Learning and Knowledge Generation in General Games

Shiven Sharma, Ziad Kobti, Scott Goodwin

Abstract—General Game Playing (GGP) aims at developing game playing agents that are able to play a variety of games and in the absence of game specific knowledge, become proficient players. Most GGP players have used standard tree-search techniques enhanced by automatic heuristic learning. In this paper we explore knowledge representation and learning in GGP using Reinforcement Learning and Ant Colony Algorithms. Knowledge is created by simulating random games. We test the quality of the knowledge by comparing the performance of players using the knowledge in a variety of games. The ideas presented in this paper provide the potential for a framework for learning and knowledge representation, given the total absence of any prior knowledge.

I. INTRODUCTION

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. However, even though these players excelled in games that they were designed for, they could not play any other games. General Game Playing (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. This approach to game playing opens up many areas of challenging research, since the design of a successful game player must take into account aspects such as learning, knowledge representation, reasoning and pattern recognition.

A. Early attempts at General Game Playing.

A class of games for which a GGP approach was taken were *positional games*, which were formalised by [1]. Some examples of positional games include Tic-Tac-Toe, Hex and the Shannon switching games. A positional game can be defined by three sets, P , A , B . Set P is a set of positions; with set A and B both containing subsets of P . In other words, sets A and B represent a collection of subsets of P , with each subset representing a specific positional situation of the game. The game is played with two players, with each player alternating in moves, which consist of choosing an element from P . The chosen element cannot be chosen again. The aim for the first player is to construct one of the sets belonging to A , whereas the aim for the second player is to construct one of the sets belonging to B . Programs that are capable of accepting rules of positional games and, with practice, learn how to play the game have been developed. [1] constructed a program that is able to learn important board configurations

in a 4 x 4 x 4 Tic-Tac-Toe game. This program plays about 12 times before it learns and is effectively able to play and start defeating opponents. A set of board configurations are described by means of a weighted graph.

B. Current approaches to General Game Playing.

The annual General Game Playing Competition [2] organised by Stanford University has been instrumental in bringing about renewed interest in GGP. The rules of the games are written in Game Description Language (GDL) [3], which is syntactically similar to prefix KIF [4]. As a consequence, most of the current research in GGP is based on the foundations laid down by the Stanford Group. 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. Communication between players and the GM takes place in the form of HTTP messages. A more detailed description of the architecture and game rules can be found at [5]. Successful players have mostly focused on automatically generating heuristics based on certain generic features identified in the game. Cluneplayer [6] was the winner of the first GGP competition, followed by Fluxplayer [7]. Both these players, along with UTexas Larg [8] 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. Another approach that has been taken is in [9], where transfer of knowledge extracted from one game to another is explored by means of a $TD(\lambda)$ based reinforcement learner. CADIA-Player [10] was the first General Game Player to use a simulation based approach, using UCT [11] to search for solutions, and was the winner of the last GGP Competition. [12] 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. [13] have discussed a co-evolutionary approach using NEAT [14], an algorithm for automatically evolving neural networks using an evolutionary approach, for GGP.

Though a number of players have been created, as discussed above, that are quite effective for General Games, to the best of our knowledge not much research has gone into exploring how to learn and create knowledge for General Games. An advantage of creating and storing knowledge is that when the same game is encountered again, knowledge already learnt can be used to give a better performance. The main aim of this paper is to explore learning and creation of such knowledge from just the game rules given in GDL. More specifically, in this paper we examine the learning

S. Sharma is a graduate student in the School of Computer Science at the University of Windsor, Windsor, ONT, Canada. Z. Kobti and S. Goodwin are with School of Computer Science at the University of Windsor, Windsor, ONT, Canada. (phone: +1-519-253-3000; fax: +1-519-973-7093; email: {sharmaw, kobti, sgoodwin}@uwindsor.ca).

and knowledge generation for General Two-Player turn-based Zero-sum games, which constitute a large part of GGP set. The ideas presented here can be extended to other types of games as well. Random simulations of game play are generated, and Temporal Difference (TD) Learning [15] is used to learn value functions for states. These values indicate the probability of winning from those states. Exploration in these simulations is done using techniques from Ant Colony Optimisation Algorithms [16]. Section 2 provides a short introduction to TD-Learning and ACO algorithms. Section 3 discusses the knowledge representation and creation algorithms. In Section 4 we compare the performance of players using the knowledge in a variety of games. Finally in Section 5, we discuss the conclusions and directions for future work.

II. PRELIMINARIES

In this section a brief introduction to Reinforcement Learning, Temporal Difference Learning and Ant Colony Optimisation Algorithms (ACO) is given.

A. Reinforcement Learning: Temporal Difference Learning

Reinforcement Learning (RL) [15] constitutes a general class of learning techniques well suited for learning via interaction with the environment. A key feature of RL is the *policy* π , which tells the agent which action to take from a given state. π 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 π , 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 π . In this paper however, we will omit the superscript π for simplicity. RL aims at learning π^* , 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

$$V(s) \leftarrow V(s) + \alpha [\underbrace{r + \gamma V(s')}_{\text{target for TD(0)}} - V(s)] \quad (1)$$

α 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 [15] and Q-Learning [17] are used to learn action-value functions.

Value functions are typically represented as tables, with one entry for each state or state-action pair. However, in

problems with large state spaces, this is not practical. In such cases, function approximation techniques are used with parameterised functional forms of states, using a parameter vector $\vec{\theta}$. State values are therefore calculated entirely from this vector, and changes are made to the parameters instead of each individual state. The representation of the parameter vector depends on the problem formulation.

One of the most famous applications of TD Learning to games is TD-Gammon [18], a backgammon player, which uses TD(λ), 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. Ant Colony Optimisation Algorithms

Ant Colony Optimisation Algorithms (ACO) were developed by [19]. 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 [20], using multiple agents (the ants) that move around in the problem space in search for the desired solutions.

We now describe the basic idea behind ACO algorithms. Two key parameters that determine the state transitions are the *desirability* (or attractiveness), η_{ij} , and the *pheromone level*, τ_{ij} of the path (or arc) between the two states i and j . η_{ij} is usually represented by a predefined heuristic, and therefore indicates an *a priori* fitness of the path. On the other hand, τ_{ij} indicated the past success of the move, and therefore represents a *posteriori* fitness of the path. The update for τ_{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)

$$p_{ij} = \frac{(\tau_{ij}^\alpha)(\eta_{ij}^\beta)}{\sum_{\kappa \in M} (\tau_{i\kappa}^\alpha)(\eta_{i\kappa}^\beta)} \quad (2)$$

α and β 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

$$\tau_{ij}(t) = \rho \tau_{ij}(t-1) + \Delta \tau_{ij} \quad (3)$$

$\Delta \tau_{ij}$ is the cumulative accumulation of pheromone by each ant that has passed between i and j and t represents the time

step. ρ 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, and also assists in exploration.

III. KNOWLEDGE REPRESENTATION AND LEARNING

For GGP, in the context of RL, the basic goal can be thought of as learning a policy (and consequently, a value function) that, given a state, returns the action (move) which gives us the greatest probability of winning. TD(0) learning is used to learn these values, and ACO ideas are used to strengthen paths between states so as to guide future simulations for learning. In our previous work, we explored using Monte-Carlo RL for training features (without dividing them temporally) along with historic values of moves [21] and using ACO algorithms for GGP by using states explicitly instead of features [22]. This paper builds upon the ideas and results seen from these approaches.

In this section we discuss in detail how learning takes place using simulations, and how knowledge is represented and created. We begin by first describing the knowledge representation scheme. Then we discuss the learning algorithms, and how the learnt knowledge can be used.

A. Knowledge Representation

We first consider the problem of representing the states. 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 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 given as 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

Fig. 1. Features in a state of Tic-Tac-Toe. The vector $\vec{\theta}$ represents the vector of features.

are represented as strings for clarity. In reality, each element θ of the vector can be viewed as a 2-tuple $\langle \varsigma, v \rangle$, consisting of the string ς representing the feature and its corresponding

value $v \in \mathbb{R}$. 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.

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

$$V(S) = \sigma \left(\sum_{v \in \vec{\theta}} v \right) \quad (4)$$

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

$$\sigma(t) = \frac{1}{1 + e^{-t}} \quad (5)$$

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. For example, with initial values of 0 for all features, the value of a state as a result of the operations described is 0.5, indicating an equal chance for a win or a loss. TD(0) learning is used to learn the values for each feature. Figure 2 illustrates how the knowledge is represented and used.

Fig. 2. Getting a state value from the knowledge. The values for each feature in the table are learnt from TD-Learning. They are added and squashed as described.

B. Temporal Segmentation of Features

We have seen how to represent the state space by using features from the game description. However, 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). Since the representation discussed above is linear, 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. The tree shown is an example of a game tree. The circular nodes are the states the player is in, and the square nodes are the afterstates, i.e. the states resulting after the player makes a move. A temporal level is associated with each level of the afterstates in the game tree. Using sets of features for each temporal level allows for the features to be associated with a smaller set of states, thereby minimising the negative effects of changes in feature values during TD(0) learning.

Fig. 3. Temporal segmentation of features. Circular nodes are nodes from which the player makes moves. Square nodes are the states that occur as a result of these moves (the afterstates).

C. Path Information

We now discuss how to use ideas from ACO algorithms to strengthen moves between states to guide simulations during learning. The ACO approach for GGP (with pheromone and desirabilities) specifies a model for agent communication, providing a way for players to communicate with each other regarding previously seen paths and their confidence in those paths.

As we have discussed before, ants transition from one state to the next based on the pheromone levels and desirabilities of the paths between the states. In the context of GGP, each ant is considered to be a player assigned a specific role (for example, black or white in Chess). Just as the features for states are associated with a particular temporal level, so are all the moves that have been made from that level. For example in Tic-Tac-Toe, for player X (player marking X), temporal level 1 will have 9 moves (one for each cell to be marked with X).

Let's see how to calculate the pheromone and desirabilities for each state. Pheromone levels in traditional ACO algorithms are represented as the reciprocal of the length of the path travelled. In the GGP case, pheromone for a path (or move) m , τ_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 (6)

$$\tau_m = \frac{\sum_{a \in Ant_m} \chi_s}{N_m} \quad (6)$$

$$\tau_\theta = \frac{\sum_{\theta \in \theta_{L,s}} \chi_s}{N_\theta}$$

Ant_m is the set of all ants a that went foraging and made move m . χ_s is the final score associated with each game sequence that includes m . N_m and N_θ 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 θ 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.

D. TD(0) Update

The final update to be considered is the update of value functions of states using TD(0) learning. The update of the value function is in essence the update of a set of features. 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 v_s of each feature θ_s present in s is done as shown in Equation (7). Note that γ 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. $|s_l|$ is the number of features in $|s_l|$.

$$\delta = \frac{r + V(s_{l'}) - V(s_l)}{|s_l|} \quad (7)$$

$$v_s = v_s + \alpha \delta$$

E. The Learning Algorithm

Now we present the algorithm for learning. The entire algorithm is given in Algorithm (1). Various implementation details regarding updates are omitted as they follow from the equations presented in the preceding subsections. State features and moves are added on-the-fly to the corresponding temporal level as and when they are observed during simulations.

Knowledge is created for both players. During simulations, moves are selected using ϵ - greedy selection. With probability $1 - \epsilon$, the move maximising $V(s) \times \tau_s^\alpha \times \eta_s^\beta$ is selected. With probability ϵ , moves are selected using Equation (2). This is expressed implicitly in line 12 of Algorithm (1). In line 10, moves are selected greedily. The probability of querying the opponents' knowledge base is controlled by a user defined variable. *Trails* in line 20 is

a tuple $\langle \text{afterstate}, \text{move} \rangle$, consisting of the move and the corresponding afterstate reached by making that move.

Algorithm 1 GGP-Knowledge

```

1: Initialise each  $Ant \in Ants$  with a unique role
2: Initialise KB as an empty set
3: Initialise  $allGameSequences$  as an empty list
4: while  $numberOfForages \leq totalForages$  do
5:   for all  $Ant \in Ants$  do
6:      $currentState \leftarrow$  current state of the game
7:      $gameSequence \leftarrow$  empty list
8:     while terminal state of game is not reached do
9:       if not turn of  $Ant.role$  then
10:        make random move  $m$  or consult KB for move  $m$ 
11:       else if turn of  $Ant.role$  then
12:        select move  $m \in legalMoveList$  using  $KB$ 
13:       end if
14:        $currentState \leftarrow updateState(currentState, m)$ 
15:        $gameSequence.add(currentState, m)$ 
16:     end while
17:      $allGameSequences.add(gameSequence)$ 
18:   Perform TD(0) update on  $gameSequence$ 
19: end for
20: for all Trails  $t \in allGameSequences$  do
21:   updatePheromone( $t$ )
22:   updateDesire( $t$ )
23: end for
24: end while

```

IV. EXPERIMENTAL RESULTS

In order to test the general quality of the knowledge, 1000 matches of several games are played between a player using the knowledge and a uniform random player (player makes moves randomly with a uniform distribution). 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, Checkers and Minichess. 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 [23], 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. 40 ants were created, which went collectively into 150 forages (for a total of 6000 simulations).

Table I shows the results of 2500 games when both players play randomly. A large number of games is used so as to get a fairly accurate distribution of outcomes between the two players. The results are expressed as a percentage.

TABLE I
RANDOM GAMES

Games	Player 1			Player 2		
	Win	Loss	Draw	Win	Loss	Draw
3 T-T-T	57.84	30.28	11.88	30.28	57.84	11.88
5 T-T-T	23.44	15.48	61.08	15.48	23.44	61.08
Connect-4	54.7	45.3	0	45.3	54.7	0
Breakthrough	51.3	48.7	0	48.7	51.3	0
Checkers	41.8	41.6	16.6	41.6	41.8	16.6
Minichess	4.36	95.64	0	95.64	4.36	0

TABLE II
PLAYER 1 USES KNOWLEDGE

Games	Player 1			Player 2		
	Win	Loss	Draw	Win	Loss	Draw
3 T-T-T	87.5	9.5	3	9.5	87.5	3
5 T-T-T	64.4	8.6	27	8.6	64.4	27
Connect-4	92.9	7.1	0	7.1	92.9	0
Breakthrough	99.3	0.7	0	0.7	99.3	0
Checkers	57.8	23.2	19	23.2	57.8	19
Minichess	100	0	0	0	100	0

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

As can be clearly seen, the knowledge results in a huge improvements in the playing abilities of the player using it. It is also interesting to note from Table I that learning can be biased for each player depending on which player makes the first move, as the percentage of winning sequences is not (relatively) evenly distributed for each player in all games. Perhaps the most interesting result is that in Minichess, where in a random case, player 1 loses quite badly compared to player 2, but outperforms player 2 when knowledge is being used (as compared to when player 2 uses knowledge).

It can be argued that in a competitive scenario, players are not going to be purely random. This is indeed true, but the main aim of our experiments is to test the quality of knowledge and consequently, its impact on game playing

TABLE III
PLAYER 2 USES KNOWLEDGE

Games	Player 1			Player 2		
	Win	Loss	Draw	Win	Loss	Draw
3 T-T-T	29.8	61.2	9	61.2	29.8	9
5 T-T-T	9.7	57.5	32.8	57.5	9.7	32.8
Connect-4	21	79	0	79	21	0
Breakthrough	2.9	97.1	0	97.1	2.9	0
Checkers	25.4	61.7	12.9	61.7	25.4	12.9
Minichess	2.6	97.4	0	97.4	2.6	0

performance. Playing against a standard, random player gives a basic, fairly accurate, comparison and test. However, it is also true that for a more thorough testing of the quality of the knowledge, games need to be played against a larger variety of players. Unfortunately, we did not have access to other GGP players that have been developed and deployed in the competitions. We hope to get in touch with the authors in the very near future and try to get access to them.

It is also important to note that in most cases, the knowledge cannot be simply used by greedily selecting moves that maximise the combination of the value functions, pheromone and desirabilities. 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 [11]. UCT, which is inspired from the UCB algorithm [24], 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 [25] and the winner of the last GGP competition CADIA-Player [10], both use UCT in their respective frameworks. CADIA-Player actually uses basic knowledge in the form of a move-history heuristic [26] in its simulations.

We attempted to use the knowledge generated as described in this paper with UCT. Knowledge in UCT can be applied in two parts, each of which corresponds to a phase in the UCT simulation. The first part in which it can be applied is when selecting nodes to descend to the next level. Typically, UCT selects all unvisited nodes at least once. Knowledge can be used to bias this selection. In the second part, upon reaching a node which has not been yet visited, a Monte-Carlo simulation is performed from that node onwards till the end (i.e. the terminal state), and a value is assigned to that node based on the outcome. Knowledge can again be used to guide the simulation, for example, by using an ϵ -greedy selection with a relatively high value of ϵ (to prevent the simulation for being too biased and deterministic). The results of these preliminary experiments are shown in Table IV. A total of 100 games are played, with roles equally distributed for using the knowledge. Matches are between a player using standard UCT, UCT_S , and a player using knowledge for simulations and node selection, UCT_K . Node selection for unvisited nodes is done relative to the values learnt for that node from the knowledge. The games used were those for which the state-space is relative large, as in smaller games the differences in performance were not high.

The results are promising, as there is a slight improvement in performance. However, we believe performance can indeed be improved further by exploring more efficient ways of using the knowledge with UCT.

Finally, we provide the size of knowledge files created to give an idea of the memory usage. Table V lists the sizes in kilobytes. The knowledge was represented as a Java Vector with one element for each role. Each element itself consisted

TABLE IV
PERCENTAGE OF WINS FOR UCT_K AGAINST UCT_S FOR 100 MATCHES
PER GAME

Game	Wins for UCT_K
Connect-4	58
Breakthrough	72
Checkers	56

TABLE V
KNOWLEDGE FILE SIZE

Game	File size (in Kilobytes)
3 T-T-T	23 KB
Minichess	101 KB
5 T-T-T	162 KB
Connect-4	411 KB
Breakthrough	2,130 KB
Checkers	4,011 KB

of a Vector of temporal levels, with each level having a Vector of features and moves.

V. CONCLUSIONS AND FUTURE WORK

This paper presented an answer to an interesting question: How to learn and generate an effective knowledge representation in an unknown domain? The unknown domain in this case is the set of General Two-player turn-based Zero-sum games. TD(0) learning along with ACO algorithms are used to learn value functions for states. These value functions represent the probability of winning a game from that state, and are used in conjunction with the pheromone and desirability values to direct game play. Value functions are represented using linear function approximation, with states being represented as a set of features. Feature sets are associated with each temporal level (i.e. a turn in the game) to allow for a more concise representation of states. The results of the matches against a random player show a significant increase in performance of the player using the learnt knowledge, and preliminary experiments with UCT also show promise. The approach presented here can easily be extended to other types of general games. An alternative to the linear representation presented in this paper is to use a non-linear function approximator, such as a neural network. This can potentially avoid the division of features into temporal levels. For example, TD-Gammon, which uses neural networks along with Reinforcement Learning, is a prime example of how non-linear function approximators can be used effectively in games with large state spaces. An obvious advantage of using knowledge is that it can use past experience to play a better game. It is also possible to keep learning and improving the knowledge during game play, and consequently improve performance. In a UCT player knowledge can be continually trained during the Monte-Carlo play out phase.

An important direction of future work will concern how to effectively utilise this knowledge with existing GGP player

architectures. An example of this was presented in the preceding section in which we used it along with a UCT player. It is possible to further improve performance by exploring ways in which it can be effectively combined with UCT. [27] observes that, for UCT-based Go, random simulations using pre-defined patterns outperform simulations using other types of knowledge. We are also looking into the automatic generation of state patterns to be used in conjunction with UCT search. For example, recognising forcing states, as described in [1], is one direction to look into.

Two major assumptions are made in our approach. 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. In the Stanford GGP competition, a fixed amount of time is given before start of play and to make each move, making knowledge generation prior to game play, especially for large games, difficult. However, simulation-based approaches, like the one used to generate knowledge in our approach, are easy to parallelise. [10] 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 retrieval.

Knowledge transfer in game playing is also an interesting direction to look into. If knowledge learnt in one game can be somehow used to boost performance in similar (but not identical) games, then that would be very useful. This has been explored in [9]. Ideally, a game player should have both game-specific and general-game knowledge. We hope to explore this avenue in the future.

GGP presents a multitude of challenges that spans disciplines such as Machine Learning, Knowledge Representation, Reasoning, Planning and Pattern Recognition. This paper looked into tackling the challenge of learning and representing knowledge. One can consider GGP to be a special case of the larger area of General Problem Solving (GPS). Though we are still far away from constructing a perfect General Problem Solver, research and exploration into GGP can provide some insights into larger, more daunting questions, especially those pertaining to GPS.

ACKNOWLEDGMENTS

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

REFERENCES

- [1] E. Koffman, "Learning through pattern recognition applied to a class of games," in *IEEE Trans. on Systems, Man and Cybernetics*, Vol SSC-4, 1968.
- [2] M. Genesereth and N. Love, "General game playing: Overview of the aai competition," *AI Magazine (Spring 2005)*, 2005.

- [3] M. Genesereth, and N. Love, "General game playing: Game description language specification," <http://games.stanford.edu>.
- [4] M. R. Genesereth and R. E. Fikes, "Knowledge interchange format, version 3.0 reference manual," Stanford University, Tech. Rep. Technical Report Logic-92-1.
- [5] <http://games.stanford.edu>.
- [6] J. Clune, "Heuristic evaluation functions for general game playing," in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, 2007.
- [7] S. Schiffler and M. Thielscher, "Fluxplayer: A successful general game player," in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, 2007, pp. 1191–1196.
- [8] B. Banerjee, G. Kuhlmann, and P. Stone, "Value function transfer for general game playing," in *ICML Workshop on Structural Knowledge Transfer for ML*, 2006.
- [9] B. Banerjee and P. Stone, "General game playing using knowledge transfer," in *The 20th International Joint Conference on Artificial Intelligence*, 2007, pp. 672–777.
- [10] Y. Bjornsson and F. H., "Simulation-based approach to general game playing," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. AAAI Press, 2008.
- [11] L. Kocsis and C. Szepesvari, "Bandit based monte-carlo planning," in *European Conference on Machine Learning (ECML)*, 2006, pp. 282–293.
- [12] S. Sharma and Z. Kobti, "A multi-agent architecture for general game playing," in *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, 2007.
- [13] J. Reisinger, E. Bahceci, I. Karpov, and R. Miikkulainen, "Coevolving strategies for general game playing," in *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, 2007.
- [14] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," pp. 99–127, 2002.
- [15] R. Sutton and A. Barto, *Reinforcement Learning, An Introduction*. MIT Press, 1998.
- [16] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization: Artificial ants as a computational intelligence technique," Universit Libre de Bruxelles, Tech. Rep. TR/IRIDIA/2006-023, September 2006.
- [17] C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge University, Cambridge, England, 1989.
- [18] G. J. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, pp. 58–68, 1995.
- [19] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Politecnico di Milano, Italy, 1992.
- [20] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the travelling salesman problem," *IEEE Transactions on Evolutionary Computation*, pp. 53–66, 1997.
- [21] S. Sharma, Z. Kobti, and S. Goodwin, "Knowledge generation for improving simulations in uct for general game playing," in *21st Australasian Joint Conference on Artificial Intelligence*, 2008, (in press).
- [22] S. Sharma, Z. Kobti, and S. Goodwin, "General game playing with ants," in *The Seventh International Conference on Simulated Evolution And Learning (SEAL'08)*, 2008, (in press).
- [23] M. Winikoff, <http://www3.cs.utwente.nl/~schooten/yprolog>.
- [24] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite time analysis of the multi-armed bandit problem," *Machine Learning*, vol. 47, no. 2/3, pp. 235–256, 2002.
- [25] S. Gelly and Y. Wang, "Modifications of uct and sequence-like simulations for monte-carlo go," in *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, 2007.
- [26] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pp. 1203–1212, 1989.
- [27] S. Gelly, "A contribution to reinforcement learning; application to computer-go," Ph.D. dissertation, University of Paris South, 2007.