A multi-agent system integrating Reinforcement Learning, Bidding and Genetic Algorithms

Dehu Qi
Lamar University
Computer Science Department
PO Box 10056
Beaumont, Texas, USA
dqi@cs.lamar.edu

Ron Sun
Rensselaer Polytechnic Institute
Cognitive Science Department
110 Eighth Street, Carnegie 302A
Troy, New York 12180
rsun@rpi.edu

November 12, 2003

Abstract

This paper presents a multi-agent reinforcement learning bidding approach (MARLBS) for performing multi-agent reinforcement learning. MARLBS integrates reinforcement learning, bidding and genetic algorithms. The general idea of our multi-agent systems is as follows: There are a number of individual agents in a team, each agent of the team has two modules: Q module and CQ module. Each agent can select actions to be performed at each step, which are done by the Q module. While the CQ module determines at each step whether the agent should continue or relinquish control. Once an agent relinquishes its control, a new agent is selected by bidding algorithms. We applied GA-based MARLBS to the Backgammon game. The experimental results show MARLBS can achieve a superior level of performance in game-playing, outperforming PubEval, while the system uses zero built-in knowledge.

1 Introduction

In the multi-agent systems, the individual agent has the power to act. The environment usually contains other agents. Multi-agent systems exist on many different levels on human experience, from social systems to living organisms. For example, human nervous system is a very complex multi-agent system. Each neuron in the brain can make decision based its own information, and

[1]Appeared in Web Intelligence and Agent Systems, p187-202, 1:(3-4), 2003

its decision may affect the performance of the whole system. Since none of neurons has access to the complete information, they must adapt to (such as communication, cooperation, co-learning, etc.) each other in order to properly coordinate their actions.

Multi-agent cooperation and co-learning is a difficult task. How a multi-agent system can be developed in which agents cooperate with each other to collectively accomplish complex tasks is a key issue in building multi-agent systems. For example, the game theoretic notion of a "coalition" has been applied in this area [12]. Although game theory provides analysis of (variously defined) states of equilibria in coalition formation that may be achieved by self-interested agents pursuing their own interests, it does not study sufficiently how these equilibria can be achieved computationally. The game theory makes the somewhat unrealistic assumption of completely rational agents that can examine all the aspects of a coalition (e.g., all the possible "objections" and "counter-objections" [25]), although limited rationality models have been investigated [16]. The problem could be NP-complete [6]. Alternatively, there are computationally simpler, more heuristic approaches toward forming cooperation structures, such as bidding [8] [2] [19], negotiation, and domain specific algorithms. However, they are, in general rudimentary and must assume some additional properties in agents (e.g., very "cooperative" agents [10] [18]).

In this paper, we will look into a GA-based multi-agent reinforcement learning approach with bidding that learns complex tasks. We integrate three mechanisms: reinforcement learning, bidding mechanisms and genetic algorithms. That is, the learning of individual agents and the learning of cooperation among agents is completely simultaneous and thus interacting. This approach extends existing work, in that it is not limited to bidding alone. For example, not just using bidding alone to form coalitions [10] or bidding alone as the sole means for learning (as in [2]). Neither is it a model of pure reinforcement learning [15] [4] [9] [11]. Furthermore, it is not a pure evolutionary system [7] [26]. It is the combination and the interaction of the three aspects: reinforcement learning, bidding and evolution.

2 The Algorithms and the Architecture

2.1 The MARLBS System

In our system, the multi-agent system (team) takes action based on environment information received. Each team is composed of several member agents. Each member receives all environment information and can take action based on it. In any given state, only one member of the team is in control. The action of the whole team is chosen by the member-in-control. In the next state, the member-in-control will decide to continue to control or relinquish control. If the member-in-control decides to give up control, the new member-in-control will be chosen from all other members in that team through the bidding process. The member who has the highest bid will be the new member-in-control. In other words, the member which will more likely benefit the whole will have more chances to be chosen as the member-in-control. A snapshot of a team with 5 members is shown in Figure 1.

The member agent learns to deal with its environment through the reinforce-

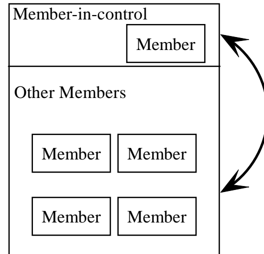

Figure 1: A snapshot of a team has 5 members. Only one member is in control in each state. The members communicate with each other through bidding.

ment learning. The member-in-control receives a reward from the environment based on its actions. During the control exchange process, the current member-in-control exchanges the reward with the next member-in-control. This is also a form of communication among members. Since all members have the same structure and have the same ability to receive the environmental information, our system is a homogeneous communicating system. Since we use the bidding algorithm in our system, the method of distributed control in our system is competitive.

We apply evolutionary algorithms for team evolution. We start our system from randomly initialized teams. After a number of episodes, we apply genetic algorithms or the genetic programming to these teams. With the help of genetic algorithms or the genetic programming, information not only is exchanged between members in a team, but also is exchanged between teams.

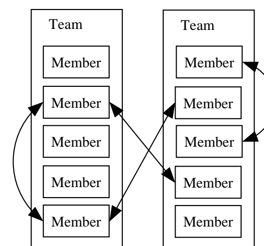

Figure 2: The teams communicate with each other through genetic algorithms or genetic programming.

2.2 The GMARLBS System

There are two methods, genetic algorithms and genetic programming, to use evolutionary algorithms to evolve the teams.

2.2.1 The Multi-Team Algorithm

The first method is a regular genetic algorithm. We train a batch of teams as the initial population. After a number of episodes, we apply crossover and mutation to these teams. The new population is composed of the currently best teams and the newly generated teams. We call this method as the multi-team algorithm.

In the multi-team algorithm, we randomly generate a set of teams and train them for a number of training episodes. In the training process, the agent learns by playing against itself. The performance of the Q and CQ module, which will be discussed in detail in section C, is improved by the reinforcement learning. In the crossover and mutation steps, teams exchange useful information to improve their performance. Only the best teams are chosen for crossover and mutation in the hope that the offspring are better than the parents The detail of crossover and mutation operator will be discussed later. Teams are chosen by using tournament selection. The detail of the multi-team algorithm is as follows:

1. Randomly generate a set of teams. The number of teams is n.
2. Train each team for a number of episodes.
3. Perform crossover and mutation to generate new teams:
   (a) Select m best teams by using tournament selection.
   (b) Generate n-m new teams by crossover. The crossover rate (the percentage of the weights that have been exchanged between two members) is α.
      i. γ percent of crossover is based on the weight exchange at the corresponding position.
      ii. *100*-γ percent of crossover is based on the weight exchange at a random position.
   (c) Apply mutation on these newly generated teams by randomly mutating selected teams. The mutation rate (the percentage of the weights that have been mutated in a member) is β.
4. Replace the population with the selected teams and the newly generated teams.
5. Go to step 2.

2.2.2 Tournament Selection

For selecting the best teams, we use the tournament selection algorithm, as following:

1. Randomly divide all teams into several groups.

2. In each group, evaluate each group member's fitness value.
3. Select the best performer in each group to form a new set.
4. Repeat step 1 until m members are remained, where m is the number of members we needed.

In our experiments, the fitness value is the winning percentage when a member playing against the benchmark agent. For tournament selection, the higher the fitness value of a member, the higher the chance for that member to be selected. However, this algorithm is not simply selecting the best m members. For example, if the best two members are assigned into the same group, the second best member won't be chosen. Members ranked under m still have a chance to be selected.

2.2.3 The Single-Team Algorithm

In the multi-team algorithm, because the mutation and crossover are done randomly, in some cases, the performance of a newly generated team is worse than that of an old team. Therefore, we propose a new algorithm: the single-team algorithm. In the single-team algorithm, crossover and mutation are only applied in one and only one team. If the new team is worse than the old team, the new team is discarded and the old team is restored. The detail of the single-team algorithm is as follows:

1. Randomly generate a team.
2. Train the team for a number of episodes.
3. Evaluate the team by comparing with the old team. If the team's performance is worse than that of the old team, restore the old team.
4. Randomly exchange weights between members. The crossover rate (the percentage of the weights that have been exchanged between two members) is α.
    (a) γ percent of crossover is based on the weight exchange at the corresponding position.
    (b) *100*-γ percent of crossover is based on the weight exchange at a random position.
5. Randomly mutate selected members. The mutation rate (the percentage of the weights that have been mutated in a member) is β.
6. Replace the team with the newly generated team.
7. Go to step 3.

2.3 Details of the Reinforcement Learning with Bidding

Each member in each team is a reinforcement learning agent. For reinforcement learning, we used the Q-learning algorithm.

The Q-learning algorithm is modified for our multi-agent systems. Each member(agent) in the team (multi-agent system) has two modules: the Q module and the CQ module. In our experiments, both modules are implemented by back-propagation neural networks. Each member can select actions to be performed at each step, which is done by the Q module in the agent. For each member, there is also a controller CQ, which determines at each step whether the agent should continue or relinquish control. Once a member relinquishes its control, to select the next agent, it conducts a bidding process among members (with regard to the current state). Based on the bids, it decides which member should take over from the current point on (as a "subcontractor"), and take the bid as its own reward. The structure of a member is shown in Figure 3.

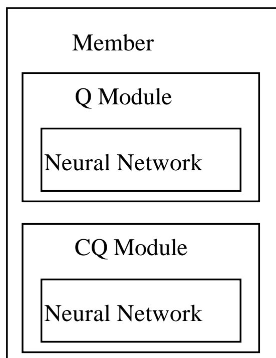

Figure 3: The structure of a member in the GA-based Multi-Agent Reinforcement Learning Bidding System.

In any given state, a member (i.e., a pair of Q and CQ) is in control. When the CQ module in the member selects "continue", the corresponding Q module will select an action with regard to the current state that will affect the environment and thus generate rewards from the environment and incur costs in the environment. When the CQ selects "end", the control is relinquished by the member. A bidding process ensures which proceeds immediately to select another member (a pair of CQ and Q) to take over. This cycle then repeats itself.

Each member decides on its best course of actions based on the total reinforcement that it expects to receive. Each member's CQ module tries to determine whether it is more advantageous to give up or to continue, in terms of maximizing the total reinforcement that it will receive. (When it gives up, it receives a bid as its reinforcement, which represents an estimate of future reinforcement by subcontractors.) Likewise, each member (its Q module) tries to determine which action to take at each step (when it decides to continue), based on total reinforcement that it expects to receive. So, together, each member decides both types of actions based on reinforcement. Furthermore, cooperation among members is formed through the afore-described mutually sharing of re-

inforcement: members utilize each other when such utilization leads to higher reinforcement.

Let state s denote the actual observation by a member at a particular moment. Assume reinforcements and costs are associated with current state, $g(s)$. In each member, there are the following two modules:

- Individual action module Q: each Q module performs actions and learns through Q-learning. Each Q module tries to receive as much reward and incur as little cost as possible before it is forced to give up (including whatever it receives at the last step).
- Individual controller CQ: Each CQ module learns when the member should continue and when the member should give up. The learning is accomplished through (separate) Q-learning. Each CQ tries to determine whether it is more advantageous to terminate the member or to let it continue, in terms of maximizing its future reinforcement, which is also the overall (discounted) reinforcement.

The overall algorithm is as follows:

1. Observe the current state s.
2. The currently active Q/CQ pair (member agent) takes charge. If there is no active pair when the system first starts, go to step 5.
3. The active CQ selects and performs a control action based on $CQ(s, ca)$ for different ca. If the action chosen by CQ is *end*, go to step 5. Otherwise, the active Q selects and performs an action based on $Q(s, a)$ for different a.
4. The active Q and CQ perform learning based on the reinforcement received (see the learning rules later). Go to step 1.
5. The bidding process determines the next pair of Q/CQ (member) to be in control. The member that relinquished control performs learning based on the winning bid (see the learning rules later).
6. Go to step 1.

When a member gives up control, bidding goes as follows: each member submits its bid, and the member with the highest bid value wins. However, during learning, for the sake of exploration, a random selection of bids is conducted based on the Boltzmann distribution:

$$prob(k) = \frac{e^{bid_k/\tau}}{\sum_l e^{bid_l/\tau}}$$

where τ is the temperature that determines the degree of randomness in bid selection. That is, the higher a bid, the more likely the bidder will win. The winner will then subcontract from the current member and the current member takes the chosen bid as its own reward.

We dictate that the bid a member submits must be its best Q value (for the current state); in other words, each member is not free to choose its own bids. A bid is fully determined by a member's experience with regard to the current

state: how much reinforcement (reward and cost) the member will accrue from this point on if it does its best. We call this an "open-book" bidding process, in which there is no possibility of intentional over-bidding or under-bidding. (However, on the other hand, due to lack of sufficient experience, a member may have a Q value that is higher or lower than the correct Q value, in which case over-bidding or under-bidding can occur). A bid submitted by a member in this way represents the expected (discounted) total reinforcement from the current point on, which is the total reward minus the total cost (including possibly its own profit as part of the cost). Note that this total represents not only what will be done by this member but also what will be done by subsequent members (subcontractors) later, due to the subsequent bidding processes (the learning process that takes this into account will be explained next). So, a member, in submitting a bid, takes into account both its own reinforcement and gains from subsequent subcontracting to other members, on the basis of its own experience thus far.

The learning rules can be specified as follows.

- For the active Q_k, the learning rule when neither the current action nor the next action by the corresponding CQ_k is *end* is the usual Q-learning rule:

  $$\Delta Q_k(s,a) = \alpha(g(s) + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a))$$

  where s' is the new state resulting from action a in state s. When the next action by CQ_k is *end*, the Q_k module receives as reward the value of CQ_k (which represents the expected value of the chosen bid at this point):

  $$\Delta Q_k(s,a) = \alpha(g(s) + \gamma CQ_k(s',end) - Q_k(s,a))$$

  where s' is the new state (resulting from action a in state s) in which control is relinquished by CQ_k. $CQ_k(s',end)$ represents the expected value of the bid that the member will accept (from subcontractors), if it gives up control at this point, the value of which is the expected (discounted) total of reinforcement that will be received from that point on by the whole system. This value is given to the Q module so that it can take it into account when deciding on its courses of action (e.g., whether to reach one giving-up point or another).

- For the corresponding CQ_k, there are also two separate learning rules, for the two different actions. When the current action by CQ_k is *continue*, the learning rule is the usual Q-learning rule:

  $$\Delta CQ_k(s,continue) =$$

  $$\alpha(g(s) + \gamma \max_{ca'} CQ_k(s',ca') - CQ_k(s,continue))$$

  where s' is the new state resulting from action *continue* in state s and ca' is any control action by CQ. That is , when CQ_k decides to continue, it accumulates reinforcement generated by the actions of the corresponding Q_k. When the current action by CQ_k is *end*, the learning rule is:

  $$\Delta CQ_k(s,end) = \alpha(\max_{a} Q_l(s,a) - CQ_k(s,end))$$

where Q_l is the Q module of the next member in control (the chosen bidder l) and $\max_a Q_l(s,a)$ is the chosen bid. That is, when a CQ ends, it takes the chosen bid as its reward, which gives it incentive to take higher bids. It learns the expected value of bids, which is the expected (discounted) total reinforcement that will be accumulated by the whole system from the current state on. So in effect, CQ makes its continue/end decisions based on comparing whether giving up control or continuing control will lead to more reinforcement. Thus members are rational in this regard. In HQ-learning [25], an agent giving up its control while it reach its subgoal.

Thus, overall, the members interact and cooperate with each other through bidding as well as individual reinforcement learning. With this *dual* process, the whole multi-agent system learns to form action sequences to facilitate learning. Cooperation among members is forged through bidding and subsequent sharing of reinforcement: a member calls upon another member when such an action leads to higher reinforcement.

3 Experimental Results

3.1 Introduction to Backgammon

One of research in artificial intelligence is programming a computer that can play board games. Board game domains such as Chess and Checkers have been popular since they have finite state spaces with well-defined rules. For these games, the state spaces are so so huge that it is usually impossible to search exhaustively the state spaces. Artificial intelligence research in game domains has primarily worked on solutions that can play a game comparable to or better than a human player. For some games, such as Backgammon [23], Chess (IBM's Deep Blue), Checkers [7], computers achieved the goal. While for other games, such as GO [3], currently computers are playing at the advanced novice level.

Backgammon is a board game played by two people with 15 checkers each on a board consisting of 24 spaces or points. The checkers are moved according to rolls of the dice. Each player tries to bring his own checkers home and bears them off before his opponent does, hitting and blocking the enemy checkers along the way.

Tesauro [24] showed that backgammon is a game that is suitable for approaches involving self-play and random initial conditions. Unlike chess, a draw in backgammon is impossible. The game between computers may take much longer than a game between competent players. However, the game played by an randomly generated network making random moves will eventually terminate. Moreover, the randomness of the dice rolls makes lookahead impossible.

To evaluate our multi-agent player, we play our player against two machine players: a benchmark player and PubEval. The benchmark player is a single agent, which has the same structure as the Q module in a member. The benchmark player is the best player from 15 candidates after a number of training episodes. In our experiments, we use the benchmark player to test the performance of our team players. The training time for the benchmark player is the same as that of the team player. For example, if the team player has been trained for 4,000 games, the benchmark player is the best single agent player from 15 candidates after 4,000 games training.

On the other hand, PubEval is a public machine player by Tesauro and it is a good evaluator for backgammon machine players. PubEval uses a linear function to evaluate the board. Every board will get a point from the linear function. PubEval will move the checker to the board that leads to the maximum possible point.

3.2 Experiment Setup with Encoding Scheme 1

Our backgammon player is a GA-based multi-agent reinforcement learning team. As mentioned before, we use the back-propagation(BP) neural network to implement the Q-learning algorithm. The initial weights for BP networks are randomly generated. The BP networks trained on backgammon use an expanded scheme to encode the local information. We test 2 encoding schemes in our experiments. In encoding scheme 1, for a player's checkers, a truncated unary encoding with five units is used to encode each checker's position (1-24, on the bar and off the board).

For encoding opponent's information, TD-Gammon's encoding scheme [23] is used. For each checker's position, a truncated unary encoding with four units is used. The first three units are encoded three cases: one checker, two checkers and three checkers, while the fourth unit encodes the number of checkers beyond 3. A total of 96 units is used to encode the information at location 1-24. In addition, 2 units are used to encode the number of opponent's checkers on the bar and off the board.

This encoding scheme thus uses 75 units for itself and 98 units for an opponent. In addition, encoding scheme 1 uses 12 units to encode the dice number and an additional 16 units to encode the player's first move, for a total of 201 input units.

The reason we need to encode the player's first move is that, at each round of the game, the player needs to move twice. However, the network only outputs one action each time. When the player makes its second move in a round, we hope it considers its first move at the current round. So we encode the player's first move into the network.

The output encoding scheme uses 16 units to encode the checker. Among these units, 1 to 15 are the checker numbers, while 0 means no action. The hidden units of the BP network for the module Q are 40 and the hidden units of the BP network for controller CQ are 16. We will discuss encoding scheme 2 later.

The initial parameter settings are as follows: the Q value discount rate is 0.95, the learning rate for reinforcement learning is 0.5, and the temperature is 0.50. The mutation rate is 0.05 and the crossover rate is 0.20. Eighty percentage of crossover is the weights exchanging at the corresponding position and twenty percentage of crossover is the weights exchanging at the random position.

The player will receive a reward at the end of a game. The reward table is in Table 1. If the loser has borne off at least one checker, the rewards for the winner and the loser will be 0.3 and -0.3. If the loser is gammoned (has not borne off any of his checkers), the rewards for the winner and the loser will be 0.6 and -0.6. If the loser is backgammoned (no bearing-off of any of his checkers and still has a checker on the bar or in the winner's home board), the rewards for the winner and the loser will be 1.0 and -1.0.

	Winner	Loser
Backgammoned	+1.0	-1.0
Gammoned	+0.6	-0.6
Other	+0.3	-0.3

Table 1: The reward table for the backgammon game

In order to determinate the best number of agents in a team for the backgammon game, we trained 25 single agents for 500 games. Then we selected the best 3, 5, 7, and 9 agents to form 4 teams which including 3, 5, 7, and 9 agents respectively. All teams use randomly generated initial CQs. Each team was tested by playing against the benchmark agent for 300 games. The Figure 4(a) shows the winning percentage of each team playing against the single agent for each 50 games of 250 games. The Figure 4(b) shows the average winning percentage of each team playing against the single agent for 250 games.

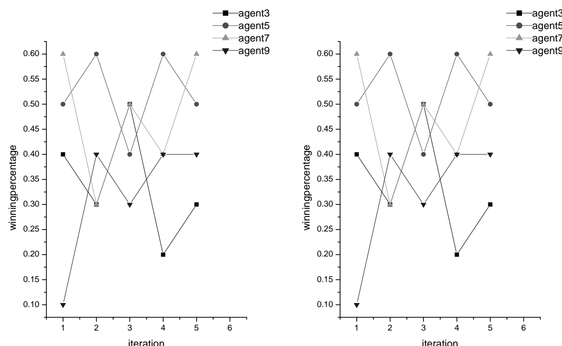

Figure 4: (a) The winning percentage of each team playing against the benchmark agent for each 50 games of 250 games; (b) The average winning percentage of each team playing against the benchmark agent for 250 games.

From these results, the team with 5 agents outperforms the others. So in the subsequent experiments, our team is composed of 5 members.

3.3 Experiment Results with Encoding Scheme 1

We implemented both the multi-team algorithm and the single-team algorithm with encoding scheme 1. For the multi-team algorithm, 15 teams are randomly generated at the beginning. After 200 games training, 5 teams are selected for next generation. By mutation and crossover of these 5 teams, 10 new teams are generated. Plus the 5 selected systems, the new population will be trained for another 200 games.

The weights are crossed over in two ways: between the corresponding positions and between random positions. Two teams are randomly chosen and one member is chosen from each team. Sixteen percent of the weights of these two chosen members is crossed over at the corresponding position and 4 percent of the weights is crossed over at random position.

For the single-team algorithm, the team is formed by selecting the best 5 single agents after 1000 games training. The team is tested after training. If its performance is better than the old team, the crossover will be done within that team and the new team will be tested. Otherwise the old team will be restored and the old team will be crossed over again.

Similar to the multi-team algorithm, weights are crossed over in two ways: between corresponding positions and between random positions. Two members are chosen from the team. Sixteen percent of weights of these two chosen members is crossed over at the corresponding position and 4 percent of weights is crossed over at random position.

During the training, after 200 games, the team is tested by playing against the benchmark agent. The test results of the multi-team algorithm and the single-team algorithm for 4,000 games are shown in Figure 5(a) and 5(b). Both algorithms' performances against the single agent show that the MARLB system has an overwhelming advantage over the single agent. Between these 2 algorithms, the multi-team algorithm has a better average winning percentage when compared to the single-team algorithm. And the best winning percentage for the multi-team algorithm is 92, while that of the single-team algorithm is 88. However, the training time needed for the single-team algorithm is much shorter than the multi-team algorithm. In the multi-team algorithm, we need to train 15 teams but only one team is needed in the single-team algorithm. The single-team algorithm has a much better winning percentage/time ratio.

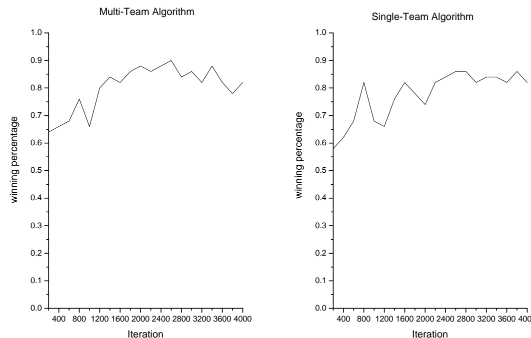

Figure 5: The winning percentage for (a)Multi-team algorithm (b)Single-team algorithm with encoding scheme 1 playing against the benchmark agent.

We also test our multi-team algorithm player with the benchmark agent in different game situations: full game, race game, and bearing-off game. The test results are shown in Figure 6.

In the less complicated situations (the race game and the bearing-off game), the advantage of the multi-agent system over the single agent system is not as large as that of the full game.

We continue to train multi-agent systems with encoding scheme 1 for the full game. The result after 400,000 games is shown in Figure 7. The highest average winning percentage is 58. The average winning percentage is the average of 5 runs, in which each run includes 50 games played against PubEval every 200 iterations.

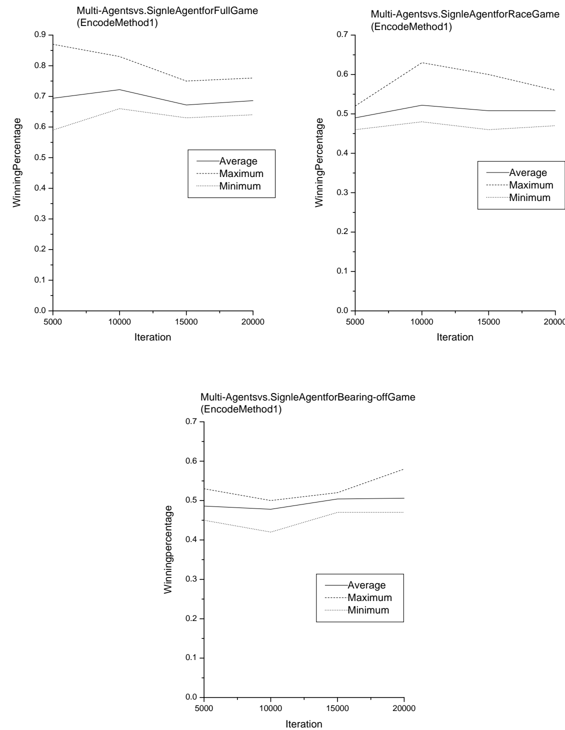

Figure 6: The winning percentage for multi-team algorithm with encoding scheme 1 playing against the benchmark agent in (a) Full Game (b) Race Game (c) Bearing-Off Game

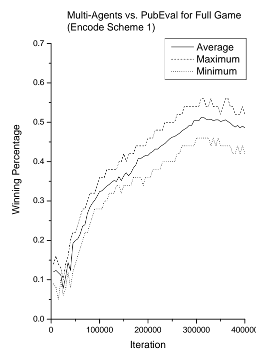

Figure 7: The winning percentage for multi-team algorithm with encoding scheme 1 playing against PubEval in full game.

All experiments were running on HP workstations (HP-UX). The average training time for 200 games of a team is 30 minutes. It took about 18 months to train this multi-team player for 400,000 games.

3.4 Experiment Setup with Encoding Scheme 2

In encoding scheme 2, TD-Gammon's encoding scheme [21] is used for both sides. This encoding scheme uses 96 units for each side, 2 units to encode the checkers on the bar and off the board for each side and 6 units to encode the dice number, for a total of 202 input units. The hidden units of BP network for the module Q are 80 and the hidden units of BP network for controller CQ are 40.

The initial parameter settings and the reward table are the same as those in the encoding scheme 1.

3.5 Experimental Results with Encoding Scheme 2

We only implement the multi-team algorithm with encoding scheme 2. The test results of the multi-team algorithm for 20,000 games in different game situations are shown in Figure 8.

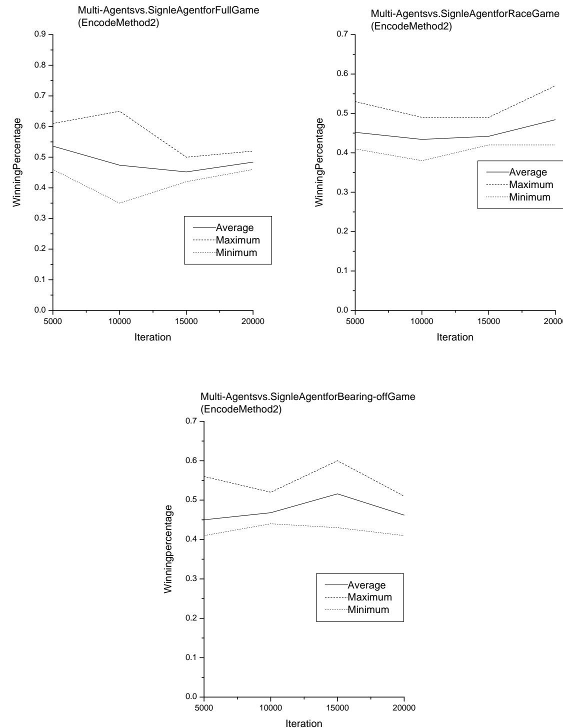

Figure 8: The winning percentage for multi-team algorithm with encoding scheme 2 playing against the benchmark agent in (a) Full Game (b) Race Game (c) Bearing-Off Game

Comparing Figure 8 with Figure 6, we can see that encoding scheme 1 and 2 show similar learning curves. The hidden units for encoding scheme 2 are as twice as that of encoding scheme 1, and the average winning percentage of encoding scheme 1 is lower than that of encoding scheme 2. For the multi-team algorithm, the encoding scheme and the number of hidden units will not significantly change the learning process.

We also continue to train the multi-agent teams with encoding scheme 2 for the full game. The result is shown in Figure 9. The highest average winning percentage is close to 60. The average winning percentage is the average of 5 runs. Each run includes 50 games played against PubEval every 200 iterations.

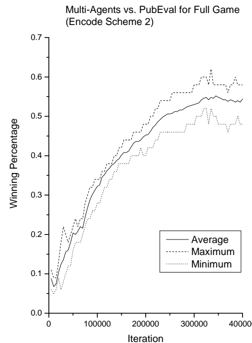

Figure 9: The winning percentage for multi-team algorithm with encoding scheme 2 playing against PubEval in Full Game.

4 Analysis and Discussions

4.1 Cooperation of Agents in the Game

During training, each team plays against itself. Figure 10 is a snapshot of the training process.

The two lines below "My turn" display the current board. A positive number in a board position means self-player's checkers are on that position, where the negative means the opponent's checkers are on that position. The number represents the number of checkers. The point is also calculated during the game as the total distance from each checker's current position to home.

The dice numbers are displayed on the fourth line. During the game, the game is divided into 3 situations: 1 represents the engaging situation, 2 represents the bearing-off situation, and 3 represents the race situation. From the fifth line, the move action, the current situation, and the member who is in control are displayed. After the player moves its checkers, the updated board is displayed again. The opponent's move is displayed in the same way except that the board is in the opponent's direction.

```
********** Round  19 **********
*My Turn*
 0  0 -2  0  0  1 |-4 -5 -2  0  0  0 |  0|  0|     My point:  55
 0  0  0  0  0  0 | 0 -1  3  4  6  1 |-1|  0|   Your point: 122
first dice: 6, second dice: 6.
18 move 6, status: 1, Member: 4.
12 move 6, status: 3, Member: 3.
 4 move 6, status: 3, Member: 3.
 6 move 6, status: 3, Member: 3.
 0  0 -2  0  0  0 |-4 -5 -2  0  0  0 |  0|  0|     My point:  33
 0  0  0  0  0  0 | 0 -1  2  4  6  1 |-1|  2|   Your point: 122

*Your Turn*
 0  0  0  0  0  0 | 0  1 -2 -4 -6 -1 |  1|-2|     My point: 122
 0  0  2  0  0  0 | 4  5  2  0  0  0 |  0|  0|   Your point:  33
first dice: 5, second dice: 4.
25 move 5, status: 2.
10 move 4, status: 2.
 0  0  0  0  0  0 | 0  2 -2 -4 -6 -1 |  0|-2|     My point: 113
 0  0  1  0  0  0 | 5  5  2  0  0  0 |  0|  0|   Your point:  33
```

Figure 10: A snapshot of the the training process.

We can see the multi-agent system change the agent during the game. For the self-player of round 19, member 4 (in control) decided the action for the first move, then it decided to end the control. After bidding, member 3 took control and selected actions for the second, third and fourth moves (The player has double moves since the two number of two dices are the same).

4.2 Member Analysis

We also test the performance of the team playing against its members. The results are in Table 2 and Table 3. All experiment results are for the multi-team algorithm with encoding scheme 1. The performance is measured by the winning percentage when a member played against its team in 50 games.

Iteration	Members in the best team play against the best team		
	Average	Best	Worst
100,000	0.456	0.62	0.38
200,000	0.46	0.52	0.36
300,000	0.456	0.5	0.38
400,000	0.444	0.48	0.40

Table 2: Members in the best team vs the best team. All data are winning percentage of a member playing against its team in 50 games.

From the experimental results, the performance of the best team is better than the average performance of the team. For the best team, performance of the whole team may not be better than that of its best member at the beginning. But after enough training, the team beats its best member. While in the worst team, the performance of the worst team is not as good as its best member. We believe the reason is that the Q module in the best team is better than that of the worst team. In other words, the best team has better cooperation than the worst team.

Iteration	Members in the worst team play against the worst team		
	Average	Best	Worst
100,000	0.48	0.56	0.32
200,000	0.476	0.58	0.42
300,000	0.46	0.54	0.38
400,000	0.46	0.52	0.36

Table 3: Members in the worst team vs the worst team. All data are winning percentage of a member playing against its team in 50 games.

4.3 Component Analysis

Furthermore, we need to know, among the three components of our system, GA, RL and Bidding, which component is more important. In order to answer this question, 4 variants of the multi-team algorithm were proposed. All variant algorithms are tested in the same way as the multi-team algorithm and the single agent algorithm, by playing against the benchmark agent. In variant 1, the genetic algorithm is not applied. Only one bidding team is trained for 4000 games. In variant 2, only the GA and RL are applied. We apply the GA to 15 single agents every 100 iterations (games). Variant 3 is the same as variant 2 except that we apply genetic algorithm to 15 single agents after 200 games. In variant 4, only GA is applied. Since there is no RL in this variant, the BP network only has the forward part, no backforward part. The test results for algorithm variant 1, 2, 3, 4 and 5 are shown in Figure 11(a), 11(b), 11(c), 12(a) and 12(b). All experiments are with encoding scheme 1.

The comparison between the single-team algorithm and the multi-team algorithm with variant algorithms is in Table 4. We can see that variant 5 has the worst average winning percentage. That shows that the GA and bidding, which forge cooperation between agents, are the most important parts in our GA based bidding RL system. Also we can see that variant 2 has the best performance. However, when compared with Figure 4, all variants' average winning percentages and highest winning percentages are lower than those of the multi-team algorithm and the single-team algorithm. In addition, all variants take longer to achieve 80% winning. In sum, all 3 components in our system, GA, RL and bidding, are important. They are synergistic. Missing any component leads to poorer performance.

4.4 Comparison with Other Methods

There has been a great deal of work in the backgammon game. The best machine player so far is Tesauro's TD-Gammon [21] [22] [23]. TD-Gammon used the TD reinforcement learning algorithm [20] to learn from itself. TD-Gammon started from random initial weights but achieved a very strong level of play. Tesauro's 1992 TD-Gammon beat Sun Microsystems' Gammontool and his own Neurogammon 1.0, which trained on expert knowledge. His 1995 player incorporated a number of hand-crafted expert-knowledge features, including concepts like existence of a prime, probability of blots being hit, and probability of escaping from behind the opponent's barrier. This new player achieved the world

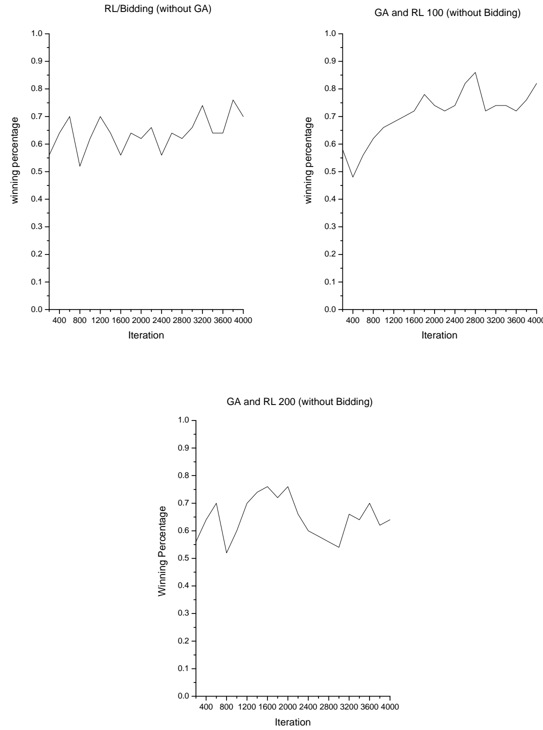

Figure 11: The winning percentage for 4,000 games with encoding scheme 1 when playing against the benchmark agent(a) RL/Bidding (without GA) (b) GA/RL 100 (without Bidding) (c) GA/RL 200 (without Bidding)

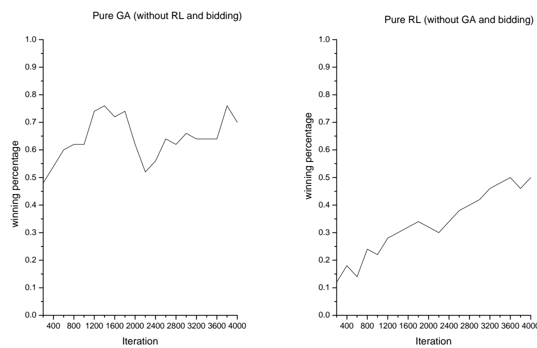

Figure 12: The winning percentage for 4,000 games with encoding scheme 1 when playing against the benchmark agent (a) Pure GA (without RL and bidding) (b) Pure RL (without GA and bidding)

Methods	Best	Average	Average of last 1,000 iterations
Single-Team	0.86	0.78	0.82
Multi-Team	0.90	0.80	0.83
RL/Bidding (without GA)	0.76	0.64	0.69
GA/RL 100 (without Bidding)	0.86	0.71	0.75
GA/RL 200 (without Bidding)	0.76	0.65	0.63
Pure GA (without RL and Bidding)	0.76	0.64	0.67
Pure RL (without GA and Bidding)	0.34	0.5	0.47

Table 4: Comparisons between the single-team algorithm and the multi-team algorithm with variant algorithms. All data is the winning percentage when playing against the benchmark agent.

master player level. Since the best player by Tesauro is not public, we can only use his PubEval, which is close to the best, to evaluate our player.

Pollack et al [13] used feed-forward neural network to develop a backgammon player called HC-Gammon. The neural network does not have an error back-propagation learning part. The player is first generated by random weights and then the network is mutated. The mutated player plays a few games against the original player. If the mutated player wins more than half of games, it survives for next generation. The best player generated by this method wins about 45 percent of games when played against PubEval.

Sanner [17] used the ACT-R theory of cognition [1] to train a backgammon player called ACT-R-Gammon. ACT-R is an empirically derived cognitive architecture intended to model the data from a wide range of cognitive science experiments. ACT-R-Gammon achieved 40 winning percentage when played against PubEval. However, since it used hand-coding of high-level function to facilitate the learning, we do not include this method in our comparison table.

The comparison with other backgammon players is in Table 5.

Among those backgammon players, HC-Gammon has the shortest time to reach 40 winning percentage when playing against PubEval. Our players reach 50 winning percentage in a shorter time. TD-gammon has a slighter better winning percentage. However, it has hand-crafted heuristic codes to improve performance. Furthermore, TD-gammon has less computational load compared to the co-evolutionary approach [5], but it is hard to be parallelized.

	Encoding Scheme 1	Encoding Scheme 2	TD-BackGammon	HC-Gammon
Winning Percentage	51.2 [56]	54.8 [62]	59.25	40 [45]
Iteration	400,000 games	400,000 games	More than 1,000,000 games	100,000 generations

Table 5: Comparisons with other backgammon players. The number is the average of winning percentage of some runs when played against PubEval. The number in brackets is the highest winning percentage.

4.5 Discussions

Cooperation in multi-agent systems. Backgammon is a game based on random numbers (dice numbers), so it is impossible to use the look ahead method as is usually done in other games. Rather than relying on the ability to look ahead, to work out the future consequences of the current state, players of backgammon rely more on judgement to accurately estimate the value of the current board state, without calculating the future outcome. That makes the backgammon unusual in comparison with the other games, such as chess and GO. Although our program did not achieve the same level as TD-Gammon, it achieves a superior level with much less training time and starting from scratch. The cooperation among agents (through bidding) helps to simplify the learning process.

Although the best team does not necessarily includes all the best members in the population, it beats other teams because it has better cooperation among agents. We believe CQ modules are very important for our multi-agent systems.

Hierarchical reinforcement learning. This work also makes some interesting connections to hierarchical reinforcement learning [14] [18]. Our approach amounts to building three-level hierarchies automatically through agent competition (bidding), without relying on extra knowledge or assumptions about domains. The lower layer is a neural network, the middle layer is reinforcement learning with bidding, and the upper layer is the genetic algorithm. All 3 components in our system, GA, RL, and bidding, are important. Missing any component leads to poorer performance.

Co-evolution. The agent in this learning system has two roles: teacher and student. The teacher's goal is to correct the student's mistakes, while the student's goal is to satisfy the teacher and avoid correction. Each agent can be either teacher or student, which depends on its performance in the current stage. The self-learning and self-teaching among agents help our backgammon player to achieves a superior level with zero built-in knowledge.

5 Conclusions

In sum, in this work, we developed a GA bidding approach for performing multi-agent reinforcement learning, to form action sequences to deal with a complex situation: the backgammon game. The experimental results show the

advantage of the bidding system over the single reinforcement learning, the pure GA approach, and the bidding reinforcement learning system without GA. The experiment shows the GA-based bidding system can achieve a superior level of performance in game-playing programs while the system uses zero built-in knowledge. The result of the experiments suggests that the bidding system may work well in general complex problems.

ACKNOWLEDGMENT

This work was supported in part by ARI grant DASW01-00-K-0012, and a University of Missouri-Columbia startup fund.

References

[1] J. R. Anderson and C. Lebiere. *The atomic components of thought.* Lawrence Elbaum Associates, Mahwah, NJ, 1998.

[2] Eric B. Baum. Manifesto for an evolutionary economics of intelligence. *Neural Networks and Machine Learning*, pages 285–344, 1998.

[3] Bruno Bouzy and Tristan Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132:39–103, 2001.

[4] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 746–752, Madison, WI, 1998.

[5] Paul J. Darwen. Why co-evolution beats temporal difference learning at backgammon for a linear architecture, but not a non-linear architecture. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1003–1010, Seoul, Korea, 2001. IEEE Press.

[6] M. d'Inverno, M. Luck, and M. Wooldridge. Cooperation structures. In Martha E. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 600–605, Nagoya, Japan, 1997. Morgan Kaufmann publishers.

[7] David B. Fogel. Evolving a checkers player without relying on human expertise. *Intelligence, ACM Press*, (Summer):21–27, 2000.

[8] J. H. Holland. *Adaption in natural and Artificial Systems.* MIT Press, Cambridge, MA, 1992.

[9] Junling Hu and Michael P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML-98)*, pages 242–250, Madison, WI, 1998. Morgan Kaufmann.

[10] Steven Ketchpel. Forming coalitions in the face of uncertain rewards. In *Proceedings of AAAI*, pages 414–419, 1994.

[11] Pattie Maes, Robert H. Guttman, and Alexandros G. Moukas. Agents that buy and sell. *Communications of the ACM*, 42(3):81–91, 1999.

[12] M. Osborne and S. Russell. *A Course on Game Theory.* MIT Press, Cambridge, MA, 1994.

[13] J. Pollack and A. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32:225–240, 1998.

[14] M. Ring. *Continual Learning in Reinforcement Environments.* Ph.D. dissertation, University of Texas at Austin, 1994.

[15] R. Salustowicz, M. Wiering, and J. Schmidhuber. Learning team strategies: Soccer case studies. *Machine Learning*, 33:263–283, 1998.

[16] Tuomas W. Sandholm and V. R. Lesser. Coalition among computationally bounded agents. *Artificial Intelligence*, 94(1-2):99–137, 1997.

[17] S. Sanner, J. Anderson, C. Lebiere, and M. Lovett. Achieving efficient and cognitively plausible learning in backgammon. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, pages 823–830. Morgan Kaufmann, 2000.

[18] Ron Sun and Todd Peterson. Multi-agent reinforcement learning: Weighting and partitioning. *Neural Networks*, 12(4-5):127–153, 1999.

[19] Ron Sun and Chad Sessions. Bidding in reinforcement learning, a paradigm for multi-agent systems. In *Proceedings of The Third International Conference on Autonomous Agents (AGENTS'99)*, Seattle, WA, 1999.

[20] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[21] Gerald Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[22] Gerald Tesauro. TD-Gammon, a self teaching backgammon program, achieves master-level play. *Neural Computing*, 6(2):215–219, 1994.

[23] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of ACM*, 38(3):58–67, 1995.

[24] Gerald Tesauro and T. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39:357–390, 1989.

[25] Marco Wiering and Juergen Schmidhuber. HQ-Learning: Discovering markovian subgoals for non-markovian reinforcement learning. Technical Report IDSIA-95-96, 1996.

[26] Xin Yao and Yong Liu. From evolving a single neural network to evolving neural network ensembles. In Mukesh J. Patel, Vasant Honavar, and Karthik Balakrishnan, editors, *Advances in the Evolutionary Synthesis of Intelligent Agents*, pages 383–428. MIT Press, Cambridge, MA, 2001.