An Artificially Intelligent Ludo Player

Andres Calderon Jaramillo and Deepak Aravindakshan
Colorado State University
{andrescj, deepakar}@cs.colostate.edu

Abstract
This project replicates results reported from 2 publications about building an artificially intelligent Ludo Player. The first publication analyzes the game complexity and proposes playing strategies for Ludo and its variant race games to create an Expert Player. The second publication from the same authors uses the aforementioned Expert Player to train a TD(λ) based player and a Q-learning based Ludo Player. This project has recreated both the Expert Player and the Q-learning based Ludo Player. Also, we have replicated results for both of these players separately against random players and against each other.

Introduction
Ludo is a board game played by 2-4 players. Each player is assigned a specific color and given four pieces. For example, in Figure 1, we have 4 colors: green, red, blue and purple. The game objective is for players to race around the board by moving their pieces from JAIL to HOME. The winner is the first player who moves all her pieces to her HOME. Note that in Figure 1, we have traced the route for the green player using green arrows. Also, we have labeled the JAIL and the HOME positions for the same player. Some important rules are as follows:
1) A player needs to roll a 6 on a die to release a piece.
2) Orange and other colored squares are safe squares.
3) 2 pieces are required to form a blockade.
4) Releasing a piece is optional when a player rolls a 6.

Motivation
Ludo is a stochastic game with a fairly complex game-play as can be seen above. The first publication (Alhajry, Alvi, and Ahmed 2011) goes into great detail analyzing the complexity of the game, and it demonstrates that the state-space complexity has a lower bound of $10^{22}$ and cannot simply be solved using enumeration. Note that the complexity of Backgammon is $10^{30}$ (Tesauro 1995), and that of Chess is $10^{40}$ (Allis 1994). That is the main motivation behind using reinforcement learning.

Reinforcement learning is a popular machine learning technique and is commonly used for game AI. Temporal difference learning was applied to Parcheesi (Matthews and Rasheed 2008), a variant of Ludo. Using experiments, they proved that training the TD(λ) player using heuristic Parcheesi players improves the learning rate tremendously. Therefore, we implemented the Expert Player proposed in the first player as well. We used this Expert Player to train our Q-learning player. The second publication (Alhajry, Alvi, and Ahmed 2012) proposes and shares results from a TD(λ) based Ludo player and a Q-learning based Ludo player. We have implemented the Q-learning based Ludo Player.
Summary of Papers

In this section, we will discuss the two publications our project is based on in more detail.

Complexity Analysis and Expert Player

The first publication (Alhajry, Alvi, and Ahmed 2011) is titled “Complexity Analysis and Playing Strategies for Ludo and its Variant Race Games.” Firstly, the paper analyzes the complexity of the game and proves the size of its state space to have a lower bound of $10^{22}$. Later, it analyzes the playing strategies to come up with a really strong heuristic player called the Expert Player in order to train a TD($\lambda$) player and a Q-Learning based Ludo Player.

In the second publication, the authors enhanced the strategies proposed in their previous paper (defensive, aggressive, fast, and mixed) in order to account for realistic game rules such as blockades and the optional release of pieces when a 6 is rolled with the die. They found that all strategies significantly outperformed random players. The enhanced strategies were then tested together. With a winning rate of $48.6 \pm 0.41\%$, it was found that the mixed strategy performed significantly better than the individual strategies. Hence, they decided to use the mixed player as the Expert Player for the rest of their publication.

Reinforcement Learning Players

Alhajry, Alvi, and Ahmed (2012) proposed three AI Ludo players. Based on the theoretical and empirical results from their previous paper (Alvi and Ahmed 2011), they finalized the formulation of an Expert Player that uses four strategies. The other two players use reinforcement learning algorithms.

The first reinforcement learning based player proposed by the authors uses the TD($\lambda$) algorithm. This method was chosen in order to allow the player to evaluate each board state and select the best move. The board evaluator is an agent that receives rewards from the environment and is independent of the four players. To represent each state, the authors used 59 inputs for each position in the board for each of the 4 players. 4 additional inputs were used to represent the player that is making the move. Hence, the total number of inputs was 240. Due to the large state space for Ludo, the authors used a neural network to approximate the value function.

The second reinforcement learning based player uses the Q-learning algorithm. This method was chosen in order to allow the player to evaluate the quality of each candidate action at each state and choose the best move. Each state was represented in a subjective manner: the board as seen by each player as opposed to an external, independent observer. Hence, it was not necessary to include the 4 additional inputs to represent the current player's turn. Similar to the TD($\lambda$) player, a neural network was used to approximate the $Q$ function.

The authors found that both the TD($\lambda$) and Q-learning players slightly outperformed the Expert Player (30% winning rate for the TD($\lambda$) player and 27% winning rate for the Q-learning player). The TD($\lambda$) player slightly outperformed the Q-learning player (27.3% vs. 22.3% winning rates). Also, the TD($\lambda$) learning data was less noisy than the Q-learning data. This suggests that TD($\lambda$) is a more stable algorithm in this domain.

Methods

Expert Player Development

Playing strategies
A player has 4 options to move her pieces. In this section, we identify the basic strategies a player might employ during her game-play.

Random Strategy
In a random strategy, a player simply makes a random move out of the possible moves. While it is not really a strategy, a player that only employs a random strategy was used as a benchmark to measure the success of other strategy players.

Fast Strategy
The fast strategy chooses to move the piece that is closest to HOME. The idea behind this strategy is that the piece that is closest to home has used the most number of die rolls and is the most valuable. If it gets knocked off to JAIL, it will be the biggest setback that a player can face.

Aggressive Strategy
The aggressive strategy chooses to move the piece which can knock off an opponent’s piece to its JAIL. The idea is that when an opponent’s piece gets knocked off, the opponent has to play the piece all over the board again. That is a significant setback for the opponent, thereby increasing the chances of an aggressive strategy player to finish first.

An aggressive strategy player will always try to make a move that knocks off an opponent, if possible. Note that blockades and safe squares prevent an opponent’s piece from getting knocked off. This needs to be considered when making an aggressive move. When no such move is possible, it resorts to a random move.

Defensive Strategy
The defensive strategy tries to save its pieces from getting knocked off as much as possible. It does so by computing the knocking range for each of its pieces. If a player’s piece is less than or equal to 6 squares away from one or more of its opponents, then it may be knocked off in a sin-
A single die roll by its opponent. In that case, it is said to be in the knocking range of its opponent. In this strategy, the piece within the knocking range of most opponent pieces is moved.

In the defensive strategy, if all the pieces are within the knocking range of the same number of opponent pieces, then a random move is chosen.

**Mixed Strategy**
The mixed strategy is a hybrid of all the strategies already described above. It is often possible that a strategy does not offer a choice on the move and we have to resort to a random move. It is wiser to choose a different strategy that might offer a choice on the move as long as it is guaranteed to outperform a random move.

All the strategies were individually played against random strategy players for several thousand episodes and it was determined that all the strategies have varying levels of success over a random player. The original paper reported the following levels of success:

- **Defensive > Aggressive > Fast > Random**

Based on that, the mixed strategy player, a hybrid of all the strategies proposed above, will try to make a defensive move first. When this is not possible, it will try to make an aggressive move. If this in turn is not possible, it will make a fast move. If that is not possible either, it will make a random move.

By playing the mixed strategy player several times against other strategy players for thousands of episodes, the original paper reported that the mixed strategy player outperformed all the other strategy players and the random players:

- **Mixed > Defensive > Aggressive > Fast > Random**

The mixed strategy player was chosen as the Expert Player.

**Q-Learning Player**

**Introduction to Reinforcement Learning**
Reinforcement learning is a semi-supervised machine learning method in which an environment rewards an agent for selecting good-quality actions. The agent is not explicitly told which move is best at each state (as in supervised learning). Instead, by interacting with the environment, it must learn which actions to select in order to maximize the total reward obtained.

A reinforcement learning problem is usually modeled as a Markov decision process (MDP). The model has four components (Alhajry, Alvi, and Ahmed 2012):
- A set $S$ of environment states that the agent can observe.
- A set $A$ of actions that the agent is allowed to select.
- A function $P_a(s, s')$ that outputs the probability that the environment transitions from state $s$ to state $s'$ after the agent takes action $a$.
- A function $R_a(s, s')$ that outputs the reward provided by the environment after transitioning from state $s$ to state $s'$ given that the agent selected action $a$.

The goal is to devise a policy that tells the agent what action to select at a specific state. An optimal policy is a function $\pi^*$: $A \rightarrow S$ that maximizes the cumulative reward received $V^*(s)$ for all possible initial states (Alhajry, Alvi, and Ahmed 2012):

$$\pi^* = \arg\max_s V^*(s), \forall s \in S$$

**Q-Learning**
A variety of algorithms have been devised in order to allow the player to find a good policy. Q-learning is a classical reinforcement learning algorithm that builds the value function $V^*$ as the agent interacts with the environment in what are called *learning episodes*. Instead of learning $V^*$ directly, the algorithm learns a function $Q(s, a)$ that outputs the quality of an action $a$ at state $s$.

The outline of the general Q-learning algorithm is as follows:

1) **Initialize the function $Q(s, a)$ with arbitrary output values.** This function can be represented in tabular form.
2) **Start a learning episode.** The agent interacts with the environment. Every time the environment rewards the agent, the $Q$ function is updated according to the following formula:

$$Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)]$$

In this formula, $s$ is the current state, $s'$ is the new state, $a$ is the action that the agent selected, $r$ is the reward provided by the environment, $\alpha$ is the learning rate, and $\gamma$ is the discount factor. The learning rate controls how fast the $Q$ function is updated. The discount factor controls how much future reward should be taken into account.

3) **Repeat step 2 as many times as desired.** Once training is complete, the new $Q(s, a)$ function can be used by an agent in order to select actions.

Our design of the Q-learning player is based on the design by Alhajry, Alvi, and Ahmed (2012). States are represented using 236 variables:
- The first 59 variables represent the current player’s board positions. Each variable is a number between 0 and 1 (inclusive) that represents the percentage of the player’s pieces that are in that position. For example, if the current player has 3 of the 4 pieces in position 5, the variable corresponding to this position has a value of 0.75.
- The next 59 variables represent the next player’s board positions, and so on to account for all 4 players.
- The first variable for each of the 4 players corresponds to the JAIL position. The initial value of this variable is 1 for all players. The last variable for each player corre-
sponds to the HOME position. A player wins when this variable becomes 1.

An action consists of moving a single piece from one position to another. Every action is represented as a tuple \((x_0 / 58, x_f / 58)\), where \(x_0\) is the initial position and \(x_f\) is the final position. The components are divided by 58 in order to obtain a number between 0 and 1 (the first position is labeled 0 and the last position is labeled 58).

The player receives a reward immediately after selecting an action. Rewards are assigned as follows in order to reflect the knowledge acquired from building the Expert Player (Alhajry, Alvi, and Ahmed 2012):

- 1.0 for winning a game.
- 0.25 for releasing a piece from JAIL.
- 0.2 for defending a vulnerable piece.
- 0.15 for knocking an opponent’s piece.
- 0.1 for moving the piece that is closest to home.
- 0.05 for forming a blockade.
- -0.25 for getting a piece knocked in the next turn.
- -1.0 for losing a game.

Rewards can be accumulated. If a move does not fall in one of these situations, no reward is given.

Similar to Alhajry, Alvi, and Ahmed (2012), we use a neural network to approximate the Q function. The network we use is a fully connected feedforward neural network with the following structure:

- 238 input units. Of these, 236 are used for the state \(s\), and 2 are used for the action tuple \(a\).
- A single hidden layer of 20 units. A symmetric sigmoid activation function is used for these units.
- 1 output unit that represents \(Q(s, a)\). A linear (unbounded) activation function is used for this unit.

We changed the Q function update rule to reflect the nature of the game. Specifically, instead of selecting the maximum estimated future Q value, we select the minimum. This is because once the current player makes a move, the new state “belongs” to the next player, and the current player should aim to minimize the opponent’s reward while maximizing its own. Hence, the formula becomes:

\[
Q(s, a) = Q(s, a) + \alpha [r + \gamma \min_{a' \in A} Q(s', a') - Q(s, a)]
\]

As in the original paper, to balance between exploration and exploitation, our player uses an \(\epsilon\)-greedy strategy during training to select actions: a random action is selected with probability \(\epsilon\), and a best action (as ranked by the Q function) is selected with probability 1 - \(\epsilon\). When not in training mode, the player always selects the best option.

### Evaluation

#### Experimental Setup

All the code was implemented using Python. All the experiments were run using Python on Linux machines for 100,000 episodes each.

#### Expert Player Evaluation

The Expert Player was evaluated by running it against the 3 basic strategy players (Defensive, Aggressive, and Fast) for 100,000 episodes. Also, it was run against 3 random players separately for 100,000 episodes.

#### Q-Learning Player Evaluation

We used the FANN library for the neural network. The FANN (Fast Artificial Neural Network) library is a C implementation of training algorithms for multilayer artificial neural networks (Nissen 2003). We used the publicly available Python bindings. The library was selected over others such as PyBrain and NeuroLab due to its native performance.

To evaluate the performance of the algorithm, we trained 4 Q-learning players by letting them play against themselves for 100,000 episodes. Similar to the setup by Alhajry, Alvi, and Ahmed (2012), all 4 players shared the same neural network for faster training. Separately, we also trained a single Q-learning player using 3 other Expert players. In both cases, for the Q-learning algorithm, we used \( \alpha = 0.5 \) and \( \gamma = 0.95 \). The neural network was trained incrementally using \( \alpha = 0.005 \) and \( \beta = 0.1 \). The low learning rate for the neural network was selected to prevent learning from becoming too unstable. Training was regularly paused to test the performance of the current neural network. We do this by making one Q-learning player use this network to play in two scenarios: a) 1,000 times against 3 random players and b) 1,000 times against 3 Expert players. For each scenario, we calculated the winning rate as the number of games won by the Q-learning player divided by 1,000. This allowed us to measure its performance in terms of the number of training episodes. For the \(\epsilon\)-greedy policy, similar to the original paper, \(\epsilon\) starts at 0.9 and decreases linearly to 0 after 10,000 episodes.

Recall that the Q-learning algorithm requires an arbitrarily initialized \(Q\) function. The neural network’s initial weights are assigned randomly to achieve the required initializations. Thus, the learning process will depend, at least initially, on the assigned initial weights. To account for this effect, we ran the experiments on 4 separate computers and aggregated the results.
Results

Expert Player

Figure 2 summarizes the results of the Expert Player against the basic strategy players. Figure 3 summarizes the results of the Expert Player against 3 random players. As evidenced by the graph, the Expert Player outclasses the basic strategy players and the random players quite easily. As reported in the original paper, the Defensive strategy is the best among the 3 basic strategies. Nonetheless, we found that the Fast strategy slightly outperformed the Aggressive strategy, contrary to the results reported in the original paper.

Figure 2: Winning rates of Expert Player vs. basic strategy players

Figure 3: Winning rates of Expert Player vs. 3 random players

Q-Learning Player

Self-Training with 4 Q-Learning Players

Figure 4 and Figure 5 summarize the winning rates of our Q-learning player training using self-play with 3 other Q-learning players. It outperforms the random player quite easily, but the winning rate plateaued at around 35%.

Figure 4: Self-training with 4 Q-learning players and testing of one Q-learning player against 3 random players. The thick line represents the average winning rate over 4 computers. The thin lines represent the maximum and minimum rates.

Figure 5: Self-training with 4 Q-learning players and testing of one Q-learning player against 3 Expert players. The thick line represents the average winning rate over 4 computers. The thin lines represent the maximum and minimum rates.

Training with 3 Expert Players

Figure 6 and Figure 7 summarize the winning rates of our Q-learning player trained using 3 Expert players. Again, it outperforms the random player quite easily. However, the winning rate decreased gradually from 50% to 35% and plateaued after 40,000 episodes.

Figure 6: Q-learning player trained with 3 Expert players and testing of one Q-learning player against 3 random players. The thick line represents the average winning rate over 4 computers. The thin lines represent the maximum and minimum rates.

Figure 7: Q-learning player trained with 3 Expert players and testing of one Q-learning player against 3 Expert players. The thick line represents the average winning rate over 4 computers. The thin lines represent the maximum and minimum rates.

However, as shown in Figure 5, it is unable to defeat the Expert Player. The learning plateaued after 20,000 episodes and remained close to 20%.
The success rate against the Expert Player was well around and above the targeted 27% around 20,000 episodes. Later, it gradually decreased and stayed slightly below 20%. At this point, our experiments provide little evidence to indicate that training the Q-learning player using Expert players is beneficial.

There is one common theme in all Q-learning training scenarios. In the early stages of training, the Q-learning player’s performance reaches a peak. Then, after a certain number of learning episodes, the player’s winning rate starts to steadily decrease to eventually reach a plateau. This suggests a number of possible issues:

1) The parameters may need to be tuned further. We chose a small learning rate for the neural network to avoid noisy data. It might be necessary to dynamically change this parameter as learning progresses in order to avoid local optima.

2) The ε-greedy strategy did not explore enough of the search space. This may have contributed to the stagnation of learning. It might be worth investigating the effect of using a small constant ε throughout the entire learning process.

3) The neural network topology may need to be altered to increase the learning capacity. This will most likely require increasing the number of learning episodes.

These problems prevented us from fully replicating the results in (Alhajry, Alvi, and Ahmed 2012): they reported a winning rate of 63 ± 1% against 3 random players and 27 ± 1% against 3 Expert players. They also reported that the player’s performance improved with increasing number of episodes.

**Conclusion**

In this project, we replicated algorithms proposed in two existing publications to play Ludo. We implemented four basic strategy players: Defensive, Aggressive, Fast, and Random. We then implemented an Expert Player that uses a mix of the basic strategies and prioritizes them to achieve better performance. Finally, we implemented a reinforcement learning based player that uses the Q-learning algorithm with a neural network to devise a policy for playing Ludo.

Our experiments showed that the Expert Player performed consistently better than players using only one of the basic strategies. It also outperformed random players. Our Q-learning player was able to learn a policy to consistently defeat random players. Unfortunately, it was not able to consistently defeat the Expert players suggesting that the mixed strategy is quite robust.

Future work can be grouped in the following categories:

1) Further tuning of parameters to avoid local optima while preventing instability in the learning process.

2) Experimentation with different reward values in order for the Q-learning player to learn a policy to consistently defeat the Expert Player.

3) Adaptation of the playing interface to allow a human player to compete against the algorithms implemented in this project.

**Member Contributions**

<table>
<thead>
<tr>
<th>Work Item</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common interfaces, classes and logic</td>
<td>Andres</td>
</tr>
<tr>
<td>Expert Player and the various strategies</td>
<td>Deepak</td>
</tr>
<tr>
<td>Q-learning player and the neural networks</td>
<td>Andres</td>
</tr>
<tr>
<td>Training with 4 Q-Learning players</td>
<td>Andres</td>
</tr>
<tr>
<td>Training with 1 Q-Learning and 3 Expert players</td>
<td>Deepak</td>
</tr>
<tr>
<td>Structure preparation for presentation and report</td>
<td>Deepak</td>
</tr>
</tbody>
</table>

**References**


Alhajry, M.; Alvi, F.; and Ahmed, M. 2012. TD (λ) and Q-Learning Based Ludo Players. In IEEE Conference on Computational Intelligence and Games, 83–90.


Nissen, S. 2003. Implementation of a Fast Artificial Neural Network Library (fann), Department of Computer Science, University of Copenhagen (DIKU).