Proceedings of the 2007 IEEE Symposium on
Computational Intelligence and Games (CIG 2007)

A Comparison of Genetic Programming and Look-up Table Learning for the Game of Spoof

Mark Wittkamp, Luigi Barone, and Lyndon While
School of Computer Science & Software Engineering
The University of Western Australia
{wittkamp,luigi,lyndon}@csse.uwa.edu.au

Abstract— **Many games require opponent modeling for optimal performance. The implicit learning and adaptive nature of evolutionary computation techniques offer a natural way to develop and explore models of an opponent's strategy without significant overhead. In this paper, we compare two learning techniques for strategy development in the game of Spoof, a simple guessing game of imperfect information. We compare a genetic programming approach with a look-up table based approach, contrasting the performance of each in different scenarios of the game. Results show both approaches have their advantages, but that the genetic programming approach achieves better performance in scenarios with little public information. We also trial both approaches against opponents who vary their strategy; results showing that the genetic programming approach is better able to respond to strategy changes than the look-up table based approach.**

Keywords: Imperfect Information Games, Spoof, Opponent Modeling, Genetic Programming, Look-up Table

I. Introduction

In games of imperfect information, players do not have complete knowledge about the state of the game and must make value decisions about their relative strength using only the public information available to them. Due to their non-deterministic nature, the task of programming satisfactory artificial opponents for these types of games is extremely difficult. The large branching factors result in significant combinatoric explosion in their game trees, rendering standard search techniques (e.g. minimax) less useful.

Spoof is a simple game requiring players to guess an unknown number using only partial knowledge about the number and publicly announced guesses by other players. Like the games of Roshambo and Iterated Prisoner's Dilemma (IPD), opponent modeling (construction of a model of an opponent's playing style, typically in order to exploit inherent weaknesses in their play) in the game of Spoof is crucial. Given a model of an opponent's strategy, the model can be analysed to discover weaknesses and predictabilities in the opponent's strategy and a counter-strategy determined.

Evolutionary computation is the term used to describe the different computational techniques that employ the principle of neo-Darwinian natural selection as an optimisation tool to solve problems with computers. The inherent learning capabilities of natural selection, capable of learning and adapting in dynamic and noisy environments, make evolutionary approaches well-suited to the application of strategy development for play against differing and potentially adapting opponents in games. Indeed, the application of evolutionary computation techniques for opponent modeling in games of imperfect information has led to some notable successes [1], [2], [3], including the game of IPD [4], [5].

Genetic programming is one such form of evolutionary computation. Introduced by Koza [6], this paradigm defines genetic operators (crossover, mutation, and fitness proportionate selection) directly over tree-like computer programs, thus offering practitioners the opportunity to evolve complex programs without having to define the structure or size of the genetic material in advance. Nodes in the tree represent functions in the evolving computer program, and terminals (leaves of the tree) represent either variables, constants, or zero argument functions with side-effects. Genetic programming has been used for a myriad of problems [7], [8], [9], including strategy development in games [10], [11].

In our previous work [12], we proposed the use of genetic programming to create computer Spoof players capable of learning and exploiting weaknesses in different opponent playing styles in order to develop successful strategies for play. In this paper, we compare the performance of our genetic programming approach to an approach based on look-up tables — learning via observation of the correct action for the different possible game states a player may face. In doing so, we also present extensions to our original genetic programming approach, allowing strategies to be developed for all guessing positions. Additionally, we examine the choice of fitness function used during evolution, comparing a fitness function based on win ratios versus one based on how well a learned model that of the opponent.

The rest of the paper is structured as follows. Section II introduces the game of Spoof in more detail, explaining the mechanics of how the game is played. In Section III, we introduce our adaptive approaches for building computer Spoof players through genetic programming and look-up table learning. Section IV compares the performance of these learning techniques, contrasting the advantages and disadvantages of both. Section IV also examines choices regarding the number of fitness samples used in the evaluation of individual strategies and different forms the fitness function may take. Section V concludes the work.

II. The Game of Spoof

Spoof is a game of imperfect information played by two or more players. The game begins by each player selecting

a number of tokens (typically coins) from 0 to 3 (called the player's *selection*), which remain hidden from all other players. In turn, each player attempts to guess the total number of coins held by all players (called the player's *guess*) with the constraint that no player may repeat a previous player's guess. The winner of the game is the player who correctly guesses the total number of coins. If no player guesses the correct total, the game is deemed a draw and is typically repeated (with the guessing order altered).

At first thought, the game may seem purely random and little can be done other than to guess the maximum of the probability distribution of possible totals. However, as players announce their guesses, they may well be providing information about the number of coins they have selected. For example, consider a two player game where the first player guesses a total of 5. Assuming rational play, this player must have selected either 2 or 3 coins, otherwise a total of 5 would be impossible. The second player can now use this information in making their guess, and should announce a total of 2 or 3 plus their own selection. Using this approach, the player improves their chances of immediately winning the game (without replay) from 25% (with no information about the first player's selection) to 50% (with knowledge that the first player's selection is one of two possibilities). Similar analysis is possible for other game states in Spoof [13], but the analysis becomes exceedingly more complex as the number of players increases.

Observe that the *position* in the game a player acts (announce a guess) induces a trade-off between what information is available and opportunity to guess a total. Being first to act means all possible totals are available to be guessed, but no information about the number of coins each player has selected is available. Being last to act provides maximal information about the selections of the other players (and, assuming rational play, may well mean the total can be determined with a high degree of certainty), but the correct total may well have already been announced by another player. A clear trade-off arises — acting first provides minimal information, but maximal opportunity; acting last provides maximal information, but minimal opportunity to guess the correct total.

Opponent modeling in the game of Spoof is crucial for optimal performance. For example, consider the problem of acting first in three player Spoof. A general strategy for acting in this position is to guess the number of coins one is holding plus 3 (as 3 is probabilistically the most likely outcome for the total of the remaining players' coins). However, this strategy is only sound if both opponents choose their hidden coins uniformly randomly. Consider instead, if both opponents tend never to hold 2 or 3 coins. The previous strategy now performs poorly, and a better opponent-specific strategy should be used instead (a better strategy will be to guess 1 or 2 more than the number of coins being held). Indeed, our experience has shown that human players often do not select their coins randomly (preferring certain coin choices or patterns over others), and more typically, provide information about their selection in the way they guess (it is especially the case that human players use the same guessing algorithm time and time again).

III. Building Adaptive Spoof Players

This section will provide a brief overview of how we develop artificial computer Spoof players using both learning methods — genetic programming and look-up tables.

As the focus of our investigation is on the examination of learning techniques, we do not attempt to build a complete Spoof playing strategy in this work. In particular, we ignore the problem of coin selection, instead forcing our computer players to select uniformly randomly from the range [0 .. 3]. This of course may be a poor choice (being able to alter the probability distribution of totals may well be advantageous), but since our test players will be non-responsive, allowing our computer players to select non-randomly would provide them with an unfair advantage. We instead force our computer players to select uniformly randomly, requiring them to use their intelligence to learn countering strategies in order to maximise performance instead of simply exploiting the non-intelligence of its opponents.

A. Genetic Programming

Genetic programming maintains a population of potential solutions which can be ranked in terms of their efficacy through an evaluation function which provides selection pressure. The population goes through a number of generations, whereby the population of program-trees evolve toward optimality as a result of selection pressure.

In our approach, guessing strategies take the role of individuals in our evolving population. Guessing strategies are represented in the form of a program-tree, consisting of both float and boolean types with a float at the root node. The evaluated program-trees are cast to an integer when the strategies are called upon to produce a guess. Invalid guesses (guesses having already been announced by a previous player) are automatically converted to the nearest available guess. This allows for less complex program-trees, as they need not be burdened with the additional task of ensuring unique guesses. Recall, coin selection is made uniformly randomly within the allowable range (0 to 3 inclusive).

We allow the genetic programming system the use of four numerical constants (`0`, `1`, `2`, and `3` to represent the four possible coin-held values), standard arithmetic operators (addition, subtraction, multiplication, and division), standard comparison operators (greater-than, less-than, and equal-to), and boolean operator nodes (negation, conjunction, and disjunction). Also, a conditional selection mechanism (the `if` function) is included to select between sub-programs of the genetic programming player (*GP Player*). Note that the `if` function expects three arguments, the first a boolean condition, and the second and third two sub-programs (the second parameter sub-program is evaluated if the first parameter evaluates to true, otherwise the third parameter sub-program in evaluated). To enable our evolving player to make an informed guess, we equip the genetic programming system

with a number of game-specific terminals that can be used in a candidate solution's program-tree. These are: the number of players in the game, the number of coins that the player has selected, and the announcements of each player prior to the GP Player. The depth of a program-tree is limited to 10.

To reduce the effects of noise (recall the information-opportunity trade-off inherent in the game), we use a fitness function based on the accuracy of the player's *desired* guess, not the actual number of wins scored by the strategy (all players must reveal their secret selections at the end of game in order to determine the winner). This means fitness is not a direct measure of success in the game, but instead measures how well the implicit model of the opponent (via a countering strategy) fits. In Section IV-E we experiment with a fitness function based on the actual number of games won instead of this pseudo-success measure. Should the genetic programming system need to choose between solutions with the same fitness, the solutions are ranked in order of program-tree size, with preference given to solutions with a smaller (parsimonious) program-tree size.

B. Look-up Table Learning

Look-up table learning is a form of reinforcement learning that can be used for learning state-action pairs from previous observations, effectively learning how to play a game by remembering the outcome of previous games. For Spoof, look-up table players (*LU players*) maintain a table of *weights*, one for every permissible guess that can be made for every possible game state (i.e. the player's own selection along with the announced guesses of all opponents that have already guessed). For example, when guessing third in three player Spoof, a look-up table player will maintain a table of 400 weightings — one for every possible game state: every possible total for the announcements made by the first and second players (0 through 9), every possible personal coin selection (0 through 3), and each possible guess that could be made (0 through 9). One obvious disadvantage of look-up table learning is scalability — the required table size grows rapidly with increasing game size, influenced both by the number of players and the guessing position of the player.

All weights in a LU player's table are initialised to zero and are incremented by one when that particular guess was found to be correct. When asked to make a guess, the LU player checks its weight table, selecting the response with the highest weight. If no such guess exists, a guess is chosen uniformly randomly from the set of all permissible guesses. Should a player's desired guess be unavailable, the nearest available guess will be chosen automatically. As with the GP player, only the accuracy of the guess is taken into account — the fitness of a player is entirely based on its desired guess regardless of whether or not that guess is unavailable due to it having already been guessed.

We have made the learning of the LU player as close as possible to that of the GP player in order to make comparisons fairer and their results more meaningful. Both the GP and LU players are able to learn only implicit opponent models of *all* opponents as a single entity, rather than developing individual opponent models for each opponent it plays against. Like the GP players, LU players make their selection uniformly randomly. As a baseline, the GP system is given 5000 generations to learn, each of which involves playing 1000 games for each of its 50 individuals. For equity, LU players are given the same total number of games to learn an appropriate opponent model.

IV. Experimental Results

In our previous work [12], we applied the genetic programming paradigm to the construction of computer Spoof players to evolve guessing strategies that maximise winnings against its opponents. We demonstrated through experiments that very effective guessing strategies were evolved, obtaining theoretically optimal strategies in most test cases, and near-optimal strategies otherwise. Our approach was for a game of simplified Spoof, considering only games of three players with our learning player guessing last (i.e. maximal information) and most of the hand-coded players used in training had deterministic guessing strategies.

In this paper, we compare the learned strategies of computer Spoof players constructed using genetic programming with those using a look-up table approach. In comparing the two learning players, we revisit our earlier work and consider a less restricted game environment, with our adaptive players playing in all different guessing positions. We also consider a broader range of fixed-strategy computer opponents for our adaptive players to play against, with particular emphasis on more players with non-deterministic guessing strategies.

We experiment with how our adaptive players are able to adjust to new environments, having already converged on a strategy for play in a different environment. We also reconsider our original means of evaluating our learning player's fitness, experimenting with the number of fitness samples used in player evaluation, and also the actual function used to measure success. With the latter, we contrast two different fitness functions — a direct-success fitness measure, based on the number of actual games won, and secondly, the pseudo-success measure we used previously that measures how well the learned model of an opponent fits reality.

A. Computer Opponents

To test how well our approaches are able to learn different opponent strategies, we create a number of fixed-strategy Spoof players for our learning players to play against. All players cast their guess values down to an integer before submitting them to the Spoof game engine as their guess. In the event of a desired guess already being taken, the game engine will automatically choose the closest integer to the desired value, checking first above and then below the desired value by ever increasing amounts until a unique value is selected.

The selection and guessing strategies for these players are detailed in Table I. This table uses the following terminology: c denotes the selected number of coins held by the player, n denotes the number of players in the game, and x is the player's guessing position. With exception to I and O, all

TABLE I

SPOOF OPPONENTS USED IN THIS STUDY

Table	Opponent	Selection	Guessing Strategy
1	Peak (P)	Randomly from [0 .. 3]	The maximum of the probabilistic distribution of possible totals assuming all players have selected randomly (i.e. $1.5n$).
2	Better Peak (BP)	Randomly from [0 .. 3]	The maximum of the probabilistic distribution of possible totals assuming all players have selected randomly, factoring in its own number of coins (i.e. $c + 1.5(n-1)$).
3	Low Peak (LP)	Randomly from [0 .. 3]	The same as BP except it assumes the lower average of 1 for each player rather than 1.5 (i.e. $c + 1(n-1)$).
4	High Peak (HP)	Randomly from [0 .. 3]	The same as BP except it assumes the higher average of 2 for each player rather than 1.5 (i.e. $c + 2(n-1)$).
5	Inside (I)	1 or 2	The same as BP.
6	Outside (O)	0 or 3	The same as BP.
7	Responsive (S)	Randomly from [0 .. 3]	Assumes previous opponents have guessed the maximum of the probabilistic distribution of possible totals after factoring in their own coins (i.e. like BP and its derivatives) and "reverse engineers" their guess in order to infer the number of coins held by the player (c_x), using 1.5 if the inference suggests an infeasible amount. Assumes also that the total for all the remaining players will be the maximum of the probabilistic distribution of possible totals. Final guess is then: $(\sum_1^{x-1} c_x) + c + 1.5(n-x)$, where x is integer position of the player. Behaves like BP when first to act.
8	Random (R)	Randomly from [0 .. 3]	Uniform random guess in the feasible range of totals, factoring in its own number of coins (i.e. $[c \,..\, 3(n-1)]$ inclusive).
9	Uniform Better Peak (UBP)	Randomly from [0 .. 3]	The same as BP except this player returns a randomised guess by adding a uniform random value between [$-n$ .. n] inclusive.
10	Gaussian Better Peak (GBP)	Randomly from [0 .. 3]	The same as BP except this player returns a randomised guess by adding a random Gaussian value with a mean of 0 and a standard deviation of $n/2$.
11	Uniform Responsive (US)	Randomly from [0 .. 3]	The same as D except this player returns a randomised guess by adding a uniform random value between [$-n$ .. n] inclusive.
12	Gaussian Responsive (GS)	Randomly from [0 .. 3]	The same as S except this player returns a randomised guess by adding a random Gaussian value with a mean of 0 and a standard deviation of $n/2$.

fixed players select their number of coins uniformly randomly within the allowable range [0 .. 3]. Players P, BP, LP, HP, I, O, and S all use deterministic guessing strategies based on the information available to them (their own selection, the number of players in the game, and the announced guesses of each opponent that is guessing beforehand). Players R, UBP, GBP, US, and GS all use non-deterministic guessing strategies. Non-deterministic guessing strategies are more difficult to counter due to the additional noise involved when attempting to build a model of the opponent strategies.

Many of our experiments are concerned with training our adaptive players to play in a certain *table*. We define the term table to represent a game of Spoof where a single adaptive player plays in a fixed guessing position against its opponent players (each of which being identical opponents taken from table I). Our adaptive players build collective models of *all* its opponents, and as such, the individual components are irrelevant. Although not reported here, experiments with mixed opponents have shown results consistent with the findings reported here.

The notation for players evolved for play in these tables is in the form $\mathrm{GPx_y}$ where x indicates the table of opponents used in training, and y indicates the guessing position of the evolved player. As an example, $\mathrm{GP4_2}$ represents a GP player trained against two High Peak (HP) players, playing (guessing) in second position. This numbering convention also applies to our look-up table players except that GP is replaced with the LU prefix.

TABLE II

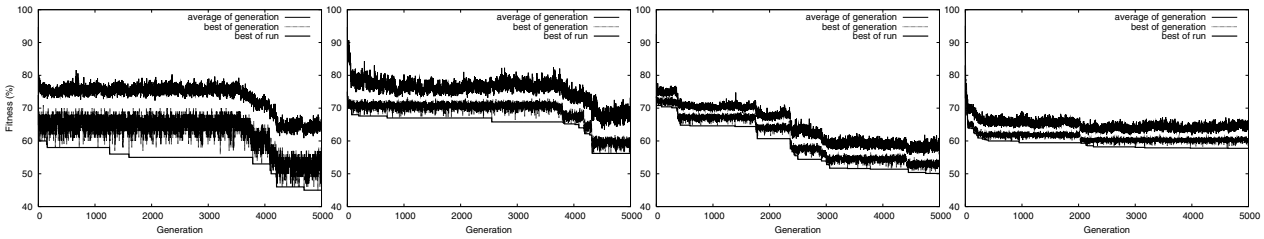
THE EFFECTS OF THE NUMBER OF GAMES PER FITNESS EVALUATION ON PERFORMANCE FOR THE GENETIC PROGRAMMING SYSTEM FOR THREE PLAYER SPOOF, GUESSING THIRD

Table	100 games	500 games	1000 games	2000 games
2	45	47	46	47
4	51	50	55	28
7	45	47	47	40
8	45	45	45	33
12	40	40	44	38

B. Number of Games Per Fitness Evaluation

In our first series of experiments, we experiment with varying the number of games played in order to evaluate individual strategies per generation of the genetic programming system. As the results of games are stochastic, multiple games must be played in order to get a reasonable measure of the "goodness" of a strategy. From this series of games, a strategy's *fitness* can then be determined by counting the number of times the strategy returns an incorrect total. We experiment with varying the number of games played in determining evolutionary fitness.

The evolution of $\mathrm{GP4_3}$ is shown in Fig. 1 with varying numbers of games used in evaluating the fitness of an individual: 100, 500, 1000 (as in our previous work), and



(a) 100 games (b) 500 games (c) 1000 games (d) 2000 games

Fig. 1. The effects of the number of games per fitness evaluation on the evolution of $\mathrm{GP4}_3$

2000. As expected, due to the increased level of noise, Fig. 1 shows that the best of generation plots are more erratic and differ by greater amounts from the average when using less games per fitness evaluation.

While at first glance it might seem that the 100 game scheme produces the best individual, it is important to remember that the fitnesses reported in Fig. 1 do not reflect the "true" fitness of an individual, but instead report the fitness from a limited number of samples. A fairer comparison of the performance of $\mathrm{GP4}_3$ (and other evolved strategies) using varied numbers of games per fitness evaluation is shown in Table II. This table reports the percentage of winning games (with drawn games repeated) for the best solution found in each experiment played over one million games.

Table II shows there is not a lot of difference between the 100 and 500 game evaluation schemes. However, the stronger selection pressure of 1000 game scheme seems to be advantageous, as can be witnessed by the improved performance at tables 4 and 12. This of course comes at the obvious cost of playing more games.

Table II shows one other interesting result. While one would expect that using more games per fitness evaluation would benefit learning (the amount of noise in the evaluation process is reduced by the greater sampling), the opposite is indeed observed for the 2000 game scheme. When using 2000 fitness samples, a distinct disadvantage is witnessed; the resultant players are typically significantly worse than those evolved using a lower number of games per fitness evaluation. The reason for this result is that with too much selection pressure, the lack of diversity among the population limits exploration. The lack of noise in the evaluation scheme causes the population to prematurely converge, becoming "trapped" in inferior regions of the solution space. This finding supports the conclusions made by Darwen, where he showed that it is possible to "over-sample" in the game of Backgammon [14]. A trade-off is indeed induced; with lower selection pressure the evolving population has more diversity allowing for a more diverse search of the solution space, but we also increase the likelihood that promising individuals are not included in the following generation.

One final observation can also be made concerning the complexity of solutions. With a fewer number of games per evaluation there is a greater chance that individuals will finish with the same overall fitness. When this occurs, secondary selection pressure (parsimony) will have a greater effect, resulting in simpler program-trees in these cases. This is confirmed by analysing the complexity of the resultant solutions. For $\mathrm{GP4}_3$ using the 100 game scheme, the final evolved program consists of 32 nodes. However, strategies evolved for the same table using 1000 and 2000 games per evaluation produced strategies containing 62 and 205 nodes respectively.

C. Performance Comparison for Three Player Spoof

In this section, we compare the learning and achieved win rates for both learning techniques for three player Spoof, playing in first, second, and third guessing positions. Drawn games are repeated with the guessing order remaining unchanged. In our previous work [12], we showed that our genetic programming approach was able to evolve strategies equal to or better than any of the hand coded non-adaptive strategies. Furthermore, by comparing each evolved strategy to the corresponding theoretically optimal countering-strategy, we showed that the strategies developed by the genetic programming system achieved optimal performance in most cases and near-optimal performance otherwise.

Table III reports the percentage of games won from one million games (draws are replayed) by the best solution found by both learning techniques at each of the tables for the different guessing positions in three player Spoof. Consistent with our earlier work, analysis of these experiments reveal that the adaptive methods produce the most successful players across all tables, reaffirming that specialisation is required for optimal play in Spoof. For brevity, results relating to the performance of the fixed strategies, as reported in [12], are not shown here.

In comparing the two learning approaches, Table III reveals that when guessing in first position for three player Spoof, GP players outperforms LU players. When playing in second position, GP players still outperforms LU players, but only by a marginal amount. However, Table III shows that in third position, the strategies learned by the LU players surpass those evolved by the GP players, allowing the LU players to obtain a higher level of performance (games won) than the GP players.

Recall that the later that a player is to act, the more potentially useful information there is to consider in making a decision (the announced guesses of the previously acted

TABLE III
PERFORMANCE OF BOTH LEARNING TECHNIQUES AT EACH TABLE FOR THREE PLAYER SPOOF

Table (x)	LUx_1	GPx_1	LUx_2	GPx_2	LUx_3	GPx_3
1	36	43	39	41	35	37
2	31	36	31	35	48	46
3	39	40	39	41	58	57
4	39	43	43	41	60	55
5	61	62	50	61	47	47
6	73	73	62	67	56	56
7	27	27	29	40	47	47
8	42	49	45	47	47	45
9	48	50	46	48	49	46
10	41	43	42	42	44	44
11	45	48	47	48	47	49
12	39	40	41	42	49	44
Avg	43	46	41	42	49	47

opponents), and hence the more complex the guessing algorithm will potentially need to be. Given the restricted size of the program-trees that can be evolved by a GP player (the depth of a program-tree is limited to 10 in our genetic programming system), we suspect that in these situations the GP players are unable to produce a program-tree of sufficient complexity to encode all required responses. In contrast, LU players maintain a separate action for every possible game state, allowing for a full mapping from every game state to desired action without affecting other game states. Perhaps the limit on functional complexity for our GP players account for the relatively weaker performance when compared to the LU players.

Further experiments have confirmed this theory. In Table III, we see that $GP4_3$ has the lowest relative performance compared to its look-up table based equivalent, winning only 55% of games played compared to the 60% achieved by $LU4_3$. Extending the maximum tree depth to 15 changes the result — $GP4_3$ now achieves a 60% win ratio of games, equal to that of the LU player. Indeed, all GP players match the performance of the LU players with this new maximum program-tree depth.

D. Strategy Analysis

It is often quite difficult to understand the expressions produced by the genetic programming system by inspection alone. Simplification helps, however some evolved strategies still remain quite complex even after removing redundant sub-expressions. As an alternative to inspection of the resultant algorithm, we can instead choose to analyse a strategy visually by examining the guesses made by the strategy for every possible game state (every combination of the variables potentially making up a player's strategy) — for three player Spoof guessing last, this is the first player's guess, the second player's guess, and the number of coins selected by the player. This graphical form also allows us to visualise the behaviour of our look-up table players, which have no associated function that controls their guessing algorithm.

Figs 2 and 3 visually depict the learned strategies for play at table 6 trained when guessing third for the genetic programming and look-up table approaches respectively. Despite their drastically different strategy plots, these two strategies are in fact both optimal players for table 6 (achieving the maximum number of wins possible for that position at the given table; again, see our earlier work [12] for a definition of optimality in this context).

Recall, the GP system bases its guess on the result of evaluating a program-tree, hence explaining the "regular" pattern for the resultant guessing algorithm. LU players amend their playing strategy on a case-by-case basis, thus resulting in the erratic landscape seen in Fig. 3.

E. Comparison of the Different Fitness Functions

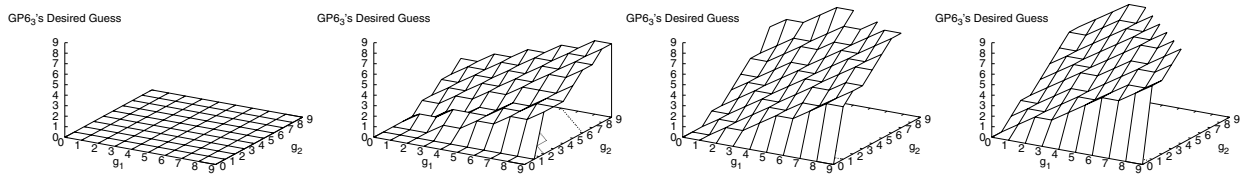
In past experiments, we evaluated our evolving strategies based on the accuracy of the learned model of their opponents (i.e. how well the opponent is able to predict the correct total), instead of simply just the number of games won by a strategy (note that these two may differ due to correct guesses already being taken). This was done in hope of minimising the effects of "luck" (due to the stochastic nature of the game), which may hinder the learning process. In this section, we revisit this choice, exploring the use of a more direct (but more noisy) fitness function based solely on the actual number of wins scored by a strategy.

While the pseudo-success measure has it advantages (it is less susceptible to noisy outcomes), it also suffers its own problems. For example, when using a pseudo-success measure, there is no distinguishing between a false guess that results in the loss of a game and a false guess that merely requires a game to be replayed. Also, as an artifact of using a pseudo-success measure, a correct guess that leads to a win is weighted the same as a correct guess in a game scenario where losing was inevitable anyway. These instances induce their own version of noise in the learning process.

Obviously if a learning player is to learn based on the number of times it wins, then drawn games must also be accounted for. We decided to disregard drawn games completely, and simply replay draws until a winner emerged. Of course, to be fair, these games are counted amongst the 1000 game training set. The reason we choose to disregard drawn games is that the true "value" of a draw would vary from strategy to strategy; for example, replaying in a situation where we win 50% of games is much better than replaying in a situation where we win only 10% of the time.

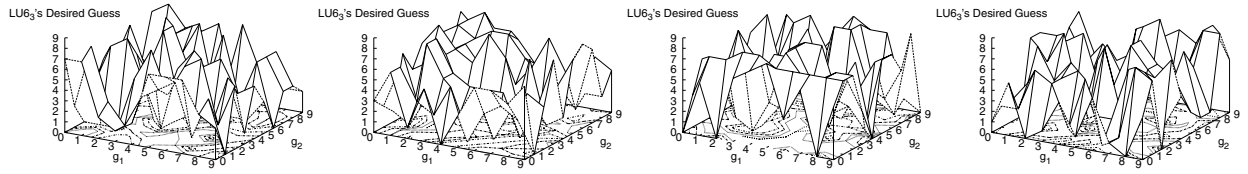
Fig. 4 compares the fitness evolution for $GP2_3$ for three player Spoof using both direct-success evaluation and pseudo-success evaluation, reporting the number of games lost by the best of generation individual. Note that while these values can not be compared directly (the two evolving strategies, by definition, use different fitness measurements), we plot in Fig. 4 the number of games lost for both players, even though the pseudo-success measure is actually used for evaluating one of them.

For the direct-success measure, a rapid decrease in fitness is realised in the first 100 or so generations of Fig. 4, followed by a gradual decrease until about generation 2300. After this time, no notable improvements occur. In contrast, using the pseudo-success measure, the development of a



(a) Selection = 0 (b) Selection = 1 (c) Selection = 2 (d) Selection = 3

Fig. 2. Visual representation of strategy $GP6_3$



(a) Selection = 0 (b) Selection = 1 (c) Selection = 2 (d) Selection = 3

Fig. 3. Visual representation of strategy $LU6_3$

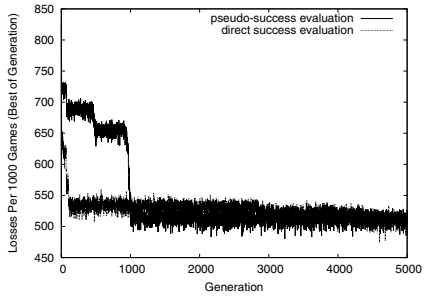


Fig. 4. Performance of the direct-success and pseudo-success fitness measures for the evolution of $GP2_3$

relatively good strategy is much slower, with the corresponding learning player making all of its progress at three "milestones" at roughly generations 100, 500, and 1000. There is no real improvement made after this point, with the player's performance remaining effectively constant until finishing with a win percentage roughly equal to that of the direct-success measure (at about 47%).

We observe that while the direct-success player learns at a much faster rate initially, its performance stagnates after the initial improvement, only gradually improving to its optimum at around generation 2300. In contrast, the pseudo-success player obtains its optimum performance much sooner, some 1300 generations prior to the direct-success player at around generation 1000. Indeed, Fig. 4 shows that the direct-success approach learns much faster earlier in the run, gaining the initial advantage before being surpassed by the pseudo-success measure in the early-to-middle stages of the run, until eventually catching up again towards the middle of the run, finishing with equal performance to the pseudo-success measure. Although only reported for $GP2_3$, this pattern is consistent for the other tables; direct-success evaluation having the initial advantage, but taking longer to reach the eventual optimum win rate.

The reason for the initial divergence in performance of the two schemes is due to the requirements imposed by each. Using the pseudo-success measure, a player needs to learn to guess correctly for *all* possible game states, not only the game states the player can win (as is the case using direct-success evaluation). With direct-success evaluation, a player can realise a rapid improvement in performance by learning the correct actions of only a subset of the game states required by the the pseudo-success measure, in essence allowing a player to build up successful functionality faster. The extra "burden" imposed by the the pseudo-success measure impedes the initial performance (measured only by wins) of a player using this scheme, but the additional information gained from the inevitable losses allows players using this scheme to obtain their optimal performance much sooner than the direct-success approach. The training process is longer because the learning player must consider a greater body of knowledge, however it is from this that it is able to more accurately model its opponents.

One may expect that because of this, players evolved using a direct-success measure will consistently perform better. However as Table IV shows, the two schemes produce strategies that are usually about equal in performance. Table IV reports the percentage of wins achieved by strategies evolved by the genetic programming system using both fitness measurement schemes. The percentages are based on

TABLE IV
PERFORMANCE OF THE DIRECT-SUCCESS AND PSEUDO-SUCCESS FITNESS MEASURES FOR DIFFERENT TABLES

Table (x)	GPx_3 Direct success	GPx_3 Pseudo-success
1	37	37
2	46	46
3	55	57
4	57	55
5	47	47
6	53	56
7	47	47
8	45	45
9	46	46
10	43	39
11	49	49
12	41	44
Avg	47	47

the results of playing one million games.

Both direct and pseudo-success approaches commonly achieve the same win rate despite the more accurate opponent model held by the pseudo-success player. The direct-success player does not need to determine the correct guess for games where the total is unavailable (the fitness measure does not reward doing so), and hence is still able to achieve the same win rate. The equivalent strategy as produced using pseudo-success evaluation contains a more accurate opponent model because it "wants" to guess correctly in unwinnable game scenarios (when the desired guess has already been taken). It is of course, in these situations, unable to win the game and thus the achieved performance remains unchanged.

One possible benefit of a direct-success approach is the saving that can be made by not wasting effort in building "fruitless" functionality to accurately predict the result of unwinnable games (a pseudo-success player will still attempt to learn the correct response in these cases). As our evolving program-trees have limited complexity, it seems wasteful to maintain parts of the tree dealing with these situations; complexity that could better be utilised to provide greater functional when dealing with winnable game states. Interestingly, this superfluous functionality that a pseudo-success player develops seems to indeed be useful when considering adaptation, as we show in Section IV-F.

Our results suggest that each scheme has its own strength, and neither dominates on the whole. But perhaps a combination of the two evaluation techniques would be beneficial in improving the learning rate of our genetic programming system. For example, against static opponents one might use a direct-success measure until the rate of improvements falls below some threshold, switching then to a pseudo-success measure for further refinement. This would allow a basic functionality to be established much quicker than using purely pseudo-success evaluation, and near-optimal functionality to be learned faster than using only direct-success evaluation. A similar approach could be used against adapting opponents, with perhaps a sudden decrease in wins over a certain number of games triggering the system to return to direct-success evaluation.

F. Learning Against Changes in Opponent Strategy

In this section we experiment with how both the GP and LU players are able to adapt to changes in opponent playing strategies. We take a number the evolved players from Table I and attempt to continue their evolution against a new set of opponents, thus simulating the scenario where opponents modified their behaviour during a run.

Table V shows the adaptive abilities of our GP and LU players. The column labelled "starting table" indicates the table the learning players were originally trained to play on. The "achieved fitness" column lists the corresponding fitness of each learned strategy on its own table. The "starting fitness" column lists the fitness of this same strategy in its new environment before any new learning takes place (i.e., immediately after its opponents switch strategies). The "ending table" column indicates the new opponent strategy the learning players now must compete against. "Ending fitness" is the achieved fitness after 5000 generations (for GP players) and after 250 million games (for LU players) of re-learning against these new strategies.

Table V shows that the starting fitnesses of the LU players are generally much higher than that of the GP players, suggesting that the strategies employed by the GP players are more widely applicable to other opponents than the strategies employed by the LU players. This is due to the GP players learning a function that maps over the entire set of game states, whereas look-up table players learn about individual game states by experience, with no effect on any other game states. The relatively erratic nature of the LU player's strategy visualisation can be seen in Fig. 3, as opposed to the more "regular" strategy observed for the GP player in Fig. 2.

The more regular functionality of the GP player seems to be what gives the GP player its advantage over the LU players after the change in opponent strategy. Since the GP player is represented as a function, game states which have not yet been encountered by the learning player may still have yield "reasonable" guesses. Contrast this with LU players, who having never witnessed correct responses for the unused game states, revert to random guessing. These "unseen" games states may be realised when playing against different opponents and hence, having a general idea of what to guess seems better than having no idea whatsoever. It is this effect that gives the GP players a "head start" on adapting toward its new opponents.

Table V also shows that the ending fitness of the GP players are considerably lower than those of the LU players (recall that fitness is measured in terms of the number of incorrect guesses), indicating that the GP system is more effective in adapting to new strategies. The main reason for this is that the structure of an evolving program-tree is much more easily manipulated than the huge look-up table employed by the LU players. The LU players can only update one entry per game, whereas the GP players can totally augment its behaviour with one small change to its program-tree design. This is obviously an area where LU players have

TABLE V

PERFORMANCE OF BOTH LEARNING TECHNIQUES AGAINST CHANGES IN STRATEGY FOR THREE PLAYER SPOOF, GUESSING THIRD

Starting Table	Achieved Fitness LUx_3	Achieved Fitness GPx_3	Ending Table	Starting Fitness of LUx_3	Starting Fitness of GPx_3	Ending Fitness of LUx_3	Ending Fitness of GPx_3
7	575	493	8	855	748	760	492
8	769	707	4	779	735	746	501
8	769	707	7	749	748	548	492
12	713	675	8	779	832	759	675

trouble. Even though a number of approaches are possible to speed up the LU player's ability to adapt (such as resetting the entire look-up table after a number losses, or perhaps altering weights by varying amounts), this approach still falls far short of the adaptability inherent with a modifiable program-tree representation. Furthermore, such approaches require the application of domain-specific knowledge, which we have chosen to minimise due to the restrictions that such assumptions place on the system.

V. CONCLUSIONS

In this paper, we compare the use of look-up table players with genetic programming for strategy development in the game of Spoof. We had previously shown that genetic programming could be used to construct automated computer Spoof players which evolved guessing strategies that consistently outperformed all of our hand-coded, non-adaptive players. Our look-up table players did not fair as well as the genetic programming players, especially in earlier guessing positions. Look-up table players had their greatest success when guessing last, where it matched and occasionally outperformed the genetic programming players. However, upon increasing the maximum allowable depth for program-trees, we found that the GP player was able to match this level of performance, indicating that a greater level of functional complexity is required in these situations.

While often achieving similar performance as our evolved strategies, we found the LU players to be less adaptable to changes in opponent strategies. The GP players consistently outperform the LU players when forced to adapt for play against a new fixed-strategy opponent (simulating opponents which have altered their strategy).

The ability of an evolving population to achieve optimal or near-optimal play is influenced by the amount of selection pressure it is subjected to. We found that it is possible to use too many samples (games) in evaluating the fitness of an individual, resulting in worse performance when compared to players evolved using less games per fitness evaluation. Also, a side-effect of lowering the number of samples is that parsimony has a greater influence on selection, resulting in solutions with a smaller program-tree size than those evolved using more games per fitness evaluation.

Our experiments with the two different fitness evaluation schemes show that both measures are capable of developing optimal guessing strategies, however the pseudo-success measure reaches its optimum level sooner. The direct-success approach takes longer to develop the same level of ability, however it is much better at rapid improvement from a neutral starting state. The pseudo-success measure minimises the effect of noise, but this results in extraneous functionality being developed which, due to complexity restraints of the GP system, could potentially limit the functionality of areas that require it. It is also interesting to note that when experimenting against opponents who change their strategies, we found such "fruitless" functionality (which LU players do not develop) to be beneficial.

REFERENCES

[1] L. Barone and L. While, "Adaptive learning for poker," in *GECCO 2000: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2000, pp. 566–573.

[2] Y. Azaria and M. Sipper, "Using GP-gammon: using genetic programming to evolve backgammon players," in *Proceedings of the 8th European Conference on Genetic Programming*. Springer, 2005, pp. 132–142.

[3] D. Fogel, "Evolving strategies in blackjack," in *Proceedings of the 2004 Congress on Evolutionary Computation (CEC '04)*. IEEE Publications, 2004, pp. 1427–1432.

[4] ——, "Evolving behaviors in the iterated prisoner's dilemma," *Evolutionary Computation*, vol. 1, no. 1, pp. 77–97, 1993.

[5] P. Hingston and G. Kendall, "Learning versus evolution in iterated prisoner's dilemma," in *Proceedings of the 2004 Congress on Evolutionary Computation (CEC '04)*. IEEE Publications, 2004, pp. 364–372.

[6] J. R. Koza, *Genetic Programming: On the Programming of Computers By Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[7] J. Lohn, G. Hornby, and D. Linden, "Evolutionary antenna design for a NASA spacecraft," in *Genetic Programming Theory and Practice II*. Springer, 2004, pp. 301–315.

[8] A. D. Parkins and A. K. Nandi, "Genetic programming techniques for hand written digit recognition," *Signal Processing*, vol. 84, no. 12, pp. 2345–2365, 2004.

[9] J.-Y. Potvin, P. Soriano, and M. Vallee, "Generating trading rules on the stock markets with genetic programming," *Computers & Operations Research*, vol. 31, no. 7, pp. 1033–1047, 2004.

[10] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler, "Co-evolving soccer softbot team coordination with genetic programming," in *RoboCup-97: Robot Soccer World Cup I*. Springer-Verlag, 1998.

[11] E. Burke, S. Gustafson, and G. Kendall, "A puzzle to challenge genetic programming," in *Proceedings of the 5th European Conference on Genetic Programming*. Springer-Verlag, 2002, pp. 238–247.

[12] M. Wittkamp and L. Barone, "Evolving adaptive play for the game of spoof using genetic programming," in *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games*. IEEE Computational Intelligence Society, 2006.

[13] "Spoof strategy," Wikipedia — The Free Encyclopedia, October 2005, URL: `http://en.wikipedia.org/wiki/Spoof_Strategy`.

[14] P. Darwen, "Computationally intensive and noisy tasks: co-evolutionary learning and temporal difference learning on Backgammon," in *Proceedings of the 2000 Congress on Evolutionary Computation (CEC '00)*. IEEE Publications, 2000, pp. 872–879.