Proceedings of the 2001 IEEE Congress on Evolutionary Computation Seoul, Korea • May 27-30, 2001

An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics

Graham Kendall
University of Nottingham
School of Computer Science & IT
Jubilee Campus
Nottingham NG8 1BB, UK
gxk@cs.nott.ac.uk

Glenn Whitwell
University of Nottingham
School of Computer Science & IT
Jubilee Campus
Nottingham NG8 1BB, UK
gxw@cs.nott.ac.uk

Abstract- Using the game of chess, this paper proposes an approach for the tuning of evaluation function parameters based on evolutionary algorithms. We introduce an iterative method for population member selection and show how the resulting win, loss, or draw information from competition can be used in conjunction with the statistical analysis of the population to develop evaluation function parameter values.

A population of evaluation function candidates are randomly generated and exposed to the proposed learning techniques. An analysis to the success of learning is given and the undeveloped and developed players are examined through competition against a commercial chess program.

1 Introduction

Since the beginning of the century mathematicians have tried to model classical games to create expert artificial players. Chess has been one of the most favoured of these with many decades of research focusing on the creation of grandmaster standard computer programs (see Fig. 1). This research culminated in the defeat of Garry Kasparov, the World Chess Champion, by IBM's purpose-built chess computer, "Deep Blue", in 1997. "Deep Blue" and since "Deeper Blue", still mainly rely on brute force methods to gain an advantage over the opponent by examining further into the game tree. However, as it is infeasible to evaluate all of the potential game paths of chess, we cannot rely on brute force methods alone. We need to develop better ways of approximating the outcome of games with evaluation functions. The automated learning of evaluation functions is a promising research area if we are to produce stronger artificial players.

In 1949, Shannon started to surmise how computers could play chess. He proposed the idea that computers would need an evaluation function to successfully compete with human players [13]. Although Shannon pioneered the work on computer chess, it is Turing who is accredited with producing the first chess automaton in 1952 [4]. The earliest publication that actively employs learning was presented in 1959 by Samuel [11]. Samuel developed a checkers program that tried to find "the highest point in multidimensional scoring space" by using two players. The results from Samuel's experiment were impressive and yet his ideas remained somewhat undeveloped for many years. In 1988, Sutton developed Samuel's ideas further and formulated methods for 'temporal difference learning' (TDL) [14]. Many researchers have since applied TDL to games [2,3,15,16]. One of the most successful of these is Tesauro's backgammon program "TD-Gammon" which achieved master-level status [15]. Tesauro had shown that TDL was a powerful tool in the development of high performance games programming. Thrun successfully applied TDL to his program "NeuroChess" [16]. TDL is reviewed in Kaebling [6].

Co-evolutionary methods try to evolve a population of good candidate solutions from a potentially poor initial population by "embedding the learner in a learning environment which responds to its own improvements in a never ending spiral" (Pollack) [9]. Moriarty gives a survey into general learning with evolutionary algorithms [8]. In 2000, Chellapilla and Fogel successfully developed strategies for playing checkers with the use of a population of neural network candidate players [5]. Barone applied adaptive learning to produce a good poker player. It was entered in a worldwide tournament involving several human expert players and achieved a ranking of top 22% [1]. Other examples of co-evolution include Pollack's Backgammon player [9] and Seo's development of strategies for the iterated prisoner dilemma [12].

2 The Chess Engine

A simple chess program was created as a platform for the learning process. The current board state was represented by a vector of length 64, where each element stores the contents of a square on the chess board. On each player's turn, the set of legal moves is generated for the current position. To select one of the moves, a minimax tree was generated with alpha-beta pruning to a fixed depth [7]. Quiescence was used to selectively extend the search tree to avoid noisy board positions where material exchanges

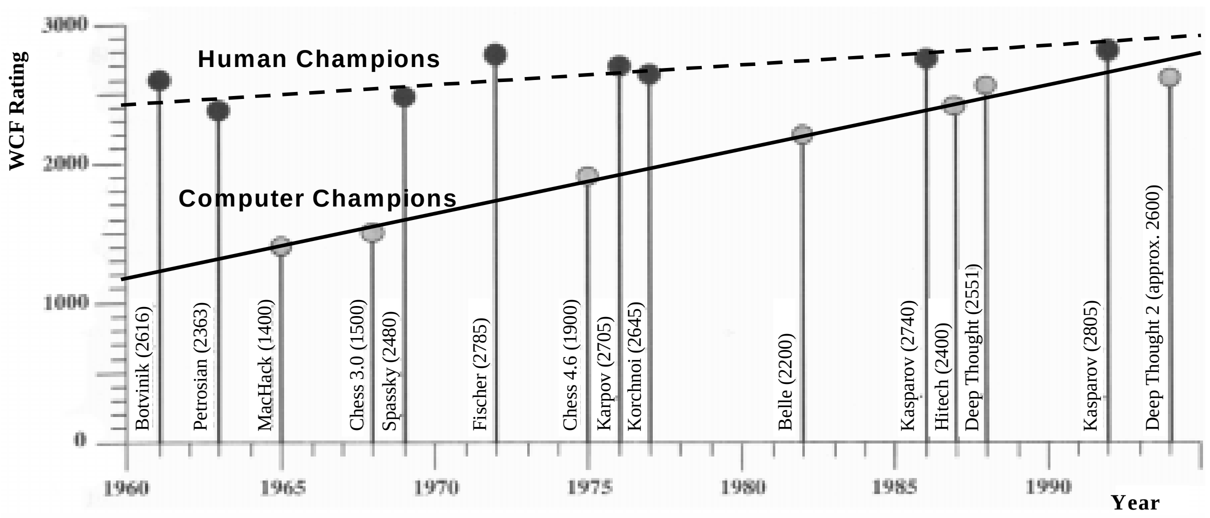

Figure 1. Comparison of the WCF Ratings of Computer Chess Champions and Human Champions from 1960-1995. The large quantity of computer chess research has resulted in computer players improving at a greater rate than their human counterparts. Diagram amended from "Artificial Intelligence – A Modern Approach", with kind permission of the authors [10].

may influence the resultant evaluation quality [4]. We chose to use a simplified version of Shannon's evaluation function involving player material and mobility parameters:

$$Evaluation = \sum_{y=0}^{6} W[y](N[y]_{white} - N[y]_{black})$$

where:

$N[6]$ = { N° pawns, N° knights, N° bishops, N° rooks, N° queens, N° kings, N° legal moves }

$W[6]$ = { weight_{pawn}, weight_{knight}, weight_{bishop}, $\text{weight}_{rook,}$ weight_{queen}, weight_{king}, $\text{weight}_{legal\ move}$ }

Also, a small fixed bonus was given for control of the center squares and advanced pawns with promotion potential. The first move of the minimax sequence giving the highest evaluation is conducted and the process repeats for the other player. Each chess game ends when one of four ending conditions is met: checkmate, stalemate, three-move repetition draw, or fifty move draw.

3 Learning Process Method

For a chess programmer, the main problem arises in choosing the respective weightings of the parameters in the evaluation function. If the evaluation function parameter weightings are less than optimal, the weakness will be emphasized throughout the entire game. A slight change in evaluation parameter weightings can be enough to completely change the entire game path and hence create a different player with a different playing strategy. Instead of the programmer having the responsibility for choosing the weightings, it can be beneficial to let the computer develop the weightings through learning techniques, as the optimal function is often very different from the assumed one.

Terminology

We shall use the following notation throughout the remainder of this paper:

P	:	Population
μ	:	Size of population
ν	:	A member of the population
ν_i	:	The i^th member of the population
κ	:	The fittest member of the population
f_{best}	:	The optimal evaluation function
y	:	A parameter of ν
V	:	A vector
c	:	A variable

We adopted a population-based approach to learning by randomly generating a population, P, of μ members where each member ν_i represents one candidate evaluation function. Thus the population can be summarised by:

$$\forall \nu_i \in P, \quad \nu_i = \{ A, B, C, D, E \}$$

where:

A = weight_{knight}
B = weight_{bishop}
C = weight_{rook}
D = weight_{queen}
E = $\text{weight}_{legal\ move}$

In the evaluation function proposed in section 2, weight_{king} and weight_{pawn} are also included. These parameters are fixed for every member of the population. The weight_{king} parameter is set to an arbitrarily high value (i.e. ∞) as the loss of the king signifies loss of the game. The weight_{pawn} parameter is set to a constant of 1 as to configure each of the other weightings in pawn units.

In our learning model we identify the relative strength between pairs of population candidates through competition. Firstly select two members of the population. Then conduct two games of chess with these players, each player playing one of the games as White and one of the games as Black. It is necessary to follow in this manner to eliminate the first move advantage. Each player will use their own evaluation function when selecting moves. After completion, the winning function is allowed to remain in the population. The losing function is expelled from the population and a mutated clone of the winner is inserted in its place. If the contest is a draw then both functions remain in the population but they are mutated. The process continues until population convergence is achieved.

3.1 Selection

We propose an iterative method for selection whereby the population is represented by a vector, V, of length μ. A variable, c, is used to monitor progress through the vector and is initially set to the first member of V. When selecting the two evaluation functions, we choose the c^{th} member of the vector, V[c], and randomly select another function from the remainder of the vector, which is V[i] (where $c < i < \mu$), as the other player. The variable c is incremented before making the next selection (see Fig. 2). When the variable c indexes the last member of the population we call this one generation of learning, the vector is then reversed. This selection method results in the quick propagation of the best function. For example, let's assume a simple learning process that simply clones the winner of the match into the loser's vector position. It can be shown that κ will occupy the last vector position at the end of each generation:

Proof

i) κ is the last member of the vector: condition holds.

ii) κ is the first member of the vector:
κ and *n* are selected where *n* > first index.
κ wins the competition (it is the best function).
κ is cloned and inserted into position *n*.
If position *n* is last element of V: condition holds.
Remove elements up to *n*. Let vector V = [*n* … μ]
Else repeat case **ii)**.

iii) κ is the i^{th} member of the vector:
Remove elements up to *i*. Let vector V = [*i* … μ]
Now follow case **ii)**.

(see Fig. 3)

By reversing the vector, we ensure that the best function will occupy the first vector position at the beginning of a generation. Therefore, the best member of the population will have more chance of propagating through the population.

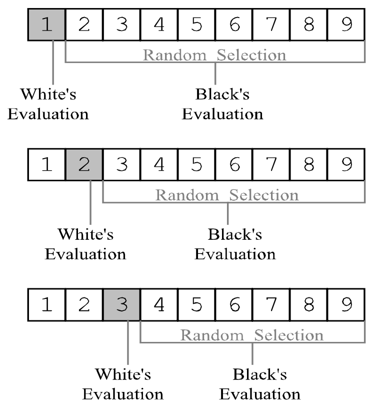

Figure 2. The first three selection iterations of a population consisting of μ = 9 members. The shaded square indicates the position identified by the variable c.

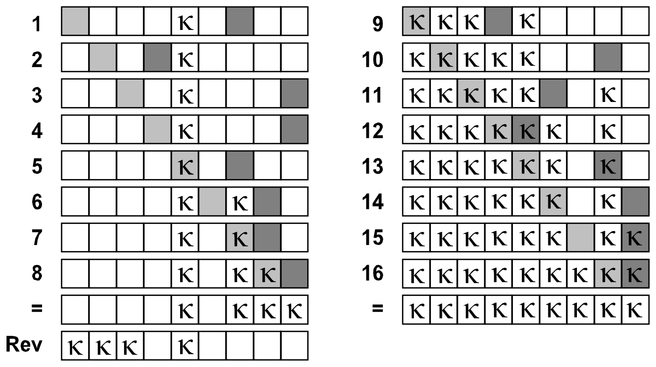

Figure 3. The diagram shows the propagation of the best function, κ, in a population of μ = 9 members (the grey squares indicate the selected players v_c and v_i for each iteration). Starting from position 5 in the vector, the best function is involved in three competitions during the first generation (iterations 5, 7, and 8). This results in several clones of κ towards the end of the vector. The vector is reversed before starting the second generation to give κ a greater opportunity to propagate. In the simulation, the vector is completely saturated with κ after two generations.

3.2 Crossover

In the proposed learning model there is no crossover – reproduction is completely asexual. After the selection and competition of two population members, there will be at most one breeding parent – the fittest one of the pair. The population member that wins the competition becomes the breeding parent and creates a clone to replace its rival. In the case of a drawn competition, both candidate parents are denied from breeding.

3.3 Mutation

So far, the learning mechanism is able to distinguish the best member of the population. But what if the best member of the population is still much worse than the optimal solution? (see Fig. 4).

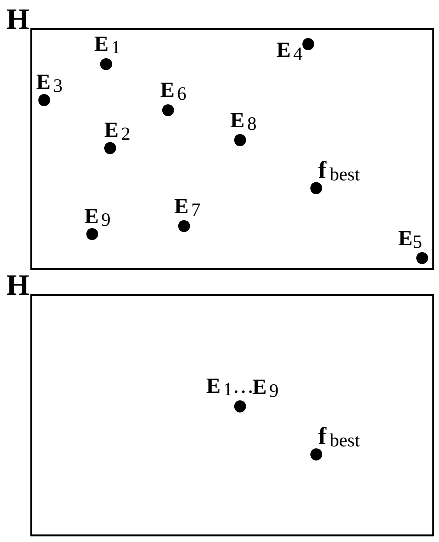

Figure 4. Assume that each possible evaluation function maps to a unique point on H-space and there is one unknown optimal solution, f_{best}. We generate a population of $\mu = 9$ members as shown in the top H-space diagram. However, the best member of the population, E_8, is still a long way from the optimal solution. After learning, the population would stabilise as shown in the bottom H-space diagram.

A good search method is a tradeoff between exploitation of search space and exploration of different areas within search space. The current system is completely exploitative and never veers outside of the initial search specifications. A suitable result will only ever be found if a member of the initial population is close to the optimal solution.

We propose a mutation system that is dependent on the dispersement of the population to create a more explorative search. This involves calculating the standard deviation for each of the evaluation parameters, y, that we are trying to develop:

$$\sigma_{(y)} = \sqrt{\frac{\sum_{n=1}^{\mu} (V_{n(y)} - average_{(y)})^2}{\mu - 1}} \qquad \forall y \in \nu$$

After each competition between pairs of candidate evaluation functions we mutate each of their parameters by a proportion of the standard deviations for those parameters depending on the outcome of competition. Therefore, when the population is diverse at the start of the learning process, we mutate by a larger amount than when the population begins to converge towards the end of learning. The functionality behind this method is that the user does not need to specify a mutation scheme as all mutation is contolled by the population's diversity. A generalised mutation formula is specified below:

$$\nu_{(y)} = \nu_{(y)} + \left((RND(0...1) - 0.5) \times R \times \sigma_{(y)}\right) \qquad \forall y \in \nu$$

The formula allows for positive and negative mutations with the value of *R* chosen based on the outcome of matches. Each parameter is completely independent from the other parameters when calculating the standard deviation. For example, one parameter, y_1, may have a low standard deviation but another, y_2, may be large. Therefore in general, parameter y_1 would be mutated less than y_2.

Depending on the results from contests, we can mutate by different amounts. At the end of each match, the following actions were chosen (with R values selected based on initial testing):

A function wins both games:

Expel loser, duplicate winner and mutate with R = 2

A win and a draw:

Expel loser, duplicate winner and mutate with R = 1
Mutate winner with R = 0.2

Drawn match:

Mutate both functions with R = 0.5

If the standard deviation of a parameter is low then all of the members of the population are converging for that parameter. But it may be that the parameter value is at a local maxima. Therefore, if a parameter, y, obtained a small standard deviation ($\sigma_{(y)} < 0.2$), then we randomly select members of the population and mutate their y parameter by a small amount.

3.4 Summary of Learning Process

The learning process can be summarised as follows:

```
1)   Generate Population of μ members in a vector
2)   Set iteration variable c = 0

While (Continue Learning)

    3)   Randomly set variable i such that c < i < μ
    4)   Select two referenced members, ν_c & ν_i
    5)   Game One: ν_c is White, ν_i is Black
    6)   Game Two: ν_i is White, ν_c is Black
    7)   If there's a winner, expel loser, duplicate winner
    8)   Mutate as specified
    9)   Increment c
    10)  If  c = μ          (completion of a generation)
              Set c = 0
              Reverse vector

End While
```

4 Experimental Results

Using the methods proposed in this paper, we conducted an experiment involving a population of $\mu = 50$ candidate evaluation functions. We assigned values for each of the candidate's parameters by randomly generating numbers in the range { 0…12 }. One member of the population was seeded with the parameter weightings as suggested by Shannon [13]:

$$A = weight_{knight} = 3$$
$$B = weight_{bishop} = 3$$
$$C = weight_{rook} = 5$$
$$D = weight_{queen} = 9$$
$$E = weight_{legal\ move} = 1^{\dagger}$$

It is worthwhile seeding the population because the seed can reduce the duration of learning. If the seed is logically close to the optimal solution then the duration of learning will be shorter. Even if the seed is the poorest member of the population then it will quickly be expelled from the population in favour of better candidate functions.
The averages for each of the parameters in this initial population gave us the evaluation function of the "undeveloped player". The values were as follows:

$AVR_{\mathbf{A}\ (knight)}$	=	6.24
$AVR_{\mathbf{B}\ (bishop)}$	=	5.80
$AVR_{\mathbf{C}\ (rook)}$	=	6.54
$AVR_{\mathbf{D}\ (queen)}$	=	5.44
$AVR_{\mathbf{E}\ (legal\ moves)}$	=	6.38

The standard deviation value for each of the parameters was approximately 3.8:

$\sigma_{\mathbf{A}\ (knight)}$	=	3.88
$\sigma_{\mathbf{B}\ (bishop)}$	=	3.60
$\sigma_{\mathbf{C}\ (rook)}$	=	3.90
$\sigma_{\mathbf{D}\ (queen)}$	=	3.75
$\sigma_{\mathbf{E}\ (legal\ moves)}$	=	3.98

The chess program was set to use a fixed look ahead of three moves. The population was allowed to undergo 45 generations of learning through 2200 iterations.

4.1 Observations

During the initial stages of learning, each game of chess took about three minutes to complete. Toward the latter stages of learning, the average game time had increased to six minutes and there were more occurrences of drawn games. This is a good indication that the initial population contained many poor candidate functions that were easily beaten. As the poor candidate functions were replaced with better ones, the games became tougher to win, hence, the game times increased and there were a greater number of drawn matches. It was also noted that the seeded function was the only original member of the population to remain unchanged through the first five generations. This indicates that the seed was the fittest member of the initial population.

4.2 Results

The averages and standard deviations for each of the parameters during the learning process are shown in Fig. 5 and Fig. 6. At the end of the experiment, the population had converged to the following average values which we will call the "developed player":

$AVR_{\mathbf{A}\ (knight)}$	=	3.22
$AVR_{\mathbf{B}\ (bishop)}$	=	3.44
$AVR_{\mathbf{C}\ (rook)}$	=	5.61
$AVR_{\mathbf{D}\ (queen)}$	=	8.91
$AVR_{\mathbf{E}\ (legal\ moves)}$	=	0.13

The standard deviations of the parameters had decreased to the following values:

$\sigma_{\mathbf{A}\ (knight)}$	=	0.28
$\sigma_{\mathbf{B}\ (bishop)}$	=	0.58
$\sigma_{\mathbf{C}\ (rook)}$	=	0.33
$\sigma_{\mathbf{D}\ (queen)}$	=	0.47
$\sigma_{\mathbf{E}\ (legal\ moves)}$	=	0.13

The reduction in the standard deviations (from around 3.8 before learning to 0.3 after learning) shows that the population had stabilised with most of the population's members containing similar parameter weightings.

A second experiment was conducted with a completely random initial population (without any seeding). The developed values from this experiment were consistent with the first experiment to one decimal place although an extended learning time was required.

4.3 Success

After several test games with human players, it was clear that the chess program's performance after learning was better than before learning. However, due to the fallible nature of humans and the lack of accessible grandmaster standard players, we decided to use a commercial chess program, "Chessmaster 2100" from "The Software Toolworks", as a control to measure the extent of the evaluation function's improvement. We set Chessmaster's look ahead to a fixed depth of three moves and disallowed the use of its opening sequence database. This eliminated any advantages that Chessmaster may have had over the chess program we had produced. Therefore, the player differences were only with their respective evaluation functions. Both the undeveloped and developed players were allowed two games with Chessmaster, once playing as White and once as Black. The results are shown in Table 1.

† Although Shannon's suggested value for mobility was 0.1, a value of 1 was selected as to show that the proposed learning method is suitably explorative.

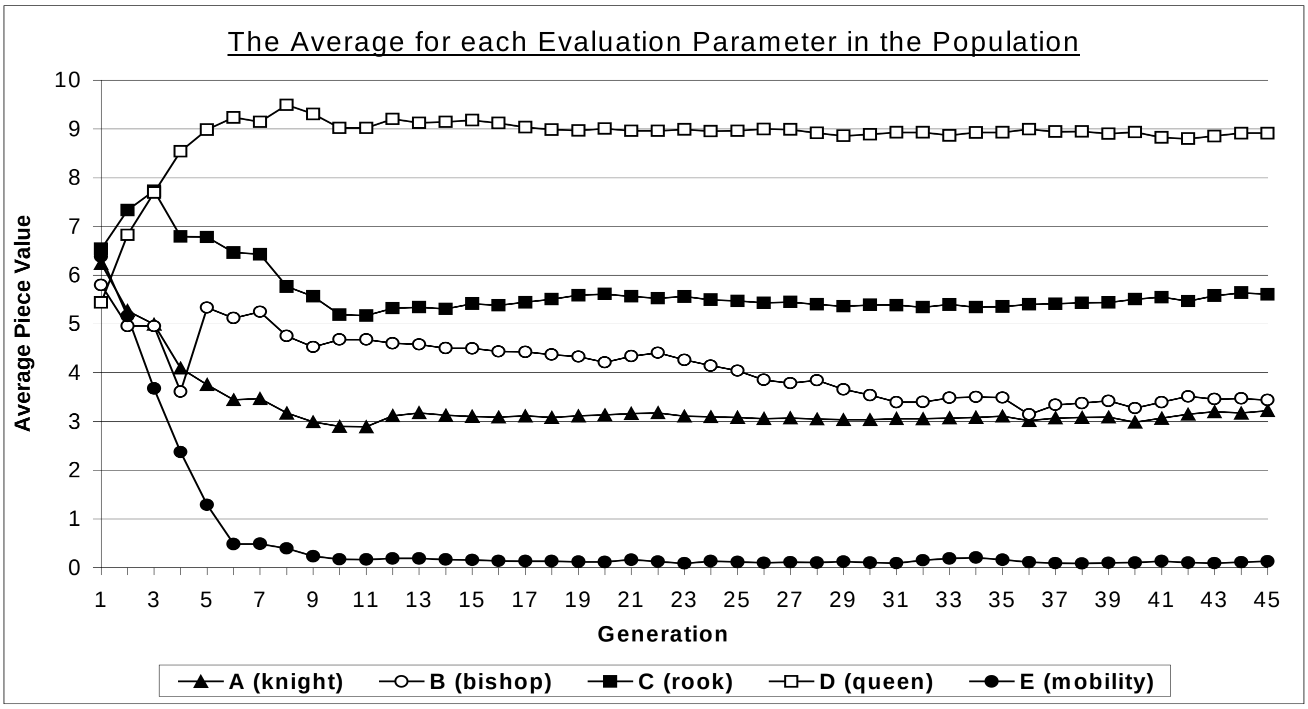

Figure 5. A chart showing the averages of the evaluation function parameters within the population during learning. We can see how the values have changed rapidly over the first 10 generations as the poorer functions of the population are beaten by better ones. As we continue through the iterations, eventually we reach a point where the values change less rapidly. For example, take the change in values from the 1st generation to the 2nd generation. Each of the coefficients have changed by approximately 1. Now take the 44th and 45th generations, the change is about 0.04 for each coefficient between the generations. The parameter averages have stabilised indicating that the learning process will not yield any further improvement.

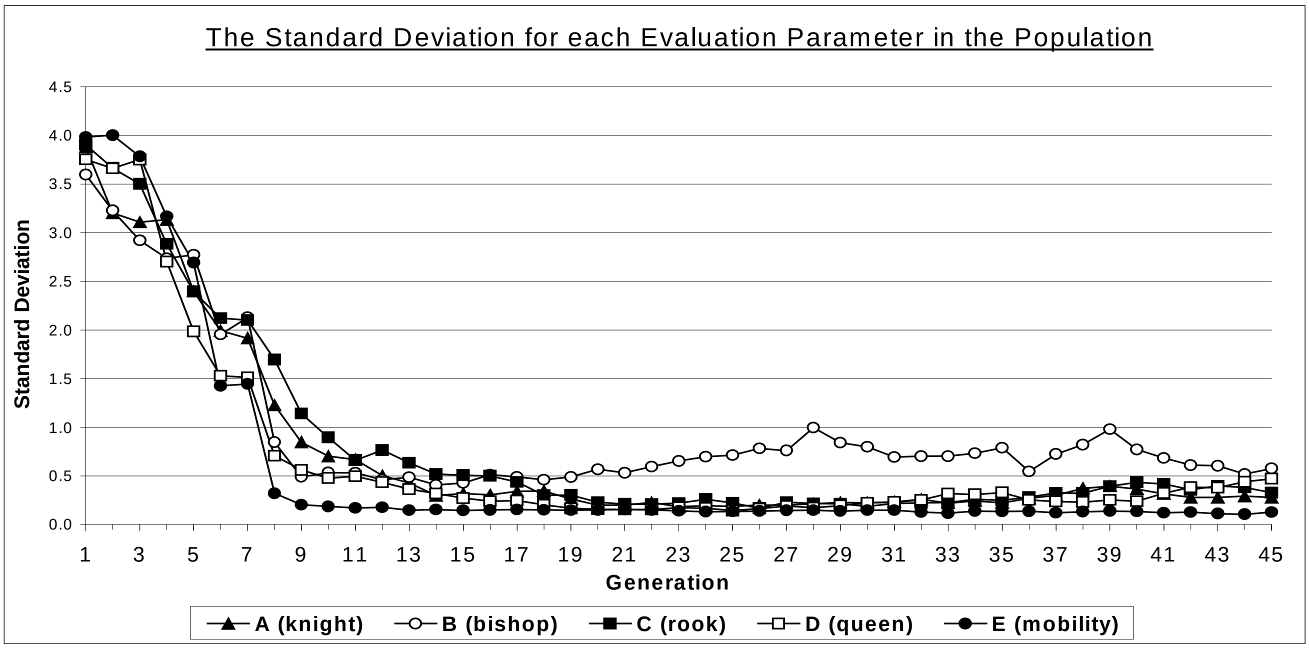

Figure 6. A chart showing the standard deviations of the evaluation function parameters during the learning process. We can see that the standard deviation of each parameter is decreasing as learning proceeds. The standard deviations are large at the beginning of learning as the initial population members' had a diverse range of values for each parameter. Therefore, mutations are greater in the earlier stages of the process. As learning continuess and better functions win, the standard deviations decrease and mutation is less.

It is important to note that the chess games are completely deterministic, there is no randomness for move selection within any program. If the players were to play again under the same conditions, the game paths would be identical.

Player	Playing as	Player's Result
Undeveloped	White	Lost in 58 moves
	Black	Lost in 59 moves
Developed	White	**Won in 69 moves**
	Black	Lost in 97 moves

Table 1. Results of test games playing against Chessmaster 2100.

Chessmaster 2100 was able to defeat the undeveloped player within 60 moves whilst playing as both White and Black. However, the developed player was a more able opponent losing to Chessmaster in 97 moves as Black. This is an improvement on the 59-move loss that the undeveloped player suffered. When playing as White, the developed player scored a victory in 69 moves. The developed player was able to beat Chessmaster in fewer moves than it took Chessmaster to win against the developed player (the developed player games are given in the appendix).

Chessmaster 8000, which provides several opponent profiles of differing quality and their USCF ratings, was used to evaluate the improvement of the developed player over the undeveloped player. This was achieved by choosing one of Chessmaster 8000's opponent profiles and playing it against one of our players. If our player won, a better-rated opponent profile was used. If the player lost, a worse opponent was selected. This continued until our players were evenly matched with their respective opponents (i.e. drawing the majority of games). The resulting ratings of the undeveloped and developed players are highlighted in Table 2.

USCF Rating	Ability	
2400+	Senior Master	
2200-2399	Master	
2000-2199	Expert	
1800-1999	Class A	
1600-1799	**Class B**	**Developed (1750)**
1400-1599	Class C	
1200-1399	Class D	
1000-1199	Class E	
800-999	Class F	
600-799	**Class G**	**Undeveloped (650)**
400-599	Class H	
200-399	Class I	
< 200	Class J	

Table 2. The Undeveloped and Developed Player USCF Ratings.

Although these ratings are unofficial, the difference is clearly evident with the developed player achieving a 270% improvement over the undeveloped player's rating.

5 Conclusions

In this paper, we have proposed a method for selection of candidate evaluation functions and also a learning mechanism that utilises the dynamics of a population in order to control mutation amounts. Mutating by a random proportion of the standard deviation of parameters is beneficial, as the population controls mutation not the user.

An experiment was conducted and the resulting evaluation function was examined through competition with Chessmaster. After learning, the performance of the population was greatly improved over the initial population and it is interesting that the values developed, both with and without seeding, are of similar nature to Shannon's mathematically derived suggestions of 1950 [13]. Although Shannon believed the bishops and knights to be of equal worth, the more recent idea of a bishop being slightly more valuable than a knight is reflected in the developed values. The resultant program played a better game of chess against human players and, in particular, the commercial software.

Unfortunately, many games are needed to give good solutions, so although increases in the game tree search depth gives more accurate results, it has the side effect of exponential increases in game lengths and, therefore, learning times. Also, the developed function will only have good weights for the particular implementation. That is, if the search depth is changed, parameters are added or removed from the evaluation function, or if the chess implementation changes, then the previously developed function may no longer play as well. In experiments using smaller populations, the learning time increased. This can be attributed to the fact that the weights take longer to converge due to the chess players having a smaller number of opponents to play against. Of course, a larger population means a longer execution time. The balance between population size, learning time and execution time is something we are considering.

Chess games are very complex, requiring different strategies in different situations such as the protection of an important pawn, defence of a file, or perhaps dominance of a board region. As our evaluation function was based on Shannon's simple linear function, its performance is limited. However, for research purposes, the advantage of using a documented evaluation function is that, after learning, immediate comparisons are possible. The developed player has been shown to have vastly improved over the undeveloped player using Chessmaster 8000. Therefore, we see no reason why the given processes could not be applied to developing stronger players by using more complex evaluation functions incorporating intragame adaptable parameters, a greater look ahead depth, and using longer learning times.

The ability for non-experts to produce competent programs that exceed the expertise of the creator, with the use of evolutionary learning methods, is a very powerful approach that we would encourage people to use when domain specific knowledge is lacking.

References

[1] Barone L., While L., “Adaptive Learning for Poker”, Proceedings of the Genetic and Evolutionary Computation Conference, July (2000), pp. 566-573.

[2] Baxter J., Tridgell A., Weaver L., “KnightCap: A chess program that learns by combining TD(lambda) with game-tree search”, Proceedings of the Fifteenth International Conference on Machine Learning,, July (1998), pp. 28-36.

[3] Beal D. F., Smith M. C., “Learning Piece values using Temporal Difference”, Journal of the International Chess Association, September (1997).

[4] Bowden B. V., “Faster Than Thought”, Chapter 25, Pitman, (1953).

[5] Chellapilla K., Fogel D. B., “Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program against Commercially Available Software”, Proceedings of the Congress on Evolutionary Computation 2000, July (2000), Vol. 2, pp. 857-863.

[6] Kaebling L. P., Littman M. L., Moore A. W., “Reinforcement Learning: A Survey”, Journal of Artificial Intelligence Research, (4) (1996), pp. 237-285.

[7] Knuth D. E., Moore R. W., “An analysis of alpha beta pruning”, Artificial Intelligence, Vol. 6 (4) (1975), pp. 293-326.

[8] Moriarty D. E., Schultz A. C., Grefenstette J. J., “Evolutionary Algorithms for Reinforcement Learning”, Journal of Artificial Intelligence Research, (11) (1999), pp. 241-276.

[9] Pollack J. B., Blair A. D., Land M., “Coevolution of a Backgammon Player”, Proceedings of the Fifth Artificial Life Conference, May (1996).

[10] Russell S., Norvig P., “Artificial Intelligence A Modern Approach”, Prentice-Hall, (1995), pp. 137.

[11] Samuel A. L., “Some Studies in Machine Learning Using the Game of Checkers”, IBM J. Res. Dev., Vol. 3 (3) (1959).

[12] Seo Y. G., Cho S. B., Yao X., “Exploiting Coalition in Co-Evolutionary Learning”, Proceedings of the Congress on Evolutionary Computation 2000, July (2000), Vol. 2, pp. 1268-1275.

[13] Shannon C., “Programming a computer for playing chess”, Philosophical Magazine, Vol. 41 (4) (1950), pp. 256.

[14] Sutton R. S., “Learning to predict by the methods of temporal differences”, Machine Learning (3)(1988), pp. 9-44

[15] Tesauro G. J., “TD-Gammon, a self-teaching backgammon program, achieves master-level play”, Neural Computation, (6) (1994), pp. 215-219.

[16] Thrun S., “Learning to Play the Game of Chess”, In Tesauro G., Touretzky D., Leen T., editors, “Advances in Neural Information Processing Systems 7”, San Fransisco, Morgen Kaufmann, (1995).

Appendix

Chessmaster 2100 (White) vs. Developed Player (Black)

1	g1f3	d7d5
2	b1c3	d8d6
3	e2e4	d5e4
4	c3e4	d6e6
5	f3g5	e6d5
6	d2d4	b8c6
7	c1e3	e7e6
8	f1e2	f8b4†
9	c2c3	h7h6
10	e2f3	d5b5
11	g5f7	e8f7
12	a2a4	b5c4
13	c3b4	c6b4
14	b2b3	b4d3†
15	e1d2	c4a6
16	d2c3	a6a5†
17	c3d3	a5f5
18	g2g4	f5a5
19	d3c2	g8f6
20	e4f6	f7f6
21	d1e1	a5b6
22	g4g5	h6g5
23	a1d1	g5g4
24	a4a5	b6d6
25	f3g4	c8d7
26	g4f3	d7c6
27	e1e2	h8h4
28	a5a6	b7b5
29	c2b1	b5b4
30	f3c6	d6c6
31	d1c1	c6d5
32	e2d1	d5f5†
33	d1c2	h4e4
34	d2d3	a8d8
35	c1c7	f5g4
36	c7a7	a5f5
37	h1g1	e4g4
38	d3f5†	e6f5
39	g1g4	f5g4
40	a7c7	d8d5
41	a6a7	d5a5
42	e3d2	g4g3
43	h2g3	g7g6
44	d2b4	a5a6
45	c7c6	a6c6
46	a7a8(Q)	c6e6
47	a8f8†	f6g5
48	f8f4†	g5h5
49	f4h4‡	

Developed Player (White) vs. Chessmaster 2100 (Black)

1	d2d4	g8f6
2	c1f4	b8c6
3	b1c3	f6h5
4	e2e3	h5f4
5	e3f4	e7e6
6	d4d5	e6d5
7	d1d5	f8e7
8	f1c4	e8g8
9	a1d1	d7d6
10	d5e4	e7f6
11	g1e2	f6c3†
12	b2c3	f8e8
13	e4d5	c8e6
14	d5e4	e6c4
15	e4c4	c6a5
16	c4d5	a5c6
17	d1d3	d8e7
18	d3e3	e7h4
19	g2g3	h4g4
20	e1d2	a8d8
21	h1b1	g4c8
22	c3c4	e8e6
23	e2c3	e6h6
24	h2h4	a7a6
25	b1e1	c6b4
26	d5b7	a6a5
27	b7c8	g7g6
28	c8d8†	g8g7
29	d8c7	h6h5
30	c7d6	h5f5
31	e3e8	f5h5
32	c3e4	h5d5
33	c4d5	h7h5
34	d6f8†	g7h7
35	f8h8‡	