Coevolving Intelligent Game Players in a Cultural Framework

Shiven Sharma, Ziad Kobti, Scott G. Goodwin

Abstract—Game playing has always provided an exciting avenue of research in Artificial Intelligence. Various methodologies and techniques have been developed to build intelligent game players. Coevolution has proven to be successful in learning how to play games with no prior game knowledge. In this paper we develop a coevolutionary system for the General Game Playing framework, where absolutely nothing is known about the game beforehand, and enhance it using Cultural Algorithms. In order to test the effectiveness of complementing coevolution with cultural algorithms, we play matches in several games between our player, a random player and one trained using standard coevolution.

I. INTRODUCTION

The design and construction of intelligent game playing agents has always been a fascinating area of Artificial Intelligence research. Techniques such as Reinforcement Learning and coevolution have been successfully applied to training game players in the absence of prior game knowledge. Successful players include TD-Gammon [1] for Backgammon and Blondie24 [2] for Checkers. TD-Gammon uses Temporal Difference (TD) Learning to train weights of a feed-forward neural network, which is used as a function approximator. Blondie24 coevolves a population of neural networks, with each network representing a single player. However, the construction of the learning architecture still relies on the description of various game elements, such as the number and types of pieces in a board game and the size and representation of the game state. For example, the neural network used to train Blondie24 cannot be used to train it to play Tic-Tac-Toe.

The framework we use for learning game knowledge is based on General Game Playing (GGP). Historically, game playing agents were designed to be good in specific games. Knowledge and heuristics were designed by experts in that game and programmed into these agents, or, as discussed above, learning architectures were designed specifically for particular games. However, even though these players excelled in games that they were designed for, they could not play any other games. GGP focuses on the creation of agents that are able to accept rules of a game, and use them to learn how to play it, eventually displaying a high level of competence in it. Given that the only details that are available are the basic rules on how to play the game, i.e. how to make a legal move, how to update states, how to check for a terminal state and how to decide who won, GGP is an interesting and challenging framework for testing and designing game playing systems.

In this paper we describe the construction of a game player that uses cultural evolution along with coevolution and TD-Learning to learn how to play general games. In our previous work [3], [4], we used coevolution to learn game strategies for General Games. Here we enhance that architecture by adding belief space knowledge shared by the entire population. The motivation is that knowledge in the belief space, which is modified by the strongest individuals in the population, will result in faster and higher quality learning as compared to using standard coevolution. The population model is based on the Ant Colony System (ACS) framework, and each individual in the population maintains its own local knowledge about the game which it learns using TD-Learning. Section 2 briefly describes the GGP competition and architecture. Section 3 provides a preliminary discussion on Reinforcement Learning, coevolution, Cultural Algorithms and Ant Colony Systems. In Section 4 we describe the architecture and various algorithms for our player. The results of matches in various games between our player and a random and standard, coevolutionary player are presented in Section 5. We discuss the results in Section 6 and provide conclusions and directions for future work in Section 7.

II. GENERAL GAME PLAYING

The annual General Game Playing Competition [5] organised by Stanford University has been instrumental in bringing about interest in GGP. The rules of the games are written in Game Description Language (GDL) [6]. Examples of game descriptions can be found at [7]. The tournaments are controlled by the Game Manager (GM) which relays the game information to each Game Player (GP) and checks for legality of moves and termination of the game. Successful players have mostly focused on automatically generating heuristics based on certain generic features identified in the game. Cluneplayer [8] was the winner of the first GGP competition, followed by Fluxplayer [9]. Both these players, along with U Texas Larg [10] use automatic feature extraction. Evaluation functions are created as a combination of these features and are updated in real-time to adapt to the game. CADIA-Player [11] was the first General Game Player to use a simulation based approach, using UCT [12] to search for solutions, and was the winner of the last GGP Competition. [13] also explored a Monte-Carlo approach in which random simulations were generated and the move with the highest win rate was selected. To improve the nature of these simulations, patterns in the sequences were extracted and used to generate new sequences. [14] have discussed a
coevolutionary approach using NEAT [15], an algorithm for automatically evolving neural networks using an evolutionary approach, for GGP.

III. PRELIMINARIES

In this section we provide a brief introduction to Reinforcement Learning (RL) and Temporal Difference Learning. Coevolution, Cultural Algorithms and Ant Colony Optimisation is given.

A. Reinforcement Learning: Temporal Difference Learning

Reinforcement Learning (RL) [16] constitutes a general class of learning techniques well suited for learning via interaction with the environment. A key feature of RL is the policy \( \pi \), which tells the agent which action to take from a given state. \( \pi \) can be either stochastic or deterministic. Upon taking an action, the agent receives an immediate reward \( r \). The goal of the agent is to maximise the cumulative reward it receives, starting from a state \( s_0 \). In order to model this goal, value functions are associated with states, denoted by \( V(s) \) and state-action pairs, denoted by \( Q(s,a) \). These value functions are associated with a policy \( \pi \), and are therefore commonly written as \( V^\pi(s) \) and \( Q^\pi(s,a) \). The functions represent the total reward (discounted or undiscounted, depending on the problem formulation) the agent can obtain from state \( s \) and following policy \( \pi \). RL aims at learning \( \pi^* \), an optimal policy, that will give the agent the maximum reward.

Temporal Difference (TD) Learning algorithms are a family of RL algorithms that learn through errors in the value functions at each temporal step in the state sequence. The simplest algorithm is TD(0), which updates the value function for a state \( s \) using the error (difference) between the value function of \( s \), \( V(s) \), and the value function of the successor state \( s' \), \( V(s') \). This update is shown as

\[
V(s) \leftarrow V(s) + \alpha \left[ r + \gamma V(s') - V(s) \right] \tag{1}
\]

\( \alpha \) is called the learning rate, and usually decreases over time. \( r \) is the reward received after transitioning from \( s \) to \( s' \). The update can be thought of as moving the previous value function for \( s \) towards the value given by the target. The update shown in Equation (1) is used to learn state value functions. Algorithms such as SARSA [16] and Q-Learning [17] are used to learn action-value functions.

One of the most famous applications of TD Learning to games is TD-Gammon [1], a backgammon player, which uses TD(\( \lambda \)), a TD Learning algorithm. Neural Networks (NN) are used to approximate the value functions of states, with each node in the NN corresponding to a single parameter. Using a number of simulated self-play games, TD gammon was able to reach the level of grandmasters in backgammon.

B. Coevolution

The principle of coevolution\(^1\) can be best explained with the help of an example. Consider the complex relationships that exist between herbivores and plants. To prevent themselves from being eaten, plants evolve to develop defence mechanisms such as toxic leaves, thick foliage, size and thorns. To overcome this, herbivores evolve, for example, long tongues and thick lips to overcome the foliage and thorns and special dietary habits such as eating clay to neutralise the toxins in the leaves. Both plants and herbivores in this case represent competing populations, each one trying to get an edge over the other. This is the principle of coevolution; using random variation and selection to evolve and learn strategies that will enable individuals to gain this edge that they need to survive.

A successful example of coevolution is the checkers player Blondie24 [2]. Each player is represented as a neural network which accepts as input a vector representing a checkers board position and outputs a number in the range \([-1, 1]\) to indicate its estimation of a moves’ quality. A population of such players play against each other, alternating between roles (red or black). The top 15 individuals are selected to spawn a new generation. The best evolved network at the end of this coevolutionary process is selected to play the game against opponents. At www.zone.com, a free checkers game website, Blondie24 was ranked amongst the top 500 of the 120,000 registered players. [18] and [19] also discuss the use of coevolution in evolving strategies for games such as checkers and tic-tac-toe. The population model uses Particle Swarms. The particles are vectors of weights of a neural network, and it is these weights that are trained using coevolution and Particle Swarm Optimisation.

C. Cultural Algorithms

Cultural Algorithms simulate cultural evolution, bringing about a more comprehensive learning and evolution than simple biological evolution. They can be used in both static and dynamic environments, and in complex multi-agent systems to provide effective simulations of learning procedures [20]. Evolution takes place at both the cultural level (the belief space) and the population level (for each individual). The belief space is the knowledge that is shared amongst the agents in the population. This model of dual-inheritance is the key feature of Cultural Algorithms, as it allows for a two-way system of learning and adaptation to take place. In a dual-inheritance system, the fit population members, as selected by an acceptance function on the basis of a fitness value, add their knowledge and experience to the belief space, thereby sharing it with all other agents in the environment. The belief space knowledge in turn helps guide the agents in the population by means of an influence function. Other evolutionary approaches, such as Genetic Algorithms, allow for evolution to take place only at the individual (or population) level, i.e., they do not support a dual-inheritance system.

To get an idea of how cultural evolution can complement coevolution, consider the following example: assume a large herd of wilderbeests are constantly being eaten by lions. Soon, a few them come up with a way to evade this unfortunate fate. In normal coevolution, this information

\(^1\)We consider competitive coevolution here.
would not be shared by other members of the herd. Instead, future generations would most likely inherit it, and so the knowledge would be passed on. However, when cultural evolution comes into play, this knowledge of evasion is put into the herds’ "belief space", and thus can be shared by other wilderbeasts. It is the same as the clever wilderbeasts going to their herd members and passing this knowledge on to them. From a gaming perspective, players that are come up with strong strategies are able to share them with other players, thereby allowing for faster learning and the emergence of stronger strategies amongst the weaker players.

D. Ant Colony Optimisation Algorithms

Ant Colony Optimisation Algorithms (ACO) were developed by [21]. They take inspiration from the behaviour of ants in nature. In nature, ants wander randomly, searching for food. Once they have found food, they return to their colony while laying down pheromone trails. These act as a guide for other ants in the future. When other ants find such a path, instead of wandering around randomly, they are more likely to follow the trail and further reinforce it by their pheromone deposits if they are successful. Since pheromone evaporates over time, shorter paths are more likely to have a stronger concentration of deposits. As a consequence, over time, short paths get favoured by more and more ants. This approach is applied in computer science to solve optimisation and path finding problems, such as in [22], using multiple agents (the ants) that move around in the problem space in search for the desired solutions.

Two key parameters that determine the state transitions are the desirability (or attractiveness), $\eta_{ij}$, and the pheromone level, $\tau_{ij}$ of the path (or arc) between the two states $i$ and $j$. $\eta_{ij}$ is usually represented by a predefined heuristic, and therefore indicates an a priori fitness of the path. On the other hand, $\tau_{ij}$ indicated the past success of the move, and therefore represents a posteriori fitness of the path. The update for $\tau_{ij}$ take place once all the ants have finished foraging. Given these two parameters, the probability of selecting a path $p_{ij}$ between states $i$ and $j$ is given by (2)

$$p_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{k \in M} (\tau_{ik})^{\alpha}(\eta_{ik})^{\beta}}$$

(2)

$\alpha$ and $\beta$ are user-defined parameters that determine how much influence should be given to the trail strength and desirability respectively. $M$ is the set of all legal moves that can be made from state $i$.

Once all the ants have finished foraging through the state space, the trails are updated as

$$\tau_{ij}(t) = \rho\tau_{ij}(t-1) + \Delta\tau_{ij}$$

(3)

$\Delta\tau_{ij}$ is the cumulative accumulation of pheromone by each ant that has passed between $i$ and $j$ and $t$ represents the time step. $\rho$ is called the evaporation parameter, and determines by what value the previous trail level decreases. This gradual evaporation prevents the ants from converging to a locally optimal solution. More resources on Ant Colony Optimisation can be found at [23].

IV. GAME PLAYER ARCHITECTURE

In this section we describe the construction of our game player. We begin with describing the population model of the various individuals in the coevolutionary setup and the way each individual represents and learns its local knowledge using TD-Learning. We then explain the cultural setup and how the belief space knowledge is represented and used by the players and influenced by select fit individuals.

A. Population model

We represent the population using the Ant Colony System model. This is similar to the model we used in our previous work, [4], [3], which used standard coevolution without a cultural framework. Each game playing agent is modelled as an ant, which can deposit pheromone along the game landscape. Details of this are explained in subsequent sections. With sufficient randomisation in sampling, the ants are able to explore different game sequences and appropriately learn, via pheromone deposits, the corresponding paths.

The population is split into two competing species, each of which represents a single role of the game. For example, in chess, one species would correspond to the role of white (i.e. moving white pieces) and the other species would be for black. Each ant has its own local knowledge which it uses, in conjunction with the cultural knowledge and the pheromone deposits, to make a move. The local knowledge is not shared as such with other ants, and allows the ant to maintain knowledge regarding its own experiences in playing the game. Each ant plays the game against a random sample of ants from the opponent species. The outcomes of these games are used to assign a fitness to each player.

The fitness each player gets for each opponent is based on how many players in its own species were able to beat that opponent. This allows players that are able to beat opponents that few others in their own species could beat to have a higher fitness. In other words, if $P_o$ is the set of all players were able to beat opponent $o$, then the fitness for a player $p \in P_o$ is given as

$$f(p) = \frac{1}{\sum_{o \in O} P_o}$$

(4)

where $O$ is the set of all opponents that the players faced. Based on this fitness, players are selected to influence the belief space and mate to create new offspring.

B. Local Knowledge Representation and Learning

We now consider the problem of representing the states in each players’ local knowledge. Since the number of states in most games is extremely large, using a table with an entry for a value function of each state is impractical. Therefore, we approximate the state representations by using features to represent each state. In the context of the game descriptions

---

2In this paper, we will use the term ant and player interchangeably.

3Other population models are also possible. For example, [18], [19] use a PSO-based approach for representing the population. Future work will investigate the effects of different population representations.
given in GDL, this can be done as follows. States in GDL are represented as a set of ordered tuples, each of which specifies a certain feature of the state. For example, in Tic-Tac-Toe, mark(1,1,X) specifies that the cell in row 1 and column 1 is marked with an X. Therefore, a state in Tic-tac-Toe is represented as a set of 9 such tuples, each specifying whether a cell is blank or contains an X or an O. Figure 1 gives an example of a state in Tic-Tac-Toe and the corresponding features associated with it. Note that the elements of the feature vector in Figure 1 are represented as strings for clarity. In reality, each element $\theta$ of the vector can be viewed as a 2-tuple $\langle \varsigma, \upsilon \rangle$, consisting of the string $\varsigma$ representing the feature and its corresponding value $\upsilon \in \mathbb{R}$. From now on, whenever we talk about features, whether we are referring to the string or the value will be clear depending on the context. In GGP, the lexicon of the game can change, but the underlying logic remains the same. Consequently, the learning that occurs is not based on the actual strings but the values that are learnt for them. In Section 7, we discuss this aspect in more detail.

These feature values are used to give an approximation of the value of the state which contains the respective features. For example, given a state $s$ represented by a feature vector $\vec{\theta}$, the value of the state is given as

$$V(s) = \sigma \left( \sum_{\upsilon \in \vec{\theta}} \upsilon \right)$$  \hspace{1cm} (5)

where the sigmoid function, a special form of the logistic function, is defined as

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$  \hspace{1cm} (6)

The sigmoid function squashes the value of the summation to be between 0 and 1. As a result, it becomes natural to consider the value of a state as the probability of winning from that state. Figure 2 illustrates how the knowledge is represented and used.

**C. Temporal Segmentation and Learning of Features**

Consider the effect of using a single set of all the features for the entire state space. Since a single feature may be shared by many states, a change in the value of a single feature affects the value of all the states that share that feature. In most cases, the change improves the value of some states (a positive effect), while degrading the value of other states (a negative effect). Given the linear representation, it is impossible for all states to be classified accurately by the features. The best we can do is try to minimise the negative effects of changes in feature values. This is done by not using a single set of features for the entire state space, but using a set of features to represent states at a unique temporal level. Each temporal level in the context of games is a turn the player is in. Figure 3 illustrates this idea. A temporal level is associated with each level of the afterstates in the game tree. Also, given this temporal representation, TD(0) learning can be delayed until the entire game is played, since the changes in feature values using TD(0) in preceding temporal levels has no effect on the values of states given by features at future levels. TD(0) learning is used to learn the values for each feature. Given a state $s_l$ at temporal level $l$ and a state $s_{l'}$ at the next temporal level $l'$, the update for feature value $\upsilon_s$ of each feature $\theta_s$ present in $s$ is done as shown in Equation (7). Note that $\gamma$ is set to 1 and $r$ is defined as 0 for each step, except at the final time step when it is equal to the final outcome of the game. $V(s_{l'})$ becomes 0 if $s_{l'}$ a terminal state.
state. \(|s_1|\) is the number of features in \(|s_1|\).

\[
\delta = \frac{r + V(s_t) - V(s)}{|s_1|}
\]

(7)

\[
v_s = v_s + \alpha \delta
\]

D. Calculations for Pheromone and Desirability

The ACO population model for GGP (with pheromone and desirabilities) specifies a means for player communication, providing a way for players to communicate with each other regarding previously seen paths through pheromone deposits and desirabilities. In the GGP case, pheromone for a path (or move) \(m\), \(\tau_m\), is represented as the average score attained through \(m\). Pheromone is not just associated with a move, but with all the features in the afterstate resulting from that move. The overall calculation of the pheromone deposit for both features and move after a series of forages (plays) has been made is shown in Equation (8)

\[
\tau_m = \frac{\sum_{a \in Ant_m} \chi_s}{N_m}
\]

(8)

\[
\tau_\theta = \frac{\theta_{L,s}}{N_\theta}
\]

\(Ant_m\) is the set of all ants \(a\) that went foraging and made move \(m\). \(\chi_s\) is the final score associated with each game sequence that includes \(m\). \(N_m\) and \(N_\theta\) are the number of times during a forage the move and feature were seen. \(\theta_{L,s}\) is the set of features in state (more specifically, the afterstate) \(s\) at temporal level \(L\). Pheromone evaporation follows the formula in Equation (3).

The desirability of a move \(m\) and feature \(\theta\) is simply the historic average score. It is similar to the way pheromone is represented, but while the pheromone is calculated as the average score per forage set, the desirability is the average score accumulated throughout the learning. In order to calculate the pheromone and desirability during action selection for learning, the average of the pheromone and desirability values for the action and the features of the resulting afterstate is taken.

E. Representing the Belief Space

Finally, we consider the task of representing the belief space. The belief space basically contains a number of afterstates (states reached by a player after making a move) that have been part of the game sequences of the fittest players. Each of these afterstates has a belief value associated with it which depends on the average score of each player chosen to influence the belief space, that has used the afterstate, and its own historic average score. Consider that a number of players \(p \in P\) have used afterstate \(S\), each having an average score of \(a_p\). Let \(A_S\) be the historic average score associated with \(S\). Then, the belief of \(S\) is given as

\[
Belief(S) = A_S + |P| \prod_{p \in P} a_p
\]

(9)

Note that the set of players \(P\) are all players that were accepted to modify the belief space, i.e. the fittest individuals of each generation. The historic score is updated by both these players and by other members who use these states during the course of game play. Intuitively, afterstates that have performed well by leading players to a win in the population, and that have had a large number of the fittest players influence them, have higher belief values. Algorithm (1) provides an overview of belief space creation. \(gseq\) is the set of all game sequences (i.e. sequence of afterstates) played by the player up till the point of updating the belief space. It is reset to the empty list after updating.

Algorithm 1 Belief Space Creation

1: \(P ← \text{top } c\% \text{ fit players}\)
2: \(\text{for all } p \in P \text{ do}\)
3: \(\text{for all GameSequence } gseq \in \text{GameSequences do}\)
4: \(\text{for all GameState } gstate \in gseq \text{ do}\)
5: \(\text{if } gstate \text{ not in beliefSapce then}\)
6: \(\text{addToBeliefSpace}(p, gstate, gseq.outcome)\)
7: \(\text{else if } gstate \text{ in beliefSapce then}\)
8: \(\text{updateState}(p, gstate, gseq.outcome)\)
9: \(\text{end if}\)
10: \(\text{end for}\)
11: \(\text{end for}\)
12: \(\text{end for}\)

Since it is impossible to store all possible states, a limit is placed on how many states can be present in the belief space. If that limit is crossed, a fixed \(r\%\) of the lowest states, ranked by Equation (9), are removed and new ones are added.

F. Putting it all together

Figure 4 shows the entire architecture of the game playing system for a single role (in a 2-player game, an identical system exists for the opponent). \(P\) are the various players, and \(S\) are the afterstates. The players play games against players from the competing species, and use TD(0) learning to update their feature values during each game. Once all players have finished playing these matches (i.e. after a single forage), pheromone and desirabilities are updated for the features and moves seen during these games. After a fixed number of forages, the fittest individuals so far are selected to mate and update the belief space. Algorithm (2) details the way the various players are controlled. \(c\) is the number of forages completed before mating and updating the belief space, and \(opponentSet\) is a randomly selected set of a pre-determined number of opponents. Algorithm (3) details how an ant (player) plays a game. Note that a random move is made based on a fixed probability for the opponent in order to ensure sufficient exploration of the game state space. Mating is done by selecting a crossover point between the feature vectors of the two parent players, and creating a new offspring by standard crossover operations using the same point at each temporal level. The new player then has part
Fig. 4. The architecture for a single role using coevolution and cultural evolution.

of the feature vectors from its father and part from its mother at each temporal level.

Moves are made using $\epsilon$-greedy selection. With probability $\epsilon$, a random move is selected. Otherwise, the move which maximises $V(s) \times \tau(s, m) \times \eta(s, m)$ is selected, where $V(s)$, $\tau(s, m)$ and $\eta(s, m)$ are the state value, pheromone and desirability values respectively for afterstate $s$ reached by making move $m$. However, if an afterstate exists in the belief space that matches any of the afterstates resulting by making move $m$, then the move resulting in the afterstate with the highest belief value is chosen instead.

Algorithm 2 Controller

1: while numberOfForages $\leq$ totalForages do
2:   for all Player ant $\in$ Ants do
3:     playGame(ant, opponentSet)
4:   end for
5:   update pheromone and desirabilities
6:   if forageCount $\%$ k = 0 then
7:     mate()
8:     updateBeliefSpace()
9:   end if
10: end while

V. EXPERIMENTS

We played 1000 matches of several games between a random player, and a player trained using coevolution and cultural evolution and one using only coevolution. For both coevolutionary players, the strongest individuals from the respective species are selected to play the game. To compare the effectiveness, we also present results of 2500 matches of the same games against two uniform random players. The games played are standard 3-by-3 Tic-Tac-Toe (3 T-T-T), large 5-by-5 Tic-Tac-Toe (5 T-T-T), Connect-4, Breakthrough and Checkers. The results are divided by role, with knowledge being used by each player. Player 1 refers to the player who makes the first move at the start of the game.

The players were all written in Java. The game rules in GDL are converted to Prolog, and YProlog [24], a Prolog inference engine in Java, is used. The learning rate was set to 0.99, and was decreased by a factor of 0.01 after each forage. The pheromone and desirability influence factors were set to 0.6 and 0.8 respectively. Opponent query probability was 0.5. 25% of the strongest players were selected for updating the belief space and mating. All in all, 20 players per species were created, which went collectively into 100 forages. Mating and belief space updates were done after every 3 forage runs, and the maximum number of states permitted in the belief space was limited to 100, with 25% of the weakest states being removed whenever the limit was crossed.

Table I shows the results of 2500 games when both players play randomly in order to give an idea of the distribution of results in a random setting.

Now we consider the results when only coevolutionary knowledge is being used. Table II shows the results when

Algorithm 3 playGame(Ant, opponentSet)

1: allGameSequences $\leftarrow$ empty list
2: for all opp $\in$ opponentSet do
3:   currentState $\leftarrow$ current state of the game
4: gameSequence $\leftarrow$ empty list
5: while terminal state of game is not reached do
6:   if turn of opp to make move then
7:     make random move $m$ or use knowledge
8:   else if turn of Ant then
9:     select move $m$ using knowledge
10: end if
11:   currentState $\leftarrow$ updateState(currentState, $m$)
12:   gameSequence.add(currentState, $m$)
13: end while
14: allGameSequences.add(gameSequence)
15: Perform TD(0) update on gameSequence
16: end for

<table>
<thead>
<tr>
<th>Games</th>
<th>Player 1</th>
<th></th>
<th>Player 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Win</td>
<td>Loss</td>
<td>Draw</td>
<td>Win</td>
</tr>
<tr>
<td>3 T-T-T</td>
<td>57.84</td>
<td>30.28</td>
<td>11.88</td>
<td>30.28</td>
</tr>
<tr>
<td>5 T-T-T</td>
<td>23.44</td>
<td>15.48</td>
<td>61.08</td>
<td>15.48</td>
</tr>
<tr>
<td>Connect-4</td>
<td>54.7</td>
<td>45.3</td>
<td>0</td>
<td>45.3</td>
</tr>
<tr>
<td>Breakthrough</td>
<td>51.3</td>
<td>48.7</td>
<td>0</td>
<td>48.7</td>
</tr>
<tr>
<td>Checkers</td>
<td>41.8</td>
<td>41.6</td>
<td>16.6</td>
<td>41.6</td>
</tr>
</tbody>
</table>
Player 1 uses knowledge and Player 2 plays randomly. Table III shows the results when Player 2 uses knowledge and Player 1 plays randomly. The results are expressed as a percentage. Players using the knowledge select moves greedily.

Consider now the results when coevolutionary learning is complemented by cultural evolution. Table IV shows the results when Player 1 uses knowledge and Player 2 plays randomly. Table V shows the results when Player 2 uses knowledge and Player 1 plays randomly.

Finally, we play a single match between the standard coevolutionary player (CoEv) and our player (Cul-CoEv). We only play one match, as moves are selected greedily and the games are therefore deterministic. As a result, the outcomes will always be the same as for these single matches. Table VI shows the outcomes for Cul-CoEv when it takes the roles of Player 1 and Player 2.

As can be seen, the player with knowledge augmented by cultural evolution outperforms the player with knowledge learnt with standard coevolution. More discussion on these results and directions for future work are discussed in the next sections.

VI. Discussion

As seen in the results in the preceding section, when coevolution is complemented with cultural evolution, performance of the player increases. Cultural evolution allows the strongest players to share their beliefs about the game with other existing players and future players. The belief space knowledge is represented as a set of afterstates. These are states that have been seen by most of the strongest players and have led to wins in the majority of matches. Therefore, other players can use these states to make moves, as these states are "tried and tested" by the strongest players. Such knowledge sharing speeds up learning and produces more accurate sampling of game sequences, which in turn trains the feature values in the local knowledge via TD(0) learning more accurately.

Having established that the knowledge does indeed work, it is important to explore ways in which it can complement other existing techniques for GGP. One such approach, which we ourselves use, is the UCT (Upper Confidence bound applied to Trees) algorithm [12]. UCT, which is inspired from the UCB algorithm [25], is a simulation-based algorithm which explores trees in an asymmetric manner using Monte-Carlo sampling. It has proven to be extremely effective for games, as the current world computer Go champion Mo-Go [26] and the winner of the last GGP competition CADIA-Player [11], both use UCT in their respective frameworks. CADIA-Player actually uses basic knowledge in the form of a move-history heuristic [27] in its simulations. We have explored how to use game knowledge with UCT in [4], [28] with some success. Future work will involve exploring how to effectively use this knowledge with UCT search.

VII. Conclusions and Future Work

In this paper, we presented an approach to learn game knowledge using coevolution complemented by cultural evolution. TD(0) learning along with ACO algorithms are used...
to learn value functions for states. The fittest players are selected from each species to update the belief space knowledge, guides players in the same and future generations. The results of the matches against a random player show a significant increase in performance of the player using the the coevolutionary cultural knowledge as compared to using standard coevolution, given the same number of individuals and forages. This is attributed to the fact that unlike in standard coevolution, cultural coevolution also allows knowledge learnt by the strongest players to influence, via the belief space, all the players.

Two major assumptions are made in our experiments. One is that the game descriptions use the same lexicon during game play. In the Stanford GGP competition, the lexicon is changed. We have developed a mechanism to recognise a game based on the underlying game logic, independent of the lexicon used to describe the rules, and are currently testing it thoroughly. Another assumption is that sufficient time is given to generate the knowledge. However, simulation-based approaches, like the one used to generate knowledge in our approach, are easy to parallelise. [11] was parallelised during the final GGP matches in the last competition, and won the tournament. Therefore, another important direction in our work will involve looking into ways to parallelise the knowledge generation algorithm. Time taken during training can also be significantly reduced by using hashing for feature and state retrieval.

An important direction in our future work is to explore different function approximation techniques. Basic linear approximation is limited in its ability to accurately classify a large number of different states, as discussed in Section 4. An alternative to the linear representation presented in this paper is to use a non-linear function approximator, such as neural networks, and use neuro-evolution to learn an appropriate network structure for the game. Another approach is to use CMAC (tile coding) [29], [16], which is a linear approximation technique that groups the state space into various tiles. The width and shape of the tiles are used to control the level of generalisation in approximation. Regarding population models, it is possible to represent the population using a different model, such as a PSO-based model or an Evolutionary Programming model. Future work will investigate the effects of different population representations, and also explore different ways of representing the belief space knowledge. Finally, as mentioned in the preceding section, effectively utilising this knowledge with our UCT player is also an important area to look into.

ACKNOWLEDGMENT

This work was funded in part by an NSERC Discovery Grant.

REFERENCES