2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Evolving a Fuzzy Goal-Driven Strategy for the Game of Geister: An Exercise in Teaching Computational Intelligence

Andrew R. Buck, *Student Member IEEE*, Tanvi Banerjee, *Student Member IEEE*, and James M. Keller, *Fellow IEEE*

***Abstract*—This paper presents an approach to designing a strategy for the game of Geister using the three main research areas of computational intelligence. We use a goal-based fuzzy inference system to evaluate the utility of possible actions and a neural network to estimate unobservable features (the true natures of the opponent ghosts). Finally, we develop a coevolutionary algorithm to learn the parameters of the strategy. The resulting autonomous gameplay agent was entered in a global competition sponsored by the IEEE Computational Intelligence Society and finished second among eight participating teams.**

I. Introduction

GEISTER (German for ghosts) is a strategic board game played between two players on a 6x6 grid. Each player is given four good ghosts and four evil ghosts that are initially placed in any configuration within the 2x4 home areas of each player as shown in Fig. 1. The nature of each ghost (good or evil) is marked on the back of the game piece and is hidden to the opponent player. The players alternate moving their ghosts forward, backward, left, or right (never diagonally) in order to achieve one of three possible victory conditions:

1) Capture all of the opponent's good ghosts
2) Have the opponent capture all evil ghosts
3) Move a good ghost off the board from one of the opponent's corner spaces

An opponent's ghost is captured by moving a ghost onto the same space, at which point its true nature is revealed.

This is a game involving incomplete information. Players do not know with complete certainty which opponent ghosts are good and which are evil. Games of this type differ from games of perfect information such as chess and Go, and require additional strategies to manage the uncertainty. One popular approach is *determinization* [1], in which the uncertain features are randomly sampled from the set of possible values. However, this approach is inefficient and requires a large computational budget to simulate all of the various possibilities. An alternate approach that we use in this paper is to model the uncertain features as fuzzy sets and use a fuzzy inference system to develop the gameplay strategy. This has the advantage of computational speed as well as allowing for a custom, hand-picked set of rules. This is particularly important for developing games on limited hardware such as mobile phones [2].

Our method for developing an autonomous agent to play Geister consists of three parts, utilizing the three main research areas of computational intelligence (CI). It was designed originally as a series of projects for an introductory course on computational intelligence, and as an entry in the "Ghost Challenge 2013" competition organized by the IEEE Computational Intelligence Society. This challenge was issued to promote student involvement and to encourage novel approaches for developing artificial agents using CI techniques.

First, we design a fuzzy inference system that evaluates the utility of possible moves. For inputs, the system uses both observable game state features and estimates of the unknown opponent ghost natures. The outputs are computed using a set of fuzzy rules and are interpreted as the value of a particular ghost pursuing a specific goal. The ghost with the highest valued goal is chosen to be moved in pursuit of that goal.

Second, we use a neural network to perform the estimation of the opponent ghost natures. Observable features such as initial position and movement are used to construct a feature vector for each opponent ghost. A neural network is then trained using a dataset gathered over many games to classify the ghosts as good or evil. During gameplay, the trained neural network is used to compute a good and evil confidence value for each opponent ghost.

Finally, we use a coevolutionary algorithm to improve our hand-built strategy. The membership functions and rules of the fuzzy inference system, along with the weights of the neural network, are encoded in a chromosome structure that represents a particular strategy. A population of these strategies competes and evolves over time, learning a better set of strategy parameters than our initial implementation.

Evolutionary methods such as coevolution have been used by many researchers to learn interesting and unique strategies for games of incomplete or imperfect information [3]. Methods such as *complexification* [4] show that by gradually increasing the complexity of a genetic representation, more elaborate and sophisticated strategies can be developed. We use these ideas to design our algorithm and construct our game-playing agent.

The remainder of this paper is organized as follows. In Section II we discuss the development of the goal-driven fuzzy inference system. Section III covers the design and training of the neural network to estimate the opponent ghost natures. Section IV outlines our coevolutionary algorithm to learn the strategy parameters. The performance of our agent in local play and the global IEEE CIS competition is

A. R. Buck, T. Banerjee, and J. M. Keller are with the Department of Electrical and Computer Engineering, University of Missouri-Columbia, MO 65211, USA (email: arb9p4@mail.missouri.edu; tsbycd@mail.missouri.edu; kellerj@missouri.edu).

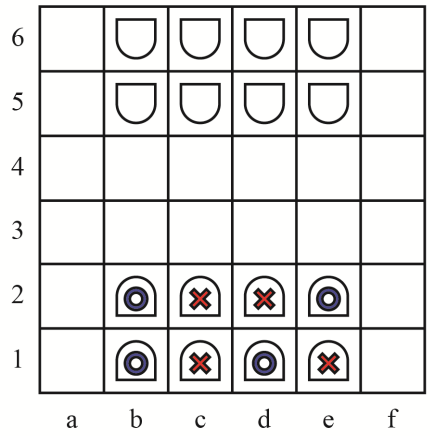

Fig. 1. Initial board position for Geister. Each player has four good ghosts (marked with a blue circle) and four evil ghosts (marked with a red cross). The markings are hidden to the other player so that the true nature of the opponent ghosts is unknown.

discussed in Section V. Lastly, our conclusions and ideas for future work are given in Section VI.

II. Creation of a Fuzzy Inference System

The core decision component of our autonomous agent is a fuzzy inference system (FIS) that can evaluate the utility of possible actions. Each ghost can move in up to four directions resulting in up to 32 possible actions on a player's turn (although, due to board boundaries and the inability to move onto friendly ghosts, the number of actions is typically much smaller). While it may be possible to design a FIS that provides the values of these actions directly, we opted instead to use a goal-driven approach in which each ghost is given a set of goals to pursue. The value of each goal is computed by the FIS for each of the player's ghosts. This allows us to use higher-level features as inputs for the FIS and to more appropriately assess if the output matches the desired behavior.

A. Input Features

Most of the input features for our system are integer measurements, either a distance (in moves) or a number of ghosts. In order to define fuzzy rules based on linguistic expressions, we define linguistic terms for each input feature and a corresponding membership function to map the crisp measurements into fuzzy values. We define a triangular membership function as Tri(a, b, c), where the interval [a, c] is the support and b is the peak of the membership function. Likewise, we define a trapezoidal membership function as Trap(a, b, c, d), where the interval [a, d] is the support and the interval [b, c] is the core of the membership function. The full list of input features is given in Table I.

The first set of input features describe the current state of the game and are the best indication of who is currently winning.

1) Captured Good (CG): The number of good opponent ghosts that have been captured.
2) Captured Evil (CE): The number of evil opponent ghosts that have been captured.
3) Lost Good (LG): The number of good ghosts that have been captured by the opponent.
4) Lost Evil (LE): The number of evil ghosts that have been captured by the opponent.

Each of these features is an integer in the range [0, 3]. If all four good ghosts or all four evil ghosts are lost by either player, the game is over.

The second set of input features applies to a specific ghost

TABLE I
Input Features to the Fuzzy Inference System

Input Linguistic Variable	*Term*	*Membership Function*
Captured Good (CG)	Low (L)	Tri(0, 0, 3)
	High (H)	Tri(0, 3, 3)
Captured Evil (CE)	Low (L)	Tri(0, 0, 3)
	High (H)	Tri(0, 3, 3)
Lost Good (LG)	Low (L)	Tri(0, 0, 3)
	High (H)	Tri(0, 3, 3)
Lost Evil (LE)	Low (L)	Tri(0, 0, 3)
	High (H)	Tri(0, 3, 3)
Closest Ghost Distance (CGD)	Adjacent (A)	Tri(0, 1, 2)
	Near (N)	Trap(0, 0, 2, 6)
	Far (F)	Trap(2, 6, 10, 10)
Closest Ghost Good Confidence (CGGC)	Low (L)	Tri(0, 0, 1)
	High (H)	Tri(0, 1, 1)
Closest Ghost Evil Confidence (CGEC)	Low (L)	Tri(0, 0, 1)
	High (H)	Tri(0, 1, 1)
Distance to Opponent Exit (DOE)	Adjacent (A)	Tri(0, 1, 2)
	Near (N)	Trap(0, 0, 2, 6)
	Far (F)	Trap(2, 6, 8, 8)
Exit Congestion (EC)	Low (L)	Tri(0, 0, 8)
	High (H)	Tri(0, 8, 8)
Opponent Distance to Home Exit (ODHE)	Adjacent (A)	Tri(0, 1, 2)
	Near (N)	Trap(0, 0, 2, 7)
	Far (F)	Trap(2, 7, 8, 8)
Distance to Home Exit (DHE)	Adjacent (A)	Tri(0, 1, 2)
	Near (N)	Trap(0, 0, 2, 4)
	Far (F)	Trap(2, 4, 10, 10)
Others Distance to Home Exit (OTDHE)	Adjacent (A)	Tri(0, 1, 2)
	Near (N)	Trap(0, 0, 2, 7)
	Far (F)	Trap(2, 7, 8, 8)

TABLE II
Output Goals of the Fuzzy Inference System

Output Goal	*Term*	*Membership Function*
Capture Closest Ghost (CAP)	Low (L)	Tri(0, 0, 1)
	Medium (M)	Tri(0.1, 0.5, 0.9)
	High (H)	Tri(.6, 1, 1)
Block (BLOCK)	Low (L)	Tri(0, 0, 1)
	Medium (M)	Tri(0.1, 0.5, 0.9)
	High (H)	Tri(.6, 1, 1)
Exit (EXIT)	Low (L)	Tri(0, 0, 1)
	Medium (M)	Tri(0.1, 0.5, 0.9)
	High (H)	Tri(.6, 1, 1)
Escape (ESC)	Low (L)	Tri(0, 0, 1)
	Medium (M)	Tri(0.1, 0.5, 0.9)
	High (H)	Tri(.6, 1, 1)
Tempt Opponent Ghost (TEMPT)	Low (L)	Tri(0, 0, 1)
	Medium (M)	Tri(0.1, 0.5, 0.9)
	High (H)	Tri(.6, 1, 1)

and measures the distance to the closest opponent ghost and its estimated nature. We consider only the closest opponent ghost for simplicity.

5) Closest Ghost Distance (CGD): The distance to the closest opponent ghost.
6) Closest Ghost Good Confidence (CGGC): The confidence that the closest ghost is good.
7) Closest Ghost Evil Confidence (CGEC): The confidence that the closest ghost is evil.

The next set of input features also applies to a specific ghost and is used to determine how easily the ghost could win the game by exiting from one of the opponent corners.

8) Distance to Opponent Exit (DOE): The distance to the closest opponent corner exit.
9) Exit Congestion (EC): The number of opponent ghosts in the rectangular region between this ghost and the closest opponent corner exit.

The final set of input features is used to determine how close the opponent is to escaping from a home corner exit and how difficult it would be to block.

10) Opponent Distance to Home Exit (ODHE): The fewest number of moves an opponent ghost would need to exit from a home corner.
11) Distance to Home Exit (DHE): The distance between this ghost and the home corner nearest to an opponent ghost.
12) Others Distance to Home Exit (OTDHE): The number of our own ghosts that are closer than this ghost to blocking the home corner exit nearest to an opponent ghost.

B. Output Goals

The outputs of the FIS represent the utility values of different goals that a ghost can pursue. These are represented in the normalized range [0, 1] and are assigned linguistic terms for use in the fuzzy rules. The membership functions of the following goals are given in Table II.

1) Capture Closest Ghost (CAP): Move this ghost toward the nearest opponent ghost and capture it.
2) Block (BLOCK): Move this ghost toward one of the home corner exits to block an opponent ghost from escaping.
3) Exit (EXIT): Move this ghost toward one of the opponent's corner exits and off the board.
4) Escape (ESC): Move this ghost away from any opponent ghosts to avoid being captured.
5) Tempt Opponent Ghost (TEMPT): Move this ghost toward a space adjacent to an opponent ghost with the hope that it will become captured.

C. Rules

We design a set of rules to indicate under which circumstances each of the output goals is a useful pursuit. The "Capture Closest Ghost" and "Block" goals can be applied to both good and evil ghosts, however the "Exit" and "Escape" goals apply only to good ghosts and the "Tempt" goal applies only to evil ghosts. These restrictions are not strictly necessary, as it may be a useful bluffing strategy for a good ghost to imitate an evil ghost or vice versa. We allow for this flexibility in the final evolutionary training of our method.

The rules for the goals are given in Tables III-VII. Each row in a rule table represents a separate rule. These were hand-picked to represent a reasonable initial strategy for the game of Geister. Along with the membership functions defined for the input and output features, this comprises a Mamdani Fuzzy Inference System. To determine the output value of a particular goal, the crisp input feature values are fuzzified according to the defined membership functions. The minimum firing strength of each antecedent in the rule determines the maximum value of the consequent membership function. The consequent membership functions are summed together and defuzzified into a crisp value using the "mean of maximum" defuzzification method.

TABLE III
RULES FOR THE GOAL "CAPTURE CLOSEST GHOST"

Inputs				***Outputs***
CE	CGD	CGGC	CGEC	CAP
H				L
			H	L
L	N	¬L		M
L	A	H		H

TABLE IV
RULES FOR THE GOAL "BLOCK"

Inputs			***Outputs***
ODHE	DHE	OTDHE	BLOCK
F			L
N	N	L	H
N	N	M	L
N	N	H	L

TABLE V
RULES FOR THE GOAL "EXIT"

Inputs					***Outputs***
CG	CE	LG	DOE	EC	EXIT
				H	L
L	H				M
		L	N	L	M
			A		H

TABLE VI
RULES FOR THE GOAL "ESCAPE"

Inputs		***Outputs***
LG	CGD	ESC
L	F	L
	N	M
H	A	H

TABLE VII
RULES FOR THE GOAL "TEMPT"

Inputs		***Outputs***
LE	CGD	TEMPT
L	F	M
H		M
H	A	H

D. Determining an Action

After applying the FIS to all ghosts to get the value of each goal, we must now determine which ghost to move. The goals for a ghost each have a location on the board. For the goal "Capture Closest Ghost," the location is the space of the closest opponent ghost. For "Block," the location is the home corner space closest to an opponent ghost. For "Exit," the location is the closest opponent corner exit. For "Escape" and "Tempt," there may be multiple goal locations. Any adjacent space with no threat of being captured on the next turn serves as a location for the "Escape" goal. The four spaces adjacent to the closest opponent ghost work as locations for the "Tempt" goal, provided that they are not already occupied by one of our own ghosts.

The goal locations for each ghost are assigned to zero, one, or two possible movement directions. If the goal is directly above, below, to the left, or to the right of the ghost, the value of that goal is added to the value of movement in that direction. If the goal is at an angle, the two directions on either side of the goal location each receive half of the goal's value. If any of the movement directions are not valid moves, such as moving off the board (except in the case of exiting from an opponent's corner) or moving onto one of our own ghosts, the value of movement in that direction is fixed to zero.

After accumulating the values of the goals for each ghost in terms of movement direction values, the list of possible moves is sorted from most valuable to least valuable. We use a probabilistic selection strategy, selecting the best possible move 80% of the time, and moving on to the next possible move the rest of the time. This move again has an 80% chance of being chosen and the process repeats until a move is picked or the list is exhausted. This approach adds a random element to our strategy making it more difficult for the opponent to guess our moves. It also helps to prevent infinite loops when playing against itself or a similar strategy.

III. Profiling the Nature of an Opponent Ghost Using a Neural Network

All of the input features in the previous section can be measured through direct observation except two: "Closest Ghost Good Confidence" and "Closest Ghost Evil Confidence." Because the true natures of the opponent ghosts are hidden, these values must be estimated. Our approach is based on the work done by Aiolli and Palazzi in [5]. In their work, they record a set of 17 features for each opponent ghost over the course of an entire game. These are then used to build a nearest neighbor classifier giving the predicted nature of the opponent ghost.

We use the same set of features as described in [5], but with a Neural Network to estimate the confidence that a ghost is good or evil. The first eight features are binary values corresponding to the initial position of the opponent ghost. Each ghost has a different one of these features that is non-zero. The next two features are also binary and indicate if the ghost was the first or second piece to be moved. The next three features are integer valued and indicate how many forward, backward, and lateral moves the ghost made. The final four features count the different ways the ghost responds to being threatened, either by capturing an opponent ghost, escaping to a safe location, moving to threaten another ghost, or not moving at all.

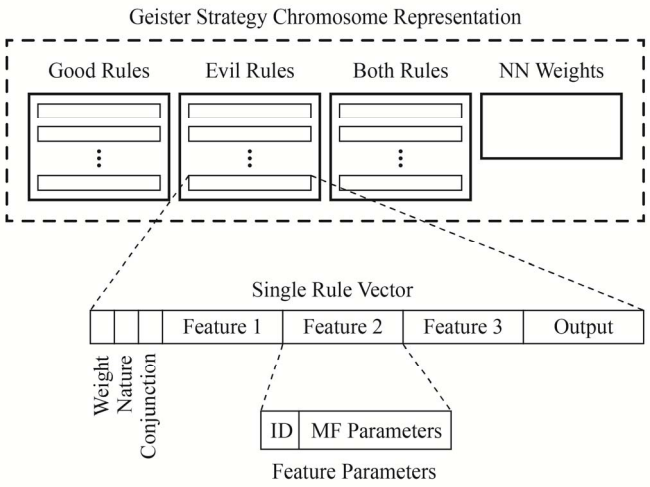

Fig. 2. Chromosome representation of a strategy. Each chromosome contains a rule base for good ghosts, evil ghosts, and one for both, along with a set of neural network weights. The rule bases contain rule vectors that describe up to three input features and a single output goal. The features are represented with an integer ID and four parameters for a trapezoidal membership function.

These features are updated and collected throughout the course of the game. The final feature values for each ghost are saved at the end of each game resulting in eight new sample vectors: four good ghost samples and four evil ghost samples. After playing 100 games with the hand-built strategy described above playing against itself, we obtain a training dataset of 800 samples with half representing good ghosts and half representing evil ghosts. During these training games, the good and evil confidence values are fixed at 0.5, ignoring the nature estimation step. While it would have been helpful to build the training dataset from games played between human players or more advanced AI players, this was not feasible at the time. We instead compensate for this deficiency by updating the weights of the neural network during the evolutionary learning phase described in the next section.

We construct a multi-layer perceptron with 10 nodes in the hidden layer, 17 inputs, and 2 outputs corresponding to the good and evil confidence values. This network is then trained on the collected dataset using the backpropagation algorithm. Once the network weights are obtained, new features from an opponent ghost sampled during gameplay can be passed through the network to obtain the confidence values that the ghost is good or evil.

IV. Learning Parameter Values with Coevolution

The strategy we have developed thus far has been based on the hand-picked rules and membership functions that make up the fuzzy inference system. While this represents a good starting point, there is still room for improvement using computational optimization techniques. We design an algorithm based on coevolution that allows multiple versions of our strategy to compete with one another in order to move

toward the optimal strategy. After many iterations of the algorithm, the strategies evolve to become more competitive than our original implementation.

A. Representation

First, we need to encode our FIS and neural network as a parameterized chromosome. An overview of our representation is given in Fig. 2. We represent the FIS as three separate rule bases: one containing rules for good ghosts, one with rules for evil ghosts, and one with rules that are applied to both. Each rule base contains a variable number of rules, with each rule taking the form of a 23-parameter vector.

The first parameter in a rule vector is a floating-point number between 0 and 1 indicating the weight of the rule. The consequent of the rule is multiplied by this value to modify its importance. The second parameter is an integer indicating if the rule is in the good, evil, or both rule bases. This is used during the initialization and mutation procedures. The third parameter is a Boolean value indicating if the conjunction for the rule should be "And" or "Or."

The remaining 20 parameters of a rule vector are grouped into four sets of five values representing the membership functions of up to three input features and one output goal. Each of these sets contains one integer parameter indicating which feature or goal the membership function applies to and four floating-point values used as parameters for a trapezoidal membership function. The domains of each feature are normalized into the range [0, 1], and the membership functions (defined for our hand-built strategy in Tables I and II) are explicitly defined in each rule. The floating-point values must be sorted in ascending order, and can also represent triangular membership functions if the two middle values are equivalent. The integer parameter for an input feature may be zero to indicate an absence of that feature, in which case the four membership function parameters are ignored. This produces a rule with fewer than three antecedents; however, each rule must have at least one antecedent and an output goal. There must also be at least one rule for good ghosts and at least one rule for evil ghosts among the three rule bases.

The parameters of the neural network are stored in a straightforward manner. The network has 17 inputs, 10 hidden nodes, and 2 outputs. Including the bias terms, this gives 180 values between the input and hidden layers and 22 values between the hidden and output layers for a total of 202 floating-point parameters. These are stored in tables, but they can be linearized into a single 202-element vector for the purposes of initialization, crossover, and mutation.

Algorithm I: Coevolutionary Algorithm

```
/* Initialization */
Initialize population P with popSize copies of our
   hand-built strategy
For i = 2 to popSize do
   P_i = mutate(P_i)
End For

generation = 0
While stopping criteria not met

   /* Evaluate Fitness */
   For i = 1 to popSize do
      fitnessValues_i = 0
      For j = 1 to numOpponents do
         Opponent = random strategy from P
         wins = 0; losses = 0; draws = 0
         Play numGames games against Opponent
         Update wins, losses, and draws
         score = 2*wins – 2*losses – draws
         fitnessValues_i += score
      End For
   End For

   /* Create Next Generation */
   P^new = {}
   Sort P by fitnessValues in descending order
   Copy first eliteSize strategies from P into P^new
   While |P^new| < |P|
      Select two parents p_1 and p_2 from P using
         tournament selection with a tournament size of t
      With probability crossoverProb, create children:
         [c_1, c_2] = crossover(p_1, p_2)
      With probability mutationProb, mutate children:
         c_1 = mutate(c_1); c_2 = mutate(c_2)
      Add c_1 and c_2 into P^new
   End While
   generation++

   /* Maintain Diversity */
   If generation is a multiple of restartRate then
      Replace last restartSize elements of P^new with new
         random strategies
   End If

End While
Return population of evolved strategies P
```

B. Initialization

The initial population for our evolutionary algorithm consists of several mutated copies of our hand-built strategy, along with an unmodified version. As the algorithm progresses, new randomly-built strategies are needed to maintain diversity and promote exploration. We create a random strategy by adding a set number of random rules (about 20) to the rule bases and by generating a set of random weights for the neural network.

Each parameter in a rule has a bounded domain, and a random rule is created by selecting each parameter from a uniform distribution over the appropriate domain. The second parameter in each rule indicates which rule base the rule belongs to. We must sort the four membership function parameters for each input feature and the output goal, and we must also ensure that each rule has at least one input antecedent. Additionally, there must be at least one rule for good ghosts and at least one rule for evil ghosts across the three rule bases.

C. Crossover

The purpose of crossover in an evolutionary algorithm is to recombine the information in two or more parents to produce offspring that share information from all of the parents. We perform crossover between two chromosome strategies by exchanging their rules and neural network parameters. Complete rules are exchanged without modification, whereas the weights in the neural network are considered individually.

Two child strategies are created initially as copies of the parents. We iterate over all of the rules from each rule base and with a certain crossover probability (80% in our method), each rule is given to the other child strategy. Likewise, we iterate over each parameter in the neural networks of the two strategies and exchange the weights using the same crossover probability.

D. Mutation

Mutation is applied to a single chromosome strategy in order to vary the parameters and explore new regions of the search space. With a certain mutation probability (10% in our method), each parameter in the rule bases and neural networks is changed to a new random value. For rule parameters that are integers, a new value is chosen within the appropriate domain. For real-valued parameters in the rules and neural networks, a small amount of Gaussian noise (σ = 0.2) is added to the value. For rule parameters with a bounded domain, this value is clipped to remain in-bounds. After selecting new values, the membership function parameters must be resorted.

E. Coevolutionary Learning

Our coevolutionary algorithm is outlined in Algorithm I and the specific parameters we used are given in Table VIII. We begin by creating a population of different strategies, all based on mutations of our original hand-built strategy. We also include the original unmodified strategy in the initial population. As the algorithm progresses, we compare the original strategy to the best strategy discovered in order to gauge how much learning has occurred.

Each generation of the algorithm starts by evaluating the fitness of each individual. A coevolutionary algorithm computes a relative fitness for each individual based on how it compares to other individuals. One strategy commonly used is to select a random sample from the population and evaluate the relative performance [6]. We select 10 opponents randomly with replacement from the population and play 20 games with each one, alternating as the first and second player. This results in 200 games of Geister being played for each fitness evaluation. We count the number of wins, losses, and draws for a strategy (a game is considered a draw if neither player has won after 100 moves), and compute the fitness as

$$fitness = 2*wins - 2*losses - draws. \quad (1)$$

It should be noted that due to the random nature of the fitness evaluation, a strategy's fitness can change between generations. This makes it difficult to keep track of the best solution found during the evolutionary learning process.

TABLE VIII
COEVOLUTIONARY ALGORITHM PARAMETERS

Parameter	*Value*	*Description*
popSize	20	Population size
numOpponents	10	Number of opponents each strategy plays against to determine fitness
numGames	20	The number of games played between opponents
eliteSize	2	Number of elite solutions that automatically survive each generation
t	2	Tournament size for selection
crossoverProb	0.8	Crossover probability
mutationProb	0.1	Mutation probability
σ	0.2	Spread of Gaussian noise for mutation
restartRate	10	Frequency that new random strategies are added to the population
restartSize	10	Number of random strategies to add with each restart

Nevertheless, we use the concept of elitism and automatically copy the two best individuals into the next generation without modification. A common approach in coevolutionary algorithms is to use a "Hall of Fame" in which the best individuals of the population are saved in an elite set, which must compete with every other individual in the population during the fitness evaluation [7]. We use elitism in our algorithm, but not a hall of fame. Including this modification could improve the performance of our algorithm in future experiments.

After copying the elite individuals into the new population, the remainder of the new population is created by selecting parents and creating offspring. We use the computed fitness values with tournament selection and a tournament size of two. This gives a reasonable selective pressure that tends to pick better individuals for reproduction, but allows lower performing strategies to reproduce as well. Then, with a probability of 0.8 we perform crossover, resulting in two children. We perform mutation on the children with a probability of 0.1 and add the children into the new population.

To maintain diversity, we replace the lower half of the population with new random strategies every 10 generations. This helps restart the algorithm in new areas of the search space and keeps the population from converging to a single strategy. The algorithm can be run for as long as time will allow, after which the top strategies in the population are returned.

V. PERFORMANCE RESULTS

A. Ghosts Challenge

Our game-playing agent was entered into the "Ghosts Challenge 2013" competition organized by the IEEE Computational Intelligence Society. The competition was open to all student members of the IEEE CIS and encouraged the use of computational intelligence techniques for developing autonomous agents. Our agent was submitted under the team name "mutigers" and competed against seven other submitted agents.

In developing our agent for the competition, we were able to run the coevolutionary algorithm described in the previous

TABLE IX
FINAL RANKINGS OF THE IEEE CIS GHOST CHALLENGE 2013

Ranking	*Team*	*Points*	*Rounds Difference*	*Matches Won*	*Matches Lost*	*Matches Drawn*	*Rounds Won*	*Rounds Lost*	*Rounds Drawn*	*Game Types*[a]						
										WC	*WL*	*WE*	*LC*	*LL*	*LE*	*D*
1	BLISS	21	209	7	0	0	274	65	11	172	17	85	0	52	13	11
2	mutigers	18	207	6	1	0	274	67	9	8	9	257	39	16	12	9
3	Eliot_CS7750	15	116	5	2	0	232	116	2	0	4	228	38	3	75	2
4	Skynet	12	9	4	3	0	172	163	15	15	0	157	49	18	96	15
5	FightGhost	9	37	3	4	0	190	153	7	10	21	159	19	9	125	7
6	RST	6	-184	2	5	0	35	219	96	7	21	7	16	12	191	96
7	WAIYNE1	1	-196	0	6	1	21	217	112	2	19	0	20	3	194	112
8	tsengine	1	-198	0	6	1	25	223	102	0	25	0	33	3	187	102

[a]Game types lists the distribution of the seven different ways a game can end for each team. The types are: WC (win by capturing all good ghosts of the opponent); WL (win by losing all evil ghosts); WE (win by exiting from an opponent corner with a good ghost); LC (lose by having all good ghosts get eaten); LL (lose by capturing all evil ghosts of the opponent); LE (lose by allowing the opponent to exit from a home corner with a good ghost); and D (draw).

section for two days. This was done using Matlab code running in parallel on a six-core i7 machine at 3.4 GHz with Windows 7. In this time, our algorithm evolved 87 generations of strategies. The final submission was rewritten in Java, as this was required to interface with the competition server. We included the top three evolved strategies and our original hand-drafted strategy in our competition agent. Our agent begins a match by playing with the best evolved strategy. After losing twice in a row, a different strategy is selected at random from the other strategies.

The competition was organized as a round-robin tournament in which each team played a match of 50 games against every other team. In addition to the 100 move limit imposed per game, agents have only 10 seconds to make each move before the server enforces a random choice. The teams were awarded 3 points for winning the most games in a match, 1 point for tying, and 0 points for losing. In the case of a tie in points, the winner is decided by the difference between the number of rounds won and lost. The final rankings of the competition are given in Table IX.

Our team, mutigers, placed second in the competition, losing only to the top scoring team, BLISS. Upon inspection of the game logs, it was discovered that mutigers, along with most of the other successful teams, preferred to attempt victory by moving a good ghost off the board from an opponent exit. However, the winning team, BLISS, favored capturing the opponent's good ghosts. Our agent was not able to successfully defend our good ghosts from BLISS to prevent them from being captured.

The implementation details of the competing agents were not released at the time of this writing, so it is difficult to make definitive conclusions about the different strategies. However, the distributions of the game types for each agent give some indication of the strategies involved. Clearly our strategy prefers to move a good ghost to the exit over capturing other ghosts. This could be due to low confidence values in our nature estimation, or a bias in our fuzzy rule bases toward the "Exit" goal. This shows the importance of having a diverse set of strategies that can counter different opponent tactics.

B. Evolving a Strategy

Due to the time constraints of the competition, the strategies we employed came from just 87 generations of evolution. We allowed the algorithm to continue running to 350 generations over the course of 10 days to observe the effect of additional learning time. The maximum and mean fitness scores for each generation are plotted in Fig. 3. Overall, these plots are very noisy showing wide fluctuations in the fitness values between generations. This occurs partly because each fitness evaluation involves playing a set of games against a random selection of opponents, and is therefore not a stable measure. Additionally, as the algorithm progresses the population as a whole becomes more competitive. This makes it more difficult for the best strategies to continue winning and allows new strategies to take the top spot. The plots show a hint of a periodic trend as a good strategy rises with a high fitness score, and then creates variations of itself through reproduction, creating more challenging competition. The initial rise during the first 30 or so generations shows the greatest amount of improvement as the population moves away from strategies based solely on our hand-drafted approach. The mean fitness value of the population stays roughly the same as the improvements learned through coevolution are balanced by the randomly introduced strategies every 10 generations, maintaining a diverse population.

To verify that the strategies in the population are improving, we play a set of games with the top strategy from a given generation against the entire population of another generation. The results of playing a match of 50 games with each of the 20 strategies in a population every 50 generations is given in Fig. 4. The colors in the matrix indicate the fitness scores obtained using Equation 1. Each row in the matrix represents the performance of the best strategy from a generation. The columns represent which generation's population the strategy had to play against.

As expected, the lower rows of the matrix are lighter, indicating higher fitness scores for the best strategies of later generations. Our expectation is that the lower-left corner should be light, indicating that the best strategy of the latest generation performed very well against the original population. Likewise, the upper-right corner should be dark to indicate that the original strategy did poorly against the most evolved population. In general this is true, but these are not the lightest or darkest cells in the matrix. We see a peak in the maximum fitness plot of Fig. 3 around 100 generations and this correlates to a light row in the matrix of Fig. 4. This was a rather good strategy, as it performed reasonably well against the other populations. We also see a dark column in

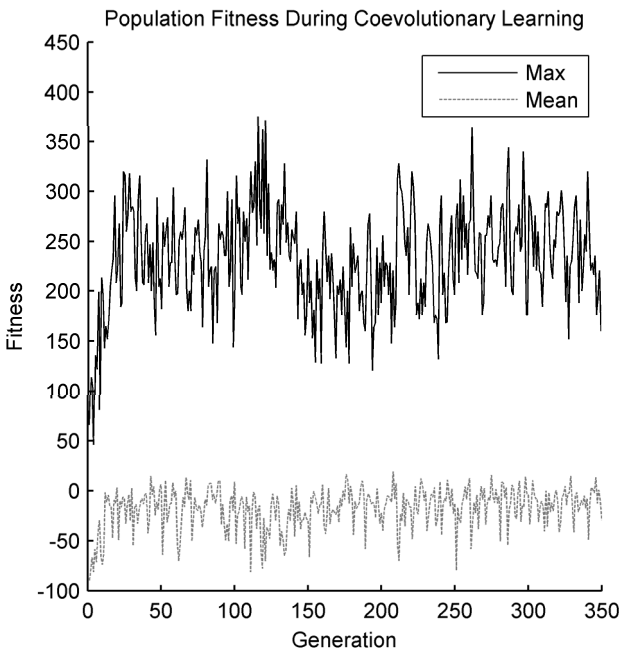

Fig. 3. Population fitness during coevolutionary learning. The fitness of each individual in the population of 20 strategies is obtained by playing 20 games against 10 random individuals and computing a score based on Equation 1.

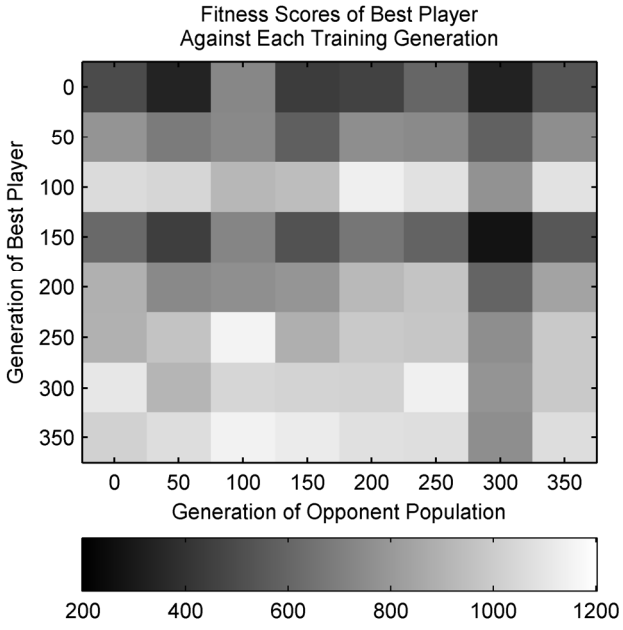

Fig. 4. Fitness scores of the best strategy every 50 generations playing against the entire population from every 50 generations of training. The best strategies from each generation played 50 games against all 20 individuals from the other generations. The fitness scores were computed using Equation 1.

the 300th generation indicating a weak population. These fluctuations in fitness seem to be common for coevolutionary algorithms with relative fitness measures.

VI. Conclusion and Future Work

Our intent with this project was to design a competitive agent for Geister using CI techniques that could be used as class projects. Although our method did not win the Ghost Challenge 2013 competition, our experiments demonstrate the applicability of coevolutionary learning using a fuzzy inference system and a neural network. Our goal-driven approach allows high-level strategies to be defined in terms of fuzzy rules and improved upon with evolutionary methods.

One improvement that would help our algorithm greatly is more training data. The neural network weights are learned initially from only our hand-drafted strategy. Multiple strategies from other sources, such as other teams in the competition or human players, would improve the quality of the training set. Additionally, our algorithm could be modified to update the weights of the neural network over the course of several games. This would allow it to adapt to various strategies on the fly and increase the accuracy of the ghost nature estimations.

Finally, our strategy could be improved by utilizing a look-ahead feature to determine which actions will be more beneficial in the future. This is done automatically by traditional game playing AI such as mini-max search or Monte Carlo Tree Search. A comparison with these methods could reveal additional strengths and weaknesses and result in a more globally competitive game playing agent.

References

[1] D. Whitehouse, E. J. Powley, and P. I. Cowling, "Determinization and information set Monte Carlo Tree Search for the card game Dou Di Zhu," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, 2011, pp. 87–94.

[2] F. Aiolli and C. E. Palazzi, "Enhancing artificial intelligence on a real mobile game," *Int. J. Comput. Games Technol.*, vol. 2009.

[3] S. Lucas and G. Kendall, "Evolutionary computation and games," *Comput. Intell. Mag.*, vol. 1, no. 1, pp. 10–18, Feb. 2006.

[4] K. O. Stanley and R. Miikkulainen, "Competitive coevolution through evolutionary complexification," *J. Artif. Intell. Res.(JAIR)*, vol. 21, pp. 63–100, 2004.

[5] F. Aiolli and C. Palazzi, "Enhancing artificial intelligence in games by learning the opponent's playing style," in *Proceedings of the IFIP-ECS Conference*, 2008, pp. 1–10.

[6] J. Reed, R. Toombs, and N. A. Barricelli, "Simulation of biological evolution and machine learning," *J. Theor. Biol.*, vol. 17, no. 3, pp. 319–342, 1967.

[7] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution.," *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, Jan. 1997.