

Virtual Witches and Warlocks: Computational  
Evolution of Teamwork and Strategy in a  
Dynamic, Heterogeneous, and Noisy 3D  
Environment

A Division III Thesis by  
Raphael Crawford-Marks

School of Cognitive Science  
Hampshire College

May 18, 2004

## **Abstract**

Games make excellent challenge problems for Artificial Intelligence. Two-player turn-based games (Backgammon, Checkers, Chess) are easy to program, and AI players can be benchmarked against humans of varying skill levels. Recently, more complicated real-time team games have received attention from researchers in the Distributed Artificial Intelligence (DAI) and Multi-Agent Systems (MAS) fields because of the dynamic environments and necessity for coordination. The RoboCup Soccer Simulator is the most popular and well-known of these environments. However, the soccer simulator is restricted to only two dimensions, and does not realistically model physics. This Division III thesis describes a simulator of the imaginary game Quidditch, and the automatic programming of quidditch-playing teams by Genetic Programming. These evolved teams of heterogeneous agents have offensive and defensive behaviors, and show the beginnings of real teamwork.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	The Game of Quidditch . . . . .	4
1.2	Evolving Programs . . . . .	4
1.3	Why Quidditch? . . . . .	7
<b>2</b>	<b>Related Work</b>	<b>8</b>
2.1	Multi-Agent Systems . . . . .	8
2.2	Genetic and Evolutionary Computation . . . . .	9
2.3	RoboCup Soccer . . . . .	12
<b>3</b>	<b>Quidditch Simulator and Evolutionary System</b>	<b>14</b>
3.1	Quidditch Simulator . . . . .	14
3.2	Quidditch Evolver . . . . .	25
3.3	Designing a Fitness Function . . . . .	26
<b>4</b>	<b>Experiments and Results</b>	<b>31</b>
4.1	Experimental Setup . . . . .	31
4.2	Results . . . . .	32
<b>5</b>	<b>Conclusions and Future Work</b>	<b>47</b>
5.1	Discussion . . . . .	48
5.2	Future Work . . . . .	50
5.3	Conclusions . . . . .	51
<b>A</b>	<b>Log of Evolutionary Runs</b>	<b>55</b>
<b>B</b>	<b>Source Code, Results, and Movies</b>	<b>59</b>

# Chapter 1

## Introduction

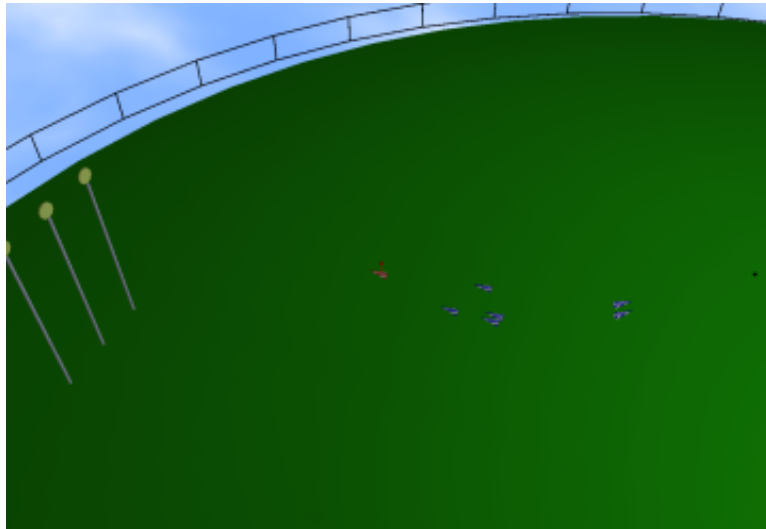


Figure 1.1: Virtual quidditch.

“Now, I want a nice fair game, all of you,” she said, once they were all gathered around her. Harry noticed that she seemed to be speaking particularly to the Slytherin Captain, Marcus Flint, a sixth year. Harry thought Flint looked as if he had some troll blood in him. Out of the corner of his eye he saw the fluttering banner high above, flashing Potter for President over the crowd. His heart skipped. He felt braver.

“Mount your brooms, please.”

Harry clambered onto his Nimbus Two Thousand.

Madam Hooch gave a loud blast on her silver whistle.

Fifteen brooms rose up, high, high into the air. They were off. [1]

Quidditch is the most popular game in the imaginary world of wizardry created by author J.K. Rowling in her series of Harry Potter books [2]. It's a fast paced game, something like a high-flying mix of basketball and soccer. Quidditch is more complex than most real sports, with fourteen players on two teams, four balls of three different types, six goal hoops, and, most significantly, flying broomsticks. Implementing quidditch as a challenge problem for AI, specifically automatically programmed AI, was proposed by Spector et al. in 2001 [3]. This Division III describes a full-featured Quidditch Simulator implemented in a robust physical simulator called BREVE, and the teams of software agents that were evolved to play it.

Using games as a problem domain for Artificial Intelligence is not a new idea. Two-player strategy games like backgammon, checkers, and chess have all been extensively explored problem domains for various AI techniques [4, 5, 6]. Computer Go is currently a very active area of research [7]. In the last twenty years, agent-based AI, known as Distributed Artificial Intelligence (DAI) and Multi-Agent Systems (MAS), has grown in popularity, and researchers have sought out games that better fit the new paradigm. Robot soccer has emerged as one of the most popular game-based problem domains for agent-based AI.

Since the publication of the Robot World Cup Initiative [8], robot soccer has emerged as a challenging and entertaining environment to test the abilities of software agents, and evaluate the strategies we use to program them. However, the RoboCup simulator has some significant drawbacks. It doesn't model any physics save for collisions, and even those are modeled very simply. RoboCup soccer also ignores the third dimension, and in effect simulates a collective air hockey game, not soccer. Virtual quidditch is an ideal candidate to succeed virtual soccer as the next step in complex, dynamic challenge environments for AI research.

Creating a Quidditch Simulator presented some unique challenges. Being an imaginary game, the rules weren't completely described, and some of the imagined rules didn't really make sense once the game was implemented. However, with a few modifications to Rowling's vision, simulated quidditch is as fast-paced and exciting as she described. Evolving quidditch-playing programs was an even more difficult task, fraught with pitfalls both foreseeable and not. This thesis describes the simulator, the evolutionary system, and the results of five evolutionary runs. Virtual quidditch may be the most complex environment in which agent cooperation has been evolved, and shows that there is promise in using evolutionary strategies in tackling complex, real-world problems.

This work has two goals: first, to create a fully functional Quidditch Simulator. The second goal is to evolve cooperative quidditch-playing teams. How does teamwork evolve? How difficult is it to evolve teamwork and strategy in a noisy, dynamic environment? What are common pitfalls and solutions? What evolutionary strategies can or should be employed to achieve good results? Indeed, what qualifies a result as "good?"

The results are far from conclusive, but do give valuable insights in to the direction of future research. When evolving teams of agents who must cooperate in a complex environment, there are many things that must be taken in to consideration, some of which were not obvious ahead of time. Chapter 3 describes obstacles met and overcome in the creation of the Quidditch Simulator. Chapter 4 describes results from a number of evolutionary runs, and the iterative revision process that occurred at each step as new information was revealed.

## 1.1 The Game of Quidditch

The rules of quidditch and the specifics of the Quidditch Simulator are described in detail in Chapter 3. In brief, quidditch is like a mix of soccer and basketball with flying broomsticks thrown in. The game is played on a long oval-shaped field called a Pitch. Two teams of seven players each compete against one another, trying to get a small red ball (the Quaffle) into one of three goal hoops at opposite ends of the pitch. The players move about by means of flying broomstick, which allows them to move about freely in 3-dimensional space.

Scoring in quidditch happens when one of the Chasers (players analogous to forwards in soccer) throws the Quaffle (a red ball made of leather, roughly the size of a basketball) through one of three goal hoops at the opponent's end of the pitch. Goals are worth ten points. The Quaffle is just like any ball, round, bouncy, filled with air. However, one of the most interesting aspects of quidditch is that the other three balls are magically enhanced and semi-intelligent. Two of these are called Bludgers. Bludgers are heavy metal balls that indiscriminantly pummel the players of both teams. The third is the Golden Snitch, a winged ball the size of a golfball. The Golden Snitch is very fast and maneuverable, and very hard to see due to it's speed and small size. Capturing the Snitch is the only way to end a quidditch match, and awards one-hundred and fifty points to the capturing team, almost always assuring a victory.

The details of quidditch gameplay and the specifics of the implementation of the simulator are described in detail in Chapter 3.

## 1.2 Evolving Programs

Most people are familiar with the theory of evolution by the time they get through high school biology. The basic idea is that our physical being (and that of all known life) is encoded into a sequence of genes, forming a long linear sequence known as DNA. When we reproduce, we pass our genes on to our offspring. The genetic material of animals that are successful at making babies will tend to survive from generation to generation, while that of unsuccessful animals will be lost. Therefore, genes that encode advantageous traits (camouflage, night vision, bigger brains, etcetera) will tend to proliferate.

Genes aren't static. A gene is an arbitrary sequence of DNA base pairs, and

sometimes the base pairs within a gene will change for some reason, perhaps radiation or a copy error during cell division. Sexual reproduction mixes the genes of two parents into one offspring. Thus, some degree of genetic variance is always introduced through mutation or sexual recombination. Offspring will tend to be very similar to their parents, but not identical. The results of genetic variance can range from the innocuous (slightly thicker eyebrows) to radical (an extra dorsal fin).

Offspring are subjected to the same law of natural selection, and those that are better at making babies will tend to proliferate, along with their slightly different genes. Over many generations, animals adapt to environments, and develop very specialized mechanisms to improve their baby-making abilities.

Computer Science has abstracted out the key elements of the theory of evolution into a field of research known as Genetic and Evolutionary Computation (GEC). Virtually all GEC methods use the same basic elements: evaluation, selection, and variation. Individuals in a population are evaluated for their ability to solve some problem. Then, specific individuals are selected to populate the next generation based on their fitness. Genetic variation is introduced via mutation, sexual recombination or by some other mechanism before the next generation is evaluated. Then the whole process is repeated until some halting condition is reached.

The type of GEC used for evolving quidditch Programs is called Genetic Programming (GP). It is largely due to Koza [9]. In Genetic Programming, genes are actually functions and terminals written in a programming language (often a LISP or LISP-like language). Genomes are executable programs. GEC and GP are discussed in more detail in Chapter 2.

## Why Evolve Things?

For ages, humans and prehumans have built tools to make the chores of life easier to perform. Over time, those tools have matured from simple manual instruments into machines of varying complexity and automation. Today, there are many fully automated machines that perform tasks faster, more efficiently, and with better accuracy than a human ever could. However, the tasks performed by such machines are often the barest, simplest subtasks of what we humans would really like them to do. Take, for example, a washing machine. In the days of servants and maids, I could tell a human to clean all the dirty clothes in my closet. I would expect within a reasonable amount of time to have all my clothes washed, sorted, and hung back in my closet. In these days of modern conveniences, I must separate my light and dark clothes myself, put each load into the washing machine separately, add the appropriate amount of detergent and/or bleach, and turn on the washer. Even then, the process is at best only half complete. I still have to figure out how to get my clothes dried. And then I have to fold the clothes myself and put them away. This process is much more complicated than telling a servant to wash my clothes.

There are many good reasons for abandoning slavery and indentured servitude. The above example is simply meant to illustrate that household appliances

could be improved significantly. The same is true for our industrial machines. At the dawn of the industrial age, and again at the dawn of the computer age, many believed that technological improvements would finally eliminate menial, dangerous and arduous tasks from the domain of human responsibility. How come we haven't realized this dream? The problem, simply put, is that the real world is very complicated. It turns out that a lot of the abilities we take for granted are very, very difficult to build into a machine. So far, the best we have been able to do is simplify the task to be performed to the point where the messiness of the real world isn't a problem. A washing machine doesn't have to know how to navigate in 3D space, how to pick up and put down objects, or how to tell the difference between a polo shirt and a house cat. A washing machine simply does the following when it is turned on: fill up with water, spin, drain, spin, stop. Even so, the messiness of the real world can confound a washing machine very easily: it will happily spin dry, dirty clothes in the event of a broken water main. More seriously, most washing machines will break if sufficiently overfilled.

The abilities of most machines have plateaued. A washing machine today can do little more than a model from twenty years ago. The reason for this is that building and programming a machine to perform a complex task in an unconstrained environment is often (if not always) an intractable problem for humans. Since the 1940s, Artificial Intelligence and Robotics have engaged the problem of building intelligent machines. At first, AI researchers constructed "intelligent" systems that often had no more intelligence than washing machines. Engaging the Real World was so difficult, that most researchers focused on highly abstract (and very simple) problems. But, as soon as they tried to apply any of their results to the real world, the messiness of the Real World caused everything to fall apart. Finally, researchers have begun using realistic physical simulations and even the Real World as a test environment for their research.

Genetic and Evolutionary Computation is a useful tool for finding solutions to complex, real-world problems. Evolution is a guided search that has shown itself to be an effective problem-solver in the real world. As far as we know, all lifeforms are the product of evolution. GEC saves time and effort by harnessing computer power to find novel solutions to problems. Furthermore, evolution has no preconceptions about what is "good," thus exploring possibilities that a human might not. But its applicability doesn't end there. GEC has proven a very useful tool for learning more about biological evolution and related phenomena. By simulating evolution on computers, we can control certain variables, abstract certain complexities, and (most importantly) speed up the process. For all but the shortest-lived forms of life, it is impossible to study more than a few generations over the course of a lifetime. By using GEC, we can perform investigations that would be prohibitively difficult to do in the real world.

### 1.3 Why Quidditch?

Being an imaginary game, how can quidditch be a good challenge problem for AI? There's certainly no possibility of building quidditch-playing robots and playing them against human opponents, which is the long-term goal for robot soccer. Even so, quidditch is still a very complex, dynamic and noisy environment. In particular, it makes full use of the third dimension and realistically simulates newtonian physics.. Evolutionary strategies and other AI techniques developed for quidditch could be applied to a variety of real world applications like unmanned flight and underwater exploration. Many video games are also richly three dimensional, and the design of computer-controlled adversaries could benefit greatly from this research.

Beyond yielding results that may have practical applications, quidditch offers a very rich and malleable environment in which one can rapidly explore the effects of different changes on an evolving population. RoboCup soccer is limiting in three ways: first, researchers have a preconceived notion of what makes a good team, and may unintentionally bias their systems based on those preconceptions. Second, the soccer simulator is two-dimensional. Third, it has only a simple model for resolving collisions and no real physics engine. Virtual quidditch is richly three-dimensional and accurately and model physics efficiently and accurately.

Quidditch is an imaginary game. No one knows what behaviors contribute to a winning strategy, or how many there are. Furthermore, whole new strategies can be discovered and new adaptive strategies tested by taking advantage of the magical aspect of quidditch. Quidditch exists in a universe of magic. As Kennilworthy Whisp so eloquently states, "Rules are of course "made to be broken" [1]. How do the game dynamics change if there are floating walls that move about the field unpredictably? What AI paradigms compensate best when players can cast vision impairment hexes on one another? Quidditch provides an environment in which researchers are limited only by their imaginations. This is certainly true for the RoboCup soccer simulator as well, since anyone can modify the source code to include "magical" features. But with virtual quidditch, such experimentation is part of the nature of the game.

## Chapter 2

# Related Work

This work is based on and related to research in a variety of fields, particularly that of Genetic and Evolutionary Computation (GEC) and Multi-Agent Systems (MAS). MAS is the study of problem-solving using multiple autonomous agents. GEC encompasses various problem-solving methods that are based on the principles of natural selection. Genetic Programming (GP), an automatic programming method using evolution is the method used to evolve quidditch-playing programs. Also discussed are work on cooperation in MAS, applications of GEC to MAS, and a particular strategy in GEC called coevolution.

### 2.1 Multi-Agent Systems

Multi-Agents Systems is a broad subfield of Artificial Intelligence. It encompasses a wide range of research directions and methods. The unifying element of all research in the field of MAS is its focus on agents: simple, contained objects whose computing power often comes from working together in large groups of similar or identical peers.

Multi-Agent Systems have been used for a wide range of applications: managing network traffic and topologies, webcrawling, artificial life simulations, military applications, and game playing. MAS research into games and game-playing are most relevant to this work.

#### Cooperation in Multi-Agent Systems

One of the main goals of this work is to investigate the evolvability of teamwork and cooperation. However, in order to asses the level of cooperation that evolves, we must first define what it means to be cooperative. This is a philosophical debate, and the definition of cooperation is now and probably will forever be an open question.

In [10], Franklin offers a detailed typology of cooperation in the context of Multi-Agent Systems. He defines a MAS as independent if each agent has it's

own set of behaviors, independent of the other agents. Following Franklin’s typology, an independent MAS can still be cooperative if from the independent agent agendas there emerges cooperative behavior. For example, a group of simple gathering agents that pick up blocks and drop them near other blocks all act independently of one another, but together end up amassing all the block in a big pile (a simple demo of this is included in BREVE).

Continuing with Franklin’s typology, a MAS is cooperative if the agent behaviors include cooperating with agents in some way. This definition is problematic because it is not clear exactly what it means to cooperate. For example, quidditch-playing agents can get the location of teammates and opponents, as well as information on who has possession of the ball. Agents could evolve strategies that rely on information about their teammates (for example, hanging back to guard the goal if a teammate has possession and is trying to score). Agent behaviors are a result of evolution across many generations, with selection based almost exclusively on winning. As such, individual quidditch-playing agents have no agenda of cooperation. At the same time, the agents are scored as a team, and creating an implicit goal of cooperation, at least insofar as not behaving in a way that would hurt the team.

Also in [10] Norman gives a very different and much more restrictive definition of cooperation. Norman paraphrases the Longman’s Dictionary, claiming that “to cooperate is to act with another or others for a common purpose and for common benefit.” Norman goes on to emphasize that to have the same purpose does not make cooperation; agents must *intend* to work together. If we adopt this definition of cooperation, the task of demonstrating cooperative behavior becomes much more difficult; we must somehow show that evolved computer programs with no internal state have intent to work together.

We thus choose to work with Franklin’s definition of cooperation in our examinations of evolved cooperative behavior. In Chapter 5, we discuss the results of various evolutionary runs and find that the evolved quidditch teams are demonstrate only non-communicative emergent cooperation, if any.

## 2.2 Genetic and Evolutionary Computation

Genetic and Evolutionary Computation (GEC) is the umbrella label for problem-solving strategies that unite computing power with key concepts lifted from the theory of evolution. The two largest branches are Genetic Algorithms, due to Holland [11], and Genetic Programming, largely due to Koza [9].

Genetic Algorithms (GAs) follow this basic scheme: create a population of individuals, each with it’s own genotype (often a bit-string or array of floating-point numbers). The genotype is interpreted in some meaningful way related to the problem at hand (e.g. mapped to agent behaviors, coefficients in a polynomial function, parameters needing optimization). The behavior manifested by the genotype is called the phenotype. Often the genomes are initialized to random values at the outset of the evolutionary run. After the population is created, each individual is evaluated by some fitness function, and scored based

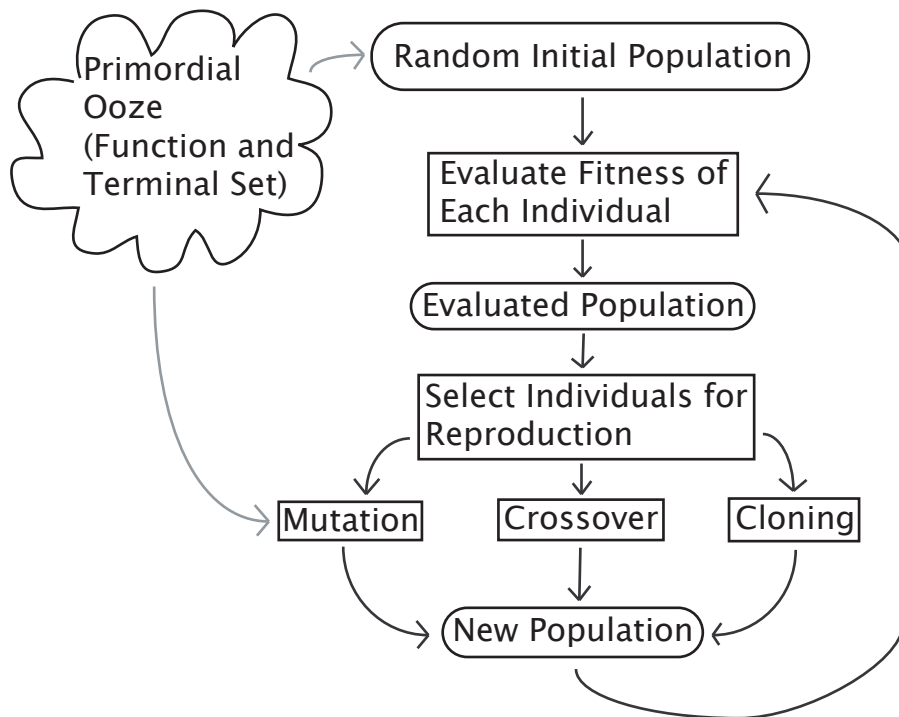


Figure 2.1: The basic structure of a Genetic Programming system. Starting with a population build from randomly selected functions and terminals, individuals are evaluated and selected based on fitness to reproduce into the next generation, using one or more of the genetic operators (crossover, mutation, cloning).

on its performance. Once all individuals have been evaluated, they are probabilistically selected for reproduction based on their fitness scores. The higher an individual scores, the more likely it is to be selected. Selected individuals populate the next generation by cloning themselves. Part of a clones genotype may be mutated to introduce genetic variance. Another way of introducing genetic variance is to have two individuals combine their genetic code into a child. Either way, a new population is created by the more successful (or less unsuccessful) individuals of the previous generation. This new generation is evaluated once again, and the selection and reproduction process is repeated until some exit condition is reached.

Genetic Programming (GP) is a variation of Genetic Algorithms. In GP, the genotype is a computer program. The most popular version of GP evolves programs with LISP-like tree structures and syntax. An advantage of GP is that there isn't any need to map a linear genome to a phenome. Evolved programs are evaluated by running them.

## Coevolution

A key ingredient of GEC is fitness-based selection. In order to perform fitness-based selection, there must be a way to quantitatively evaluate the performance of each individual. In most GP, this is done with a *fitness function*, a chunk of code that evaluates the output of a candidate program on a series of test inputs and returns a numeric score. For example, take the fitness function for polynomial symbolic regression. The function presents a variety of numeric inputs to the candidate program, and sums the square of the difference between the candidate's output and the output of the desired polynomial function. The lower the score returned by the fitness function, the better.

Visualizing GP as a search through an  $n$ -dimensional space of potential solutions, the fitness function can be thought of as creating an  $(n + 1)$ -dimensional fitness landscape, over whose features evolution must travel, avoiding valleys and climbing peaks. The fitness function must be designed such that the landscape it creates is neither deceptive nor too steep. For a problem with only two variables, the fitness landscape would be a 3D surface. In three dimensions, a deceptive landscape is one with volcanoes and islands. A volcano is a hill of better and better fitness that abruptly ends in a steep trough. An island is a hill of high fitness scores surrounded by a valley of poor fitness scores. A steep fitness function is one where changes in the landscape are very sudden from one area in the search space to a neighboring area, making it very difficult for the evolutionary algorithm to find and retain high-scoring solutions.

When dealing with more complex problems or ones that involve competition between two or more players, designing a fitness function that creates a smooth landscape becomes much more difficult. There are two main challenges: first, the behaviors or strategies that result in good performance may not be known. Second, it may be necessary to ramp-up the difficulty of the fitness function in order to keep the fitness landscape from being too steep. In many cases, a very elegant solution to this problem is to create an implicit fitness function via

*coevolution*. In coevolution, individuals are pitted against one another. This creates an implicit fitness function whose difficulty is proportional to the skill level of the population.

Coevolution, like the basic elements of GEC, is also taken from biology. Real biological evolution is a massively parallel process where the fitness (in terms of baby-making abilities) of an individual is determined in the context of their environment, which is largely composed of other evolving individuals. New traits and strategies employed by one species can greatly change the selective pressure on another species, particularly when there is direct interaction between the two, as with a predator/prey relationship. In such a case, a so-called “arms race” can emerge, in which two competing species rapidly (in terms of evolutionary time) evolve new strategies to maintain the upper hand in a fierce competition for survival. Such arms races have been observed in artificial evolution as well. For example, an arms race emerged in a genetic programming experiment that evolved plants of a particular shape and height, which then had to compete for virtual sunlight [12].

Coevolution has been applied to a wide variety of domains, but is a particularly useful strategy in domains where a simple, continuous fitness function is difficult to program. Game-playing and Artificial Life simulations are two such domains. Coevolution has been successfully applied to two-player games like tic-tac-toe [13] and backgammon [14]. In multi-agent domains, coevolution has been used to evolve teams of predators in various predator/prey simulations [15, 16, 17]. More recently, entire teams of soccer-playing agents have been coevolved to compete in the annual RoboCup competition [18, 19].

## 2.3 RoboCup Soccer

The Robot World Cup Initiative (RoboCup) was put forward in the mid-nineties and has as its goal the development of a human-competitive team of soccer-playing robots by the year 2050. Each year, there is a RoboCup World Cup with two divisions: one for physical robots, and another for simulated agents.

The RoboCup Soccer simulator has two parts: a soccer server that provides the field, calculates positions, enforces rules, and keeps score; and a client for each player on the field. Communication between the clients and server is done via UDP/IP protocol. The soccer field is two-dimensional, and players are mobile circles. Physics simulation is greatly simplified in the simulator. For example, collisions are corrected by moving the colliding objects until they no longer overlap, and reducing their velocities by a constant.

The annual RoboCup competition has grown in popularity every year. Many AI strategies have been put to the test in the Robot Soccer domain: stimulus-response, neural networks, genetic algorithms and genetic programming. Team CMUnited, which was champion of the RoboCup simulated league in 1998 and 1999, employed a layered learning approach. Simple agent behaviors were learned first, and then gradually more complicated behaviors were learned with the simple behaviors available to use in the process [20, 21, 22]. The Cyberoos

teams combine a subsumption-style architecture [23] with more symbolic top-down reasoning systems [24].

Genetic Programming was first applied to the RoboCup domain by Sean Luke for RoboCup-97 [18]. He created a function set of hand-coded behaviors from simple (dribbling the ball) to fairly complex (intercepting a moving ball, passing a ball to a moving teammate). Teams were encoded in two different ways: players on *homogeneous* teams all used the same evolved program to coordinate their behavior. Players on *pseudo-heterogeneous* teams were divided into three squads, each with their own evolving program. Each program was also divided into a kick tree and a move tree. The move tree was executed at every timestep. The kick tree was only executed if the soccer ball was within kicking distance from the player. Several modifications had to be made to the RoboCup Simulator Server in order to perform enough evaluations to achieve evolution. At competition time for Robocup-97, the best evolved team (which was homogeneous) was entered, and placed in the middle of the field of competition. This result showed the viability and promise of applying automatic programming techniques to domains as complex as RoboCup.

The following year, Andre and Teller took automatically programmed soccer teams a step further with team Darwin United [19]. Working only with the primitive hand-coded functions, they showed it was possible to evolve a competitive team without significant human assistance. The evolutionary setup for quidditch bears a lot of similarities to Andre and Tellers system. An individual in the evolving population encodes an entire team. Teams are *heterogeneous*, meaning each player is controlled by a separate program so teams are composed of a program for each player. Fitness is scored using a system of “lexical dominance” where certain sub-behaviors are given small fitness rewards, but such rewards are effectively drowned out when teams learn to score goals and win games. Among other innovations not implemented for quidditch (but worth exploring in the future) are team Automatically Defined Functions (ADFs) and sequential fitness filters. These are discussed in Chapter 5.

## Chapter 3

# Quidditch Simulator and Evolutionary System

The Quidditch Simulator is a BREVE simulation. The rules of quidditch as described in [1] are implemented as faithfully as possible in the simulator. However, being an imaginary game, some rules turned out to be unnecessary, while others needed to be changed in order to achieve decent gameplay. The first part of this chapter describes the basic setup of the simulator such as the dimensions of the field, player positions, and rules of play.

The Quidditch Evolver is a separate BREVE simulation that handles the evolution of player and ball teams. The evolver maintains a populations of teams and uses the simulator as its fitness function. The second part of this chapter describes the Quidditch Evolver.

The tandem of the simulator and evolver went through several major revisions as results from previous runs shed light on improvements that could be made and raised new research questions. The refinement process for particular evolutionary parameters and tweaking of specific rules was largely trial and error, as it was unknown exactly what settings would yield a good starting point for evolution.

### 3.1 Quidditch Simulator

The Quidditch Simulator (QS) is implemented in BREVE, a physical simulator with a 3-D graphical frontend [25]. Players are controlled by programs written in the Push programming language [26]. This section describes BREVE, Push, the architecture of the simulator, and some differences in simulated quidditch from Rowling's original imagined version.

INTEGER.+, INTEGER.-, INTEGER.\*, INTEGER./, INTEGER.%,  
 INTEGER.>, INTEGER.<, FLOAT.+, FLOAT.-, FLOAT.\*, FLOAT./,  
 FLOAT.%, FLOAT.>, FLOAT.<, BOOLEAN.AND, BOOLEAN.OR,  
 BOOLEAN.NOT, CODE.DUP, CODE.POP, CODE.SWAP, CODE.=,  
 CODE.RAND, CODE.QUOTE, CODE.ATOM, CODE.CAR, CODE.CDR,  
 CODE.CONS, CODE.DO\*, CODE.IF, CODE.SIZE, CODE.LENGTH,  
 CODE.LIST, CODE.APPEND, CODE.NTH, CODE.NTHCDR, POINT.+,  
 POINT.-, POINT./, POINT.\*, POINT.FROM1FLOAT,  
 POINT.FROM3FLOATS, FLOAT.FROMPOINT, POINT.CROSS

Table 3.1: Base Push instruction set for quidditch

## BREVE

BREVE is a robust 3D environment that greatly simplifies the creation of complex simulations by handling by handling some of the more tedious and complex tasks, such as physics and 3D rendering in real-time [25]. It’s object-oriented scripting language, *steve*, is powerful yet easy to use, and allows the programmer to take full advantage of BREVE’s built-in class hierarchy. Detailed documentation on BREVE as well as the latest BREVE downloads can be found at <http://www.spiderland.org/breve>.

## Push

Push is a stack-based programming language developed by Spector with an eye towards automatic programming, specifically genetic programming. Push programs look very LISP-like, with symbols grouped in balanced (and possibly nested) pairs of parentheses. However, Push programs are interpreted very differently from LISP, and in this respect are more similar to programs in other stack based languages like Forth or Postscript. For a detailed description of the Push programming language, see [27, 26]. The Push instruction set used in the evolutionary runs is shown in Table 3.1.

## The Game of quidditch

### Balls

**The Quaffle** The Quaffle is a round, inflated leather ball, painted red. In the early 18th century, witch Daisy Pennifold enchanted the Quaffle to fall as though sinking through water. The “Pennifold Quaffle” is still used today. In 1875, another enchantment was added to the Quaffle; a “gripping charm” allowing Chaser to easily keep hold of the Quaffle with one hand.

**The Bludgers** Bludgers started out as enchanted rocks, sometimes carved into the shape of a ball. Modern Bludgers are heavy iron balls, 10 inches in

diameter. Bludgers are bewitched to chase the player closest to them. Therefore Beaters must try to knock Bludgers as far away from their teammates as possible.

**The Golden Snitch** The Golden Snitch is a walnut-sized golden ball with thin, translucent wings. It is fast, highly maneuverable and semi-intelligent, employing all its abilities to avoid being caught by either Seeker.

## Players

**The Keeper** Keepers are like soccer goalies. They hover near their own goals to fend off shots from opposing Chasers. Keepers have all the same abilities as Chasers, so they do not exist as a distinct player class in the Quidditch Simulator. Rather, if Keeper-like behavior turns out to be adaptive, then one or more Chasers can evolve to act as a Keeper.

**The Chasers** Chasers are somewhat analogous to soccer forwards. They can hold on to the Quaffle, pass it back and forth, and throw it at the opposing team's goal hoops. Normally there are only three Chasers per team, but in the Quidditch Simulator there are four because there is no Keeper.

**The Beaters** Beaters defend their teammates from the Bludgers. They are equipped with wooden bats to knock Bludgers away from their teammates.

**The Seeker** The Seeker is tasked with capturing the Golden Snitch. They are usually the fastest and most agile player on the team. When the Golden Snitch is caught, the capturing team is awarded one-hundred and fifty points, and the game ends.

## Gameplay

Quidditch is played over an oval-shaped field five-hundred feet long and one-hundred and eighty feet wide. This is called the Pitch. At each end of the pitch are three goal hoops. Quidditch Through The Ages does not specify the height of the goal hoops. In the Quidditch Simulator, they are 15 meters high.

At the start of the game, all players are grounded on their team's half of the pitch. The referee whistles, and the balls are released at midfield. The Quaffle is thrown into the air by the referee. At this point the quidditch match has begun, and does not end until the Snitch is caught or both team captains consent to end the game.

As soon as the referee whistles the start of the game, the Keepers rush to their respective scoring areas to defend the goals (there are no Keepers in the Quidditch Simulator, if this behavior is adaptive then hopefully it will be adopted by one of the four Chasers). The Chasers lift off and scramble after the Quaffle. The Beaters track the Bludgers and assume strategic positions to defend their teammates. The Seekers will often climb high into the air to get a

1. Though there is no limit imposed on the height to which a player may rise during the game, he or she must not stray over the boundary lines of the pitch. Should a player fly over the boundary, his or her team must surrender the Quaffle to the opposing team.
2. The captain of a team may call for “time out” by signalling to the referee. this is the only time players’ feet are allowed to touch the ground during a match. time out may be extended to a two-hour period of a game has lasted more than twelve hours. Failure to return to the pitch after two hours leads to the team’s disqualification.
3. The referee may award penalties against a team. The Chaser taking the penalty will fly from the central circle towards the scoring area. All players other than the opposing Keeper must keep well back while the penalty is taken.
4. The Quaffle may be taken from another player’s grasp but under no circumstances must one player seize hold of any part of another player’s anatomy.
5. In the case of injury, no substitution of players will take place. The team will play on without the injured player.
6. Wands may be taken on to the pitch but must under no circumstances whatsoever be used against opposing team members, any opposing team member’s broom, the referee, any of the balls, or any member of the crowd.
7. A game of quidditch ends only when the Golden Snitch has been caught, or by mutual consent of the two team Captains.

Table 3.2: Rules of quidditch [1]

good view of the whole pitch. They circle around the pitch until spotting the Snitch, at which point they go into high-speed pursuit.

See Table 3.2 for a list of quidditch Rules. There are a number of fouls in quidditch, some of which are described in *Quidditch Through The Ages*. Figure 3.3 lists the fouls described in *Quidditch Through The Ages*.

### **Changes to quidditch in the Quidditch Simulator**

A number of small changes have been made to the setup of the quidditch field and the rules of quidditch. These changes were made for a variety of reasons: to promote more balanced gameplay, to facilitate evolution, and to reduce simulator complexity.

**Starting Positions** Balls and players do not start on the ground. This was done primarily because it was sometimes difficult for the Snitch to escape the

Name	Applies to	Description
Blagging	All Players	Seizing the opponent's broom tail to slow or hinder
Blatching	All players	Flying with intent to collide
Blurting	All Players	Locking broom handles with a view to steering opponent off course
Bumpling	Beaters only	Hitting Bludger towards the crowd, necessitating a halt of the game as officials rush to protect bystanders. Sometimes used by unscrupulous players to prevent an opposing Chaser scoring
Cobbing	All players	Excessive use of elbows towards opponents
Flacking	Keeper only	Sticking any portion of anatomy through goal hoop to punch Quaffle out. The Keeper is supposed to block the goal hoop from the front rather than the rear.
Haverstacking	Chasers only	Hand still on Quaffle as it goes through goal hoop (Quaffle must be thrown)
Quaffle-packing	Chasers only	Tampering with Quaffle, e.g., puncturing it so that it falls more quickly or zigzags
Snitchnip	All players but Seekers	Any player other than Seeker touching or catching the Golden Snitch
Stooging	Chasers only	More than one Chaser in the scoring area

Table 3.3: Common quidditch Fouls [1]

Seekers if it started from ground level. See Figure 3.1 for screenshots of the starting positions of the players and balls. The Snitch does not have a set starting position, but is instead placed randomly on the field, at least 10 meters from the nearest Seeker.

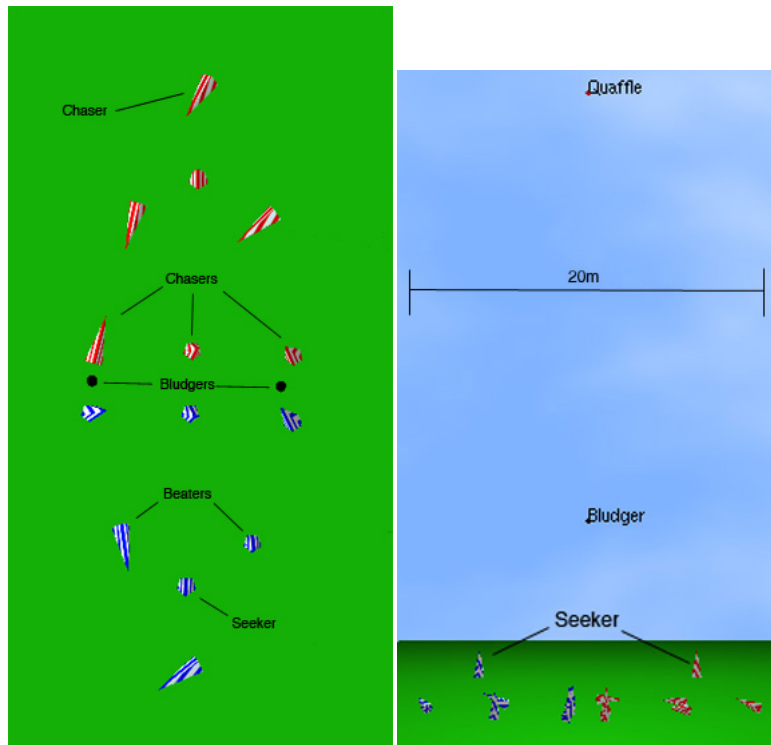


Figure 3.1: Quidditch Simulator starting positions.

**Pitch Size and Shape** The pitch is circular instead of oval, with a radius of 85 meters. The circular shape was chosen simply because the BREVE Shape class can be initialized as a circular disk. Creating an oval would have required a bit of extra programming and didn't seem to benefit the game dynamics. Any mobile object moving farther than 85 meters from the center of the pitch (including the vertical Y axis) is gently "bounced" back towards midfield. It was necessary to create this boundary to prevent players or balls from climbing infinitely high (the Snitch was particularly fond of this strategy).

**Goal Shapes and Scoring** Goals are solid disks instead of hoops. Goals are scored when the Quaffle collides with the Goal disk. In BREVE it is not possible to create concave shapes. Collision detection is elegantly handled by BREVE,

whereas detection of the Quaffle passing through a hoop would have been much more difficult to program.

Each team has one extra Chaser. This is because Keepers (as described in Quidditch Through The Ages) are simply Chasers that elect to defend instead of attack. This being the case, there was no reason to create a separate Keeper class. If it is adaptive to have a Keeper then hopefully one or more Chasers will evolve defensive behaviors. Indeed, some very simple defensive behavior was observed during each of the three large evolutionary runs in which one of the four Chasers would fall back to the center goal and orbit it, occasionally intercepting shots from the opposing team.

The value of catching the Snitch has also been modified. Instead of being worth one-hundred and fifty points, capturing the Snitch is worth  $10 + (2 * score)$  (up to 150) points for the capturing team. This change was made because teams were evolving to *only* chase the Snitch, completely ignoring the Quaffle.

**Ending the Game** As imagined, quidditch games do not end until the Snitch is caught or by mutual agreement of both team captains. This is unworkable for artificial evolution. Even with the significant speedup over real-time provided by the BREVE engine, games that last twelve simulated hours or longer would slow evolutionary runs to a crawl. Also, it is often easy to judge which is the better team very quickly, especially at the beginning of a run when many teams don't even move.

There are three conditions which will end a game in the Quidditch Simulator. First is the Snitch being caught. Second, there is a score limit of 200. Third, there is a variable time limit. The value of the limit is specified by a command-line parameter. This allows the Quidditch Evolver to specify a time limit proportional to the generation of the run when calling the Quidditch Simulator. At the beginning of runs, games are limited to 10 simulated seconds (2 or 3 real seconds). As runs progress, the time limit is set to  $10 + (generation/4) * 5$  seconds.

## The Simulator Architecture

The Quidditch Simulator makes heavy use of inheritance to implement the various actors in the simulation. All mobile objects in the simulation are subclasses of the QMobile class, which in turn is a subclass of the BREVE's built-in Mobile class. A detailed documentation of BREVE's class hierarchy can be found at <http://www.spiderland.org/breve/docs/>. The immediate subclasses of QMobile are QBall and QPlayer which implement properties specific to balls and players, respectively. Each ball type is a subclass of QBall: Quaffle, Bludger and Snitch. Each player type is a subclass of QPlayer: Chaser, Beater and Seeker. According to Quidditch Through The Ages, Keepers have all the same permissions and abilities as Chaser, but simply defend the goal instead of attacking. Thus, Keepers are not implemented as a separate subclass of QPlayer in the simulator.

Game State	Meaning
STATE_INGAME_POSSESSION_TEAM1	Team One has possession of the Quaffle
STATE_INGAME_POSSESSION_TEAM2	Team Two has possession of the Quaffle
STATE_INGAME_LOOSE_BALL	Quaffle is loose
STATE_INGAME_BALL_THROWN	Quaffle has been thrown
STATE_INGAME_GOAL_SCORED	Goal was just scored
STATE_PREGAME	Game has not yet started
STATE_GAMEOVER	Game is over

Table 3.4: quidditch Game States

Player objects are instantiated by a QuidditchTeam object, balls by a QuidditchBalls object. The QuidditchTeam/Balls classes are subclasses of the BREVE Abstract class. The QuidditchTeam/Balls classes handle things like initializing the players or balls, placing them at the correct starting locations, returning information about them when queried, propagating events, and reading and loading Push programs from the filesystem.

Other important simulation objects are Goals, the ScoreTracker, and the generically-named Field object, which acts as the simulation controller. The Goal class is a subclass of the BREVE Stationary class. Each Goal object (three for each team - six total) is initialized by an instance of class Goals, a subclass of Abstract that does for goals what QuidditchTeam does for players. The ScoreTracker class is also a subclass of Abstract. The ScoreTracker performs many of the same functions that a scoreboard would at a basketball or football game. It keeps track of the score, which team has possession, the state of the game (The list of game states can be seen in Table 3.4), and how much time is left. ScoreTracker also writes the fitness score of each team to a file after the game is over.

Field is a subclass of BREVE's PhysicalControl class. Upon initialization, Field creates two instances of the QuidditchTeam class, one instance of QuidditchBalls, one of Goals, and one of ScoreTracker. It also creates the Pitch (the quidditch Playing field) and sets a number of camera and graphics parameters.

Communication of game data between objects is facilitated by the simulation controller through use of Event objects. Whenever something occurs on the field, such as a player catching the Quaffle, or a goal being scored, the object involved with the even initializes a specific subclass of Event and passes it to the controller to be broadcast to all objects in the simulation. Every class in the simulation has a `handle` method which is called whenever an event is broadcast. The handle method checks the event type to see if the event is relevant, and if so executes some code in response to the information contained within the event.

For example, if a Goal object detects a collision with a Quaffle, it generates

Function Name	Description
ally-Chaser01, ally-Chaser02, ally-Chaser03, ally-Chaser04	Vector to ally Chasers.
ally-Beater01, ally-Beater02 ally-Seeker01	Vector to ally Beaters. Vector to ally Seeker.
opp-Chaser01, opp-Chaser02, opp-Chaser03, opp-Chaser04	Vector to opposing Chasers.
opp-Beater01, opp-Beater02 opp-Seeker01	Vector to opposing Beaters. Vector to opposing Seeker.
Quaffle01	Vector to Quaffle.
Bludger01, Bludger02	Vector to Bludgers.
Snitch01	Vector to Golden Snitch.
my-goal01, my-goal02, my-goal03	Vector to own goal hoops.
opp-goal01, opp-goal02, opp-goal03	Vector to opponent goal hoops.

Table 3.5: Absolute sensor functions for players.

a GoalScored event, and stores its ID in the GoalScored event. In the ScoreTracker’s `handle` method, if the event object is of type GoalScored, then the scoretracker gets the ID of the goal that generated the event, finds out to which team it belonged, and then credits 10 points to the opposing team.

## Sensors and Actuators

Agents interact with the quidditch world through sensors and actuators. Actuators give agents the ability to act within the world. All agents (players and balls except the Quaffle) are equipped with the same simple actuator. They have an invisible thruster which can be instantaneously pointed in any direction to propel the agent. The force exerted by the thruster is variable up to a set maximum. Agent speeds are also limited to about 50 kilometers per hour in order to keep collision calculations reasonable.

Sensors provide information about the state of the world. Players are equipped with noisy omnidirectional radar. They can sense anything in the game world, but the accuracy of the information they get from their sensor falls in proportion to their distance from the object being sensed. Within 25 meters, no noise is added to the sensors. Over 25 meters, mean 0, standard deviation ( $distance/5$ ) gaussian noise is added to each component of the sensed vector location.

For this research, players were controlled by Push programs. Custom Push functions that return information about the game world were created and added to the terminal set to provide agents with information about the world. The Push terminals that provide sensory information are listed in Tables 3.5, 3.6,

Function Name	Description
<code>my-object-of-interest</code>	Vector to my object of interest. Snitch for Seekers; nearest Bludger for Beaters; Quaffle or nearest goal for Chasers, depending on possession status.
<code>nearest-ally-Chaser</code>	Vector to the nearest ally Chaser.
<code>nearest-ally-Beater</code>	Vector to the nearest ally Beater.
<code>nearest-ally-Seeker</code>	Vector to the nearest ally Seeker. (identical to <code>ally-Seeker01</code> ).
<code>nearest-opp-Chaser</code>	Vector to the nearest opposing Chaser.
<code>nearest-opp-Beater</code>	Vector to the nearest opposing Beater.
<code>nearest-opp-Seeker</code>	Vector to the nearest opposing Seeker. (identical to <code>opp-Seeker01</code> ).
<code>nearest-Bludger</code>	Vector to the nearest Bludger.
<code>nearest-goal-to-defend</code>	Vector to the player's nearest own goal.
<code>nearest-goal-to-attack</code>	Vector to the player's nearest target goal.

Table 3.6: Relative sensor functions for players.

Function Name	Description
<code>possession-time-left</code>	Time left until player is forced to relinquish possession of the Quaffle.
<code>possession-ally-team</code>	True if self or ally has possession of the Quaffle.
<code>possession-opp-team</code>	True if a player on the opposing team has possession of the Quaffle.
<code>possession-self</code>	True is player has possession of the Quaffle.
<code>possession-ally-Chaser01</code> , <code>possession-ally-Chaser02</code> , <code>possession-ally-Chaser03</code> , <code>possession-ally-Chaser04</code>	Possession status (True or False) of each ally Chaser.
<code>possession-opp-Chaser01</code> , <code>possession-opp-Chaser02</code> , <code>possession-opp-Chaser03</code> , <code>possession-opp-Chaser04</code>	Possession status (True or False) of each opposing Chaser.
<code>possession-nearest-ally-Chaser</code>	True if nearest ally Chaser has possession of the Quaffle.
<code>possession-nearest-opp-Chaser</code>	True if nearest opposing Chaser has possession of the Quaffle.
<code>ball-thrown</code>	True if ball has been thrown.
<code>ball-loose</code>	True if ball is loose.
<code>scoreDif</code>	Current score difference.

Table 3.7: Omniscient sensor functions for players.

Push Function	Description
<code>randI</code> ,	Random integer.
<code>randF</code> ,	Random float.
<code>randV</code> ,	Random vector.
<code>randC</code> ,	Random Push code.
<code>close-to</code> ,	True if player is within $n$ meters of a specified vector location.

Table 3.8: Utility functions for players.

Push Function	Description
<code>first-object-of-interest</code>	Vector to first object of interest.
<code>second-object-of-interest</code>	Vector to second object of interest.

Table 3.9: Sensor functions for balls.

Push Function	Description
<code>move</code>	Applies movement force with direction and magnitude specified by the top of the vector stack.
<code>move-as-fast-as-possible</code>	Applies maximum movement force in direction specified by the top of the vector stack.
<code>throw</code>	Throws the Quaffle in the direction specified by the top of the vector stack. (Chasers only)
<code>queue-throw</code>	Pushes <code>throw</code> instruction onto the code stack. (Chasers only)
<code>queue-move</code>	Pushes <code>move</code> instruction onto the code stack.
<code>flee</code>	Like <code>move</code> but in opposite direction. (Balls only)

Table 3.10: Actuator functions for balls and players.

3.7 and 3.9.

## 3.2 Quidditch Evolver

The Quidditch Evolver is separate from the Quidditch Simulator, but is also written in BREVE. The Quidditch Evolver (QE) implements all the functionality of a traditional GP system, minus fitness evaluations (which are performed by the Quidditch Simulator). The QE initializes the ball and player team populations, calls the Quidditch Simulator to obtain fitness scores, performs fitness-based tournament selections on the population, and creates a next-generation population using mutation, crossover, trade, and reproduction operators.

At startup, the QE initializes two populations, one of teams of players and another of teams of balls. In both cases, an entire team (containing a multiple Push programs) is a single individual in the population. Both populations have the same number of individuals.

Once initialized, the QE iterates through the populations of player and ball teams, placing the current teams into a pool with two or more randomly selected player teams. It then coordinates a round robin between all the teams in the pool. For example, a round robin with player teams { p1, p2, p3 } and ball team { b1 } would be played out in 3 games: { {p1, p2, b1}, {p1, p3, b1}, {p2, p3, b1} }. Note that every player team plays at least  $n - 1$  games, where  $n$  is the pool size. On average, every player team plays  $n$  games. Ball teams always play only  $n - 1$  games. At the end of each game, the QE gathers fitness data from an output file left by the Quidditch Simulator. It then passes the player and ball programs to the simulator via the filesystem.

Once evaluation has been performed, the QE probabilistically selects individuals based on fitness to populate the next generation. There are four genetic operators that facilitate this: crossover, mutation, trading, and reproduction. These genetic operators are applied to each member of the team separately, meaning that member A of a team may be the product of mutation, while member B may be the product of crossover, C of reproduction, and so on. Reproduction simply clones the selected individual and places the clone in the new population. Trading swaps the entire program tree for one member with the program tree of a member of a different team. Crossover mixes part of the program tree of one member with part of the program tree of another member. Mutation is like reproduction, except that a randomly chosen subtree of the clone is replaced by a subtree of random code.

Crossover and trading is restricted to programs of the same type. Chasers may only crossover with other Chasers, Beaters with Beaters, and so forth. Each PushTeam is a collection of Push program trees, one for each member of the team (see Figure 3.2). A team of players has seven program trees, a team of balls has three. For a team of players, trees in slots 0 through 3 control the Chasers, 4 and 5 control Beaters, and 6 controls the Seeker.

After repeatedly selecting individuals based on their fitness and facilitating their reproduction with the four genetic operators, a new generation is created, ready to be evaluated in round robin play. The QE repeats this process until a specified generation limit is reached.

Every time a new generation is created, two player teams and one ball team are selected via tournament selection and written to a shared directory. They are then replaced by reading in other player and ball teams from the shared directory. This immigration mechanism allows for sharing of genetic code between evolutionary runs of different nodes of a computer cluster.

Every ten generations starting at generation 20, a tournament of champions is held. A tournament of champions provides useful insight into the progress of a coevolutionary run. Without a monotonic fitness function, it is sometimes hard to tell if there is actual improvement across generations, or if evolution is stuck in a cycle of suboptimal behaviors, each better than some and worse than others. It also helps reintroduce old behaviors which may have gone extinct, but can now exploit a new fitness environment. The tournament of champions is simple: it places the best of generation player teams from generation 0, 10, 20, ...,  $n$  into a round robin pool with the best current ball team. After every team has played every other team, the one with the highest average score is inserted into the current generation. The tournament of champions is done only with player teams.

### 3.3 Designing a Fitness Function

In “traditional” GA and GP problems, the solution is either known or a function can be designed to determine if the solution has been found or how far off the candidate is from the solution. For example, the fitness function for most

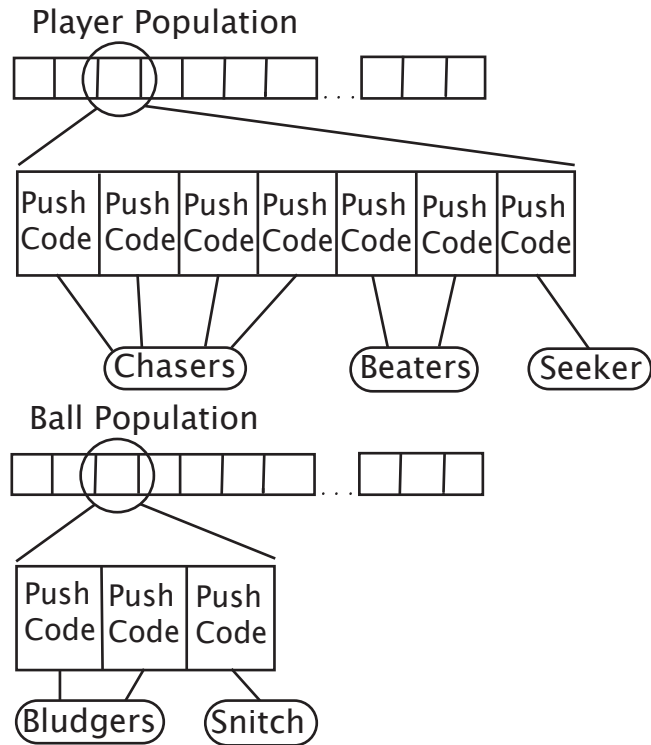


Figure 3.2: Each individual in the population is an entire PushTeam.

symbolic regression problems simply calculates the discrepancy between the program output and the expected output. The higher the discrepancy, the worse the fitness. A discrepancy of zero means a solution has been found.

For slightly more complex problems, such as the backer-upper, artificial ant, and lawnmower problems described in [9, 28], it is still easy to write a simple fitness function that will confer better fitness scores the better the performance. For example, the fitness function for the backer-upper problem confers higher fitness scores inversely proportional to the ending distance of the back of the truck from the loading dock. Once a program has been discovered, one could further optimize behavior by rewarding programs that reach the loading dock in the fewest number of time steps. In either case, it is still easy to write a straightforward fitness function that will provide a smooth fitness landscape for the evolving population to explore.

The first problem one encounters when trying to evolve solutions to game-based problems is how to write a fitness function that provides a smooth fitness landscape. Take checkers as an example. To score an evolving population of checkers-playing programs, the fitness function could be a moderately skilled

shareware checkers program. Immediately we run into a big problem: at the outset of an evolutionary run, the individuals of the population are randomly generated programs. In all likelihood, none of the programs will have a prayer of beating even a moderately skilled AI opponent. This creates a fitness cliff that is impossible for programs to climb.

Learning from this mistake, we might instead use a canned AI opponent with a variable skill level, and ramp up the difficulty automatically once a certain percentage of the population can score victories. An issue with this is finding a difficulty setting that is easy enough to smooth out the fitness cliff without accidentally rewarding simplistic and suboptimal strategies over the beginnings of more complex and subtle ones. Even if we overcome this obstacle, we are still faced with another difficulty: what to do when the evolved programs overmatch the AI adversary?

Often, in these types of situations, the best solution is to use *coevolution*. Coevolution does away with explicit fitness functions. Instead, programs compete against one another, creating an implicit fitness function that increases in difficulty as the skill levels of the population increase. Coevolution has shown to be a very effective method, and has been applied in a wide variety of domains, including two-player games [14], predator/prey simulations [15, 16, 17], and the RoboCup Soccer Simulator [18, 19].

In all of the example problems described in the beginning of this section (except the artificial ant problem), fitness is determined by the performance of a single individual. For the artificial ant scenario, fitness is determined by the performance of a team of identical agents. However, the difference is largely superficial, because it is still the performance of one program that is ultimately earning a fitness score, even though that program is being run by multiple autonomous agents. Some very interesting challenges relating to credit assignment arise when fitness is determined by the collective actions of heterogeneous agents. How can the contributions of individual agents be rewarded in proportion to their value? When evolving a pack of predator agents, it may be adaptive for several agents to corral the prey while only one goes in for the kill. It is impossible to quantitatively measure the contributions of each agent. This is known as the *credit assignment problem* [29].

An individual genome in the Quidditch Evolver is actually a collection of distinct program trees for each player on the team. That is, a single genome contains seven Push programs, each one controlling one of the Chasers, Beaters or Seeker. At the end of an evaluation, the team receives a score based on its performance. This effectively addresses the credit assignment problem.

However, there still exist other complications. The ultimate goal of the Quidditch Evolver is to evolve teams that play quidditch well. Thus, one possible method of determining fitness would be to base it entirely upon wins and goal differential. However, at the start of many runs, no team ever scores a goal. Unfortunately this simple method creates a fitness cliff that must be smoothed out.

One way of smoothing out this cliff is to have certain agent behaviors contribute small fitness bonuses to the overall score. Behaviors like moving, throw-

ing the Quaffle, beating Bludgers, etc... contribute tiny amounts to the fitness score. There are two pitfalls that must be avoided when using this method: first, rewarding certain “good” behaviors can bias the evolutionary system towards convergence at a local optimum. For example, an early version of Quidditch Evolver rewarded teams by a small amount proportional to the possession time. Though the reward was tiny, it was enough to bias the system toward evolving a strategy of keep-away, and no goals were ever scored. The second and more subtle pitfall is the possibility that there are adaptive behaviors that escape the imagination of the designers, and that small biases towards certain behaviors might very well lead away from other novel, adaptive ones.

The fitness function that was finally implemented is very similar to the system of “lexical dominance” used by Andre and Teller [19]. This type of fitness scoring was researched in depth under the name Fitness Switching (FS) and Coevolutionary Fitness Switching (CFS) by Zhang et al. [30]. CFS is a means of augmenting GP for the evolution of complex behaviors. Using this strategy, individuals receive final fitness score that is the sum of the output of  $n$  micro fitness functions, each rewarding a specific micro-behavior. A micro-behavior is some simple behavior that, when used with other micro behaviors, yields the desired macro behavior. With simple multi-agent problems, it is possible to exhaustively identify every micro behavior that is needed to achieve the desired macro behavior, and thus create a comprehensive set of micro fitness functions. Zhang and Cho (in [30]) call this sequential evolution, because each behavior can be evolved in sequence, up to the optimal global behavior.

With problem domains as complex as Robotic Soccer and quidditch, it is not possible to design a sequence of fitness functions comprehensive enough to achieve sequential evolution. Furthermore, quidditch is an imaginary game, making it is impossible to know exactly what micro-behaviors are adaptive, or what the eventual macro-behavior should look like. The strategy adopted in quidditch uses CFS to bootstrap evolution to certain point, after which the only way to achieve higher fitness scores is by scoring goals.

### **Player Fitness Scoring**

Here are the fitness scoring algorithms for players and balls. The important thing to note about this method of fitness scoring (particularly for players) is that the small bonuses conferred for mobility, touching the Quaffle, beating, etc... are effectively drowned out as soon as scoring occurs in the game. This helps evolution by giving little fitness “nudges” in the right direction, without changing the game over which they are being optimized in any significant way.

**Touching the Quaffle** The team receives a bonus of 0.1 fitness points if one of the Chasers touches the Quaffle during the game.

**Possession Time** The team receives 0.01 fitness points for every ten timesteps that the Quaffle is in held by one of the Chasers. The maximum bonus for possession time is 0.2 points. This bonus was not added until Run 4.

**Throwing** Each Chaser contributes 0.05 fitness points if it throws the ball at some point during the game. If each Chaser throws the ball, the team receives a fitness bonus of 0.2 points. This bonus was removed for Runs 2 and 3.

**Mobility** Each player contributes a 0.1 fitness points if it is mobile (i.e. it executes `move` or `move-as-fast-as-possible` with a nonzero vector). If all seven players move, the team receives a fitness bonus of 0.7 points.

**Beating** Each Beater contributes 0.01 fitness points for every 2, 4, 8, ..., (power of two) times they beat a Bludger. The maximum contribution from each Beater is 0.5 points, but in practice it the bonus is rarely more than about 0.15 (about 0.3 from both Beaters combined).

**Score Difference** The score difference at the end of the game is added on to the teams fitness score. This can be positive, negative or zero. Except in cases where the difference is zero, the goal difference will be a multiple of 10.

**Team Score** The team score (goals plus Snitch bonus, if it was caught) is added on to the fitness score. This is always zero or a positive multiple of 10.

### **Ball Fitness Scoring**

**Mobility** No bonus is given to teams for having mobile players. Rather, for each immobile ball the team receives a fitness penalty of -10 points. If none of the balls are mobile, the team receives a total penalty of -30 points.

**Bludger Hits** Each Bludger contributes 0.5 fitness points for every 2, 4, 8, ..., (power of two) hits they score on non-Beater players, up to a maximum of 10 points each (20 points total).

**Snitch Caught** The team is penalized -20 points if the Snitch is caught.

# Chapter 4

## Experiments and Results

### 4.1 Experimental Setup

Evolutionary runs were performed on a 13-node computer cluster at Hampshire College. Each node is a linux machine with dual Athlon 1.2 GHz processors. A separate instance of the Quidditch Evolver was run on each node, with randomly selected individuals “emigrating” to other nodes via a shared file system. Even with the considerable computing power available, each of the runs described below took over 48 hours to reach completion.

The base function and terminal set used is the union of the Push function set (see Table 3.1) plus all quidditch-specific sensor, utility and actuator functions (Tables 3.5, 3.6, 3.7, 3.8, 3.9 and 3.10). Table 4.1 shows the different Push interpreter settings used by the Quidditch Evolver and Simulator. The specific settings of these parameters are described in the next section, along with the results of each run.

Parameter	Meaning	Initial Value
MAX_RANDOM_CODE_SIZE	Specifies the maximum size of a randomly generated program.	30
MAX_RANDOM_TREE_SIZE	Specifies the maximum size of a randomly generated subtree.	20
MAX_CODE_SIZE	Maximum size of a program.	60
Push_EXECUTION_LIMIT	Maximum number of execution steps before the interpreter stops.	100

Table 4.1: Push Interpreter Settings for quidditch.

## 4.2 Results

### Run 1

#### Player Teams

The initial random teams performed very poorly. Many players didn't move, and those that did often wandered randomly or chased their teammates. However, the code stack of each player was seeded with ( `move` ) and ( `throw` ), and the vector stack was seeded with `my-object-of-interest`. Therefore, simply executing `move`, `move-as-fast-as-possible`, or `CODE.DO` would cause a player to pursue its object of interest (Chasers would pursue the Quaffle, Beaters the nearest Bludger, Seekers the Snitch). For Chasers, if `throw` or a second `CODE.DO` appeared in their code, they would pursue the Quaffle and throw it at the nearest opponent goal. Indeed, some of the best of Generation 0 programs contain one or two players that do this. Another adaptive behavior exhibited by some of the best teams from Generation 0 is a Seeker that relentlessly pursues the Snitch. Most Generation 0 Snitches have poor avoidance behavior, if any, and are easily caught by simply charging it.

By Generation 10, most of the good teams have one or two Chasers who perform the “chase and throw” behavior, and a Seeker that actively pursues the Snitch. Occasionally, a Beater will pursue Bludgers as well. The ball teams have improved significantly by Generation 10, and the best ones will have a Snitch that is hard to catch, and two Bludgers that aggressively attack the nearest players.

Around Generation 20 or so, ball teams have evolved good behaviors. Both Bludgers attack nearby players, while the Snitch is almost uncatchable. The latter forces player teams to score goals in order to win, since they cannot rely on their Seekers. Here is where “kiddie-quidditch” really begins to emerge. The name is inspired by Luke et al.'s observation of “kiddie-soccer” behaviors in [17]. Essentially, the evolved Chasers play quidditch much like six-year-olds play soccer. They all charge the ball and throw it towards the goal, with no regard for defense. As in [17], kiddie-quidditch is a very attractive suboptimal behavior, discovered fairly quickly by every evolutionary run. Many of the adjustments to various system settings from run to run were done in an attempt to help the population escape from kiddie-quidditch into more fertile areas of the search space.

No defensive behaviors or new offensive strategies emerged, save one slight modification to the kiddie-quidditch strategy. Around Generation 30, one of the four Chasers would hold on to the ball and carry it to the opposing goal. There, it would orbit the goal until the ten second time limit on possession expired and it was forced to drop the Quaffle. The other three Chasers would still simply throw the ball towards the goal. This appears to be advantageous because it brings the Quaffle very close to the opposing team goals. Because of sensor noise, long-range shots are often wildly inaccurate. When the Quaffle is dropped near the goal by the ball-holder, it is so close that any shot at the goal

is almost always 100% accurate. Perhaps, had the run been allowed to continue for more generations, the ball-holding behavior would have evolved a control structure (perhaps using `close-to`) which would cause the player to hold on to the Quaffle until it was within a certain range of the opposing teams goal hoops.

Results from the tournament of champions shows a steady increase in player team performance until the last twenty generations, when relative fitness scores level off (see Table 4.3. This is evidence that kiddie-quidditch is a strong enough local optimum to halt evolutionary improvement in its tracks.

### Ball Teams

All balls move at the beginning of evolution, though not always because their own code instructs them to do so. If the simulator detects that a ball is not moving, control of the ball is handed over to a bit of hand-written code, and the ball team is hit with a huge fitness penalty. This is done so that the Snitch will never be stationary. While the simulator does force balls to move, it does not force them to move intelligently. Some of the early-generation Snitches would fly right at Seekers, instantly ending the game. Some Bludgers would chase after Beaters getting repeatedly whacked all over the pitch.

By Generation 10, all three balls on the best teams have evolved halfway decent, if very simple, behaviors. Those behaviors run along the lines of mindless pursuit and mindless flight. The two Bludgers will simply accelerate directly at the nearest (non-Beater) player, while the Snitch accelerates directly away from the nearest Seeker. While the relentless pursuit of Bludgers is quite effective, relentless flight turns out to be somewhat problematic. The Snitch often only flees the nearest Seeker, sometimes flying directly towards the other Seeker and not realizing it until too late. Other times, the Snitch will flee and flee and flee in one direction until it bumps up against the “invisible bubble” border of the world, and is bounced right in to one of the Seekers pursuing it.

In later generations, Bludgers appear to continue with the relentless pursuit strategy. Since they are scored based on the number of collisions with non-Beater players, relentless pursuit turns out to be a fairly effective strategy. The Snitch, on the other hand, evolves a more complex avoidance algorithm that takes in to account the locations of both Seekers, and doesn't simply flee mindlessly.

The average fitness of ball teams for every generation is a good indication of how often Snitches are being caught. Ball teams receive a fitness hit of 20 points if the Snitch is caught, which in practice will always result in a negative overall fitness score. Therefore, negative average fitness scores can be evidence of frequent Snitch-catchings. Figure 4.5 shows the overall average ball team fitness score across the entire run. As you can see, by Generation 25, average fitness scores go positive and never turn back.

Generations	52
Mutation Rate	40%
Crossover Rate	40%
Trade Rate	10%
Reproduction Rate	10%
Within-team Xover Probability	0.1
Pool Size	3
Tournament Size	5
Population Size (per Node)	30
Score Limit	100
Immigration	Yes
Nodes	12
Individuals Evaluated	18,720

Table 4.2: Settings for Run 1.

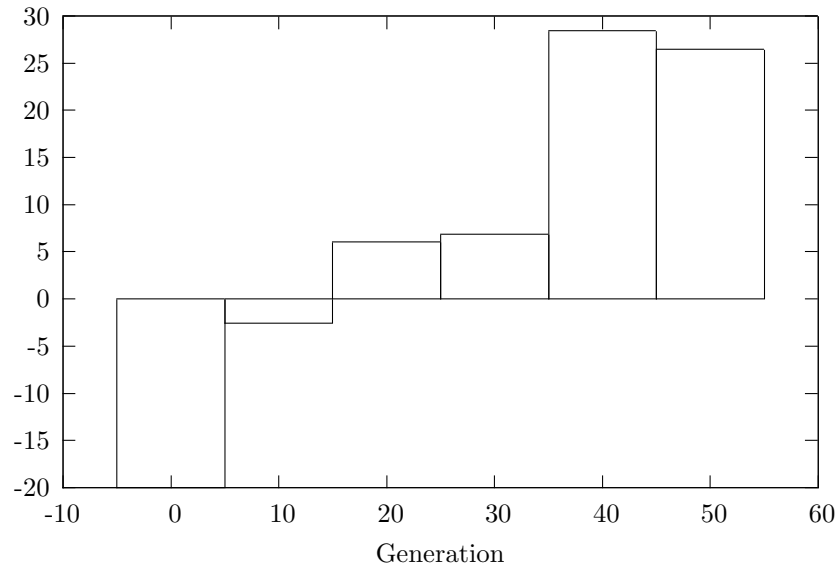


Table 4.3: RUN 1: Average Relative Fitness in the Generation 50 Tournament of Champions. This shows the average fitness score (as determined by the fitness function described in section 3.3) for each team after playing in a round robin pool composed of the best player teams of every tenth generation.

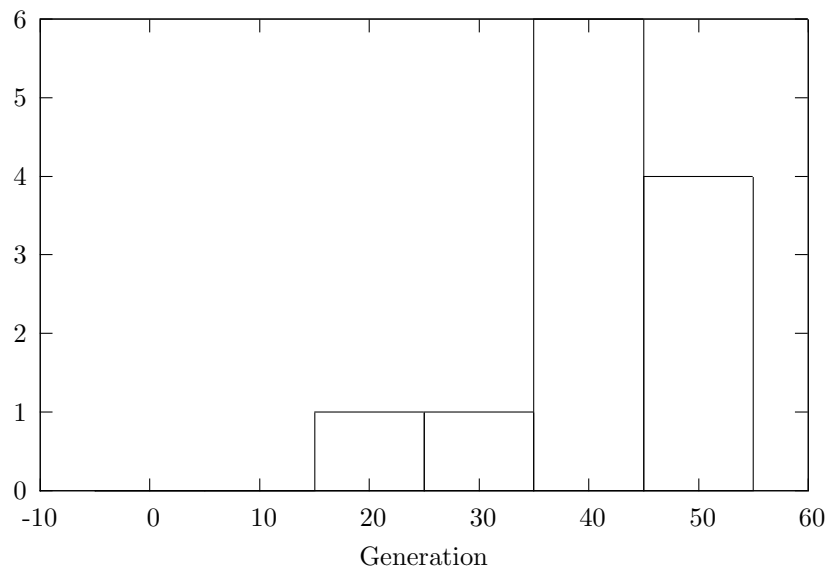


Table 4.4: RUN 1: Number of pool wins in the Generation 50 Tournament of Champions. This is the number of wins by each participant in a round robin pool composed of the best player teams of every tenth generation. A tie counts as 0.5 wins for each team.

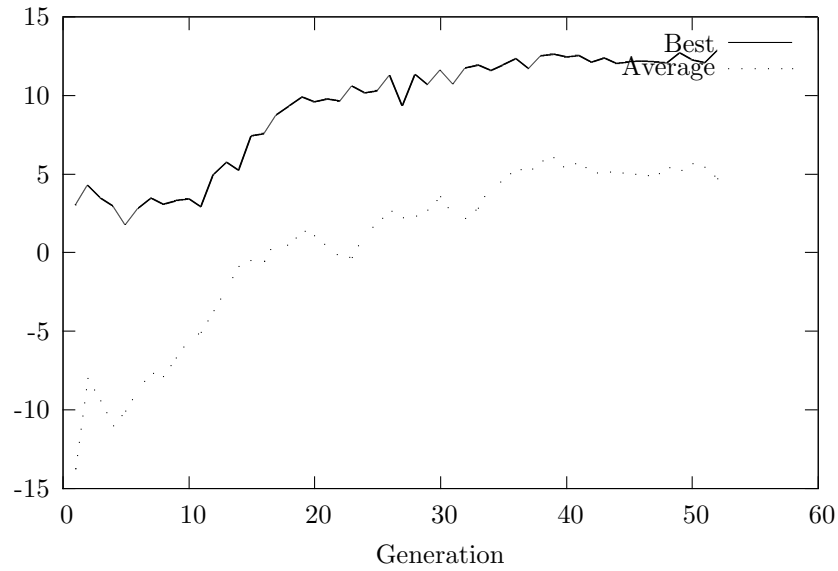


Table 4.5: RUN 1: Average and best fitness of ball teams across all nodes.

Generations	52
Mutation Rate	40%
Crossover Rate	40%
Trade Rate	10%
Reproduction Rate	10%
Within-team Xover Probability	0.1
Pool Size	3
Tournament Size	5
Population Size (per Node)	30
Score Limit	100
Immigration	Yes
Nodes	12
Individuals Evaluated	18,720

Table 4.6: Settings for Run 2.

## Run 2

Run 2 uses the exact same parameters as Run 1. The only change is that the fitness nudge for Chasers throwing the Quaffle was removed. This was done in an attempt to encourage Chasers to hold on the ball, and possibly evolve some more adaptive behaviors where the ball is only thrown if within a certain range of the goal. Unfortunately, this change appears to have had little effect on the evolutionary course of the system. Because there is no bonus for throwing the ball, the player population takes longer to discover kiddie-quidditch, and the best teams of generation 0-20 simply chase the Snitch and do little if anything with the Quaffle. However, once the Snitch becomes sufficiently difficult to catch, kiddie-quidditch quickly becomes the dominant strategy and remains so through the end of the run.

## Run 3

Run 3 also used almost all the same parameters as the previous two. Node 10 of the cluster was repaired, increasing the number of nodes from 12 to 13, and thus the number of individuals evaluated from 18,720 to 20,280. For this run `MAX_CODE_SIZE` was doubled from 60 to 120 points (a point is an atom or pair of parentheses). `PUSH_EXECUTION_LIMIT` was also doubled from 100 to 200 executions. The motivation for this change was the fact that player program sizes quickly reached the 60-point ceiling. It was hoped that by increasing the maximum program size and execution limit, it might be easier to evolve control structures that would be the basis for more adaptive strategies. Unfortunately, this was not the case. Once again, kiddie-quidditch becomes the dominant strategy between generations 30 and 40, and remains so through the end of the run.

## Run 4

There were several changes for this run: `MAX_RANDOM_CODE_SIZE` was increased to 70 points (from 30) more than doubling the maximum size of the initial randomly generated programs. This was done to get a better initial sampling of the search space, due to the enormous number of possible combinations of the elements in the function and terminal set. Restricted crossover was instated, meaning that crossover could only operate on two programs from the same index. That is, the first Chaser on a team could only be paired with other Chasers in the first position for crossover. Within-team crossover (which only occurs in 5% of crossover operations) is exempt from this restriction, and may operate on any two players of the same type. Restricted crossover was implemented in the hopes of encouraging position-specific behaviors. The `my-object-of-interest` sensor was removed, making it harder to discover the simple chase-and-throw behavior that is the basis for kiddie-quidditch. Stacks were no longer seeded with any instructions or vectors for the same reason. Lastly, the fitness nudge for ball-throwing and possession time was added. This was done so that Chasers

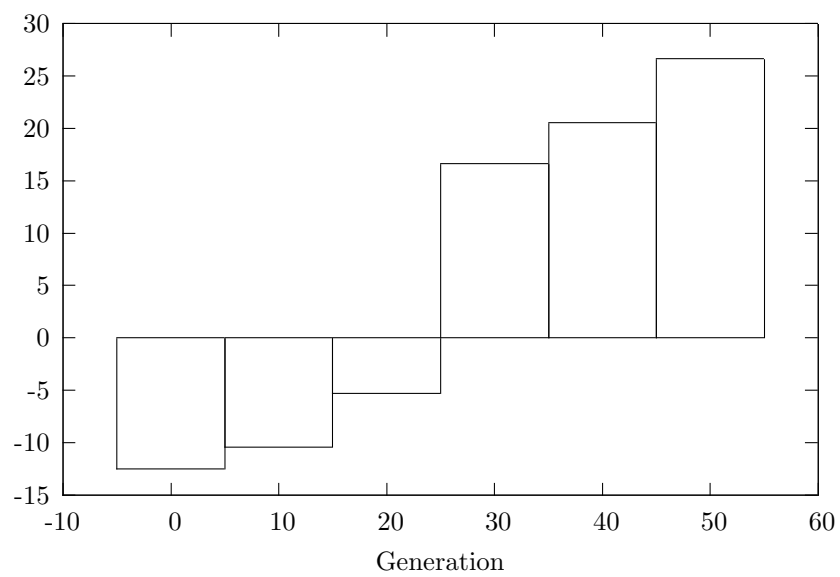


Table 4.7: RUN 2: Average Relative Fitness in the Generation 50 Tournament of Champions. This shows the average fitness score (as determined by the fitness function described in section 3.3) for each team after playing in a round robin pool composed of the best player teams of every tenth generation.

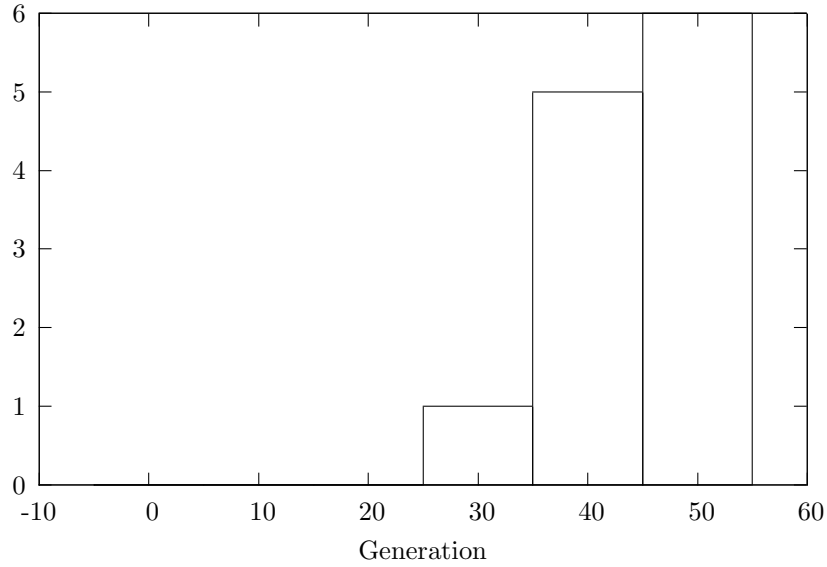


Table 4.8: RUN 2: Number of pool wins in the Generation 50 Tournament of Champions. This is the number of wins by each participant in a round robin pool composed of the best player teams of every tenth generation. A tie counts as 0.5 wins for each team.

Generations	52
Mutation Rate	40%
Crossover Rate	40%
Trade Rate	10%
Reproduction Rate	10%
Within-team Xover Probability	0.1
Pool Size	3
Tournament Size	5
Population Size (per Node)	30
Score Limit	200
Immigration	Yes
Nodes	13
Individuals Evaluated	20,280

Table 4.9: Settings for Run 3.

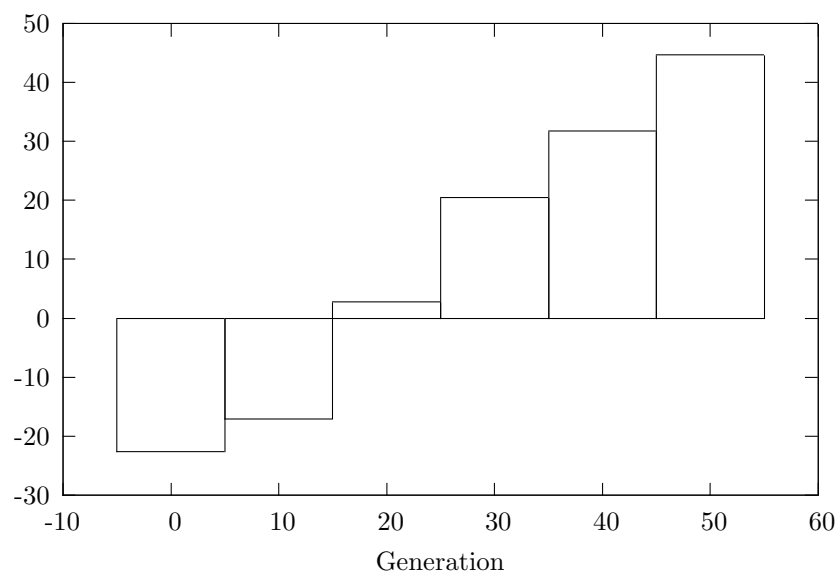


Table 4.10: RUN 3: Average Relative Fitness in the Generation 50 Tournament of Champions. This shows the average fitness score (as determined by the fitness function described in section 3.3) for each team after playing in a round robin pool composed of the best player teams of every tenth generation.

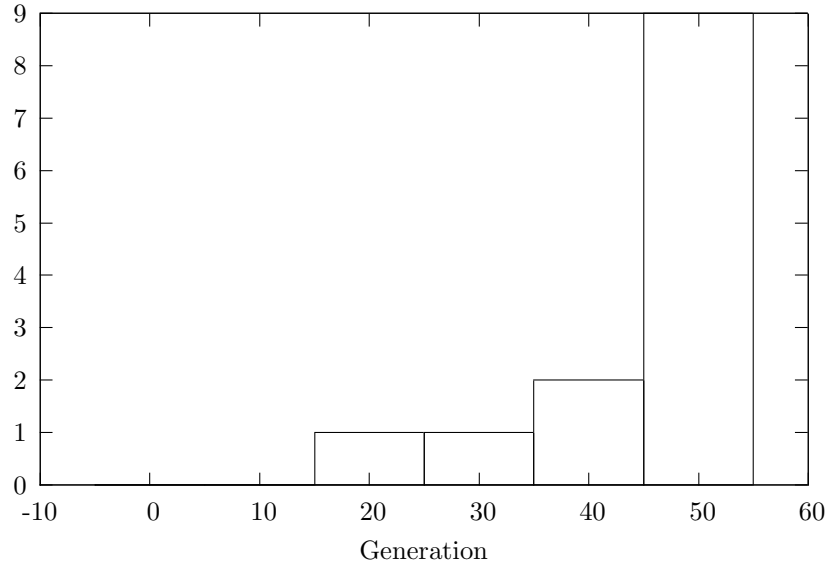


Table 4.11: RUN 3: Number of pool wins in the Generation 50 Tournament of Champions. This is the number of wins by each participant in a round robin pool composed of the best player teams of every tenth generation. A tie counts as 0.5 wins for each team.

Generations	52
Mutation Rate	40%
Crossover Rate	40%
Trade Rate	10%
Reproduction Rate	10%
Within-team Xover Probability	.05
Pool Size	3
Tournament Size	5
Population Size (per Node)	30
Score Limit	200
Immigration	Yes
Nodes	13
Individuals Evaluated	20,280

Table 4.12: Settings for Run 4.

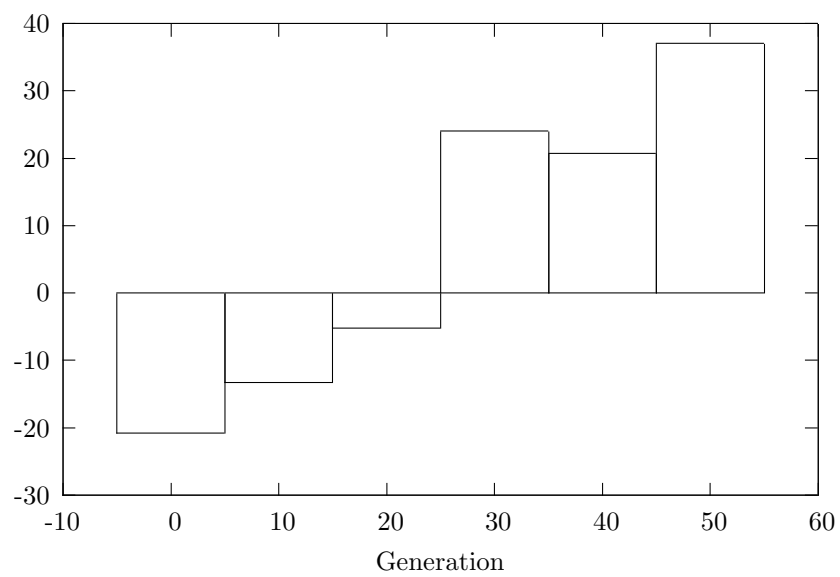


Table 4.13: RUN 4: Average Relative Fitness in the Generation 50 Tournament of Champions. This shows the average fitness score (as determined by the fitness function described in section 3.3) for each team after playing in a round robin pool composed of the best player teams of every tenth generation.

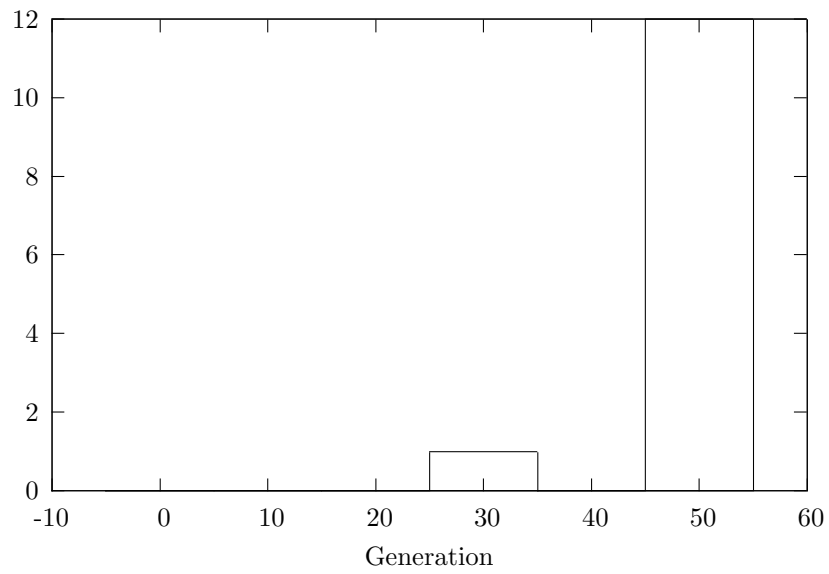


Table 4.14: RUN 4: Number of pool wins in the Generation 50 Tournament of Champions. This is the number of wins by each participant in a round robin pool composed of the best player teams of every tenth generation. A tie counts as 0.5 wins for each team.

would be equally encouraged to hold on to the Quaffle and throw it.

These changes slowed the progress to kiddie-quidditch. Many of the best player teams from generations 0, 10 and 20 were exclusively Snitch-catching teams. However, once the Snitch became sufficiently difficult to catch, all populations eventually converged on kiddie-quidditch.

## Run 5

When Run 4 ended, all player populations had converged to kiddie-quidditch. However, data from the Generation 50 tournament of Champions (Table 4.13) showed that the best Programs of generation 50 were significantly better than those from generations 30 and 40. It was unknown if this was due to the fact that the populations took longer to discover kiddie-quidditch, or if they finally had the tools to escape from that local optimum and move on to better behaviors.

Run 5 used the same settings as Run 4, but with a slightly smaller population size (20) and for almost twice as many generations (101). The goal was to see if evolution could sustain the fitness improvement showed in generation 50 of Run 4, and if so, if it would the populations would eventually escape the suboptimal strategy of kiddie-quidditch.

The best end-of-run teams from Run 5 still played kiddie-quidditch, but many of the teams showed the beginnings of separate offensive and defensive strategies. On some teams, one or two Chasers would hang out in front of their own goals while the rest of the Chasers continued to play chase-and-throw kiddie-quidditch. Occasionally, these defensive Chasers would intercept shots by the opposing team. One team appeared to go a step further: in addition to a defensive Chaser, an offensive Chaser would fly around near the opposing team's goals, and then pursue the quaffle when it came near.

These developments are very intriguing and indicate that perhaps with some more time, evolution could escape from kiddie-quidditch altogether and move on to better strategies. Many genetic programming runs evaluate fifty thousand, a hundred thousand or more individuals before finding a solution. The 26,000 individuals evaluated is a relatively small number, especially given the combinatorics of the function and terminal set. More time, a larger population size and more computational muscle could yield significantly better quidditch-playing teams.

Generations	101
Mutation Rate	40%
Crossover Rate	40%
Trade Rate	10%
Reproduction Rate	10%
Within-team Xover Probability	.05
Pool Size	3
Tournament Size	5
Population Size (per Node)	20
Score Limit	200
Immigration	Yes
Nodes	13
Individuals Evaluated	26,000

Table 4.15: Settings for Run 5.

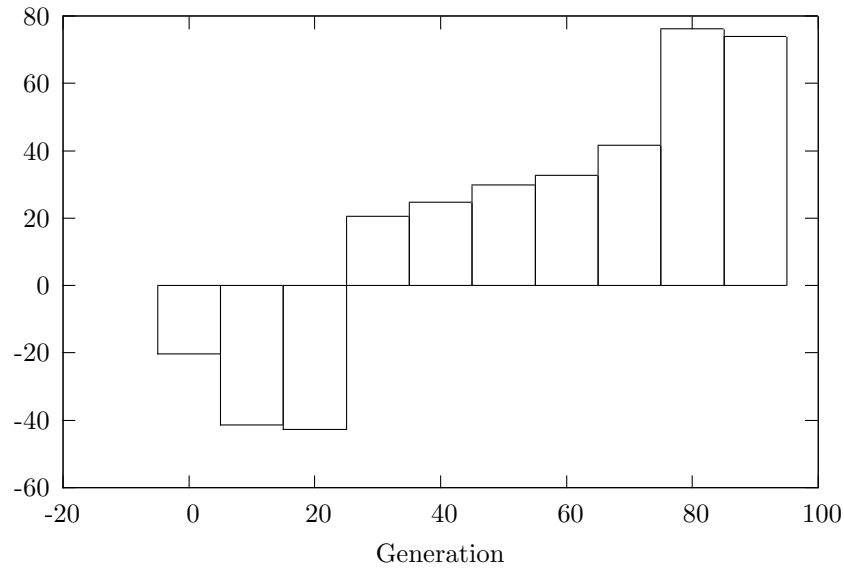


Table 4.16: RUN 5: Average Relative Fitness in the Generation 90 Tournament of Champions. This shows the average fitness score (as determined by the fitness function described in section 3.3) for each team after playing in a round robin pool composed of the best player teams of every tenth generation.

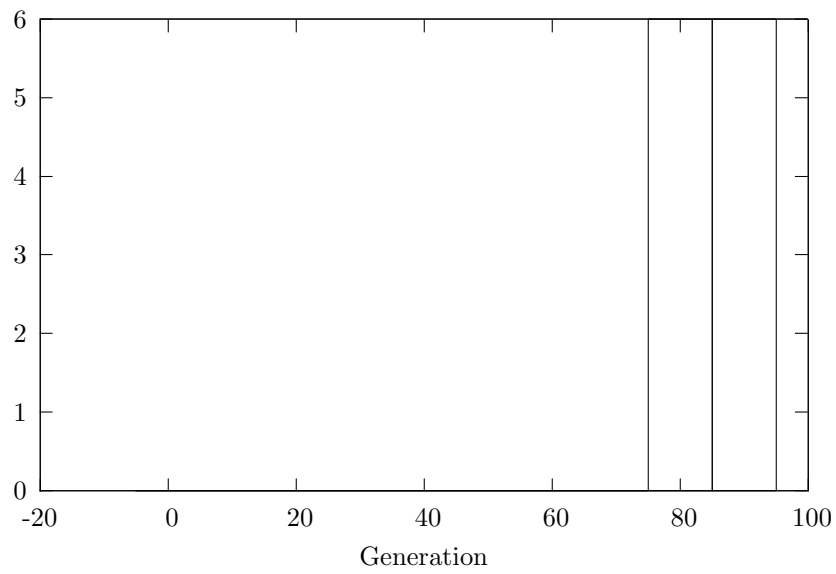


Table 4.17: RUN 5: Number of pool wins in the Generation 90 Tournament of Champions. This is the number of wins by each participant in a round robin pool composed of the best player teams of every tenth generation. A tie counts as 0.5 wins for each team.

## Chapter 5

# Conclusions and Future Work

Though complex strategies and teamwork failed to emerge, even a game of kiddie-quidditch is very entertaining to watch. With Chasers trying to score, Beaters whacking Bludgers, and Seekers zooming after the Snitch all at the same time, evolved quidditch is fascinatingly chaotic. The Quidditch Simulator is already fairly robust, and with some more work could easily become as challenging and useful a problem domain as the RoboCup Soccer Simulator. BREVE runs on Windows, Mac OS X, and Linux, and is designed such that plugins providing additional functionality can easily be written in C. Already, a plugin for the Stuttgart Neural Network Simulator (SNNS) kernel is underway, which would give BREVE a robust tool set for incorporating neural networks into simulations.

Virtual quidditch provides several advantages over the RoboCup Simulator. First, it realistically and efficiently simulates physics, including collisions, friction and gravity. Second, it is richly 3 dimensional: not only can the Quaffle be thrown up and down (in addition to North-South-East-West), all players and balls can fly freely through 3D space. Third, quidditch exists in a magical universe. No one on earth has ever seen a real quidditch game or has any idea what strategies are effective. Researchers working on simulated soccer are familiar with the game and what strategies they believe to be effective. Researchers run the risk of gaming their teams toward certain preconceived strategies, even if that is not their intention.

Genetic and Evolutionary Computation can be an effective strategy in the design of teams of cooperative agents. Evolution has no knowledge of soccer or quidditch. Genetic material that generates good behaviors tends to survive from one generation to the next. Because of this, evolution may find and exploit areas of the search space that human programmers would ignore because of preconceived notions about what constitutes “good” behavior. Furthermore, GEC is largely hands-off. Once the initial conditions have been set up, one only

has to give the system time to run until a sufficiently skilled team of agents has been evolved. As problem domains become increasingly complex, the amount of human effort needed to design agents that perform well is also increasing. GEC can both reduce the amount of human time and effort required to generate a solution, and also increase the quality of solutions that are generated.

## 5.1 Discussion

There were two goals of this Division III: first, to construct a robust and stable Quidditch Simulator with an eye towards automatic programming of quidditch-playing teams. This goal has been accomplished. The Quidditch Simulator implements all the rules of quidditch (with certain modifications, see Chapter Three) in a simulation that effectively models newtonian mechanics. The players in the simulator can be controlled by native steve code or by Push programs read in from the filesystem.

The QS can already be generally useful as a challenge problem for Artificial Intelligence. The architecture of the Quidditch Simulator is heavily object oriented, with inter-object communication occurring via the broadcasting and reception of Event objects. Each aspect of the simulator is implemented in a robust, independent fashion. This will ease the task of modifying and extending parts of the simulator in the future. Agents have simple actuators (an omnidirectional thruster plus a ball-specific action) and noisy sensors that give them a limited world-view. Each agent is autonomous - it is impossible to create a team that is controlled by some centralized process.

With some polishing and a few key additions the QS can join the RoboCup simulator as a widely used problem domain for Distributed Artificial Intelligence, Multi-Agent Systems and Genetic and Evolutionary Computation. Notable future additions include networked communication between players (clients) and the simulator (server), inter-agent communication, and more a more vision-like perception system.

The second goal of this work was to evolve teams of cooperative quidditch-playing agents. Both players and balls were evolved, with varying degrees of success. Player teams tended to get stuck in a suboptimal behavior of kiddie-quidditch, an inefficient every-man-for-himself strategy that ignores passing, ball control, and defense. Ball teams, on the other hand, evolved very effective behaviors. The objectives for the three balls are very different than that of the players. While the players are competing with one another to score points, the balls indiscriminantly try to hinder the players. The ball teams evolved very good behaviors, at least relative to the players. In all, the second goal of this work was met with only moderate success, but the difficulties encountered offer valuable insights into future improvements.

It is worth noting that there are two different types of coevolution occurring here: intraspecies and interspecies coevolution. Ball teams are scored according to their performance against player teams. That is, ball teams receive higher scores for performing better against players (by Bludgers beating them and

Snitches avoiding them). Improving fitness scores mean that ball teams are improving relative to the player teams they are competing against. Player teams on the other hand are scored in part for their performance against other player teams, and in part for their performance against ball teams. This complicates the fitness landscape for player teams, because the difficulty of the game for which they are being optimized changes with every change in strategies employed by ball teams and by player teams.

The apparent ease with which the player population discovers kiddie-quidditch is both promising and disappointing. It is disappointing because kiddie-quidditch is a suboptimal solution, and is often frustrating to watch as a spectator because one can't help but silently beg the evolved players to pass the ball or play defense. It's promising because it shows that automatic programming by means of GP can find better and better solutions over time in as complex an environment as virtual quidditch. One or more of the improvements outlined in the next section could give the evolving player population the tools it needs to escape from the kiddie-quidditch trap.

One change that would likely keep the population from converging to kiddie-quidditch would be to add a strong bias against it in the fitness function. Players could be heavily penalized for being very close to each other (on average). Such a bias would be very likely to keep the population from converging to kiddie-quidditch. However, in order to be successful, the penalty would have to be strong enough to counteract the positive effects of goal-scoring, thus significantly changing the game over which the population is being evolved. While such a hack may overcome the immediate problem of kiddie-quidditch, it gives little insight into what settings will give evolution the tools it needs to escape from local optima on its own. If one was evolving a quidditch team for competition, this would be a good hack to use in order to evolution to explore other strategies. However, given the scope and stated goals of this work, it was more important to explore changes to the evolutionary framework in an attempt to find a more general solution.

Even without escaping from kiddie-quidditch, evolving populations have shown the beginnings of more complex strategy and teamwork. Some Chasers on the best teams from Run 5 took on offensive and defensive roles, hovering near either their own goals or those of the opposing team until the Quidditch came nearby. While this still isn't explicit teamwork in the sense of ball passing or other coordinated maneuvers, it is a promising sign. Perhaps with more time and computational muscle, enough teams can be evaluated for evolution to actually move beyond kiddie-quidditch.

Ball teams, faced with a somewhat easier task, evolve very good strategies relative to player teams. The task for ball teams is easier for two reason: first, the combinatorics of the problem aren't as complex because there are significantly fewer functions and terminals available to construct programs. Second, fitness scoring is much simpler for balls: the Snitch shouldn't get caught and Bludgers should score as many hits as possible. This creates a smooth and easily climbable fitness landscape for the population of evolving ball teams. By generation 25, the Snitch is very difficult to catch and the two Bludgers score

many hits by constantly charging the nearest non-Beater.

## 5.2 Future Work

Future improvements upon this work can be divided into two categories: improvements to the Quidditch Simulator, and improvements to the Quidditch Evolver. Changes to the Quidditch Simulator are motivated by improving the “realism,” balance of the game and usability, whereas changes to the Quidditch Evolver are motivated by improving the performance of genetically programmed balls and players.

### Quidditch Simulator

**Stamina** Players (and perhaps balls as well) should have a limited amount of stamina that is used up by acting in the game world and is regenerated at a constant rate. The more effort players put into an action (i.e. the faster they move or the harder they throw) the more stamina is consumed. Players cannot expend more effort than they have stamina.

**Communication** Players should be allowed to communicate by “talking” to one another. This type of communication is implemented in the RoboCup Soccer Simulator with `say` and `hear` commands. A similar implementation would work well in virtual quidditch.

**Vision-like sensors** Currently, agents have noisy, but still very unrealistic sensors. They are equipped with omnidirectional radar that degrades in accuracy over large distances. A more vision-like sensor system would be much more realistic. There are different ways to implement such a feature, depending on how vision-like one wants the sensors to be. One easy way of implementing such sensors would use the noisy radar that agents currently have, but limit its scope to a forward-looking cone.

**Client/Server architecture** Using network code currently being developed for BREVE, the QS should be altered to run as a server, with player and ball teams connecting remotely. This would enable long-distance competitions and create the possibility of pairing GEC with the computational muscle distributed computing.

### Quidditch Evolver

**Separate throw and move trees** In [18], soccer-playing softbots had two program trees, one for movement and one for ball-kicking. Given the apparent difficulty of evolving control structures to decide when to throw the ball and when to hold on to it, giving Chasers a separate program tree dedicated to ball throwing may help evolution overcome that difficulty. The move tree would be

executed at every timestep, and the throw tree would be executed if the agent has possession of the Quaffle.

**Separate populations for each player and ball class** As discussed earlier, quidditch is really three sub-games being played in parallel. As such, it may be more effective to separate the different classes of players and balls into separate evolving populations, and create a customized fitness function for each.

**Shared ADF's for entire teams** Andre and Teller successfully evolved team-wide Automatically Defined Functions (ADFs) in [19]. ADFs have been shown to significantly increase problem-solving power and program parsimony in genetic programming runs [28]. Push has the capability to evolve modular programs, but evolution appears not to have taken advantage of this for quidditch. Furthermore, any evolved modularity in player program trees would be useful only to that specific player. Giving each team (or each group of players of a specific class) a set of evolvable ADFs could help teams evolve building-block behaviors that can be combined to create more complex offensive and defensive strategies.

**Internal Memory** Another big obstacle to evolving more complex behaviors is the lack of any capability for storing information across over time. Internal memory would give agents the ability to use internal state when deciding what to do, and could also enable them to make calculations about the trajectory of other objects in the game world by comparing previous and current locations. Internal memory could be easily implemented by not clearing the memory stacks after each iteration. However, a maximum stack depth would have to be enforced to keep the stacks from growing unmanageable over the course of a game.

## 5.3 Conclusions

What can we take away from these experiments? What have we learned about virtual quidditch as a challenge problem for software agents? What have we learned about using GP to evolve teams of agents for quidditch? Did the evolved player or ball teams learn any kind of cooperative behavior? While there is much room to explore all of these questions further, the experiments described here give some interesting early answers.

quidditch is attractive as a challenge problem for a variety of reasons. It is a fast-paced and dynamic game in which players must make full use of all three dimensions in order to be successful. Since it is imaginary, adaptive quidditch-playing strategies are unknown and can't be built-in to a team of agents; this makes it an ideal testing ground for automatic programming techniques. This implementation of virtual quidditch also offers realistic simulation of physics, noisy sensors, faster-than-realtime gameplay, and the possibility of evolving teams of both balls and players. Quidditch turns out to be a fascinating game to watch, though sometimes difficult to follow, because it is really

three sub-games being played in parallel: Chasers playing a flying soccer game with the Quaffle, Beaters whacking Bludgers, and Seekers chasing the Snitch. Except for Beaters, who could judiciously apply Bludger-whacks in order to disrupt the opposing team and better defend their own, these three sub-games do not overlap at all.

The fact that quidditch is really three sub-games running simultaneously changes the way it should be approached as an automatic programming problem. In the experiments performed here, there were only two populations: ball teams and player teams. Individuals on each team would live or die by the collective performance of the team. Selective pressure on Beaters and Seekers was much weaker than on Chasers, particularly once goal scoring was commonplace and the Snitch became difficult to catch. In many runs, Beaters would stop moving or doing anything intelligent after the first twenty generations. The effect of Beaters on the outcome of game comes from their ability to defend teammates from and harass opponents with the Bludgers. However, with kiddie-quidditch the dominant strategy, Bludgers have very little effect on goal-scoring because colliding with one Chaser still leaves three to continue the chase-and-throw strategy. Seekers face a similar difficulty. Seekers only affect the outcome of the game if they catch the Snitch. Once Snitches become sufficiently difficult to catch, Seekers tend to lose their effect on games. By conferring fitness to the entire team through a system of lexical dominance, it is very difficult to promote better Snitch catching behavior by degrees as opposed to the all-or-nothing approach used in this work. It may be much more effective to evolve a separate population for each ball and player role in quidditch. Instead of ball and player teams, there would be separate evolving populations of Chasers, Beaters, Seekers, Bludgers and Snitches, each with their own distinct fitness criteria.

Another difficulty in the application of GP to quidditch is the attractiveness of suboptimal behaviors like kiddie-quidditch. Kiddie-quidditch is easy to discover, and better strategies that include defense and passing are evidently very difficult for GP to discover. Getting stuck in local optima is one of the biggest problems for the application of automatic programming techniques to complex problems like virtual soccer and quidditch. When an evolutionary run gets mired in a local optimum, the computational effort that should be working to discover and optimize adaptive strategies is instead wasted as it recycles the same strategy over and over. This eliminates the advantages in speed and reduced human effort that automatic programming offers over other AI techniques. Luckily, this obstacle was encountered and overcome in both [18, 19] when evolving RoboCup soccer teams. By implementing some of the methods used by researchers in the field of simulated soccer, it is very possible that evolving populations of quidditch-playing teams can be equipped with the tools necessary to escape local optima or avoid them altogether.

Intuitively, it seems that in order to play quidditch well, players must act cooperatively. As with earthly ball games like soccer and basketball, ball control and ball movement are probably important factors in deciding the outcome of the game. In order to achieve these, Chasers must work cooperatively, as a

team. Although high-level behaviors like passing and defensive schemes did not evolve, did other cooperative behaviors emerge that could be precursors to such behavior? In short, the answer is no. The most cooperative behavior that was observed is the ball-holding behavior of one Chaser, who carries the ball to the vicinity of the opponent's goals and eventually drops it. The other Chasers pursue the Quaffle, and when it is dropped, have the opportunity to take a very high-percentage shot on goal. The promising signs of offensive and defensive behavior from Run 5 could conceivably evolve into more complex cooperation, but as it stands the best teams from Run 5 are still very uncooperative.

In all, this work represents a very promising initial foray into building a Quidditch Simulator and applying GP to the creation of quidditch-playing agents. Furthermore, the experiments performed here have shed light on what direction revisions to the Quidditch Simulator and Evolver should take in order to improve usability and performance. With some revisions, the performance of genetically programmed teams could be greatly improved, and the Quidditch Simulator could become a generally useful tool for AI researchers.

# Thanks and Acknowledgments

This Division III was prepared using L<sup>A</sup>T<sub>E</sub>X with TexShop for Mac OS X. It would not have been possible without the help and guidance of Lee Spector, Jaime Davila, and Jon Klein.

**Division III Committee** Lee Spector and Jaime Davila

**Thanks** J.K. Rowling for creating Quidditch and writing such entertaining books; Lee Spector for being my professor, advisor, committee chair, and mentor; the Crawford-Marks family for spawning and raising me; Ryan Moore for keeping everything up and running smoothly; Dave Thomson, Andrea Davis, Eric Anderson, Liz Nichols, Jake Thomas, Nick Wells, Talia Bigelow, Carmen Ruiz Medina, Jesse Oates; Mod 21; the Beyond Beginning Kayaking class; Red Scare; OPRA and the Aldersons; the Computer Science Bibliography; CiteSeer; the GP mailing list; and Rudy Rucker for introducing a 16-year-old computer geek to Artificial Life and Evolutionary Computation and making it sound so damn Rock 'n' Roll.

# Appendix A

## Log of Evolutionary Runs

02192004 - All Nodes;

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSEOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 7. GENERATIONS 10. TEAM\_POPULATION\_SIZE 30.

NOTES: Failed after one generation due to bug in QEVOLVE.TZ.

02202004 - All Nodes; 11:20-11:40am

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSEOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 7. GENERATIONS 10. TEAM\_POPULATION\_SIZE 30. GAME\_TIME\_LIMIT\_IN\_SECONDS 200. GAME\_SCORE\_LIMIT 100.

NOTES: Stopped runs on Feb 24, 11:10am. Some runs had crashed before that. Some evolution. Probable memory leak caused all runs to slow down significantly after 1 generation, so no nodes completed more than a few generations, even after running for almost five days. The population on one node did learn to chase the snitch.

02262004 - All Nodes; 7:25-7:30pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSEOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 7. GENERATIONS 30. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION (2 individuals per population).

NOTES: (Run stalled after one generation due to bug. Fixed and restarted at 11:55pm) Runs finished in under 24 hours, however after briefly perusing the log files, it appears that no significant evolution occurred. Will investigate further to determine if this is a problem of combinatorics, fitness landscape, or buggy code.

02282004 - All nodes; 12:40pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSEOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 7. GENERATIONS 30. TEAM\_POPULATION\_SIZE 30.

GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION (2 individuals per population).

NOTES: There was occasional scoring, but no 'good' behavior was preserved accross generations. Population size may be too small, or genetic operators too destructive? Not sure.

02292004 - All nodes; 3:00pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 4. TOURNAMENT\_SIZE 7. GENERATIONS 100. TEAM\_POPULATION\_SIZE 50. GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION (2 individuals per population). I also am 'seeding' the stacks with move and throw instructions on the CODE stack, and an object-of-interest vector on the POINT stack.

NOTES: (Ended run after 29 Generations, 03022004, 4:30pm) Repeatedly saw the beginnings of good behavior (players going after the correct ball, or throwing, occasionally even scoring) but for some reason good behavior was not being preserved from generation to generation. I discovered that my create-new-generation method was getting individuals from the same list which is was modifying. The effect of this was that high scoring individuals were being subjected to countless genetic operations every generation, and what's more, were not even proliferating in the population. I believe that fixing this bug will yield very significant improvements.

03132004 - All nodes; ?:?? am/pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 4. TOURNAMENT\_SIZE 5. GENERATIONS 100. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION (2 individuals per population).

NOTES: Ran until all nodes were generation 50-something. Kiddie-quidditch emerged, and on some nodes it looked like players were learning to hold on to the ball until close to the goal and then throw it. Discovered a bug in the ball catching code. In response to the behavior I saw, I added two functions for the next run. 'close-to' returns T if the agent is within N distance (top of integer stack) of vector location V (top of vector stack). 'move-as-fast-as-possible' moves at maximum thrust in direction specified by top of vector stack.

03222004 - All nodes; 10:25 pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 15. TRADE\_PROBABILITY 10. POOL\_SIZE 4. TOURNAMENT\_SIZE 5. GENERATIONS 61. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION (improved).

NOTES: Hex (the computer cluster) was shut down due to a A/C failure. Generations had reached the early thirties on every node. 'Cherry-picking' quidditch was the most common evolved strategy, w

here agents learned to compensate for the downward acceleration of the quaffle, and would loft it at the goal from midfield, scoring dozens of goals in a matter of a minute or two of simulation time. High scores were beginning to decrease right before the server was shut down. Perhaps because the strategy had spread throughout the population, or perhaps because some individuals were learning to defend against it.

04062004 - All nodes (minus n10); 11:30am

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 10. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 5. GENERATIONS 51. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION.

NOTES: Added perception noise. This prevented "cherry-picking" quidditch from emerging. Replaced by "kiddie-quidditch", another suboptimal strategy.

04092004 - All nodes (minus n10); 12:40pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 10. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 5. GENERATIONS 51. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 100. VARIABLE GAME TIME LIMIT. IMMIGRATION.

NOTES: Same parameters as previous run, except there was no fitness 'nudge' for throwing the ball. My hope was to encourage holding on to the ball, and maybe avoid evolving kiddie quidditch.

04112004 - All nodes; 3:00pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 10. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 5. GENERATIONS 51. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 200. VARIABLE GAME TIME LIMIT. IMMIGRATION. Doubled MAX\_CODE\_SIZE from 60 to 120. Doubled PUSH\_EXECUTION\_LIMIT from 100 to 200.

NOTES:

04132004 - All nodes; 9:44pm

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 5. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 5. GENERATIONS 51. TEAM\_POPULATION\_SIZE 30. GAME\_SCORE\_LIMIT 200. VARIABLE GAME TIME LIMIT. IMMIGRATION. MAX\_CODE\_SIZE 120. PUSH\_EXECUTION\_LIMIT 200. MAX\_RANDOM\_CODE\_SIZE 75 (up from 30).

NOTES: Removed my-object-of-interest sensor. Added fitness nudge for ball-throwing and for possession time.

04152004 - All nodes; 11:30pm.

PARAMETERS: MUTATION\_PROBABILITY 40. CROSSOVER\_PROBABILITY 40. CROSSOVER\_WITHIN\_TEAM\_PROBABILITY 5. TRADE\_PROBABILITY 10. POOL\_SIZE 3. TOURNAMENT\_SIZE 5. GENERATIONS 100. TEAM\_POPULATION\_SIZE 20. GAME\_SCORE\_LIMIT 200. VARIABLE GAME TIME LIMIT. IMMIGRATION. MAX\_

CODE\_SIZE 120. PUSH\_EXECUTION\_LIMIT 200. MAX\_RANDOM\_CODE\_SIZE 75.  
NOTES:

## Appendix B

# Source Code, Results, and Movies

Included with this text is a CD-ROM containing the complete source code the the Quidditch Simulator and Quidditch Evolver, all the log files and results from the evolutionary runs, and some quicktime movies of evolved teams at various stages.

Directory	Contents
/QSEE/	Source code for QS and QE
/Quidditch_Results/	Raw data from runs, organized by date.
/Results - 0***2004/	Selected movies and demo files of evolved teams.
/breveSNNS/	Alpha version of the SNNS/BREVE plugin.
/div3.pdf	A digital copy of this document.

There are some QuickTime movies included for some of the runs that show the best of generation teams at ten-generation intervals. The best of generation teams of any generation for any run can be viewed by running them in the quidditch simulator. To run the teams in the QS, do the following:

1. Copy the `/ballprogramList_0***_Gen*` and `playerprogramList_0***_Gen*` from `Results - 0***2004/` to `/QSEE/`.
2. Open up `/QSEE/QSEE.tz` and edit lines 87 and 90 to refer to the appropriate files (the ones you just copied).
3. Run `breve` on `QSEE.tz`.

# Bibliography

- [1] K. Whisp and J. Rowling, *Quidditch Through the Ages*. New York, NY: Scholastic Press, 2001.
- [2] J. Rowling and M. Grandpre, *Harry Potter and the Sorcerer's Stone*. Scholastic, Inc., 1998.
- [3] L. Spector, R. Moore, and A. Robinson, "Virtual quidditch: A challenge problem for automatically programmed software agents," in *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers* (E. D. Goodman, ed.), (San Francisco, California, USA), pp. 384–389, 2001.
- [4] G. Tesauro, "Practical issues in temporal difference learning," in *Advances in Neural Information Processing Systems* (J. E. Moody, S. J. Hanson, and R. P. Lippmann, eds.), vol. 4, pp. 259–266, Morgan Kaufmann Publishers, Inc., 1992.
- [5] J. Schaeffer and R. Lake, "Solving the game of checkers."
- [6] H. Simon and J. Schaeffer, "The game of chess," 1992.
- [7] B. Bouzy and T. Cazenave, "Computer go: An AI oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.
- [8] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, "RoboCup: The robot world cup initiative," in *Proceedings of the First International Conference on Autonomous Agents (Agents'97)* (W. L. Johnson and B. Hayes-Roth, eds.), (New York), pp. 340–347, ACM Press, 5–8, 1997.
- [9] J. R. Koza, *Genetic Programming*. Cambridge, MA: MIT Press, 1992.
- [10] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman, "On cooperation in multi-agent systems," *The Knowledge Engineering Review*, vol. 12, no. 3, pp. 309–314, 1997.
- [11] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1975.

- [12] M. Ebner, A. Grigore, A. Heffner, and J. Albert, “Coevolution produces an arms race among virtual plants,” in *Proceedings of the 5th European Conference on Genetic Programming*, pp. 316–325, Springer-Verlag, 2002.
- [13] C. D. Rosin and R. K. Belew, “Methods for competitive co-evolution: Finding opponents worth beating,” in *Proceedings of the Sixth International Conference on Genetic Algorithms* (L. Eshelman, ed.), (San Francisco, CA), pp. 373–380, Morgan Kaufmann, 1995.
- [14] J. B. Pollack, A. D. Blair, and M. Land, “Coevolution of a backgammon player,” in *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems* (C. G. Langton and K. Shimohara, eds.), (Cambridge, MA), pp. 92–98, The MIT Press, 1997.
- [15] T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright, “Evolving multi-agent coordination strategies with genetic programming,” Tech. Rep. UTULSA-MCS-95-04, 31, 1995.
- [16] T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright, “Evolving a team,” in *Working Notes for the AAAI Symposium on Genetic Programming* (E. V. Siegel and J. R. Koza, eds.), (Cambridge, MA), AAAI, 1995.
- [17] S. Luke and L. Spector, “Evolving teamwork and coordination with genetic programming,” in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 150–156, MIT Press, 28–31 1996.
- [18] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler, “Co-evolving soccer softbot team coordination with genetic programming,” in *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, (Nagoya, Japan), 1997.
- [19] D. Andre and A. Teller, “Evolving team darwin united,” 1999.
- [20] P. Stone and M. Veloso, “A layered approach to learning client behaviors in the RoboCup soccer server,” *Applied Artificial Intelligence*, vol. 12, pp. 165–188, 1998.
- [21] P. Stone and M. M. Veloso, “The CMUnited-97 simulator team,” in *RoboCup*, pp. 389–397, 1997.
- [22] P. Stone, M. Veloso, and P. Riley, “The CMUnited-98 champion simulator team,” in *RoboCup-98: Robot Soccer World Cup II* (Asada and Kitano, eds.), pp. 61–76, Springer, 1999.
- [23] R. A. Brooks, “Intelligence without representation,” *Artif. Intell.*, vol. 47, no. 1-3, pp. 139–159, 1991.

- [24] M. Prokopenko, M. Butler, and T. Howard, “Cyberroos2000: Experiments with emergent tactical behaviour.”
- [25] J. Klein, “Breve: a 3d environment for the simulation of decentralized systems and artificial life,” in *Artificial Life VIII: Proc. of the 8th Int. Workshop on the Synthesis and Simulation of Living Systems*, MIT Press, 2002.
- [26] L. Spector, C. Perry, and J. Klein, “Push 2.0 programming language description.” <http://hampshire.edu/l spectator/push2-description.html>, 2003.
- [27] L. Spector and A. Robinson, “Genetic programming and autoconstructive evolution with the push programming language,” *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, 2002.
- [28] J. R. Koza, *Genetic programming II: automatic discovery of reusable programs*. MIT Press, 1994.
- [29] J. Grefenstette, “Credit assignment in rule discovery systems based on genetic algorithms,” *Mach. Lang.*, vol. 3, no. 2-3, pp. 225–245, 1988.
- [30] B.-T. Zhang and D.-Y. Cho, “Coevolutionary fitness switching: Learning complex collective behaviors using genetic programming,” in *Advances in Genetic Programming 3* (L. Spector, W. B. Langdon, U.-M. O’Reilly, and P. J. Angeline, eds.), pp. 425–445, Cambridge, MA, USA: MIT Press, 1999.