Fighting the Landlord Game AI with Minimax Search

Zizhuo Wang
SIST, ShanghaiTech University
wangzzh@shanghaitech.edu.cn

Jinrui Wang
SIST, ShanghaiTech University
wangjr@shanghaitech.edu.cn

Xiner Xu
SIST, ShanghaiTech University
xuxe@shanghaitech.edu.cn

Suan Xia
SIST, ShanghaiTech University
xiasa@shanghaitech.edu.cn

Wenyi Li
SIST, ShanghaiTech University
liwy1@shanghaitech.edu.cn

Abstract—Fighting the landlord is one of the most famous poker games in China, which is a typically game featuring asymmetric information and randomness. This paper aims at creating an AI to play this poker game well. While implementing, the prime algorithm being used is the Minimax Search with Alpha-Beta pruning. At the end, though the trained AI is able to play against human players and have a winning rate over fifty percent, there are still much to do with the algorithm efficiency and the AI's ability to handle tough situations.

Index Terms—AI, poker, Minimax search, Alpha-Beta pruning

I. Introduction

This paper is aimed at creating a Fighting the Landlord AI. Unlike the pacman agent which is involved in the class, the AI is playing the game under a stochastic environment since the cards are delivered randomly. To solve these complex situations, the AI should be able to learn the value of different moves and make the optimal choice. In this way, implementing the Minimax Search on the AI can be a good method.

The Minimax Search is an algorithm of a state-space search tree. It compute each node's minimax value, which means the best achievable utility against a rational (optimal) adversary. To apply this algorithm, each player should learn the possible moves and the states of other players. Apparently, considering all possible moves is infeasible due to limited resources, so here we introduce the Alpha-Beta pruning to enhance performance.

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games. Though there are three agents in the Fight the Landlord game, it can be simplified to two because two farmers can be seen as one agent with the same goal. For the landlord agent, the landlord is a maximizer and the two farmers are minimizers. On the contrary, For the farmer agent, two farmers are the maximizers and the landlord is the maximizer. By implementing Alpha-Beta pruning, the algorithm will be simplified and have better performance.

II. Literature Review and Background

A. The basic game rules of Fighting the Landlord

Fighting the landlord which is also known as Dou Di Zhu is a famous Chinese poker game. The game involves three players. One plays as the landlord ,obtaining 20 cards and the other two are farmers obtain 17 cards who shall fight against the landlord together. The game uses a full poker deck (54 cards). The rank of cards are as followed: 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, 1, 2, black joker, red joker. As the game begin, the landlord goes first. There are many types of cards admitted in this game, such as pair, trio pair, airplane, bomb and so on. Only the higher rank of the same card combination can take over the smaller one except for the bomb, which can be played to beat every play and take the initiative except higher rank bomb. The side whoever runs out of cards first will win the game. Note that the cooperation between two farmers are important to beat the landlord.

B. Minimax algorithm with Alpha–beta pruning

Minimax algorithm is a tree search algorithm that is mainly used in adversarial games that usually refers to 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.[2] A Minimax algorithm consists of a maximizer and minimizer. The maximizer always tries to select the action that returns maximum utility value and the minimizer always tries to select the action that returns minimum utility value. To make it short, Minimax is an algorithm that finds the least value among the largest possibilities of the failure. There are some notes of Minimax. Firstly, Minimax is a kind of pessimistic algorithm. The adversary is supposed to be smart enough to lead its adversary to the direction of least rewards. Secondly, Minimax does not find the best solution in theory. Since the best solution in theory depends on the adversary's making fault, a smart enough adversary will not allow this situation occurring. All in all, decisions are made according to the best choice in all

the worst cases. 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.

The simply generated Minimax tree is too large to deal with, so Alpha-Beta pruning is applied to cut off the time complexity from $O(b^m)$ to $O(b^{\frac{m}{2}})$. When it comes to the general configuration, let's consider the MIN version. To compute the min-value at some node n, n's children are looped over and n's estimate is decreasing. Then let a be the best value that can get at any choice point along the current path from the root. If n becomes worse than a, then n's other children can be neglected because for the finally chosen n, the nodes along the path must have the value of n. For the MAX version, it is symmetric to the MIN version.

Reference [1] contains a parallel version for minimax implemented with GPU. It hightlights GPUs are very effective at speeding up tree search when branching factors stay constant, and performs poorly with varying branching factor.

III. Implementation

A. Infrastructure

For the ease of programming and porting, Python 3 is chosen to write the program. A trivial portion of project 2 from CS181 is taken to support the user input. A text based UI is presented for manual agents. The code is deliberated structured to be extensible, allowing a fourth player to join if needed be, since a certain variation of the game rules consists of one landlord and three farmer.

B. Generating actions and game states

Unlike course project pacman, generating actions is actually nothing trivial. It is easy to generate everything given enough time, but to meet performance targets care must be taken to prevent unnecessary overhead.

Three principles are followed throughout this portion of code:

- Use numpy and its efficient ndarray to store card combinations.
- Use caching wherever possible.
- Minimize GC overhead.

numpy's ndarray is more memory efficient than python's list. By following this principle along, a 21.2% memory usage decrease is observed. The added benefit of having an elegant API towards vector calculus also smooths the development.

A special variation of LRU caches are used extensively throughout this portion of the project. As the minimax algorithm goes in rounds, some part of the generated action and game states can be reused in the next round. This both reduces GC overhead and reduced time wasted on generating identical items twice (or more, depending on the depth used).

TABLE I
Card to card value mapping

Card	3	4	5	6	7	8	9	10
Value	-7	-6	-5	-4	-3	-2	-1	0
Card	J	Q	K	A	2	BlackJoker	RedJoker	
Value	1	2	3	4	5	6	7	

Extra attention is directed towards temporaries. By reasonably reducing temporaries, GC overhead is reduced to minimal.

C. Minimax with alpha beta pruning

The traditional minimax is improved with parallelism. A pool of worker thread is employed to walk the tree. They will start at each immediate successor of root node. When they finish, they will notify other workers about the new alpha (or beta). The other works will look at his current alpha (or beta) and update it if needed. They will the cut off immediate if the new alpha (or beta) is bigger (or smaller) than current beta (or alpha).

D. Utility function

Four utility functions are implemented

- $$Utility_P^0 = -len(Hand_P)$$

  This trivial utility function is used as a baseline of comparison for other utility functions. It simply measures the count of remaining cards, with no regards to its actual value.
  In effect, this utility function incentivize the player to play the longest combination first.

- $$Utility_P^1 = \sum_i^{len(Hand_P)} Value(Hand_P[i])$$

  This utility function is an improvement over the trivial utility function. It now calculates the actual value of cards.
  Since card A and 2 are considered bigger than card 3 through K, an mapping is used to find the value of each card as shown in Table I.
  In effect, this utility function incentivize the player to play the greatest card first.

- $$Utility_P^2 = \alpha\ Utility_P^0 + \beta\ Utility_P^1$$

  This utility function is a combination of above to functions. The two constants α and β are magic numbers found by trail and error that performs the best.
  In effect, this utility function incentivize the player to play the greatest card and longest combination first.

- $$Utility_P^3 = \frac{1}{\sum_{comb} Value(comb)}$$

TABLE II
combination type multiplier, n is number of cards

Combination Type	Single	Pair	Trio	Chain
Value	1	3	5	$3n$
Card	Pairs Chain	Bomb	King Bomb	
Value	$6n$	$12n$	$14n$	

TABLE III
Utility function comparisons

Index	Position	Win Count
0 [*]	Farmer	56
0 [*]	Landlord	44
1	Farmer	64
1	Landlord	55
2	Farmer	78
2	Landlord	67
3	Farmer	58
3	Landlord	50

[*]: This is baseline win rate

This utility function is the reciprocal of value of all possible card combinations.
The value of a combination is the value of the smallest card multiplied by a combination type specific constant as shown in Table II. These constants are given by experience from human players and some trial and error.
In effect, this utility function incentivize the player to play the card with least value first.

IV. Evaluation

A. Utility functions

Utility function 1 through 3 are used to play against a minimax agent using baseline utility function. Each test is ran 100 times to minimize the effect of randomness. The result is shown in Table III.

This test shows that utility 1 is better than baseline but worse than utility 2. This is as expected as 2 is a combination of 1 and 2.

Utility 3 is somewhat the same as baseline. After inspecting the random tests, it is found that it makes player heavily biased towards playing every card in single.

B. Against human

The agent is first evaluated for its success rate as shown in Figure 1.

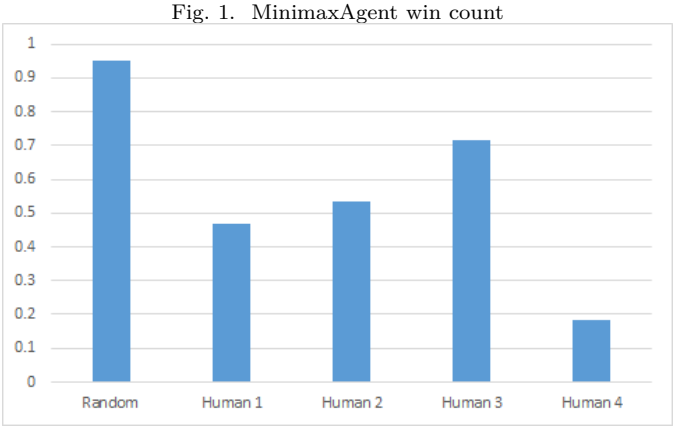
C. Performance

Throughout all 687 recorded tests, the average time per step of is 2713ms for fully automated tests and 3302ms for tests that involves one human player.

V. Conclusion

The limited number of human players and varying skills of human player make the evaluation rather difficult, but it does shows that minimax agent is an average level player.

Fig. 1. MinimaxAgent win count



The magic multiplier chosen in utility function 2 should be able to be trained using reinforcement learning. However, the poor performance made training time too long for this technique to be plausible.

On performance wise, the agent is suboptimal. The target average time per step is no more than 1500 ms, whereas the best average time per step in each game is 1623ms. On retrospect, the implementing the same algorithm in C may be able to boost the performance slightly. This would also remove the need to care about GC and make caching magnitudes easier for state and action generating.

References

[1] Kamil Rocki and Reiji Suda. “Parallel Minimax Tree Searching on GPU”. In: Parallel Processing and Applied Mathematics. Ed. by Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 449–456. isbn: 978-3-642-14390-8.

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