Fighting the Landlord

Chenlin Liu
Department of Computer Science
University of Virginia
Charlottesville, USA
cl2trg@virginia.edu

Tiancheng Ren
Department of Computer Science
University of Virginia
Charlottesville, USA
tr6cz@virginia.edu

Zetao Wang
Department of Computer Science
University of Virginia
Charlottesville, USA
zw3hk@virginia.edu

Ketian Tu
Department of Computer Science
University of Virginia
Charlottesville, USA
kt9sh@virginia.edu

Xinyu Hou
Department of Computer Science
University of Virginia
Charlottesville, USA
xh4eu@virginia.edu

***Abstract*—Fighting the Landlord is the most popular card game in China. The game requires all players to take all the potential scenarios into consideration and judge the opponents' strategies in advance, so we found it useful to apply AI algorithms to simulate the game for better performance. This paper summarized how we implemented two AI algorithms: Minimax with Alpha-Beta Pruning and Monte-Carlo Tree Search. After implementing algorithms, we collected data from multiple game scenarios, and compared the performances of different algorithms. We found that Minimax with Alpha-Beta Pruning using our third evaluation function, which took both the card value and the longest combo available in hand into account, worked best among all the evaluation functions. We also found that Monte-Carlo Tree Search with a longer simulation time limit has a better performance than that of a shorter simulation time limit.**

Keywords—Fighting the Landlord, Minimax, Alpha-Beta Pruning, Monte-Carlo Tree Search

I. INTRODUCTION

A. Problem Definition and Importance

Fighting the landlord is the most popular card game in China. Although the basic rules are easy to learn, accumulating points and advanced editions make it hard to master, requiring mathematical and strategic thinking and planned execution.

In this game, players need to play and apply several strategies to win the game; three players need to use their strategies to decide to call the card advantage (landlord player) or cooperate with another player (farmer player). Also, the landlord and the farmers have different strategies [1] against their opponents. It is interesting to analyze if we apply three AI agents and record their behavior in this card game to see what strategy they will apply in different situations and figure out the best solution.

On the other hand, the current Fighting the Landlord online population is high (just below the game Mahjong); it is one of China's most popular entertainment games. The demand for the Fighting the Landlord app is high, and users are looking for a more accessible platform to play the game. It is important for companies to apply the Fighting the Landlord AI on their commercial apps to gain more playability to make a more competitive app in the market.

B. Novelty

It is challenging for average human players to remember all the cards played by all opponents and decide whether they should be more aggressive or conservative. Therefore, to have a better performance, we think of implementing a Minimax algorithm with Alpha-beta Pruning and Monte-Carlo Tree Search [2], so that players will know their potential moves and pick the appropriate one.

The innovation lies in that, by adjusting evaluation functions and sampling time, we will collect the data about the win rate and cross-comparing them to improve their performance. The existing literature analyzing game strategies only proposed their own methods. However, our project not only provided particular modified implementations but also focused on comparing the costs of time and win rate of these algorithms.

II. BACKGROUND

A. Game Rule

Fighting the Landlord is a card game with three players [5]. The set of the poker contains fifty-four cards (including two jokers). The rank of the card from highest to lowest is red joker, black joker, 2, A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3. Players will leave three cards on the table, face down, and evenly distribute the remaining cards (seventeen cards for each player.)

Then the players will decide who is the landlord. Once the landlord is determined, the landlord will show the base cards to all players and take them as hand cards. After that, the landlord will get twenty cards, and two farmers will have seventeen cards as their initial hand cards. The farmers will form a team and play

against the Landlord. The player who gets rid of all his or her cards in hand would be the winner of the game.

During each round, the landlord will play his or her cards first, and the farmers will follow to play their cards. All players need to play the same combination as that of the previous player but with a higher rank; players could choose a pass if they do not or cannot follow. If the rest of the players can not follow the last card combination, the player who plays the last card combination could play any card combination as the player's wish.

Table 1 shows the types of combinations a player can play. The complexity increases from top to bottom. The more complex combination a player plays, the harder it is for the other plays to follow. In the algorithm part of our presentation, the evaluation function will be based on these combinations.

TABLE I. TYPES OF CARD COMBINATIONS

Combination Name	**Combination Formation**
Single	Ranking from three (the lowest) up to red joker
Pair	Two cards of the same rank, from three (low) up to two (high). Example: 4,4
Pair Sequence	At least three pairs of consecutive ranks, from 3 up to ace. Twos and jokers cannot be used.
Triplet with single or pair	A triplet (Three cards in the same rank) with any single card or pairs added
Sequence	At least three pairs of consecutive ranks, from 3 up to ace. Twos and jokers cannot be used.
Triplet Sequence	At least two triplets of consecutive ranks from three up to ace.
Triplet Sequence with single or pair	At least two triplets of consecutive ranks from three up to ace. Two extra single cards can be added to each triplet. The attached cards must be different from all the triplets and from each other.
Quadplex with single or pair	Twos and jokers can be attached, but you cannot use both jokers in one quadplex set. Quadplex sets are ranked according to the rank of the quad.
Bomb	Four cards of the same rank. A bomb can beat everything except the super bomb, and a higher ranked bomb can beat a lower-ranked one.
Super Bomb	A pair of jokers. It is the highest combination and beats everything else, including bombs.

III. APPROACH

The approach was divided into two phase: environment setup and algorithm implementation. In the environment setup section, we would talk about how we built the foundation for the agents to play the game. In the algorithm implementation section, we would talk about the two algorithms we used: Minimax with Alpha-Beta Pruning and Monte-Carlo Tree Search.

A. Environment Setup

We initialized our game by first introducing a set of rules and card combinations into the environment. For this purpose, we constructed a card combination class stored within the Rule.py file. This class was responsible for finding all possible card combinations across all card types in the game. This class was crucial as it helped agents find out the possibilities for the next move. To implement this class, we stored combination types (e.g., single, pair, etc.) into the constructor using a set. We then wrote a function that iterated through every card type and appended combinations into corresponding sets. We also wrote a function that created the fifty-four cards, shuffled the deck, and then returned three piles of seventeen cards and three base cards for the landlord.

After setting up the basic rules and card combinations, we then implemented another class called Card.py. Within the file, we constructed a class named “Hand” that had two following functions:

- get_all_combos(hand): The function took in the agent’s initial cards and helped the agent rearrange the cards into different combinations. The function used Counter() imported from the Python collection package, found the intersection between the current card and all possible combinations, and appended these intersections into a successor array.
- get_successors(previous_round,current_hand, combos): This function took three parameters: the cards type and combination of the previous play, the current hand cards available for the agent, and all the possible combinations for the agent. The function returned all possible card combinations that an agent could use considering another agent’s previous play. To find the successors, we found the intersections between hand cards and all the combos. Then, according to the previous play, we appended the possible combinations to the successor array. For example, if the previous play was a bomb, unless we also had a bomb and the card value of the bomb was bigger than the previous play, we couldn’t add the bomb into the successors.

After the game rule and handling of available combinations were made, we then built an input parser class that was useful for reading the user's input for the game. Within the InputParser

class, an argument parser was initialized to read the user's choice of landlord agent, farmer agents, and evaluation function.

After setting up the game, we then started building the environment. We established a board state that included the following fields:

- Discarded Card: Storing cards that were already played by agents. The array was accumulated as more and more cards are played by three agents.
- Previous Play: Storing the play type and corresponding cards that were used by the previous agent. This field was important because it helped the following agent to generate successors.
- Hands: This field was a dictionary which stores cards of three characters in the game. The card received by each player was determined by the deal() function declared in the Rule.py file, and the landlord got the three base cards at the start of the game.
- Combos: This field was a dictionary. It helped each player to find available combinations for its cards by invoking the get_all_combos() function in the Card.py file.
- Turn: This field recorded the current player in the game state. The value of this field changed every time the player switched from one to another.
- Agent Order: This field stored the order of the players in the game. The Landlord was always the first player to play, and two farmers' positions were decided by a random.sample() function of Python.

The board state also includes several functions:

- Get_next_player(): This function found the next player. It extracted the current index of the player, incremented by one, and then returned the player object of the following player using the agent order field.
- Is_win(): This function found out if the game was over. For the landlord to win, the landlord agent should have zero cards in his hand. For the farmers to win, one of two farmer agents must have an empty hand.
- Is_lose(): This function found out if one player lost the game. It took in the id of the agent and returned a Boolean value.

The board state class used functions and classes from other files to construct an environment for starting the game. At last, we declared another class within the file called GameBoard(), which used the information of the environment to make changes to the states. The main function defined in this class was next_state(). The function took in the action of the current agent and returned a new game state. The function updated the discard card field by adding cards played by the action parameter, and it updated the previous play field with the action parameter. Additionally, the function examined whether the game was over by comparing the current number of hand cards with zero. If the number was zero, then the function would end the game. If the game was not over, then the function would return the next game state with these fields updated. The class also introduced a get_action() function. The function took in the agent id and returned all possible card combinations held by that agent. The previously written get_successors() function was used in this function.

B. Algorithm Implementation

In the farmer_agents.py and Landlord_agents.py, we mainly implemented three different agents to apply our algorithms and strategies.

- Reflex Agent: reflex agent only followed basic rules and played randomly. It played only when it had bigger combinations in hand and did not have any strategies.
- Alpha-Beta Agent: Alpha-Beta Agent was an advanced edition of Minimax Agent [6]. It can get information from other players' hands. A farmer agent treated all two farmers as maximizers since they would cooperate with each other to maximize their value and treated the landlord as a minimizer. On the other hand, the landlord would view both two farmers as minimizers and himself as a maximizer. In the function of getting the min value, it would loop all actions for the next state and calculate the min value that was the minimum between the current min value and the max value obtained from the get max value function. Then if the min value was smaller than alpha, we simply returned the min value. Finally, we updated the beta value by comparing the min value and beta and choosing the smaller one. Similarly, we implemented the function of getting max value in the same way only while we were finding the max value, we compared the current max value with the value that got from the function of getting min value.
- MCT Agent: The MCT agent contained two major functions: get action and run algorithm. For the get action, it ran from the current state, recorded the data, and returned the action with the highest win rate. For the algorithm running part, it went through four steps. First, it selected: the agent began at the current state and recursively chose the next action until it reached the bottom node. The selection of the next action would be based on previous recursion data. Second, it expanded: the child node of the leaf node would be created and initialized with a value. Then, it simulated by choosing the next action among given choices and stopped when the game ended or reached its max depth. Finally, it back propagated by updating the visited tree nodes with the simulation results.

There are three evaluations [4] that we used to determine the value of a card combination:

- Card Value: This evaluation function would sum up the card values in the current hand and return the value. Using this function, the agent always played cards with the least value first.
- Longest Combo: This evaluation function [3] would return the value of 20 deducting the length of the hand card. For this function, the agent always played the longest combo available in the current hand.

- Evaluation: This function would return the sum of above two evaluation functions. This meant that the agent took both the card value and longest combo into consideration when planning the next move.

IV. DATA COLLECTION

After implementing the algorithms, we built up a function to simulate different game scenarios and collected data about the game simulations. The simulations made it easier to evaluate the algorithms and see the deficiencies of the implementation.

Since the landlord is the farmers' opponent, we considered the victory of one of the farmers as the shared victory of the farmers. In each game scenario, farmers use the same strategy while the landlord uses another strategy. We planned to evaluate the effectiveness of the two algorithms in each scenario by comparing the landlord's win rate to the farmers' win rate, because they use different algorithms to decide their actions. Before simulating different game scenarios, we ran a simulation in which all players use reflex agents to check whether the win rates between the landlord and the farmers are balanced. If the landlord's win rate and that of the farmers were inherently different, we would have to do some extra operations to make the data representative of the algorithms' effectiveness.

The following scenario was used to test the win rates of the landlord and the farmers using reflex agents:

- Scenario 1: Farmer1, farmer2, and the landlord all used reflex agents.

Scenario 1 was simulated 1000 times. If the ratio of win rate of the landlord and that of the farmers is not approximately one to one, we would do reverse tests for each scenario (e.g., if 50 tests of the farmers using alpha-beta pruning agents with the first evaluation function and the landlord using the reflex agent are conducted, we would run another 50 tests of farmers using reflex agents and the landlord using the alpha-beta pruning agent with the first evaluation function).

The following scenarios compared the performance of alpha-beta pruning agents using different evaluation functions:

- Scenario 2: Farmer1 and farmer2 used alpha-beta pruning agents with the first evaluation function, and the landlord used alpha-beta pruning agent with the second evaluation function.
- Scenario 3: Farmer1 and farmer2 used alpha-beta pruning agents with the first evaluation function, and the landlord used alpha-beta pruning agent with the third evaluation function.
- Scenario 4: Farmer1 and farmer2 used alpha-beta pruning agents with the second evaluation function, and the landlord used alpha-beta pruning agent with the third evaluation function.

The following scenario compared the performance of monte-carlo tree search agents using different simulation time limit (e.g., 5 second and 10 second):

- Scenario 5: Farmer1 and farmer2 used Monte Carlo Tree Search with a 10-second simulation time limit, and the landlord used Monte Carlo Tree Search with a 5-second simulation time limit.

The following scenario compared the performance of monte-carlo tree search agent using 10 second simulation time limit with that of alpha-beta pruning agent with the third evaluation function:

- Scenario 6: Farmer1 and farmer2 used alpha-beta pruning agents with the third evaluation function, and the landlord used Monte Carlo Tree Search with a 10-second simulation time limit.

We realized that the agents would like to play card combinations with lower ranks to follow the previous players. Sometimes in the late state of the game, the game's tendency would simply become compared to which player has more high-rank card, the agent who has more high-rank card will have the high probability to win the game. To solve this problem and make the agent play more similarly to human players, we checked the agents' algorithm and investigated the card evaluation part. We found out that the agents would play the lowest rank card or card combination to ensure those cards would not stay a long time stuck in the agent's hand card in the late game.

Therefore, we decided to adjust the card rating method. In the real game scenario, the player would try to play a high-rank card or card combination to force the opponent to choose a pass, which would ensure the player could play the lowest card in the hand card. In our new evaluation method, we changed the card rating for some cards. The high-rank card would stay the same, we raised the rank of the low-rank card and combination and lowered the middle-rank of the card and combination. These changes would allow the agent to play a higher rank card combination to interrupt and disturb other player's play strategy.

V. RESULTS

Table 2 that farmers had a significantly higher win rate when all players used reflex agents and decided their actions by random. To make sure that the comparison between win rates from players using different algorithms correctly represented the effectiveness of the algorithms, in addition to simulating the game scenarios as planned, performing each game scenario in reverse order was also needed. In other words, we did the simulation of each game scenario where we switched the algorithms used by the farmers and the landlord. In this way, for each algorithm we explored in each scenario, we would get the count of victories from both landlord and farmers. Then, we summed up the count of victories from players using the same algorithm to find the win rate of each algorithm, and this operation offset the effect of the unbalanced win rates between the landlord and the farmers. After that, we assumed that the win rates under different algorithms would be approximately 50% if the effectiveness of the two algorithms explored in a scenario were approximately the same. Since reverse tests were performed, we presented the win rates of the algorithms in each scenario instead of the win rates of the landlord and the farmers.

TABLE II. RESULT OF SCENARIO 1

	Agent Used	Win Counts	Win Rate
Landlord	Reflex	353	35.3%
Farmers	Reflex	647	64.7%

We then examined the performance of the three evaluation functions we designed for alpha-beta pruning. In every scenario, the alpha-beta pruning algorithm is set with a depth limit of 2.

Table 3 shows that players using alpha-beta pruning agents with evaluation function (1) had a significantly higher win rate than players who used alpha-beta pruning agents with evaluation function (2). Players using alpha-beta pruning agents with evaluation function (3) had a slightly higher win rate than players who use alpha-beta pruning agents with evaluation function (1). Lastly, players using alpha-beta pruning agents with evaluation function (3) had a significantly higher win rate than players who used alpha-beta pruning agents with evaluation function (2).

TABLE III. RESULTS OF SCENARIOS 2-4

		Eval Function Used	Win Counts
Regular Test	Landlord	Longest_combo (2)	12
	Farmers	Card_value (1)	38
Reverse Test	Landlord	Card_value (1)	33
	Farmers	Longest_combo (2)	17
Regular Test	Landlord	Evaluation (3)	24
	Farmers	Card_value (1)	26
Reverse Test	Landlord	Card_value (1)	20
	Farmers	Evaluation (3)	30
Regular Test	Landlord	Evaluation (3)	33
	Farmers	Longest_combo (2)	17
Reverse Test	Landlord	Longest_combo (2)	9
	Farmers	Evaluation (3)	41

In Table 4, we saw that the evaluation function (3) had the best performance among the three evaluation functions we had, while the evaluation function (2) had the worst performance among the three. With evaluation function (2), players always chose to play the longest card combinations. This strategy ensured that the number of cards reduced quickly at the beginning of the game, but it usually broke down other shorter combinations with higher ratings at the same time. For this reason, it got harder for the player to follow up the previous players or get rid of the cards with low ratings as the game went on. The evaluation function (3) considered the length of the combinations as well as the rating of the cards. In this way, the players made more complicated decisions for their actions. By choosing the long combinations with low ratings, players saved more higher ranked combinations in hand and got rid of more cards. It is interesting to note that the performance of evaluation function (3) does not improve a lot in comparison with the first evaluation function. The reason might be that the consideration of the length of combinations did not play a significant role in the evaluation when we used evaluation function (3).

TABLE IV. EVALUATION FUNCTIONS COMPARISON

	Win Counts	Win Rate
Card_value (1)	71	71%
Longest_combo (2)	29	29%
Card_value (1)	46	46%
Evaluation (3)	54	54%
Longest_combo (2)	26	26%
Evaluation (3)	74	74%

After the exploration of the performance of alpha-beta pruning with different evaluation functions, we examined the effectiveness of Monte-Carlo Tree Search agents with different simulation time limits. In these scenarios, the Monte-Carlo Tree Search agents always used a random rollout policy.

Table 5 and 6 show that players using MCTS with a 10-second simulation time limit had a slightly higher win rate than the players who used MCTS with a 5-second simulation time limit, and the players using Alpha-Beta Pruning agent with evaluation function (3) had a higher win rate than the players who used MCTS with a 10-second simulation time limit.

TABLE V. RESULTS OF SCENARIOS 5-6

		Eval Func/Algo Used	Win Counts
Regular Test	Landlord	MCTS 5s	26
	Farmers	MCTS 10s	24
Reverse Test	Landlord	MCTS 10s	30
	Farmers	MCTS 5s	20
Regular Test	Landlord	MCTS 10s	12
	Farmers	Evaluation (3)	38
Reverse Test	Landlord	Evaluation (3)	20
	Farmers	MCTS 10s	30

TABLE VI. ALGORITHM COMPARISON

	Win Counts	Win Rate
MCTS 5s	46	46%
MCTS 10s	54	54%
Evaluation (3)	58	58%
MCTS 10s	42	42%

Table 7 and 8 show the result comparing the new card evaluation method and the old card evaluation method.

TABLE VII. RESULTS OF OLD AND NEW CARD RATINGS

		Eval Function Used	**Win Counts**
Regular Test	Landlord	Random	16
	Farmers	New Rating	84
Reverse Test	Landlord	New Rating	70
	Farmers	Random	30
Regular Test	Landlord	Random	17
	Farmers	Old Rating	83
Reverse Test	Landlord	Old Rating	47
	Farmers	Random	53

TABLE VIII. CARD RATINGS COMPARISON

	Win Counts	**Win Rate**
Random	46	23%
New Rating	154	77%
Random	70	35%
Old Rating	130	65%

The win rate of agents using the new rating was higher than that of the old rating. We noticed that the agents' behaviors also changed. The agent would play a higher rank card to stop other agents and put it in an initiative position to lead the card play. This was a strategy commonly used by human players.

According to our data, compared with the farmer players' win rate, it had no significant difference, and the landlord's win rate increased. It meant that this strategy might not be useful for players who would want to cooperate; in other words, if two farmers both played this strategy with the new rating, it would be hard for them to cooperate with each other since they both want to gain the initiative position.

On the other hand, this strategy was more fit with the landlord player because the landlord had the initiative position in the beginning and had more cards, meaning there was a higher probability for the landlord to obtain high-rank cards and complex card combinations.

VI. DICUSSION

A. Limitations and Possible Improvements

There were a few limitations and possible improvements in this study. The first one is our implementation of the Monte-Carlo Tree Search Agent. MCTS with a higher simulation time limit usually had a better performance than one with a lower simulation time limit, because it explored more situations in the simulation. However, when we ran the data for MCT agents with different simulation time limits, we noticed that the difference between the performance of the two MCT agents was not as large as we expected. The reason might be that we did not have time to generate a large number of data for this scenario to see the difference. We supposed that the difference between the performance would be more significant if we compared the win rate of MCT with a 5-second simulation time limit to MCT with a much higher time limit (like a 30-second limit). However, running the simulations of MCTS with a 30-second simulation time limit was very time-consuming. We were not able to execute this plan due to time constraints. We planned to run the simulations of MCTS with a 30-second simulation after submitting this report, and compared its performance with Alpha-Beta Pruning agents using the evaluation function (3).

Second, we only ran each scenario 100 times. The number of simulations for each scenario was not large enough. The data we collected might be affected by random noise and led to incorrect conclusions. In order to gain more confidence in our results, we planned to run more simulations in the future.

Lastly, the simulations took a very long time to run. We did not create more evaluation functions because that would make the run time of the simulations even longer. Instead, we changed the rating of the cards and made the agents' decision-making process more similar to that of human players.

B. Roles and Responsibilites of Each Team Member

To make sure that we were able to deliver all the functionalities that are useful for the game, Fighting the Landlord, we discussed and set up a timeline to keep track of our progress. We were able to build the agents successfully before the class presentation. We also used other useful techniques learned from the class to apply to the project. To keep track of the progress, we set up weekly meetings and communicated effectively through platforms like GroupMe, emails, and Zoom.

We met on every Monday morning to talk about the tasks we needed to accomplish by the end of the week and met again on Friday night to see if we had achieved our goal. Different members were also responsible for different aspects for the project. Xinyu and Chenlin put their efforts on setting up and managing the repository used for our collaboration and analyzed the algorithms to the result in this report. Also, Tiancheng and Zetao focused on summarizing result data from our implement and delivering them to the final report. Ketian concluded the important functionality method we applied on the project.

At the same time, everyone participated in the development of the Agent as well as other related functionalities, and we communicated with each other with respect. Ketian and Tiancheng were responsible for implementing the game environment and ensuring the stability of the game performance, they also worked on testing the algorithms and giving feedback to the team to figure out a better solution or implementation. Zetao focused on collecting and analyzing the data, developing the optimization of the agent, and implementing the rule of the game. Xinyu and Chenlin implemented the agent's structure and evaluation function, they compared different algorithms and picked up the most efficient algorithm fit with our game and implemented them.

REFERENCES

[1] Bošanský, Branislav, Lisý, Viliam, Lanctot, Marc, Čermák, Jiří, Winands, Mark H.M. *Algorithms for computing strategies in two-player simultaneous move games.* Artificial Intelligence, 00043702, Aug2016, Vol. 237*).*

[2] Chaudhry, Muhammad Umar, Jee-Hyong Lee, *MOTiFS: Monte Carlo Tree Search Based Feature Selection*, Entropy, 10994300, May2018, Vol. 20, Issue 5I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[3] Lorentz, Richard, *Using evaluation functions in Monte-Carlo Tree Search.* Theoretical Computer Science, 03043975, Sep2016, Vol. 644.

[4] Matsuzaki, Kiminori, Kitamura, Naoki, International Computer Games Association Journal, *Do evaluation functions really improve Monte-Carlo tree search?*, 13896911, 2018, Vol. 40, Issue 3.

[5] McLeod. J,"Dou Dizhu (斗地主)," *Rules of Card Games: Dou Dizhu*. [Online]. Available: https://www.pagat.com/climbing/doudizhu.html. [Accessed: 04-Dec-2020].

[6] Wibowo H. A., "Create AI for Your Own Board Game From Scratch-Minimax-Part 2," *Medium*, 28-Nov-2018. [Online]. Available: https://towardsdatascience.com/create-ai-for-your-own-board-game-from-scratch-minimax-part-2-517e1c1e3362. [Accessed: 04-Dec-2020].