Learning in *RoboCup* Keepaway using Evolutionary Algorithms

Anthony Di Pietro **Lyndon While** **Luigi Barone**

Department of Computer Science & Software Engineering
The University of Western Australia
Western Australia 6009
Email: {anthony, lyndon, luigi}@cs.uwa.edu.au
Telephone: +61 8 9380 2720

Abstract

Manually coordinating the efforts of many autonomous agents can be a formidable challenge, so the idea of using machine learning techniques (such as evolutionary algorithms) to produce such coordination is attractive. We present a study using evolutionary algorithms to train autonomous agents to play the game of *keepaway* — a sub-problem of the *RoboCup* robotic soccer league. Our results exceed those previously produced using other methods.

1 INTRODUCTION

Multi-agent systems is an active area of artificial intelligence research that focuses on the interaction of artificially intelligent agents. Manually coordinating the efforts of many autonomous agents can be a formidable challenge, so the idea of using machine learning to produce such coordination is attractive.

Team sports, such as soccer, involve autonomous humans collaborating to achieve a common goal. The *RoboCup* initiative challenges AI researchers to achieve this same collaboration among autonomous software agents in a complex, noisy, real-time environment.

Evolutionary algorithms are a robust technique for learning in such environments. A population of candidates is generated with an initial solution encoded into each candidate, and the population evolves to produce better solutions.

This paper describes a study where autonomous agents learn to play the game of *keepaway* (a sub-problem of the *RoboCup* robotic soccer league) using evolutionary algorithms, and compares our approach to others that have been used previously.

Section 2 describes the *RoboCup* set-up, introduces the keepaway sub-problem, and discusses previous work in this area. Section 3 describes the evolutionary algorithm that we use and gives details of its various components, especially the genetic representation (Section 3.2) and the fitness function (Section 3.3). Section 4 describes our experiments and results, and Sections 5 and 6 conclude the paper.

2 THE *ROBOCUP* SIMULATION LEAGUE

The *RoboCup* initiative (Kitano et al., 1997) aims to create a robot soccer team that can beat the human world champion team by the year 2050. It is hoped that this ambitious goal will promote research and collaboration in a variety of fields. There are several *RoboCup* leagues for physical robots of different sizes, and there is also a simulator league in which software agents connect via a network to the *RoboCup* soccer server and play in a virtual environment.

The soccer server provides a virtual soccer field, simulates the physics of the ball and players, and enforces the rules of the game. The field, physics, and rules used by the server are based on those of real soccer, but there are significant differences: for example, the goal width is doubled and the world is two dimensional — and the referee always makes perfect judgements! The soccer server communicates with the players via a client/server protocol, allowing them to receive information about the game state and send commands to perform actions. The game state available to the players is incomplete: a player receives only the information that would be available to a real player in their position. Furthermore, each client may control only one player, and players are allowed to communicate only with limited virtual speech via the soccer server, so coordination among agents is difficult. The soccer server manual (Chen et al., 2001) states,

> One of [the] purposes of `soccerserver` is [the] evaluation of multi-agent systems, in which efficiency of communication between agents is one of the criteria. Users must realize control of multiple clients by such restricted communication.

The soccer server also incorporates complex game parameters (such as stamina and inertia), and adds noise to the movement of objects and to the players' senses.

RoboCup tournaments are held regularly, allowing researchers to demonstrate the effectiveness of their techniques by competing against other teams in a complex, noisy, real-time environment.

2.1 THE KEEPAWAY SUB-PROBLEM

While the *RoboCup* competition itself presents a valid application for machine learning, its complexity makes many machine learning experiments prohibitively time-consuming; better/faster experimental results can be expected for simpler problems.

One such sub-problem in *RoboCup* is the game of *keepaway*. In keepaway, one team (the *keepers* or *forwards*) starts with possession of the ball, and the other team (the *takers* or *defenders*) must take the ball from them. The game ends when a taker gets possession of the ball, or when the ball leaves the area of play. The keepers' objective is to maximise the duration of the game, while the takers' objective is the opposite.

Figure 1 shows a keepaway game in progress.

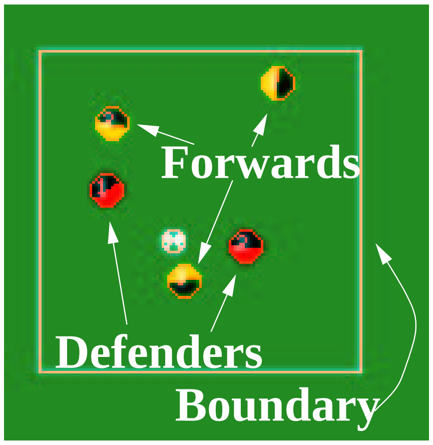

Figure 1: A 3 vs. 2 Keepaway Game In Progress (Diagram From Stone et al. (2001)).

2.2 MACHINE LEARNING IN KEEPAWAY

Thus far, the best *RoboCup* teams have been manually programmed agents (Reis and Lau, 2001; Stone and McAllester, 2001). These teams were created based on the assumption that because *RoboCup* is a simulation of real soccer, implementing strategies known to work in real soccer will result in a good *RoboCup* team.

However, this fails to account for the fundamental differences between *RoboCup* and real soccer, and the high-level differences that these create. Also, the *RoboCup* rules and soccer server are constantly changing, so maintaining a manually programmed team can be difficult. Hence the idea of applying machine learning techniques to *RoboCup* is gaining ground (e.g. (Luke et al., 1998; Andre and Teller, 1999))[1].

Stone et al. (2001) used reinforcement learning to train keepers for 3 vs. 2 keepaway (i.e. three keepers vs. two takers) on a 20m $\times$ 20m playing field. They used a software coach to set up games, enforce the rules, terminate the games, and provide feedback to the players. They defined the following high-level behaviour functions for keepers to use.

HoldBall(): Remain stationary while keeping possession of the ball and turning it such that it is as far away from the opponents as possible.

PassBall(*f*): Kick the ball directly to teammate *f*.

GoToBall(): Intercept a moving ball or move directly to a stationary ball.

GetOpen(): Move to a position that is free from opponents and open for a pass from the ball's current position (using SPAR (Veloso et al., 1998)).

The keepers' decision-making process was simplified by using the policy space shown in Figure 2. The takers always called `GoToBall()` (or `HoldBall()` if they already had the ball). This meant that decisions were required only for the keeper in possession of the ball, and that the decision was a choice of three possible actions: `HoldBall()`, `PassBall(1)` and `PassBall(2)`.

Stone *et al.* also implemented three benchmark policies: making random decisions; always holding the ball; and a manually-coded policy that holds the ball unless a taker is within 10m *and* a safe pass is available. The reinforcement learning generated policies

[1]We describe here only applications of machine learning to keepaway.

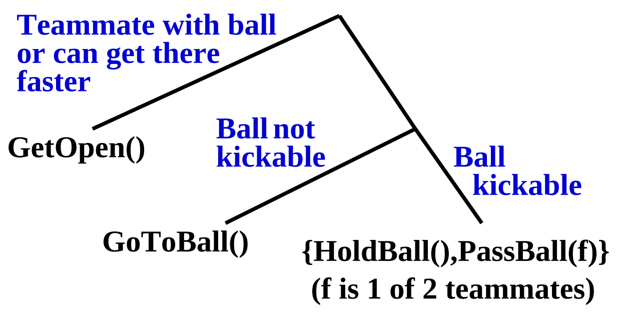

Figure 2: The Keepers' Policy Space (Diagram From Stone et al. (2001)).

that achieved game durations of 14 seconds, whereas all of the benchmark policies (including the manually-coded one) could manage durations of only 5 seconds.

Gustafson (2001) used genetic programming (Koza, 1992; Luke et al., 1998) to evolve keepers for 3 vs. 1 keepaway, using the *TeamBots* (Belch) simulator. Games were of a fixed duration and the keepers' objective was to minimise the number of turnovers per game. He also used somewhat different physics: the ball moved twice as fast as the takers, which moved twice as fast as the keepers.

Using his best layered learning genetic programming parameters, Gustafson produced a final population with a mean fitness of 70 turnovers, and a best fitness of 9 turnovers. These results are not easily comparable to Stone's because the problem domain and fitness evaluation are fundamentally different.

2.3 OUR KEEPAWAY SET-UP

Following Stone *et al.*, we generated keepers for 3 vs. 2 keepaway on a 20m $\times$ 20m playing field in the *RoboCup* soccer server. We reconfigured the soccer server to use 50ms cycles, i.e. to run at double speed, without otherwise changing the mechanics of the game. We based the takers on the freely-available *CMUnited-99* (Stone et al., 2000) source code that always called `GoToBall()` (or `HoldBall()` if it already had the ball).

We developed a software coach to set up games, enforce the rules, terminate games, and provide feedback to the players. A game ended when the ball went out of bounds, or when the ball had been within the kickable area of at least one taker for 5 or more consecutive server cycles. We introduced the additional constraint that if a game lasted for 600 server cycles (one minute of *RoboCup* game time), it would automatically be terminated. This constraint was added to impose a limit on degenerate luck in the noisy *RoboCup* environment.

3 EVOLUTIONARY ALGORITHMS

Evolutionary computation is a broad term that encompasses all methods of using the principles of biological evolution (Darwin, 1859) to solve problems on a computer.

We chose an evolutionary algorithm (EA) for this study because they are known to work well in noisy and unknown domains (Darwen, 2000). In an EA, the population is modelled as a set of *candidates*, and the parameters that vary between individuals are encoded into the candidates as genetic attributes. These attributes may be initialised with random or other values. The evolutionary process is then simulated to produce new candidates that represent better solutions.

To instantiate this technique for a particular problem, we have to specify the representation used for candidates' keepaway policies, the fitness function used to evaluate policies, and the operators and parameters used for reproduction, mutation and selection. We describe these aspects of the set-up in Sections 3.1–3.3.

3.1 EVOLUTIONARY OPERATORS

We used a $(1 + 1)$ evolutionary strategy. In each generation, each candidate produces one child, then the population members for the next generation are chosen from the combined parents and children populations.

Reproduction is simulated by creating a duplicate of the candidate with mutation. We implemented mutation using a Gaussian random distribution with zero mean difference. The standard deviation of the distribution determines how aggressively the system tries to evolve. We used a value of 0.1, resulting in about 18% of children surviving into the next generation.

The selection function determines which candidates will survive into the next generation , and which will die. We used a simple scheme where the n candidates with the highest fitness ratings survived (from the $2n$ parents+children available).

3.2 PROBLEM REPRESENTATION

We allowed our keepers the same high-level actions at each cycle as Stone's (see Section 2.2). However, because these high-level primitives (and the corresponding low-level skills) were independently implemented, our keepers could be expected to perform differently and to learn different strategies. Furthermore, whereas Stone *et al.* used SPAR (Veloso et al., 1998) for their `GetOpen()` function, we used a superior algorithm given in Stone and McAllester (2001).

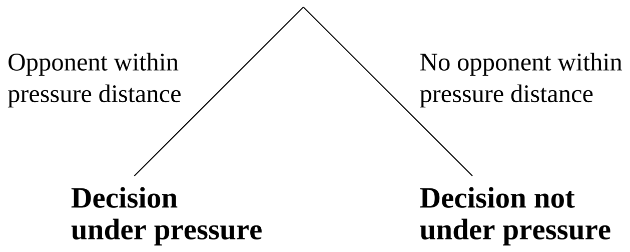
Opponent within pressure distance
No opponent within pressure distance
Decision under pressure
Decision not under pressure

Figure 3: The Agents' Overall Decision-Making Framework.

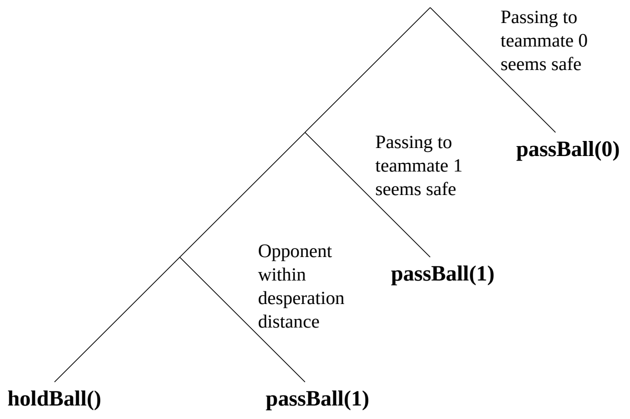
Passing to teammate 0 seems safe
passBall(0)
Passing to teammate 1 seems safe
passBall(1)
Opponent within desperation distance
holdBall()
passBall(1)

Figure 4: The Agents' Decision-Making Framework When Under Pressure.

Our keepers used the same policy space as Stone's (Figure 2), thus a decision was required only for the keeper in possession of the ball. We further decomposed this case using the decision-making framework shown in Figures 3, 4, 5. This is a simplistic initial framework that is likely to be extended in the future. Note that `teammate 0` is the teammate nearest to the ball and `teammate 1` is the other teammate.

If there is an opponent within the *pressure distance* (an evolved parameter), the agent acts "under pressure" (Figure 4); otherwise, it acts "not under pressure" (Figure 5).

When "under pressure", the agent wants to pass the ball. If passing to `teammate 0` seems "safe", it will do so; otherwise, it will consider passing to `teammate 1`. If neither pass seems "safe", the agent holds the ball, unless there is an opponent within the *desperation distance* (another evolved parameter), in which case it clears the ball by passing to `teammate 1`.

When "not under pressure", the agent will pass only if it can improve the strategic utility of the state by centralising the ball. It passes to the most central teammate that is "very safe" to pass to, if any.

A pass is deemed safe only if the following three values are all "large enough":

- the distance to the recipient;
- the minimum angle formed by the recipient, the ball, and each opponent; and
- the distance between the recipient and each opponent.

Each of these distances and angles is an evolved parameter. Thus the decision-making framework used a total of twelve evolved parameters:

- the pressure distance;
- the desperation distance;
- five parameters for assessing passes to `teammate 0`: two distances and an angle for acting "under pressure", plus one distance and an angle for acting "not under pressure"; and
- the same five parameters for `teammate 1`.

Each parameter is a real value in the range $[0, 1]$, representing a proportion of the range available. Distances were scaled by the maximum diagonal length of the field, and angles were scaled by $180°$ (the maximum absolute angle size). The only exception to this was that the desperation distance was scaled by the pressure distance. This forced the desperation distance to be less than the pressure distance, so that there was always some taker-proximity at which the agent would clear the ball.

3.3 FITNESS EVALUATION

The noisy environment of the *RoboCup* soccer server induces noisy fitness evaluations. Beyer (2000) states that coping with noisy fitness evaluations in an EA is still in its infancy. He describes three techniques, the simplest of which are increasing the population size, and resampling and averaging the fitness.

As the population size and the number of fitness samples/evaluation are increased, generations take longer to simulate. Thus one must choose a balance between population size, number of samples/evaluation, and number of generations simulated. Darwen (2000) claims that to obtain the best result from the available CPU time, one should use a "generously large" population, and that the number of fitness samples used should be just enough such that using more does not improve learning (clearly this is problem dependent).

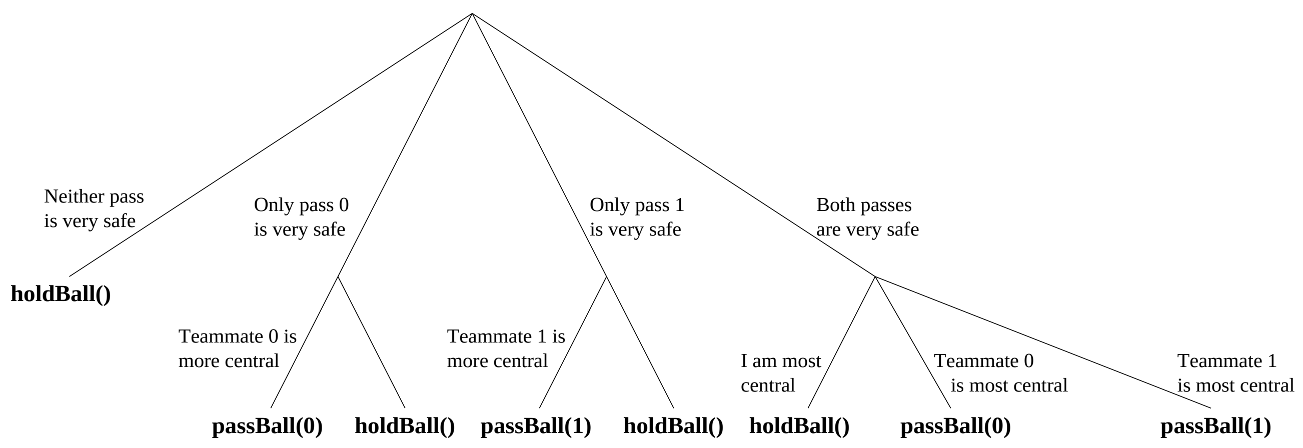
Neither pass is very safe
Only pass 0 is very safe
Only pass 1 is very safe
Both passes are very safe
holdBall()
Teammate 0 is more central
Teammate 1 is more central
I am most central
Teammate 0 is most central
Teammate 1 is most central
passBall(0)
holdBall()
passBall(1)
holdBall()
holdBall()
passBall(0)
passBall(1)

Figure 5: The Agents' Decision-Making Framework When Not Under Pressure.

The noisy fitness evaluations of keepaway forced us to reevaluate all candidates in each generation, otherwise a mediocre candidate might bias the population by receiving a fortuitously good fitness. However, this reevaluation is very expensive. If we use n samples/evaluation in a population of size p, and if each candidate is completely reevaluated in each generation, this requires a total of $2np$ samples/generation.

We used a different approach based on keeping a *moving average* of the candidates' samples. When a candidate is first generated (as a child), its fitness is sampled n times, and its *fitness array* is initialised with n copies of the average of these samples. In each subsequent generation, the fitness is sampled once only, and this new sample replaces the oldest sample remaining in its fitness array. The fitness estimate for a candidate in a given generation is the average of the samples in its fitness array at that time, ignoring outliers.

This technique reduces the number of samples/generation from $2np$ to $(n + 1)p$, allowing us to run nearly twice as many generations in a given run-time. Note that this technique is applicable only where candidates can persist in the population.

In addition to the noisy environment of the *RoboCup* soccer server, fitness evaluations are affected by the choice of team members. Fitness samples are assigned equally to all players on a team. A good player that is teamed with two inferior players is unlikely to achieve a good game duration; conversely, a bad player that is teamed with two superior players is likely to achieve a good game duration. To minimise this effect, before each round of reevaluation we randomised the order in which the candidates were scheduled for evaluation. Thus if a player's fitness was sampled n times in a generation, it would play in n randomly selected teams.

Each keepaway game starts with one keeper in each of three corners. Both takers are placed in the other corner, and the ball is dropped in a randomly-selected corner occupied by a keeper. This means that one of the keepers will always start between two teammates and opposite the takers. This creates the potential to evolve specialised roles depending on player position. To exploit this potential for specialisation, we used a different population for each of the three keepers. This meant using three small populations instead of one large population.

4 RESULTS AND COMPARISONS

Darwen (2000) suggests that two important parameters that effect EA performance in a noisy environment are population size and number of fitness samples/evaluation. Our first round of experiments focused on varying these parameters to determine the range of parameters which behaved the "best". We experimented with population sizes ranging from 3 to 200, with the number of fitness samples ranging from 1 to 200. We found that smaller population sizes did not maintain enough diversity to cope with the noisy fitness evaluations, while larger population sizes learnt more slowly with no apparent benefit. Similarly, using too few fitness samples resulted in a fitness approximation that was too noisy and produced erratic results, while using too many fitness samples resulted in slower learning with no apparent benefit. This round of experiments lead us to conclude that a population size of 10 to 40 should be used, with samples/evaluation between 5 and 10.

Our second round of experiments tested each of the population sizes 10, 20, 30, and 40, with 5, 7, and 9 fitness samples. We found that a population size of 20 with 7 fitness samples learned fastest, but were concerned that the population size may be too small to maintain diversity (hence limiting further improvement), and to retain what had been learnt (exposing the population to the danger of a poor candidate being awarded an over-generous fitness due to limited sampling). We observed that in the experiments with a population size of 10, the populations almost always lost diversity after their performance levelled off, sometimes resulting in a degradation in fitness due a "lucky" candidate dominating the small population. Populations of size 20 occasionally exhibited this behaviour, but to a lesser extent, and with swifter recovery. Populations of size 30 never exhibited any significant problems with fitness retention. Populations of size 40 learnt more slowly with no apparent benefit.

Based on these experiments, we concluded that the fastest learning speed was obtained using a population size of 20 with 7 fitness samples, but that for a reliable balance between learning speed and fitness retention, a population size of 30 with 9 fitness samples should be used. We then ran a third round of experiments using population sizes of 20 and 30, with 7 and 9 fitness samples, over a large number of runs, to confirm that our results could be reliably reproduced.

Finally, in our fourth round of experiments, we took a typical run with a population size of 30 and 9 fitness samples, and retroactively evaluated each member of the initial and final populations over thousands of games to accurately estimate their actual fitnesses. Meanwhile, we ran experiments with population sizes of 40 with 9 fitness samples for a longer time to confirm that the runs with a population size of 30 with 9 fitness samples were finding good solutions.

Figure 6 shows a scatter-graph of the surviving candidates from each generation of one keeper population for a run using a population size of 30 and 9 fitness samples. We observed similar trends in the populations of the other two players. The lines represent the best (maximum), average, and worst (minimum) fitnesses of the surviving candidates of each generation. The time for the run was 53 hours, (67 generations).

The highest average fitness was approximately 300 cycles (30 seconds of *RoboCup* game time). This was attained within 25 generations, or 20 hours of running time, after which the average fitness levels off. The degradation in fitness from generations 30 to 55 is a result of noise in the fitness evaluations. The first generation in which the best fitness was greater than 300 cycles was generation 8, which corresponds to 6.4 hours of running time; however, because the fitness evaluation is noisy, it is likely that the "real" fitness of this candidate was lower.

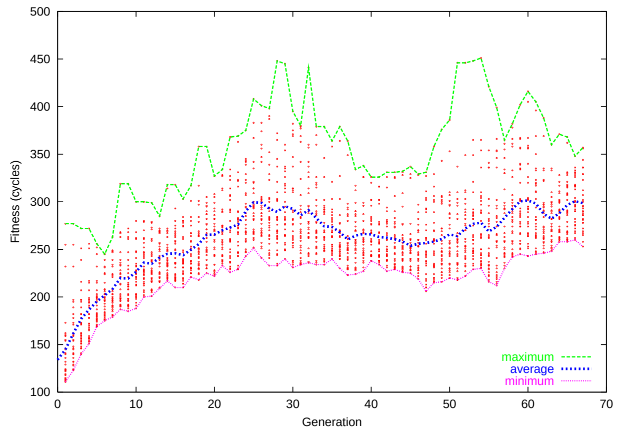

Figure 6: Fitness Versus Generations Using A Population Size Of 30 With 9 Fitness Samples.

The scatter-graph reveals that no candidates obtained an estimated fitness greater than 350 cycles in the first 15 generations. This is significant because it suggests that some learning is required before such high fitness estimates can be attained. Thus although there is an element of luck in playing keepaway, skill is an overriding factor in determining the game duration and hence a candidate's chance of survival (at least initially).

Figure 7 plots the distribution of the "accurately" estimated fitnesses for the initial and final populations from Figure 6. Note that the plotted accurately estimated fitness differs from the EA's coarse estimation (9 fitness samples); an accurate fitness estimate is determined retroactively by testing the candidate for thousands of fitness samples (to within ± 1 second).

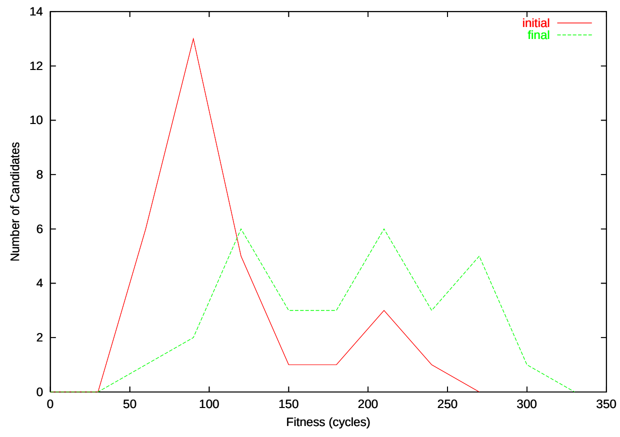

Figure 7: Fitness Distribution Of The Initial And Final Populations From Figure 6.

Retroactive testing of the final population showed that although the accurately estimated fitnesses of the surviving candidates were usually lower than the coarsely estimated fitnesses, there were several candidates with high accurately estimated fitnesses, and in some cases the accurately estimated fitness was higher than the coarsely estimated fitness. The candidate with the best accurately estimated fitness was rated the 6^{th} best member in the population by the EA. We call this candidate *Stuart*. We computed the 95% confidence interval for the mean fitness of this candidate as [319, 339] cycles. We call the second best candidate (using the accurately estimated fitness) in the final population *Bowser*, and the third best candidate *Vince*.

We also retroactively tested the initial (randomly generated) population to accurately estimate the average fitness before learning. The average fitness of the initial population was 128 cycles.

We were concerned about the high fitnesses of some of the members of the initial population, so we ran additional experiments (with a population size of 30 and 9 fitness samples) that began with a poor initial population. We confirmed that the same fitness values were learnt from the poor initial population, suggesting that our results are independent of the initial population.

Table 1 compares the results of this work with that of the reinforcement learning approach undertaken by Stone et al. We observe that Stone was able to improve game duration from approximately 5.5 seconds to 14.5 seconds (an improvement of 9.0 seconds) with approximately 20 hours of learning. Our results show improvement in the average game duration from 12.8 seconds to 24.8 seconds (an improvement of 12.0 seconds). Note this value is somewhat smaller than the estimated fitness determined by the EA – the accurate fitness estimate uses many more samples, with the EA including only those candidates that survive into the next generation (intuition suggests that surviving candidates will have been luckier than rejected candidates). The best member in the final population of the EA was found to have an accurately estimated game duration of 32.9 seconds. We expect that the differences in the static fixed strategies (always holding the ball and random) between the two works are due to differences between the underlying basic skills of the agents (`HoldBall()`, `GoToBall()`, and `GetOpen()`).

Figure 8 compares the strategies evolved by the players *Stuart*, *Bowser*, and *Vince*. It shows the frequency of hold durations between passes or turnovers – shorter hold durations mean that the player passes more often. We observe that *Bowser* and *Vince* are similar in that they both pass infrequently (approximately 4% of the time). However, investigation of these candidates shows that the EA has evolved a similar strategy using different representations. *Bowser* only passes when desperate; consequently, all his passes are clearances to his farther teammate. *Vince* however, can pass when not desperate, distributing his passes equally between his two teammates. *Stuart* however passes much more often, passing to both players. He uses strategic passes to centralise the ball when not under pressure.

Table 1: Comparison Of Our Results To Stone's.

Strategy	Stone et al.	This work
Always Hold	4.7	7.2
Random	4.9	13.5
Learning (Initial Average)	5.5	12.8
Learning (Final Average)	14.5	24.8

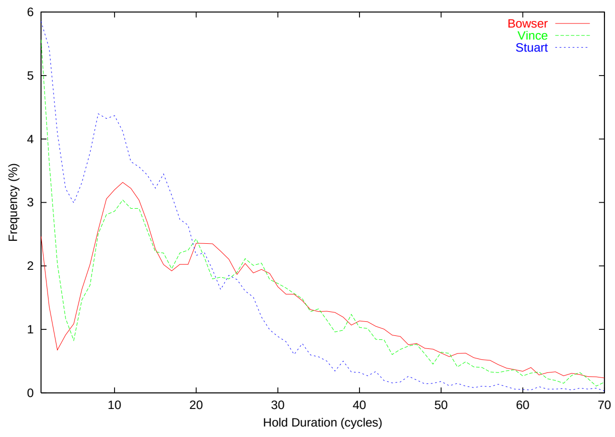

Figure 8: Frequency Of Hold Durations For *Stuart*, *Bowser*, and *Vince*.

5 CONCLUSIONS

We constructed an evolutionary algorithm to learn policies for playing the game of 3 vs. 2 keepaway, a sub-problem of the *RoboCup* soccer simulation league. We investigated the effects of the noisy fitness evaluation on the results from the system, and we used several techniques to alleviate this effect. In particular, we implemented a technique based on maintaining moving averages of fitness samples to enable us to use a good-sized population with several fitness samples/evaluation, yet still with a reasonable run-time.

Our results exceeded those from previous studies of this problem. Although the results are based on different keeper implementations, the best previous learning system improved 9 seconds in its game duration, whereas the population in our system improved by 12 seconds. Although we ran our system for longer than in previous studies, in fact it achieved its best results well before the end of most experiments, so the learning is not as slow as it may appear at first sight.

The idea of using moving averages to improve the learning rate (in the real-time sense) in the context of noisy fitness evaluations may find uses in other applications. We expect that this technique can be used wherever individuals can persist in the population.

6 FUTURE WORK

We plan to extend this work in several directions, both within the *RoboCup* domain, and otherwise.

Within the domain of keepaway, we plan to improve the framework that agents use for representing decisions. We also aim to make the problem more complex (possibly by introducing a second objective), with the aim of encouraging specialisation in the separate player populations. We might also experiment with different population structures, e.g. using one population of players, or using a population of teams. We may also translate the work to other *RoboCup* subproblems, such as developing specific skills that exploit the physics of *RoboCup*.

We plan to further investigate the effects of noise, both in this domain and more generally. The use of moving averages has helped to improve the results, and we will apply the same technique in other domains to refine it further. Connected to this, we will also investigate other possible ways to make fitness evaluations based on multiple (noisy) samples, e.g. using the median of a group of samples, as opposed to the mean.

References

D. Andre and A. Teller. Evolving Team Darwin United. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer-Verlag, Berlin, 1999.

T. Belch. *TeamBots* software and documentation. URL `http://www.teambots.org/`.

H.-G. Beyer. Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2–4):239–267, 2000.

M. Chen, E. Foroughi, F. Heintz, Z. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wang, and X. Yin. *RoboCup Soccer Server (Users Manual)*, June 2001.

P. J. Darwen. Computationally intensive and noisy tasks: Co-evolutionary learning and temporal difference learning on Backgammon. In *Proc. 2000 Congress on Evolutionary Computation*, pages 872–879, Piscataway, NJ, 2000. IEEE Service Center.

C. Darwin. *The Origin of Species*. Penguin Classics, London, 1859.

S. M. Gustafson and W. H. Hsu. Layered learning in genetic programming for a cooperative robot soccer problem. In *European Conference on Genetic Programming*, pages 291–301, 2001.

H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada. The robocup synthetic agent challenge. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.

J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler. Co-evolving soccer softbot team coordination with genetic programming. In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*. Springer-Verlag, Berlin, 1998.

L. P. Reis and N. Lau. FC Portugal team description: RoboCup 2000 simulation league champion. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*. Springer-Verlag, Berlin, 2001.

P. Stone and D. McAllester. An architecture for action selection in robotic soccer, 2001.

P. Stone, P. Riley, and M. Veloso. The CMUnited-99 champion simulator team. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer-Verlag, Berlin, 2000.

P. Stone, R. S. Sutton, and S. Singh. Reinforcement learning for 3 vs. 2 Keepaway. In P. Stone, T. Balch, and G. Kraetszchmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, pages 249–258. Springer-Verlag, Berlin, 2001.

M. Veloso, P. Stone, and M. Bowling. Anticipation: a key for collaboration in a team of agents: A case study in robotic soccer. *Proc. SPIE Sensor Fusion and Decentralized Control in Robotic Systems II*, 3839, September 1998.