USING COEVOLUTION IN COMPLEX DOMAINS

by

Ali Alanjawi

M.S. in Computer Science, Kuwait University, 2000

B.S. in Mathematics, Kuwait University 1996

Submitted to the Graduate Faculty of

Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2009

UNIVERSITY OF PITTSBURGH

ARTS AND SCIENCES

This dissertation was presented

by

Ali Alanjawi

It was defended on

February 26, 2009

and approved by

Robert Daley, PhD, Professor

Milos Hauskrecht, PhD, Associate Professor

Sangyeun Cho , PhD, Assistant Professor

John Grefenstette, PhD, Professor, Department of Bioinformatics and Computational Biology, George Mason University.

Dissertation Director: Robert Daley, PhD, Professor

USING COEVOLUTION IN COMPLEX DOMAINS

Ali Alanjawi, PhD

University of Pittsburgh, 2009

Genetic Algorithms is a computational model inspired by Darwin's theory of evolution. It has a broad range of applications from function optimization to solving robotic control problems. Coevolution is an extension of Genetic Algorithms in which more than one population is evolved at the same time. Coevolution can be done in two ways: cooperatively, in which populations jointly try to solve an evolutionary problem, or competitively. Coevolution has been shown to be useful in solving many problems, yet its application in complex domains still needs to be demonstrated.

Robotic soccer is a complex domain that has a dynamic and noisy environment. Many Reinforcement Learning techniques have been applied to the robotic soccer domain, since it is a great test bed for many machine learning methods. However, the success of Reinforcement Learning methods has been limited due to the huge state space of the domain. Evolutionary Algorithms have also been used to tackle this domain; nevertheless, their application has been limited to a small subset of the domain, and no attempt has been shown to be successful in acting on solving the whole problem.

This thesis will try to answer the question of whether coevolution can be applied successfully to complex domains. Three techniques are introduced to tackle the robotic soccer problem. First, an incremental learning algorithm is used to achieve a desirable performance of some soccer tasks. Second, a hierarchical coevolution paradigm is introduced to allow coevolution to scale up in solving the problem. Third, an orchestration mechanism is utilized to manage the learning processes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

"I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection." — Charles Darwin (1859) from "The Origin of Species"

Darwin's theory of evolutionary selection holds that variation within species occurs randomly, and that the survival of each organism is determined by its ability to adapt to its surrounding environment. In computer science, these principles were adapted to construct an evolutionary simulation in which individuals represent prospective solutions in problem solving. Darwin's natural selection can be modeled by imposing a selection distribution on the population of solutions such that the better ones have a higher probability of being recombined into new solutions, thereby preserving the attributes that made them viable.

For the past thirty years the principles of evolution have been applied to solve problems in many fields. Nevertheless, their application in complex problems is limited. In this work, evolutionary algorithms will be applied in a complex, noisy, and dynamic environment. This may expand the horizon of successful usage of evolution in difficult domains and hopefully will allow us to unleash some of coevolution's unexplained dynamics.

1.1 MOTIVATION

In the last decade, machine learning has had a number of successful applications, ranging from developing the world champion of backgammon synthetic players (Tesauro 1995), to autonomous vehicles that can drive on public highways (Pomerleau and Jochem 1996). Coevolution has been shown to have potential in learning difficult tasks and has been applied successfully in various domains by many researchers. (Potter and De Jong 2000; Paredis

1995; Daley, Schultz, and Grefenstette 1999) Yet, successful application of coevolution to real life problems is limited. These complex problems provide a challenging environment for any machine learning technique.

Stone (1998) applied many machine learning techniques in the domain of robotic soccer. For this research, evolutionary and coevolutionary approaches will be applied to this domain. Robotic soccer is an interesting domain for coevolution because both cooperative and competitive behaviors can be observed. In addition to expanding the limited knowledge that describes how coevolution works, applying it in such a complex domain will give us a better understanding of its dynamics and capabilities.

1.2 OBJECTIVES AND APPROACH

The principal question addressed in this thesis is:

> **Can coevolution be successful in allowing agents to learn to cooperate and become individually skilled in a complex, dynamic, and noisy environment?**

The environment must have the following properties: noisy sensors and actuators, limited sensing capabilities, and agents divided into groups with conflicting objectives but sharing the same goal within each group. Robotic soccer has all of these characteristics, and it is considered to be an excellent test bed for machine learning techniques; therefore, a simulated robotic soccer domain will be used as an environment to answer the question stated above.

To assess the effectiveness of coevolution, a coevolutionary model will be developed that has the following key characteristics:

- Incremental evolution
- Task decomposition
- Orchestration of coevolutionary learning

Genetic algorithms are used in developing the model in the robotic soccer domain because they provide a broader range of methods to encode problems into genetic materials than other evolutionary algorithms. Encoding is an important aspect of solving problems using genetic algorithms. In the robotic soccer domain, two attributes can be translated into genetic materials: sensor and action values. Agents must take action if sensors have certain values; therefore, representing the environment as a set of if-then rules is a natural choice. A rules representation has many advantages over other representations (like lisp programs or neural networks). First, it is easy to implement and does not have extra overhead like determining the number of nodes and hidden layers in neural networks. Second, it is human readable, so where it fails can be analyzed and determined. Also, representing genetic materials as rules restricts the genetic algorithms search to where it is needed – i.e. to the sensors and actions values – unlike with lisp programs, where the algorithm might search in useless places (such as whether two values should be added or multiplied).

SAMUEL is a system that uses genetic algorithms in solving robotic problems. It uses a set of if-then rules representation, and it has the ability to coevolve more than one population in various ways. It also has parallel execution on multi-platform support for faster evaluations. Therefore, SAMUEL is used as a testing vehicle for the experiments in this thesis.

1.3 MAIN CONTRIBUTIONS

The main contributions of this dissertation can be outlined as follows:

- **Incremental learning** applies evolutionary algorithms incrementally to achieve a desired complex behavior. In incremental learning, evolutionary learning shows a faster learning rate, and provides a better performance than in non-incremental learning.
- **Hierarchical coevolution** is applied in a complex domain in which learning the domain's goal directly is intractable. Dividing a complex problem into subcomponents and coevolving them in a layered approach allows for learning to proceed incrementally from low level tasks to higher level ones. The goal task, which is at the highest level of the hierarchy, is learned with a satisfactory performance. In this paradigm, a new method of applying learning to a squad of agents in which they share their learning process concurrently is applied. This approach simplifies the learning process and makes it viable for a complex domain that has 22 interactive agents.

- **Orchestration** is a new method created that manages the interaction between convolutionary learning processes. Instead of using a fixed policy for concurrent learning, a dynamic policy is used based on a set of performance metrics which are gathered during the learning process. This method showed a significant performance enhancement over static policies.
- A team of soccer players was learned entirely by a machine learning algorithm and showed a competitive performance against other hand-crafted soccer teams. This demonstrates that coevolution can be successful in allowing agents to learn to cooperate and become individually skilled in a complex, dynamic, and noisy environment such as RoboCup soccer.

1.4 DISSERTATION OUTLINE

In Chapter Two, a background of evolutionary algorithms and an overview of related work are provided. Chapter Three presents the main idea of this thesis and provides a description of tools used. Incremental learning is discussed in Chapter Four. Chapter Five demonstrates how hierarchical coevolution is used in the RoboCup soccer domain. Orchestration techniques are delineated in Chapter Six. The last chapter concludes the dissertation research and provides some future research directions.

2.0 REVIEW OF THE LITERATURE AND RELATED WORK

The topic of this thesis is the application of coevolution to robotic soccer. An introduction to the general field of evolutionary algorithms will now be presented to familiarize the reader with some terminology that will be used throughout this thesis. The three classes of evolutionary algorithms – evolution strategies, evolutionary programming, and genetic algorithms – will be briefly described in section 2.1. The coevolution model used for this research is genetic algorithms; consequently, genetic algorithms will have a more detailed treatment in section 2.2. A literature review of coevolution will be given in the subsequent section. Robotic soccer is considered a reinforcement learning problem and most of the works on robotic soccer are based on reinforcement learning techniques, so a brief introduction to reinforcement learning will be given in section 2.4. Finally, the last section will review the related works on robotic soccer.

2.1 EVOLUTIONARY ALGORITHMS

Historical Overview

Evolutionary Algorithms are a class of computational algorithms based on Darwin's theory of evolution. Many researchers in the 1950s and early 1960s were inspired by biological evolution to use it in solving computational problems. Box (1957), Friedman (1959), Bremermann (1962), Reed, Toombs, and Barricelli (1967) worked on this idea; however, none of their works got great attention or were followed up by others at that time. In 1964, Rechenberg introduced "evolution strategies" and used them to optimize real-valued functions in engineering problems. Further development by Schwefel (1975, 1981) anchored the basis of evolution strategies, which remains an active area of research (Beyer and Arnold 2003; Willmes and Bäck 2003; Oyman, Beyer, and Schwefel 2000). Fogel, Owens, and Walsh (1966) developed "evolutionary programming". They used randomized mutation, a natural evolution operator, to evolve finite state machines by changing their state transitions. Both evolution strategies and evolutionary programming have been successfully used in many function optimization problems. "Genetic algorithms" was invented by John Holland in the 1960s. Holland focused on natural adaptation and ways in which it can be used as a computational model. Consequently, genetic algorithms use more domain-independent representation and have been widely used in many applications in various fields (Smith 1983; De Jong and Oates 2002; Grefenstette 1989). Evolution strategies, evolutionary programming, and genetic algorithms form the so-called evolutionary algorithms.

Biological Inspiration

There are many differences among the three methodologies that form evolutionary algorithms, and each has a different theory behind it. However, they were all inspired by the Darwinian principles of evolution. Before describing how principles of evolution are applied in evolutionary algorithms, some biological terminology is given below.

Living organisms are characterized by chromosomes, which reside in each individual cell. Chromosomes consist of genes. Genes act as a trait of the organism; for example, genes can designate the color of the hair of an individual. Any alternative form of a gene is called an "allele". As many organisms have more than one chromosome in their cells, the collection of all chromosomes is called a "genome". "Genotype" refers to the particular sets of genes in a genome. There is a variety of genomes, because there is a variety of organisms; thus, organisms have different physical characteristics, which are called "phenotypes".

Recombination and mutation are the most fundamental operators in Darwin's theory. Recombination occurs when two organisms (parents) mate to produce a new offspring. In recombination (also called crossover), the genes are exchanged between each pair of chromosomes, one from each parent, to form new chromosomes. These chromosomes form the basis of the new offspring's genome. The offspring's genome might also then mutate. In other words, one or more genes of the offspring's chromosome will have different alleles than

```
Algorithm 1 (Evolutionary Algorithm)
t=0
Initialize P(t)*
Evaluate P(t)
While (termination criterion not fulfilled)
    Begin
        Select parents $P'(t)$ from $P(t)$
        Recombine $P'(t)$ to form offspring $P''(t)$
        Mutate $P''(t) \rightarrow P'''(t)$
        Evaluate $P'''(t)$
        Select the fittest for the next generation: $P'''(t) \rightarrow P(t+1)$
        t=t+1
    End

* $P(t)$ refers to the population at generation t.
```

Figure 2.1: An outline of an Evolutionary Algorithm

that of their parents. Each organism has a "fitness" that is defined by the organism's ability to survive and reproduce; hence survival is for the fittest. The cycle of recombination and mutation results in evolution.

Evolutionary algorithms simulate the evolution of organisms as a model of computation. Each additional application of evolution should yield a better (fitter) solution than the previous one. Therefore, a solution for a particular problem can be achieved by initially creating a random solution and then evolving it into an optimal (or near-optimal) solution. This simulation can be easily implemented once a solution of the problem can be represented as a set of genes.

Figure 2.1 illustrates an outline of an evolutionary algorithm. A population of n individuals[1] (solutions) is first initialized. Initialization can be at random, or at a specific value taken from any knowledge known about the particular problem being solved. After that, each individual in the population is evaluated according to its fitness in the environment. The evolution proceeds from generation t to generation $t+1$ by repeated use of selection, recombination, mutation, and evaluation. Selection is done twice, first for selecting parents at the beginning, and second for selecting offspring at the end of the cycle. After the parents

[1] Individuals and chromosomes will be used exchangeably to refer to the same thing.

are selected, they are recombined to form the new offspring, which are mutated and then evaluated. Based on this evaluation the population of the next cycle is selected from these offspring.

The algorithm described above can be seen as a general evolutionary algorithm although each class of evolutionary algorithms has its own representation and utilizes different applications of the Darwinian operators. Evolution strategies, evolutionary programming, and genetic algorithms are described in more detail in the following subsections.

2.1.1 Evolution Strategies

Evolution strategies were originally introduced by Rechenberg (1964) to solve problems in hydrodynamics[1], where individuals are represented by vectors of real numbers. The original algorithm was based on evolving one parent and then mutating it to generate one offspring. The fitter of the parent or the offspring is selected to survive into the next generation. Mutation, which is the only operator used, is done by disturbing the (real-valued) individual by an amount produced from a Gaussian distribution. The variance of the Gaussian distribution can be adaptive over time. Rechenberg invented the "1/5 success rule" to adjust the Gaussian variance (Rechenberg 1973). This rule stated that the variance should be increased if the ratio between the number of successful mutations and the total number of mutations is greater than 1/5, otherwise the variance should be decreased. This rule has been shown to produce a high rate of convergence in some functions, like sphere and corridor functions.

Rechenberg also proposed using a population of size greater than one and generating one offspring which would replace the worst parent individual. Although, this method has not been widely used, it facilitates more successful evolution strategies, namely (μ,λ) and (μ+λ) evolution strategies, which were introduced by Schwefel (1975, 1981). The parameter μ refers to the number of parents in the population, while λ refers to the number of offspring generated. In the (μ,λ) strategy, the best μ individuals are selected from the λ offspring generated, while in the (μ+λ) strategy, μ individuals are selected from both the parents and the λ offspring generated [2]. Schwefel also introduced recombination into evolution strategies

[1] Such as shape optimization for a bent pipe and a flashing nozzle.

[2] Given these definitions, the original evolution strategy can be denoted as (1+1).

(Schwefel 1981).

In general, a typical evolution strategy algorithm starts by initializing and evaluating a population of individuals. Then pairs of individuals are randomly selected to generate λ offspring by (Gaussian) mutation. If recombination is used, pairs of parents are recombined first to form offspring before mutation is applied. Then, depending on which strategy is used, (μ,λ) or (μ+λ), μ fittest individuals are selected for the next generation.

It has been proven that (μ+λ) has a global convergence (with a probability of 1), i.e., it will indeed converge to an optimal solution. The global convergence of (μ,λ) is still an open question.

2.1.2 Evolutionary Programming

Evolutionary programming was developed by Fogel, Owens, and Walsh in 1966. They attempted to solve some prediction problems by evolving finite state machines. The state transitions of the finite state machines are randomly mutated and then evaluated by the number of symbols predicted correctly. The best half of the mutated machines and the best half of the original machines are selected for the next generation[1]. Evolutionary programming is extended and used in real-valued function optimization by D. Fogel, who also introduced adaptive mutation (Fogel 1992). Recombination is not used in evolutionary programming; hence mutation is considered the only operator.

2.1.3 Genetic Algorithms

The idea of genetic algorithms was introduced by John Holland in 1960s at the University of Michigan. Holland's goal was to study the adaptation phenomenon in nature and to discover ways in which it could be applied to computer systems. Unlike evolution strategies and evolutionary programming, genetic algorithms model the evolutionary process at the genome level. That is to say, rather than evolving real-valued individuals, genetic algorithms evolve individuals that consist of genes, which might be represented as numbers, strings, or any kind of data structure. This representation gives genetic algorithms a broad spectrum

[1] Using evolution strategies terminology, this method might be called (μ+μ).

of applications once a proper coding is found between the problem definition and the genes. For this reason, it is the class of evolutionary algorithms that will be used in this thesis.

Three operators were originally introduced by Holland: crossover, mutation, and inversion. Crossover exchanges a part of the genes of two individuals. Mutation randomly changes alleles for some genes of an individual. Inversion reverses the order of the genes of an individual. Holland's algorithm proceeds by initializing the population, and the evolutionary process begins by selecting parents and applying crossover to them to generate offspring. Then mutation and inversion are applied to those offspring, which are then evaluated for selection in the next generation.

Holland also provided a theoretical basis for genetic algorithms based on "schemata". A schema is a set of individuals in an adaptation system (sometimes called building blocks), which may include a "don't care" state. For example, each schemata in a binary chromosome can be specified by a string of the same length of the chromosome containing 0, 1, or $*$ (don't care symbol). The "order" of a schema is the number of non $*$ positions specified. According to Holland's schema theorem, short low-order, above-average schemata receive exponentially increasing trials in following generations. Holland's analysis suggests the following: selection increasingly focuses the search on subsets of the population in which individuals have above-average fitness, whereas crossover puts high-fitness "building blocks" together on the same string in order to create increasingly fitter individuals. Mutation makes sure that diversity is never lost by introducing new alleles into chromosomes (Holland 1975).

2.2 GENETIC ALGORITHMS AS A PROBLEM SOLVER

Genetic algorithms can be seen as a generic stochastic search model. A search usually begins with a randomized initial value and proceeds toward the goal. What makes a search reach a goal in genetic algorithms is selection, which is based on the performance of individuals on its given tasks. The performance of individuals is usually measured by a function called the "fitness function" . The nature of the algorithm makes it successful in searching through an enormous number of possibilities for solutions. In addition, its simple operators make it work well when unclear or scarce knowledge of a problem exists.

As previously described, the algorithm begins with a random population and proceeds with an iterative application of evaluation, selection, recombination, and mutation. Each chromosome will represent a solution for the problem encountered. The original genetic algorithm, introduced by Holland, represents chromosomes as a string of bits. Other representations, such as real values or trees, are possible as well. The following example explains how genetic algorithms can be used to solve problems:

Example 2.1

Let f be a function defined as $f(x) = -x^2 + 8x - 4$. Our goal is to find a (integer) value of x that optimizes f in the interval 0 to 31. We can represent each solution as a binary representation of x. Since x ranges from 0 to 31, we only need 5 digits. Therefore, our chromosome will have 5 genes. For evaluation, we can simply use f, since we want its maximum value. Selection, then, will be based on f, and after many applications of selection and genetic operators, the population will eventually converge to the desired value that maximizes f.

Selection methods are discussed in the next subsection, followed by a description of genetic operators. Application of genetic algorithms in evolving sets of rules, artificial neural networks, and lisp programs are also discussed subsequently.

2.2.1 Selection Methods

One of the important tasks in genetic algorithms is to select individuals from the population that will create offspring for the next generation. Selection should highlight the fitter individuals in the population to be passed on to the next generation; therefore, repeated application of selection and other operators presumably evolves a better solution. However, selection requires great attention as it requires a balance between increasing population diversity and increasing the fitness of the population's individuals. That is, too strong of a selection method will yield a convergence to a suboptimal solution, whereas too weak of a selection method will result in a too slow evolution. Many selection methods have been proposed in the genetic algorithm literature; below is a detailed review of the most often used methods.

Fitness-Proportional Selection

Holland used this method in his original genetic algorithm. Individuals are selected according to their proportional fitness to the other individuals in the population. An individual is selected with probability equal to its fitness divided by the average fitness of individuals in the population. Usually, the fitness-proportional method causes premature convergence. It emphasizes exploitation at early stages of the evolution process, where exploration would be more helpful. In the first generations, individuals tend to be diverse and their fitness distribution has a high variance. A fitness-proportional method will select the fittest among those, leaving a large region of the search space unexplored. Individuals in later generations will have very similar fitness, and selection will not explore more search space.

Fitness-Sharing Selection

Fitness-Sharing was introduced by Goldberg and Richardson (1987). Unlike the fitness-proportional method, this method will punish individuals that share the same fitness with many individuals. Moreover, distinct individuals will receive more selection pressure, due to their "uniqueness", even if they have low fitness. This will allow the population to converge to several optima instead of only one.

Sigma Scaling

To solve the premature convergence problem associated with the fitness-proportional selection method, sigma scaling was proposed by Forrest (1985). Instead of using the fitness value of individuals to base the selection on, a function of the individual's fitness with the fitness' variance is used. The idea is to make this function less sensitive to the fitness value when the variance is high, so no exploitation happens at early stages and make it more sensitive to the individual's fitness when the variance is small, thereby allowing evolution to progress.

Boltzmann Selection

This method is used to give a variable emphasis to fitness during selection as the evolution progresses. The key idea is to allow a variable, usually called temperature, to control the selection. High temperature values at the beginning of the search allow more diverse individuals to appear in the population. Lowering the temperature as evolution progresses allows more highly fit individuals to remain in the population.

Rank Selection

Another method used to prevent premature convergence is rank selection. Individuals are ranked according to their fitness values. Instead of using fitness values in selection, rank is used. This method has a good side and a bad side. It avoids the premature convergence problem, but it also disregards the values of the individual's fitness when its helpful to know how far one individual is from its nearest neighbor. Furthermore, individuals need to be sorted in order to be ranked, which drastically increases the execution time of selection.

Elitism

Kenneth De Jong (1975) introduced elitism to be used in selection. Elitism keeps a set of the best individuals at each generation and prevents them from being not selected or destroyed by mutation or crossover. This decreases the disruption effect of genetic operators and improves the evolution progress significantly.

Tournament Selection

In this method two individuals are selected randomly from the population to play a tournament. The fittest is selected for the next generation. To maintain diversity in the population, sometimes the worst individual is selected instead, according to some given probability. The individuals then are returned to the population, and another two individuals are selected.

Steady-State Selection

This method is similar to the elitism scheme described above. In steady-state selection, only a few individuals are replaced at each generation. A small number of the least fit individuals are selected to be replaced by the new offspring generated from crossover and mutation. This is useful in incremental learning problems, where each step of the generation is a constructive step towards the goal and, thus, only a few individuals are replaced.

2.2.2 Genetic Algorithms' Operators

After selection is made, genetic operators are applied to the selected individuals of the population. While selection emphasizes exploitation, genetic operators stress exploration in the population. Although a variety of operators have been introduced in genetic algorithms, many depend on the representation used or the problem being solved. Crossover and mutation are the most commonly used operators.

Crossover

As described in section 2.1.3, crossover acts as the main feature of genetic algorithms. Two individuals are selected and exchange parts of their genes to form an offspring. In the original crossover, introduced by Holland (1975), a point is chosen at random, where the swap of genes is made as illustrated in Figure 2.3.a. This is called single-point crossover. Sometimes chromosomes are disrupted by a single-point crossover, especially when the chromosome is long. Another variation of crossover is to use two points instead, where the genes are exchanged between them, as shown in Figure 2.3.b.

Multiple-point crossover is also used by many researchers. The most commonly used type is uniform crossover. It simply replaces each gene from each chromosome with the probability p, where p usually ranges from 0.5 to 0.8.

Mutation

Mutation is considered a minor operator in genetic algorithms; however, it has a major role in evolution strategies and evolutionary programming. Originally, mutation was considered to add more diversity to the population. By swapping some of the genes' values with other alleles, mutation creates new individuals that have not been seen in the population before. Many researchers believe that mutation could play a greater role in genetic algorithms. Spears (1993) showed that both mutation and crossover have the same ability for disrupting chromosomes. Muhlenbein argued that a hill-climbing strategy is better than genetic algorithms with crossover, and that mutation has been underestimated (Mühlenbei 1992). It is not clear which operator should be used in a particular representation with a particular selection method; therefore, how much mutation or crossover should be used is

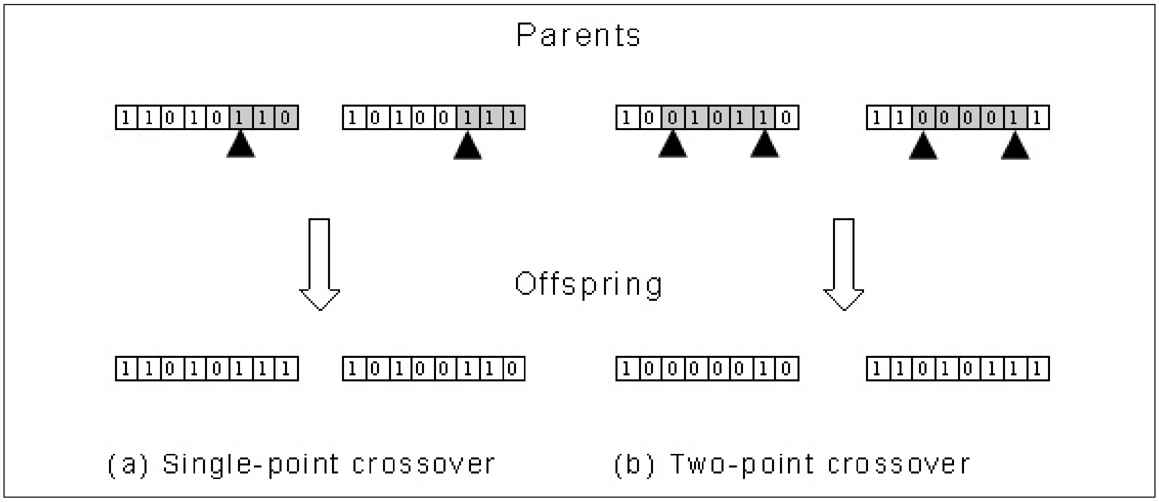


Figure 2.2: Single-point and two-point crossover

still an open research question.

Many other operators have also been explored. For example, De Jong introduced a crowding operator (De Jong 1975), where the newly generated offspring replaces individuals that are similar to itself in the population. This prevents too many similar individuals from existing at the same time; i.e. prevents crowding. This is useful in introducing diversity into the population as evolution takes place.

2.2.3 Chromosome Representation

Genetic algorithms have a broad spectrum of application because of their flexibility in representing many problems in the genetic paradigm. In its simplest form, a chromosome can be represented as a string of bits. However, any kind of data can be used as long as it provides a meaningful representation and the genetic operators are well-defined. Researchers have chosen to use many representations, depending on the problem they encountered. In the context of robotic learning, the most commonly used representations are if-then rules, neural networks, and lisp programs representations. These representations are described in detail below.

2.2.3.1 Evolving Set of Rules There are two approaches that have been used in representing rules in a population. The first approach is to represent each individual with one rule; this is referred to as the Michigan approach. The second approach is to represent each individual using a set of rules. This is referred to as the Pitt approach.

Michigan Approach

One of the early attempts at using genetic algorithms for evolving a set of rules was the learning classifier system (Holland and Reitman 1978; Holland 1986). In the learning classifier system, if-then rules (called classifiers) are evolved using genetic algorithms. Each rule is represented by a string consisting of 0, 1 and #. The population of rules is evaluated by applying it to a specific problem of stimulus-response cycles. Individuals correspond to parts of the solution; i.e. the whole population receives the same fitness. Each rule has an associated fitness on which selection is based. However, assigning a fitness for each rule is not an easy task, and results in a problem usually called the credit assignment problem. The bucket brigade (Holland 1986) is a bedding algorithm that can be used to solve this problem. Different approaches have been taken by other researchers for tackling the credit assignment problem. For Example, Wilson created the simplest possible classifier system called the Zeroth Order Classifier System (ZCS) (Wilson 1994). Based on ZCS, Wilson gradually added more components to his system, generating a more complex classifier system (XCS) in which the strength of each rule is calculated more accurately, based on its contribution to the whole solution (Wilson 1995).

Giordana, Saitta, and Zini (1994) and Giordana and Neri (1996) developed a system which uses genetic algorithms to evolve a set of rules called REGAL. Rules are represented as a disjunctive normal form[1]. A specially designed operator called universal suffrage is used to cluster individuals. Thus, similar individuals will breed together (speciation).

Dorigo and Colometti (1998) developed a system called ALECSYS. This system has been used to learn to control an autonomous robot moving in a real environment. It breaks down the task into subtasks that are learned by a learning classifier system. As the decomposition is done by a human, each subtask can have its own credit assignment.

[1] Rather than representing a rule as $(A \rightarrow B)$, in disjunctive normal form a rule is represented in the form of $(A \vee B)$.

Pitt Approach

Smith (1980, 1983) developed another system (SL-1) that uses genetic algorithms to evolve rules. Rather than the population consisting of rules, each individual in the population consists of a set of rules that forms a whole solution. In a sense, this approach is analogous to a bit representation in which each chromosome represents a solution rather than a part of the solution. Although there is no credit assignment problem that may arise, sometimes a problem may occur in that a high fitness value can be assigned to a bad rule that happens to be in a chromosome with other good rules. This problem is referred to as hitchhiking (Das and Whitley 1991). Another problem that may also be encountered is that individuals tend to grow uncontrollably as they evolve. This problem is often called "bloat". Smith (1980) defined a mechanism that penalizes long individuals to prevent bloating.

Subsequent to Smith's system, a refined system, SL-2, was introduced by Schaffer and Grefenstette (1985), followed later by a system called SAMUEL (Grefenstette 1989; Grefenstette, Ramsey, and Schultz 1990). SAMUEL uses ideas from various classifier systems and SL-2. Individuals consist of sets of rules, and each rule has an associated strength. The associated strength is used in recombination to cluster individuals, which then form the next offspring. Interestingly, rules in SAMUEL are represented by a more human-readable syntax rather than binary strings. SAMUEL also introduced other operators, Lamarckian operators, into the evolutionary process. These operators use a rule's strength to specialize, generalize or delete it in the evolutionary process. A more detailed treatment of these operators is given in Chapter Three.

Evolving rules are also used in the domain of concept learning. One of the early systems developed in this domain was GABIL, a system developed by De Jong, Spears, and Gordon (1993). It is similar to SL-1, but rules are represented in disjunctive normal form. Another system that uses the Pitt approach is GIL (Janikow 1993). GIL utilizes a number of special operators for concept learning. For example, it has both generalization operators and specialization operators. Bloat is controlled nicely in GIL. This is because applying generalization more often at the beginning results in more complex individuals, and specializing further later on shortens the long individuals.

2.2.3.2 Evolving Artificial Neural Networks An artificial neural network is a model used in machine learning inspired by human neurons. A neural network consists of nodes and links between them. Weights are assigned for each link, and a node is "fired" if its corresponding weight is greater than a given threshold. Input nodes (layer) represent the problem specification, whereas the output nodes give the results of the task which needs to be learned. Between the input and the output nodes there are hidden nodes (layers); the more complex the problem, the more hidden nodes are needed to learn a specific task. One of the most successful algorithms used in learning artificial neural networks is the back-propagation algorithm (Rumelhart, Hinton, and Williams 1986).

Genetic algorithms are used in learning artificial neural networks in three ways. The first way is to learn the weights of the links, i.e., substituting for the back-propagation algorithm. The second way is to learn the topology of the network, that is, how many nodes exist in each layer. The third way is to learn both the weight assignment and the topology of the network. Examples of the use of these three ways are given below.

One of the first attempts at using genetic algorithms in learning weights of artificial neural network was done by Montana and Davis (1989). The topology of their network was fixed and was fully connected. They were interested in classifying underwater sonic waves as interesting and not interesting, based on the judgment of experts. They concluded that using genetic algorithms is better (and faster) than back-propagation in terms of sum of squared errors.

GENITOR (Whitley and Kauth 1988; Whitley 1989) uses genetic algorithms to evolve weights of an artificial neural network. In this system, chromosomes consist of the real-valued weights of the network and a crossover gene. The crossover gene controls the crossover and mutation probability during the evolution process. Along with the "adaptive" mutation and crossover operators, a steady-state method is used for selection.

Miller, Todd, and Hegde (1989) used genetic algorithms to evolve artificial neural networks' topology. Individuals of the population represent the networks' topology. Evaluation was done by applying the back-propagation algorithm and the fitness was the sum of squared errors.

The SANE (Symbiotic Adaptive Neuro-Evolution) system uses genetic algorithms to build neural networks (Moriarty and Miikkulainen 1996a; Moriarty and Miikkulainen 1998). Each individual in the population represents a single neuron of the network. The evolution process begins with selecting a set of neurons from the population, then forming a functional network from them. The formed network is then evaluated on the given task, and each participating neuron receives the appropriate fitness value. Individuals (neurons) may participate in more than one network; therefore their fitness is calculated cumulatively. This maintains diversity in the population since several types of neurons, which are formed from evolutionary pressure, are needed to form a neural network. The SANE system has been used in solving many problems. For example it was used in discovering complex strategies in Othello (Moriarty and Miikkulainen 1995), and on an obstacle avoidance problem in autonomous robots (Moriarty and Miikkulainen 1996b).

2.2.3.3 Evolving Lisp Programs John Koza used genetic algorithms to evolve Lisp programs. He called this method "genetic programming" (Koza 1992; Koza 1994). Lisp programs can be expressed as a parse tree. The program's operations can be ordered in the parse tree in such a way that the root is the operation, and its children are either a parameter or a sub-tree. Figure 2.3 shows an example of a parse tree.

The algorithm starts by initializing a population of programs (trees). This can be done by first choosing a number of operations, for example $\{+, -, *, /\}$, and a number of terminals. Trees are then randomly generated from those operators and terminals. Trees must correspond to correct programs. Evaluation of programs is achieved by choosing a set of inputs with known correct outputs, and measuring how many correct answers are produced by the program for this set. The fittest is the program that generates the highest number of correct answers. After evaluating the programs, parents are selected to generate offspring by crossover and mutation. Crossover is done by selecting a breaking point in the parents' trees, and then exchanging the sub-trees under the breaking points between each other. The original algorithm that was introduced by Koza did not use mutation. Koza, instead, utilized a huge population to guarantee enough diversity of programs so that exchanging parts of them would lead to the correct program. With this method, the chromosome size keeps increasing as evolution advances. Thus, individuals (programs) become more complex as

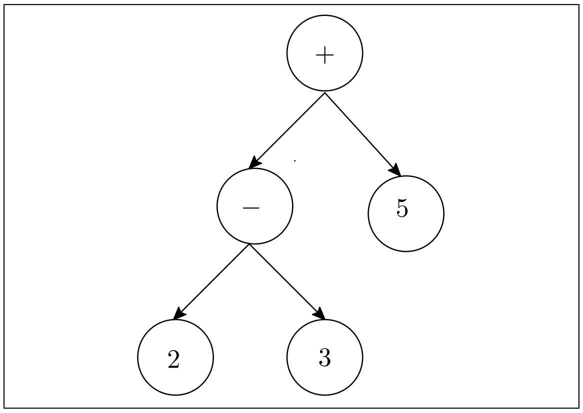


Figure 2.3: An example of a parse tree

they evolve.

2.3 COEVOLUTION

Coevolution is an extension of standard evolutionary algorithms, in which two or more populations are evolved together. The fitness of individuals in one population depends on the fitness of individuals in the other population(s). Thus, fitness is dynamic rather than static.

Coevolution has received a great deal of attention from many researchers. One of the first attempts at using coevolution was in the 1980's by Axelrod (Axelrod 1984; Axelrod 1989) who examined the Prisoners' Dilemma problem with the aim of investigating the cooperation and competition behaviors of evolution.

Some theoretical work has also been done in an attempt to understand coevolution dynamics. Ficici and Pollack (2000) used evolutionary game theory to analyze some aspects of coevolutionary dynamics. Many restrictions, such as infinite population size, were applied to coevolution in order to model it with evolutionary game theory. Wiegand, Liles, and De Jong (2002) extended Ficici and Pollack (2000)'s work by investigating the dynamics of coevolution with more relaxed assumptions. Ficici and Pollack (2001) and Bucci and Pollack (2002) created a mathematical framework for studying coevolutionary dynamics. Their model was based on the Pareto dominance concept[1]. Some analysis of coevolution learning has also been done by Shapiro (1998). He hypothesized that coevolving test cases with solutions does not necessarily improve generalization. By mathematically analyzing a simple toy problem, he showed that coevolution results in oscillation with low fitness states; however, it yields an improved generalization if the learning parameter is suitably scaled.

Coevolution may take the form of competition between two populations, where the fitness of individuals in one population is the inverse of the fitness of individuals in the other population. Such competition creates an arms-race between individuals in the two populations, and thus, they become fitter as they evolve. This arms-race may create a problem often referred to as the "Red Queen Effect"[2], in which the progress of one population is due to a change in the other population's fitness and not to its own high fitness. This makes progress in coevolution hard to measure. Cliff and Miller (1995) proposed some solutions to this problem. Paredis (1997) also analyzed this problem. Stanely and Miikkulainen (2002) actually introduce a method, the "dominance tournament" method, for measuring progress in coevolution. The basic idea of this method is to keep track of the fittest individual of each generation (champion) that beats all previous generations' champions. The number of champions which beat all previous champions found in the entire evolution process reach what is referred to as the "dominance level". The progress of coevolution is measured by how many dominance levels learning can achieve.

Many experiments have been performed to assess the performance of coevolution. For example Panait and Luke (2002) experimented with various fitness functions in the game of Nim. Pagie and Mitchell (2002) did a comparative study of evolutionary and coevolutionary searches. Their results were in favor of coevolution. Panait, Wiegand, and Luke (2004) did an analysis of cooperative coevolution in the function optimization domain. Popovici and Jong (2006a) analyzed coevolutionary dynamics based on trajectories of best-of-generation individuals.

Coevolution can be categorized into two models, competitive and cooperative models.

[1] "An individual is (Pareto) dominated if there is some other individual which does at least as well as it does against all others and better against at least one" (Bucci and Pollack 2002).

[2] The name comes from Lewis Carroll's book "Alice in Wonderland", in which the Red Queen explains to Alice that running made her remain in the same place because the landscape is moving with her.

The next sections review those models in further detail.

2.3.1 Competitive Coevolution

Competitive coevolution is based initially on two (or more) populations, each of which tries to overwhelm the other(s). This behavior is seen widely in nature, as with predator-prey or parasite-host relationships. Competition between species enhances their evolution, because each species tries to defeat the other; thus, in one generation one species might conquer the other species, while in another generation another species takes over. This kind of oscillation between species eventually creates high-fitness individuals.

Hillis's work on sorting networks was one of the first successful attempts in applying the coevolution principle (Hillis 1990). In his work, a population of sorters (hosts) was coevolved with a population of input vectors (parasites). The hosts tried to sort a vector of numbers by building sequences of comparison-exchange pairs of numbers. The parasites attempted to find difficult input test cases for the hosts to sort. The parasites were evaluated according to their performance in sorting the hosts' test cases. The fitness of the hosts was complementary to the parasites' fitness. Hillis found that coevolution provides two advantages: it prevents the population from being stuck in a local optima, and it increases the efficiency of testing. Using a coevolutionary approach, a sorting network of 16 numbers with only 61 exchanges was discovered. The results were interesting in that the best known sorting network at that time had 60 exchanges.

Paredis (1994b, 1994a, 1996) applied competitive coevolution to neural network learning and constraint satisfaction problems. In line with Hillis's work, two populations were evolved; one population represented solutions to the given problem (prey), and the other population provided test cases (predator). Paredis used the steady-state method for selection, and introduced a new mechanism for fitness evaluation called the life-time fitness evaluation (LTFE). Instead of evaluating predators against prey in the same generation, in LTFE, prey are evaluated against predators from previous generations as well. This allows solutions to be evaluated against a larger number of test problems, which boosts the search. In the neural network problem, Paredis evolved the weights of a fixed topology network. He experimented with different multi-point crossovers and compared the results between traditional genetic algorithms and coevolution. Only coevolution achieved 95% accuracy of a neural network[1]. The other problem Paredis used to demonstrate the power of coevolution is the 50 queens problem[2]. The traditional genetic algorithm failed to provide a solution, while coevolution succeeded 7 out of 10 times in giving a valid solution to the 50 queens problem.

Miller and Cliff (1994) experimented with a predator-prey simulation of robots, where robots are represented by neural networks, and the whole network (topology and weights) is evolved. In their work, they showed a drawback of coevolution in that an increase in individual fitness does not necessarily mean an increase in performance ("Red Queen effect"). Consequently, they provided tools for tracking the progress of coevolution. Paredis (1997) also gave insight into this problem in evolving cellular automata. Wahde and Nordahl (1998) experimented with predator-prey robots as well. Unlike in Miller and Cliff's work, they used both open and confined regions (arenas) and focused on giving insight into the dynamics of coevolution strategies.

Another example of successful use of coevolution is Sims's virtual creatures (Sims 1994). Sims developed a computer graphics simulator for the physics of robots composed of rectangular solids and several controlled joints. Then he coevolved the structure of robots with the robots' control. Some of the interesting creatures are able to walk.

Pollack and his colleagues extended Sims's work to build actual robots (Pollack et al. 1999; Pollack et al. 2000). They used Lego pieces (and other "disassemblable" pieces) to build their robots based on the simulation. One of their goals was to exhibit a kind of locomotion. Some of the behaviors they achieved are crawling and ratcheting. In other experiments, they coevolved the Lego bricks to form building structures conforming to the physical properties of the Lego plastics (Funes and Pollack 1998). Interesting buildings such as long bridges and a horizontal crane arm were created.

Rosin (1996, 1997) introduced several heuristics to boost the performance of coevolution. Competitive fitness sharing was introduced to evaluate individuals. In competitive fitness sharing, rather than evaluating hosts on the number of parasites that they defeat, hosts are

[1] With 100,000 generation limit for coevolution, and 50,000 for standard genetic algorithm.

[2] The 50 queen problem is the problem of putting 50 chess queens on a 50x50 chessboard such that none of them is able to capture any other using the standard chess queen's moves

rewarded when they defeat parasites that few other hosts can defeat, even if they don't defeat many parasites. In other words, individuals that defeat only one highly-fit parasite are found interesting and deserving of survival in the population. The same idea can be incorporated when choosing a sample of individuals for evaluation. Rosin called this method "shared sampling". With this method, in order to evaluate a host, a sample of parasites should be selected which few hosts can defeat. This enhances the coevolutionary process and yields strong hosts in the population. The other heuristic pioneered by Rosin is the "hall of fame". This method extends the elitism method discussed previously for the purpose of testing, i.e., a set of elite individuals is taken from all generations seen to a certain point, and are used to test or evaluate individuals in the current generation. Note that this set is not necessarily present in the population. The above heuristics have been used to support coevolution in solving a number of game learning problems, in particular Nim, 3D Tic-tac-toe, and Go.

Coevolution has also been applied to many games. Angeline and Pollack (1993) applied coevolution in tic-tac-toe. Moriarty and Miikkulainen (1995) applied coevolution to discover complex Othello strategies. More recently, Lubberts and Miikkulainen (2001) coevolved two (neural network) players of Go, playing on a small board. An early attempt of Reynolds (1994) with the game of Tag is another successful application of competitive coevolution. A complex behavior was attained, although a specific selection and evaluation were used to maintain diversity in the population. Blair, Sklar, and Funes (1998) used competitive coevolution in evolving neural networks of players for a game called "Tron", a game in which two robots inside an arena move at a constant speed, making only right angle turns and leaving solid wall trails behind them. The winner is the one who does not crash into the trailed walls. The coevolutionary approach outperformed a genetic programming algorithm and interesting behaviors were also noted.

Pollack and Blair (1998) used a simple coevolutionary approach in the game of backgammon with significant success. In a comparison with Tesauro's system TD-Gammon (Tesauro 1992), which is considered one of the most successful applications of temporal difference in machine learning, the simple coevolution wins 40% of the games at generation 10,000. Similar results were accomplished by Darwen when using coevolution in backgammon (Darwen 2001).

Juillé and Pollack introduced the “Ideal trainer” method(Juillé and Pollack 1998; Juillé 1999). The idea is to expose learners to problems just beyond what they know how to solve although defining a “little more difficult” problem is somehow subjective and acquiring some knowledge of the problem being solved. One of the applications they used to demonstrate this method is evolving cellular automata. The task to be performed on cellular automata was to find rules for swapping the cells’ value with its neighborhood in such a way that a particular density of one value is reached. The most commonly used configuration has 149 cells of binary numbers, with a neighborhood of 3 cells. Finding rules to achieve 1’s density of 0.5 is considered difficult. Juillé and Pollack applied the coevolutionary model with the ideal trainer and achieved an accuracy of 0.83, which is the best known so far. Juillé and Pollack also applied coevolution in solving the intertwined spirals classification problem (Juillé and Pollack 1996).

One of the most successful applications of competitive coevolution is coevolving autonomous robots. Floreano and others used coevolution in evolving predator-prey behavior in two robots (Floreno 1998; Floreno, Nolfi, and Mondada 1998). The experiments were done on a simulation that was specially designed for their robots. Instead of using a mathematical model for the sensors and the motors of the robot, they trained their simulation to act like a real robot, i.e. with noisy sensors and motors. Their robots showed an interesting behavior. The red queen effect was present in early generations. They discuss this thoroughly in (Floreno and Nolfi 1997).

Competitive coevolution has been applied in many domains with successful results. It has been shown that competitive coevolution produces potentially high-quality solutions for many problems. The dynamics of competitive coevolution bootstrap individuals’ fitness in the population. Some problems may arise, however, such as the red queen effect, in which the fitness landscape of each population is continuously changed by the competing population.

2.3.2 Cooperative Coevolution

Although evolutionary algorithms have been successfully applied to many problems, their success have been limited with regards to solving complex problems which need to be decomposed into smaller problems in order to be solved. Cooperative coevolution enables evolutionary algorithms to scale up for such complex problems. Each individual represents only a partial solution to the problem. Complete solutions are formed by grouping several individuals together. Thus, the goal of an individual is to find one part of the solution and cooperate with other individuals with other partial solutions to create a whole solution.

Evolving cooperative populations was applied to a job-shop scheduling problem by Husbands (1991, 1994). In a typical job-shop scheduling problem, each job consists of several operations that must be processed in a specified sequence. This sequence is also referred to as a process plan. One machine can process one operation at a time and operations cannot be interrupted. The problem is to determine the sequence of operations to be processed on each machine. Husbands evolved process plans for a particular component to be manufactured. Each process plan was represented in a separate population. The plans (populations) interacted with each other as they shared the same resources (machines). If several plans competed for the same resource during an overlapping time interval, an arbitrator which is coevolved along with the other populations resolved the conflict.

Paredis (1995) used a cooperative model, also referred to as symbiotic, to coevolve two populations. The first population contained permutations (orderings) of a solution, and the other one consisted of the solution of the problem to be solved. The order of genes in a solution may have an important role in enabling a genetic algorithm to solve the problem. This ordering may be neglected when using genetic algorithms. The inversion operator, introduced by Holland (1975), tackled this problem; however, its success was limited since inversions were applied randomly (Paredis 1995). Paredis proposed the reordering of genes by coevolving orderings with solutions. First, a permutation is selected from the permutations' population. In addition, two individuals from the solution's population are selected. The genes of the selected individuals are reordered, according to the selected permutation, and then the genetic operators, one-point crossover and mutation, are applied to generate offspring. The fitness of the offspring is calculated, and the fittest are added to the solutions' population. This process is repeated a number of times. The average of how good the solutions are in contrast to their parents' determines the fitness of the permutation.

Potter (Potter 1997; Potter and De Jong 2000) developed a cooperative model in which a number of populations (species) looked at different decompositions of a problem. The idea of his approach was to have each population specialize on a particular component of the problem as an emergent property of the model, rather than decomposing the problem into different parts by hand. In cases such such these, the best individuals in each population form a set of "representatives" that is used in evaluation; individuals are rewarded based on how well they cooperate with the representatives of other populations. Potter also introduced the birth and death of species. This property allows the coevolved subcomponents to emerge into an appropriate number of subcomponents as they evolve. Potter applied his approach to many problems, for example string cover, cascade neural networks, and concept learning. He found many interesting results. First, cooperative coevolution was able to discover important environmental niches of the problem being solved; second, subcomponents with the appropriate level of generality emerged along with the available niches; and third, coevolved subcomponents were able to adapt to changing fitness landscapes.

Cooperative coevolution was also applied in evolving robot behaviors. Potter, De Jong, and Grefentette (1995) applied cooperative coevolution to evolve robot behaviors using the SAMUEL system. They experimented with a food gathering problem, where a robot moves around an obstacle-free room, with food pellets appearing at random locations and times. The robot must consume food pellets to replace lost energy. To increase the complexity of the task, another hand-coded robot is competing with the robot for the food pellets. Energy decreases as the robot moves and, to a less extent, as time passes, and increases if the robot gets the food. The fitness of the robot is determined by its average energy over a set of episodes. The authors coevolve two populations, which they initialize with a set of primitive roles, one with food present and another with food absent. In their comparison with standard evolution, the coevolutionary approach achieved better results. An interesting phenomena noted in their experiment is the emergence of a third species that has an unusual behavior, that is, "If the robot is not hungry then it will not waste energy seeking food." (Potter, De Jong, and Grefentette 1995)

Daley, Schultz, and Grefenstette (1999) also explored the application of coevolution to a learning problem for mobile robots, also using the SAMUEL system. They investigate various coevolutionary models in a tracking task with fuel constraints. A robot must "track" another robot in an arena. As the robot moves, it loses energy, and thus needs to refuel by "docking" into a docking station. Beside tracking and docking, the robot must learn when to dock and when to track – that is the "executive" task. Two experiment regimes were studied: mutual and independent regimes. In the mutual regime, the three tasks are coevolved together, so the best-so-far executive task is used in the evaluation of the tracking and docking tasks. In the independent regime, the docking and tracking are learned independently and then used to evaluate the executive task. They found out that the mutual regime failed to solve the problem due to the fact that the executive task starved the docking task, while the independent regime solved the problem satisfactorily.

Moriarty and Miikkulainen (1998) developed a "neuro-evolution" system (called SANE), in which a neural network is trained using an evolutionary algorithm to solve a robot obstacle-avoidance problem. The robot has to move inside a maze and to avoid thumping the walls. The sensors of the robots are used as input to a neural network, which outputs the speed of each wheel of the robot. The network is trained to move the robots using a symbiotic coevolutionary approach. They coevolved the hidden neurons of the neural network and a "blueprint" of the network. The blueprint determines which neurons from the evolved population must form the neural network. In their results, neural networks with SANE outperformed those with non-coevolutionary approaches (standard evolution with the elitist method) in terms of the number of generations needed to achieve a certain performance.

Recently, many researchers have successfully applied a cooperative coevolution in various domains. Wiegand, Liles, and De Jong (2001) applied cooperative coevolution in function optimization problems and empirically investigated many problems that emerge with coevolution such as credit assignment, selection pressure, and collaborators' pool size problems. Popovici and Jong (2006b) investigated the effect of interaction frequency between coevolved populations in a function optimization problem. Bucci and Pollack (2005) used Pareto coevolution to find a global optimal solution for a function optimization problem for which conventional coevolutionary methods failed. Bugajska and Schultz (2000) used coevolution in learning collision-free navigation of a micro-air vehicle. Yong and Miikkulaainen (2001) investigated cooperation of robots in a predator-prey problem. They used coevolution to teach a team of three predators (neural networks) to chase prey. They also explored the effect of the presence of communication between the coevolved robots.

Scalability is an important issue to consider when solving problems with evolutionary algorithms. It is difficult to design a model that can solve complex problems efficiently. Cooperative coevolution has been proven to be a good model for solving many problems. Furthermore, it gives an insight into solving complex problems, as well, by allowing subcomponents of the problem being solved to interact and cooperate to solve this problem.

2.4 REINFORCEMENT LEARNING

In a reinforcement learning problem, an agent must learn behaviors through trial-and-error interactions with an environment (Kaelbing, Littman, and Moore 1996). The agent receives perceptions from the environment and performs some actions. After performing some actions, a critic provides a reinforcement. Depending on the agent's actions, the critic may punish or reward the agent. The agent's goal is to learn a policy (a mapping from states to actions) that maximizes its rewards.

Reinforcement Learning is a learning paradigm that specifies a class of problems rather than a technique. Evolutionary algorithms, which are capable of solving reinforcement learning problems, can also be classified as reinforcement learning techniques. Kaelbing, Littman, and Moore (1996) classify reinforcement learning methods into two types: methods that search the utility value of taking actions in states of the world, and methods that search behaviors (policies). Evolutionary algorithms can be categorized into the latter class(Kaelbing, Littman, and Moore 1996).

In reinforcement learning problems, many algorithms have been developed to learn the optimal policy of an agent. These algorithms can be seen as online dynamic programming algorithms. Value iteration (Bellman 1957) is one example of these algorithms, where agents learn, iteratively, a set of action values based on the results of the agent's actions. Alternatively, a task may be learned by updating the policy's functions without learning the agent's action values. This approach is called policy iteration.

Temporal difference (TD(λ)) ,which is introduced by Sutton (1984), is another method of learning in reinforcement learning. Reinforcement learning problems may have a temporal credit assignment where it is hard to assign a reward value for consecutive actions in a given run. Temporal difference is an attempt to solve this difficulty. This method learns value functions of the world's states by predicting the difference between two successive states. That is to say, the value function of a state is updated not only according to the final reward, but also using the addition of the difference between the value of the state and its successor. The parameter λ controls how an instance of experience affects its preceding states. If $\lambda = 0$, the value function is updated for the most recently visited state. When $\lambda = 1$, on the other hand all the states are updated according to the number of times they have been visited.

In the late eighties, Watkins (1989) combined a trial-and-error sampling of the problem space with an iterative temporal difference method for updating the function values.This method is called Q-learning. In Q-learning, the agent maintains a value for each state and action pair that represents a prediction of the worth of taking that action from that state. These values are represented by a $Q(s, a)$ function, hence the name Q-learning. The Q function is updated according to the difference between the values of the current state and the best action's value of the resulting state (Watkins and Dayan 1992).

Reinforcement learning techniques have had some notable successes. One example is Teasuro's backgammon player TD-Gammon (Tesauro 1995). It used a temporal difference method to learn how to play backgammon by playing against a version of itself. Human-designed features were added to TD-Gammon, which greatly improved its performance. TD-Gammon played competitively with a top-ranked human player, and it is considered one of the best players in the world (Tesauro 1995). Another successful application of reinforcement learning is Crites and Barto (1996) algorithm in learning an elevator scheduling task. In a simulation of four elevators in a ten-floor building, Q-learning was used to learn the best scheduling algorithm to provide the minimum overall waiting time for passengers. The algorithm was slightly better than the best known algorithm and twice as good as the controller most frequently used in real elevator systems.

Many other reinforcement learning methods have been developed in the past decade. the reader may refer to (Kaelbing, Littman, and Moore 1996), (Moriarty, Schultz, and Grefenstette 1999), and (Sutton and Barto 1998) for more details.

2.5 ROBOCUP SOCCER

The Robot World Cup (RoboCup) is a yearly competition of soccer which is played by robots. The competition attracts many universities and organizations all over the world. It encourages initiative in artificial intelligence and intelligent robotics research by providing a standard problem where a wide range of technologies can be integrated and examined (Kitano et al. 1995). RoboCup's ultimate long-term goal is stated as follows:

> "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of the FIFA, against the winner of the most recent world cup for human players." (Kitano and Asada 1998)

The RoboCup organization has introduced several robotic soccer leagues, each of which focuses on a different abstraction level of the overall problem. Currently, the most important leagues are the following:

- Middle Size Robot League (F-2000). In this league each team consists of a maximum of four robots about 75cm in height and 50cm in diameter. The playing field is approximately $9x5$ meters and the robots have no global information about the world.
- Small Size Robot League (F-180). In this league each team consists of five robots about 20cm in height and 15cm in diameter. The playing field's size is that of a ping pong table. An overhead camera provides a global view of the world for each robot.
- Sony Legged Robot League. In this league each team consists of three Sony quadrupled robots (better known as AIBOs). The playing field is similar in size to that for the Small Size League. The robots have no global view of the world but use various colored landmarks which are placed around the field to localize themselves.
- Simulation League. In this league each team consists of eleven software robots which operate in a simulated environment.
- Humanoid League. RoboCup introduced Humanoid League for the first time at the RoboCup-2002 robotic soccer world championship in Fukuoka, Japan. The robots have a size range from 40cm to 180cm and can perform some basic moves of soccer, such as shooting.

2.5.1 Learning in RoboCup

RoboCup provided a real challenge for machine learning. Despite the fact that non-learning programs still play better soccer[1], many learning-based programs have shown a competitive performance. Many researchers started tackling soccer obstacles since 1996. For example, one of the early works is in learning passing behavior using neural networks (Matsubara, Noda, and Hiraki 1996). The following subsections overview some of the learning methods used in RoboCup.

2.5.1.1 Reinforcement Learning Approaches One of the remarkable works on learning in the domain of soccer is (Stone 1998). Stone developed a multi-agent machine learning paradigm called layered learning. This paradigm was designed to enable agents to learn how to work together towards a common goal in an environment that is too complex to learn direct mapping from sensors to actuators. Stone's layered learning provides a bottom-up hierarchical approach to learning agent's behaviors at various levels of the hierarchy. In Stone's framework, the learning task at each level directly affects the learning at the next higher level. Stone uses a three-layer hierarchy. The first layer contains low-level individual agent skills such as ball interception. The second layer contains multi-agent behaviors at the level of one player interacting with another, for example pass evaluation (likelihood of a successful pass). When learning this behavior, the agents can use the learned ball-interception skill as part of the multi-agent behavior. The third layer contains collaborative team behaviors such as pass selection (which teammate should get the ball). Here the agents can use their learned pass evaluation skill to create the input space for learning the pass selection behavior. Each task in each layer may be learned differently. For instance, the ball-interception behavior is learned using a neural network, whereas the pass evaluation behavior in the second layer is learned using the C4.5 decision tree algorithm (Stone and Veloso 1998). Subsequently, the pass selection behavior in the third layer is learned using a new multi-agent reinforcement learning method called Team-Partitioned Opaque-Transition Reinforcement

[1] In the simulation league, the RoboCup 2002 winner, TsinghuAeolus (Yao et al. 2003), and the RoboCup 2003 winner, UvA Trilearn (de Boer and Kok 2002) are both heavily hand-coded though they provided other important aspects of the game such as synchronization and system analysis issues.

Learning (TPOT-RL). This method maximizes the long-term discounted reward in multi-agent environments where the agents have only limited information about environmental state transitions (Stone, Veloso, and Riley 1999).

Since the work of Stone (1998), many learning methods have been exploited in the robot soccer domain. Some of these works have been applied to softbots and others to real robots. For example, Asada, Uchibe, and Hosoda (1999) used reinforcement learning to enable learning of pass and shooting behaviors in vision-based robots. Mehta (2000) also used reinforcement learning to improve soccer robots by incorporating approximation techniques to reduce the size of the input space. Yao, Chen, and Sun (2002a) combined Q-learning with adversarial planning to enable learning of a kicking behavior.

Going a step further, Kostiadis (2002) used Kanerva's sparse distributed memory together with Q-learning to provide a decision-making mechanism for soccer agents (Kostiadis 2002). An emergent cooperation behavior between agents has been realized, although there is no notion of communication between the agents. Such cooperation behavior was present because of the decision-making mechanism the agents learned.

(Riedmiller and Withopf 2005; Withopf and Riedmiller 2005) compared different RL methods on a grid-based soccer domain. In this domain, two agents must get and kick the ball to a goal in the field. They devised a Q-learning variant that can handle the semi-deterministic Markov Decision Process nature of the domain. Other researchers,viz. Yao, Chen, and Sun (2002b), and Tomoharu Nakashima and Ishibuchi (2003), also applied Q-learning in a small soccer problem (like positioning and kicking). Riedmiller and Gabel (2007) provides an overview of RL techniques used in the Brainstormer team during RoboCup competitions. Fard et al. (2007) used a game theory-based solution to tackle a coaching problem. Kalyanakrishnan, Liu, and Stone (2007) introduces an RL approach with communication for a multi-agent soccer attacking scenario.

2.5.1.2 Genetic Programming Approaches Genetic programming is one of the evolutionary methods that has been widely used in learning soccer behaviors. Hohn (1997) investigated the possibility of evolving functions which predict the success of a pass skill. In this study, a soccer game of two players, who pass the ball to each other, and one opponent, whose objective is to intercept the ball, was played a number of times. The behavior of each player was hand-coded, and the collection of all trials for different initial positions of the players of the game has recorded. Symbolic regression, a genetic programming technique, was used to approximate a function of sample points to determine the probability of a successful pass. The functions, which were evolved, were simple mathematical operators (such as +, -, and sin). Hohn generated a program that has 99.95% passing accuracy and uses a small number of symbolic functions.

Sean Luke (Luke 1998; Luke et al. 1997) used genetic programming to develop a program that plays soccer. Luke used high level functions to evolve player's behaviors. Some of these functions, such as pass decision, were developed using genetic programming as did Hohn (1997). Other functions, such as ball interception and blocking, were hand-coded, because evolving them was found to be too difficult (Luke et al. 1997). In Luke's program, evaluation is done by selecting two teams from the population to play a game of soccer against each other, with their fitness being assessed based on the goal difference (competitive fitness). In early generations, behaviors like kiddie-soccer (where all players chase after the ball and attempt to shoot it straightforward to the goal) were prevailing. But such behavior disappeared in later generations, yielding more organized playing behaviors. Some of the most successful programs have learned to chase and kick the ball as well as developed some primitive defensive abilities. Wilson (1998) also evolved a soccer team using genetic programming. His first attempt at evolving low-level functions (such as kick, turn, and if-then-else) was unsuccessful. Programs merely reached the point where they behave like kiddie-soccer. However, after adding some hand-coded high-level functions, programs were able to develop better behaviors.

De Klepper (1999) investigated the effects of adding high-level functions to evolve soccer behavior using genetic programming. These high-level functions include shooting towards the goal, passing to the closest teammate, and dashing to one half of the field. He found that programs evolved using genetic programming with high-level functions show a superior level of skill to those which use only low-level functions.

Andre and Teller (1999) used genetic programming to evolve soccer players. Their intention was to allow learning of intelligent behaviors from primitive functionality. Unlike other researchers, they used genetic programming with low-level functions called automatically defined functions (ADFs) (introduced by (Koza 1992)). These ADFs include running to position, kicking the ball, and scoring a goal. A team was developed to play soccer by evolving eleven players. As in (Luke et al. 1997), a competitive fitness function was used for evaluation. The evaluation was advanced in stages. First, agents acted in an empty field. Then they played against a simple hand-coded team, in which each player stayed in a spot and kicked the ball towards the opponent's half. After that, agents played against the 1997 RoboCup champion, AT Humboldt, and tried to score at least one goal. Finally, they played a three-game tournament with other teams from the population who had also made it past the previous stages. Andre and Teller's team, Darwin United, won one game and tied in another in the RoboCup 99 competition.

Gustafson and Hsu (2000, 2001) applied genetic programming in a mini-soccer game called keep-away. In keep-away soccer, three players try to pass the ball to each other, with the presence of an opponent whose job is to intercept the ball. The goal of the game is to maximize the number of passes and to minimize the number of intercepts. The authors' approach was to apply a layered learning paradigm to genetic programming for learning of a passing skill against a fixed opponent. In their experiments, two layers were used. The first layer was used for learning how to pass accurately. The second layer was used for learning how to minimize the number of ball interceptions. The first layer is learned by evolving ADFs for 40% of the maximum number of generations allowed. The remaining number of generations (60%) is then used to learn the second layer. In other words, the fitness function is changed from maximizing the number passes to minimizing the number of ball interceptions when 40% of the population life-span is reached. Overall, their results did not show a big difference between the layered approach and the standard genetic programming approach[1].

Evolutionary algorithms are usually used in robotic simulation domains due to that learning in real robots domain takes a very long time. However, Walker and Messom (2002, 2002) used genetic programming in evolving real robots to play soccer. They used a special simula-

[1] The fitness function used for standard genetic programming is to minimize the number of ball intercepts.

tion made for their robots to evolve two robots in a small arena. They used an island model[1] of distributed genetic programming, which is implemented using message passing interface (MPI). Their robots were able to move toward the ball and kick it.

2.5.1.3 Coevolutionary Approaches Salustowicz, Wiering, and Schmidhuber (1997, 1998) experimented with soccer simulation to study multi-agent learning. They compared several learning methods: temporal difference Q-learning with linear neural networks (TD-Q), probabilistic incremental program evolution (PIPE), and the coevolution version of PIPE (CO-PIPE). TD-Q selects actions according to linear neural networks trained with the delta rule to map player's inputs to evaluate alternative actions. PIPE is based on probability vector coding of programs used in genetic programming. PIPE synthesize programs that are guided by a probability distribution collected from programs' experience and that calculate action probabilities from input (Salustowicz, Wiering, and Schmidhuber 1998). CO-PIPE was found to outperform the other methods when playing against a base player. Moreover, CO-PIPE learning was found to be faster than the other methods.

Uchibe, Nakamura, and Asada (1999) investigate cooperative behavior in a small soccer game. This game involved three robots: two attackers, and a goal keeper. They used genetic programming with four functions: shoot towards the goal, pass to teammate, avoid collision, and search for the ball. The two attackers were first coevolved using genetic programming to score goals without the third robot. Some cooperation behavior was achieved as one robot passed the ball the another one who then shot the ball to the goal. In another experiment, a stationary robot was added. Although the position of the attackers was kept the same, the behaviors acquired were different because of the stationary robot. In a third experiment, they coevolved three robots. The attackers had to cooperate to compete with the goal keeper. Interesting behaviors were attained. For example, the goal keeper, whose purpose is to intercept the ball, was able to shoot the ball to the attackers' goal. This suggests that the task of the goal keeper was easier than the attackers' task.

Coelho and D. Weingaertner (2001) tackled a team formation problem using a cooperative

[1] The island model is based on the idea that isolated subpopulations with occasional migration will maintain more diversity and reach higher fitness values than single interbreeding populations.

coevolutionary genetic algorithm. The task to be learned was to find the best formation of a team of five robots. The formation was represented as the region in which each player was allowed to act in. A hand-coded algorithm was used for all the players, including the opponents. In evaluation, the team of the coevolved five players played a number of games against a hand-coded base player whose formation was fixed. The best team that is learned by coevolution was able to beat the base team by four goals on the average. This results showed a successful application of coevolution in a soccer formation problem.

Matkovic (2002) explored competitive coevolutionary behavior in an ascii soccer game, in which a kicker and a goalie played in a simulated soccer environment. The simulation was implemented by an ascii screen, where each object was presented as a character. Using genetic programming with primitive functions, Matkovic (2002) compared a single population method with a coevolutionary one. In the single population, the kicker and the goalie learned separately, whereas in coevolution both coevolved competitively. The results showed a superiority of the coevolutionary method over the single population for both the kicker and the goalie.

Whiteson et al. (2003, 2005) investigated the keep-away problem with layered learning and coevolutionary methods using neuro-evolution. The SANE system (Moriarty and Miikkulainen 1996a)was used to enable different neural networks to learn the keep-away task. Moriarty and Miikkulainen (1996a) compared many learning methods. First was tabula-rasa learning, in which learning was done by creating a monolithic network that attempts to learn a direct mapping from sensors to actions. Second, they used learning with task decomposition; the task is decomposed into five different sub-tasks, each of which has its own neural network. These tasks were: intercept, pass, pass evaluation, and get open. Then a fixed decision tree was applied which used these sub-tasks to perform the keep-away task. Third was layered learning, in which each of the sub-tasks was learned before the other. Fourth, they used coevolution, in which each sub-task was coevolved with the others simultaneously. It was necessary for these sub-tasks to cooperate to achieve their goal. The results obtained showed an advantage of the coevolutionary approach against the other methods.

The keep-away problem has been studied more by many researchers in 2000s. Kalyanakrishnan, Stone, and Liu (2008) and Kalyanakrishnan and Stone (2007) applied RL to tackle the problem. A comparison between RL methods (based on the previous cited work) and EA methods was made by Taylor, Whiteson, and Stone (2007) and Taylor, Whiteson, and Stone (2006). They found that Evolutionary methods have advantageous performance over RL methods, but they are slower in learning. Whiteson, Taylor, and Stone (2007) provides a broader analysis on the same comparison on more benchmark tests, and proposes a way to make both methods (i.e. RL and EA) work together in solving the keep-away problem.

Østergaard and Lund (2002, 2003) explored the coevolutionary approach and tested whether it could be used to evolve soccer behaviors for the Khapera robot soccer task. In a Khapera robot soccer game, two Khapera robots are pitted against each other in an arena. The game ends when a goal is scored or four minutes has elapsed. An evolutionary and a coevolutionary approach in which the two robots competitively coevolved together were compared. The results concluded that coevolution is more robust with respect to handling different opponent strategies than the one evolved with a single population although its learning time was greater.

2.5.2 Discussion

RoboCup soccer has shown itself to be an interesting test-bed for many reinforcement learning and evolutionary techniques. Layered learning has been one of the most successful methods used to enable robots to learn to play soccer; however, hand-coded players still have had a dominant winning rate in all the recent RoboCup competitions. Evolutionary techniques when applied to RoboCup soccer have showed some successful application, e.g. as in (Wilson 1998) and, to a lesser extent, in (Luke et al. 1997). Both Wilson (1998) and Luke et al. (1997) used the genetic programming paradigm. Genetic programming is also used by many other researchers in evolving soccer behavior as well, probably because it is easy to incorporate hand-coded high-level functions into the algorithm.

Many researchers have utilized coevolution for RoboCup soccer. Some of these researchers' work showed advantageous behaviors of robots when using coevolution over using other methods in small soccer games. Scaling up coevolution to a full soccer game will be interesting; nevertheless, it will not be a simple task. Whether coevolution can scale up to learn such complex tasks and how it can be accomplished is the subject of this dissertation.

3.0 INCREMENTAL LEARNING THROUGH EVOLUTION

Learning a complex task in a noisy and unpredictable environment is not a simple task. The state space for such a task is so large that it is difficult for any learning algorithm to successfully learn it, at least at today's level of computing technology. Evolutionary algorithms have a potential learning capability in complex tasks since they allow learning in an unpredictable and changing environment. To demonstrate this, evolutionary algorithms will be applied to a complex task with the forementioned properties, namely the soccer game. Soccer has a huge state space that makes it very complex to learn directly from perceptions to actions, even for a human. In this research, three key elements are considered necessary for learning success. First, the task must be divided into many smaller tasks that can be learned via evolutionary algorithms. Second, the learning must be incremental, that is, the difficulty of the task should gradually increase as evolution progresses. Third, the evolutionary algorithms need to be enhanced by coevolution. In this study, the learning experience will be divided into levels. At each level, some tasks must be learned before moving to the tasks at the next level. Moreover, the tasks themselves at each level will be learned incrementally, i.e. the task's difficulty increases as the population evolves. Both types of coevolution, cooperative and competitive, will be used to assess the evolutionary process. The author believes that the coexistence of both cooperative and competitive populations during evolution will greatly aid agents learning to play soccer[1].

To assess this claim, the SAMUEL system will be used to evolve robots to play soccer. These robots are software robots, or softbots, and will learn in a simulated environment of soccer. The RoboCup Soccer Simulation Server will be used as the environment in which the SAMUEL softbots will be evaluated. An overview of SAMUEL is given below, followed

[1] That is how the evolution process occurs in real life.

by a description of the soccer server. The last part of this chapter will explain the learning strategy that will be used and my experimental design.

3.1 SAMUEL

SAMUEL, which stands for Strategy Acquisition Method Using Evolutionary Learning, is a machine learning system that uses evolutionary algorithms to solve reinforcement learning problems. It explores decision policies in simulations and modifies those policies based on acts in the simulated environment. Policies are represented by if-condition-then-action rules. Consequently, it is most appropriate for sequential decision problems such as controlling robot behaviors. SAMUEL supports parallel execution and coevolution and includes Java-based visualizing tools, making it a powerful tool for experimenting with robot behaviors and evolutionary algorithms. A brief description of SAMUEL is given below. A complete manual may be found in (Grefenstette 1997).

3.1.1 Learning in SAMUEL

SAMUEL learns policies acting in a simulated environment by incorporating genetic algorithms and other rule-based learning methods (Lamarckian evolution[1]). Each rule has a strength value associated with it. These values are updated and used in the Lamarckian learning process. The user decides the environment, provides the sensory information to its agents, and applies the actions of those agents in the environment as well as the fitness after a number of trials, called episodes. The user also has to provide basic policies as an initial population.

The learning process begins by applying genetic operators to the initial population, which consists of the basic rules provided by the user. Then the population is evaluated before applying Lamarckian methods. The updated set of policies is evaluated again and the learning process is repeated until the user-defined end criterion is met. Figure 3.1 illustrates

[1] Lamarckian operators are different from Darwinian operators in the sense that the environment directly affects the organism's behavior rather than affecting only the selection course.

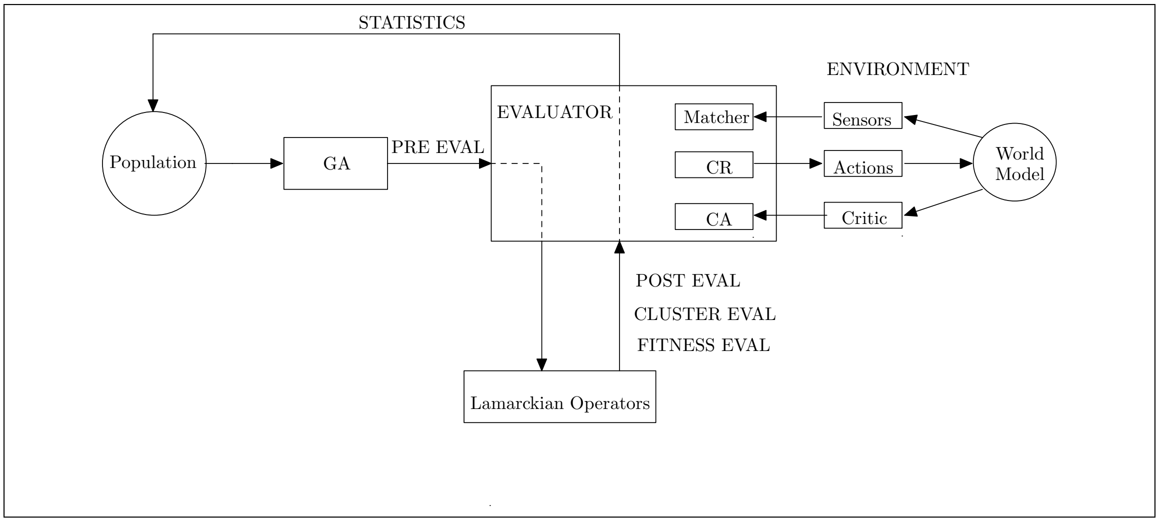


Figure 3.1: SAMUEL system architecture

this process. The genetic and Lamarckian learning methods are described below, followed by a description of the SAMUEL evaluation process.

3.1.1.1 Genetic Operators The initial population consists of sets of rules, called rulebases, where each set is a set of if-condition-then-action rules. At each generation, the genetic algorithm evaluates the population and determines its rulebases' fitness. Then it selects high performing rulebases from the population and applies genetic operators to it. The operators used are: mutation, creep, and crossover. The cycle of generation is repeated until a user-defined criterion is fulfilled.

Mutation

Mutation constructs new rules by making random changes to the existing rules. It only changes a single value in any condition or action. Figure 3.2.a illustrates how mutation works. This operator can change a single value condition to any arbitrary one.

Creep

Mutation is a disruptive operator; assume we have a high performance rule. When applying mutation to it, its fitness might be lowered drastically. Therefore, a less disruptive mutation

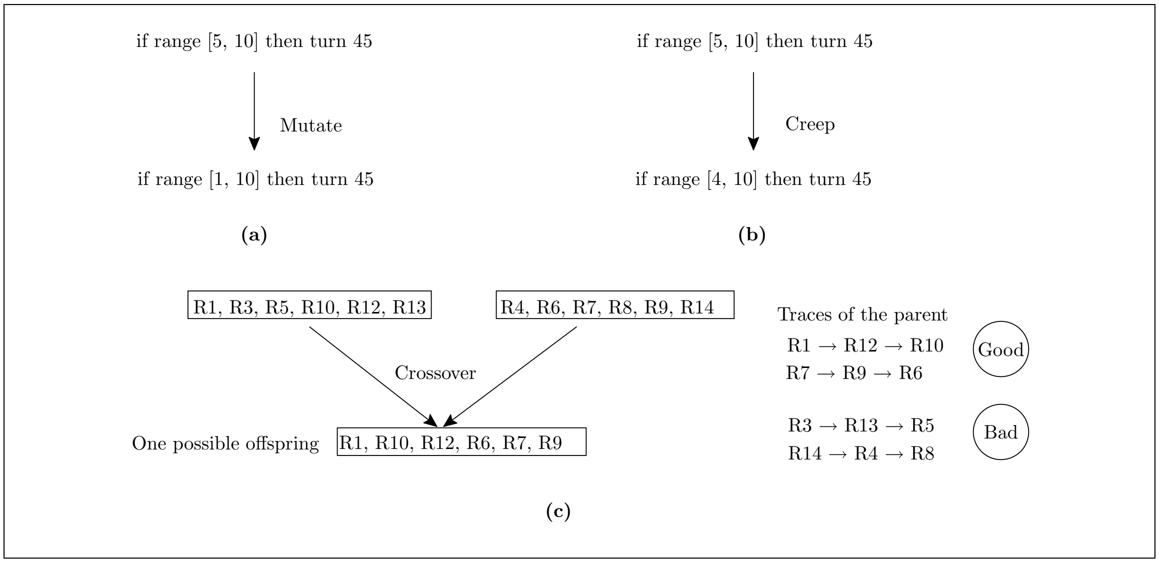


Figure 3.2: Genetic Algorithms operators in SAMUEL

is important to make slight changes to the rules. Creep is a restricted form of mutation that changes the values of a condition or an action to the nearest value. For example, in Figure 3.2.b, creep may change the condition "range [5, 10]" to "range [4, 10]".[1]The value 5 is chosen randomly and changed to the nearest value 4. This can be seen as a generalization, in that the range becomes larger than before. Creep might change the value 5 to 6, i.e. to specialize, instead. The user can control the trend of creep to generalize or to specialize.

Crossover

SAMUEL uses crossover to create plausible new rulebases. A pair of rulebases is randomly chosen and rules are uniformly exchanged between them to form two new rulebases. SAMUEL also uses a restricted form of crossover called clustered crossover. The rules of the rulebases are first evaluated, and those which are fired during a successful episode are clustered together. Then the clustered rules are assigned to one of the offspring. This way we allow the good sequence of rules to be passed to the next generation, along with the high-quality rules. Figure 3.2.c illustrates how crossover generates one possible offspring.

[1] Assuming granularity is 1.

3.1.1.2 Lamarckian Operators SAMUEL utilizes Lamarckian evolution to add new rules to its population. The Lamarckian operators used are: specialization, generalization, rule covering, avoidance, merging and deletion. The population of rulebases is evaluated first, and then its performance triggers these operators.

Specialization

A specialization operator creates new rules that have a smaller range in their conditions or actions. Figure 3.3.a shows how this type of operator works. During evaluation, SAMUEL keeps track of low-strength general rules in a number of episodes and uses those rules as candidates for specialization. The values in the range to be changed will be decreased to be around the actual sensor reading during the evaluated episodes. The added new rules then will compete with the more general one, and the poorer rule will be deleted in a subsequent generation.

Generalization

A generalization operator creates new rules that are more general than high-strength rules during evaluation. The range of a rule in a condition is changed to cover more values around the actual sensor reading, as seen in Figure 3.3.b. This type of operator will add new high-strength rules that can be triggered more often, since they are more general.

Covering

In some episodes the sensor readings might not match with any of the existing rules. At this point, matching is applied to the closest rule. This is called partial matching. The covering operator will create a new rule from the partially matched rules. The new rules will have to be more general to cover the actual sensor readings. Figure 3.3.c illustrates this.

Avoidance

An avoidance operator creates new rules from low-strength rules that have arbitrary different action values; i.e. only the action's value is changed. This can be seen as a form of mutation, but it is triggered by low-payoff rules instead of rules chosen at random.

Merging

A merging operator creates a new rule from two existing rules having identical actions. The user can specify whether to consider two rules that have equivalent strengths or not. This is illustrated in Figure 3.3.d. Note that it is also possible to have more than one condition merged together to form a new rule.

Deletion

Rules can be deleted explicitly rather than not being selected for reproduction. This operator allows the population to get rid of useless rules[1]. Rule deletion is triggered if the number of rules in the current rulebase reaches a certain point, which is defined by the user. A rule cannot be deleted if it is too young or if it is fixed[2]. A rule is considered for deletion if all its actions meet the following criteria:

1. The rule is idle, i.e. it has not been fired for a long time (α).
2. The rule has a strength less than the strength threshold (β).
3. The rule is subsumed by a more general rule that has higher strength than its strength by a factor equal to the subsume threshold (δ).
4. The rule is chosen randomly if the delete threshold (γ) is less than 1.

Note that α, β, δ, and γ are user-defined parameters. Deletion is a powerful operator that allows the user to control the ruleset size; however, the threshold parameter needs to be set carefully to avoid deleting potentially useful rules.

3.1.1.3 Evaluation The population is evaluated four times in SAMUEL. The first evaluation, which is called PRE EVAL, is used for gathering behavioral traces for the Lamarckian operators. The population is evaluated again, POST EVAL, to assess and adjust rule strengths after applying the Lamarckian operators. After POST EVAL, the population is evaluated to cluster rules to be used in crossover. In this phase the rule strengths are not adjusted, since the aim is only to gather the agent experience to cluster rules. The final eval-

[1] Deletion is done after crossover and before mutation.

[2] A user-defined parameter, incubation period, specifies the number of generations a rule must survive before being considered for deletion.

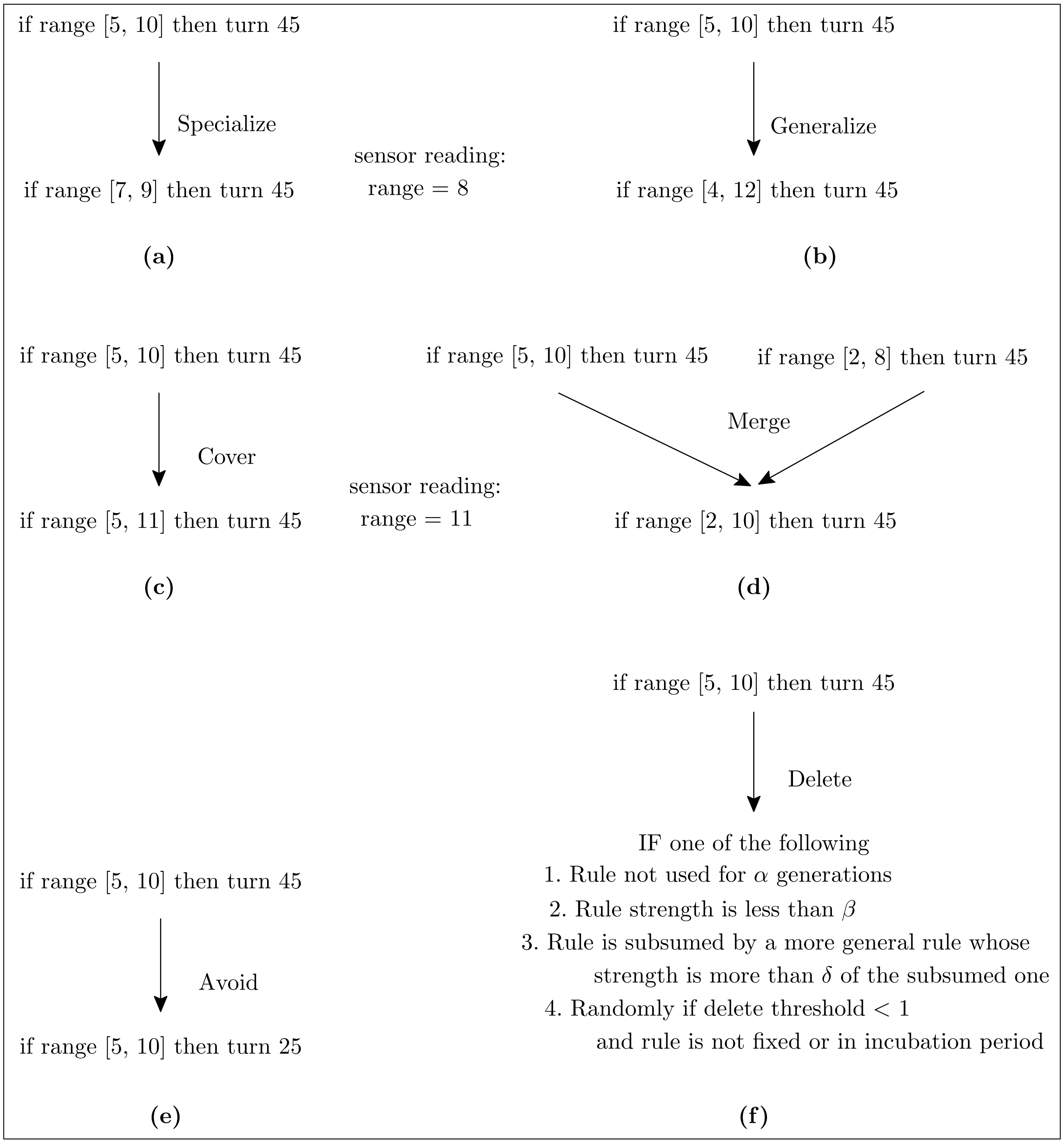

if range [5, 10] then turn 45
Specialize
if range [7, 9] then turn 45
(a)
sensor reading:
range = 8
if range [5, 10] then turn 45
Generalize
if range [4, 12] then turn 45
(b)
if range [5, 10] then turn 45
Cover
if range [5, 11] then turn 45
(c)
sensor reading:
range = 11
if range [5, 10] then turn 45
if range [2, 8] then turn 45
Merge
if range [2, 10] then turn 45
(d)
if range [5, 10] then turn 45
Avoid
if range [5, 10] then turn 25
(e)
if range [5, 10] then turn 45
Delete
IF one of the following
1. Rule not used for α generations
2. Rule strength is less than β
3. Rule is subsumed by a more general rule whose
strength is more than δ of the subsumed one
4. Randomly if delete threshold < 1
and rule is not fixed or in incubation period
(f)

Figure 3.3: Lamrckian operators in SAMUEL

uation, FITNESS EVAL, is used to obtain the final fitness of each rulebase in the population, upon which selection for the next generation is based.

The evaluation model is based on a Competition-based Production System (CPS) (Grefenstette 1997). It consists of three parts: matching, conflict resolution, and credit assignment. These three modules interact with the simulation environment to evaluate the whole rulebase and to adjust its rules' strengths. Figure 3.1 illustrates this interaction. Below is a brief description of each of the three modules.

Matching

When the sensor readings are ready, the matcher tries to find the set of rules in the rulebase that matches the sensor readings. The matched rules, also called the match set, are passed to the conflict resolution module. If no rule matches the current sensor readings, then a partial match takes place in which the closest rules to the sensor readings are placed in the match set.

Conflict resolution

After receiving the sensory information from the environment, only one action can be executed. However, the match set may contain more than one rule. In this case, the rule that has the highest strength is executed. If more than one rule has the same strength, then one is chosen randomly.

Credit assignment

Each cycle of sensor readings and action execution is repeated until an episode is complete. The environment provides a payoff (critic) which determines the agent's evaluation in the completed episode. Each rule's strength is updated according to a profit-sharing-plan. In this way, all the rules used in an episode will share the payoff, and their strength will be changed accordingly. If the payoff is higher than the strength of the rule, then the rule strength is increased; otherwise, it is decreased. The increment (or decrement) value added to the rules' strength is controlled by a learning-rate parameter specified by the user. The evaluation is repeated for a number of episodes, and the average payoff is obtained for each rulebase in the population.

3.1.1.4 Coevolution Modes SAMUEL allows the user to evolve more than one population in various modes. Populations may be evolved in a different time-frame, asynchronously, or at the same time, synchronously. SAMUEL also supports a round-robin mode.

Asynchronous mode

In this mode, each agent has its own population that runs separately. Agents are paired for evaluations even though they might be at different generations. For example, an agent at generation 12 might be evaluated with another agent from another population at generation 30.

Synchronous mode

Evaluation in this mode is done synchronously. That is to say, an agent at generation 12 will be evaluated with another agent from another population also at generation 12.

Round-robin mode

In this mode, only one population will be evolving, while the others will remain static for a given number of generations. Each population will become active when it gets a token for a number of generations and will then pass the token to another population.

In coevolution, evaluation between individuals in different populations can be done in many ways. SAMUEL supports four different policies: current, best-so-far, random-best, and latest-best. To illustrate this, assume that two populations, A and B, evolve together. Individuals in A will be evaluated with i) a random member selected from B's current rulebases, ii) a set of B's best-so-far rulebases (among all generations), iii) a random member of B's best rulebases, or iv) B's latest best rulebases.

3.2 ROBOCUP SOCCER SIMULATION SERVER

First introduced in 1993 by Itsaka Noda at NTL lab, Japan, the RoboCup Soccer Simulation Server is a simulation system which enables two teams of autonomous softbots to play a match of soccer in a two-dimensional space. The simulator has been used in several international competitions since 1996 and undergone many modifications since then. The current version is Soccer Server 11, which the following description is based on[1].

The simulation consists of three parts: a server, a monitor, and a logplayer. The server is the main part of the simulator and is used to control all aspects of a soccer game. The monitor is a visual aid that allows a user to watch a soccer game being played in real time. The monitor is also used to start the game. The logplayer is a utility to preview previously played games.

The server uses client-server architecture to simulate a game of soccer. The server sends its sensory information and receives players' actions through a UDP port. In order for the players, softbots, to play in the simulation, they must connect to the server first. After the game is started, the server will send its sensory information to all connected players periodically (every 150ms). Each player receives different information based on its position in the field and its view direction. Players also send their actions to the server periodically (every 100ms). Actions sent in between simulation cycles might not be executed, depending on the action type and the number of actions sent in the same interval. Therefore, the server is a discrete-event driven model and its sensor-action cycle is asynchronous. This makes the server a very complex domain as the players must act asynchronously in real time[2].

3.2.1 Sensors

The server sends three types of sensory information to the players: visual, aural, and body. Below is a brief description of these sensors.

[1] For a more detailed description the reader may refer to (Foroughi et al. 2002) or visit the official RoboCup Soccer web site at http://www.robocup.org.

[2] Each player needs to send actions whenever possible; therefore, in some cycles, a player must do so without waiting for sensory information from the server.

3.2.1.1 Visual Sensor Each player receives periodically (a 150ms) visual information about the field from the server allowing it to see objects that are in front of it. The server sends the relative distance and angle of the player to the objects that the player can see. Some Gaussian noise is added to the information returned by the server; this makes the environment more realistic and more challenging. The further the object is from the player, the noisier the information sent and the harder it is to recognize. For example, a player can read his teammate's jersey number if the teammate is within a few yards of him, but may not be able even to recognize whether the other player is a teammate or an opponent if the distance is too far. Since the player can only get information about what he sees relatively, he cannot sense his location in the field. However, the player can see many flags (52 flags) and lines spread around the field, and from this, he can gain an idea where his location is in the field.

3.2.1.2 Aural Sensor Occasionally, players receive aural messages sent by other players, a coach or a referee. The aural sensor is useful for realizing some important event has happened on the field, for example, the ball going out of bounds. It is also valuable for communicating between players, although it is very noisy and has a low-bandwidth channel.

3.2.1.3 Body Sensor Along with the visual and aural sensors, the server sends body sensory information to the player. This information includes the physical status of the player, for example, the player's current speed or the count of a particular action.

3.2.2 Actions

To act in the simulation game, players send their actions to the server as commands although it is not guaranteed these commands will be executed in the order sent, since they are sent via a UDP port. Following is a brief description of each of the currently available commands for the players.

3.2.2.1 Kick The kick command allows players to kick the ball. It takes two arguments: the power of the kick, and the angle towards which the ball is kicked relative to the body of the kicker. A player can kick the ball only if it is near enough to the player. The actual direction of the ball after issuing a kick command cannot be determined exactly because it depends on the position of the player relative to the ball's position before being kicked. Noise is also added to the angle and the power.

3.2.2.2 Dash The dash command can be used to accelerate the player in the direction his body is facing. It takes one argument: the power of the dash. The more power provided, the faster the player goes. It is important to note that issuing a dash command moves the player to a new position in the next cycle only; therefore, successive dash commands must be issued to keep a player moving. Each player has a stamina value set at the beginning. The more the player dashes, the more he consumes his stamina. That means a player cannot keep dashing all the time at full power. His actual dash power will degrade until all of his stamina is consumed and he eventually stops.

3.2.2.3 Turn A player can change his body direction by sending a turn command to the server. The turn takes the desired angle as an argument. After executing this command, the player's direction will be changed; however, this may not always be to the exact direction the player wants because it depends on the speed of the player, the desired angle, and the random noise that is added.

3.2.2.4 Turn Neck Players can turn their neck relative to their body. The changing of the neck direction allows players to change their view but not their body direction. This command is useful to allow players to see what is on their left or right while running in a straight line.

3.2.2.5 Tackle This is a fairly new command added in version 8. This command gives a player the ability to tackle a ball. Tackle takes one argument: the power of the tackle. Depending on the power of the tackle and the ball distance, a player can get possession of the ball, i.e., the ball placed in front of the player within a kickable distance. The execution of tackle is probabilistic; thus, not all tackle commands are successful.

3.2.2.6 Catch At the beginning of the game, each team may designate a player to be a goalie. The goalie can catch the ball only in the goalie area; otherwise, the catch is considered a foul. The catch command can be issued only by the goalie and it takes one argument: the direction of the catch. The goalie can catch the ball if it is near enough to be caught. If the goalie issues a catch command and it is unsuccessful then he is banned from issuing it again for a few cycles. This is practical as in this way, attackers can score goals.

The soccer server provides other commands. For example, "change view" allows players to alter their view, giving a choice of a narrow viewing angle or a wide viewing angle. It also gives the players the ability to view with higher quality but with delayed perception. Players can send messages to their teammates using the "say" command. These messages are broadcast so that even the opponents can hear them; however, it provides players with the ability to communicate. Players can ask for the current score and their body status by using "score" and "body sense" commands, respectively. The server also allows players to move to a certain position instead of dashing using the "move" command. A move is not allowed during the game, but it is useful for setting up the players' initial positions before the game starts. The goalie may also move in his area when he catches the ball. Table 3.1 provides a summary of the actions provided by the server.

3.2.3 Game Control

There are several play modes used to inform the players of the status of the game, for example, when the ball is out of bounds or an offside foul has occurred. A list of all play modes used in the server is given in Table 3.2. The soccer server contains an automated referee which controls the game. The referee changes the play mode of the game depending on the game situation. Whenever the play mode changes, the automated referee announces this by sending a message to all the players.

The soccer server supports heterogeneous players. Each team can choose from several different types of players with different characteristics. The different player types have different abilities. For example, some players will be faster than others, but they will have

Table 3.1: List of soccer server commands

Command	Format	Arguments
kick	(kick double double)	power, direction
dash	(dash double)	power
turn	(turn double)	direction
turn neck	(turn_neck double)	direction
tackle	(tackle double)	power
catch	(catch double)	direction
change view	(change_view str str)	width, quality
say	(say str)	message
move	(move double double)	x, y position
sense body	(sense_body)	
score	(score)	

less accurate kicks.

It is possible to define a coach agent that receives noise-free global information about all the objects on the soccer field. The coach can give advice to his players by sending messages to them. He is also responsible for selecting heterogeneous player types and substituting players during the game and can be used to automatically create training sessions.

3.3 LEARNING SOCCER BEHAVIORS

Soccer is all about winning. In order to win, one team has to score more goals than the other. A team can only score a goal if it has possession of the ball. If it loses the ball it tries to prevent the other team from scoring by defending against an attack. These basic aspects are fundamental to understanding the game of soccer; however, achieving its objective, winning, requires a great effort. To achieve proficient playing, three elements must be satisfied. The first element is possession of the basic skills necessary to play the game such as passing and shooting. The second element is skill in making decisions about what actions are appropriate

Table 3.2: List of soccer server play modes

Play modes	Description
before kick off	at beginning of a half
time over	after the game
play on	during normal play
kick off left/right	start of a half
kick in left/right	ball out of bound from side pitch
free kick left/right	after a foul
corner kick left/right	ball out of bound from goal pitch
goal kick left/right	goalie catch the ball
after goal left/right	goal scored
drop ball	ball is not put into play
offside left/right	announcing offside foul
penalty kicks	withdraw breaking after the two halves

or inappropriate in a given situation. Achieving this element is a question of experience and intelligence. The third element is effective cooperation. Soccer is played with eleven players; therefore, winning can't be accomplished without cooperation.

Learning how to play soccer is a difficult task. It takes years to understand the basic elements of the game. Consequently, learning must be done in stages. If you compare a kindergarten soccer game with a professional game, you notice a big difference in the three elements of soccer. In kindergarten soccer, most of the players are chasing the ball, merely trying to kick it in the opponent's goal. In professional soccer, on the other hand, cooperation between players is apparent.

There are quite a few similarities between human learning and machine learning in the context of soccer. In both cases, the learning process must go through a number of phases, and it is incremental in nature. Also, learning is achieved by playing soccer and assessing it with a critic so that players know if they are playing well or not. In the case of human learning, a coach gives guidance through training sessions and verbal comments. In the case of machine learning, it is the programmer's task to design training sessions and assign a critic for each of session.

The following subsections provide an overview of the learning phases. Each of these levels teaches players skills in addition to the basic three elements discussed above. For each task, a training session is designed to enable robots to learn using evolutionary algorithms. Some of these sessions may be learned using one population, others may use more than one population.

3.3.1 Low-level skills

At this level, players learn basic soccer skills. These skills include moving to a desired position in the field, running after the ball, kicking the ball, running with the ball, and catching the ball. These skills are analogous to initial skills that children on a soccer team learn. Tasks learned at this level require no interaction between players; players only deal with the ball and learn how to control it.

3.3.1.1 Dash to position This skill enables an agent to move to any point in the field. The point is static, and the agent must get to it as soon as possible. The ideal behavior of the agent is to turn to the point and dash straight towards it.

3.3.1.2 Chase the ball Since the ball is the center of the game, players are watchful to get possession of it. This skill can be learned by letting the ball move in any direction in the field and requiring the agent to try to get it. The distance and the speed of the ball control the difficulty of this task.

3.3.1.3 Kick the ball to a position Kicking accuracy is important for scoring goals as well as passing the ball to teammates. An agent has to kick the ball to a given point in the field. The ball may be placed close to the agent so he can kick it; however, the point might be set at a random place far from him, making the task harder. If the point is put behind the agent, the task is even more difficult. If the point is a teammate, then this skill becomes a passing skill; furthermore, if it is on the goal line, then this skill turns into a shooting skill.

3.3.1.4 Dribble The most essential skill that junior soccer players must learn first is how to control the ball. This skill enables the agent to move while still having control over the ball. The agent's task is to move from one point to another while kicking the ball near to him at all times.

3.3.1.5 Catch a ball Catching the ball is important for the goalie agent as it prevents the team being scored upon. This skill enables the goalie to learn how to catch a moving ball. A ball might pass near to him at different speeds. The agent must decide when and where to catch the ball.

3.3.2 High-level skills

Players at this level learn how to scale up their basic skills to achieve proficient playing. Unlike those in previous levels, tasks in this level involve interactions between players, as each player competes with other teammates to perform its task. For example, an attacker learns how to shoot the ball in the presence of a goalkeeper, who learns how to prevent goals. The learning process at this level can be designed as mini-games involving two to three players. Each player then tries to learn its own task.

3.3.2.1 Block a pass This skill enables a player to block the path of an opponent's pass. The obvious way to achieve this is to stay in a point of a line between the passer and the receiver; however, when the passer has more than one potential receiver, the task becomes more difficult. This skill will use the dash to position skill learned from the previous level to achieve its goal.

3.3.2.2 Intercept the ball This task enables a player to intercept the ball. This is similar to the chasing the ball skill in the previous level, but with the difference of an opponent's presence. For example if the opponent has the ball, the player may want to tackle him to get the ball, instead of chasing the opponent and the ball while the opponent is dribbling.

3.3.2.3 Mark an opponent One of the defense skills soccer players must learn is to mark an opponent, that is, to guard him one-on-one to stop him from being useful to the opposing team. Being close to an opponent makes it easy to act quickly, preventing the opponent from receiving the ball or kicking it.

3.3.2.4 Maneuver with the ball The dribble skill in the previous level allows a player to move with the ball in a straight line. The maneuver skill enables the player to move with the ball along any path in the presence of opponents. This skill allows the player to avoid opponents in his path to get to his targeted position.

3.3.2.5 Pass This skill enables a player to pass the ball to his teammate. Passing is important in soccer because it allows players to move the ball from one position to another quickly. The pass may be a direct pass, where the player kicks the ball to the exact location of his teammate, or an indirect pass, where the player kicks the ball to a location where his teammate can get it.

3.3.2.6 Get open for a pass This skill enables a player to move to a position where he can receive a pass from his teammate. The position should be far enough from any opponents, so that the pass won't get intercepted or blocked and close enough to the passer for it to be successful. The position must also be as close as possible to the opponent's goal.

3.3.2.7 Move to a strategic position This skill enables players to find their strategic positions if they are not in a useful position. When the ball is far from a player, it is important that the player move to a location that has potential for usefulness on the field. Strategic positions may differ for each player, depending on the player's role.

3.3.2.8 Shoot to goal This skill enables a player to score goals. The player learns to kick the ball into the goal with the presence of a goalie. While the goalie tries to catch the ball, the shooter tries to make his kick hard enough for the goalie not to catch it.

3.3.2.9 Goaltending This skill enables a goalie to prevent opponents from scoring. The goalie's main task is to catch the ball. However, moving to a position where he can catch it is vital. The goalie must learn how to place himself in a position so that opponents find it hard to score a goal.

3.3.3 Pre-decision-level skills

Some skills learned in the previous levels need some special refinements. For example, a player executing a pass skill must decide to whom to pass it. This executive decision is important to improve the passing skill. At this level, players learn how to execute a certain task to improve their overall performance.

3.3.3.1 Executive Pass For this skill, players learn to decide to which teammate the pass should go. A mini-game of three to four players may be played with opponents to practice pass selection. The obvious rule of thumb is to pass to the teammate who is most open, i.e., someone where no opponents can block the pass. However, evaluating who is the most open teammate is not easy to determine in all situations.

3.3.4 Strategy-level skills

This is the highest level in the learning process. Players learn how to use the previously learned skills to achieve their goal, i.e. playing a good soccer game. At this level players learn when to use their skills, that is, when to execute a particular skill. Decision making is important for playing soccer, but learning such a task is very difficult. Professional soccer players may learn most of their soccer skills before reaching the age of twelve; however, learning how to make these decisions takes several more years. What makes this task hard to learn is that its state space is so big that it is impossible even for a human to analyze. Furthermore, it is not obvious which choice should be made for every situation in the game. The best way to learn this task is to actually play the game and decide which actions give favorable results. Therefore, at this level, players will learn by playing full soccer games.

There are many other things that must be taken into consideration to make a decision. Players must cooperate to achieve their goal; moreover, each one of them must have a specific role. For example, some players' task is defending their goal, and others' task is to score goals. In a real soccer game, players are grouped into three roles. Besides a goalie, there are defenders, midfielders, and attackers. Each team can assign a different number of players to each group. This is usually referred to as team formation[1]. Which formation a team decides to use can be thought of as the strategy of the team. This also needs to be learned.

[1] Formation is typically described as three numbers. For example a formation 3-4-3 denotes 3 defenders, 4 midfielders, and 3 attackers.

3.4 EXPERIMENTAL DESIGN

As mentioned in the previous chapters, the SAMUEL system is used to enable learning for a team to compete in RoboCup soccer. The soccer server is used as the environment in SAMUEL. The version of SAMUEL that is used was written in Java, while the soccer server was written in C/C++. In order to make the link between the two codes, Java native interface (JNI) has been utilized. The soccer server uses UDP ports as a method of communication between the players and the server; consequently, it took a long time to execute one step inside the simulator (at least a millisecond). Moreover, the soccer server uses a time signal-interrupt to provide a real-time environment. To speed up the evaluation process, Java code was implanted inside the soccer server to override the slow UDP and timing codes.

It is important to visualize the behavior of the agents with a graphical interface. Therefore, a Java-based demonstration tool is implemented. The GUI was very helpful in explaining the agents' behaviors as the learning progressed.

The soccer simulation server will be used along with SAMUEL to enable learning of soccer skills at all levels. Each skill will have its own population and will be evolved to achieve a desired performance. To achieve our goal, the learning will be divided into three phases:

Incremental evolution

In this phase, skills at the low-skills level will be learned separately through evolution. A fitness function for each skill will be designed to drive an initial population to perform an acceptable behavior. Such fitness functions have an incremental nature. That is to say, at earlier generations a simple (easy) fitness will be assigned while in later generations the fitness will become harder. Another way of implementing incremental learning is to increment the difficulty of the environment around the learning agent. For example, a shooting task can be learned incrementally by omitting a goalie from the environment at early stages and then assigning a goalie with increasing competency as the learning stages progress. Note that the shooter fitness is not changed through the stages, only the environment, yet incremental learning is still attained.

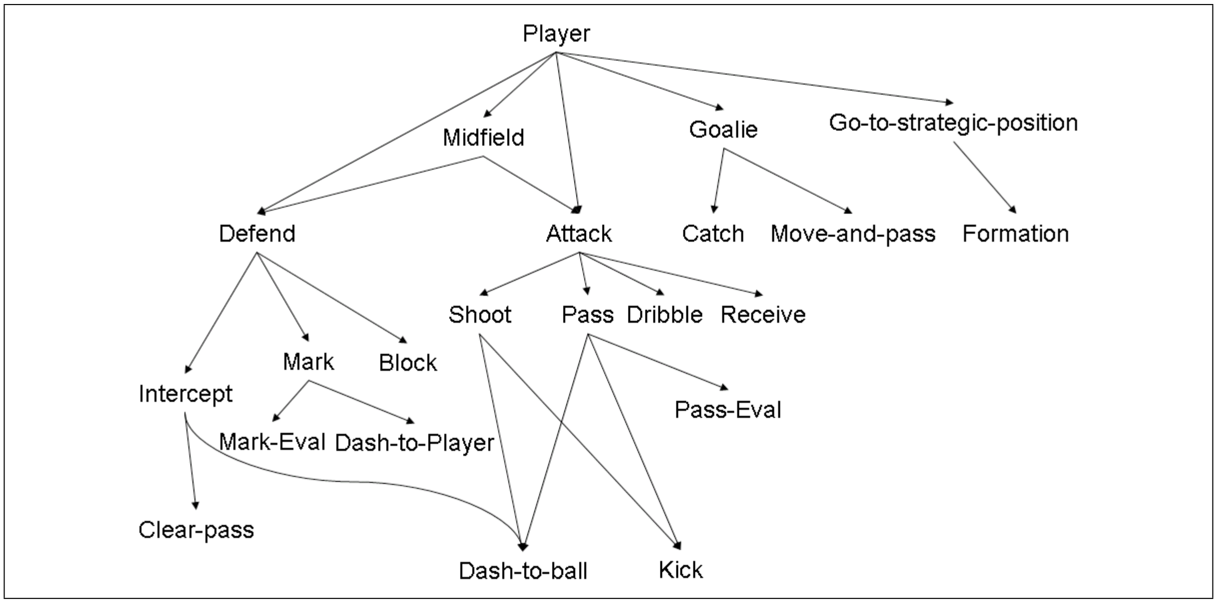


Figure 3.4: An example of a soccer task decomposition

Hierarchical coevolution

Skills in the high-levels will be learned though competitive coevolution. Learning experiments can be seen as mini-game training sessions, which will consist of two to three players. Each skill starts with an initial population and competes with another skill's population. For example a passer may be evolved with a pass blocker. In some cases, a competitive-cooperative coevolution will be used, such as when learning a passer and a receiver against a pass blocker and a marker.

At the top level, populations will be evaluated based on the final score of a full game in which the opponent could be a fixed team (or set of teams) such as a previous RoboCup champion. Figure 3.4 shows an example of task hierarchy of a possible soccer learning strategy. The arrows indicate the dependency between tasks. Note that, in some levels of the tree, all the tasks maybe executed, while in others, only one is executed.

The soccer game involves 11 players in the field, where each may have his own role in the game. Learning all of them at the same is a tedious task.The reader may have noticed that, we have not mentioned how these players are going to cooperate in achieving the learning goal. In our design, the learning is divided into tasks, and all players need those tasks to perform in the game. Beside the goalie, soccer has three player types: defenders, midfielders, and attackers. Each of the defenders and attackers have their own skills to learn. The midfielders must be familiar with the set of skills for both defenders and attackers skills as they have to perform both roles in the soccer game. They must know when to use either role (attack or defend) during the game as well. In real soccer, each player may play a different role within the group. For example, a left wing attacker has a different role than the middle attacker. However, learning each role adds more complexity (and time) to the overall learning process. For this research, players in each group will be considered homogeneous. The number of players in each type is determined by the formation strategy pursued. Each type can be learned in a mini-game setup, where more than one player is involved in the game. Players with the same type will share the same learned policy during the game. Thus, all players have one rulebase or “policy” but may collect different experiences or “traces”. This approach makes the learning easier when there are fewer learners and resolves the problem of credit assignment that may come up.

This design can be seen as a concurrent multi-agent learning technique, in which multiple learning processes attempt to improve parts of a complex problem at the same time. However, agents here are learned in ”squads” instead of each agent having its own learner. According to (Panait and Luke 2005), this approach has not yet been seen in any concurrent learning literature.

Orchestration

When dealing with a complex design that is composed of many tasks to learn, time management is essential. We have a limited resource, computing time, that needs to be shared among different tasks. The question is for how long each task should be learned and which ones should be learned before the others. Scheduling is not a new problem in computer science, devising a policy for a fair assignment, like round-robin, has already been thought out. However, fairness is not what we are looking for here. We need a policy that maximizes our learning goal.

Blame assignment

To utilize our limited computation time best, we need to assign more time to learning the task that is responsible for bad performance at the top level of our learning hierarchy. Deciding which task to blame can be done in may ways. One way is to assign the blame to the task that had the highest percentage of execution time during a game. For example, if our team lost a game, and defending was executed most of the time, then it should be learned more than other tasks, or if passing was executed more than other skills in attacking, then it should be assigned more time to learn than other attacking skills.

The other way to design blame assignment is to consider some performance metrics to assess the blame. Some statistics may be gathered during a game which give an insight into which task should be learned more. For example, if the percentage of goalie catches is low, then that indicates the goalie may need more time to learn, or if the percentage of opponents' interception is high, then that suggests the passing skill needs more learning time and so on.

During the life-span of the evolutionary course, the orchestration mechanism will manage the timing of each population's evolution. It will also be responsible for making sure that the low-level skills are learned to a certain extent before the other skills. Furthermore, some skills may be learned more than others, in which case the evolutionary process may be halted by the orchestration mechanism for awhile, giving the computation resources to other skills. This orchestration mechanism will allow the entire process of evolutions to be seen as one gigantic evolution that consists of many populations interacting with each other in different ways. Such as evolution would be more like a form of real-life evolution[1].

[1] After all, that is the ultimate goal for a researcher in this field.

4.0 INCREMENTAL LEARNING

As presented in Chapter Three, low-level tasks are essential for learning the higher-level ones. Some of these tasks are complex, and hence, will be learned in an incremental fashion. In this chapter, the incremental learning of these tasks is described. First, in Section 4.1, a ball interception task is described and the experimental setup is sketched out. Then, in Section 4.2, incremental learning of a shooting skill is explained. Section 4.3 presents detailed empirical results demonstrating the effectiveness of incremental learning in learning those tasks. Section 4.4 provides a discussion of these experiments.

In tackling a complex task such as one in the RoboCup Soccer domain, conventional evolutionary methods suffer from inefficient performance. The task can be too demanding to exert selective pressure on the population during early generations to drive the population to achieve a fruitful solution. As a result, the population can be trapped in a local maxima of the solution space, or it may take a great amount of time to reach an acceptable solution.

In incremental learning, instead of evaluating a population on one task throughout the course of evolution, it is first evaluated on a relatively easy task and then on harder tasks. This may allow the Genetic Algorithm to discover a region of the solution space on the easier task that makes the harder tasks more accessible.

There are two ways to increase the task's difficulty incrementally: first by increasing the fitness function difficulty of the task, and second, by increasing the environment difficulty of the tasks. The former technique is utilized in the shooting task, while the latter is used in ball interception, as described in the following sections.

4.1 LEARNING BALL INTERCEPTION

The first skill that a young soccer player must learn is controlling the ball. Therefore, before learning any complex soccer skills, robotic soccer agents must acquire the ball control skill before accomplishing high-level tasks. To control the ball, the most essential task in robotic soccer is the ability to intercept the ball. Obviously, a soccer player cannot kick the ball before reaching it first. Therefore, this skill is the first task for our robotic soccer agents to learn.

4.1.1 Intercepting a static ball

In this task, the only objects involved are the ball and a learning agent. The agent needs to locate the ball in the field and dash to it. At the beginning of our experiment, the ball is placed at a random spot in the field, with the agent set at random distance from it. The agent's task is to reach the ball.

Figure 4.1 shows a typical initial setup for this experiment. The agent is placed within a distance from the ball no greater than that which an agent which can reach if traveling at maximum speed in the direction of the ball. Note that the agent may not face the ball, i.e. it is not necessary for the agent to sense the ball at the beginning. The agent faces a random direction in the field.

The agent senses the ball's distance, direction and heading. These values have a granularity of 1. Three actions that can be performed by the agent: dashing, turning, and deciding. Dashing determines the speed of the agent and has values ranging from 10 to 100, with granularity set to 30. The agent can turn in 5 degree increments in a 360 degree range of direction. The deciding action determines whether dashing or turning should be executed in a given step. In early experiments, it was noticed that the agent might be in a situation where it unceasingly turns throughout an episode. Thus, a limit on the number of consecutive turns that the agent can execute was set to 3. That means the agent must execute a dashing command after three consecutive turns regardless of the decided action values.

A population of 50 agents is set to evolve for 100 generations, 50 for each stage. At each generation, the agent will undergo 10 episodes of evaluations to determine its fitness[1]. The maximum time for each episode is set to 30 steps, where each step corresponds to one soccer simulation cycle.

An episode is ended if the agent successfully reaches the ball, i.e. the ball becomes kickable, or 30 steps elapses. An agent's evaluation is based on how fast the agent takes to get to the ball, or how far the agent went towards the ball if he runs out of time during an episode.

The fitness function is defined as follows:

$$f = \begin{cases} d \times p & \text{if timeout,} \\ \frac{PT}{t} & \text{otherwise} \end{cases}$$

where d is the relative distance traveled toward the ball, p is a penalty set to 0.5, PT is the ideal time for a robotic soccer agent to reach the ball at maximum speed, and t is the time it takes the agent to reach the ball.

The first part of the fitness function motivates the agent to move towards the ball, while the second part makes sure it reaches the ball as fast as possible.

4.1.2 Intercepting a moving ball

In soccer, the ball is in motion most of the time; hence, intercepting a moving ball is important to achieving other high-level skills. Intercepting a moving ball is more difficult than intercepting a static ball because the ball's movement is not predictable due to the soccer simulator noise and because the agent may not sense the ball all the time as it moves. When an agent is trying to intercept the ball, the ball is moving either toward him or away from him. If it is moving away from the agent, then the agent can still track the ball; however, when the ball is moving towards the agent, the ball can move past the agent's sight and become invisible to him. And since the ball's motion is noisy, the agent cannot predict its motion while it is not visible.

[1] Other SAMUEL evaluations, such as cluster-evaluation, are set to 10 episodes as well.

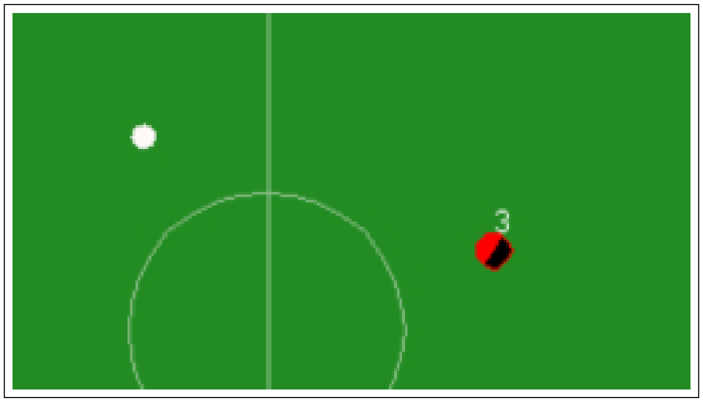


Figure 4.1: Initial setup of ball intercepting experiment

The experimental setup for a moving ball task is carried out similar to that with a static ball, except now the ball is set to accelerate towards the agent at a random angle. The fitness is still measured by how fast the agent takes to reach the ball if it succeeds, or how far the agent goes to reach the ball if it runs out of time.

The agent will learn the goal task, which is intercepting a moving ball, by incrementally learning how to intercept a static ball, then how to intercept a moving ball. Note that only the environment is changed here; the fitness functions are the same in both tasks.

4.2 LEARNING BALL SHOOTING

The soccer game is won by scoring more goals than the opponents; thus, learning to score goals is an important aspect of the game. The shooting task allows soccer players to kick the ball towards the opponent's goal. The opponents goal is not empty though; a goalie is present to defend its goal. The more skillful the goalie is, the more difficult the shooting task becomes. Assuming the opponent's goalie is a competent one, learning to shoot against him becomes difficult. Consequently, a shooting task can be learned incrementally by having a goalie at different levels of competency.

Before a player can shoot the ball, he must first reach it. Once the shooter reaches the ball, he then can kick it towards the opponent's goal. To learn the shooting task incrementally, learning will be carried out in four stages. In the first stage the shooter's task is to get to the ball. The second stage is to kick the ball towards the goal in the absence of a goalie. In the third stage, the shooter learns to score in the presence of a goalie. At the last stage, the shooter learns to score in the presence of a more competent goalie than the one in the previous stage.

For the first stage, the previously learned ball interception skill will be used. In this experiment, the ball is kept stationary; hence, intercepting a static ball is used. After the agent reaches the ball, he must kick the ball towards the goal.

In the second stage, the goal is empty. The agent is evaluated on how good his kick is. If the agent kicks the ball inside the goal, he gets 100% of the payoff. Otherwise, the payoff is determined by how far the ball is from the goal and how far the angle of the kick is from the angle of line between the ball's original position and the center of the goal.

In the third stage, the agent kicks the ball in the presence of a goalie. As in the previous stages, if the agent scores a goal, he gets 100% of the payoff. If the ball is caught by the goalie, then the agent's payoff is calculated based on how far the ball is from the goalie's original position, which is the center of the goal. The farther the ball was from the center of the goal, the better the payoff. The reader may note that this is exactly the opposite of the payoff in the previous stage. The agent's utility now is determined by how far the agent kicks the ball away from the center of the goal (where the goalie is) instead of how close it is to the center of the goal.

The last stage utilizes a more competent goalie than the third stage. The payoff is calculated exactly the same as in stage three. Thus, we only change the environment of the agent, not his fitness.

The experiments start by placing the ball within a certain shooting range from the goal, about 20 to 30 meters. The agent is placed within a random close-distance from the ball, within 5 meters. The agent can sense the ball, the goal, and the goalie. The agent's actions are: turning and kicking. The agent assigns the turn angle, the kick angle and the kick power. Initially, in this experiment the agent had a third action to decide: whether to turn or kick the ball in a given cycle. The results for the shooting skill were good in this setup; however, when the agent used his shooting skill in the presence of defenders, it turned out that the agent would take a long time before deciding to kick the ball and would usually lose the ball before kicking. To prevent the agent from taking too long a time to shoot or from turning forever, the number of turns allowed was changed to be limited to one and after that the agent must execute a kick action. In this way, the agent's action was limited to turning and kicking only. The agent can decide to directly kick the ball, i.e without turning first, by setting the turning angle to zero.

As stated at the beginning of this chapter, incremental learning can be based on environment change or fitness function change. Incremental learning of this skill was done in four stages and by examining the transition of those stages the type of incremental learning can be determined. From the first stage to the second, the fitness is incrementally changed. From the second stage to the third stage, the environment is changed by adding a goalie, and the fitness is changed, as well, to reflect the addition of the goalie's catching event to the environment. From the third stage to the forth stage, the fitness function of the agent is kept the same, only the environment is incrementally changed by introducing a more challenging goalie than the previous stage.

To measure the effect of incremental learning, a set of non-incremental experiments was utilized for comparison. That is, the fourth stage experimental setup was used for an agent that has not been learned in the previous stages. Another experiment was added just to compare the kicking behavior alone, i.e. starting with the ball intercepting skill learned previously and directly trying to learn a shooting skill against a competitive goalie.

The results of these experiments are provided in the following section.

4.3 EXPERIMENTAL RESULTS

4.3.1 Ball Interception

To test the agents' performance, we ran 10 runs for each experiment. Figure 4.2 shows the average performance for intercepting a static ball. The average best performance is 90% and is reached at around 35 generations. The best performance among all the 10 runs is 100%. The standard error is large at early generations, and narrows down in the later generations, suggesting that there is a diversity in the population in early generations which lessens in later generations as the population gains better fitness values.

The performance level for incrementally learning to intercept a moving ball is shown in Figure 4.3. The first 50 generations is where the agent tries to intercept a static ball. The remaining 50 generations is where the ball is set to be moving. That causes a sharp drop in performance in generation 51. The reader may note the rapid increase in performance for generations 50-100. This suggests that either the task of intercepting a moving ball is easier than the one with a static ball or that the agent benefits from the previous learning, making the latter task easier.

In order to study the effect of incremental learning, we devised a control experiment in which the agent tries to learn to intercept a moving ball without any learning experience. Figure 4.4 illustrates the average performance of the agent for 10 runs with a maximum of 100 generations for each run. The best run among all 10 experiments is illustrated in Figure 4.5. The best performance was 92%, and it was reached after 86 generations.

Figure 4.6 plots the average performance of intercepting the ball for the both incremental learning experiments and the control experiments. The error bars are removed for better visualization of the difference between the two types of learning. There is no significant difference in the best average performance between the two; however, the number of generations needed to reach the best performance is certainly different. In the non-incremental learning experiment, the best performance was reached after around 85 generations while the best performance in the incremental learning experiment was reached after only 75 generations. This suggests that the incremental learning reduced the time needed to reach the best performance when compared to the non-incremental one.

4.3.2 Ball Shooting

As in the ball interception task, 10 runs were conducted to measure the performance of the agent on the shooting task at each stage. For the first stage, the agent's task is to intercept the ball. Figure 4.2 shows the performance for intercepting a static ball as mentioned in the previous subsection.

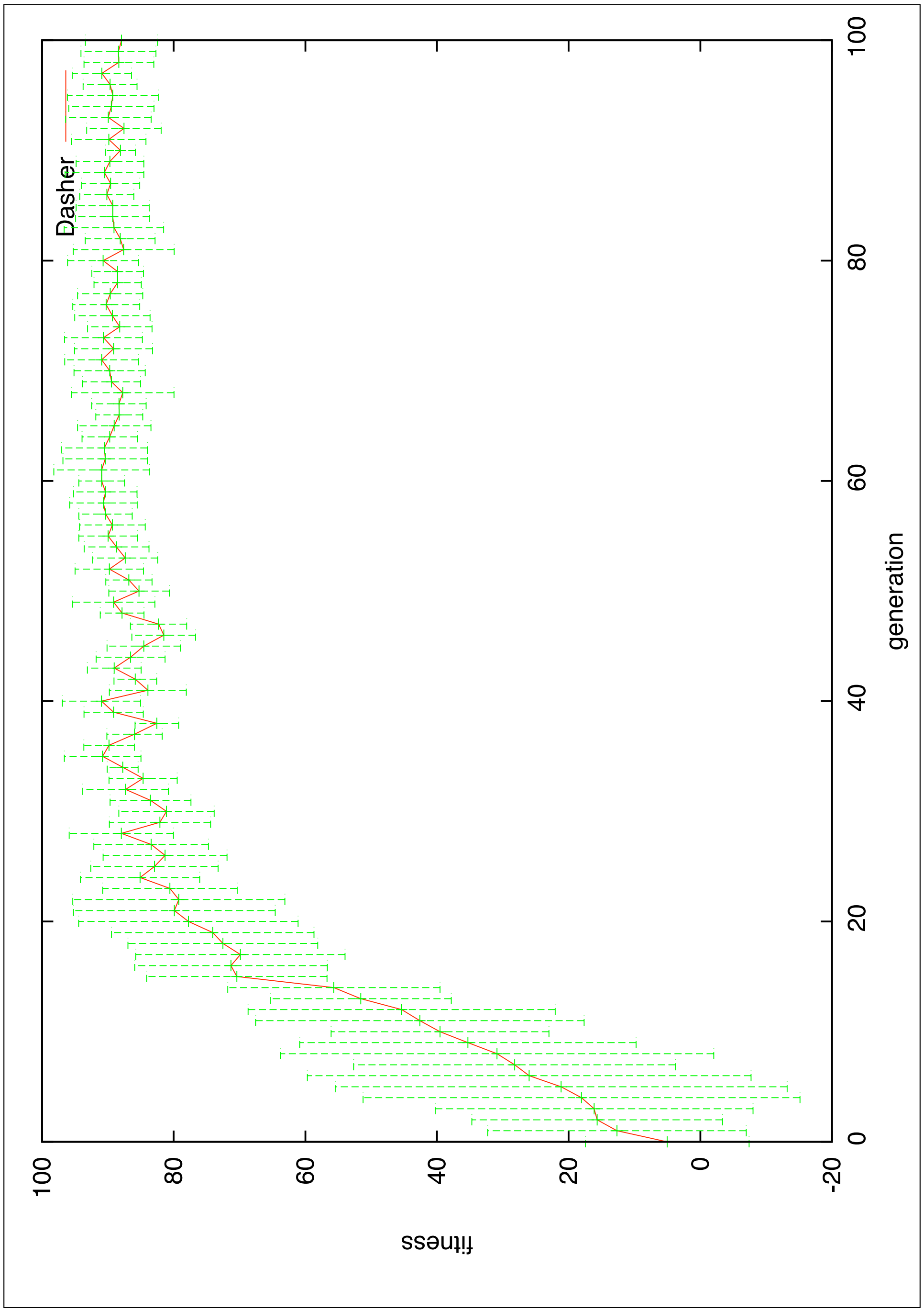


Figure 4.2: Performance of learning static ball interception

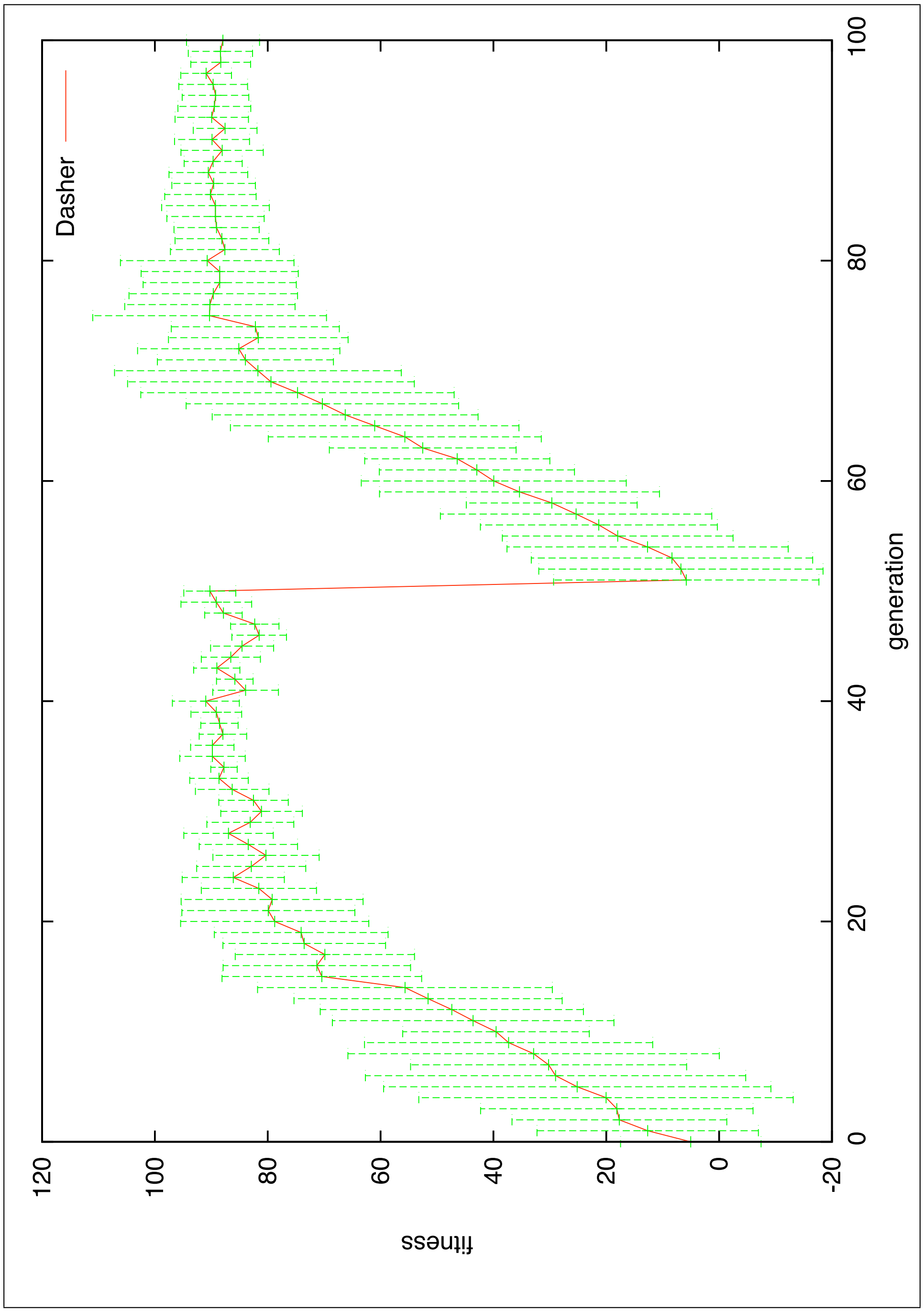


Figure 4.3: Incremental learning performance of moving ball interception

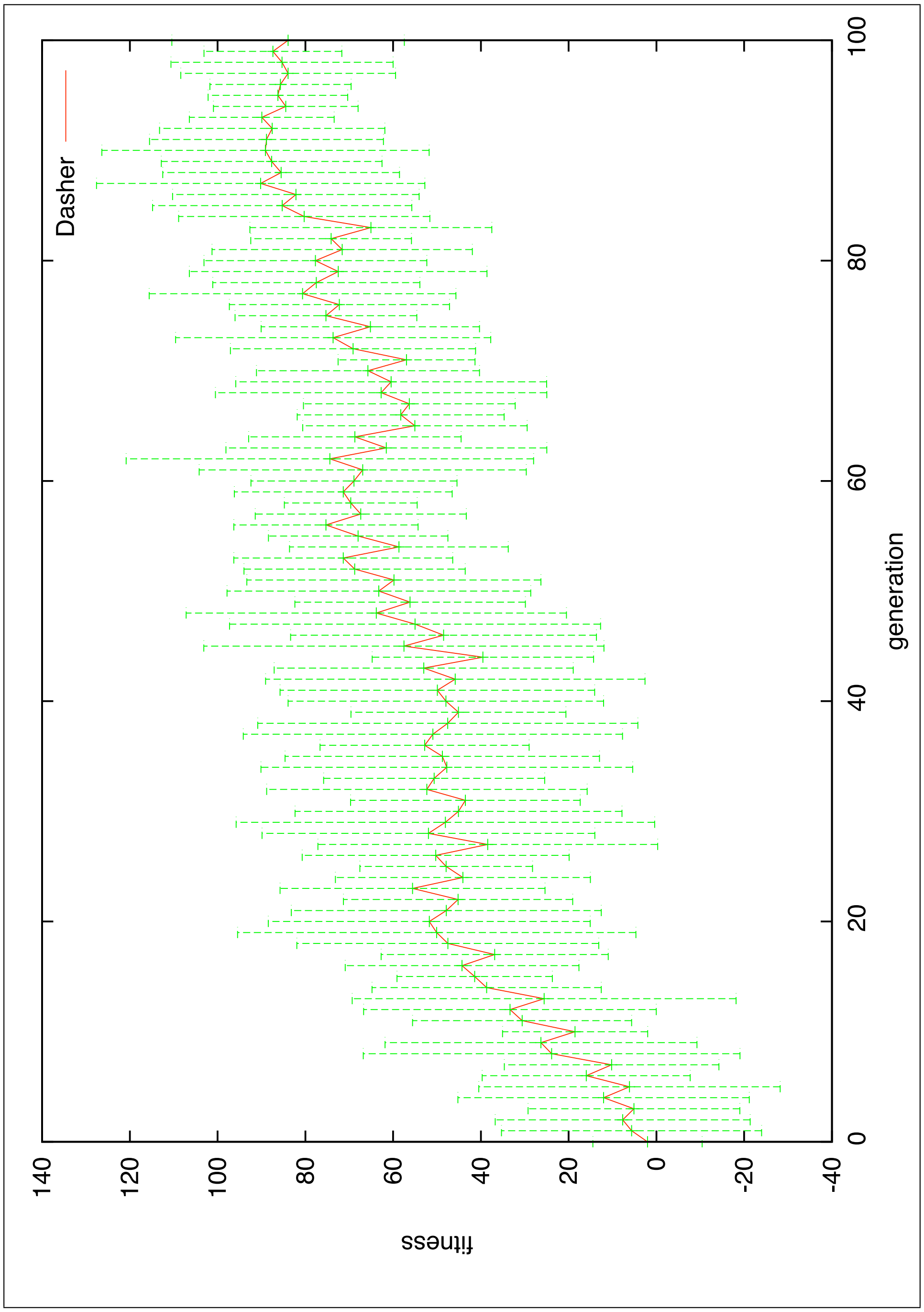


Figure 4.4: Non-incremental learning performance of moving ball interception

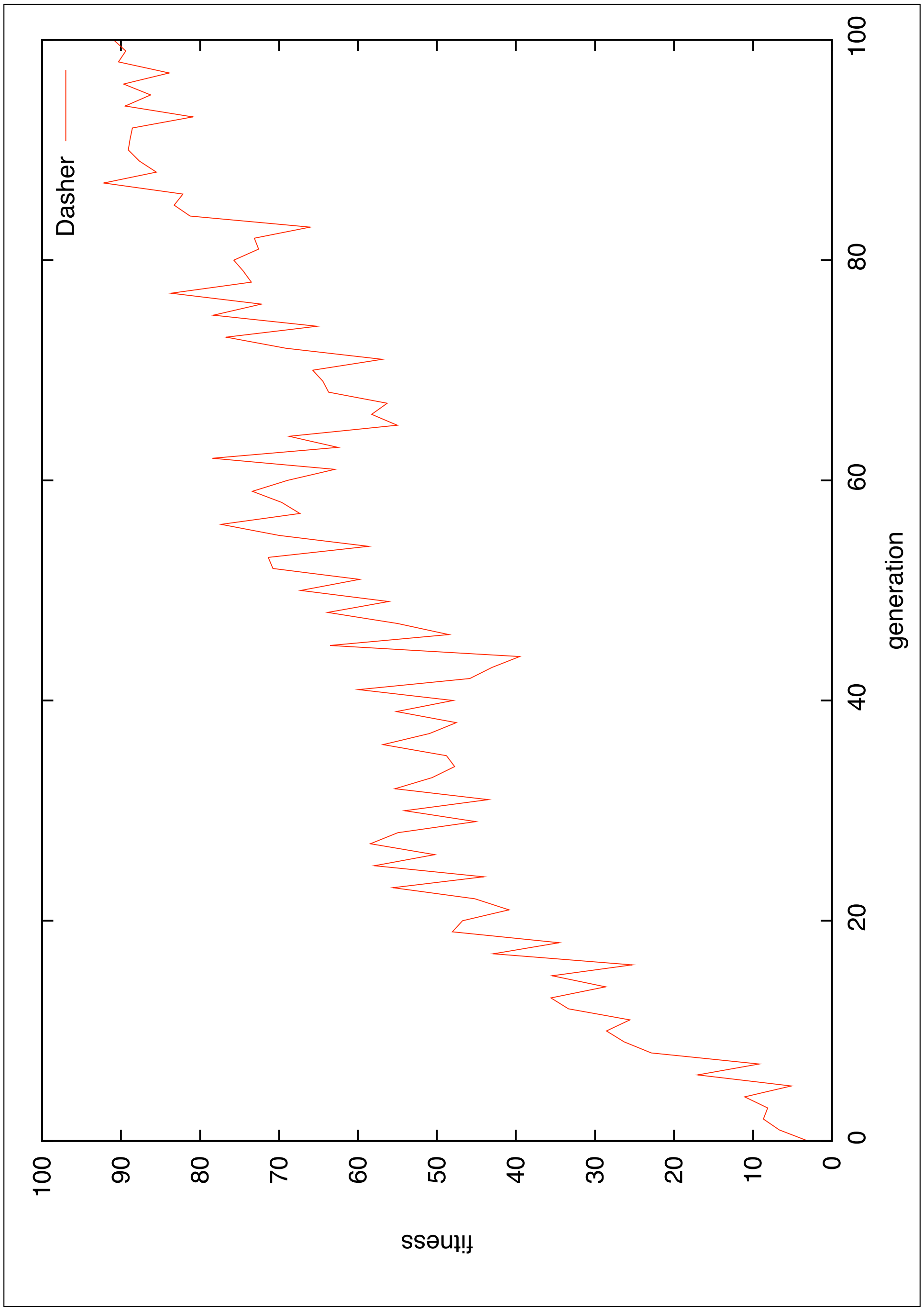


Figure 4.5: Non-incremental learning performance of moving ball interception (best run)

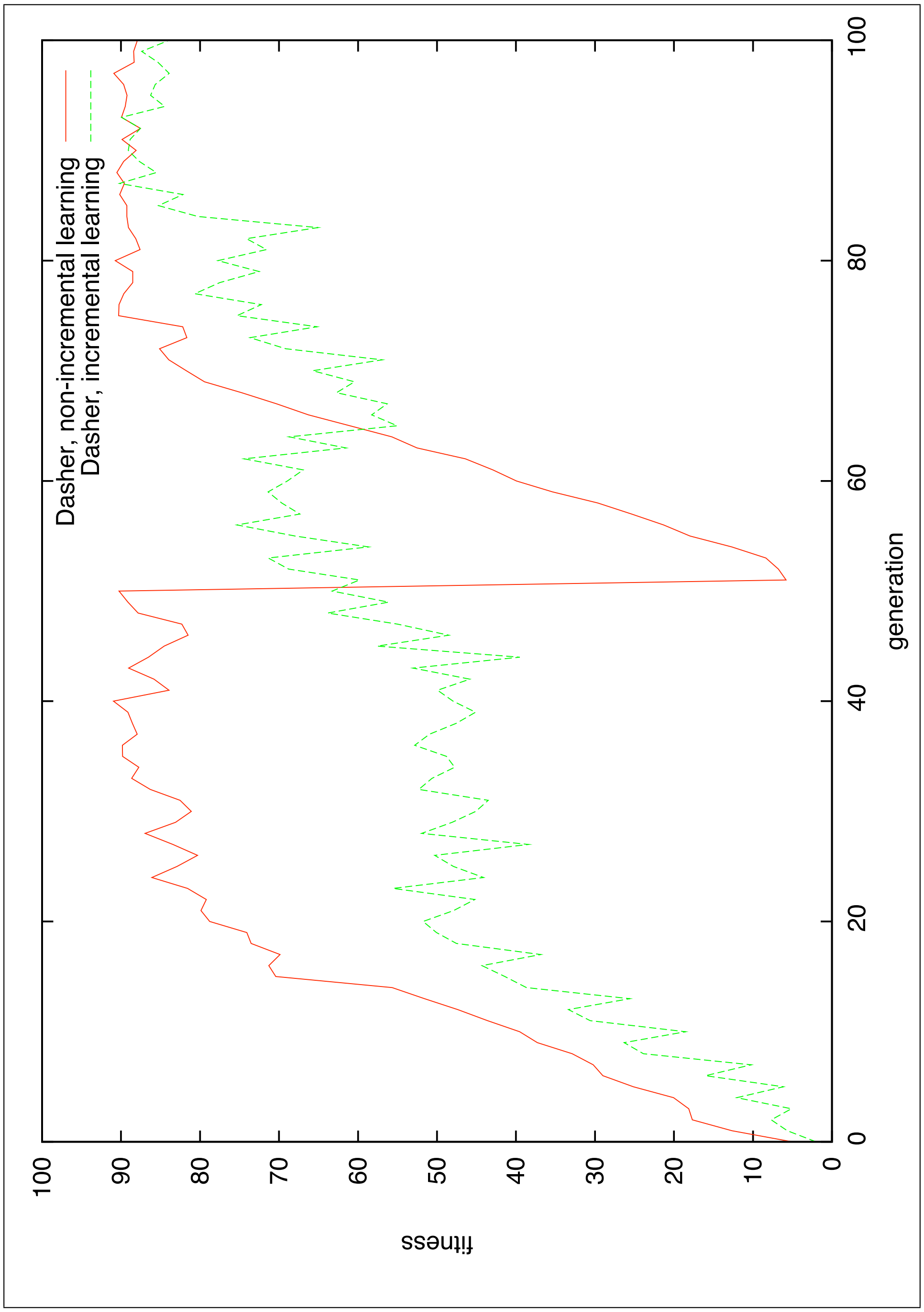


Figure 4.6: Performance of incremental vs. non-incremental learning ball interception

For the second stage, the agent must learn to kick the ball into an empty goal; however, the agent must first reach the ball in order to be able to kick it. A dummy non-learning rulebase agent was devised to decide whether to intercept or kick the ball. This rulebase simply senses if the ball is kickable and its action is to decide to intercept or kick. Figure 4.7 shows the average performance of the agent in the second stage. As suggested by the graph, this task is easily learned in a few generations. The best performance was reached after only 23 generations. An average best performance of about 95% was achieved, with a maximum payoff of 100% apparent in the population in most of the 10 runs.

Figure 4.8 displays the performance of the agent in the third stage. The existence of a goalie makes this task more difficult than that in the previous stage. The goalie rulebase at this stage was taken from previous experiments of training a goalie, which had a moderate performance. As seen in the graph, the agent reached a best performance of 76% on average. Examining some of the agent's behavior visually gives us some insight into how difficult the task is. At early generations, the agents struggle to learn to kick the ball away from the goalie, which was learned in the previous stage. As learning progressed, the agent kicks the ball near the posts; however, in many cases, the ball just misses the goal and ends up out of bounds. That behavior can also be seen in the graph of Figure 4.8. In the middle of the graph, the standard error was high, suggesting the oscillation of high payoff that is achieved when scoring, and low payoff when the ball is kicked out of bounds. In the last few generations, the agent tends to kick the ball a little bit away from the center of the goal, but not near either of the goal posts. This seems to be the best strategy learned by the agent as in many cases the goalie at this stage was not able to catch the ball if it was kicked less than 2 meters away from him.

The fourth stage performance is illustrated in Figure 4.9. Here, the average best performance is 65%. The goalie was able to catch most of the balls kicked. Consequently, the agent had to kick the ball near the posts to achieve a reasonable performance. When examining the graphical behavior of the best performer agent, this was the case. The agent in this stage has the same behavior as the one at the middle generation of the third stage. The agent tends to kick the ball towards the posts; nevertheless, the ball ends out of bounds which causes a reduced payoff some times.

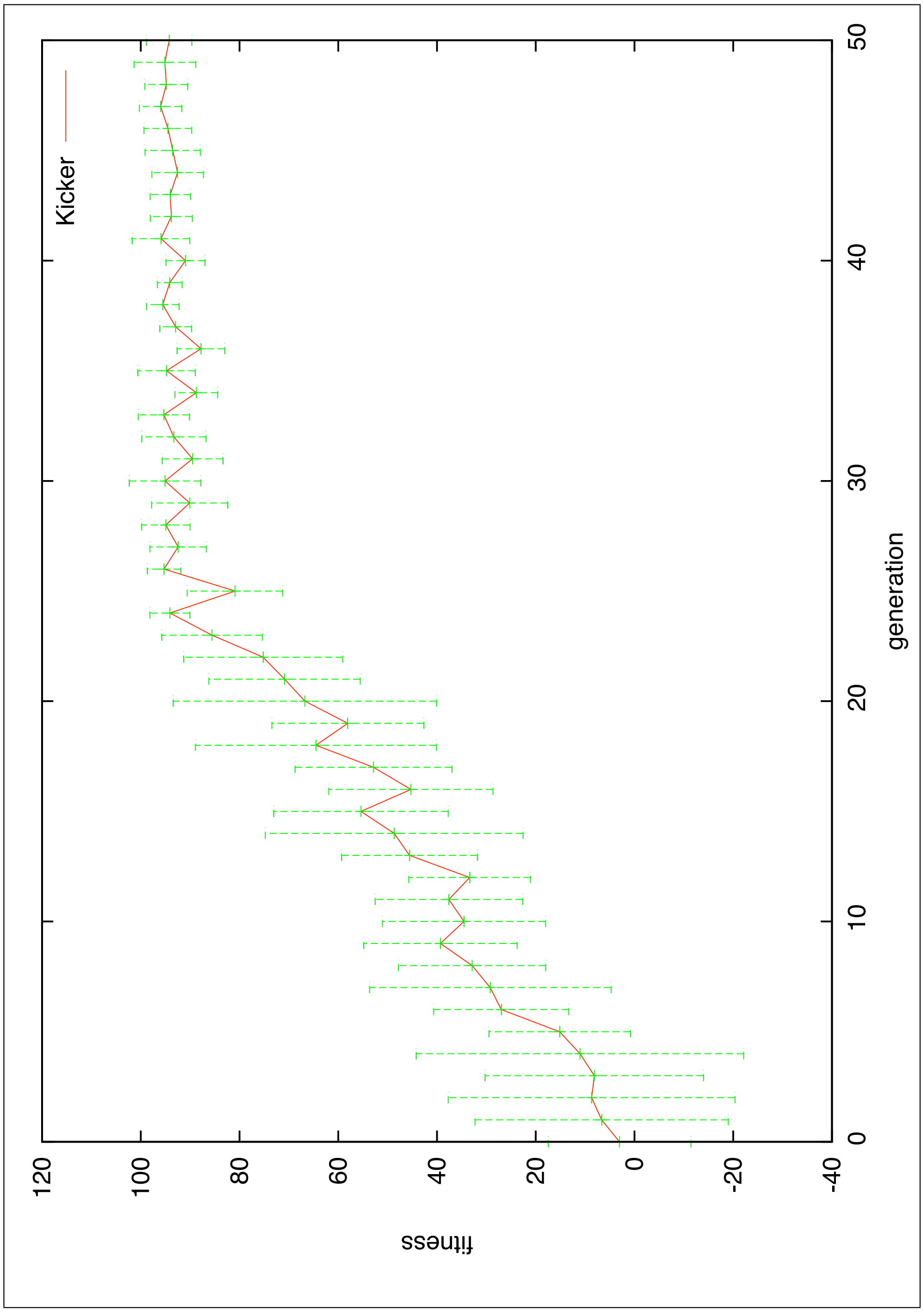


Figure 4.7: Performance of incremental learning ball shooting – Stage 2

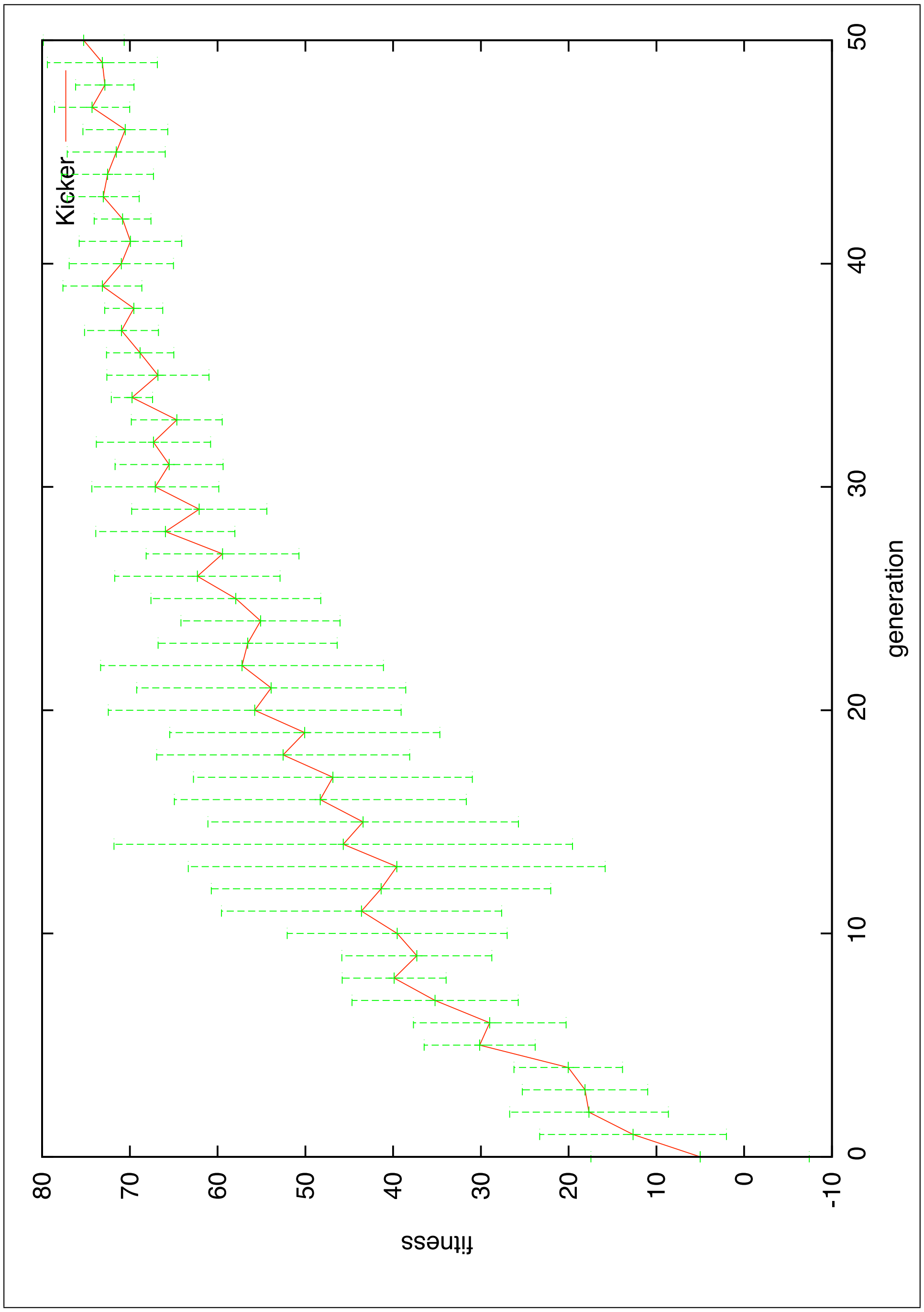


Figure 4.8: Performance of incremental learning ball shooting – Stage 3

To show the effect of incremental learning, a non-incremental learning experiment was executed. Figure 4.10 plots the average performance for incremental and non-incremental learning. The standard error bars were omitted for better viewing of the performance difference. In this particular experiment, the non-incremental learning experimental setup is simply the same as stage four; however the agent has no prior knowledge, even of intercepting a ball. The agent here senses the ball, the goalie, and the goal, and must execute a dash, a kick or a turn action. A "decider" action is introduced to decide what action to execute at a particular step. As can be seen in the graph, the agent failed miserably to learn the task of shooting within the 200 generations allotted. The average best performance for the non-incremetal learning was 20%.

Figure 4.11 plots the performance for incremental learning and another non-incremental learning regime. At this time, the non-incremental agent learned to intercept the ball; however, it tries to learn to shoot against the fourth stage goalie directly. The best performance of this agent (in 150 generations) was 36%. This is much less than the average performance of the incremental learning agent.

4.4 DISCUSSION

In this chapter, incremental learning is used to learn tasks. With respect to the ball interception task, incremental learning shows that learning can reach a reasonable performance faster than a non-incremental one. In the shooting task experiments, the incremental learning shows a superior performance to the non-incremental one and shows that the incremental learning is a great factor in achieving an acceptable performance on a ball shooting task.

In incremental learning, one must decide the appropriate time to transfer learning from one stage to another. In this chapter, the number of generations (50 generations) was simply used to trigger learning between stages. Sometimes, it is not known what is the best number of generations to be used in a particular problem. However, population fitness values can be used to set a threshold to trigger a learning stage shift. Average fitness, maximum fitness, and minimum fitness thresholds are some that were experimented with in this research and using them provided results comparable to use of a generational threshold. However, it becomes evident that it is hard to decide on what value is appropriate for fitness thresholds, especially when time is a constraint.

Incremental learning has proven to be a useful technique when learning a difficult task like ball shooting. However, when the task is too complex, such as when playing soccer, it becomes hard to come up with a way to incrementally learn a task the same way as laid out in this chapter. A better way to learn such a complex task is to decompose it into smaller tasks which can be learned easily. The next chapter illustrates how task decomposition can be used to tackle learning of a complex task.

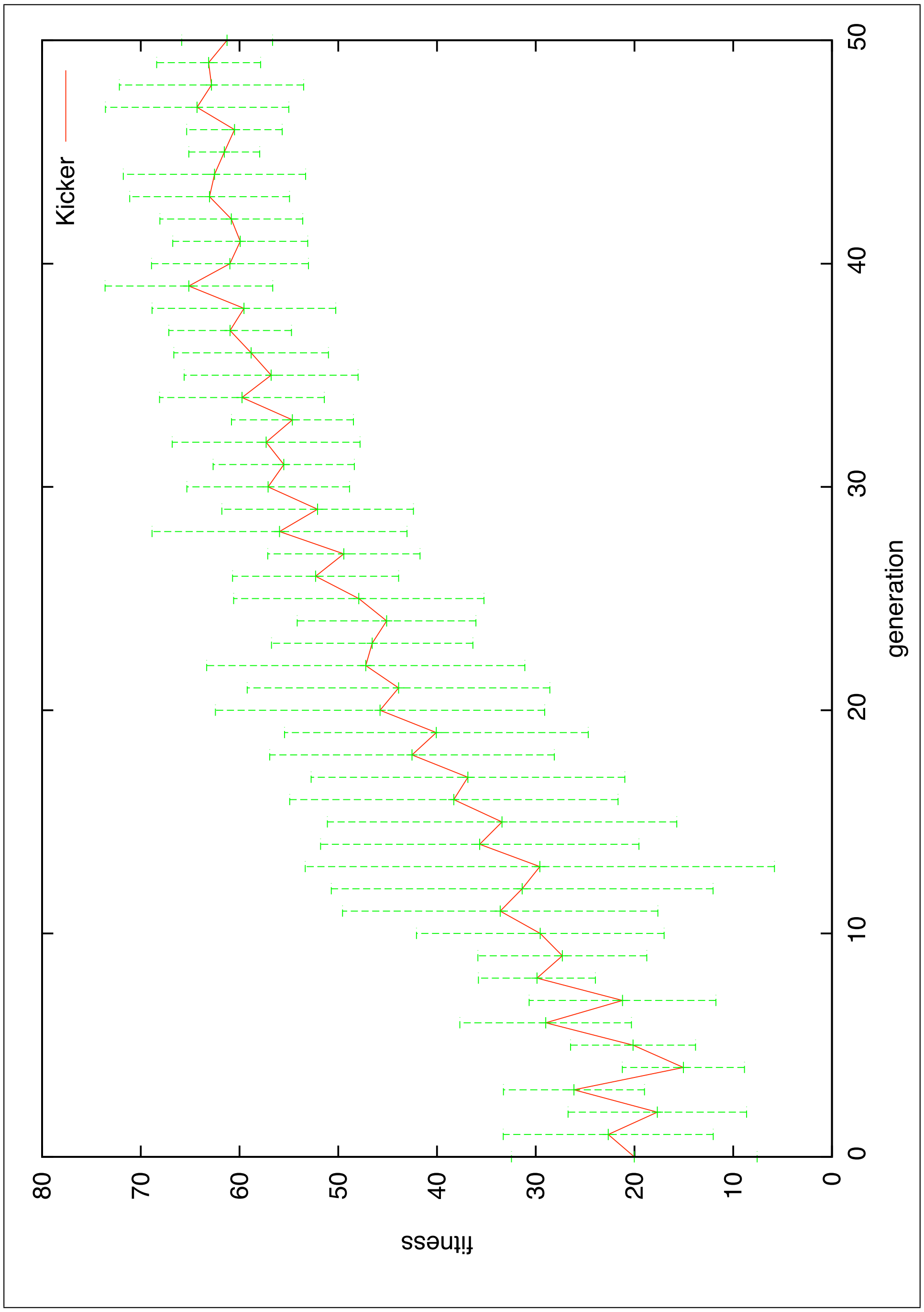


Figure 4.9: Performance of incremental learning ball shooting – Stage 4

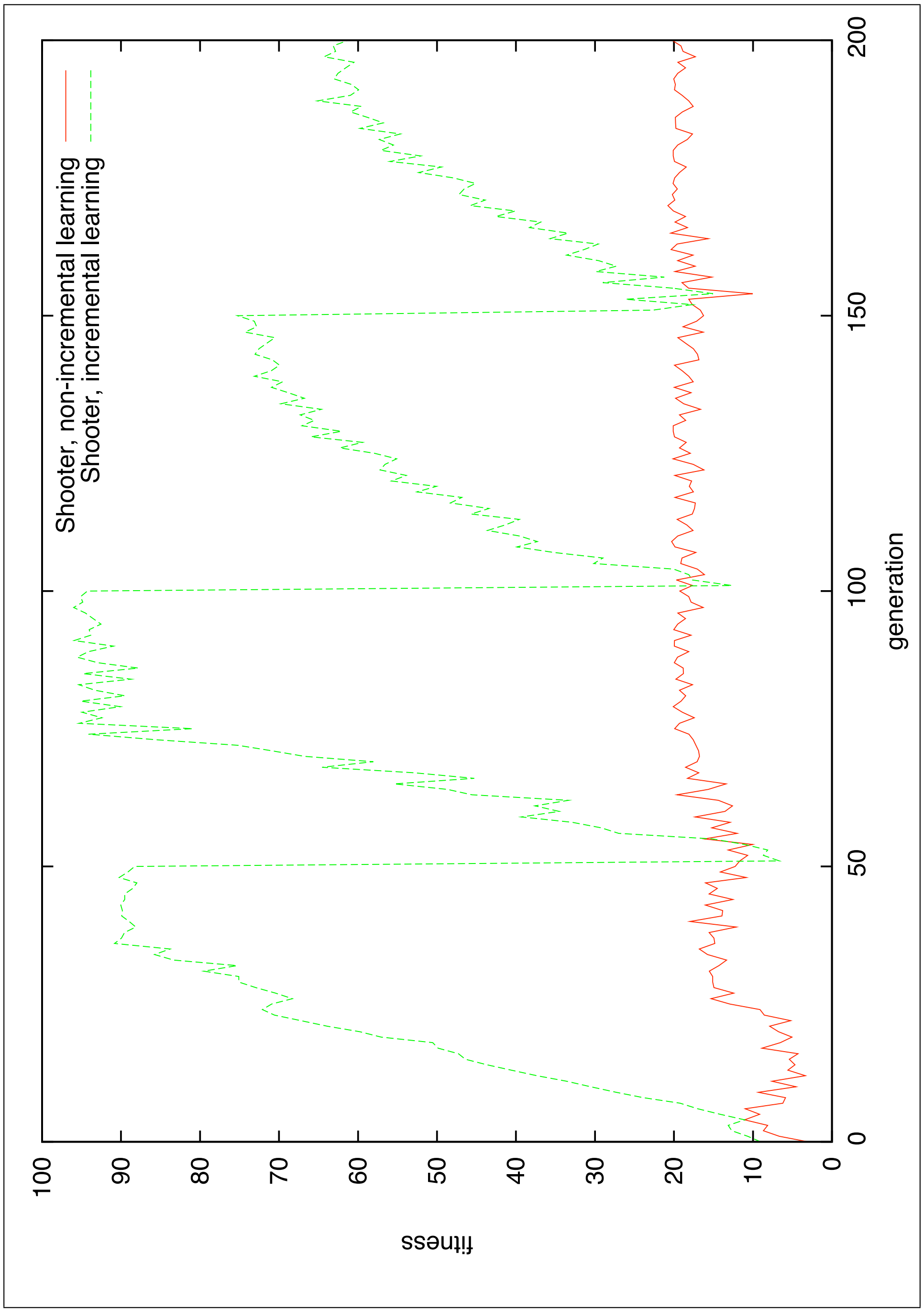


Figure 4.10: Performance of incremental vs. non-incremental learning ball shooting 1

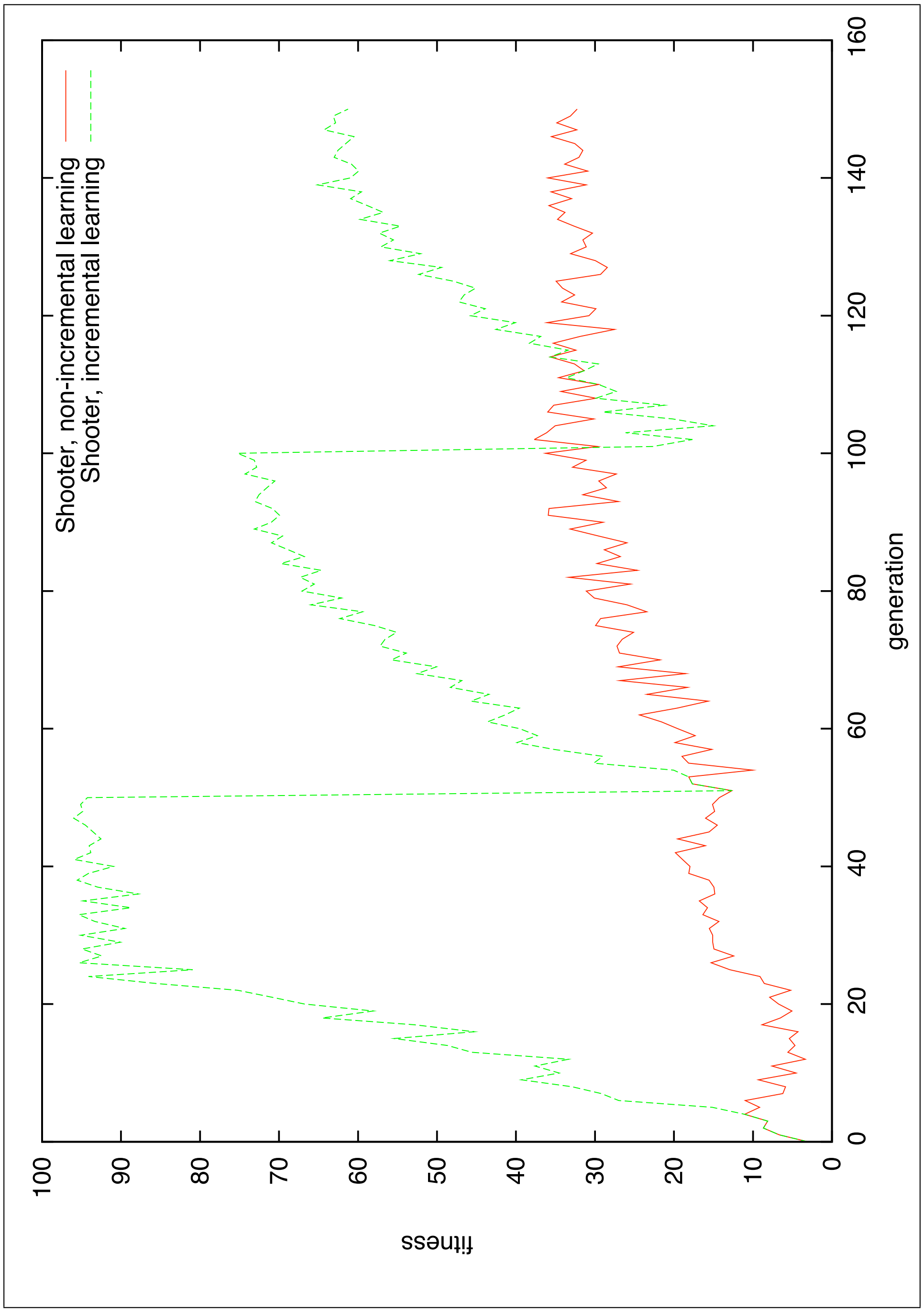


Figure 4.11: Performance of incremental vs. non-incremental learning ball shooting 2

5.0 HIERARCHICAL COEVOLUTION

Our goal is to have an agent that learns how to play soccer in a robotic simulation. In order to achieve this goal, one must divide the task of playing soccer into smaller tasks which can be more easily learned. In Chapter 3, a hierarchical tree of subtasks, Figure 3.4, shows the relationships among these subtasks, and which subtask is used by other ones. At the top of the hierarchal tree is our goal task, which is playing soccer. Therefore, one way to carry on learning is to allow the learning to proceed from the lower level of the tree to the top levels. In this chapter, the learning of these subtasks is described.

This chapter is organized as follows. Section 5.1 describes the basic skills of intercepting, kicking and catching the ball. Section 5.2 covers soccer skills which are involved in ball maneuvering. These skills include ball dribbling. opponent's marking, ball passing, pass receiving, and pass blocking. Section 5.3 describes high-level skills, namely attacking, and defending. Section 5.4 illustrates how agents play a full game of robotic soccer. Section 5.5 is devoted to discussion and remarks.

5.1 SETTING THE STAGE

Intercepting a ball is a basic skill that a soccer player must acquire before learning any other skills. Hence this task needs to be learned fully, before any other skills can start learning. Learning ball intercepting, described in Chapter 4, is incorporated with the learning of a variety of skills. For example, shooting skill, the ability to kick the ball successfully, must be used to reach the ball before executing a kicking action. In a soccer school, the first thing a player may learn is how to kick the ball. Shooting involves another skill, that of goal tending.

This makes it seem natural to learn both tasks at the same time. Below is an outline of an experiment that characterizes the learning of those skills together.

Coevolving shooting and goal tending skills

Two agents were coevolved synchronously; a goalie and a shooter. The goalie's task was to prevent the ball from entering the goal. The goalie sensed the ball distance, heading and direction, and its actions were: turning, dashing, and catching the ball. The initial setup was positioning the goalie at the center of the goal, and the ball at a random position 20-30 meters from the goal. The Shooter was placed within 5 meters of the ball. The Shooter used the ball interception skill to reach the ball before kicking it towards the goal. While the Shooter's fitness is calculated as described in Chapter 4, the goalie's fitness is calculated in exactly the opposite way. If the goalie catches the ball, he gets 100% of the payoff. If the player scores with the ball, then the goalie's payoff is calculated based on how far the ball was from the center of the goal. The farther the ball is from the center of the goal, the better the payoff is.

Figure 5.1 shows the performance of both players in 100 generations. In early generations the goalie showed superior performance over the shooter. That is expected since the task for the goalie is easier in the sense that the shooter is just trying to learn how to kick the ball to the goal. Then the shooter had a better performance than the goalie, followed by a better performance for the goalie. This behavior is typical in competitive coevolution; it is called an arms race. The fitness of both players oscillated, like a sin/cosine function, as they proceeded in time. Overall the goalie had the upper hand and showed a better fitness most of the time.

The visual behavior of the goalie and the shooter is interesting. In the early generations, the goalie tended to stay at its position and tried to catch the ball. Then as the learning progressed, the shooter learned to kick the ball to the right of the goalie to score. Then the goalie moved to the right side, catching most of the balls kicked there. Afterwards, the shooter tended to kick the ball to left side of the goal. Then the goalie started going to the left side to catch the ball. It seems both the shooter and the goalie were trying to counter each others behaviors as learning progressed.

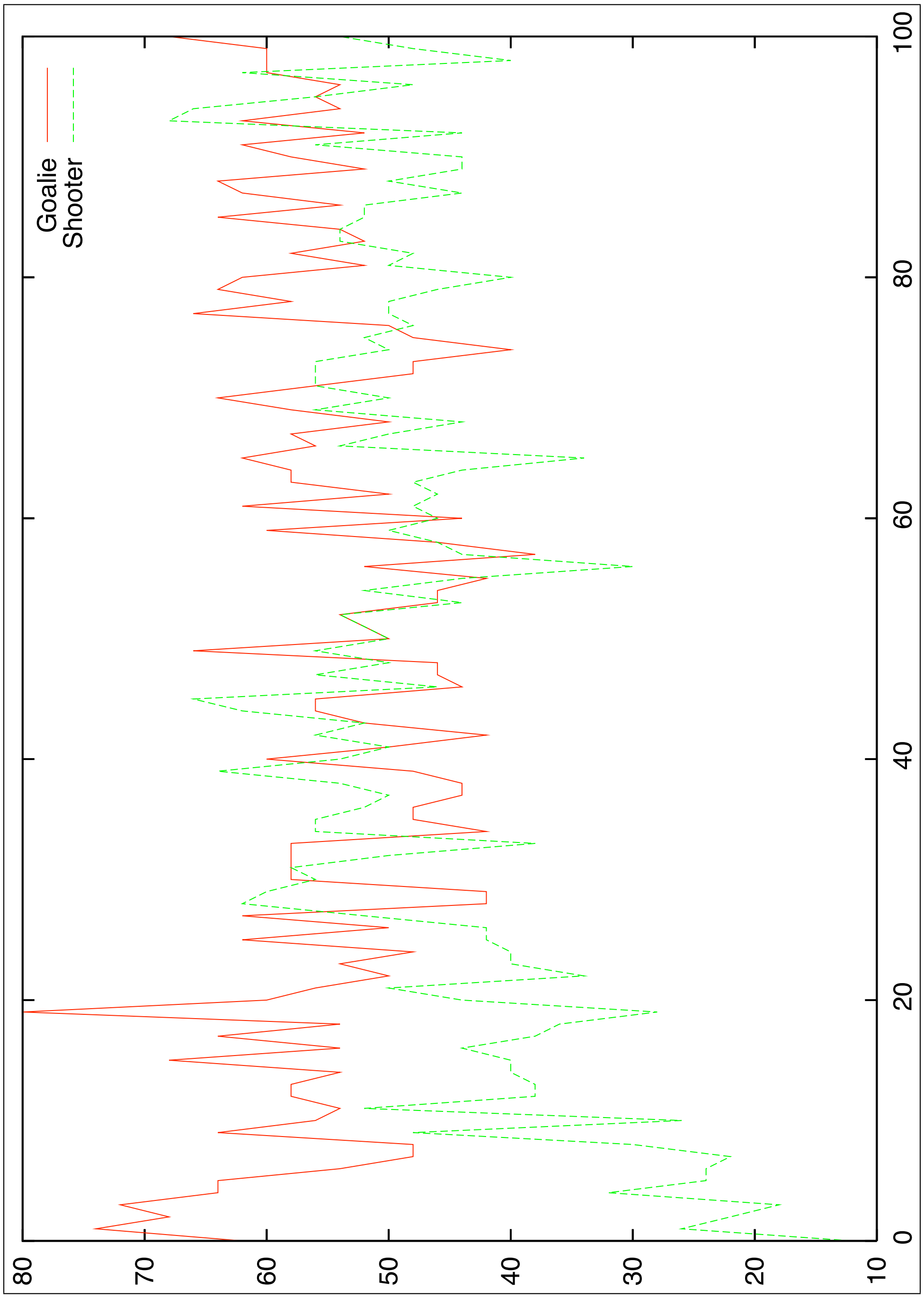


Figure 5.1: Performance of goalie and shooter

The behavior of the coevolution experiment motivated us to evolve the shooting and goal tending tasks separately. Our goal is to attain a high level of soccer skills, not to develop a behavior in which skills adapt to each other. Therefore, the shooting task was learned separately in an incremental learning fashion as discussed in the previous chapter, and the goalie was learned with a non-learning ball kicker as described below.

Evolving goal tending with hand-crafted shooter

In this experiment, the goalie is pitted against a shooter that kicks the ball towards the goal at a random angle. The goalie uses the same sensors and actions and utilizes the same fitness function as in the coevolution experiment.

Figure 5.2 illustrates the performance of the goalie in 100 generation. The best performance of 73% on average was achieved at generation 68. The confidence interval, as shown in Figure 5.2, suggests that the population is still diverse, and a gain in performance is possible if more generations than 100 are evolved. At early generations, the goalie behavior tends to be the same as in the previous experiment, i.e, going always to one side then shifting to always going to the other side. However, that behavior disappears after 51 generations. The goalie's behavior then becomes comparable to a professional goalkeeper. Moreover, the goalie moves forwards to narrow the angle between the ball and the goal posts which makes scoring more challenging.

5.2 BALL MANEUVER SKILLS

This section describes the learning of many defensive and offensive tasks used in RoboCup soccer. Some of these tasks are learned separately, viz. dribbling, marking, and ball clearing, while others are coevolved concurrently with each other, such as passing and receiving a ball.

Dribbling

The task to be learned here is to move from one place to another in the field while keeping control of the ball. The player was positioned at a random place in the field and its task was to reach a target point (which is the goal in this experiment). Another player that tries to intercept the ball was also positioned at a random spot in the field between the dribbler

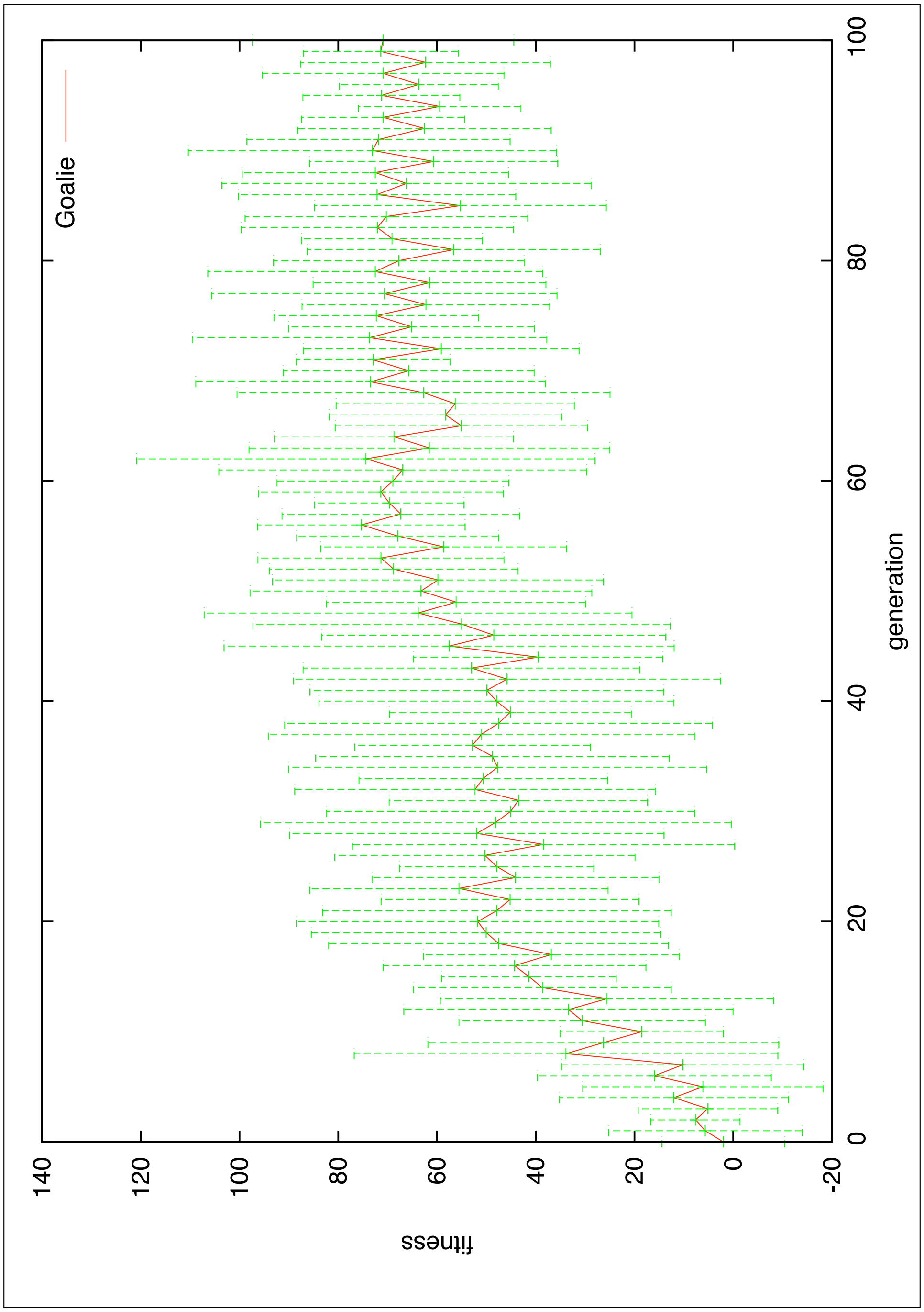


Figure 5.2: Performance of goalie

and the target. The player learning dribbling senses the ball and the opponent's distance, direction and heading and the target distance and direction. Its action options were: turn, dash, or kick the ball. The episode ended when the player reached his target or lost the ball or a fixed time elapsed. The fitness function was calculated according to whether the player reached its target (50% of payoff) and the amount of time the player had control of the ball provided that the player moved at least 30% towards the target (50% of payoff). The player was also penalized if the opponent intercepted the ball with 20% of its payoff. The dribbling task is difficult, since the player must learn how to alternate between moving and kicking the ball until he reaches his target and avoiding the interceptor.

Figure 5.3 shows the performance of the player for 10 runs with 95% confidence intervals. The curve provides evidence that this task was not easy. At early generations, the fitness values started from below 10 and progressed to about 40 on average until 200 generations[1]. The confidence intervals were very large, suggesting that the performance varied from one experiment to another. The best performance achieved was 52.

The visual behavior of the player shows how this task was difficult. It is hard because the player has two objectives: to reach his target, and to keep the ball with him all the time. A typical behavior of a sample of episodes at early generations shows that the player was to reach the target without the ball; however, in later generations, the player learned to dribble with the ball but without successfully reaching its target, due to timeout or the ball being intercepted, in many times.

Marking

Marking is a task in which a player follows an opponent. The player senses the opponent's distance, direction and heading, and executes a turn or dash command. The fitness of the player is determined by the percentage of time the player is close in distance to its opponent (3 meters) compared to the overall episode time. Figure 5.4 plots the performance of the marking task in 10 runs. The player was able to achieve an average performance of 80% after only 65 generations.

[1] Due to the difficulty of this task, the number of generations was extended to 200.

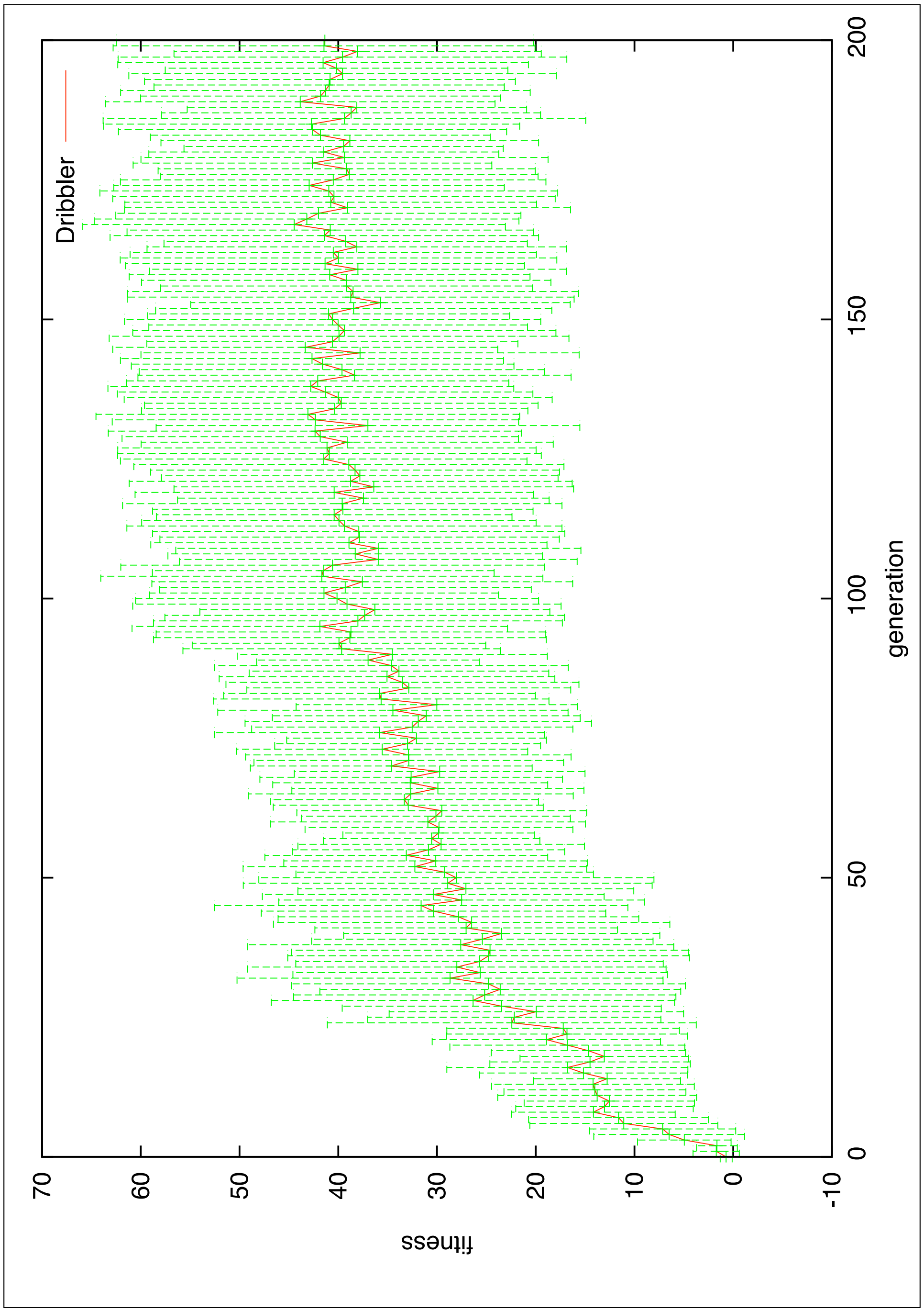


Figure 5.3: Performance of dribbling

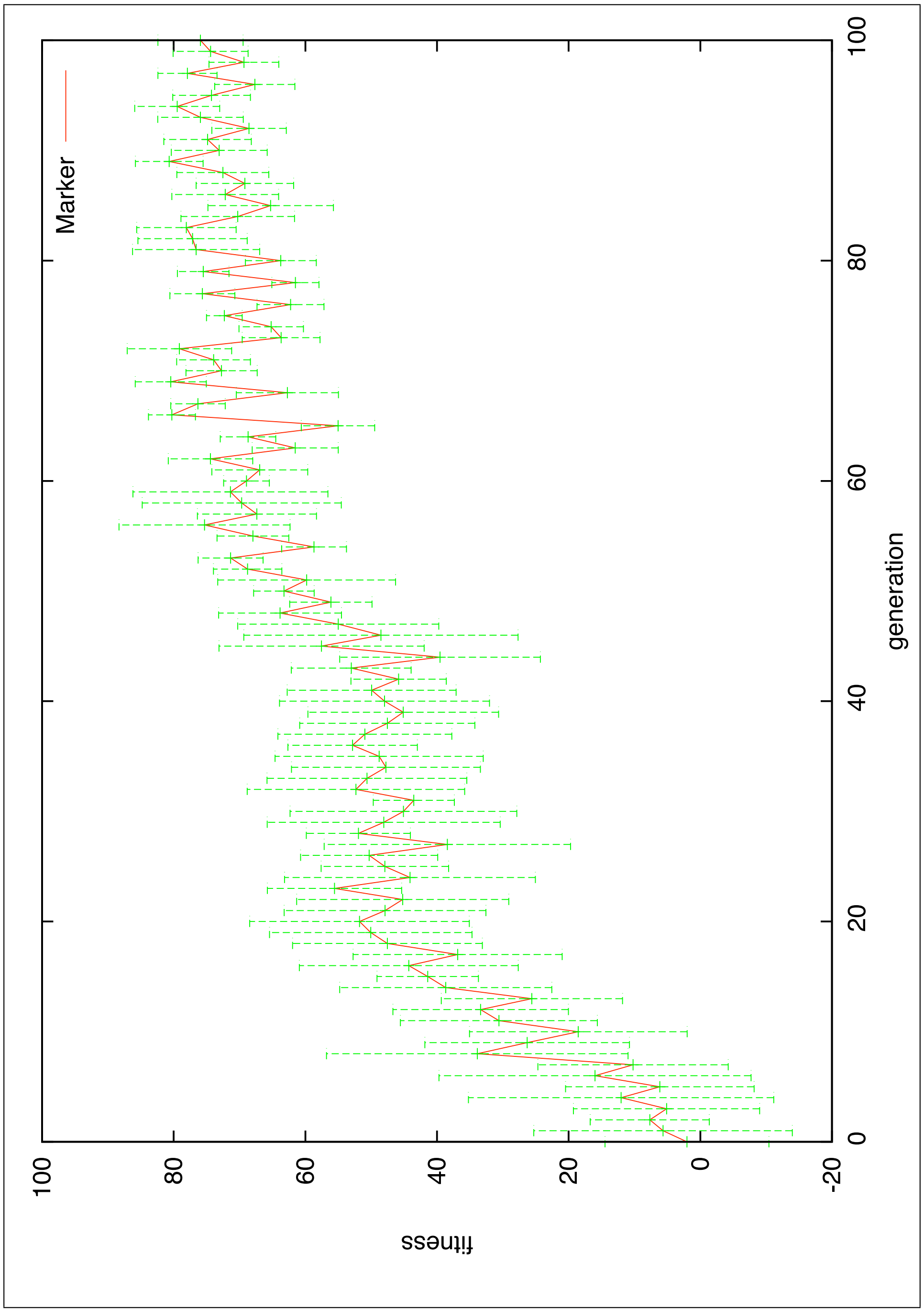


Figure 5.4: Performance of marking

Passing and receiving

A passing skill was learned concurrently with a getting open skill. The task in this experiment was to kick the ball towards a teammate in the presence of a ball interceptor. Initially, the interceptor was placed on the line between the passer and the receiver. The passer agent would sense his teammate's distance, direction and heading, and his action choices were to turn or to kick the ball. The fitness function of the passer is based on how far the ball travels towards the receiver and how different the angle of the kick was from the line between the passer and the receiver. The passer got a 70% penalty if the ball was intercepted by the opponent, and he was rewarded with 75% if the pass was successful one[1].

For the task of getting open, the learning player senses the passer's distance and direction, the ball distance, direction and heading, and a pass-success sensor. The pass-success sensor determines the probability of a successful pass – one in which the ball will not be intercepted by an opponent[2]. The player's actions at this point are to turn or dash. The goal of getting open is two-fold: first, to maximize the pass success probability, and second, to get closer to the opponent's goal. Hence, the fitness is calculated in two parts. The first pays off 100% if the player improves the pass success probability from its initial setting by at least 50%. The second pays off 100% if the player moves from its initial position to the closest possible position to the opponents goal at maximum speed. Naturally, The player gets a partial credit relative to its performance in achieving these goals.

Figure 5.5 and Figure 5.6 show the performance for the passing and the getting open skills, respectively.

Pass evaluation, mark evaluation and blocking

After learning how to pass the ball and how to mark an opponent, three tasks were learned together at the same time in this experiment. The first, a pass evaluation task, involves determining which teammate is the most valuable candidate to receive the pass. The pass should reach the most open teammate and/or the one who is closer to the opponent's goal. The player senses all players distances, directions and headings, and its action is to provide

[1] That means the fitness is at most 30% if the ball is intercepted, and at least 75% if the pass is successful.

[2] This is done by calculating the relative distance of the opponent to an oval area drawn between the position of the passer and the receiver.

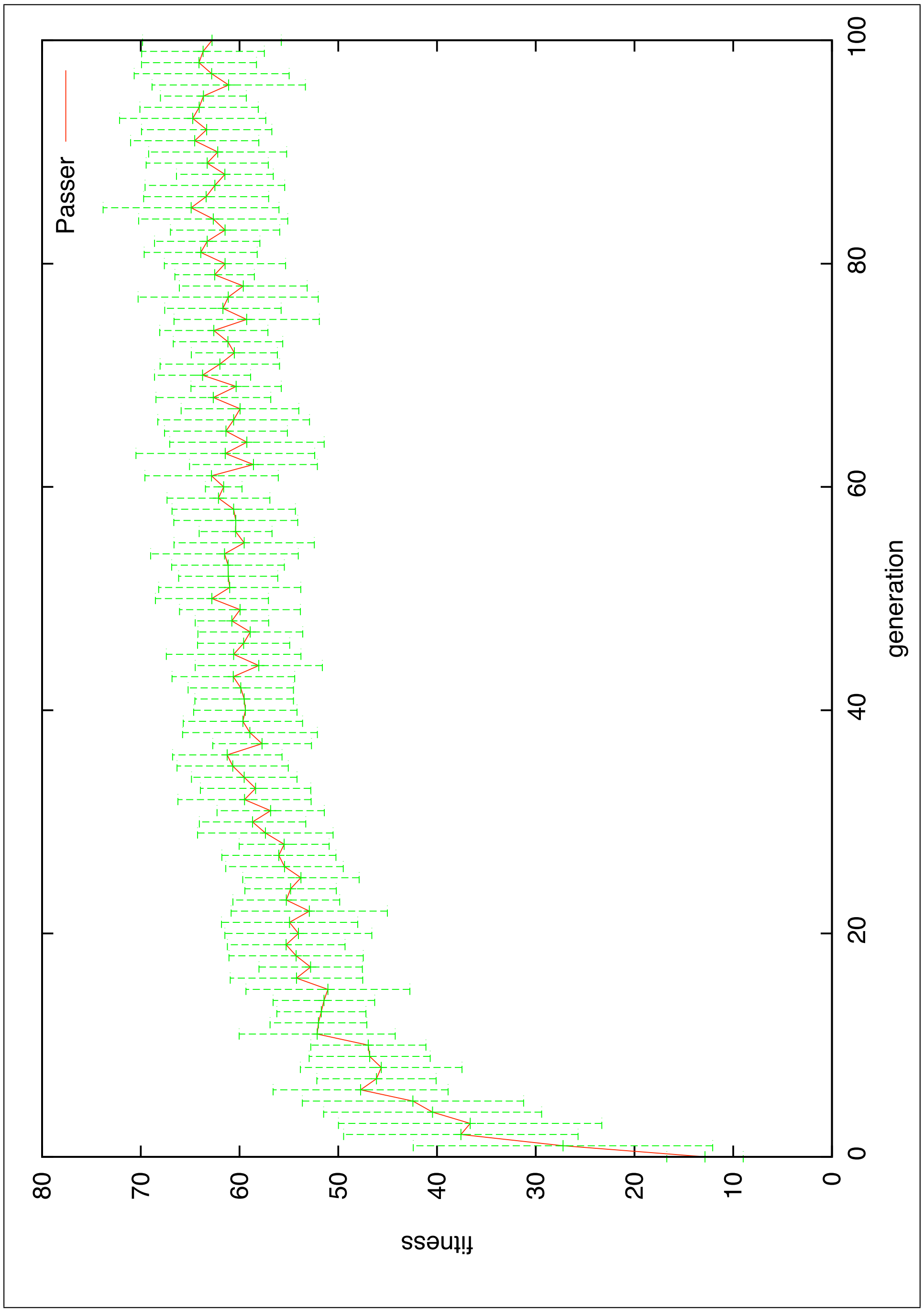


Figure 5.5: Performance of passing

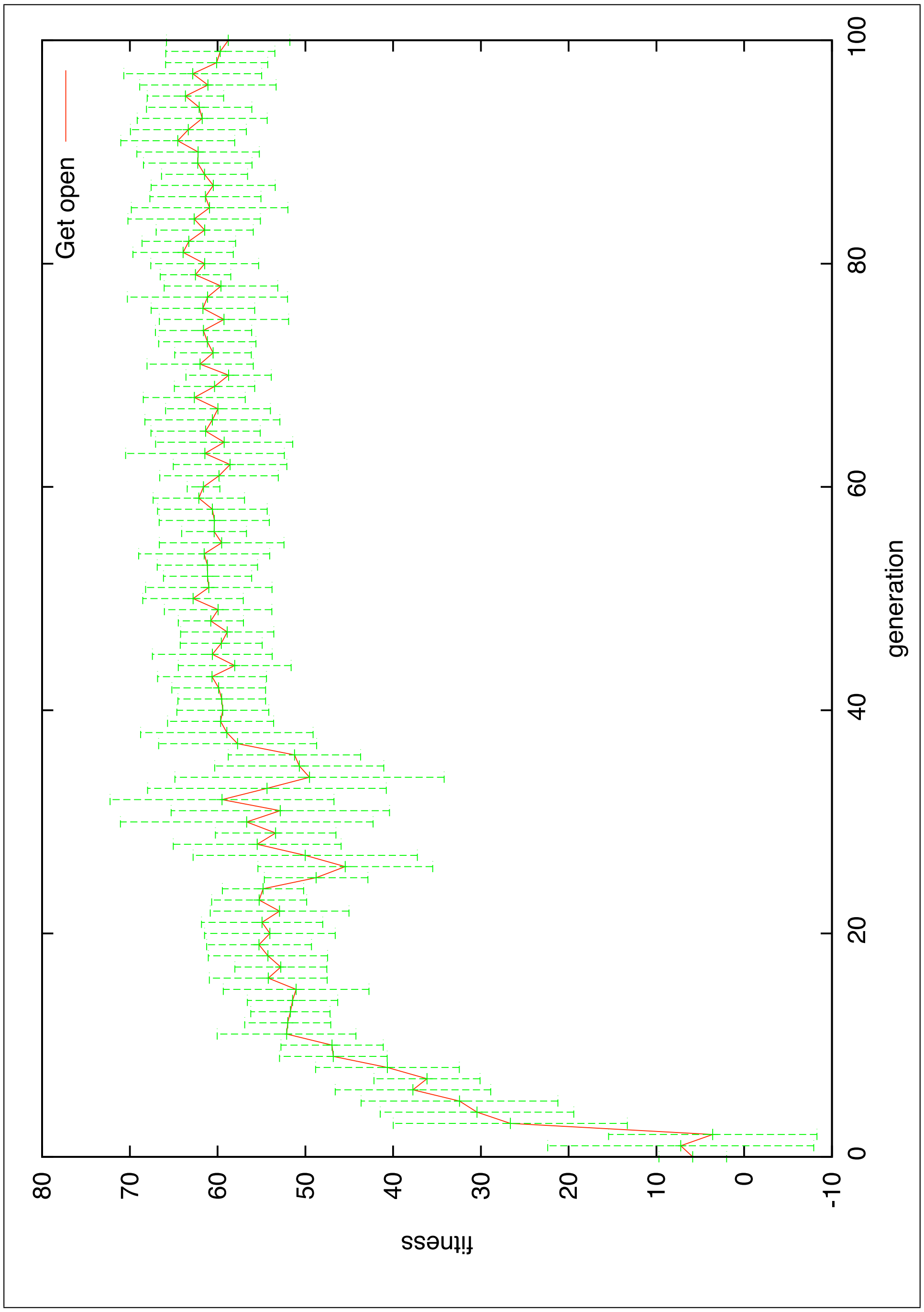


Figure 5.6: Performance of get open

a number that determines which teammate the ball should be passed to. Figure 5.7 shows the performance for the pass evaluation skill.

In the mark evaluation task, the player must select an opponent to mark, or guard. The player senses the three teammate's, and four opponent's distance, direction and heading, and it has to choose an opponent to guard as an action. At this time, the marker should mark the closest opponent[1]. Therefore, the fitness is simply calculated: 100%, 75%, 50%, and 0% if the player chosen is the closest, the second-closest, the third-closest, or the fourth-closest to the player, respectively. Figure 5.8 plots the performance of the mark evaluation task in the 10 runs.

The objective of the third task, blocking, is to position a player between the ball and a candidate receiver. In this task, the player senses the distance, direction and heading of two opponents: the one who has the ball and the opponent closest to the player. The player's action choices are then to turn or to dash. The fitness function is determined by how long the player is in a blocking path between the ball kicker and the other opponent. For example, if the player positioned himself between the ball kicker and the other player[2] all the time he gets a fitness of 100%. Figure 5.9 shows the performance for blocking in 10 runs.

Clearing the ball

Once a defender intercepts the ball, he will usually clear the ball from the defense zone of the field. This reduces the probably of the attackers scoring a goal. In this task, the learning player is pitted against three opponents and three teammates. The player can sense the distance, direction and heading of all the players. A player need not sense the ball, since clearing the ball will be executed only once the player has control over the ball. The player's action choices are to turn or to kick the ball. The fitness function is calculated based on the outcome of the kick. The closer the kick angle towards a teammate, the higher the payoff assigned. If the kick is intercepted by a teammate, the player get at least 85% of the fitness. The player gets penalized 70% if the ball is intercepted by an opponent, and 40% if the ball is kicked out of bounds.

[1] However, in future work, a more sophisticated criteria shall be used.

[2] This is calculated by a parallelogram created by three points: ball position, opponent's position, and a point distanced x meters from the center line of the previous two points, where x is the distance such that a player can intercept the ball if a passer kicked the ball at maximum speed towards the receiver.

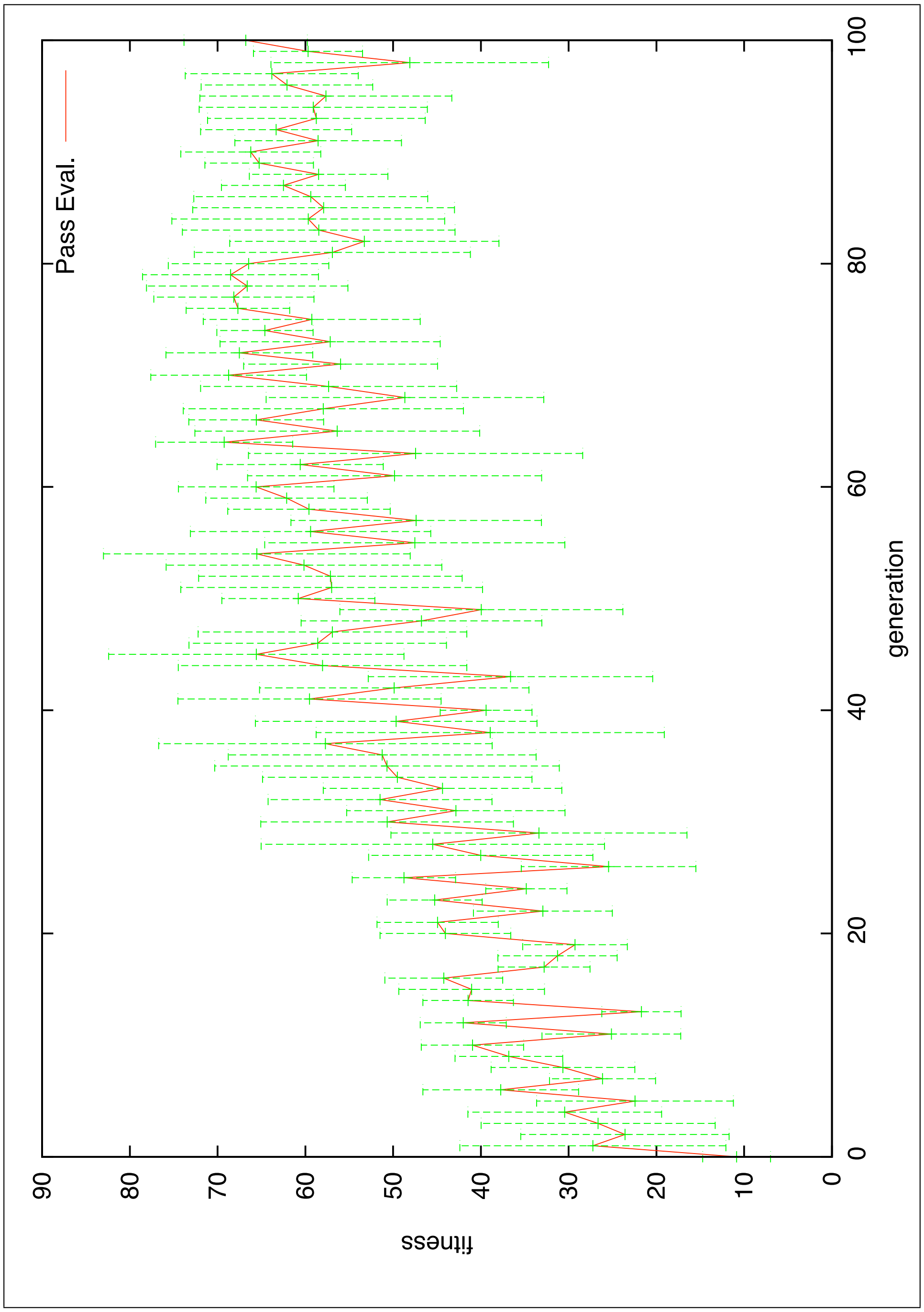


Figure 5.7: Performance of pass evaluation

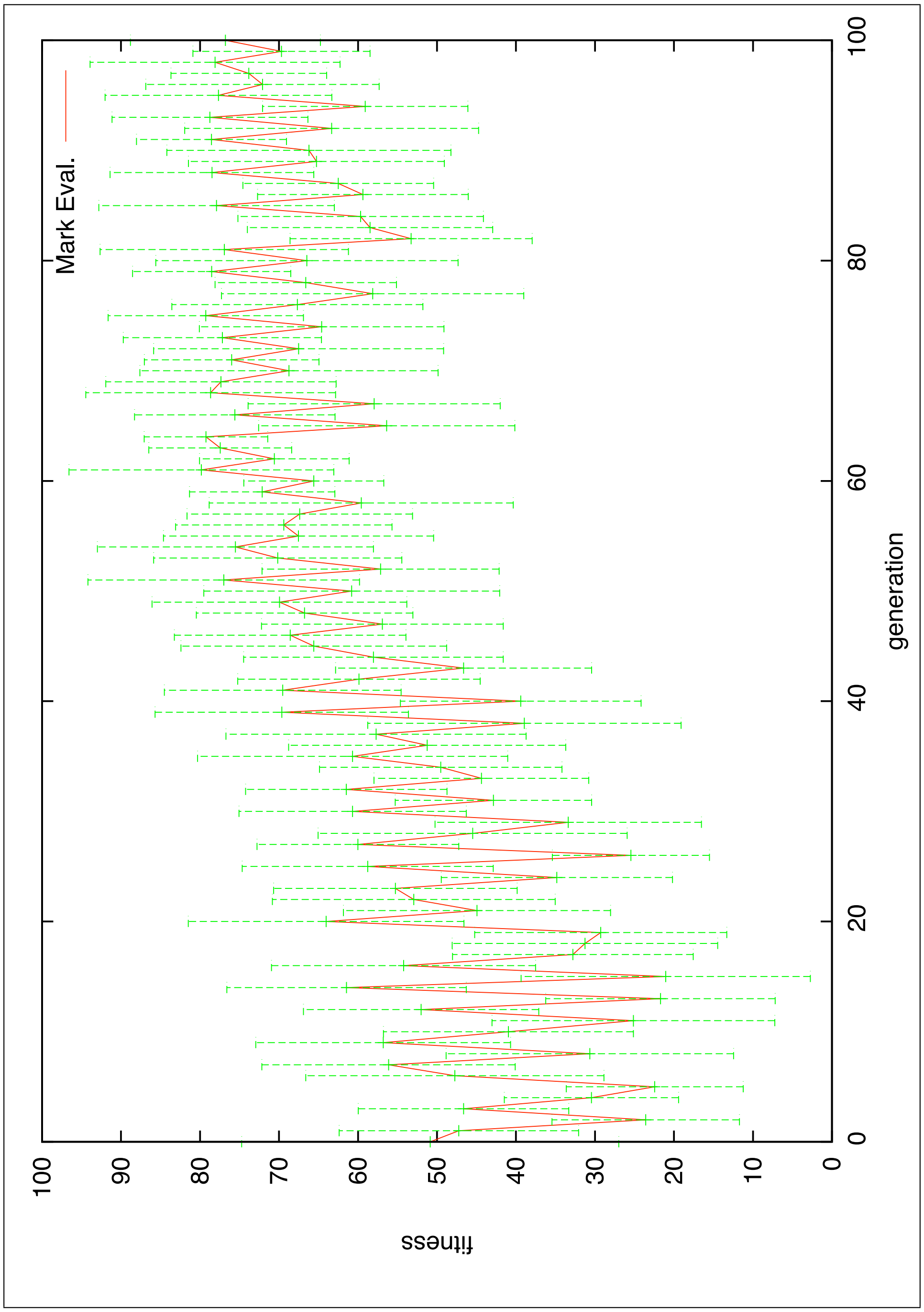


Figure 5.8: Performance of mark evaluation

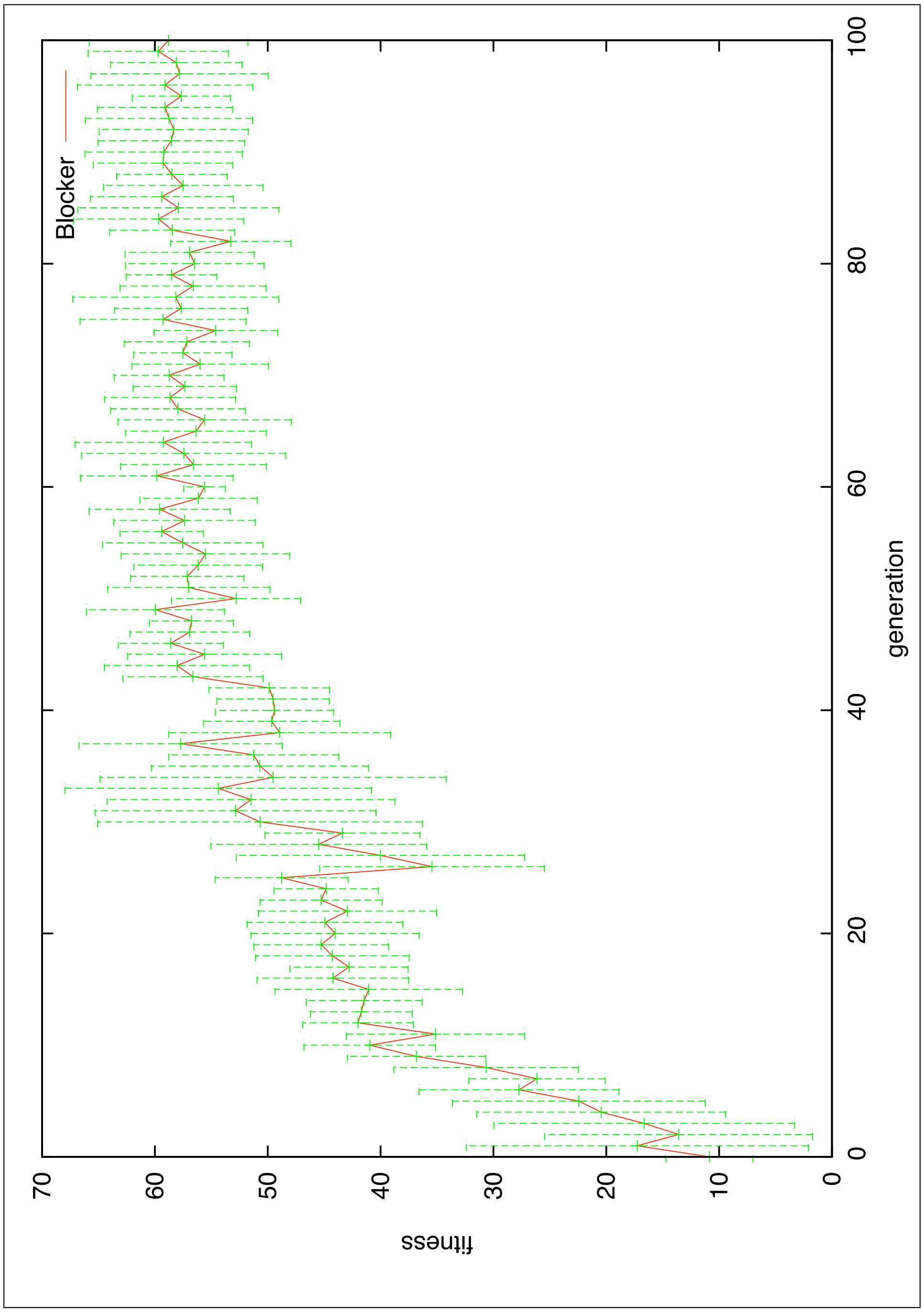


Figure 5.9: Performance of blocking

The performance for clearing the ball is illustrated in Figurer 5.10. The graph shows that the performance stabilizes in the 60% vicinity after 60 generations.

The clearing the ball task is similar to passing the ball, but without needing the ball to be passed to a particular teammate. As this skill depends highly on the performance of the other players in the field, it is learned after the other skills have been learned in this section. The goalie utilizes this skill as well after catching the ball.

5.3 MINI-GAMES

At this stage, agents are ready to learn a short soccer game. Two groups of agents are coevolved synchronously, viz. attackers and defenders. The aim is to allow agents in one group to build an attack in order to score, and agents in the other group to use their defensive skills to counter that attack. This mini-game ends when the attack ends in one of the following situations: a goal is scored, a goalie catches the ball, the defenders gain control of the ball and move it from the danger zone[1], the ball is kicked out of bounds, or the game times out.

The experiment is set up as follows. Three players sharing an attack behavior are pitted against three other players that share a defense behavior. A goalie that uses the previously learned goal tending skill is used as a player on the defender team. All of the players are placed in one half of the field, with the defenders being closer to their goal. Rather than using low-level sensors from the soccer simulator, the attacker and defender use some high-level sensors. Attackers can sense the distance, direction and heading of the following: the ball, the closest teammate, and the closest opponent. The closest teammate and opponents of an agent are calculated based on the distances of all the players the agent can sense. With the field being divided equally into eight rectangular regions, an attacker agent can sense its location in the field as well in the form of a number between 1 and 8 . Defenders can sense the distance, direction and heading of the closest teammate and opponent. Another sensor is also introduced to a defender to test whether he is the closest defender to the ball or not. These high-level sensors help attackers and defenders to better choose the best action they

[1] The danger zone for defenders is their own half of the field.

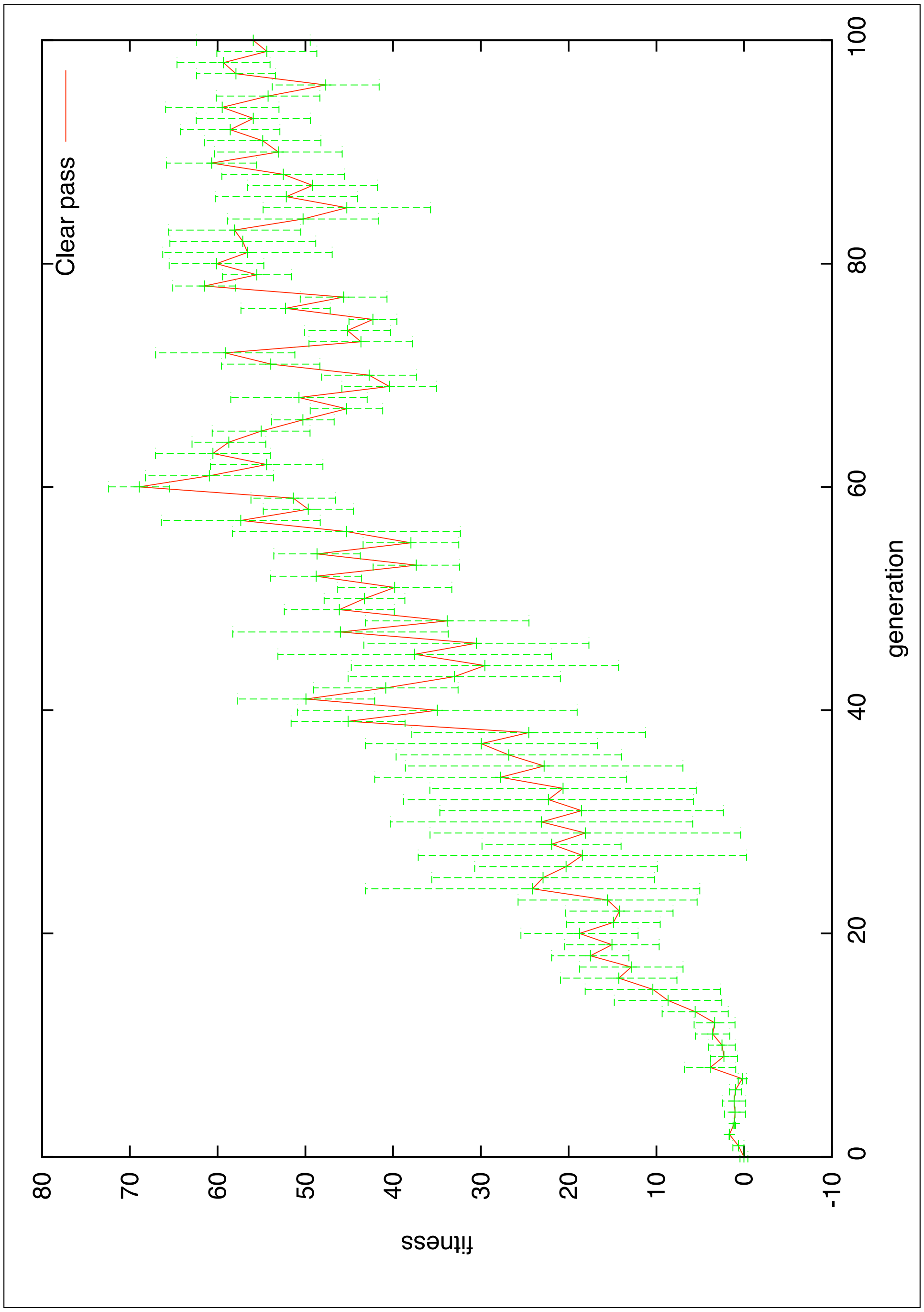


Figure 5.10: Performance of ball clearing

Table 5.1: Attacker and defender fitness

Ending criteria	Attacker	Defender
Ball is cleared	0%	100%
Goal is scored	100%	0%
Ball is catched	60%	30%
Ball is kicked out of bound by a defender	50%	40%
Ball is kicked out of bound by an attacker	30%	80%
Timeout	20%	50%

should take in any situation.

The action of both attackers and defenders is to choose a task from the previously learned skills. An attacker must choose to dribble, shoot, pass the ball or get open for a pass. On the other hand, a defender must choose whether to intercept the ball, mark an opponent, or block a possible pass path. After choosing an action to perform, the designated task is executed until that task ends, or a previously determined time elapses. Then the next cycle of the step is executed the same way until an ending situation of the mini-game is reached.

The fitness function is determined by how the mini-game ends. A constant value is assigned for each ending situation for both attackers and defenders. In the current experiments, the values used are shown in Table 5.1.

The performance for attackers and defenders are shown in Figure 5.11 and Figure 5.12, respectively. It can be noticed that in both figures, the standard error varies. This suggests that the population has a huge deviation. It also may suggest that the tasks were difficult to learn. Looking at the graphical behaviors of some runs, it shows that the attackers learn some attacking behaviors, and they indeed were able to score some goals during the ten episodes used during test runs. The defenders behavior showed some drawbacks, though. It was difficult for the defenders to intercept the ball before the attackers shot the ball towards the goal. The blocking skill was not used as it should have been, resulting in more successful passes for the attackers. Nevertheless, the goalie was able to catch many goals, which resulted in few goals scored during the test runs.

It is not clear whether the performance of the defend or attack skills can be enhanced by evolving for more than 100 generations. These experiments take a long time to finish, so exploring behaviors for more generations is not being considered at the time being. However, if time and better machines were available, it would be interesting to see what behavior results might be obtained for a longer evolution.

5.4 SCALING UP TO FULL GAMES

The previous experiments show learning of soccer skills in limited and artificial scenarios which do not reflect the full range of game situations. In this section, the extension of the learned behaviors into a full multi-agent behavior that is capable of controlling players throughout the entire game of soccer is discussed.

The learned skills are used when the player is in the vicinity of the ball; however, players also need to act when the ball is far away. In a professional soccer game, when not near the ball, a soccer player moves to a location in the field that is determined by a game plan. This plan is created by the team's coach. Although coaching is an interesting skill for machine learning, learning it is beyond the scope of this research.

Another aspect of the game that needs to be considered is when to execute attacking or defending skills. Below a description of how a team of learned agents is built to play a full game of soccer is provided.

Formation, and strategic positioning

Formation, typically described as three numbers, determines the game plan set by a soccer coach. For example a formation 3-4-3 denotes 3 defenders, 4 midfielders, and 3 attackers. A coach can change the formation during the game by instructing players to reposition themselves in the field. Since coaching is not considered as a learning task in this research, a fixed formation is used during the game. A user-defined set of positions determines where

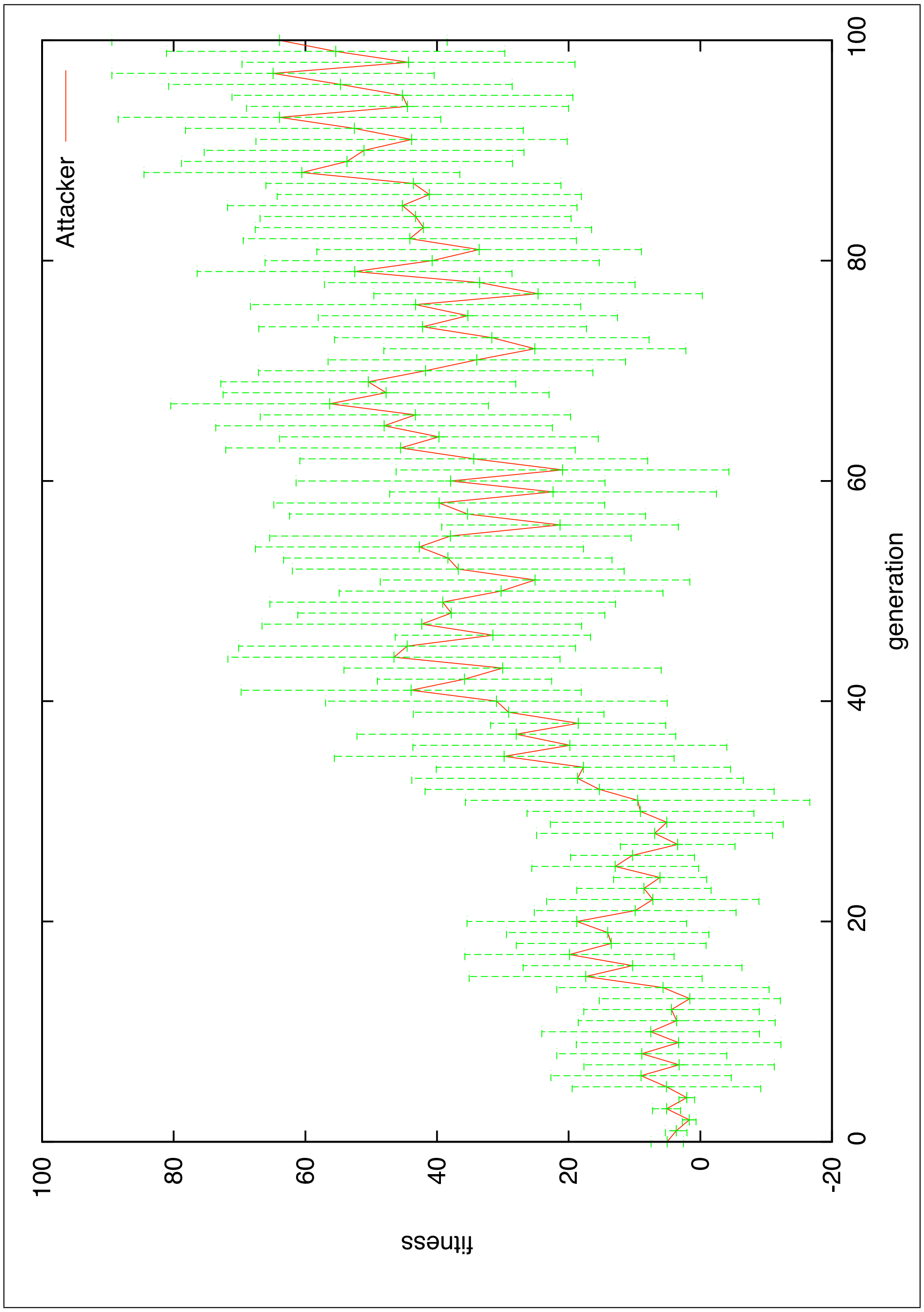


Figure 5.11: Performance of attacker

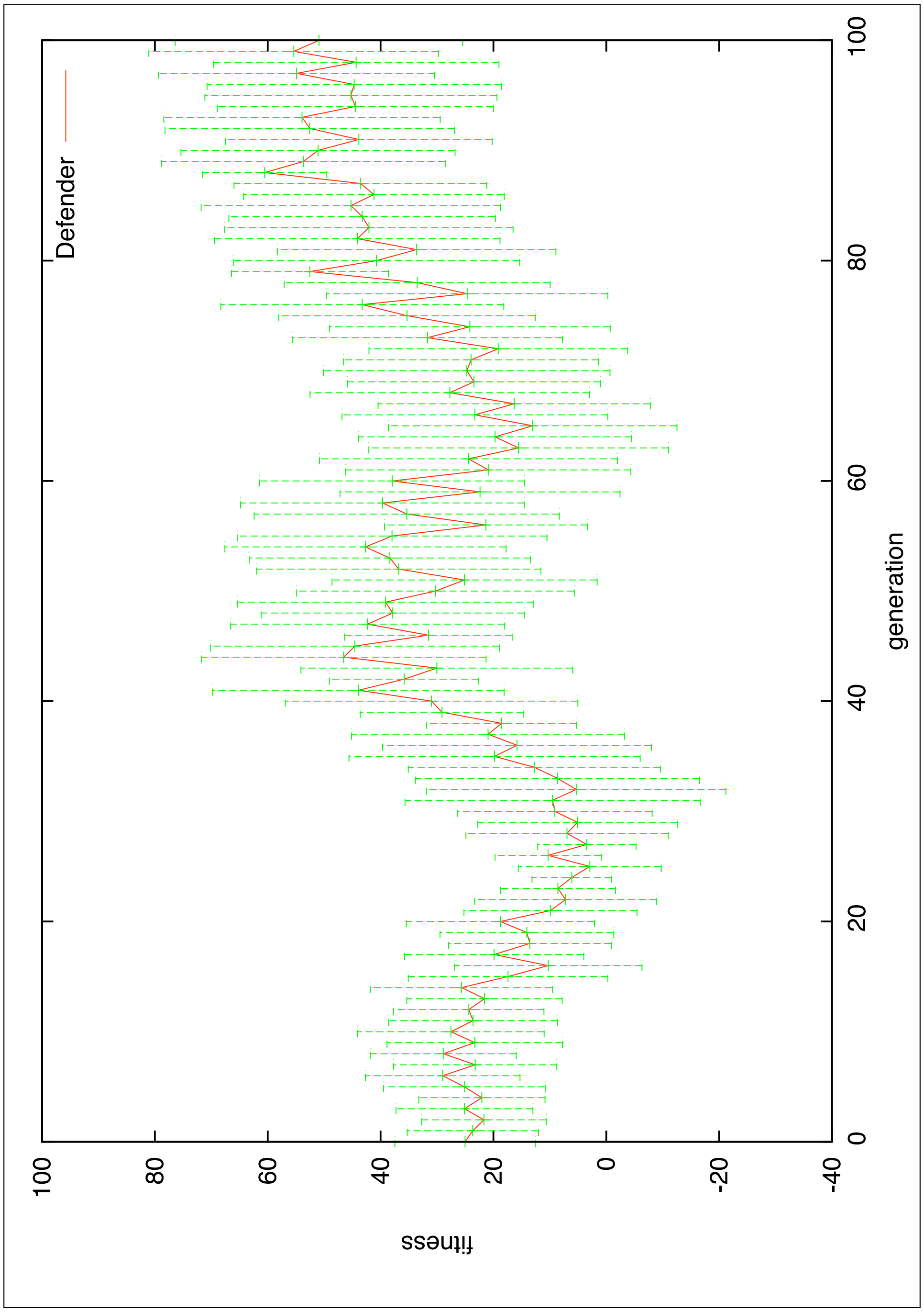


Figure 5.12: Performance of defender

the player should position himself in the event the ball is far from him.

The goal in this experiment is to dash towards a target, which is the strategic position set by the user. The strategic positioning skill is learned in a way similar to dashing towards the ball. The agent senses the direction to its strategic position and its actions are to turn at an angle or dash with a fixed dash power. The dash power is fixed to avoid stamina degradation during the game. The fitness of the agent is calculated based on how fast the agent reaches his target position, in the same fashion as calculating the fitness for the ball intercepting skill.

Figure 5.13 shows the performance for the strategic positioning skill. As the learning curve shows, this skill is easily learned. The best performance of 95% is reached after only 28 generations.

A soccer player

In order to play a full soccer game, two hand-coded agents were introduced. The first one is a midfielder agent that determines whether a player should attack or defend. A midfielder agent can sense which team has possession of the ball. If the agent's team has the ball, then it executes an attacking skill; otherwise a defending skill is executed.

The second hand-coded agent is a "player" agent that decides which of the top-level actions to execute. The player agent senses the jersey number of the robot that it represents in the soccer simulator, and the ball's distance. It chooses an action from the following: execute a strategic positioning skill or act as a goalie, a midfielder, an attacker, or a defender. The goalie has a unique jersey number determining his role in the game. Each jersey number determines the strategic position that is passed to the strategic positioning skill. Depending on the formation used, each jersey number has a specific role. For example, when using a 4-4-2 formation, players who have jersey numbers from 2 to 5 are defenders. Players who have jersey numbers between 6-9 are midfielders. Players with jersey numbers 10 and 11 are considered attackers. In the event that the player agent senses the ball to be far away, it executes a strategic positioning skill instead of the other roles. The reader may note that both of these hand-coded agents can be learned agents instead.

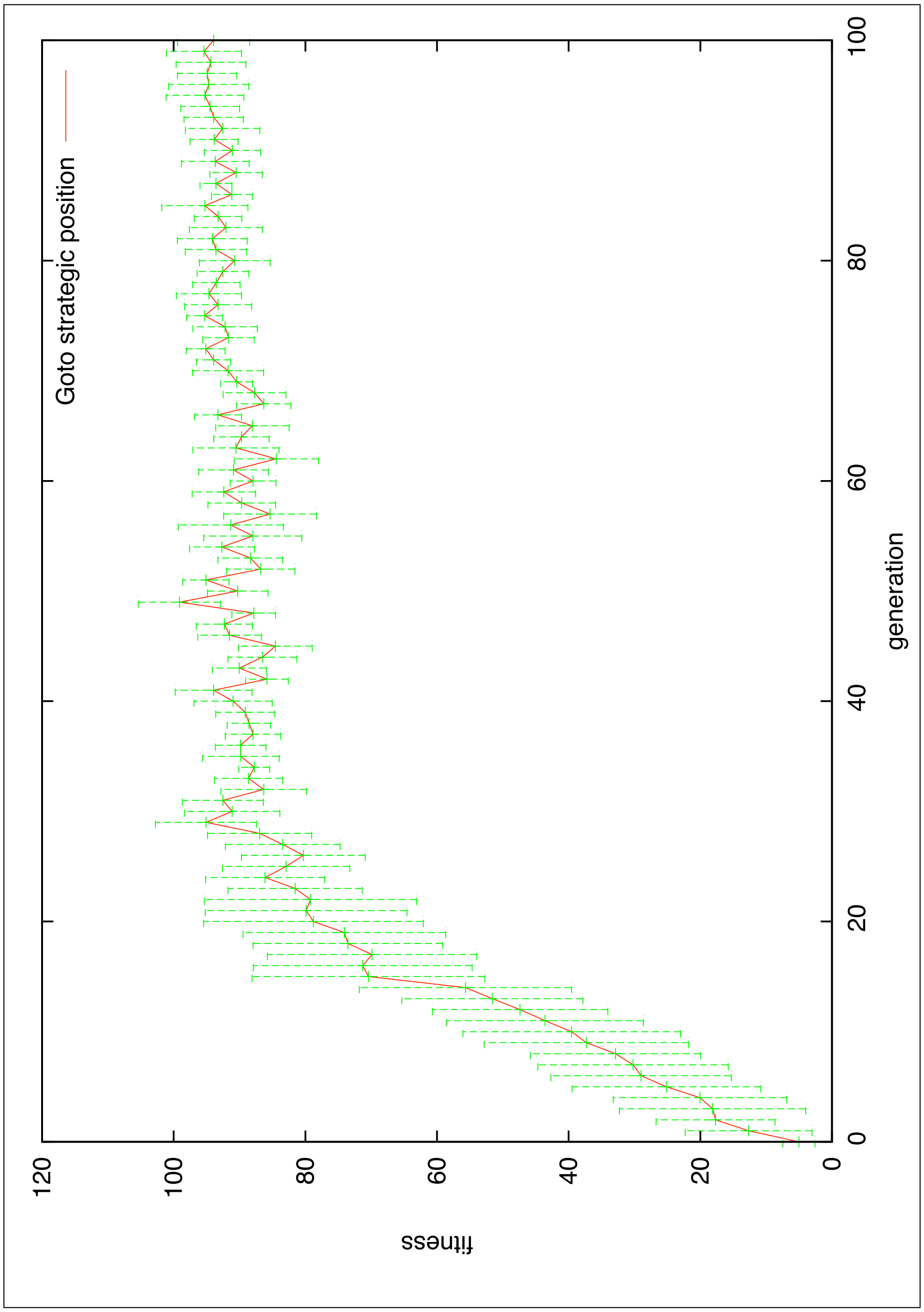


Figure 5.13: Performance of strategic positioning

5.5 DISCUSSION

Many tasks are discussed in this chapter, each of which has a different learning rate. Some tasks are learned faster than others. For example, the goalie task is learned faster than the shooter task. Coevolving a goalie with a shooter asynchronically may result in a different performance than the one shown in Section 5.1. This suggests that controlling the rate of learning is an important factor when coevolving many tasks together.

In Section 5.4, scaling the learned tasks to a full game was discussed. However, performance was not illustrated. In the next chapter, the performance of the coevolved team against other RoboCup teams is provided. The coevolved team is pitted against three RoboCup teams, and its performance and robustness are assessed.

In this chapter, many learning tasks are used to construct a team of agents that can play a full game of soccer. Some of these tasks are coevolved together at the same time, while others are learned separately. It is important to take the task's dependency into consideration during learning since their learning affects each other. The order in which the learning is carried out in this chapter and the number of generations used for each experiment were chosen because of prior knowledge of the game of soccer and the dependency graph illustrated in Figure 3.4. However, determining a different order or evolving through a different number of generations may result in a different performance. Hence, the organizing of the overall learning process is investigated in the next chapter.

6.0 ORCHESTRATION

The previous chapter illustrated how task decomposition is used to accomplish a learning task in a complex problem. This chapter provides a description of a mechanism to manage the computing resources in the learning of the variety of tasks which are needed to achieve the learning of a soccer playing agent. The time needed to achieve an acceptable performance in learning each of the decomposed tasks is different, and hence, a prior knowledge of the difficultly of tasks is needed to better allocate time for learning them. In the previous chapter, most of the tasks were run for a fixed number of generations (100). In one case, learning converged well before reaching 100 generations. This suggests that performance shows no gains past a certain point and continuing learning after that point is a waste of resources. In another case, this fixed number of generations (100) is simply not enough to achieve an acceptable performance and so more generations are needed to accomplish it. Those are problems that were easily noticeable in the experiments' results from the previous chapter.

When learning more than one task using coevolution, the performance of one task affects the learning of the other tasks. For example, in Chapter Four, learning the shooting task using two goalies, which have different competency levels, gave a better performance than learning the shooting task using one competent goalie.

Therefore, a given prior knowledge of which tasks must be learned together and at which level is needed. In this chapter, a method that can be used to orchestrate the coevolutionary learning without having a prior knowledge about the expected interactions between the decomposed tasks that need to be learned is shown.

Scheduling resources can be done in two ways: statically and dynamically. In a static scheduling, pre-determined criteria are used to determine the scheduling. For example, in a round robin scheme, a fixed amount of resources is given to each task. In a dynamic scheduling, the performance level of tasks determines the scheduling scheme, such as having a criteria like “the best performed task gets the least amount of resources”. In this chapter, both types of scheduling are tested and described in Section 6.1. The results are then described in Section 6.2.

6.1 EXPERIMENTAL DESIGN

Dividing a complex task into smaller ones is a key factor in solving large problems. However, managing the time (and resources) among those subtasks is critical. In this study, a control mechanism was devised to orchestrate the learning of the subtasks which are described in Chapter 5. This mechanism is responsible for ordering the learning tasks, allocating a time slot for learning each task and revisiting a learned task again to achieve better performance due to a changing behavior in related tasks. This is similar to resource management in an anytime learning system[1].

Four policies are implemented for this orchestration. The first one is a simple round-robin policy, the second, a static hand-coded policy. The third is a dynamic time-based blame assignment policy, and the fourth is a performance-based blame assignment policy. In the first two policies, the performance of the tasks as they are learned is not used to determine how to orchestrate their learning. These static methods are used to set a baseline for comparison with the dynamic orchestration methods.

For the sake of a fair comparison, the total number of evaluations of all tasks is fixed. To set up the orchestration experiments, the learning of all 11 tasks is carried out the same way, in the same order as in Chapter 5. At the beginning, each individual task is learned for 30 generations and then halted. Then the learning for each task is continued according to each of the four policies until the total number of evaluations of all tasks reaches a preset number[2]. For the static orchestration methods, i.e. round robin and heuristic methods, each task is learned for an additional 100 generations after the initial 30 generations. For the

[1] Anytime learning is an approach to learning and adapting continuously in a changing environment. For more details, the reader may refer to (Grefenstette and Ramsey 1992) and (Parker and Mills 1999).

[2] The intercepting a ball, moving to strategic position, and clearing a ball tasks are learned prior to and are not part of the orchestration experiments.

dynamic orchestration methods, the learning is divided into time-slots of size 100 generations and all the tasks share 100 generations in each time-slot.

6.1.1 Round Robin scheduling

In round robin scheduling, each subtask is evolved for 10 generations and then suspended. The order of execution is similar to the order adopted in Chapter 5, which is as follows: shoot, goalie catch, dribble, mark, pass, get open, pass evaluation, mark evaluation, block, attack, and defend. The same experiment setup for all the subtasks in Chapter 5 is applied here. The reader may note that some of the tasks which are coevolved synchronously in the previous chapter are now coevolved asynchronously in a round robin fashion. Consequently, each subtask's learning process is highly dependent on the others' performance.

Round robin scheduling is the simplest form of managing evolving multi-tasks, and, as mentioned earlier, used as a baseline for comparison with the other methods which are implemented in this chapter. One important aspect of round robin scheduling is to choose the amount of time that should be allocated for each task (token size). In this experiment, ten generations is used. A small token will elicit a similar behavior to coevolving subtasks synchronously. Choosing a large token would result in different behaviors depending on the order of task execution, and most probably lead to a poor performance since the subtasks are highly dependent on each other. Considering the maximum allowed evaluations, with each subtask being evolved for 100 generations cumulatively, a token size of ten generations is a reasonable choice to make. Choosing various token sizes has not been considered in this suite of experiments. That is one future direction that one might pursue.

6.1.2 Heuristics based scheduling

Orchestration has two aspects: time and order. The latter is the focus of this method. All the subtasks are evolved one after another for 100 generations. Some subtasks are coevolved together synchronously, while others are evolved separately. Basically, the evolution proceeds in the same way, in the same order, as was applied in Chapter 5. The order used is based on our knowledge of the game of soccer. The first heuristic used is the sub-division of our goal task into subtasks in a hierarchical tree as seen in Figure 3.4. The order of evolving these subtasks always moves in the direction from the bottom to the top of the tree. That is to say, a subtask on a lower level of the tree is always executed before a task in a higher level of the tree.

The second heuristic used is to coevolve some subtasks together at the same time. In real world soccer, teaching soccer is done through a series of training sessions. These sessions are conducted to allow players to comprehend one or more aspects of the soccer game. Utilizing the knowledge of a professional soccer coach and some books on coaching real soccer(van Lingen 1997) a setup of experiments is implemented to learn some of the tasks together at the same time. For example, a passer is learned together with a blocker.

6.1.3 Frequency blame assignment

The two methods, described above use a static policy to manage the evolution of the subtasks. In this section, orchestration is based on a dynamic policy. That is, deciding which subtask is evolved at a given time is based on ongoing learning performance. In order to dynamically assign which task is active at a given time, a blame assignment algorithm is utilized. This algorithm decides which particular task should be given precedence in learning over other tasks.

In a RoboCup soccer game, an agent may execute one subtask at a time. Therefore, the amount of time that each subtask takes in a soccer game provides great insight into the cause of the game's results. For example, if a team that spends most of its time defending loses the game, this suggests that the team failed to gain control of the ball most of the time and hence had weak defensive skills. Frequency blame assignment then, uses the time that each subtask takes in a soccer game to decide the future assignment of time each subtask should spend in learning.

Since the frequency blame assignment algorithm makes use of the percentage of time each subtask uses in a game of soccer, a game of RoboCup soccer[1] is played between a team of players which are based on the subtask's learned-so-far rulebases and a team from the RoboCup

[1] Due to time constraints, one half of a game is used instead of the usual two halves.

community. The UvA Trilearn[1] team is a competitive previous RoboCup champion, and a well documented team[2]; hence, it is used in the blame assignment implementation.

To manage the learning of all the tasks, the total amount of time allocated for learning is divided into time-slots of an equal size. In each time-slot, the blame assignment algorithm is triggered to decide how long each subtask should be learned for that given time-slot by playing a soccer game with UvA Trilearn team. To fairly compare this method to the other static orchestration methods and to compensate for the time used in playing a soccer game between time-slots, the total number of generations allowed is set to 1000[3]. Since 1000 generations was chosen for learning all the subtasks, a time-slot of size 100 is a reasonable choice. This means, the blame assignment algorithm is executed 10 times during the learning process. Furthermore, all the subtasks share 100 generations in each time-slot, and the blame assignment algorithm determines the percentage of generations each subtask should be allocated out of the 100 generations. The order of the sub-task's execution is kept the same as that used in the heuristic based method.

As mentioned in Chapter 3, a modified version of the soccer simulator that does not use UDP communication between the players and the soccer simulator is used in these learning experiments to speed up the learning process. Instead, a direct procedure call to get the sensors' information and to execute the actions is used. However, in order to play a soccer game with other teams from the RoboCup community, a communication layer has to be implemented. This adds another overhead of communication, and creates issues related to synchronization, as well as causes a difference in the team's behaviors, i.e., slower execution between the version used in learning and the one used in blame assignment. Also, the version with the communication layer exhibited differences in implementation of some high-level sensors such as sensing the agent's location in the goto-strategic-position skill. Nonetheless, all the experimental results shown in section 6.2 are based on the version with the communication layer.

In managing subtasks' execution, frequency blame assignment can be used in two ways.

[1] A 2003 World RoboCup champion developed at the University of Amsterdam.

[2] The reader may refer to (de Boer and Kok 2002) for a full thesis.

[3] In the static orchestration methods, the total number of generations allowed after the initial setup is set to 1100 ($11 tasks \times 100 generations$).

First, tasks that have high frequency during the played game can be given more learning time than those with low frequency. Second, tasks that have low frequency during the game can be given more learning time than the high frequency ones. Each of these methods has its advantages. The argument that "blame the tasks which were executed most" favors use of the first. The argument of "enhance the least used tasks so they can be used more" favors use of the second. Both cases were implemented in this research, and their results are provided in section 6.2.

6.1.4 Performance metrics blame assignment

While, the score of a soccer game can be used to asses the performance of a team, it fails to provide comprehensive information which one might use to enhance the performance of the team. Much information that can be gathered during the game can be used to analyze the performance of the various tasks involved in playing the game. For example, the percentage of successful passes provides good information on how the passing skills performed in a particular game.

The merit of orchestration is the ability to dynamically manage the learning process as an online learning algorithm. The aim here is to orchestrate the learning in order to maximize the overall performance of the learning agents; therefore, using the fitness function as a performance measure won't provide us with an accurate figure of real performance in a full game of soccer.

In the performance metrics blame assignment method, a set of metrics is used to decide how the subtasks should be executed. The implementation of this method is similar to frequency blame assignment; however, instead of using task frequency to assign the percentage of learning time each subtask should have in the future time-slot, a set of metrics collected during a soccer game is used.

Tables 6.1 summarizes the performance metrics which are collected during the game. In the performance metric blame assignment, each subtask uses a subset of these metrics in calculating the percentage of learning time in a future time-slot. Each subtask uses a utility function defined as follows:

Table 6.1: A list of performance metrics used in the blame assignment

Metric	Description
goals	Number of goals scored
shoots	Number of shoots kicked
dribble dist.	Overall traveled distance with ball in control
catches	Number of goalie catches
goal kicks	Number of goal kicks awarded
ball possession	Percentage of the ball possession
passes	Number of successful passes
pass misses	Number of intercepted passes
lost ball	Number of ball interceptions
forward passes	Accumulated distance of passes towards opponent goal

$$f = \sum_{i=0}^{n} w_i \times m_i,$$

where m_i is a value which determines the blame assignment of a particular metric for the given subtask, and w_i is a weight associated with each metric m_i. Both values are in the $[0, 1]$ interval.

Each performance metric contributes in a different way to the blame assignment in each subtask. For example, when assigning a blame to the attacking task using the goal-scored and pass-success metrics, the weight associated with the goal-scored metric should be higher than the pass-success metric since the first contributes more to the attacking skill than the later. Therefore, the utility function assigns a weighted sum of metrics for each of the subtasks.

Since there is no clear criteria to determine the exact value for each weight associated with each metric, weights will be fixed for all the subtasks as given in following equation:

$$w_i = 0.5 \times w_{i-1},$$

Table 6.2: Subtasks performance metrics assignment

Subtask	m_0	m_1	m_2	m_3
Shoot	goals	catches	goal kicks	shoots
Catch	catches	goals	shoots	goal kicks
Pass	passes	pass misses	forward passes	ball possession
Pass eval.	passes	pass misses	ball possession	goals
Dribble	dribble dist.	lost ball	ball possession	shoots
Get open	passes	pass misses	ball possession	forward passes
Block	passes	pass misses	ball possession	goals
Mark	ball possession	passes misses	lost ball	shoots
Mark eval.	ball possession	passes misses	lost ball	shoots
Defend	goals	shoots	ball possession	goal kicks
Attack	goals	shoots	dribble dist.	passes

where w_0 is set to 0.5. Now the metric m_i has to be ordered according to its contribution to the blame assignment. To simplify the calculations, each subtask utilizes exactly four metrics. Table 6.2 shows each subtask with the associated ordered performance metrics used in this experiment.

After calculating all the utility functions associated with each subtask, these functions are normalized to determine the percentage of time each subtask should be executed in the next time-slot.

6.2 RESULTS

In order to evaluate the different orchestration schemes used, a set of three Robocup teams is used. The learning process was repeated in three runs. The resulting set of rulebases at each run for each orchestration mechanism is used to play 10 games against each of those three teams, and the final score is reported. The duration of each game is set to 3600 simulation cycles, which corresponds to half of a game usually conducted during the official world Robocup competitions.

The teams used are: UvA Trilearn, FC Portugal, and BrainStormer. FC Portugal is a 2000 World RoboCup champion developed by a team from the University of Porto, and the University of Aveiro, Portugal. BrainStormer is a 2007 and 2008 World champion developed at the University of Osnabruck, Germany. Both of UvA Trilearn and FC Portugal are hand-coded teams, while BrainStormer utilized Reinforcement Learning methods, such as Q-Learning and Temporal Difference, in the development of the team[1].

Table 6.3 shows the performance of each orchestration method used in the thirty games played with the three RoboCup teams for each run. The first two rows provide the number of goals conceded and scored, respectively. The third row illustrates the goal difference ($goals\ scored - goals\ conceded$). The next three rows give the total number of games won, withdrawn, and lost by the evolved team in the thirty games. The last row give the total number of points that are usually assigned in a soccer game. A quick look at the number of points for each orchestration methods illustrates the superiority of the performance metrics blame assignment method over the other methods used. The Round Robin method showed the worst performance.

A t-test is used to test the difference in performance (based on goal difference) of the orchestration methods in all the games played[2]. The tests show there is a significant difference between the performance metrics blame assignment and the other methods, with more than 97% confidence interval. The tests also show a significant difference between the heuristic based method and both round robin (with 99% confidence interval) and low frequency blame assignment (with a 98% confidence interval). A difference between the Hi and Lo frequency blame assignment was not found with 90% confidence interval. The test failed to show any difference between the round robin method and low frequency blame assignment with even as low as 83% confidence intervals, or any difference between the heuristic based and Hi frequency blame assignment methods with 89% confidence interval. From these statistical tests, we can deduce that with a 90% confidence interval, the only methods that were the same

[1] Not all of the team's skills were learned, however, BrainStormer's developers have a long-term goal to release a team that obtains its behavior by entirely employing a Reinforcement Learning methodology.

[2] Ninety games in total (3 runs × 30 games).

in performance are Lo frequency blame assignment and round robin, Hi and Lo frequency blame assignment, and heuristics based and Hi frequency blame assignment methods.

Tables 6.4, 6.5, and 6.6 present the performance of the orchestration methods against each one of the RoboCup teams.

6.3 DISCUSSION

The results in the previous section show distinct performance differences among the four orchestration methods used. The round robin gives the least favorable performance, while the performance metrics blame assignment demonstrates a significant advantage over the other methods. This suggests there is a great effect on performance in the way the subtasks are coevolved together. It is obvious as well that learning these subtasks varies in difficulty. It is also evident that they are dependent on each other in such a way that learning one subtask affects the learning of the others.

In round robin scheduling, all the subtasks have an equal share of the learning time. However, from the experimental results, we can deduce that some tasks require more time than others. Furthermore, the coevolution of the subtasks is synchronized within 10 generations. Since each subtask has its own learning rate, such synchronization would seem to hurt the overall performance. Such an assertion has not been proved. However, a synchronized coevolving of all the tasks in an experiment maybe needed to further examine the effect of synchronization on overall performance.

The closest performance to the metric blame assignment method is the heuristic orchestration. In the heuristic method, learning is carried out in stages. At each stage, subtasks are learned after the learning of all the subtasks in the previous stage are learned. The metric blame assignment allows the learning of the subtasks to interact and coevolve together regardless of the stages. Hence, it can be asserted that learning an individual task first and then using it to learn other tasks will not necessarily give the best overall performance. In many cases, the tasks need to co-adapt in order to achieve a high overall performance.

When comparing the performance of coevolution versus other RoboCup teams, as shown in Figures 6.4, 6.5 and 6.6, it can be noticed that the learned team was able to score goals

Table 6.3: Performance of Orchestration

	Round Robin				Freq. Hi				Freq. Lo				Heuristics				Perf. metrics			
Run	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ
Conceded	114	98	127	113	67	68	64	66.3	108	99	103	103.3	63	67	56	62	41	46	47	44.7
Scored	7	7	10	8	7	9	10	8.7	22	22	21	21.7	20	22	26	22.7	34	33	24	30.3
Goal diff.	-107	-91	-117	-105	-60	-59	-54	-57.7	-86	-77	-82	-81.7	-43	-45	-30	-39.3	-7	-13	-23	-14.3
Wins	0	0	0	0	3	2	4	3	4	3	5	4	2	3	6	3.7	10	9	7	8.7
Draws	2	2	1	1.7	3	4	5	4	2	4	1	2.3	8	5	6	6.3	5	5	7	5.7
Losses	28	28	29	28.3	24	24	21	23	24	23	24	23.7	20	22	18	20	15	16	16	15.7
Points	2	2	1	1.7	12	10	17	13	14	13	16	14.3	14	14	24	17.3	35	32	28	31.7

Table 6.4: Performance of Orchestration against UvA Trilearn

	Round Robin				Freq. Hi				Freq. Lo				Heuristics				Perf. metrics			
Run	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ
Conceded	19	17	23	19.7	8	10	6	8	16	15	14	15	10	13	13	12	4	8	10	7.3
Scored	3	3	6	4	6	8	9	7.7	15	13	14	14	8	10	17	11.7	18	16	16	16.7
Goal diff.	-16	-14	-17	-15.7	-2	-2	3	-0.3	-1	-2	0	-1	-2	-3	4	-0.3	14	8	6	9.3
Wins	0	0	0	0	3	2	4	3	4	3	5	4	2	3	5	3.3	8	8	7	7.7
Draws	2	2	1	1.7	3	4	5	4	2	3	0	1.7	5	2	3	3.3	0	0	1	0.3
Losses	8	8	9	8.3	4	4	1	3	4	4	5	4.3	3	5	2	3.3	2	2	2	2
Points	2	2	1	1.7	12	10	17	13	14	12	15	13.7	11	11	18	13.3	24	24	22	23.3

Table 6.5: Performance of Orchestration against FC Portugal

	Round Robin				Freq. Hi				Freq. Lo				Heuristics				Perf. metrics			
Run	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ
Conceded	36	31	39	35.3	30	28	28	28.7	44	41	35	40	23	20	17	20	15	17	15	15.7
Scored	4	4	4	4	1	1	1	1	4	6	4	4.7	9	9	7	8.3	12	11	4	9
Goal diff.	-32	-27	-35	-31.3	-29	-27	-27	-27.7	-40	-35	-31	-35.3	-14	-11	-10	-11.7	-3	-6	-11	-6.7
Wins	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0.3	2	1	0	1
Draws	0	0	0	0	0	0	0	0	0	0	0	0	3	3	2	2.7	4	4	4	4
Losses	10	10	10	10	10	10	10	10	10	10	10	10	7	7	7	7	4	5	6	5
Points	0	0	0	0	0	0	0	0	0	0	0	0	3	3	5	3.7	10	7	4	7

Table 6.6: Performance of Orchestration against Brainstormer

	Round Robin				Freq. Hi				Freq. Lo				Heuristics				Perf. metrics			
Run	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ	1	2	3	μ
Conceded	59	50	65	58	29	30	30	29.7	48	43	54	48.3	30	34	26	30	22	21	22	21.7
Scored	0	0	0	0	0	0	0	0	3	3	3	3	3	3	2	2.7	4	6	4	4.7
Goal diff.	-59	-50	-65	-58	-29	-30	-30	-29.7	-45	-40	-51	-45.3	-27	-31	-24	-27.3	-18	-15	-18	-17
Wins	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Draws	0	0	0	0	0	0	0	0	0	1	1	0.7	0	0	1	0.3	1	1	2	1.3
Losses	10	10	10	10	10	10	10	10	10	9	9	9.3	10	10	9	9.7	9	9	8	8.7
Points	0	0	0	0	0	0	0	0	0	1	1	0.7	0	0	1	0.3	1	1	2	1.3

against all the RoboCup teams. Although, the RoboCup teams which are used have different strength the learned team was able to play competitively. For example, the team learned in run No. 1 with the heuristic method was able to score 8, 9 and 3 goals against UvA Trilearn, FC Portugal and Brainstormer, respectively. It is obvious that the Brainstormer team is the best team among the others. It scored 90 goals in all the games played with the team learned with the heuristic method in all runs, while the other teams (UvA Trilearn and FC Portugal) scored only 96 goals combined. This suggests that the learned team still showed defensive skills when played against the RoboCup teams. The high score of the Brainstormer team shows that this team has better attacking skills than the other teams and not because of a bad defensive performance of the learned team. Indeed, this shows that coevolution can generate a robust team that demonstrates an acceptable performance when playing against different RoboCup teams.

It is hard to divide a big task into smaller ones which have equal learning difficulty. Thus, learning rate should be considered when learning these smaller tasks together. This chapter showed the effect of learning rate on coevolution. A performance metric blame assignment was used to control the learning rate in this research. This research suggests that further investigations of other ways of controlling the learning rate and its effect on coevolution is an essential research direction to divulge coevolution's dynamics and capabilities.

6.3.1 Blame assignment

In Section 6.2, the dynamic orchestration methods showed different performances. Although all of these methods utilized the same amount of time for learning and used the same fitness functions, they generated different soccer behaviors. In this subsection, the effect of blame assignment methods on learning is discussed.

Tables 6.7, 6.8, and 6.9 display the learning distribution for each subtask at each time-slot of the learning process of the first run for the performance metrics frequency Hi and frequency Lo blame assignment methods, respectively.[1] The columns of the tables give the number of generations allocated for each subtask. The first time-slot, (0), denotes the initial

[1] The learning distribution of the other runs gave a comparable results, therefore, only the learning distribution of the first run is reported in the tables of this subsection.

generations used in the experimental setup. The last column provides the total number of generations each subtask required in the entire evolutionary process.

Performance metrics blame assignment

Table 6.7 shows the learning distribution for the performance metrics blame assignment. The shooting task was assigned more time at the beginning of the learning process than at the end. The goalie task had a similar assignment. However, it was assigned more time than the shooter at the beginning of learning. Since the shooting task learning depends on the goalie task, assignments of the goalie time is important. The reader may notice that the goalie learning rate was reduced after time-slot 4, while the shooter gain more time around this time-slot. By allowing the goalie to learn faster, the shooter task was better able to enhance its performance as discussed in Chapter 4.

Another point worth noting in Table 6.7 is the dribbling task distribution. The performance metrics assigned more of the learning time to this task than any other task, and as the learning progressed, the assignment of the dribbling task got higher. Figure 6.5 shows that the average performance of dribbling performance in the three runs was still inadequate even after learning for more than 200 generations. This suggests that the dribbling task is hard to learn. The performance metrics still assigned a great deal of time for it to learn. Since dribbling performance was low, its performance metric gathered during the game advised that it needed more learning time. As the learning progressed, other tasks' performances were enhanced, and thus, less blame was assigned to them while more blame was assigned to the dribbling task.

The attacking and defending skills were assigned comparable times. Slightly more time was given for the defending task than for the attacking task at early time-slots, and the opposite was true for the last four time-slots. The defending task needed more time than the attacking task at the beginning of learning since the opponent is hand-coded team had better skills.

Frequency Hi blame assignment

Table 6.8 illustrates the learning distribution for the frequency Hi method. The goalie was assigned the highest number of generations. This suggests that a better goalie behavior was achieved than with the other blame assignment methods as shown in Figure 6.2. Since the goalie task is independent of any other task, the good performance of the goalie task under this method should be expected. However, the shooting task depends on the goalie task and hence, learning with a more competitive goalie makes the shooter task harder to learn, especially when the learning of the shooting task started with a highly skilled goalie.

Since the frequency blame assignment gives more learning time for the most executed task during the soccer game the defending skills (including marking and blocking) were given more time than any attacking skills. At the beginning of learning, the opponent team had more soccer skills than the learning team. Therefore, most of the time the learning team spent during the game was in defending. Blocking, marking, and defending tasks were all assigned more time than attacking skills. Figures 6.7, 6.8, 6.9, and 6.10 reveal that the frequency Hi method allows similar or better performances than the other methods.

For the attacking skills, the frequency Hi assigned less time at early time-slots, and provided more time at later time-slots. This suggests that the learned team was executing more attacking skills during the evaluation soccer game as the learning progressed.

Frequency Lo blame assignment

The learning distribution for the frequency Lo blame assignment method is provided in Table 6.9. As this table shows, the shooting task was assigned the highest number of generations, while the goalie task was assigned the lowest. Since this method assigned more learning time to the tasks that were least executed during the evaluation game, (the opposite of the frequency Hi blame assignment method), the goalie and the defending skills tasks were given less learning time than the attacking skills tasks. This provides a lower goalie performance in this method than in the others, as can be seen in Figure 6.2. The attacking task was given more learning time than any other tasks. This suggests that the learned team spent most of its time defending rather than attacking but that it was learning how to attack most of the time. Therefore, while the team, which was developed based on the frequency Lo method, developed more effective attacking skills than those in the frequency Hi method its defensive skills were under-learned. In the frequency Hi method, the defending task was given more time than the attacking one, but at later time-slots the method assigned less time to it. In the frequency Lo, the reverse was true, except that at later time-slots, the attacking task assignment was not reduced to the extent that the defending task in the frequency Hi method was reduced.

A comparison between the three dynamic blame assignments for each of the learning tasks is illustrated in Figure 6.1 through Figure 6.11. In Figure 6.1 the shooter task of the frequency Hi method scored lower than the other two methods, while the goalie task of the frequency Lo method scored lower than the goalie tasks in the other methods as shown in Figure 6.2. The goalie of the frequency Hi method showed better performance than the goalie of the performance metric method, since it was assigned a higher learning time. The same performance can be noticed in Figure 6.5 with the dribbling task of the performance metrics method.

In general, the frequency Hi method produced better performance in defending skills tasks than the frequency Lo method, while the opposite is true for attacking skills tasks. The performance metrics method has similar or better performances in both attacking skills (except passing) and defending skills, especially dribbling and marking tasks. The passing tasks for the frequency Lo were performed better than in the other methods, since it was assigned more learning time than passing tasks in the other methods (147 generations in run No. 1).

Figures 6.10 and 6.11 show the performances for attacking and defending tasks. Since these two tasks were coevolved together, their performances are highly biased by each other. Figure 6.10 shows that the performance of the defending tasks in the frequency Hi method is better than in the performance metrics method. This does not necessarily suggest that the frequency Hi generates better defending skill than the performance metric method, since the defending task evaluation depends on the attacking task performance. In Figure 6.11, the performance of the attacking task in the frequency Hi is the lowest among the other methods. This could mean that better performance of the defending task in Figure 6.10 is due to the lower performance of the attacking task as shown in Figure 6.11. The same behavior can be seen when comparing the attacking task of the frequency Lo with that of the performance metrics method. Since the performances of both the attacking and the defending tasks in the performance metrics method were high, this suggests that both of these tasks may

Table 6.7: Learning distribution of Performance metrics blame assignment (Run No. 1)

	0	1	2	3	4	5	6	7	8	9	10	Total
Shoot	30	10	9	13	15	13	17	15	8	6	5	141
Catch	30	12	13	14	7	5	2	2	5	4	3	97
Pass	30	7	9	7	7	7	8	9	7	6	5	102
Pass eval.	30	7	9	8	8	9	10	10	7	6	4	108
Dribble	30	10	10	11	13	14	13	16	20	31	49	217
Get open	30	7	9	7	11	10	9	9	8	7	5	112
Block	30	8	10	6	12	9	8	7	8	7	4	109
Mark	30	9	7	7	6	8	9	7	9	8	6	106
Mark eval.	30	9	7	7	6	8	9	7	9	8	6	106
Defend	30	10	8	12	8	9	8	8	9	9	7	118
Attack	30	11	9	8	7	8	7	10	10	8	6	114

have superior performance over the ones in the frequency based methods. Obviously, one cannot be certain of such a claim unless they can be pitted together (for example an attacker from the performance metrics method with a defender from the frequency Hi method). Yet, testing both tasks with another baseline task should provide an insight into which task was learned better. In Table 6.3, the results of games played with other RoboCup teams showed that the goal difference of the performance metrics method is less than those of the other two frequency based methods suggesting that the performance metrics methods has better attacking and defending and/or goalie skills than the other methods.

Table 6.8: Learning distribution of Frequency Hi blame assignment (Run No. 1)

	0	1	2	3	4	5	6	7	8	9	10	Total
Shoot	30	2	3	3	6	9	10	9	12	14	16	114
Catch	30	30	33	29	24	17	13	14	11	9	8	218
Pass	30	3	5	7	8	8	11	10	11	12	10	115
Pass eval.	30	3	5	7	8	8	11	10	11	12	10	115
Dribble	30	2	3	3	5	4	3	7	6	3	3	69
Get open	30	3	4	8	8	9	8	9	10	11	13	113
Block	30	13	10	7	6	8	7	7	6	6	5	105
Mark	30	12	10	11	9	9	8	7	6	6	7	115
Mark eval.	30	12	10	11	9	9	8	7	6	6	7	115
Defend	30	14	12	10	10	11	11	10	9	7	8	132
Attack	30	6	5	4	7	8	10	10	12	14	13	119

Table 6.9: Learning distribution of Frequency Lo blame assignment (Run No. 1)

	0	1	2	3	4	5	6	7	8	9	10	Total
Shoot	30	16	15	15	14	16	15	14	13	13	12	173
Catch	30	1	2	3	4	3	3	2	3	4	5	60
Pass	30	14	13	12	11	11	11	12	13	10	10	147
Pass eval.	30	14	13	12	11	11	11	12	13	10	10	147
Dribble	30	15	13	14	10	8	7	7	6	7	6	123
Get open	30	14	12	10	10	9	9	10	9	9	10	132
Block	30	1	3	4	5	4	5	6	7	7	8	80
Mark	30	2	3	4	6	6	7	7	7	8	8	88
Mark eval.	30	2	3	4	6	6	7	7	7	8	8	88
Defend	30	6	8	7	9	10	10	9	9	11	11	120
Attack	30	15	15	15	14	16	15	14	13	13	12	172

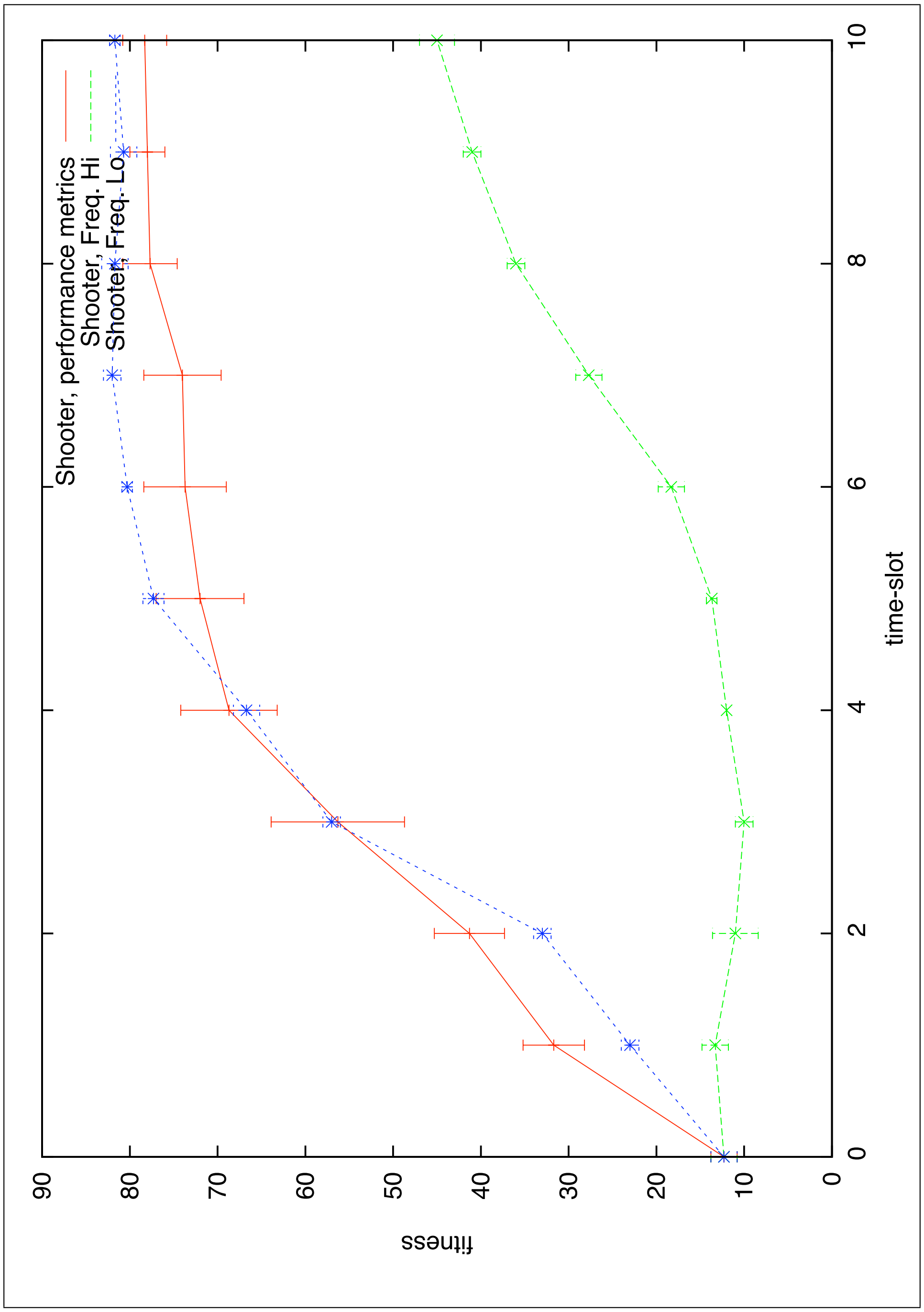


Figure 6.1: Blame assignment methods learning progress – Shoot

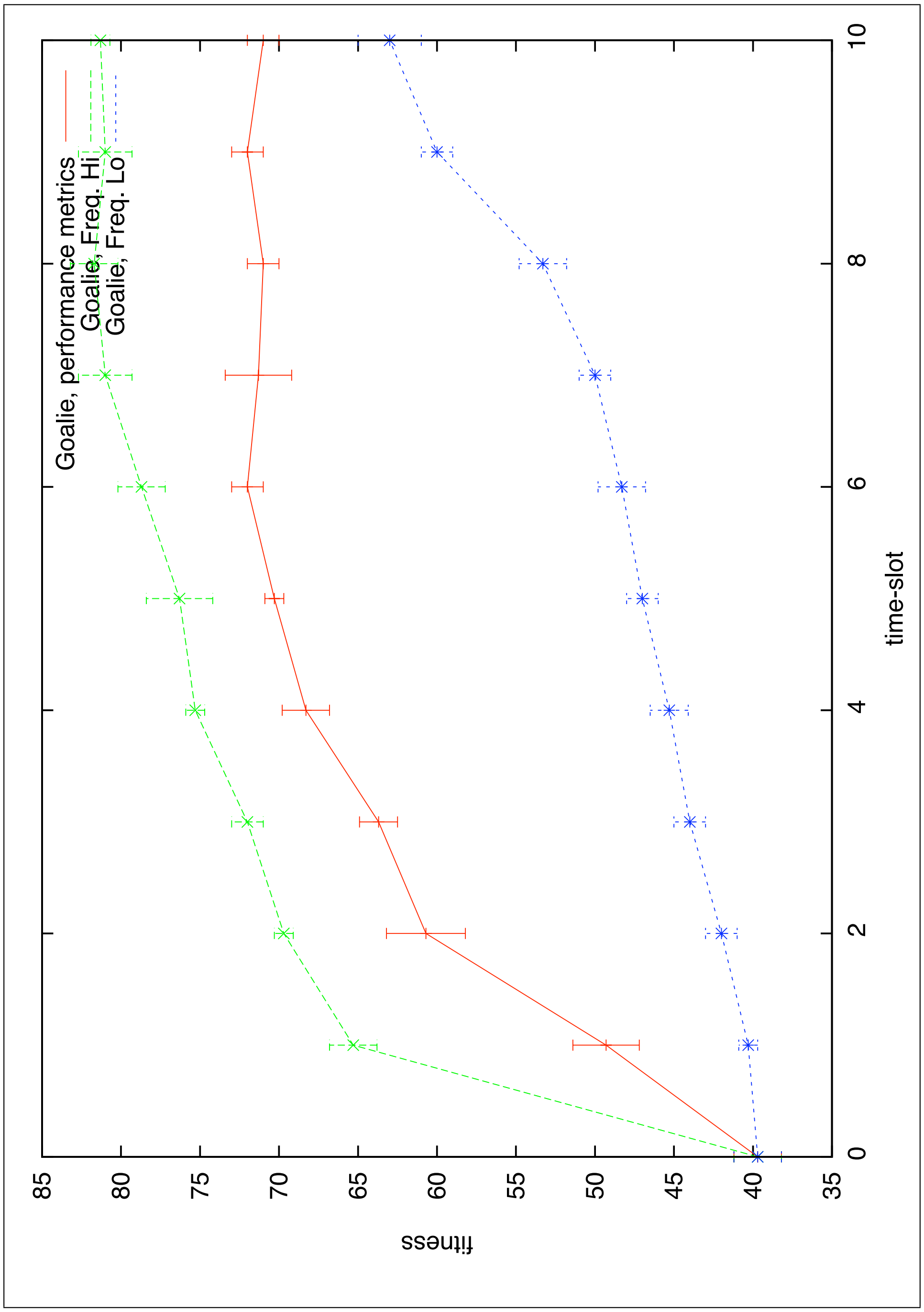


Figure 6.2: Blame assignment methods learning progress – Goalie catch

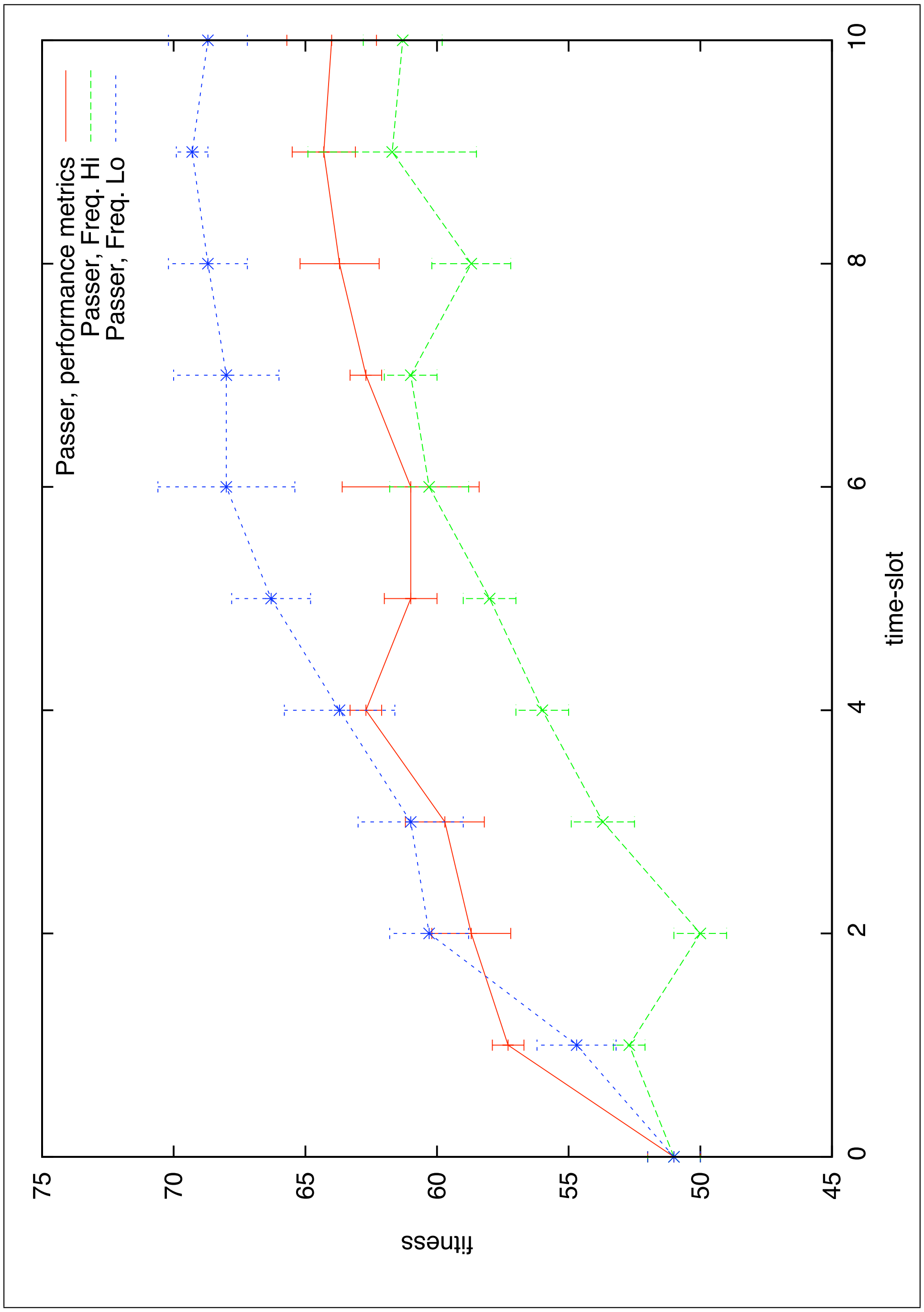


Figure 6.3: Blame assignment methods learning progress – Pass

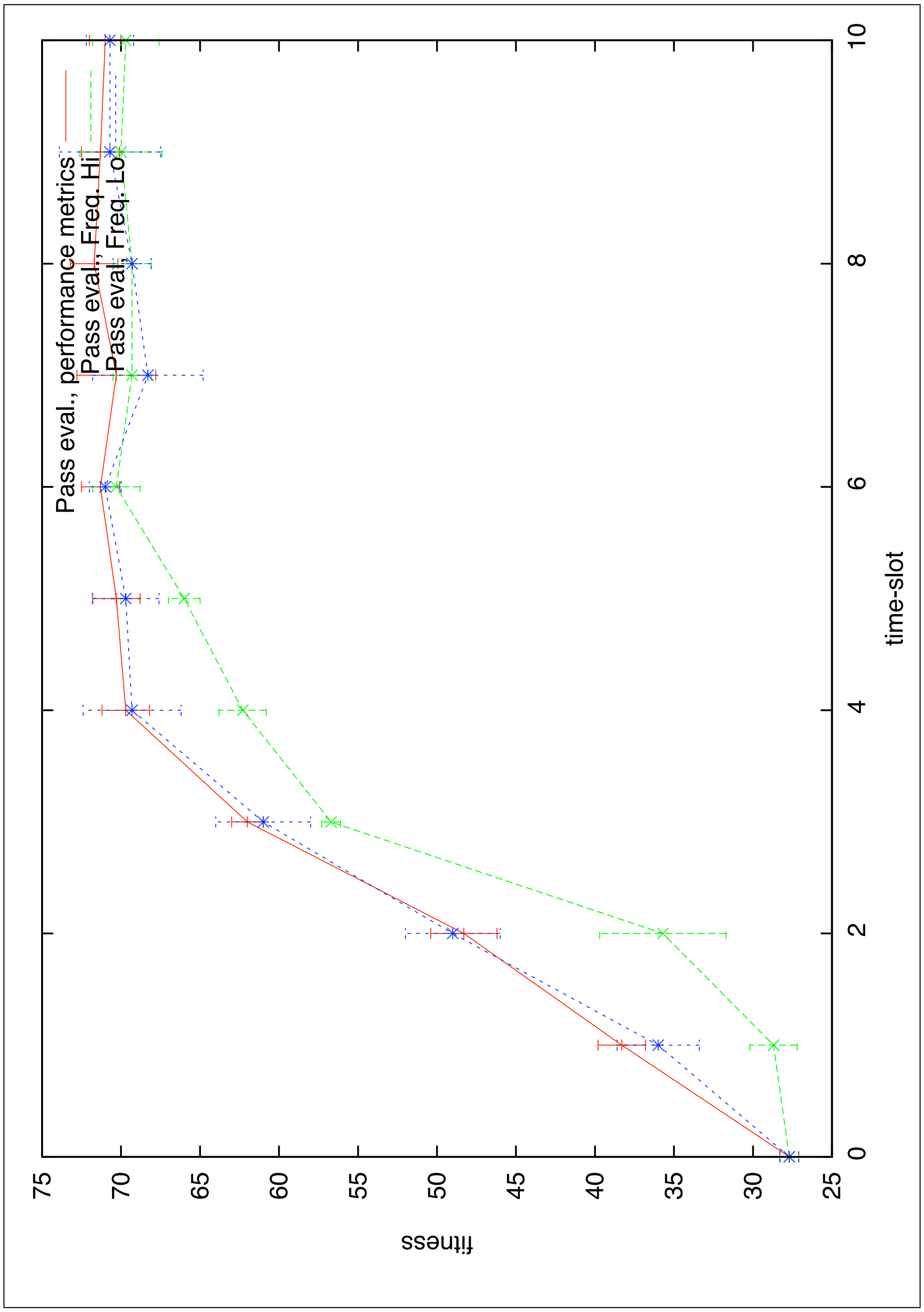


Figure 6.4: Blame assignment methods learning progress – Pass evaluation

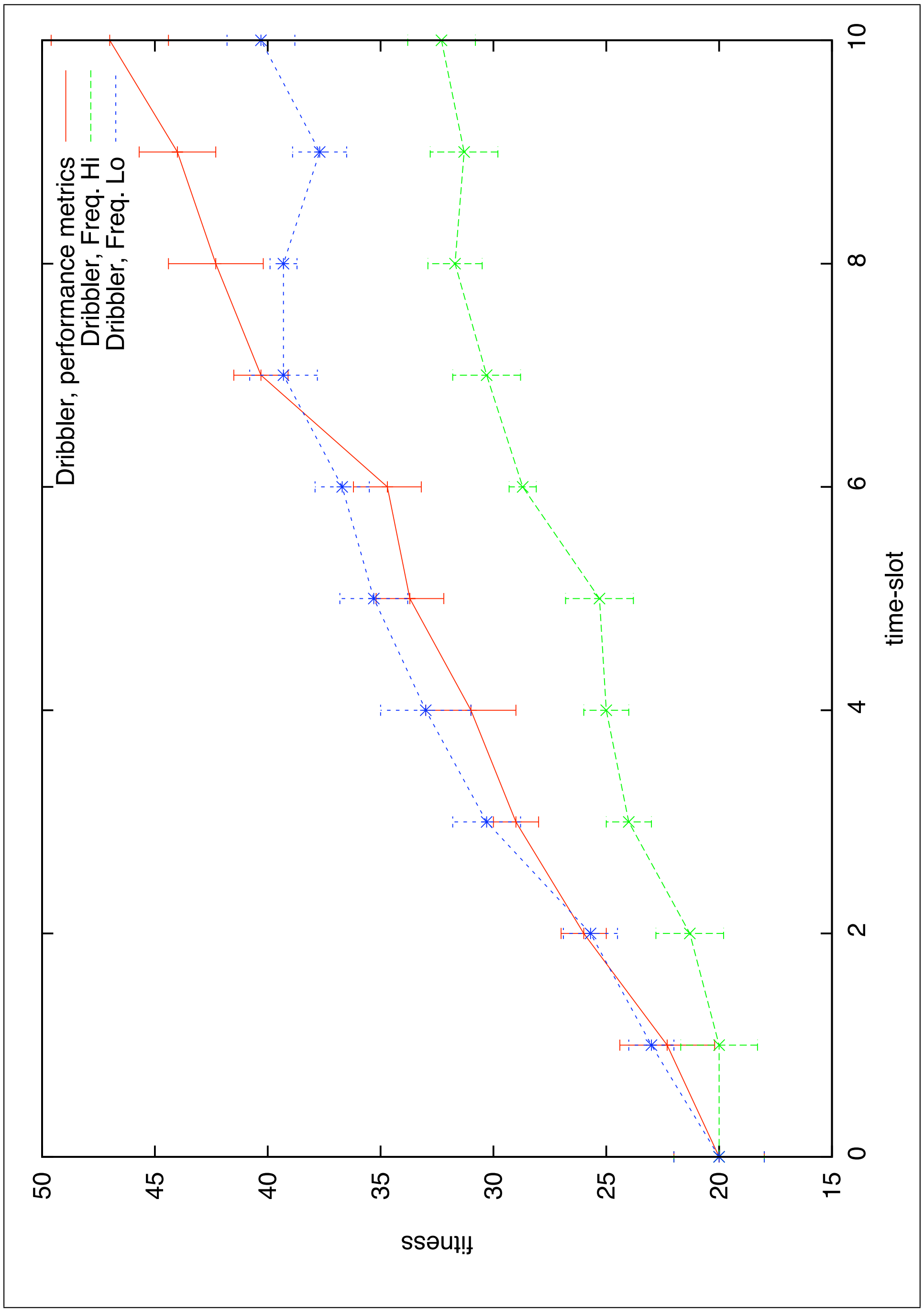


Figure 6.5: Blame assignment methods learning progress – Dribble

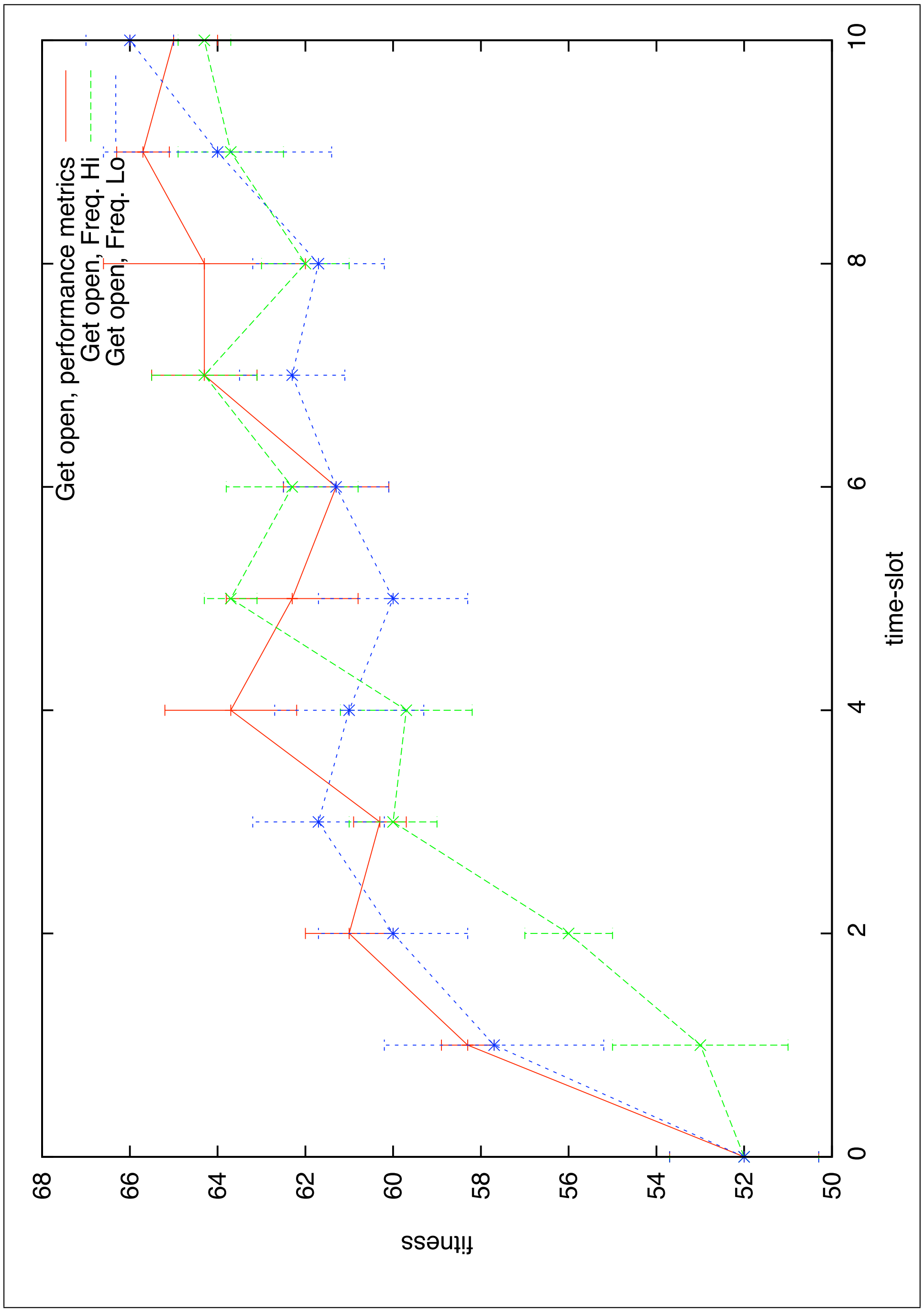


Figure 6.6: Blame assignment methods learning progress – Get open

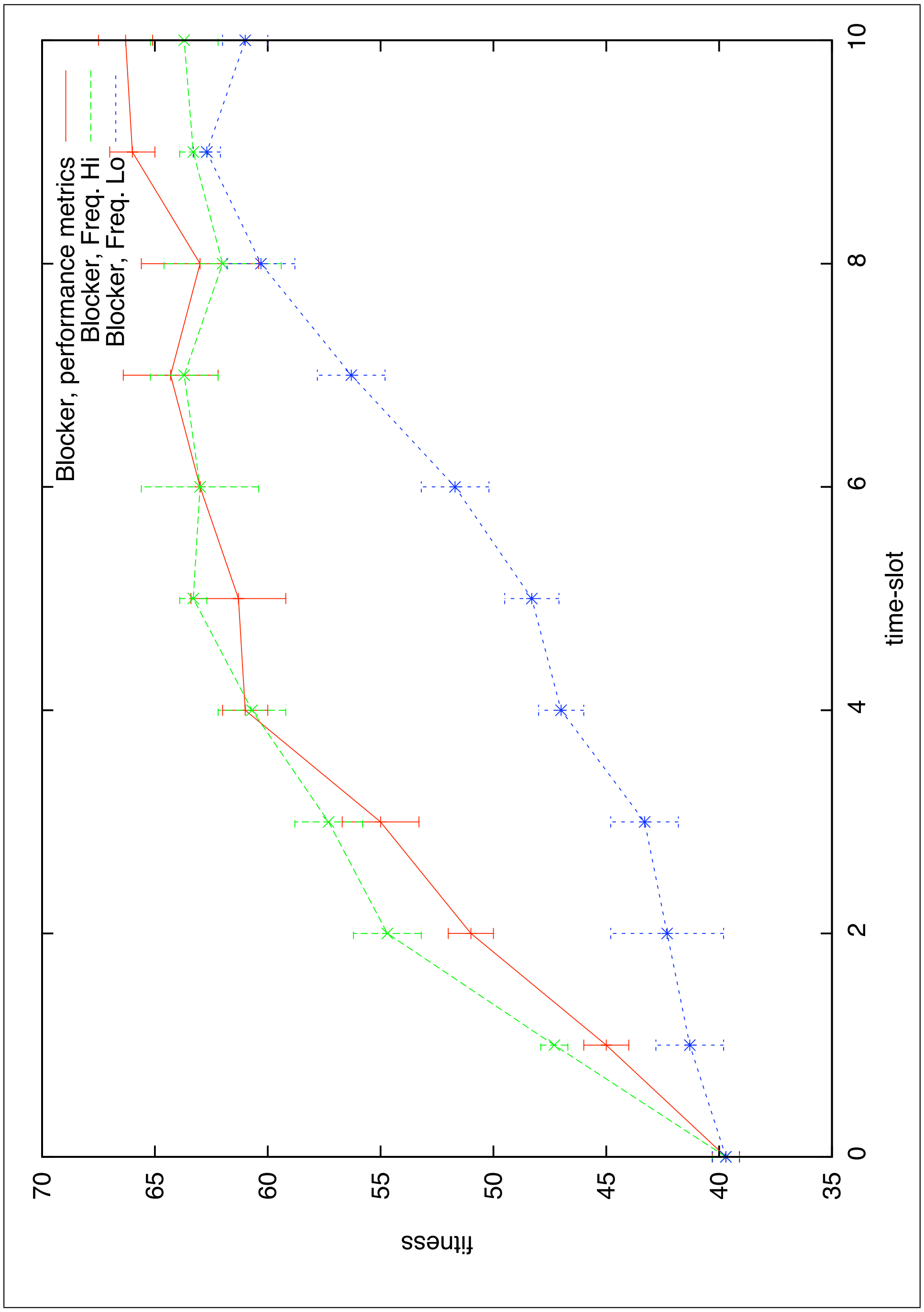


Figure 6.7: Blame assignment methods learning progress – Block

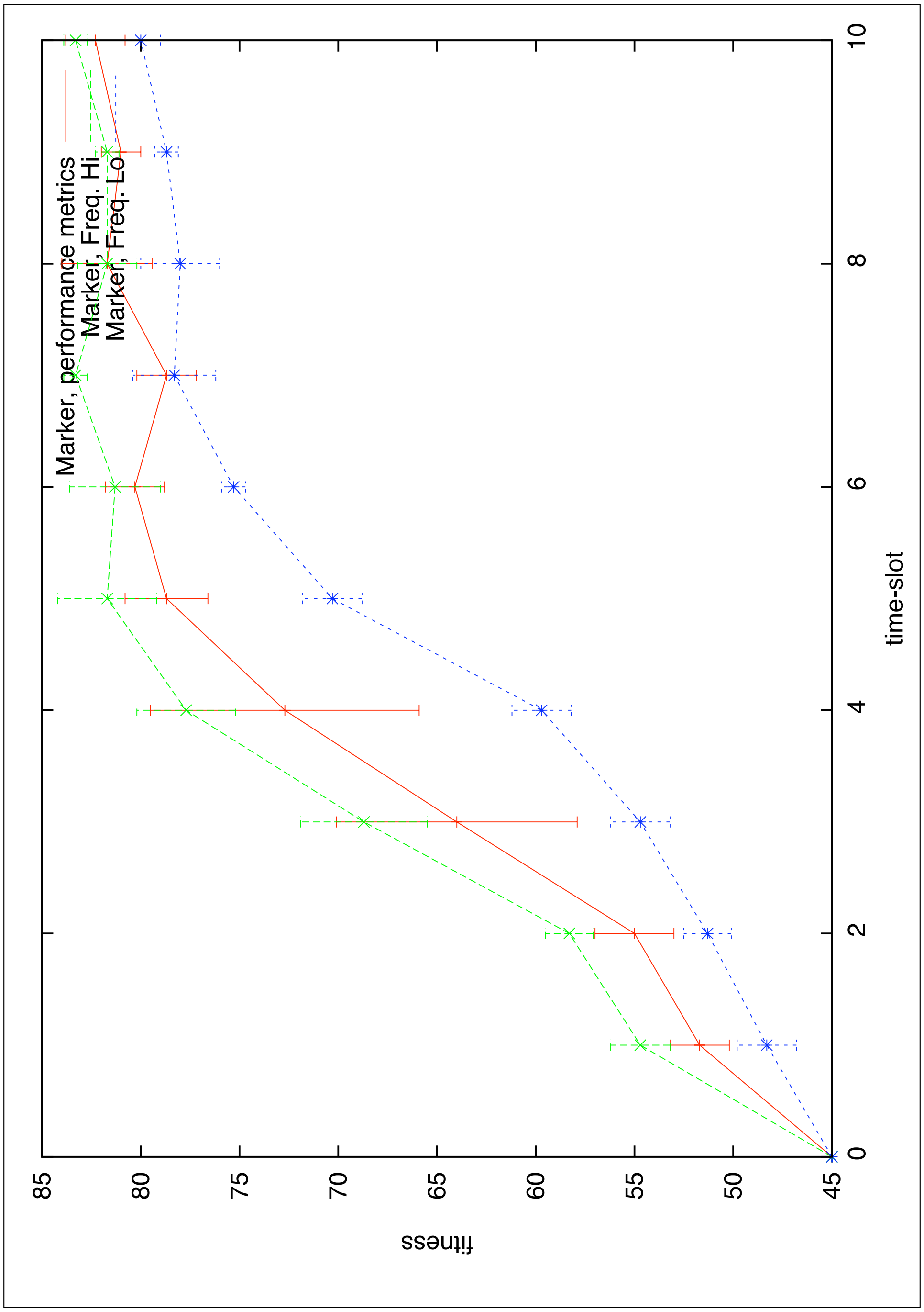


Figure 6.8: Blame assignment methods learning progress – Mark

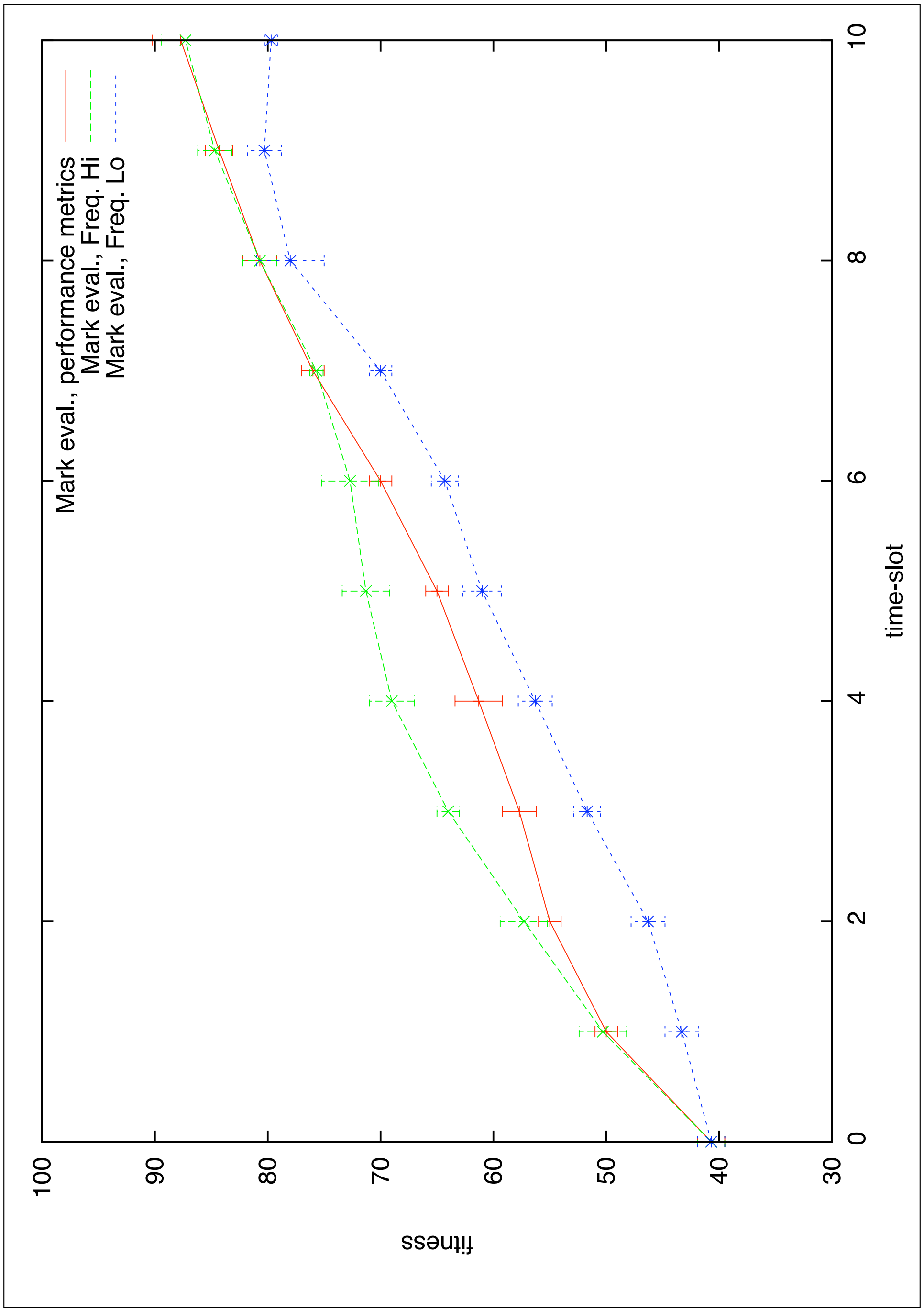


Figure 6.9: Blame assignment methods learning progress – Mark evaluation

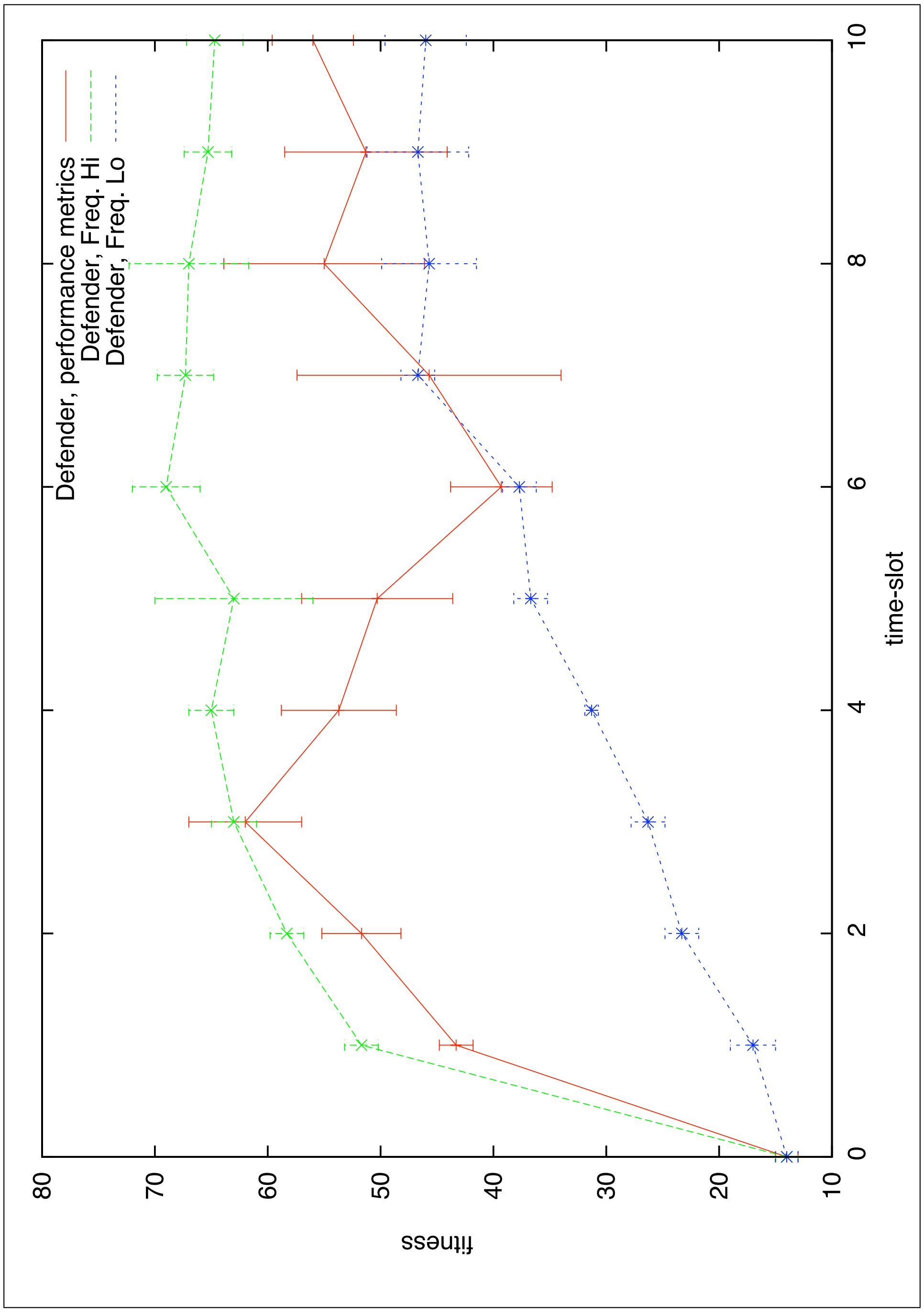


Figure 6.10: Blame assignment methods learning progress – Defend

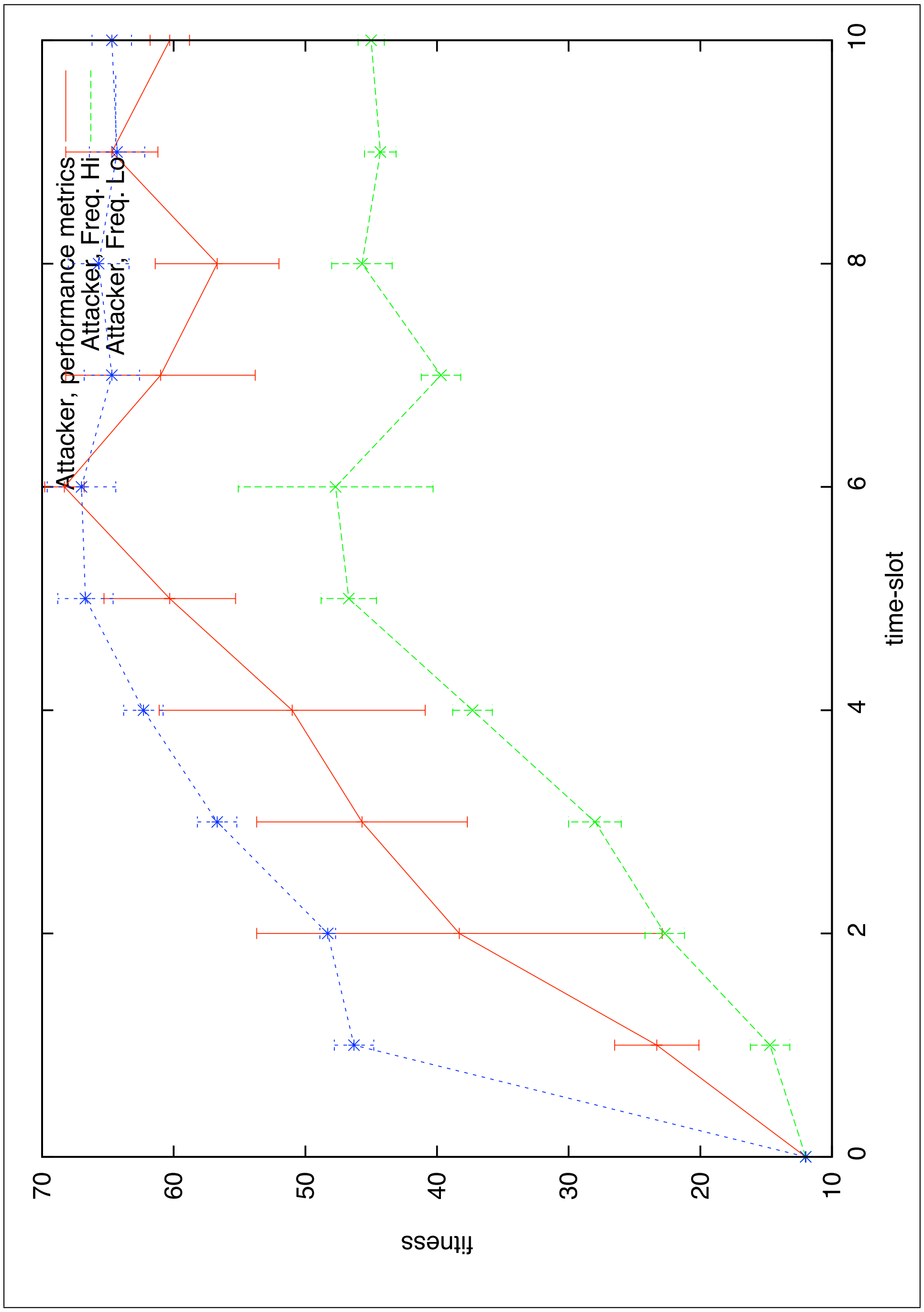


Figure 6.11: Blame assignment methods learning progress – Attack

7.0 CONCLUSION

This dissertation contributes several techniques for building cooperative agents in a real-time, noisy, and adversarial environment using Evolutionary Algorithms. Section 7.1 summarizes the dissertation's scientific contributions to the fields of Evolutionary Algorithms and Multi-Agent Systems. A discussion of future research directions is provided in Section 7.2.

7.1 CONTRIBUTIONS

The main contributions of this dissertation are listed as follows:

- **Incremental learning**

  Chapter 4 illustrates that evolution can be applied incrementally to achieve a desired complex behavior. In this approach, evolutionary learning shows a faster learning rate and provides a better performance than a non-incremental approach for the ball intercepting and shooting tasks.

- **Hierarchical coevolution**

  A hierarchical coevolutionary paradigm was introduced and applied in a complex domain, in which learning the domain's goal directly is intractable. Dividing a complex problem into subcomponents and coevolving them in a layered approach allowed for learning to proceed incrementally from low levels to higher ones. The goal task, which was at the highest level of the hierarchy, learned a desired behavior and provided a satisfactory performance. Without such an approach, such a performance cannot be achieved due to the complexity that makes a straightforward learning approach futile.

  In a Multi-Agent System, each agent may have its own learning process. In Chapter 5, a method of applying learning to a squad of agents in which they share their learning process concurrently is applied. This approach simplified the learning process, and made it viable for a complex domain having 22 interactive agents.

- **Orchestration**

  In Chapter 6, a new method is devised to manage the interaction between coevolutionary learning processes. Instead of using a fixed policy for concurrent learning, a dynamic policy is used based on a set of performance metrics which are gathered during the learning process. This method showed a significant performance enhancement over static policies. This extends the benefits of using coevolution in learning complex tasks, and shows its potential as an online learning algorithm.

- Coevolution is used in learning a team of robots to play a soccer game in a simulated environment. A decent behavior was achieved, revealing the great potential of coevolution in complex domains. A team of soccer players was learned entirely by a machine learning algorithm and showed a competitive performance against other hand-crafted soccer teams. This demonstrates that coevolution can be successful in allowing agents to learn to cooperate and become individually skilled in a complex, dynamic, and noisy environment such as RoboCup soccer.

7.2 FUTURE DIRECTIONS

The contributions of this dissertation stretch over many domains, namely, that of RoboCup soccer, Multi-Agent Systems, and Evolutionary Algorithms. In this section, a discussion of the shortcomings and future research directions regarding the main research areas covered in this dissertation is presented.

RoboCup soccer

In this research, many aspects of the RoboCup domains were simplified and so were not considered. For example, some actions, like tackle and turn head, and some sensors, such as agent's body sensors, were not used. Coaching and formation are two other challenges that can be tackled in future research.

In addition, there are other aspects that must be considered in developing a competitive RoboCup team. The handling of communication between the players and the server in real-time, determining a way to synchronize with the server time, and devising a world model that would provide more information than the limited (and noisy) players' sensors are just a handful of the problems that need to be tackled. This thesis did not cover those aspects. Therefore, to develop a champ RoboCup team these aspects have to be addressed.

In recent years, the RoboCup community has started a three-dimensional soccer simulation. The third dimension added more complexity to the environment, which makes implementing an agent a more challenging task. Although 3D simulation soccer is in its infancy, it would be an interesting research direction to investigate the use of coevolution in such an environment.

Multi-Agent System

In a Multi-Agent System, learning can be applied in may ways. In homogeneous learning, identical behaviors are assigned to all agents, while in heterogeneous learning each agent is assigned a different behavior. Alternatively, in a hybrid approach agents are split into different squads. All agents in the same squad are assigned the same behavior. Only the hybrid approach is investigated in this dissertation, as a squad of attackers and defenders were learned concurrently. It would be useful to test other learning approaches within evolutionary algorithms. Another direction to pursue is to investigate the differences between concurrent and non-cuncurrent learning methods.

Evolutionary Algorithms

In an evolutionary algorithm, many parameters need to be assigned. Population size, number of generations and mutation rate are just a handful of them. One future direction is to make a sensitivity analysis of those parameters and see their effect on evolution.

Coevolution can be done in various ways as well. In this work, only the best individuals were used in the evaluation of the coevolved population. It is important to investigate other methods of evaluation, such as using the last generation or random individuals in the evaluation. In the round robin regime, token size was not analyzed. It would be an interesting research direction to analyze the effect of token size on coevolution.

In Chapter 5, it was shown that adaptation between coevolved populations plays an important role in performance. Additionally, each population had a different learning rate and when a proper method was used to orchestrate their interacting learning it bestowed an improved performance. This opens the door to a new research direction – investigating the effect of learning rate on the performance of coevolved populations.

In this research, coevolution has been applied in simulated robotic soccer. Many other domains share aspects of RoboCup soccer, so coevolution may be used to examine them as well. For example, some aspects of this research have been applied in a Cyber Mouse competition(Alanjawi and Liberato 2007), where a robotic mouse navigates through a maze to reach cheese in a real-time simulated environment. This dissertation shows that coevolution is a good candidate to be used in different simulated environments such as a robotic football or search and rescue domains.

7.3 CONCLUDING REMARKS

This dissertation shows that coevolution can be successful in allowing agents in the RoboCup soccer domain to learn to cooperate and become individually skilled. In such a domain, interaction between agents highly affects their learning. Coevolution allows agents to adapt to each others' behavior as the learning proceeds. It is important to point out the crucial impact of adaptation on learning in such domains. The author hopes that this dissertation will prove useful in addressing the use of coevolution in complex problems, and, ultimately, will improve our understanding of coevolution and its capabilities. Overall, the author believes AI algorithms can tackle more complex obstacles in our lives, and its effectiveness will be proven in solving more sophisticated problems as computer hardware improves.

APPENDIX

IMPLEMENTATION DETAILS

The implementation of experiments of this dissertation is based on the SAMUEL system. To build a SAMUEL domain, the user must create domain code, and a number of ancillary files. Section A.1 provides some details on the domain's code. Sections A.2, A.3 and A.4 give a description of the ancillary files. The reader may refer to the SAMUEL manual (Grefenstette 1997) for more detailed information on how to build a SAMUEL domain.

A.1 DOMAIN'S CODE ORGANIZATION

The version of SAMUEL that was used was written in Java. Therefore, to build a SAMUEL domain, a main class "Player" that subclasses the abstract class "SamServant" was implemented. Figure A.1 illustrates the java class organization of the soccer domain that was implemented. The "Player" class implements all of the abstract methods of "SamServant" that deals with the evaluation such as acquiring sensors and applying actions methods. The "Player" class creates SAMUEL agents for each learning task, and sets up the soccer environment using "SoccerEnv" class. Depending on the run mode, i.e. learning or demonstration mode, the "SoccerEnv" class interacts with the RoboCup Soccer server by sending players' actions and getting players' sensor information. In learning mode, the "SoccerEnv" class uses JNI methods which are implemented in a soccer server C/C++ library. In demonstration mode, the "SoccerEnv" class uses the "RoboPlayer" package to communicate with the soccer server using UDP ports. The “RoboPlayer” class is responsible for all data communication between “SoccerEnv” and the Robocup soccer server process, including parsing the soccer server messages and handling network synchronization issues. The “Player” class also implements a graphical interface for demonstrating learning tasks as can be seen in Figure A.2. It uses the “SoccerScope” package to display robotic players as they act in a soccer field.

Each learning task, i.e. shooting, passing, etc..., has its own sensor/action methods. The main class, “Player”, creates a “tPlayer” object for each task. For each learning task, a corresponding class that implements a “tPlayer” class was implemented. For example, for an attacking task, the “tAttack” class was implemented that extends the “tplayer” class. The “tAttack” class implements all the methods needed by SAMUEL which are related to the attacking task, e.g. sensing, acting, episode ending, and payoff calculation methods. At each evaluation step, the “Player” class decides which agents and which “tplayer” objects must be active during the evaluation step.

For orchestration (Chapter 6), two scripts were implemented to gather frequency and performance measurements. The scripts accept a RoboCup soccer game file (.rcg, or .rcl) as input and output the tasks’ execution frequency and the performance metrics values.

A.2 ATTRIBUTES

Each task in SAMUEL has an attribute file that contains the structure of its sensors and actions. Many learning tasks share the same structure of soccer sensors or actions. In the soccer environment, the ball and the player sensors are mostly used in all tasks. A player can sense the ball’s distance, heading, and bearing. The structure of the ball’s sensors is illustrated in Figure A.3. Player’s sensor has the same structure as the ball’s sensors. High-level tasks, such as attacking, have a selection action, in which a subtask must be executed, as illustrated in Figure A.4. Figure A.5 shows the structure of actions for dashing tasks, e.g. blocking, intercepting, getting open, while shooting actions are illustrated in Figure A.6.

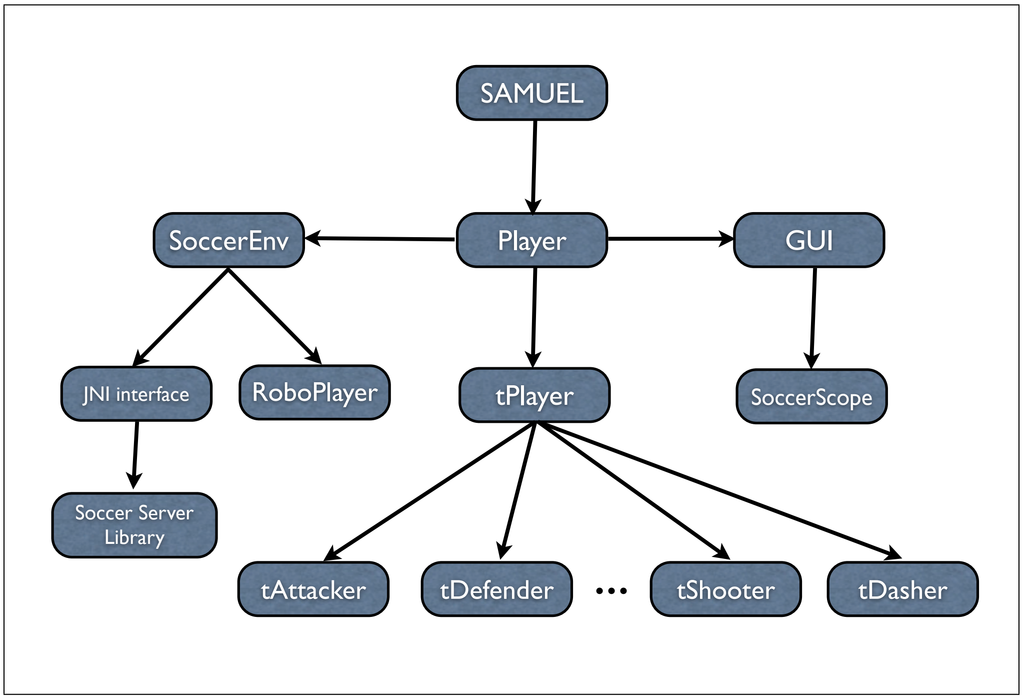
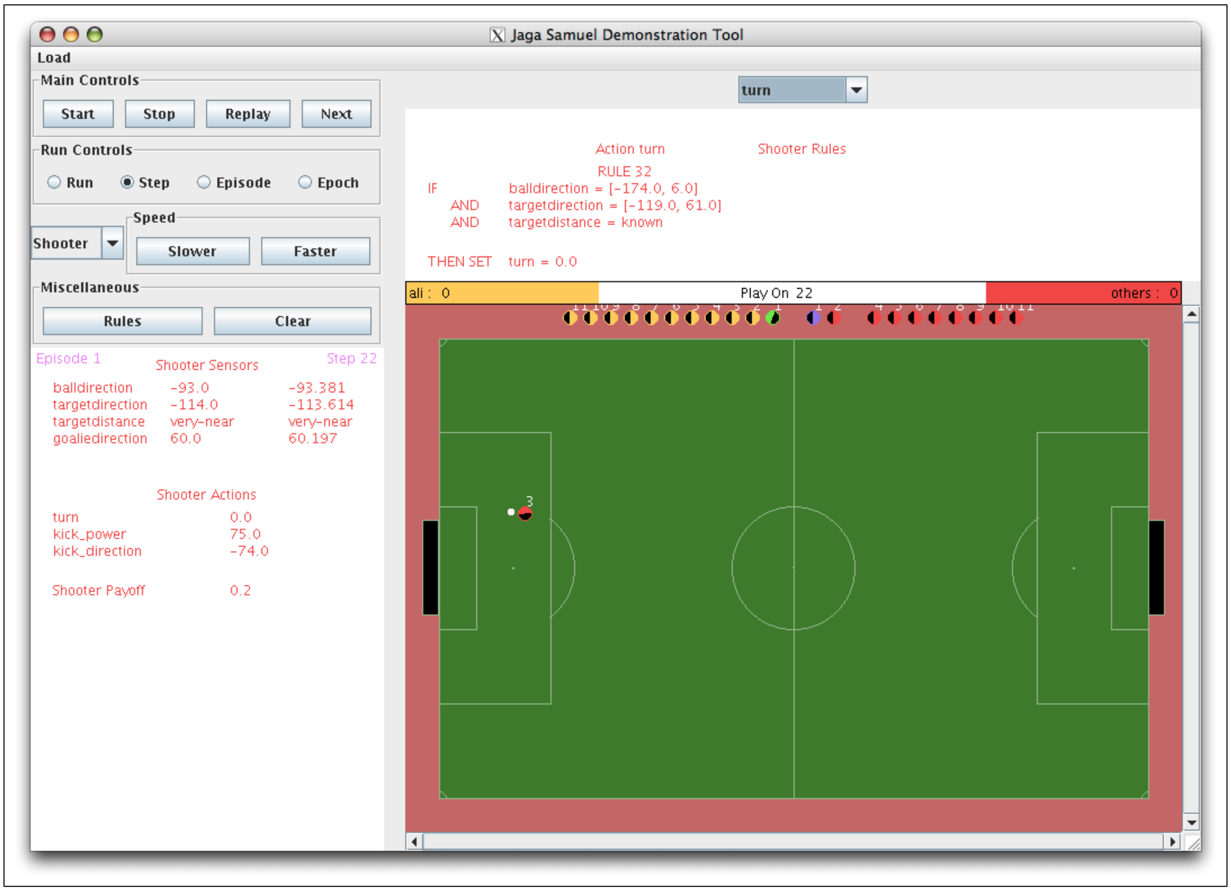


Figure A.1: Class organization



○ Run ● Step ○ Episode ○ Epoch

Shooter

Episode 1 Shooter Sensors Step 22

balldirection	-93.0	-93.381
targetdirection	-114.0	-113.614
targetdistance	very-near	very-near
goaliedirection	60.0	60.197

Shooter Actions

turn	0.0
kick_power	75.0
kick_direction	-74.0

Shooter Payoff	0.2

Action turn Shooter Rules

RULE 32

```
IF        balldirection = [-174.0, 6.0]
   AND    targetdirection = [-119.0, 61.0]
   AND    targetdistance = known

THEN SET  turn = 0.0
```

ali : 0 Play On 22 others : 0

Figure A.2: Soccer Demonstration GUI

```
condition 1:
name = BallD
type = structured
match = symbolic
order = none
leaf-values = 8
unknown
very-far
far
near
very-near
close
very-close
kickable
interior-values = 1
name = known
children = 7
very-far far near very-near close very-close kickable
exact = 1
condition 2:
name = BallB
type = cyclic
low  = -180
high = 180
step = 1
exact = 1
condition 3:
name = BallH
type = cyclic
low  = -180
high = 180
step = 1
exact = 1
```

Figure A.3: Ball's sensors

```
action 1:
name = task
type = structured
match = numeric
low  = 22
high = 25
order = linear
leaf-values = 4
Shoot
Pass
Open
Dribble
interior-values = 0
```

Figure A.4: Attacking action

A.3 INITIAL RULEBASES

The initial rulebases for each task were generated randomly using the attribute files. In some experiments, fixed rulebases were used for non-learning agents. For example, when learning a shooting task, the attacking rulebase was fixed to always select a shooting action as illustrated in Figure A.7.

A.4 PARAMETERS

Figure A.8 shows an example of a parameter file for a dashing experiment. Besides SAMUEL's parameters, the user can specify a number of parameters for the soccer learning environment. For example the initial ball and players' positions in the field can be specified by a circle (a point and a radios) in which the ball (or the player) will be placed randomly inside it. The ball can also be placed randomly in a position relative to a specified player.

```
action 1:
name = turn
type = cyclic
low  = -180
high = 180
step = 1
action 2:
name = dash_power
type = linear
low  = 10
high = 100
step = 30
action 3:
name = command
type = structured
match = numeric
low  = 0
high = 1
order = linear
leaf-values = 2
Turn
Dash
interior-values = 0
```

Figure A.5: Dashing actions

```
action 1:
name = turn
type = cyclic
low  = -180
high = 180
step = 10
action 2:
name = kick_power
type = linear
low  = 25
high = 100
step = 25
action 3:
name = kick_direction
type = cyclic
low  = -90
high = 90
step = 1
action 4:
name = command
type = structured
match = numeric
low  = 0
high = 1
order = linear
leaf-values = 2
Turn
Kick
interior-values = 0
```

Figure A.6: Shooting actions

```
RULEBASE 73

RULE 0

        IF

        THEN SET task = Shoot

Id 0 Parent 0 Created 0 Fixed 3 Matched 0 Partially-matched 0 Dest 0

Action task Bid 0 Active 0 Fired 0 Mean 0.5 Std 0.0 Strength 0.5 Act 1.0


gen: 0 trial: 0 value: 0.0

parent1: 0 parent2: 0 seed: 0
```

Figure A.7: An example of a fixed initial rulebase for attacking task

```
popsize = 50
bestsize = 10
gens = 100
length = 250
pre_eval = 10
post_eval = 10
cluster_eval = 10
fitness = 10
extract = 0
test = 20
best_interval = 5
rulebase_type = Attacker,Defender,Dasher:ga:best,Shooter,Passer,PassEval,Open,\
Dribbler,ClearPass,Mark,MarkEval,Block,Goalie,MoveNPass,Exec,Gotosp
attributefile = attributes
initfile = init
# ExpType         1.dasher only, 2.shooter 3. goalie, 4. passer
#                        5.dribbler 6 marker 7. blocker 8. def/att
ExpType = 1
ExecMaxTime = 100
attMaxTime = 30
defMaxTime = 30
shootMaxTime = 30
passMaxTime = 30
dribbleMaxTime = 30
openMaxTime = 30
dashMaxTime = 30
blockMaxTime = 30
markMaxTime = 30
clearMaxTime = 30
movenpassMaxTime = 30
GotospMaxTime = 30
# formation can be 442, 532, 352, 443 ...
formation = 442
# Players playerNum,x,y,Max_distance_from_pt,min_face_directionangle,
#                        max_face_directionangle;other players...
# Ball ball_x,ball_y,max_distance_from_pt,direction_from_pt OR
#        playerID,max_dist_from_player,direction_from_player
# Example:  Players 1,0.0,13.0,5.0,50,70;12,0.0,15.0,10.0,-45,45, Ball 1,5.0,90
Players = 13,0.0,0.0,20.0,-120,120
Ball = 13,12.0,120
logging = 0
```

Figure A.8: An example of an experiment parameter file

BIBLIOGRAPHY

Alanjawi, Ali, and Frank Liberato. 2007. "EvoRobert System Description." *CiberMouse at the 28th IEEE Real-Time Systems Symposium.*

Andre, D., and A. Teller. 1999. "Evolving team darwin united." Edited by M. Asada and H. Kitano, *RoboCup–98: Robot Soccer World Cup II.* Berlin: Springer Verlag.

Angeline, P., and J. Pollack. 1993. "Competitive Environments Evolve Better Solution for Complex Tasks." *Proc. of the 5th International Conference on Genetic Algorithms.*

Asada, M., H. Uchibe, and K. Hosoda. 1999. "Cooperative behavior acquisition for mobile robots in dynamically changing real worlds via vision–based reinforcement learning and development." *Articial Intelligence* 110:275–292.

Axelrod, R. 1984. *The Evolution of Cooperation.* New York: Basic Books.

———. 1989. Chapter 3 of *Genetic Algorithms and Simulated Annealing*, edited by L. David, 32–41. Morgan Kaufmann.

Bellman, R. 1957. *Dynamic Programming.* Princeton, NJ: Princeton University Press.

Beyer, H., and D. Arnold. 2003. "The Steady State Behavior of $(\mu/\mu_l, \lambda)$–ES on Ellipsoidal Fitness Models Disturbed by Noise." Edited by Cantú-Paz et al., *GECCO.* Springer Verlag, 525–536.

Blair, A., E. Sklar, and P. Funes. 1998. "Co–evolution, Determinism and Robustness." *Asia–Pacific Conf. on Simulation evolution and learning.*

Box, G. 1957. "Evolutionary operation: A method for increasing industrial productivity." *Journal of Royal Statistical Society C* 6 (2): 81–101.

Bremermann, H. 1962. "Optimization through evolution and recombination." Edited by M. Yovits, G. Jacobi, and G. Goldstein, *Self–Organizing Systems.* Spartan Books.

Bucci, A., and J. Pollack. 2002. "A Mathematical Framework for the Study of Coevolution." *FOGA VII.*

Bucci, A., and J.B. Pollack. 2005. "On Identifying Global Optima in Cooperative Coevolution." *Proceedings of the Genetic and Evolutionary Computation Conference.*

Bugajska, M., and A. Schultz. 2000, July. "Co–evolution of Form and Function in the Design of Autonomous Agents: Micro Air Vehicle Project." *GECCO 2000 Workshop on Evolution of Sensors in Nature, Hardware, and Simulation.* Las Vegas, NV.

Cliff, D., and G. Miller. 1995. "Tracking The Red Queen: Measurements of adaptive progress in co–evolutionary simulations." *Proc. of the 3rd European Conf. on artificial life.*

Coelho, A., and I. Ricarte D. Weingaertner, RR. Gudwin. 2001. "Emergence of multiagent spatial coordination strategies through artificial coevolution." *Computers and Graphics* 25 (6): 1013–1023.

Crites, R., and A. Barto. 1996. "Improving elevator performance using reinforcement learning." Edited by D. Touretzky, M. Mozer, and M. Hasselmo, *Neural Information Processing Systems 8.*

Daley, R., A. Schultz, and J. Grefenstette. 1999. "Co–evolution of Robot Behaviors." *Proc. of the SPIE Int. Symposium on Intelligent Systems and Advanced Manufacturing.*

Darwen, P. 2001. "Why Co–evolution beats Temporal Difference learning at Backgammon for a linear architecture, but not a non–linear architecture." *Congress on Evolutionary Computation.*

Darwin, C. 1859. *On the Origin of Species by Means of Natural Selection.* London: John Murry.

Das, R., and D. Whitley. 1991. "The only challenging problems are deceptive: Global search by solving order–1 hyperplanes." Edited by R. Belew and L. Booker, *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann, 166–173.

de Boer, Remco, and Jelle Kok. 2002, February. "The Incremental Development of a Synthetic Multi–Agent System: The UvA Trilearn 2001 Roboti Soccer Simulation Team." Master's thesis, Faculty of Science, University of Amsterdam.

De Jong, E., and T. Oates. 2002. "A Coevolutionary Approach to Representation Development." *Proceedings of the ICML 2002 workshop on Development of Representations.*

De Jong, K. 1975. "Analysis of Behavior of a Class of Genetic Adaptive Systems." Ph.D. diss., University of Michigan, Ann Arbor.

De Jong, K., W. Spears, and D. Gordon. 1993. "Using genetic algorithms for concept learning." *Machine Learning* 13 (2): 161–188.

De Klepper, Nathan. 1999. "High Level Functions in Genetic Programming for Robosoccer." Honours Thesis, RMIT, Department of Computer Science.

Dorigo, M., and M. Colometti. 1998. *Robot Shaping: An Experiment in Behavioral Engineering.* Cambridge, MA: MIT Press.

Fard, Amin Milani, Vahid Salmani, Mahmoud Naghibzadeh, Sedigheh Khajouie Nejad, and Hamed Ahmadi. 2007. "Game Theory-based Data Mining Technique for Strategy Making of a Soccer Simulation Coach Agent." Edited by Heinrich C. Mayr and Dimitris Karagiannis, *Information Systems Technology and its Applications, 6th International Conference ISTA'2007, May 23-25, 2007, Kharkiv, Ukraine*, Volume 107 of *LNI.* GI, 54–65.

Ficici, S., and J. Pollack. 2000. "A Game–Theoretic Approach to the Simple Coevolutionary Algorithm." Edited by M. Schoenauer, *PPSN VI.* Springer Verlag.

———. 2001. "Pareto Optimality in Coevolutionary Learning." *The 6th European Conf. on Artificial Life.*

Floreno, D. 1998. Chapter Evolutionary Robotics in Behavior Engineering and Artificial Life. of *Evolutionary Robotics II.* Ontario Canda, AAI Books.

Floreno, D., and S. Nolfi. 1997. "God save the Red Queen! competition in co–evolutionary robotics." *Proc. of the 2nd Annual Conf. of Genetic Algorithms.*

Floreno, D., S. Nolfi, and F. Mondada. 1998. "Competitive co–evolutionary robotics: from theory to practice." Edited by R. Pfeifer, *From Animals to Ainmates: Proceedings of the Forth Internationa Conference on Simulation of Adaptive Behavior.* MIT Press–Bradford Books.

Fogel, D. 1992. "Evolving Artificial Intelligence." Ph.D. diss., University of California, San Diego.

Fogel, L., A. Owens, and M. Walsh. 1966. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons.

Foroughi, E., F. Heintz, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, and T. Steffens. 2002. *RoboCup Soccer Server User Manual: for Soccer Server version 7.07 and later.* At http://sourceforge.net/projects/sserver.

Forrest, S. 1985. Documenation for Prisoners Dilemma and Norms Programs That Use the Genetic Algorithm. Uiversity of Michigan, Ann Arbor.

Friedman, G. 1959. "Digital simulation of an evolutionary process." *General Systems Yearbook* 4:171–184.

Funes, P., and J. Pollack. 1998. "Evolutionary Body Building: Adaptive physical designs for robots." *Artificial Life*, vol. 4.

Giordana, A., and F. Neri. 1996. "Search–intensive concept induction." *Evolutionary Computation* 3 (4): 375–416.

Giordana, A., L. Saitta, and F. Zini. 1994. “Learning disjunctive concepts by means of genetic algorithms.” Edited by W. Cohen and H. Hirsh, *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, 96–104.

Goldberg, D., and J. Richardson. 1987. “Genetic algorithms with sharing for multimodal function optimization.” Edited by J. Grefenstette, *Proceedings of the Second International Conference on Genetic Algorithms.* Lawrence Erlbaum Associates, 41–49.

Grefenstette, J. 1989. “A system for learning control strategies with genetic algorithms.” Edited by J. Schaffer, *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann, 183–190.

Grefenstette, J., C. Ramsey, and A. Schultz. 1990. “Learning sequential decision rules using simulation models and competition.” *Machine Learning* 5 (4): 355–381.

Grefenstette, John. 1997. *The User’s Guide to SAMUEL – 97: An Evolutionary Learning System.* Navy Center for Applied Research in Artificial Intelligence.

Grefenstette, John J., and Connie Loggia Ramsey. 1992. “An Approach to Anytime Learning.” *Proc.of the Ninth Int. Machine Learning Workshop.* San Mateo, CA: Morgan Kaufmann, 189–195.

Gustafson, S. M., and W. H. Hsu. 2000, July. “Genetic Programming for Strategy Learning in Soccer–Playing Agents: A KDD–Based Architecture.” *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO–2000) Workshop Program.* Las Vegas, NV.

Gustafson, Steven M., and William H. Hsu. 2001. “Layered Learning in Genetic Programming for a Co–operative Robot Soccer Problem.” Edited by Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, *Genetic Programming, Proceedings of EuroGP’2001*, Volume 2038. Springer–Verlag, 291–301.

Hillis, W. 1990. “Co–evolving Parasites Improve Simulated Evolution As an Optimization Procedure.” *Physica D* 42:228–234.

Hohn, Charles. 1997. “Evolving predictive functions from observed data for simulated robots.” Master Honours Thesis, Dept. of Computer Science, University of Maryland at College Park.

Holland, J. 1975. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press.

———. 1986. “Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule–based systems.” Edited by R. Michalski, J. Carbonell, and T. Mitchell, *Machine Learning*, Volume 2. Morgan Kaufman, 593–623.

Holland, J., and J. Reitman. 1978. “Cognitive systems based on adaptive algorithms.” Edited by D. Waterman and F. Hayes-Roth, *Pattern–Directed Inference Systems.* Academic Press.

Husbands, P. 1994. “Distributed coevolutionary genetic algorithms for multi–criteria and multi–constraint optimisation.” Edited by Fogarty, *Proceedings of Evolutionary computing, AISB Workshop Selected Papers.* Springer Verlag, 150–165. (LNCS 865).

Husbands, P., and F. Mill. 1991. “Simulated co–evolution as the mechanism for emergent planning and scheduling.” Edited by R. Belew and L. Booker, *Proceedings of the Forth International Conference on Genetic Algorithms,.* Morgan Kaufmann., 264–270.

Janikow, C. 1993. “A knowledge–intensive genetic algorithm for supervised learning.” *Machine Learning* 13 (2): 189–228.

Juillé, H. 1999. “Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm.” Ph.D. diss., Brandeis University.

Juillé, H., and J. Pollack. 1996. “Co–evolving Intertwined Spirals.” *Proc. of the 5th Annual Conf. on Evolutionary programming.*

———. 1998. “Coevolutionary Learning: a Case Study.” *Proc. of the 5th Int. Conf. on Machine Learning.*

Kaelbing, L., M. Littman, and A. Moore. 1996. “Reinforcement Learning: A survey.” *Journal of AI research*, vol. 4.

Kalyanakrishnan, Shivaram, Yaxin Liu, and Peter Stone. 2007. “Half Field Offense in RoboCup Soccer: A Multiagent Reinforcement Learning Case Study.” In *RoboCup-2006: Robot Soccer World Cup X*, edited by Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorenti, and Tomoichi Takahashi. Berlin: Springer Verlag. To appear.

Kalyanakrishnan, Shivaram, and Peter Stone. 2007, May. “Batch Reinforcement Learning in a Complex Domain.” *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems.*

Kalyanakrishnan, Shivaram, Peter Stone, and Yaxin Liu. 2008. “Model-based Reinforcement Learning in a Complex Domain.” In *RoboCup-2007: Robot Soccer World Cup XI*, edited by Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert. Berlin: Springer Verlag. To appear.

Kitano, H., and M. Asada. 1998. “RoboCup Humanoid Challenge: That’s One Small Step for A Robot, One Giant Leap for Mankind.” *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS–98).*

Kitano, H., M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. 1995. “RoboCup: The Robot World Cup Initiative.” *Proceedings of the IJCAI–95 Workshop on Entertainment and AI/Alife.*

Kostiadis, K. 2002. “Learning to Co–operate in Multi–Agent Systems.” Ph.D. diss., University of Essex, Wivenhoe Park, Colchester CO4 3SQ, United Kingdom.

Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

———. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Lubberts, A., and R. Miikkulainen. 2001. “Co–Evolving a Go–Playing Neural Network.” *GECCO.*

Luke, S. 1998. “Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97.” Edited by Koza et al., *Proceedings of the Third Annual Genetic Programming Conference (GP98).* San Fransisco: Morgan Kaufmann, 204–222.

Luke, Sean, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. 1997. “Co–evolving Soccer Softbot Team Coordination with Genetic Programming.” Edited by H. Kitano, *Proceedings of the RoboCup–97 Workshop at the 15th International Joint Conference on Artificial Intelligence (IJCAI97).* Japan.

Matkovic, Michael. 2002. “Co–evolving Competitive Behaviours in Genetic Programming.” Honours Thesis, RMIT, Department of Computer Science.

Matsubara, Hitoshi, Itsuki Noda, and Kazuo Hiraki. 1996, March. “Learning of cooperative actions in multi–agent systems: a case study of pass play in soccer.” *Adaptation, Coevolution and Learning in Multiagent Systems: Papers from the 1996 AAAI Spring Symposium*, Volume Menlo Park, CA. AAAI Press, 63–67.

Mehta, Manish. 2000. “Robotic Soccer Simulator Using Reinforcement Learning with Clustring.” Master’s thesis, The University of Texas at Arlington.

Messom, C. H., and M. Walker. 2002. “Evolving Cooperative Robotic Behaviour using Distributed Genetic Programming.” *International Conference on Control Automation, Robotics and Vision.* Singapore, 215–219.

Miller, F., P. Todd, and S. Hegde. 1989. “Designing neural networks using genetic algorithms.” *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann., 379–348.

Miller, G., and D. Cliff. 1994. “Protean behavior in dynamic games: Arguments for the coevolution of pursuit–evasion tactics.” Edited by D. Cliff, P. Husbands, J. Meyer, and S Wilson, *From Animals to Animates: Proc. of the Third Int. Conf. on Simulation of Adaptive Bahavior.* MIT Press, 411–420.

Montana, D., and L. Davis. 1989. “Training feedforward neural networks using genetic algorithms.” Edited by S. Sridharan, *Eleventh International Joint Conference on Artificial Intelligence,.* Morgan Kaufmann, 762–767.

Moriarty, D., and R. Miikkulainen. 1995. "Discovering Complex Othello Strategies Through Evolutionary Neural Networks." *Connection Science* 7, no. 3.

———. 1996a. "Efficient reinforcement learing through symbiotic evolution." *Machine Learning* 22:11–32.

———. 1996b. "Evolving obsticle avoidance behavior in a robot arm." *From Animals to Ainmates: Proceedings of the Forth Internationa Conference on Simulation of Adaptive Behavior.* 468–475.

———. 1998. "Forming Neural Networks through Efficient and Adaptive Coevolution." *Evolutionary Computation* 4, no. 4.

Moriarty, D., A. Schultz, and J. Grefenstette. 1999. "Evolutionary Algorithms for Reinforcement Learning." *Journal of AI Research*, vol. 11.

Mühlenbei, H. 1992. "How genetic algorithms really works: 1. Mutation and hillclimbing." Edited by R. Männe and B. Manderick, *PPSN II.*

Østergaard, Esben H., and Henrik H. Lund. 2002, August. "Co–Evolving Robot Soccer Behavior." *Proceedings of the Simulation of Adaptive Behavior (SAB–02).* Edinburgh, UK, 351–352.

———. 2003, March. "Co–Evolving Complex Robot Behavior." *Proceedings of ICES'03, The 5th International Conference on Evolvable Systems: From Biology to Hardware.* Trondheim, Norway, 308–319.

Oyman, A., H. Beyer, and H. Schwefel. 2000. "Analysis of the $(1, \lambda)$– ES on the parabolic ridge." *Evolutionary Computation* 8 (3): 249–265.

Pagie, L., and M. Mitchell. 2002. "A comparison of evolutionary and coevolutionary search." *International Journal of Computational Intelligence and Applications* 2 (1): 53–69.

Panait, L., and S. Luke. 2002. "A Comparative Study of Two Competitive Fitness Functions." *GECCO.*

Panait, Liviu, and Sean Luke. 2005. "Cooperative Multi-Agent Learning: The State of the Art." *Autonomous Agents and Multi-Agent Systems* 11 (3): 387–434.

Panait, Liviu, R. Paul Wiegand, and Sean Luke. 2004. "A Sensitivity Analysis of a Cooperative Coevolutionary Algorithm Biased for Optimization." *Genetic and Evolutionary Computation Conference.*

Paredis, J. 1994a. "Coevolutionary constraint satisfaction." Edited by Y. Davidor, H. Schwefel, and R. Manner, *PPSN III.* Springer Verlag. (LNCS 866).

———. 1994b. "Steps towards coevolutionary classification neural networks." Edited by R. Brooks and P. Maes, *Proc. of Artificial Life IV.* MIT Press – Bradford Books.

———. 1995. “The symbiotic evolution of solutions and their representations.” Edited by L. Eshelman, *Proceedings of the 6th ICGA.* Morgan Kaufmann., 359–365.

———. 1996. “Coevolutionary Computation.” *Journal of Artificial Life*, vol. 2.

———. 1997. “Coevolving Cellular Automata: Be Aware of the Red Queen.” *Proc. of the 8th ICGA.*

Parker, Gary B., and Jonathan W. Mills. 1999. “Adaptive Hexapod Gait Control Using Anytime Learning with Fitness Biasing.” Edited by Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 1. Orlando, Florida, USA: Morgan Kaufmann, 519–524.

Pollack, J., and A. Blair. 1998. “Co–evolution in the Successful Learning of Backgammon Strategy.” *Machine Learning* 32, no. 3.

Pollack, J., H. Lipson, S. Ficici, P. Funes, G. Hornby, and R. Watson. 2000. “Evolutionary Techniques in Physical Robotics.” *Proceeding of the 3rd international conference of Evolvable Systems: from biology to hardware.*

Pollack, J., H. Lipson, P. Funes, S. Ficici, and G. Hrnby. 1999. “Coevolutionary Robotics.” *First NASA/DOD Workshop on Evolvable Hardware.*

Pomerleau, D., and T. Jochem. 1996. “Image Processor Drives Across America.” *Photonics Spectra*, April, 80–85.

Popovici, Elena, and Kenneth De Jong. 2006a. “The Dynamics of the Best Individuals in Co-Evolution.” *Natural Computing* 5 (3): 229–255.

———. 2006b. “The Effects of Interaction Frequency on the Optimization Performance of Cooperative Coevolution.”

Potter, M. 1997. “The Design and Analysis of a Computational Model of Co–operative Coevolution.” Ph.D. diss., George Mason University.

Potter, M., and K. De Jong. 2000. “Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents.” *Evolutionary Computation* 8, no. 1.

Potter, M., K. De Jong, and J. Grefentette. 1995. “A Coevolutionary Approch to Learning Sequential Deision Rules.” Edited by L. Eshelman, *Proc. of the 6th ICGA*. Morgan Kaufmann, 366–372.

Rechenberg, I. 1964. “Cybernetic solution path of an experimental problem.” *Royal Aircraft Establishment.* English translation of lecture given at the Annual Conference of the WGLR at Berlin in September, 1964.

———. 1973. “Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.” *Stuttgart–Bad Cannstatt*. Frommann Holzboog.

Reed, J., R. Toombs, and N. Barricelli. 1967. “Simulation of biological evolution and machine learning.” *Journal of Theoretical Biology* 17:319–342.

Reynolds, C. 1994. “Competition, coevolution, and the game of tag.” Edited by R. Brooks and P. Maes, *Artificial Life IV*. MIT Press.

Riedmiller, M., and T. Gabel. 2007. “On Experiences in a Complex and Competitive Gaming Domain: Reinforcement Learning Meets RoboCup.” *Proceedings of the 3rd IEEE Symposium on Computational Intelligence and Games (CIG 2007)*. Honolulu, USA: IEEE Press, 17–23.

Riedmiller, M., and D. Withopf. 2005. “Effective Methods for Reinforcement Learning in Large Multi-AgentDomains.” *it – Information Technology Journal* 47 (5): 241–249.

Rosin, C. 1997. “Coevolutionary Search Among Adversaries.” Ph.D. diss., University of California, San Diego.

Rosin, C., and R. Belew. 1996. “New Methods for Competitive Coevolution.” Technical Report CS96–491, University of California, San Diego.

Rumelhart, D., G. Hinton, and R. Williams. 1986. “Learning internal representations by error propagation.” Edited by D. Rumelhart and J. McClelland, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Volume 1. MIT Press, 318–362.

Salustowicz, R., M. Wiering, and J. Schmidhuber. 1998. “Learning Team Strategies: Soccer Case Studies.” *Machine Learning* 33 (2–3): 263–282.

Sałustowicz, R. P., M. A. Wiering, and J. Schmidhuber. 1997. “Evolving Soccer Strategies.” Edited by N. Kasabov, R. Kozma, K. Ko, R. O’Shea, G. Coghill, and T. Gedeon, *Progress in Connectionist–based Information Systems: Proceedings of the Fourth International Conference on Neural Information Processing ICONIP’97*, Volume 1. Singapore: Springer Verlag, 502–505.

Schaffer, J., and J. Grefenstette. 1985. “Multi–objective learning via genetic algorithms.” *Proc. of the Ninth Int. Joint Conf. on Artificial Intelligence*. Morgan Kaufmann, 593–595.

Schwefel, H. 1975. “Evolutionsstrategie und numerische Optimierung.” Ph.D. diss., Technische Universität Berlin.

———. 1981. *Numerical Optimization of Computer Models*. John Wiley & Sons.

Shapiro, J. 1998. “Does Data–Model Co–evolution Improve Generalization Performance of Evolving Learners?” *Proceedings of PPSN V.* (LNCS 1498).

Sims, K. 1994. "Evolving 3D morphology and behavior by competition." Edited by R. Brooks and P. Maes, *Artificial Life IV.* MIT Press, 28–39.

Smith, S. 1980. "A Learning System Based on Genetic Adaptive Algorithms." Ph.D. diss., University of Pittsburgh.

———. 1983. "Flexible learning of problem solving heuristics through adaptive search." Edited by A. Bundy, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* William Kaufmann, 422–425.

Spears, W. 1993. "Crossover or Mutation?" Edited by L. Whitley, *Proc. of FOGA II.* Morgan Kaufmann.

Stanely, K., and R. Miikkulainen. 2002. "The Dominance Tournament Method of Monitoring Progress in Coevolution." *GECCO.*

Stone, P. 1998. "Layered Learning in Multi–Agent Systems." Ph.D. diss., Carnegie Mellon University.

Stone, P., and M. Veloso. 1998, November. "Using Decision Tree Confdence Factors for Multi-Agent Control." Edited by H. Kitano, *RoboCup-97: Robot Soccer World Cup I.* Springer Verlag, 99–111.

Stone, P., M. Veloso, and P. Riley. 1999. "Team–Partitioned, Opaque–Transition Reinforcement Learning." *RoboCup-98: Robot Soccer World Cup II.* Berlin: Springer Verlag.

Sutton, R. S. 1984. "Temporal Credit Assignment in Reinforcement Learning." Ph.D. diss., University of Massachusetts, Amherst, MA.

Sutton, Richard, and Andrew Barto. 1998. *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

Taylor, Matthew, Shimon Whiteson, and Peter Stone. 2006, July. "Comparing Evolutionary and Temporal Difference Methods in a Reinforcement Learning Domain." *Proceedings of the Genetic and Evolutionary Computation Conference.* 1321–28.

Taylor, Matthew E. Shimon Whiteson, and Peter Stone. 2007, July. "Temporal Difference and Policy Search Methods for Reinforcement Learning: An Empirical Comparison." *Proceedings of the Twenty-Second Conference on Artificial Intelligence.* Nectar Track.

Tesauro, G. 1992. "Practicle issues in Temporal difference learning." *Machine Learning* 8:257–277.

Tesauro, G. 1995. "Temporal di.erence learning and TD–Gammon." *Communications of the ACM* 38 (3): 58–67.

Tomoharu Nakashima, Masayo Udo, and Hisao Ishibuchi. 2003, July. "Acquiring Position

Skill in a Soccer Game using a Fuzzy Q-learning." *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation.* Kobe, Japan.

Uchibe, Eiji, Masateru Nakamura, and Minoru Asada. 1999. "Cooperative Behavior Acquisition in a Multiple Mobile Robot Environment by Co–evolution." *Lecture Notes in Computer Science* 1604:273–287.

van Lingen, Bert. 1997. *Coaching Soccer: The Official Coaching Book of the Dutch Soccer Association.* Reedswain.

Wahde, M., and M. Nordahl. 1998. "Coevolving Pursuit–Evasion Strategies in Open and Confined Regions." *Proceedings of the 6th International Conference on Artificial Life.*

Walker, M. 2002. "Evolution of a Robotic Soccer Player." *Research Letters in the Information and Mathematical Sciences* 3 (1): 15–23 (April).

Watkins, C. 1989. "Learning from Delayed Rewards." Ph.D. diss., King's College, Cambridge, UK.

Watkins, C., and P. Dayan. 1992. "Q–learning." *Machine Learning* 8 (3): 279–292.

Whiteson, Shimon, Nate Kohl, Risto Miikkulainen, and Peter Stone. 2003. "Evolving Keepaway Soccer Players through Task Decomposition." *Genetic and Evolutionary Computation Conference (GECCO–2003).*

———. 2005. "Evolving Keepaway Soccer Players through Task Decomposition." *Machine Learning* 59 (1): 5–30 (May).

Whiteson, Shimon, Matthew E. Taylor, and Peter Stone. 2007. "Empirical Studies in Action Selection for Reinforcement Learning." *Adaptive Behavior* 15 (1): 33–50.

Whitley, D. 1989. "The GENITOR algorithm and selective pressure." *Proc. of the Third Int. Conf. on Genetic Algorithms.* Morgan Kaufmann, 116–121.

Whitley, D., and J. Kauth. 1988. "GENITOR: A different genetic algorithm." *Proc. of the Rocky Mountain Conf. on Artificial Intelligence.* Denver CO, 118–130.

Wiegand, R., W. Liles, and K. De Jong. 2001. "An Empirical Analysis of Collaboration Methods in Cooperative Coevolutionary Algorithms." *GECCO.*

———. 2002. "Modeling Variation in Cooperative Coevolution Using Evolutionary Game Theory." *FOGA VII.*

Willmes, L., and T. Bäck. 2003. "Multi–criteria Airfoil Design with Evolution Strategies." Edited by C. Fonseca, P. Fleming, E. Zitzler, K. Deb, and L. Thiele, *Second EMO.* Faro, Portugal, 782–795. (LNCS 2632).

Wilson, Peter. 1998. “Development of a Team of Soccer Playing Robots by Genetic Programming.” Honours Thesis, RMIT, Department of Computer Science.

Wilson, S. 1994. “ZCS: A zeroth level classifier system.” *Evolutionary Computation* 1, no. 2.

———. 1995. “Classifier fitness based on accuracy.” *Evolutionary Computation* 3, no. 2.

Withopf, D., and M. Riedmiller. 2005. “Comparing Different Methods to Speed-Up Reinforcement Learning in a Complex Domain.” *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics.* Big Island, USA: IEEE Press, 3185–3190.

Yao, Jinyi, Jiang Chen, and Zengqi Sun. 2002a. “An application in RoboCup combining Q–learning with Adversarial Planning.” *The 4th World Congress on Intelligent Control and Automation (WCICA’2002).*

———. 2002b. “An application in RoboCup combining Q-learning with Adversarial Planning.” *The World Congress on Intelligent Control and Automation(WCICA’02).*

Yao, Jinyi, Ni Lao, Fan Yang, Yunpeng Cai, and Zengqi Sun. 2003. “Technical Solutions of TsinghuAeolus for Robotic Soccer.” *RoboCup–2003 Symposium.*

Yong, C., and R. Miikkulaainen. 2001. “Cooperative Coevolution of Multi–Agent Systems.” Technical Report AI–01–287, University of Texas.