Fighting the Landlord Game AI

Leyi Qiang
College of Computer and Information Science
Northeastern University,
Boston, USA

Abstract

Fighting the Landlord is a famous Chinese poker game that is played under a stochastic, partially-observable environment. The aim of this paper is to implement an AI to play the Fighting the Landlord game with a good performance. Two algorithms are discussed and implemented in this paper: Minimax with Alpha-Beta pruning and Monte-Carlo tree search. The experiment simulates the game under different configurations. The results indicate that agents that use Monte-Carlo Tree Search algorithm with large sampling time can have a better performance than the agents that use Minimax algorithm.

Introduction

This paper is concerned with creating a computer to play Fighting the Landlord, a 3-player gambling poker game. Fighting the Landlord has several features that are different than many other games like Pacman [5] and chess [11]. This game is played under a stochastic environment where each player receives cards randomly. Moreover, players are played under partially-observable environment where they can only observe the discarded cards and the cards on their own hand. To solve these problems, I investigate on the application of Minimax Algorithms and Monte-Carlo Tree Search (MCTS) and compared the performance of each algorithms under multiple variances.

To successfully apply Minimax algorithm, each player needs to know everything about the possible moves and the next states of other players. In order to apply minimax algorithm to the agent, an easier version of the game is implemented, each player can check other players' hand and thus make the environment fully observable. However, due to the uncertainty of the environment, each state has large branching factors, which causes minimax infeasible to run under large depth. Alpha-Beta pruning and better evaluation function is provided to mitigate the problem.

To solve this problem and to allow the program to have better performance under partially observable environment, MCTS is used. MCTS algorithm has been used for computer Go programs and has been proven to have good performance in the Go game [6]. The main idea of MCTS is to sample and simulate large amount of games from current game state and choose a subsequent state that has maximum winning rate. This mechanism helps agent to determine an approximate optimal action without traverse through all branches.

This paper firstly reviews some background information related to this project. Including minimax algorithm with alpha-beta pruning, MCTS and the rules of Fighting the landlord game. The methodology session talks about the way I performed the experiment, including the goal of the experiment, the information and game state variables I used to set up the environment, detailed implementation of each techniques and the method I used to collect data. The next two session presents the result and analysis of my data, which records and compares the win rate using a combination of different configurations and algorithms. In the end, I provide conclusions of the game and some possible future works related to this project.

Literature Review and Background

Minimax Algorithm with Alpha-Beta Pruning

Minimax [9] is a tree search algorithm that mainly used in two-player, turn-based zero-sum games. This algorithm uses a tree, where each node is the state of the game and each branch is the action the agent can do. A minimax algorithm consists of a maximizer (MAX) and a minimizer (MIN), where MAX tries to select the action that returns maximum utility value and the MIN tries to select the action that returns minimum utility value. Each leaf node represents the terminal state alone with a utility value from MAX's aspect. During the game, MAX moves first, following by MIN, they will take turns until the game reaches a terminal state or the tree search reaches maximum depth. Given a state *s*, the decision is defined as following:

$$MINIMAX(s) = \begin{cases} UTILITY(s)\ if\ IS\_TERMINAL(s) \\ max_{a\in Actions(s)} MINIMAX(RESULT(s,a))\ if\ PLAYER(s) = MAX \\ min_{a\in Actions(s)} MINIMAX(RESULT(s,a)\ if\ PLAYER(s) = MIN \end{cases}$$

This algorithm has been widely used in many games such as Pacman [5] and Tic-Tac-Toe [12]. A minimax agent performs optimally if the game is solvable (has finite number of states) and its opponent also performs optimally and act like a minimizer.

The drawback of using Minimax is that the number of states it needs to explore grows exponentially based on the depth of the tree. It takes *O(b^d)* time to run where *b* is the total number of next states and *d* is the depth. This problem can be mitigated by applying Alpha-Beta Pruning so that the program will not need to go through every node in the tree. A node *n* will not be reached if the agent has a better choice node at parents of *n*. A Minimax algorithm with Alpha-Beta pruning requires only *O($b^{d/2}$)* time to run.

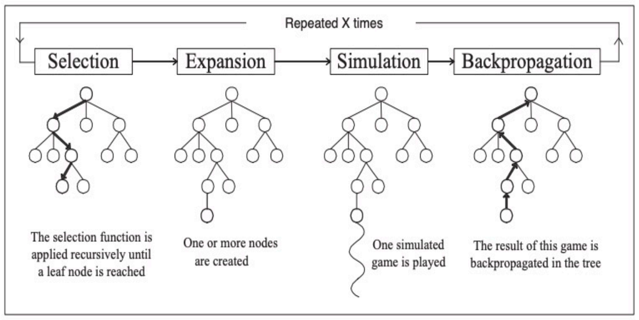
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1: Outline of a Monte-Carlo Tree Search [3].

Monte-Carlo Tree Search

Monte-Carlo Tree Search [3] is a tree search algorithm. It predicts a best move by running a large amount of simulation and calculating the win rate of each node. The algorithm runs in the following steps (see Figure 1): selection – expansion – simulation – back-propagation. In the selection part, the agent starts at current state and select the next action recursively until a leaf node is reached. The selection of next action is based on the data collected in previous simulations. Some selection algorithms can be used to balance between exploration and exploitation and improve the efficiency of selection such as UCT [8] and UCB [1] algorithms. In the expansion part, the child node of the leaf node is created and initialized. In the simulation part, the programs simulate the game by randomly select next action. The simulation stops when the game ends or it reaches maximum depth. In the end, the programs go into back-propagation part, which update the tree nodes traversed during previous parts with the simulation result.

Since MCTS choose an action by sampling simulation with random or pseudo-random moves. The agent can have great performance without knowing any rules of the game. This algorithm has many potential usages on many difficult games, which the game state has a large branching factors or has a partially observable environment. MCTS is not only used in some traditional games for better performance like Go [6], but also some modern video games such as Magic: The Gathering [4] and Hearthstone [10].

Fighting the Landlord

Fighting the Landlord (Dou Di Zhu) [7] is a famous Chinese poker game. The game consists of three players: A landlord who plays alone, and two farmers group together against the landlord. This game uses a 54-card deck including red joker and black joker. Other than two jokers, 2 is the highest rank card, followed by Ace and 3 is the lowest ranked card (rank from low to high: 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, 2, A, black joker, red joker). Suits of cards are irrelevant in this game. One player shuffles the cards, deals 17 cards to each farmer and 20 cards to the landlord. Players can only see the cards in his own hand.

Name	**Description**
Single	Single card ranking as explained above
Pair	Two cards of the same rank
Trio	Three cards of the same rank
Chain	Five or more consecutive numbered cards, rank is compared based on the lowest ranked card in the chain. Twos and jokers cannot be used
Pairs Chain	3 or more consecutive numbered pairs. Twos and jokers cannot be used
Trio Single	Trio with a single card. Rank according to the rank of trio
Trio Pair	Trio attached with a pair
Airplane	Two or more consecutive trios. Twos and jokers cannot be used
Airplane with Small Wings	Airplane, but an extra single card is attached to each trio
Airplane with Large Wings	Airplane, but an extra pair is attached to each trio
Four with Two	Four-of-a-kind, with two single cards as a kicker
Four with Pairs	Four-of-a-kind, with two pairs as a kicker
Bomb	Four-of-a-kind, can beat every play except king bomb or higher rank bomb
King Bomb	A pair of jokers

Table 1: Types of legal combination an agent can play.

To play the game, the landlord starts first, he can play a single card or any possible legal combinations (see Table 1).

Each subsequent player in counter-clockwise order must either choose to pass (do not play any card) or beat previous play. To beat previous play, the player needs play the same combination and length of card as previous play, and the play needs to have a higher rank than the previous play. Each player keeps playing until two consecutive players choose pass, then the next player starts again and is allowed to lead any card or combinations.

The game ends if any of the player runs out of cards. The landlord wins if he runs out of cards first. The two farmers win if any farmer runs out of cards in hand. Note that the two farmers are formed as a team to play against landlord in this game, a farmer can win by helping the other farmer to win. Thus, cooperation between farmers is a commonly used strategy in the game.

Methodology

The goal of this research is to find a computer program that has better performance when playing Fighting the Landlord game. Thus, this experience compares performances of different programs playing the game under different setups.

Environment Setup

The game consists of three agents: landlord, farmer1 and farmer2. A board state is used to record all the data needed for agents to compute next steps. Including:

- Current turn: The player that is currently doing actions. It loops in an order of landlord - farmer1 - farmer2.
- Previous play: The cards previous agent played. Can be empty if the previous agent chose to pass
- Discarded card: Although discarded cards are not observable in the actual game. Professional players remember all discarded cards. So, I store all the discarded cards for the computer agent
- Hands: Current cards in each player's hand. In the fully observable version of the game, an agent can obtain the information of hands for all players. Whereas in the partially observable version, an agent only obtains the information of its own hand

To initialize the game state, all cards are shuffled. Each farmer receives 17 cards and landlord receives 20 cards randomly. Discarded cards and previous play are set to empty and current turn is set to landlord.

In an agent's turn, he can do one of the two actions: pass or play. If he chooses to pass, previous play is set to empty. If he chooses to play, he needs to play a single card, or any legal combination based on the rules of the game. The played cards will be removed from hands and added to discarded cards and previous play. After the agent has done any of the action, current turn will be switched to the next agent.

Techniques

Three types of agent are implemented in the research: reflect agent, Minimax with Alpha-Aeta pruning agent and MCTS agent. A reflect agent only choose his next action randomly. In both minimax algorithm and MCTS, a tree structure is used to perform the calculation. A board state represents a node of the tree, and the action of an agent represents the branch of the tree.

1. Minimax Algorithm Since Minimax requires agents to read the full information of the game. A minimax agent is able to obtain the information of the other players' hand. A farmer agent treats himself and the other farmer agent as maximizer, as they cooperate with each other to get the maximum utility value and the landlord is a minimizer. From the landlord agent's perspective, the landlord is a maximizer and the two farmers are minimizers.

In the beginning of a Fighting the Landlord game, each state has more than 50 successors. The agent needs to go through 50^3 states at depth 3, which makes the program infeasible to run. Thus, alpha-beta pruning is applied to reduce the running time. The algorithm used for minimax with alpha-beta pruning is expressed as following:

Code 1 Minimax with Alpha-Beta Pruning

```
1: function GETALPHABETAACTION(state)
2:   $\alpha \leftarrow -\infty, \beta \leftarrow \infty, d \leftarrow 0$
3:   $v \leftarrow$ GETMAXVALUE(state, d, $\alpha$, $\beta$):
4:   return action with value v in GETACTIONS(state)

1: function GETMAXVALUE(state, d, $\alpha$, $\beta$)
2:   if ISTERMINAL(state) or d == max_depth then
3:       return UTILITY(state)
4:   end if
5:   $v \leftarrow -\infty$
6:   if state.current_turn == famer2 then
7:       for each a in GETACTIONS(state) do
8:           $v' \leftarrow$ GETMINVALUE(NEXTSTATE(state, a), d, $\alpha$, $\beta$)
9:           $v \leftarrow$ MAX(v, v')
10:          if $v \geq \beta$ then
11:              return v
12:          end if
13:          $\alpha \leftarrow$ MAX($\alpha$, v)
14:      end for
15:  else
16:      for each a in GETACTIONS(state) do
17:          $v' \leftarrow$ GETMAXVALUE(NEXTSTATE(state, a), d, $\alpha$, $\beta$)
18:          $v \leftarrow$ MAX(v, v')
19:          if $v \geq \beta$ then
20:              return v
21:          end if
22:          $\alpha \leftarrow$ MAX($\alpha$, v)
```

```
23:     end for
24:   end if
25:   return $v$
```

```
 1: function GetMinValue($state$, $d$, $\alpha$, $\beta$)
 2:   if IsTerminal($state$) or $d == max\_depth$ then
 3:     return Utility($state$)
 4:   end if
 5:   $v \leftarrow \infty$
 6:   for each $a$ in GetActions($state$) do
 7:     $v' \leftarrow$ GetMaxValue(NextState($state$, $a$), $d$, $\alpha$,
          $\beta$ )
 8:     $v \leftarrow$ Min($v$, $v'$)
 9:     if $v \leq \alpha$ then
10:       return $v$
11:     end if
12:     $\beta \leftarrow$ Min($\beta$, v)
13:   end for
```

When the game terminates, the utility function returns negative 1000 if the agent loses and positive 1000 if the agent win. In addition, three types of utility functions are provided for player *P* when the game is not terminated.

$$Utility_P = 20 - Len(Hand_P) \qquad (1)$$

The first type of utility function as shown above ensures the agent plays longest combination first. The value will always be positive since the maximum length of a hand is 20. The utility value is inversely proportional with the total cards in hand: the less cards an agent has in hand, the more utility value he can gain.

$$Utility_P = \sum_{i=0}^{Len(Hand_P)} Value(Hand_{Pi}) \qquad (2)$$

The second type of utility function allows agents to play least ranked cards first. I granted 3 a value of -2 and the value increases by 1 as the rank of card increases. Negative values are assigned to lowest ranked cards so that players will gain utility when they play these cards. The third utility function combines the two mechanism together by adding the two equations.

$$Utility_P = Utility_P + Utility'_P \qquad (3)$$

2. Monte-Carlo Tree Search The MCTS agent runs two major function: get action and run MCTS. In the get action part, it runs a large amount of MCTS starting from current state. After the agent has recorded enough data. It selects and returns the move with the highest win rate.

In the run MCTS part, the program goes through four steps as mentioned in the previous session. In the selection part, if next states of current state are all recorded (the state is not fully expanded), the program then chooses next move using UCB1 algorithm [1] by selecting the node n with the highest calculated value. The value is calculated as following:

$$\text{Value}_\text{n} = \frac{\text{Wins}_\text{n}}{\text{Plays}_\text{n}} + \sqrt{\frac{2\ ln(Total\ Plays)}{Plays_n}}$$

If current state is not fully expanded. There is a 50% chance to select next state randomly or select visited state using UCB1 algorithm. This ensures the agent has enough exploitation as it is rarely to have a fully expanded node.

Code 2 Monte-Carlo Tree Search

```
 1: function GetMCTSAction($state$)
 2:   $timeout \leftarrow$ 30 seconds
 3:   while not $timeout$ do
 4:     $parent\_states$, $selected\_state$ = Selection($state$)
 5:     if IsTerminal($selected\_state$) then
 6:       if IsWin($agent$) then $r = 1$ else $r = 0$
 7:     else
 8:       $r$ = Simulation($selected\_state$)
 9:     end if
10:     BackPropagation($parent\_states$, $r$)
11:   end while
12:   return $action$ with Max($wins/plays$) in GetAc
          tions($state$)
```

```
 1: function Selection($state$)
 2:   $parent\_states \leftarrow$ []
 3:   next_states ← GetNextStates($state$)
 3:   if IsFullyExpanded($next\_states$) then
 4:     $selected\_state$ = UCB1($next\_states$)
 5:     Append($parent\_states$, $selected\_state$)
 6:     Selection(selected_state)
 7:   else if Random(0, 1) < 0.5 then
 8:     $selected\_state$ = UCB1($next\_states$)
 9:     Append($parent\_states$, $selected\_state$)
10:     Selection(selected_state)
11:   else
12:     $selected\_state$ = Random($next\_states$)
13:     Append($parent\_states$, $selected\_state$)
14:     return $parent\_states$, $selected\_state$
15:   end if
```

```
 1: function RollOut($state$)
 2:   while not IsTerminated($state$)
 3:     $action$ = Random(GetActions($state$))
 4:     $selected\_state$ = Result($state$, $action$)
 5:     if IsWin($agent$, $selected\_state$) then return 1
 8:   end while
 9:   return 0
```

```
 1: function BackPropagation($parent\_states$, $r$)
 2:   for $state$ in $parent\_states$
 3:     $plays[state]$ += 1
 4:     $wins[state]$ += r
```

```
5: end for
```

Data Collection

To collect data and compare the win rate using different configurations, multiple scenarios are created. In every scenario, the Alpha-Beta Pruning algorithm is set with a depth limit of 2, and the MCTS algorithm always uses a random rollout policy. These scenarios are:

- All Alpha-Beta agents where farmer1 and farmer2 use utility function (1), landlord uses utility function (2) and vice versa
- All Alpha-Beta agents where farmer1 and farmer2 use utility function (1), landlord uses utility function (3) and vice versa
- All Alpha-Beta agents where farmer1 and farmer2 use utility function (2), landlord uses utility function (3) and vice versa
- Farmer1 and farmer 2 use Alpha-Beta pruning with utility function (3) against reflex landlord agent and vice versa
- Farmer1 and farmer2 uses MCTS with 30 seconds simulation time limit, landlord uses reflex agent (3) and vice versa
- Farmer1 and farmer2 uses MCTS with 5 seconds simulation time limit, landlord uses Alpha-Beta pruning with utility function (3) and vice versa
- Farmer1 and farmer2 uses MCTS with 30 seconds simulation time limit, landlord uses Alpha-Beta pruning with utility function (3) and vice versa

Each scenario is simulated 100 times, and the number of wins for farmer and landlord is recorded. In order calculate the total wins of an algorithm, I added the total wins of the algorithm for both landlord and farmer and divided the total played time.

	Landlord	Farmer	Total Win Rate
Utility (1)	27	36	31.5%
Utility (3)	64	73	68.5%
Utility (2)	37	43	40.0%
Utility (3)	57	63	60.0%
Utility (2)	54	66	60.0%
Utility (1)	34	46	40.0%

Table 1: Winning counts and win rate of landlord and farmers agents plays against each other using different utility function.

Result

The data from Table 1 shows that utility function (1) has the least performance among the three utility functions. One major reason is that the agents only consider the cards length of the play regardless of the rank of each cards. This type of agent runs out of card fast in the beginning, but get thwarted in the end because he does not have any high rank cards left to beat previous play. The utility function (2) has better performance than the utility function (1). Although the function only calculates the sum of card values in hand, the length of the card is also considered: Since the lower rank cards as negative values, the more lower rank cards an agent play, the more utility value the agent will gain. A drawback of this function is that the agents prefers to reserve hands when he has high value cards in hand. The third utility function shows the best performance because it not only considers the length of the cards in hand, but also the rank of the cards. It beat

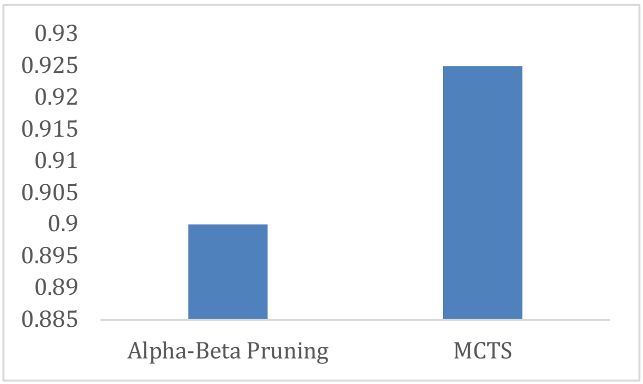

Figure 2: Win rate of Alpha-Beta Pruning agents with utility function (3) and win rate of MCTS with 30 seconds simulation time limit.

Figure 2 shows that when running Alpha-Beta pruning and MCTS agents against reflex agents, both of the two algorithms have greater performance and the win rates are over 90%. Since Fighting the Landlord is a stochastic game, it is not guaranteed there exist a way to win a game for every round, which give reflex agents a 10% chance to win. The result also shows that agents that use MCTS have 92.5% winning rate against reflex agents, which is slightly higher than the win rate of minimax agent. However, from this result it is still not guaranteed that MCTS has better performance than Alpha-Beta Pruning due to the randomness of the environment, and thus more experience data is provided below to compare the performance between Alpha-Beta pruning and MCTS.

	Landlord	farmer	Total Win Rate
Alpha-Beta	72	55	63.5%
MCTS (5s)	45	28	36.5%
Alpha-Beta	41	33	37.0%
MCTS (30s)	67	59	63.0%

Table 2: Winning counts of agents use Alpha-Beta pruning against agents use MCTS of different sampling times.

When running MCTS with short sampling time (5 seconds). The performance is worse than Alpha-Beta pruning as shown in Table 2. This is because within 5 seconds, the MCTS agent is not able to collect and explore enough data to make a good decision. Note that the difference of winning rate of landlord (72 for Alpha-Beta pruning and 28 for MCTS) is a lot larger than the farmer (55 for Alpha-Beta pruning and 45 for MCTS). This is because a landlord agent has more cards than the farmer agents, which leads to a larger branching factors and further reduces the accuracy of MCTS. After setting the sampling time to 30 seconds, the agents are able to collect enough data and fully explore different actions. Since the depth of Alpha-Beta pruning is limited to a depth of 2, the agent is not able to fully explore the entire tree and has to rely on the utility function, which reduces the performances.

Conclusion

The purpose of this research is to find a computer that has a good performance to play Fighting the Landlord game. Minimax Algorithm with Alpha-Beta pruning and Monte-Carlo Tree Search is implemented to be tested on the experiment. Since Fighting the Landlord game is played under a stochastic environment and it contains a large variance of combinations. Large branching factors becomes one of the major issues of the performance. Minimax algorithm is limited to a depth of two due to the long running time, and thus a better utility function is required to improve the performance of the algorithm. MCTS is a better solution of this game. However, it also requires a large amount of sampling time. Overall, the results of the data indicate that both Alpha-Beta pruning and MCTS with good configurations has an improved performance, and MCTS with larger sampling time has a better winning rate than Minimax with Alpha-Beta pruning.

Future work and improvement may also be required for this research. Due to the limitation of my time, I only collected 100 scenarios for each variance. I would like to run more simulations to have a more accurate data. In addition, I also plan to implement more rollout policies for MCTS to further improve the performance of the algorithm.

References

[1] Auer, P., Cesa-Bianchi, C., and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47: 235-256

[2] Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. 2002. The challenge of poker. *AI* 134(1):201–240.

[3] Chaslot, G., Bakkes, S., Szita I. and Spronck, P. 2008. Monte-Carlo Tree Search: A New Framework for Game AI. 216-217: AAAI Press.

[4] Cowling, P.I., Ward, C.D., and Powley, E.J. 2012. Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. In *IEEE Transactions on Computational Intelligence and AI in Games* 4(4): 241-257

[5] DeNero, J., and Klein, D. 1954. Teaching Introductory Artificial Intelligence with Pac-Man. 1885-1889: AAAI Press.

[6] Gelly, S., and Silver, D. 2011. Monte-Carlo Tree Search and Rapid Action value estimation in computer Go. *Artificial Intelligence* 175(11): 1856-1875.

[7] McLeod, J. Dou Dizhu. 2010. [Online]. Available: https://www.pagat.com/climbing/doudizhu.html

[8] Kocsis, L., and Szepesvari, C. 2006. Bandit based Monte-Carlo Planning. *Machine Learning: ECML 2006*: 282-293.

[9] Russell, S., and Norvig, P. 2003. *Artificial Intelligence, A Modern Approach*, Prentice Hall

[10] Santos, A., Santos, P.A., and Melo, F.S. Monte Carlo Tree Search Experiments in Hearthstone

[11] Shannon, C.E., 1950. Programming a computer for playing chess. *Philosophical Magazine* 41(7): 256-275.

[12] Sriram, S., Vijayarangan, R., and Yuan, X. 2009. Implementing a No-Loss State in the Game of Tic-Tac_toe Using A Customized Decision Tree Algorithm. *2009 International Conference on Information and Automation*: 1211-1216