1)
        lastSet(:,count)=[];
    end
    count=count-1;
end
symbolSet=lastSet;
```

Appendix B

Reinforcement learning prediction code based on the on-policy, temporal-difference learning. The look-up table of state-action values maintained and updated by the agent is represented by the “SARSA” matrix in the following code:

```
function [SARSA,R,optact,symbolSet] =
predictEgenpair(N,f,E,alpha,gama)
%FUNCTION
[SARSA,,R,optact,symbolSet,t]=predictEgenpair(N,f,E,alpha,gama)
%  Predict pattern of symbols generated by simulator ‘genpair’;
pattern: pattern "b,a" occur f percent of the time;
%  genpair generates 5 symbols (a,b,...) represented by 5-tupple
binary vector, plus all combinations
%  Inputs:
%       f= frequency of occurrence of the symbol pair 'b,a'.
%       E= epsilon for epsilon-greedy algorithm
```

```
 %       alpha= step size paremeter
%        gama= gamma, discount factor
%        Ouputs:
%        SARSA= matrix of state-action values obtained by SARSA
update rule
%        As=Action selected on trial j, j=1:N
%        Ss=State on trial j, j=1:N
%        R= reward on action j, j=1:N
%        optact= check if action is optimal in each step (if action is a
%        after occurrence of b, it is optimal.
%        epi_length= length of episode in # of consecutive state
transition
%       gen3pattern; alphabet=3;
%       genpair;                          % generate the sequence of N
symbols with desired f
numChoices = length(symbolSet);        % symbolSet is defined in
gen5
numStates=numChoices + 1;              % States correspond to action
choices plus 'Terminal' state
```

```
 SARSA=zeros(numStates,numChoices);
As=zeros(alphabet,N);                % Storage for action
R=zeros(N,1);                        % Storage for averaging reward
Q=zeros(1,numChoices);
optact=zeros(N,1);
opt=0;
%          Initialize action
%*********************************************************
terminal=1;              % start from terminal state until you discover
starting state
cQ=ceil(rand*numChoices);
action=symbolSet(:,cQ);
%  Now we're ready to start interaction
%*********************************************************
for j=1:N
   if terminal
      sim=simulator(:,j);
```

```
[state,reward]=compare(action,sim,numChoices,numStates,symbolSet
);   % Interact with simulator
      if reward==1
      % if action==simulator(:,j)
         cS=state;
         %cS=cQ;
         Q=SARSA(cS,:);            % a starting state is discovered
         cQ=Egreedy(Q,E);          % select action in this state
         action=symbolSet(:,cQ);     % action for the next round
         terminal=0;
         cR=0;
      else
         cQ=ceil(rand*numChoices);   % select another random action
and go back to terminal state
         action=symbolSet(:,cQ);     % action for the next round
         terminal=1;
         cR=0;
      end
```

```
   else
%*********************************************************
      %     In Episode: get cR, update state-action values according to:
      %     SARSA(cS,cQ)=SARSA(cS,cQ)+alpha*[cR +
gama*SARSA(new_cS,new_cQ)-SARSA(cS,cQ)];
%*********************************************************
     sim=simulator(:,j);

[state,reward]=compare(action,sim,numChoices,numStates,symbolSet
);   % Interact with simulator
       if reward>0
          %if action==simulator(:,j)
          %cR=1;
          cR=reward;
          new_cS=state;
          new_cS=cQ;
          new_Q=SARSA(new_cS,:);
          new_cQ=Egreedy(new_Q,E);
```

```
            if (action(2)==1 & symbolSet(1,new_cQ)==1)  % if next
action has 'a' and current action had 'b'
            %if (action==b & symbolSet(:,new_cQ)==a)
               opt=1;
            else
               opt=0;
            end
            SARSA_1=SARSA(cS,cQ);
            SARSA_2=SARSA(new_cS,new_cQ);
            SARSA_3=gama*SARSA_2;
            SARSA_4=cR+SARSA_3-SARSA_1;
            SARSA(cS,cQ)=SARSA_1+alpha*SARSA_4;
            cS=new_cS;                  % for the NEXT round, get cS and
cQ & action
            cQ=new_cQ;
            action=symbolSet(:,cQ);       % action for the next round
            terminal=0;

         else
```

```
 cR=reward;
%cR=0;
new_cS=numStates;
new_Q=SARSA(new_cS,:);          % we are at terminal state
new_cQ=ceil(rand*numChoices);       % this will be next
action, taken at random
%if (action==b & symbolSet(:,new_cQ)==a)
if (action(2)==1 & symbolSet(1,new_cQ)==1)
   opt=1;
else
   opt=0;
end
SARSA_1=SARSA(cS,cQ);
SARSA_2=0;
SARSA_3=0;
SARSA_4=cR+SARSA_3-SARSA_1;
SARSA(cS,cQ)=SARSA_1+alpha*SARSA_4;
cQ=new_cQ;
action=symbolSet(:,cQ);       % action for the next round
```

```
            terminal=1;
           episode=0;
        end
        %UPDATE FOR NEXT GO AROUND.
        Ss(j)=cS;                % keep a list of index of states visited
        R(j)=cR;
        optact(j)=opt;
    end
    As(:,j)=action;              % keep a list of actions
end
```

Appendix C

The following code is one of the variations used in deriving the results in this thesis. The code is based on genetic algorithms we used to illustrate prediction of patterns involving a symbol pair in a discrete sequence.

```
% Genetic Algorithm in sequence prediction
getpop_genpair;     % initialize 'pop', the first population; and all
generated symbols by 'genpair' simulator
f1=zeros(generationsize,1);
f2=zeros(generationsize,1);
string_fitness=zeros(generationsize,1);
count=0;
for i=1:N
   if mod(i,2)==0  % compute 2nd predicion and get string fitness of
2nd gene
       v2=simulator(:,i);
```

```
        for j=1:generationsize
            gene=eye(alphabet,chrom2(j));     % chrom1,chrom2 defined
in getpop_genpair function
            gene2=gene(:,chrom2(j));              % get corresponding gene
in binary vector form
            f2(j)=fitness(gene2,v2);
            %string_fitness(j)=(f1(j) * f2(j))*4;    % multiply by 4 to get a
round maximum fitness of 100.
            string1=f1(j)*f2(j);
            %string2=(50^isequal(string1,25)) * 2;
            string2=f1(j)*f2(j)*4;              % try this: still get max 100 but
give partial fitness also
            string_fitness(j)=string2;


        end
            count=count+1;          % plot results at even N's: 2,4,6,...
which is count:1,2,3,...
        avefitness_gen(count)=sum(string_fitness ./ generationsize);
        maxfitness_gen(count)=max(string_fitness);
```

```
                    % also find ratio of fit individuals all (b,a)'s and
(a,b)'s
      p1=pop(:,1);
      p2=pop(:,2);
      ch1=ismember(p1,2);
      ch2=ismember(p2,1);
      fit=and(ch1,ch2);
      x=find(fit);
      numfit=length(x);
      fit_gen(count)= numfit/generationsize;

      t(count)=cputime-t0;          % measure elapsed time at each step

    % ********* find next generation ********

      gen_index=selrws(string_fitness,generationsize);    % select
fittest chromosomes for mating
```

```
        gen=pop(gen_index,:);                % fittest chromosomes to be
recombined
        new_pop1=recdis(gen);                % ready for mutation
        new_pop2=mutintg(new_pop1,.1,alphabet);
        pop=new_pop2;                        % start the process again with
this new population
        chrom1=pop(:,1)';
        chrom2=pop(:,2)';

        f1=zeros(generationsize,1);          % initialize fitness for next
generation
        f2=zeros(generationsize,1);
        string_fitness=zeros(generationsize,1);

    else          % just compute 1st prediction and get fitness of 1st gene
in chromosome

        v2=simulator(:,i);  % observe the first incoming symbol in
sequence
```

```
        for j=1:generationsize
          %f1(j)=fitness(pop(1:5,j),v2);
          gene=eye(alphabet,chrom1(j));  % convert decimal to binary
gene for fitness function
          gene1=gene(:,chrom1(j));          % chrom1, chrom2 defined in
getpop_genpair funcion
          f1(j)=fitness(gene1,v2);
       end


   end
end
```

Appendix D

A variation of the mutation operator:

```
% mutintg fundtion: mutation for integer chromosomes,
function NewChrom = mutintg(OldChrom,MutRate,alphabet)
% get population size (Nind) and chromosome length (VarLength)
[Nind,VarLength]=size(OldChrom);
% Check input parameters
% if nargin < 3, MutRate = []; end
if isnan(MutRate), MutRate = []; end
if isempty(MutRate), MutRate=0.7/VarLength;
end
MutRate=MutRate(1);
% Perform mutation of individuals
```

```
 MutValue=ceil(alphabet * rand(Nind,VarLength));     % matrix of
muted values
range=rand(Nind,VarLength) < MutRate;
i=find(range);                              % locations in OldChrom where
you want mutation
m=MutValue(i);                               % raplace value w/ mutated
value in those locations
OldChrom(i)=m;
NewChrom=OldChrom;
```

Bibliography

[1] Barret, M., Little, M., Poylisher, A., (2001) *Intelligent agent for vulnerability assessment of computer networks,* Telcordia.

[2] Barto, A.G., and Duff, M., (1994) Monte Carlo matrix inversion and reinforcement learning in J.D. Cohen, G. Tesauro, and J. Alspector (eds.), *Advances in Neural Information Processing Systems: Proceedings of the 103 Conference, pp. 687 – 694.* Morgan Kaufmann, San Francisco.

[3] Barto, A.G., Sutton, R.S., Brouner, P.S. (1981) Associative search networks: A reinforcement learning associative memory. *Biological Cybernetics*, 40: 201-211.

[4] Barry, D.A., Fristedt, B. (1985) Bandit Problems. Chapman & Hall.

[5] Bertsekas, D.P. (1995) *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, MA.

[6] Conner, M., Shirag, P., Little, M. (Jan, 2001) *Genetic Algorithm / Artificial Life Evolution of Security Vulnerability Agents*, U.S. Army Research Lab.

[7] Ekart, A., (2003) *Using genetic algorithms for improved discrete sequence prediction*, International Multi-conference in Computer Science and Computer Engineering.

[8] Feldbaum, A.A. (1965) Optimal control systems, Academic Press, New York.

[9] Goldberg, D. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison, Wiley.

[10] Harmon, Mance, E., Harmon, Stephanie, S. (2001) Reinforcement Learning: A tutorial, Wright Laboratory.

[11] Hassan, K., Conner, M., (July 2002) “Prediction Feedback Shift Register (LFSR) Using Genetic Algorithms”,, Genetic Evolution and Computation Conference. Workshop Proceedings, pp. 14 – 18, New York, NY.

[12] Holland, J., (1992) *Adaptation in natural and artificial Systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press.

[13] Holland, J., H., (1975) *"Adaptation in Natural and Artificial Systems."*, Ann Arbor, MI: University of Michigan Press.

[14] Barry, D.A., Fristedt, B.: (1985) *Bandit Problems*, Chapman & Hall.

[15] John, G.H., (1994) *When the best isn't optimal: Q-learning with exploration.* In proceedings of the Tweljth National Conference in Artificial Intelligence, Seattle, WA., pp 1464.

[16] John, G.H., (1995) *When the best move isn't optimal: Q-learning with exploration*. Unpublished manuscript, available at URL. ftp://starry.stanford/edu/pub/gjohn/papers/rein-pips/p/s/

[17] Kaebling, L.P., Littleman, M.L., & Moore, A.W. (1996) Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4. 237 – 285

[18] Kalos, M.H., and Whitlock, P.A. (1986) *Monte Carlo Methods*, Wiley New York.

[19] Klopf, A.H. (1972) *Brain function and adaptive systems - A hetero-static theory.* Technical Report AFC RL-72-0164, Air Force. Cambridge Research Laboratories, Bedford, MA. A summary appears in Proceedings of the International

Conference on Systems, Man, and Cybernetics, IEEE Systems, Man, and Cybernetics Society, Dallas, TX, 1974.

[20] Larid, P., and Saul, R. (1994) Discrete sequence prediction and its applications. *Machine Learning*, 15: 43-68.

[21] Michie, D., and Chambers, R.A. (1968) Boxes: An experiment in adaptive control. In E. Dale and D. Michie (eds.), *Machine Intelligence* 2, pp. 137-152. Oliver and Boyd, Edinburgh.

[22] Minsky, M.L. (1967) *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ.

[23] Mitchell, T. (1997) *Machine Learning*, McGraw-Hill, vol. 8.1997.

[24] Pearl, J. (1984) *Heuristics: Intelligence Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA.

[25] Ross, S. (1983) *Introduction to Stochastic Dynamic Programming*. Academic Press. New York.

[26] Rubinstein, R.Y. (1981) *Simulation and the Monte Carlo Methods*. Wiley, New York.

[27] Rummery, G.A., and Niranjan, M., (1994) *On-line Q-learning using connectionist systems.* Technical report. CUED/IN FENG/TR. 166. Engineering Department, Cambridge University.

[28] Rummery, G.A., (1994) *Problem solving with reinforcement learning,* Ph.D. thesis, Cambridge University, Engineering Department.

[29] Samuel, A.L. (1959) Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development,* 11:610-617.

[30] Sharif, H., Conner, M., (2003) *A reinforcement learning-based algorithm to predict vulnerability occurrence in a wireless networking environment,* Proceedings of the International Multi-Conference in Computer Science & Computer Engineering.

[31] Sharif, H., Conner, M., (2004) *Discrete sequence prediction using machine learning methods,* Proceedings of the International Multi-Conference in Computer Science & Computer Engineering.

[32] Sharif, H., Conner, M., (2005) *Machine learning applications in wireless networks,* Journal of Machine Learning Research (submitted).

[33] Singh, S., and Dayan, P. (1998) Analytical mean squared error curves for temporal difference learning, *Machine Learning*.

[34] Sun, R. (2000) Introduction to sequence learning. In *Sequence Learning Paradigms, Algorithm, and Adaptations,* volume 1828 of LNAI, pages 1-10.

[35] Sun, R., and Giles, C.L. (2001) sequence learning: from recognition and prediction to sequential decision making. *IEEE Intelligent Systems,* July/August: 67-70.

[36] Sutton, R.S., (1996) Generalizations in reinforcement learning: Successful examples using sparse coarse coding. In D.S. Touretzky, M.C. Mozer and M.E. Hasselmo (des.), *Advances in Neural Information Processing Systems:* Proceedings of the 1995 Conference, pp. 1038-1044. MIT Press, Cambridge, MA.

[37] Sutton, R.S., Barto, A., (1998) *Reinforcement Learning,* The MIT Press.

[38] Thathchar, M.A.L. and Sastry, P.S. (1985) *A new approach to the design of reinforcement schemes for learning automata*, IEEE Transactions on Systems, Man, and Cybernetics, 15: 168-174.

[39] Thrun, S.B. (1992) The role of exploration in learning control. In 'D.A. White & D.A. Sofge (eds.), *Handbook of intelligent control: neural, fuzzy, and adaptive approaches*. New York, NY Van Nostrand Reinhold.

[40] Watkins, C.J.C.H. (1989) *Learning from Delayed Reward*, Ph.D. thesis, Cambridge University.

[41] Watkins, C.J.C.H., and Dayan, P. (1992) Q-learning. *Machine Learning.* 8:279-292.

[42] Witten, I.H. (1976) The apparent conflict between estimation and control – A survey of the two-armed bandit problem. *Journal of the Franklin Institute*, 301:161-189.

[43] Wittle, P. (1982) *Optimization Over Time*, vol. 1. Wiley, New York.

[44] Wittle, P. (1983) *Optimization Over Time*, vol. 2. Wiley, New York.

[45] White, D.J. (1969) *Dynamic Programming*, Holden-Day, San Francisco.