Coevolving Quidditch Players Using Genetic Programming

Student: Michael Ahern

Advisor: Sergio Alvarez

6th May 2005

Table of Contents

List of Tables

Abstract

Ever since the invention of the computer people have been fascinated by the idea of Artificial Intelligence (AI). Although general purpose AI remains science fiction, AI and Machine Learning (ML) techniques have been used to develop everything from autonomous Martian rovers to computers that drive cars. Although it is ideal to build physical systems to test algorithms, often times cost constraints require initial development to be done using rich game-like simulators. Building upon this line of research, my thesis describes the automatic programming of simulated agents playing a "Quidditch-like" game using genetic programming.

Chapter 1: Introduction

For those who have neither read the books nor seen the films, Quidditch is the most popular sport played in J. K. Rowling's series of *Harry Potter* books. The game resembles a fusion between basketball and soccer played on flying brooms. Two teams, consisting of seven players, as well as four balls (three of which are intelligent) play the game. Within each team there are three players who act as chasers, who attempt to throw the quaffle through one of the three goals, while a fourth player acts as a keeper, guarding the team's own goals. Next there are two beaters, whose job it is to defend the team from the two bludgers. The bludgers, in turn, are nasty semi-intelligent balls that seek to disrupt the players by crashing into them. Finally there is the seeker, whose job is to catch the golden snitch, an intelligent winged golf ball, the capture of which ends the game.

Looking at the problems being solved using game-like genetic programming systems, Spector, Moore, and Robinson (Spector, Moore, and Robinson, 2001) proposed Quidditch as a possible problem. Unlike previous work such as RoboCup (Luke, Hon, et al. 1997), Quidditch presented a richly 3-D environment, with agents having three degrees of freedom, bound only by the ground. In addition, the game is also richly heterogeneous, as actors such as the chasers and the bludgers vary widely in terms of their physical characteristics, intentions, and actions.

Before tackling the entire problem, the game was simplified to better understand how to develop the full system. Using a full-featured simulator as a base (Crawford-Marks 2004, see appendix A), the game was stripped down to just the chasers and keeper, removing the bludgers, snitch, beaters, and seeker (see chapter 2). Although the resulting system failed to learn how to score, the system showed steady evolutionary progress towards "kiddie-Quidditch" (rushing for the ball, then throwing it immediately and chasing after the ball again). Based on the results of this work, given more time it is believed that such a system could eventually learn how to score and possibly acquire more advanced game play strategy.

1.1 Genetic Algorithms

Genetic algorithms (GA's) are a machine-learning (ML) technique loosely modeled on biological evolution. Using this technique, primordial algorithms are

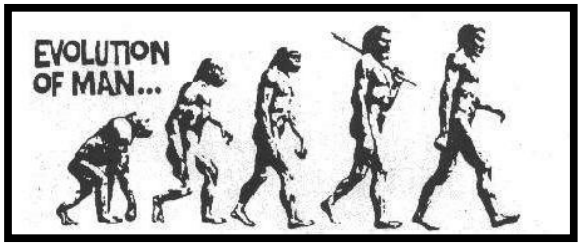

Figure 1: The evolution of Man

"evolved" (or optimized) along some evolutionary path towards an "objective". Thinking about this in biological terms one can think of the evolution of ape to man (fig. 1). Although at the intermediate stages, the "ape-men" are not "men" per se, at each stage of evolution individuals appear to be increasingly "human-like" (standing more upright, being less hairy, having a flatter forehead, etc) than the *previous* stage of evolution. Likewise, as the algorithm is "optimized", each intermediate stage of optimization, while not yet having reached the objective, it is somehow *closer* to the goal than the previous generation of algorithms.

In contrast to neural-nets, the *representation* of GA's is somewhat arbitrary. Often genetic algorithms are represented by strings of characters, bits or integers, but they can also be used to represent entire computer programs (Goldberg, 1989; Koza, 1992). Secondly, again in contrast to neural-nets, GA's are a *population-based* approach. This means that rather than optimizing a single algorithm within the search space, GA's attempt to optimize a group or *population* of algorithms instead.

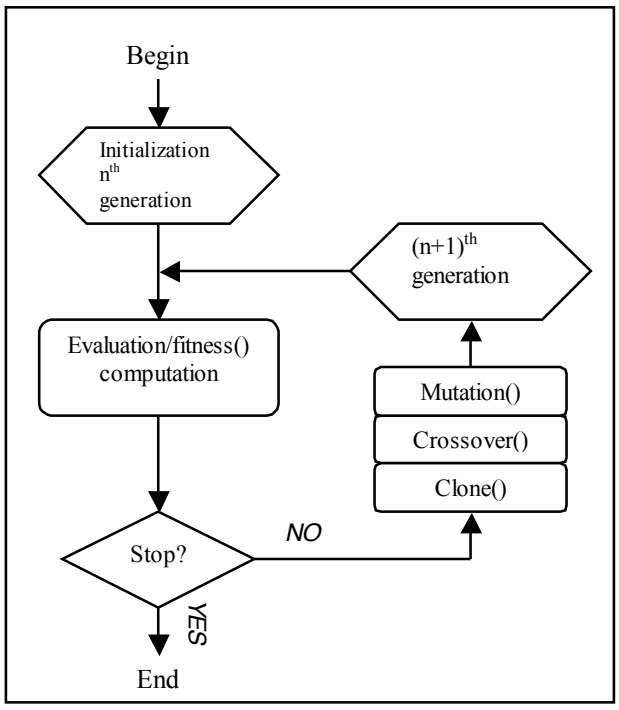

Figure 2: GA Flowchart

The process of evolving a population of individuals can best be understood by examining figure 2. After initialization, a fitness() function computes the relative strength of each individual in the population. Next, the stopping criteria are checked to see if some sort of objective or halting condition has been reached. If the conditions are not met, a selection is made based on the fitness scores of the population. After the selection a set of genetic operators (clone(), cross-over(), and mutate()) are applied to the individuals, creating the next generation (n+1). Finally, this generation is reevaluated and the stopping criteria are rechecked, repeating the cycle again.

To see how this would work over a population of "colors", examine figure 3. In this example an initial population representing "colors" is created, resulting in three red, three blue, and two yellow "colors". Next, each of the individuals is scored by the fitness function, in terms of their closeness to purple. As blue and red are closer to purple than

yellow they receive a higher fitness score. After selection, the clone() function preferentially copies two red, two blue, and one yellow individual into the next generation. Then the crossover() operator splits the red and blue objects in half, mixing one half of each to produce two purple objects. Finally, the mutate() operator takes a blue object and somehow alters its "genes" to produce the light blue color seen in generation n+1.

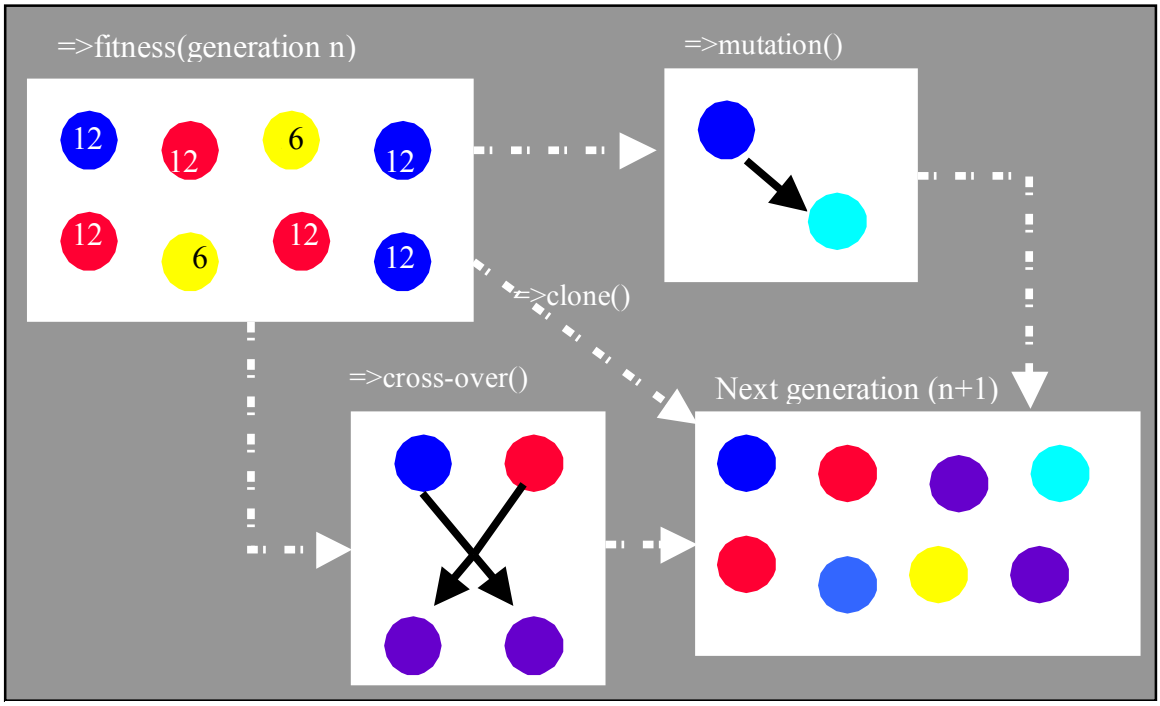

Figure 3: Genetic Algorithm Example

1.2 Genetic Programming

Genetic programs (GP's) are exactly like GA's except in that their representation is an actual computer program rather than a binary string or an Lab color value. Typically, following the work of Koza (Koza, 1992), LISP-like function trees are used to represent these programs (see figure 4). Rather than being an arbitrary program written in C, the function set, following from the problem, is chosen by the user. The challenge then with this model is to select a set of functions and non-terminals that will be flexible enough to represent the problem solution, while being constrained enough to make the search tractable. Take, for example, the simple case trying to evolve the function: "sin(x) +

sqrt(x^2 + y)". Given that problem one would need a function set that included "sin", "sqrt", "x", "y", "cos", and so forth, in order to represent the solution. With more complicated problems, the function set is not quite as obvious. Hence, the task of the genetic programmer is to somehow provide a flexible solution that includes enough hints to make the problem tractable.

1.3 Previous Research

1.3.1 RoboCup Software

The RoboCup tournament was started in the mid-nineties with the goal of developing a human-competitive team of robotic soccer players by 2050. Each year, the Robo World Cup has been held, pitting teams of either physical robots or simulated soccer players against each other.

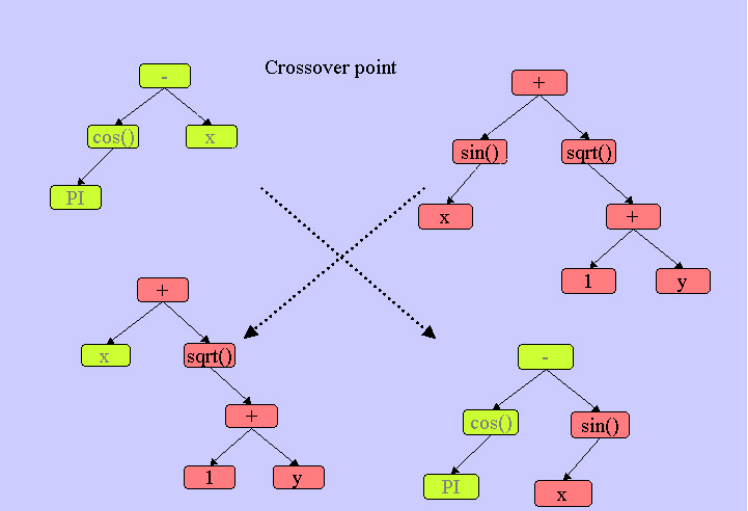

Figure 4: Genetic Programming crossover example (Mitchell, 2001)

The soccer simulator itself has two parts: the simulator, which keeps track of the score keeping, providing player input, and refereeing, and a client, connecting to the system over a network, which controls the players. The players themselves are two-dimensional circles projected onto a two-dimensional soccer field. In addition, the physics model has been greatly simplified, especially in regards to object collisions.

Genetic Programming was applied to the problem by the University of Maryland team headed by Sean Luke (Luke, Holm, et. al. 1997) for the RoboCup97 tournament. The group attempted two evolutionary approaches, one with a homogeneous team and one with a semi-homogeneous team (three types of player), selecting the strongest team (a homogeneous one) to compete in the tournament. In addition each player consisted of two program trees – one for kicking and one for moving – one of which was executed at each time step. The kick tree was executed whenever the ball was in "range", first nudging the player into a position to kick, then executing the tree to determine the kick direction and magnitude. The move tree was executed whenever the ball was out of range, returning a magnitude/directional vector to determine what direction to move the player and how fast to do it. The two program trees themselves consisted of a combination of hand-coded and evolved functions ranging from a simple action such as

"dribble" to more complex actions such as "blocking the goal". This implementation was fairly successful pointing towards later work in the same domain. The two-program tree methodology presented here also served as the basis for the implementation presented in this paper.

1.3.2 Virtual Witches and Warlocks

Addressing Spector, Moore, and Robinson's proposal, Raphael Crawford-Marks (Crawford-Marks, 2004) attempted to develop a Quidditch GA based on earlier work headed by Lee Spector to develop a Quidditch simulator. The simulator developed implemented the full game of Quidditch as described in *Quidditch Through the Ages* (Whisp and Rowling, 2001), including possession time limits, fowls, and score tracking. The evolver created teams using a "stack-based" language called *Push*. In addition to player teams, a set of "ball" was coevolved as well. Each player team consisted of seven players, one stack for each player on the Quidditch team. Each ball team contained three ball stacks, one for each of the three intelligent Quidditch balls. After running the simulator for a specified period of time or until a score limit was reached the simulator ended the simulation and returned the fitness scores to the evolver application.

One of the interesting design choices was the use of the *Push* language. Push is a language specifically designed for genetic programming. Although Push programs resemble LISP programs syntactically, their execution is "stack-based", more closely matching languages such as Postscript. In detail, execution of a program P can be described as:

```
Exec = on input P:
      If P is a single instruction then execute it.
      Else if P is a literal then push it onto the appropriate
         stack.
      Else (P must be a list) sequentially execute each of the
         Push programs in P.
```

Thus even for a simple Push program such as:

```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
```

The end result would be the following three stacks:

```
BOOLEAN STACK: ( TRUE )# TRUE, FALSE, OR
FLOAT STACK: ( 9.3 )    # 4.1, 5.2, +
INTEGER STACK: ( 6 )    # 2, 3, *
```

(Klein, 2005). For the Crawford-Marks implementation, the system was based around a *THROW* and a *MOVE* stack, from which functions such as "throw" and "move" found in the code stack commanded the player to execute the pop the top *throw* or *move* vector from their analogous stacks.

While Crawford-Marks had some success with this implementation, the system had only rudimentary game playing skills, as two players would hang back to defend the goal, and another two would engage in “kiddie-Quidditch”, grabbing the quaffle, throwing it, then chasing after it again.

One of the interesting design choices of the system was that teams were evaluated as a group rather than scored individually. While it is easy to make a strong case for the chaser players, considering how heterogeneous the Quidditch environment is, this methodology seemed prone to penalizing strong squads of like-players within the team for the ineptitude of their other teammates. Thus it was decided that before attempting to evolve the entire Quidditch-team, it first made sense to understand how these “squads” could be evolved independently. The resulting work in this thesis uses the simulator provided by Crawford-Marks’s work, simplifying the game to just the chasers and the quaffle.

Chapter 2: The Quidditch Simulator and Quidditch Evolver

The Quidditch evolution system can be thought of as comprising of two parts: the Quidditch Evolver and the Quidditch Simulator. In brief, the Evolver, which represents the bulk of this thesis work, maintains the population of TreesGenomes (described later): handling selection; applying the genetic operators; and using the Quidditch Simulator to compute the fitness scores. The Quidditch simulator, as derived from Crawford-Marks' work (Crawford-Marks, 2004), takes two TreesGenomes as input from the Evolver, plays them against each other, and computes their fitness scores.

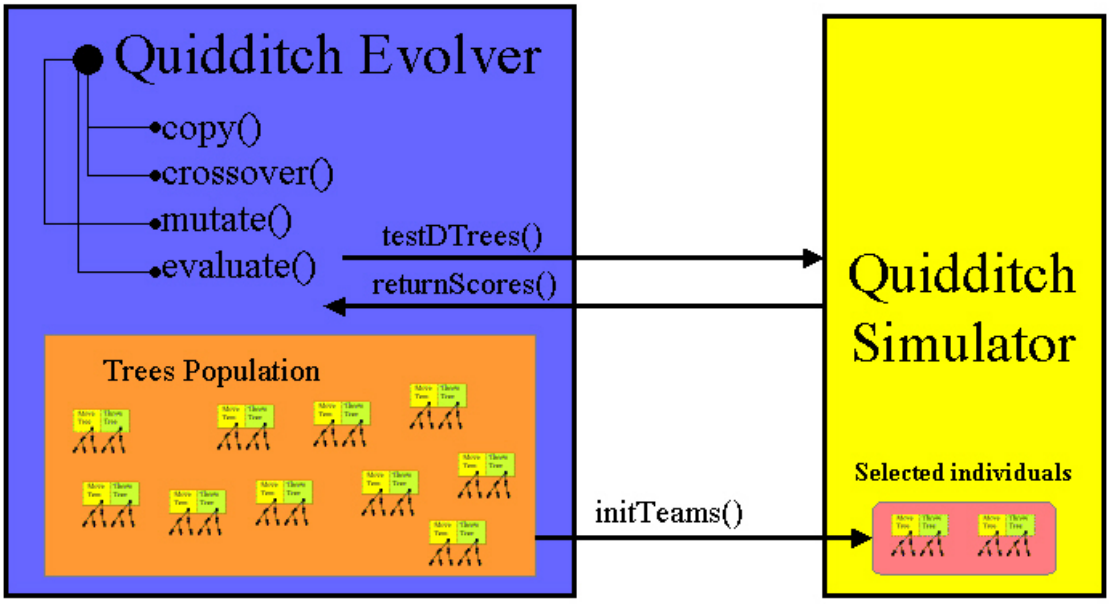

Figure 5: System Architecture

2.1 The Trees and DTree Genomes

The Trees and DTree implementation is modeled closely on the RoboCup methodology (Luke, Hohn, et al. 1997). As with RoboCup, the players on each team are homogeneous, possessing a "TreesGenome" composed of *move* and *throw* "DTrees". If the player does not possess the quaffle, only the move tree is executed. Otherwise, only the throw tree is executed. Despite the name, the throw tree can return either a throw vector or a move vector. The move tree on the other hand, returns a vector representing the desired location to move to. Throw vectors contain a location vector of the target, a velocity vector of the object that the throw is directed at, as well as a flag to indicate whether the vector is a throw or move instruction. Once the point-vectors are returned

from the top of their respective trees, action is taken. In the case of move-related vectors, the vectors are vectorized to (targetPoint – currentPoint) and fed to the "thrust" function of the chaser. Non-move throw vectors are then fed to the "generate-throw" behavior, which accounts for the physics properties of the ball and returns a directional vector to feed to the chaser "throw" function.

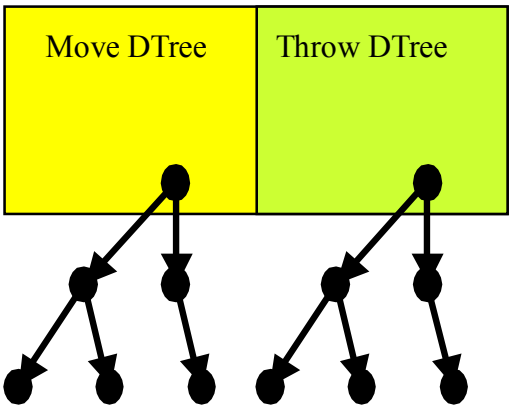

Figure 6: TreesGenome - Move & DTree

2.1.1 Representation

The DTree's themselves work like Lisp programs, with each function specifying a set of permissible child nodes. Individual functions were evolved, as in the case of the "generate-throw" behavior, or hand coded, as in most cases. While there are a couple of specific behavioral functions, such as "(block-goal)", the majority of the functions are sensory in nature ((goal), (home), (mate), etc) or functional operators ((if-[T|I|V]), (throw), (move), etc.), so that hand coding of these functions made sense. In terms of implementation, DTree is a C++ tree genome derived from Mathew Wall's GATreeGenome classes (Wall, 2004). While a couple of functions are written in *C++,* the vast majority of them make calls to wrapped *Breve* functions (described in the simulator section).

- **Code Fragment:**

```
(if-v(AND(NOT(opp-closer))
         (ball-loose))
     (ball)
     (block-goal))
```

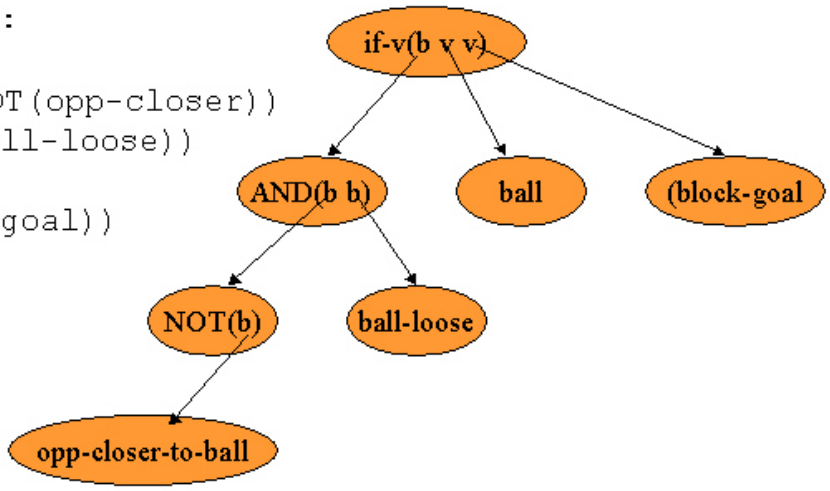

Figure 7: Example Move Tree

2.1.2 DTree Object Types and Function Parameters

Looking at the example in figure 7, one can note that the definition of AND takes two *Booleans* as parameters. In order to guarantee that each DTree is actually executable, trees are type specific in terms of what sort of function nodes are *required* to be child nodes of a particular function. A full listing of the DTree object types is given in table 1.

Table 1: DTree Object Types

Type	Accepted Values
Boolean	{ *true, false* }
Integer	{ 0, 1, 2, … 11 }
Vector	{ (x,y,z) | x,y,z are doubles }
Throw	{ (P, V, throwFlag) | P is a vector representing the position of the target; V is a vector representing the velocity of the target; throwFlag is a boolean indicating whether or not this vector is a move or throw instruction }

2.1.3 DTree Genetic Operators

In order to reproduce, the TreesGenome employs three genetic operators: clone(), crossover(), and mutation(). In terms of reproduction, the *move* and *throw* trees are treated as independent populations, hence only *move* trees breed with *move* trees, and only *throw* trees breed with *throw* trees. The various genetic operators are described below.

The Clone Operator

The clone operator works as expected, copying the DTree as is from one generation to the next.

The Crossover Operator

DTree crossover is restrictive in the sense that it can only occur at type-same crossover positions. What this means is that only Boolean returning sub-trees can be switched with Boolean returning sub-trees, only integer returning sub-trees with integer returning sub-trees, and so forth.

Crossover-Point Selection Algorithm

```
SelectCross = on (mom, dad | mom, dad are DTree's)
      momP = rand(1:|mom|)
      while (noValidCrossPoint(mom,momP,dad)
            momP = rand(1:|mom|)
      dadPoint = rand(1,|dad|)
      while (notValidCrossPoint(mom,momP,dad,dadP)
            dadP = rand(1,|dad|)
      return (momP,dadP)
```

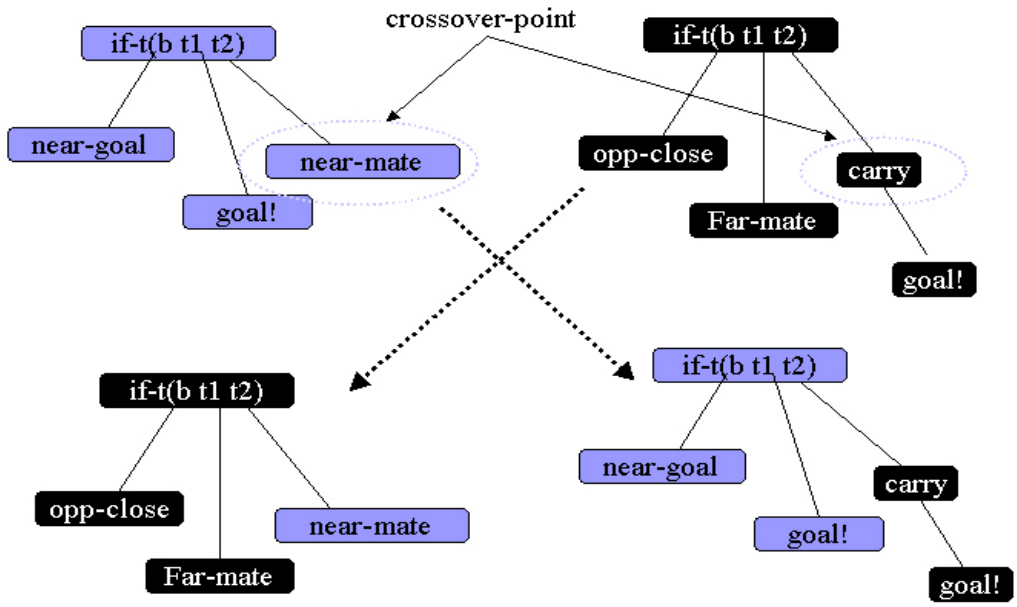

Figure 8: Example DTree Crossover

The Mutation Operator

Mutation in DTree is constructive rather than destructive. For a given genome, therefore, a loaded coin is tossed with a probability P (P = mutation probability) that a mutation should occur. If the coin returns true, a completely new "mutant" tree is randomly generated with the same return type as the original tree. The original tree and the mutant tree then swap sub-trees, using the algorithm described in section 2.2.2. At the end of the process, the tree containing the original tree's top half is kept while the other trees are discarded.

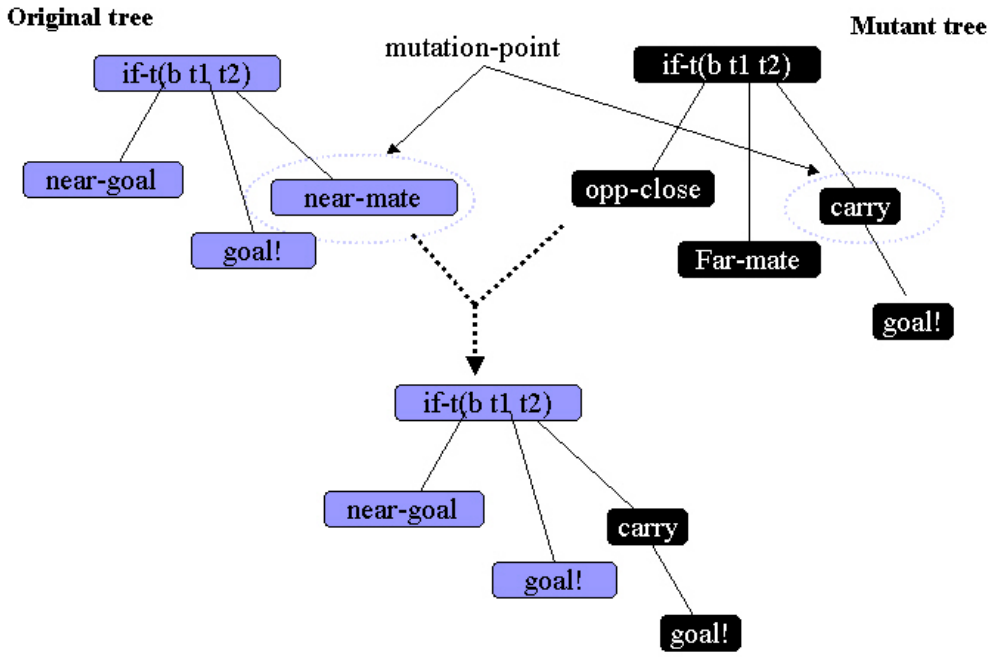

Figure 9: DTree Mutation Example

2.2 DTree Functions and Rationale

DTree contains a combination of conditional (if-V, if-T), logical (AND, OR, NOT), state (boolean), input (vector/throw), and action (vector/throw) functions. The rationale behind this architecture was that the combination of conditional and logical functions would allow the DTree to evolve a set of specific behaviors depending on the state. DTrees would have the capability of evolving behavioral "subroutines" that could be exchanged and modified by crossovers and mutations. In addition, specific action functions could be used to give DTree a leg up in terms of evolution.

2.2.1 Input Functions

DTree input functions are provided by the simulator. Executing in the simulator context these functions provide positional vectors and throws for the player, the player's home location, the player's teammates' positions, the ball, and so forth. A full listing of these functions is contained in Appendix B.

2.2.2 State Functions

Like the input functions, the simulator provides the state functions. Executing within the simulator context these functions provide information about the game state including the following: does the player's team have the ball, is the ball loose, is the ball within a certain distance from the goal, etc. A full listing of these functions is contained in Appendix B.

2.2.3 Action Functions

Action functions were intended to give the Quidditch players a head up in evolution. The rationale behind these functions was that certain behaviors could more quickly be coded by hand than evolved. Given enough time, in theory these behaviors may have come about spontaneously. However, as the evolutionary runs were constricted by time and CPU power, a number of action functions were built into the system. A complete listing of action functions is given in table 2.

Table 2: Action Function Definitions

Function Name	Function Description
(Block-Goal)	Moves the player towards the nearest point on a line-segment between the ball and the nearest defended goal to the ball.
(Steal-Ball)	If an opponent possesses the ball, move to a point in an intercept path with the opponent's trajectory.
(Block-Opp)	Move to the closes point on a line segment between the nearest opponent and the Quaffle.
(Away-Opps)	Move in a trajectory opposite the weighted sum of my opponents' position vectors.
(Away-Mates)	Move in a trajectory opposite the weighted sum of my teammates' position vectors.

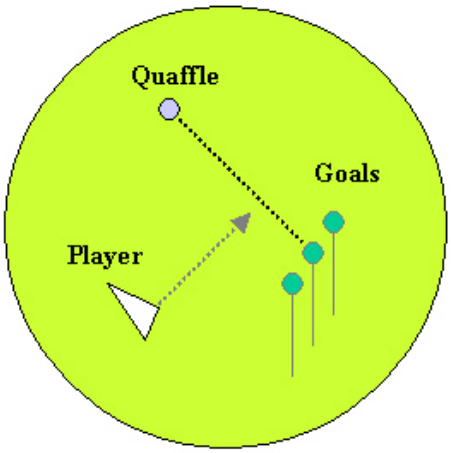

Figure 9: Block-Goal

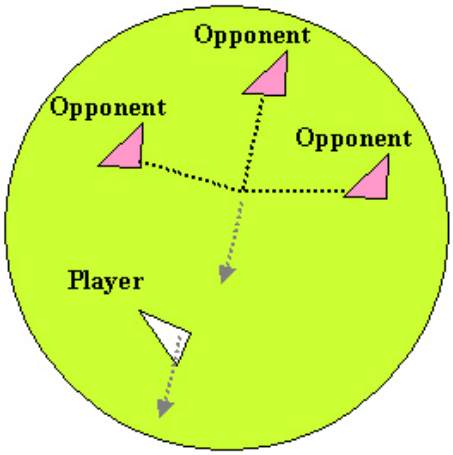

Figure 10: Away-Opps

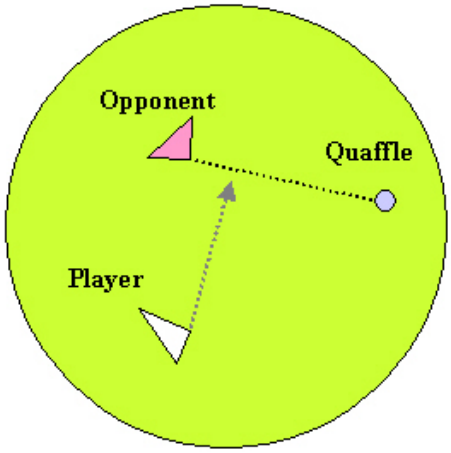

Figure 11: Block-Opp

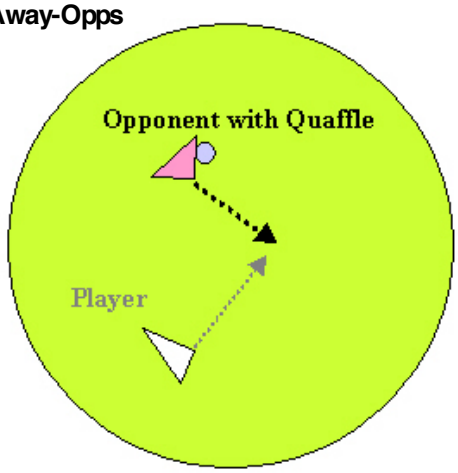

Figure 12: Steal-Ball

2.4 The Quidditch Simulator

The Quidditch Simulator is a Breve simulation derived from the work of Raphael Crawford-Marks, Lee Spector and Jon Klein at Hampshire College (Crawford-Marks 2004). While the initial system implemented the entire game of Quidditch (see Appendix A), the derived simulator omits the non-chaser player and ball elements (i.e. seekers, bludgers, beaters, and the snitch), while retaining the physics engine and game play dynamics of the original simulator. Although sensor and game state information remained unchanged, significant work went into allowing this information to be accessed by the various DTree functions.

2.4.1 Breve

Breve is an open-source software package specifically designed to ease the creation of 3D simulations of decentralized systems and artificial life. Breve includes a set of object primitives and derivable classes that model everything from mobile static objects to joints. Additional methods and classes handle forces, collision detection, and so forth. Documentation, distributions, and further information can be obtained from the project website at <http://www.spiderland.org/breve>.

2.4.2 The (Modified) Game of Quidditch

The Quaffle

Like everything in *Harry Potter,* the quaffle does not abide by the normal laws of physics. According to *Quidditch through the Ages*: "The Quaffle is enchanted to fall as though sinking through water." In addition, rather than needing basketball-player hands to grip, the ball is enchanted with "a "gripping charm" allowing a Chaser to hold the ball with one hand." (GrandPre and Rowling, 1998).

The Keeper

The Keeper is analogous to a soccer goalkeeper. The role of the Keeper is to hang back and defend the goal from attacking chasers. Unlike soccer goalies, the keeper does not have any distinctive rules that apply to her regarding game play, hence they are essentially a fourth chaser except in their role specialization. Hence, for the purposes of this simulator, the keeper is just a fourth chaser.

The Chasers

Despite the influence of the soccer metaphor on Quidditch, the Chasers are best thought of as a team of basketball players. Like a team of basketball players, all the players on the team are responsible for both attacking and defending. Any one of them is able to hold the Quaffle, pass, or attempt to make a shot at the goal.

Game Play

Upon whistle blow the ball is dropped 35 meters off the ground in the middle of the playing field. Players then rush to the center to grab the ball or move to assisting positions. Play continues until 70 points are scored (7 goals at 10 points a goal) or a time limit of 15+(generation/4) simulated seconds (Note: more detailed information regarding the simulator and its inner workings can be found in appendix A).

2.4.3 The Evaluation Function

The Quidditch player fitness function was inherited as is from the Hampshire group. With some modification, the basic algorithm evaluates three aspects of game play: touching the quaffle (.1 points); possession time (.01 * timeSteps of possession) of the quaffle; throwing the quaffle (+.05 * numChasersWhoThrow); goal scoring (+10 per goal); and defense ability (+10*(goalsScored – goalsScoredOn)). In early evolutionary runs possession time, quaffle touching, etc. are intended to direct the algorithm in the right direction. As the players improve, however, goal scoring was intended to replace these aspects as a performance metric.

Chapter 3: Run Results

3.1 Run Setup

Due to time constraints and bugs, only one evolutionary run of the system was completed. The system was tested on a Dell Inspiron 500m® laptop with a 1.3Ghz Pentium M processor and 640 MB of system memory. With this setup, the run involved populations of 100 individuals and took approximately 16 hours. See table 3 for a detail of run parameters.

Table 3: Run Parameter Descriptions

Parameter	Meaning	Value
Ngen	Number of generations to run.	100
Popsize	Size of TreesGenome population	100
baseTime*	Number of seconds to allow games to run for.	15.0

* Note: Due to a bug, all runs actually ran for only 10 seconds.

3.2 Run Results

As would be expected, the early generations of the system produced players that did not perform particularly well. The vast majority of generation 0 would either hover in place or swarm about aimlessly (see figure 13). As one can see in this figure, team 1 (red), does not seem to have moved – in fact they never move from their starting positions. Team 2 (blue), however, is swarming aimlessly about a center of mass. Although the variations are endless, the vast majority of generation 0 players behave in this manner.

Still, even in generation 0, a couple of individuals would actually demonstrate game-playing behavior. Examining figure 15 one finds a team that has been initialized to go straight for the ball. However, once they have the ball they are unsure of what to do with it, throwing it immediately, rather than carrying it to the goal. In another case (figure 14), one individual was even lucky enough to have the ball land on him. In this case, the individual carried it off into the distance before the simulation ended.

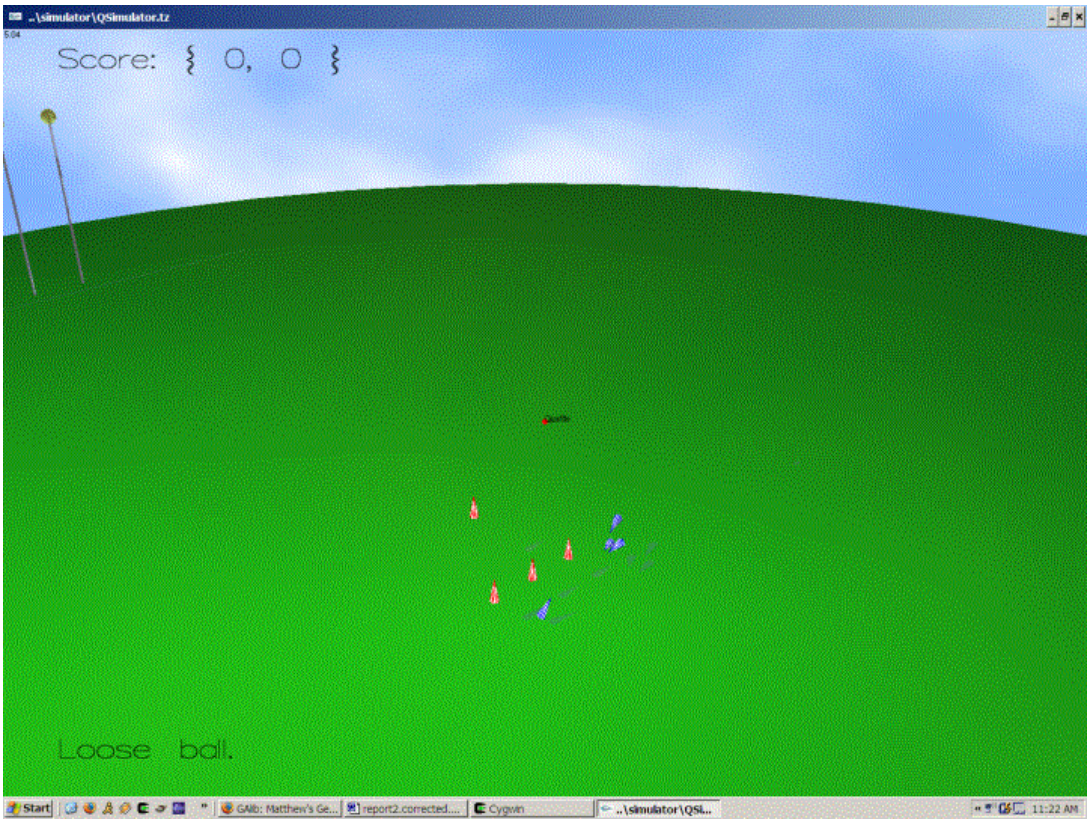

Figure 13: Generation 0 – Random Movement

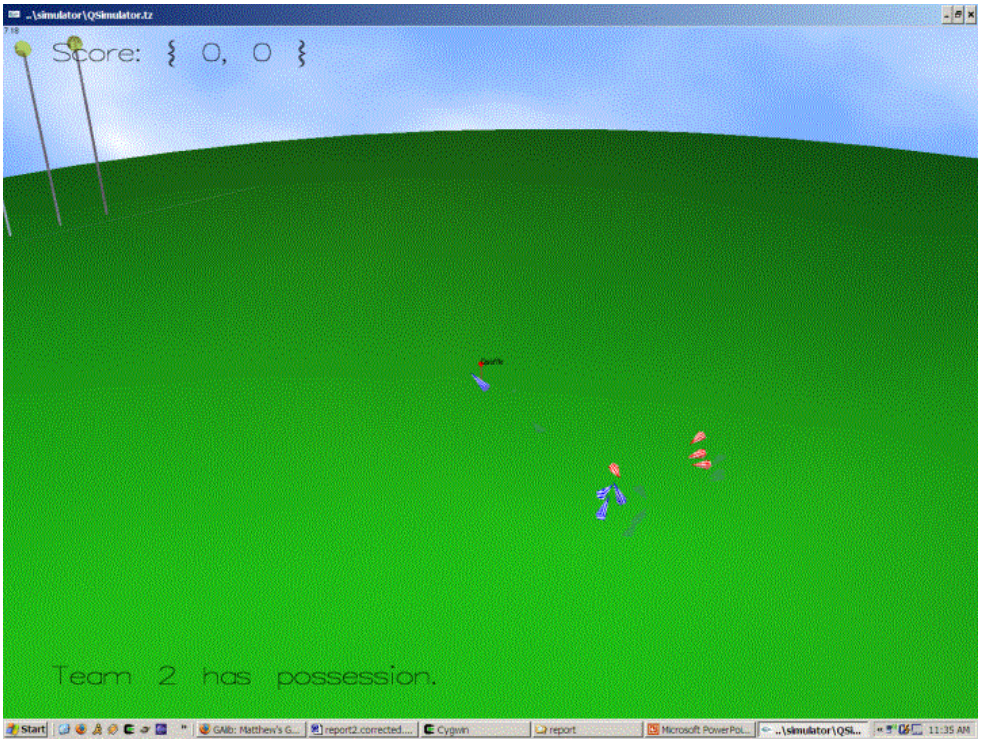

Figure 14: Generation 0 – Carry Behavior

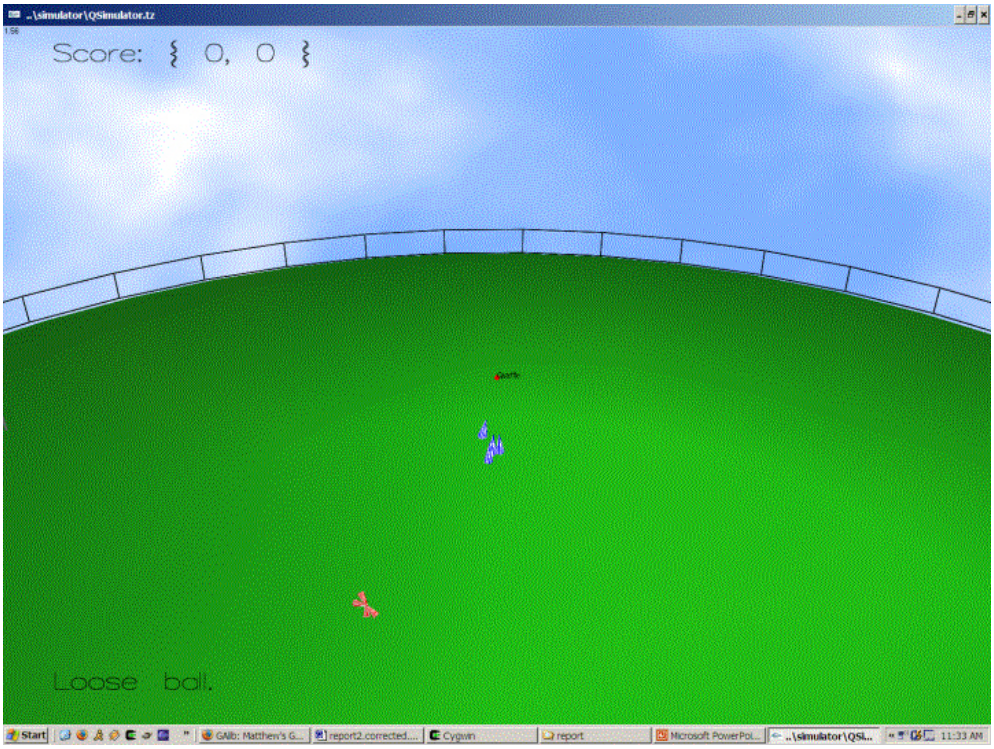

Figure 15: Generation 0 – Go to Ball Behavior

By the intermediate generations (15-20), all the players learned to go for the ball. At this stage, players generally threw the ball immediately at nothing in particular (figure 16). Here both teams of players swarmed the ball, shoving, grabbing, and throwing – just how one might imagine little kids learning how to play soccer.

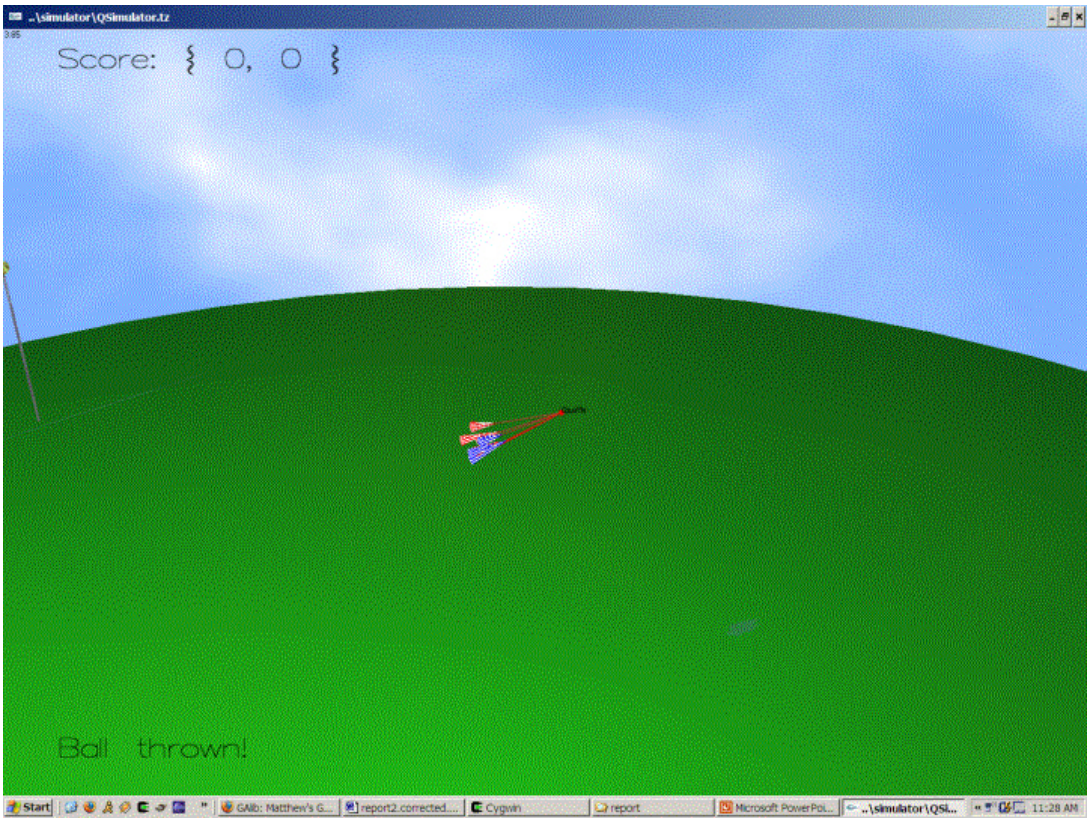

Figure 16: Gen 20 – Swarm Ball Behavior

Gradually, between generation 20 and 100, players learned to hold the ball longer before throwing it (figure 17). Occasional spikes resulted when the player was even fortunate enough to score, which happened rarely, due to a bug in the *generate-throw* method. In addition, game time remained short (limited to 10 seconds) throughout the evolutionary run. Thus, although by generation 100 all the players had learned to hold the ball as well as throw it, with some of the strongest members of the population even making shots on the goal (figure 18), scoring remained elusive.

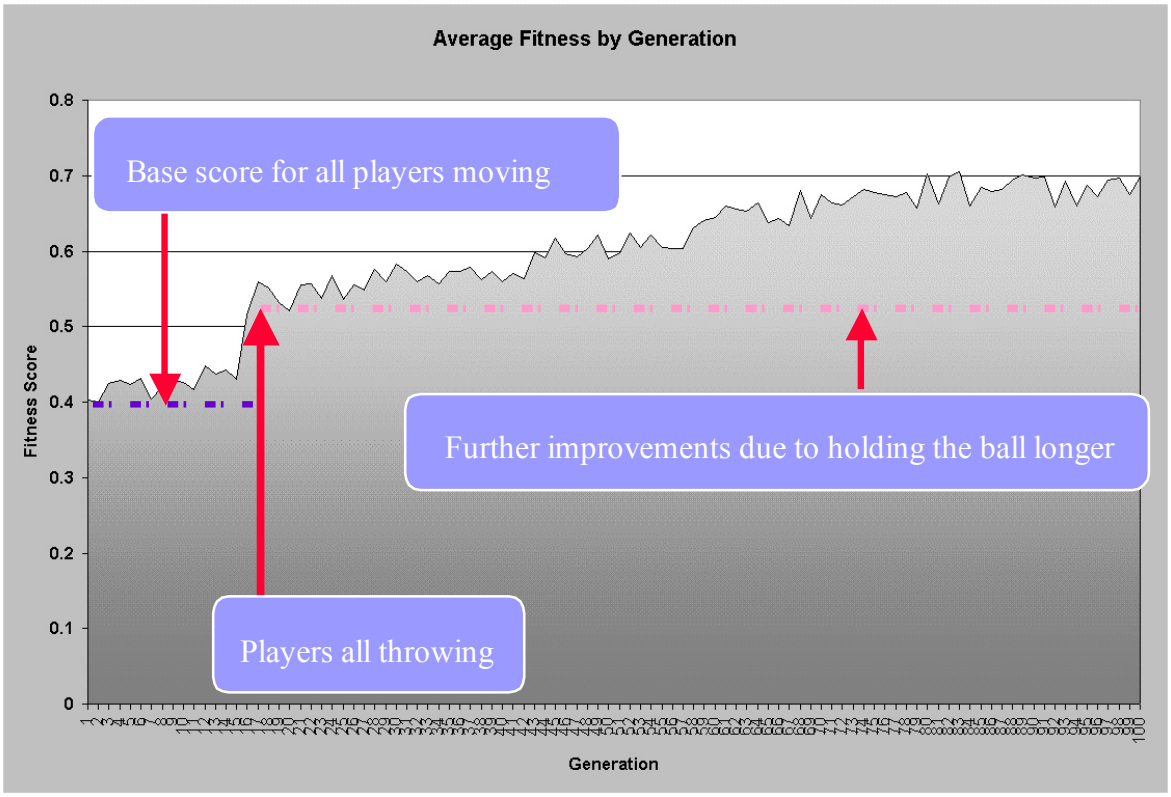

Figure 17: Graph of Run Results

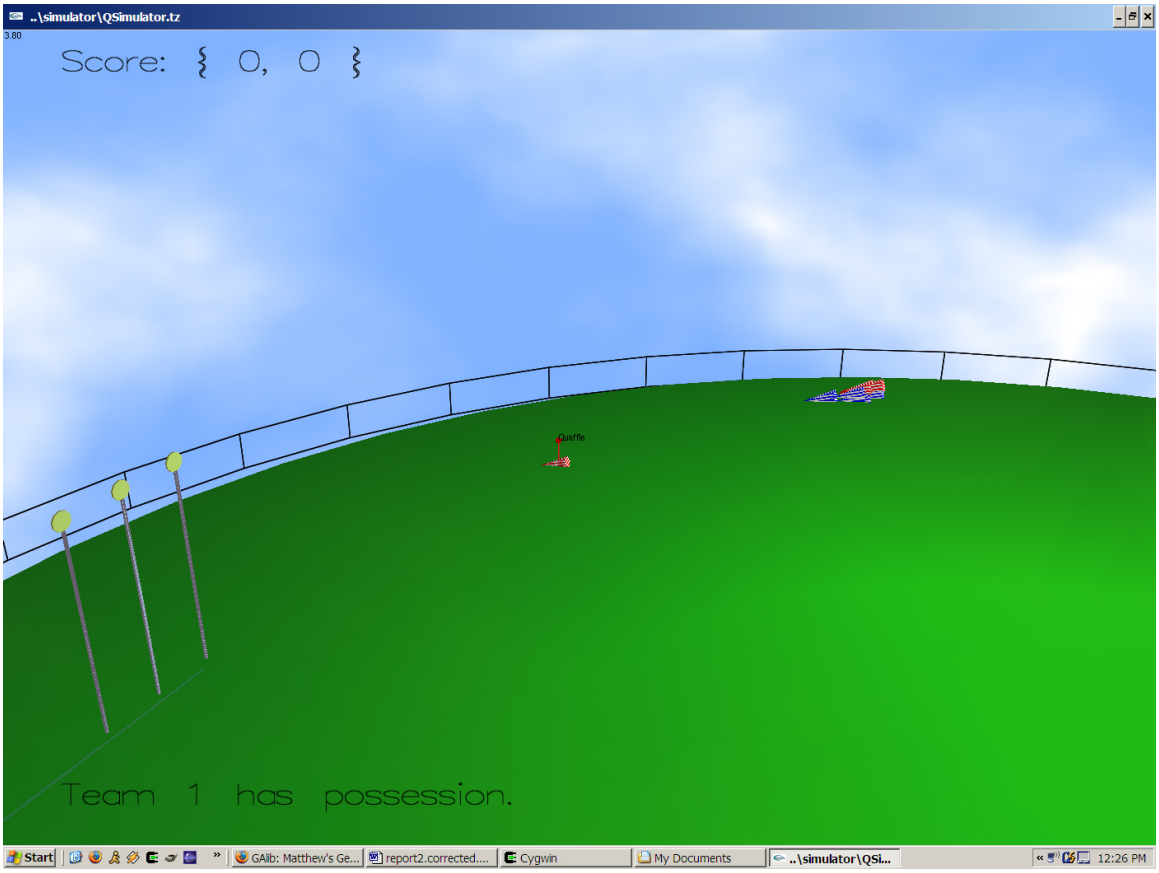

Figure 18: Generation 100 – Shooting on Goal

Chapter 4: Conclusions and Future Work

Even though the system never progressed beyond a version of "kiddie-Quidditch" (charging the ball, dashing for the goal, and throwing), the results were encouraging. Primordial players moved from random behavior uncoordinated activities to a more advanced game strategy. In the initial runs, certain players demonstrated elements of more advanced game strategy, but these elements were never combined or coordinated. During the intermediate stages of evolution, these elements of game play strategy diffused throughout the population, with players learning to go after the ball. Finally, as the system evolved, the later stages of evolution were marked by an increasingly coherent, albeit simple, strategy of game play ("kiddie-Quidditch").

In terms of representation, DTree emerge as the proper representation until well into the research. Later, by the time DTree was mature enough to tackle the modified game of Quidditch, a C++ Quidditch simulator had been under development until the late discovery of Breve, followed soon after by the Hampshire College Quidditch Simulator. Due to these time constraints, a couple of bugs that were discovered during development were not properly addressed. For instance, although game length was supposed to gradually increase in later evolutionary runs, a bug in the system limited game length to 10 simulated seconds. Without longer matches, the most effective means of receiving a high fitness score was simply to maximize the chances of grabbing the ball and throwing it at the goal. Thus, it seems probable that rather than devise some sort of defensive strategykiddie-Quidditch created a local maximum in the score space.

Despite these shortcomings, the evolutionary programming demonstrated itself as a promising machine learning method for addressing the problem of playing virtual Quidditch. Even given the immaturity of the DTree system and the integration complications posed by Breve, the system still successfully evolved a simple strategy for game play. Given further time and improvements, the system promises to advance beyond the limits of kiddie-Quidditch.

4.1 Future Work

Future work can be roughly divided into two categories, either involving the simulator or involving the Evolver. Simulator improvements center around increasing realism while evolver improvements concern bug fixes and AI performance (Crawford-Marks, 2004).

4.1.1 Evolver Improvements

As mentioned in the beginning of this chapter, the most pressing issue to be addressed by future work is the timing bug. Beyond that, a number of the behavioral functions, most notably the generate-throw function seems to perform less than optimally. Although there was little discussion of these functions, in terms of game states, there is a class of probabilistic game-state functions, such as "throw-near-goal-if", that were based on (C * (1 / dist)) approximations of throw accuracy. In addition to improving the throw function, more careful analysis of these probabilities should be carried out.

Beyond these bugs, the distinction between the *Throw* and *Move* trees is somewhat questionable. Rather than using the throw tree to determine movements, it may make more sense to execute the move tree at each time step and use the throw tree just to determine where and whether to throw the ball.

Next, in terms of implementation language, the choice of C++ was less than optimal. Although DTree proved fairly robust, simulating the LISP stack required a large coding overhead. Looking at Push, which integrates seamlessly with Breve, and Java for which there are a number of GA packages, the implementation language of DTree should be reconsidered.

Finally, at some point more heterogeneity should be added to the system. Rather than just having one Chaser brain, experiments should be conducted coevolving a team with multiple chaser brains. Gradually, after this system could be made to work (or proven inferior), further work should be done to implement a full seven-player Quidditch team and answer the full challenge proposed by Spector's paper.

4.1.2 Simulator

Architecture – As it stands now, the simulator does not control the player input as tightly as could be desired. In order to increase the control of the input, the simulator should probably be moved to a client/server model. Although there is a bit of network overhead in this model, by having an always-running simulator, approximately 1.5 seconds out of a 4.5 second simulation run is spent on system initialization, this delay could be removed. Thus, given that messages would likely be short, moving to such an architecture could actually result in significantly faster simulation runs.

Vision-Like Sensors - In terms of realism, sensor functions should be made more "vision-like" in the sense that players should have a limited field of vision. Although there is a drop-off in the accuracy of player vision due to distance, vision is currently omni-directional.

Communication – Currently there is no way for players to communicate their intentions with one another. Implementing *say* and *hear* methods on the server would add another level of realism to the simulation.

Appendix A: Quidditch and the Hampshire College Simulator

NOTE: The following section is from Raphael Crawford-Marks' division III thesis, pp. 14-24. The text was modified slightly due to table numbering changes, but is essentially as is for the purposes of reference.

The Game of Quidditch

Balls

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

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 feed 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 16 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 game play, to facilitate evolution, and to reduce simulator complexity.

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 signaling 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.

Quidditch Rules 1

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 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.

Pitch Size and Shape - The pitch is circular instead of ovular, 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.

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
Bumphing	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 seeker	Any player other than Seeker touching or catching the Golden Snitch
Stooging	Chasers only	More than one Chaser in the scoring area

Table 4: Common Quidditch Fouls

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) 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.

Table 5: Quidditch Game States

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

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 a GoalScored event, and stores its ID in the GoalScored event. In the Score-Tracker'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.

Appendix B: Full DTree Quidditch Function Listing

Logical and Integer Functions

Key: t – throw, i – integer, b – boolean, v – vector. *Max* is the maximum throwing distance.

(if-t b t1 t2)	T	If *b* is true return *t1*, else return *t2*.
(if-v b v1 v2)	V	If *b* is *true* return *v1*, else return *v2*.
(if-i b i1 i2)	I	If *b* is *true* return *i1*, else return *i2*.
(and b1 b2)	B	If *b1* & *b2* return *true*, else return *false*.
(or b1 b2)	B	If *b1* || *b2* return *true*, else return *false*.
(not b)	B	If *b* return *false*, else return *true*.
(0,1,2,3,4,5,6,7,8,9,10,11)	I	Constant integer values.

Boolean State Functions

(opp-closer)	B	*True* if opponent closer to the ball, else true.
(mate-closer)	B	*True* if a teammate is closer to the ball, else false.
(opp-near-score i)	B	*True* if opponent has the ball within *i*/10 units of the goal I defend.
(team-near-score i)	B	*True* if a player on my team has the ball within *i*/10 units of the goal I am seeking to score on.
(team-has-ball)	B	*True* if players team has the ball, otherwise *false*.
(opp-team-has-ball)	B	*True* if opposing team has the ball; otherwise *false*.
(mate-has-ball i)	B	*True* if teammate THIRD(i) has the ball; otherwise *false*.
(opp-has-ball i)	B	*True* if opposing team has the ball; otherwise *false*.
(ball-loose)	B	*True* if no team possesses the.
(of-me i)	B	Return *true* if ball is within *i* units of me, else *false*.
(of-home i)	B	Return *true* if ball is within *i* units of my home, else *false*.
(of-goal i)	B	Return *true* if ball is within *i* units of the target goal, else *false*.
(opp-close i)	B	Return *true* if an opponent is within *i* units of me.
(mate-close I)	B	Return *true* if a mate is within *i* units of me.
(team-has-ball)	B	Return *true* if one of my teammates has the ball. Otherwise return *false*.

Vector Input Functions

Function Syntax	Returns	Description
(home)	V	A vector to my home.
(home-mate i)	V	A vector to the home of teammate THIRD(*i*)
(ball)	V	A vector to the ball.
(goal i)	V	A vector/throw to the goal THIRD(*i*).
(closest-goal)	V	A vector/throw to the goal.
(mate I v)	V	A vector to teammate THIRD(*i*).
(move t)	V	Convert a throw vector to a move vector.

Vector Action Functions

Function Syntax	Returns	Description
(block-goal)	V	A vector to the closest point on the line segment between the ball and the goal I defend.
(away-mates)	V	A vector away from known teammates, computed as the inverse of Sigma{m{vect teammates} frac{max - ‖ m ‖}{‖m‖} * m }
(away-opps)	V	A vector away from known opponents, computed as the inverse of Sigma{m{vect teammates} frac{max - ‖ m ‖}{‖m‖} * m }

Throw / P-Throw Functions

Function Syntax	Returns	Description
(far-mate i t)	T	A throw vector to the most offensive-positioned teammate who can receive the ball with at least a frac{*i + 1*}{12} probability. Otherwise return *t.*
(near-mate i t)	T	A throw vector to the most offensive-positioned teammate who can receive the ball with at least a frac{*i + 1*}{12} probability
(mate-m i1 i2 t)	T	A throw vector to teammate mate(*i1*)if her position is known and she can receive the ball with at least a frac{*i2* + 1}{12} probability. If not, return *t.*
(throw-goal-if i1 i2 t)	T	A throw vector to the goal {THIRD(I)} if throw will be successful within frac{*i*+1}{12} probability, otherwise return *t.*
(throw-near-goal-if i)	T	A throw vector to the nearest goal if throw will be successful within frac{*i*+1}{12} probability, otherwise return *t.*
(carry v)	T	Converts a vector to a move throw.
(carryT t)	T	Turn a throw vector into a carry vector.
(throw v)	T	Returns a throw vector towards the vector v.

Bibliography

Crawford-Marks, R. *Virtual Witches and Warlocks*. Thesis. Hampshire College, 2004. 10 Apr. 2005 <http://alum.hampshire.edu/~rpc01/div3.pdf>.

Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning* (1989). Addison-Wesley Professional: New York, NY.

Klein, J. (2005). *Breve Documentation.* Retrieved 4 Apr. 2005 from the Breve web site: http://www.spiderland.org/breve/breve_docs/.

Koza, J. R., *Genetic Programming*. Cambridge, MA: MIT Press, 1992.

Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J., "Co-evolving softbot team coordination with genetic programming," in *Proceedings of the First International Workshop on RoboCup, at the International Conference on Artificial Intelligence*, (Nagoya, Japan), 1997.

Mitchell, T., *Machine Learning*. Boston, MA: McGraw-Hill, 1997.

Rowling, J. and Grandpre, M., *Harry Potter and the Sorcerer's Stone*. Scholastic, Inc., 1998.

Spector, L., Moore, R., and Robinson, A., "Virtual Quidditch: A challenge problem for automatically programmed software agents," Hampshire College 2001. 2 Apr. 2005 <http://hampshire.edu/lspector/pubs/quidditch-cite.pdf>.

Wall, M (2004). *GALib Documentation.* Retrieved Sept. 10, 2004, from the GALib web site: http://lancet.mit.edu/galib-2.4/.

Whisp, K. and Rowling, J., *Quidditch through the Ages*. New York, NY: Scholastic Press, 2001.