[[1]](#footnote-0)

Chinese Checker AI Game

Spades (2130034050)

Ramona (2130034023)

Doris (2130034021)

Aslina (2130034006)

Ferne (2130034033)

***Abstract*— This paper looks into a board game named Chinese Checker and use artificial intelligence (AI) algorithms to implement the game on M2M (machine to machine) and H2M (human to machine). In addition, in H2M, there are four levels for users to choose. In this project, we developed an AI-based Chinese checkers game with four algorithms: Minmax, Greedy, Alpha-Beta Pruning, and a self-implemented Residual Convolutional Neural Network (CNN) algorithm. The Minmax algorithm is used in zero-sum games, aiming to minimize damage and maximize winning chances by considering the opponent to be perfect. Greedy algorithm finds the optimal result step by step but may get trapped in local best solutions. Alpha-Beta Pruning improves Minmax by reducing search space, but premature termination and search order sensitivity are challenges. The self-implemented Residual CNN algorithm predicts opponent moves and winning probabilities using a neural network. Implementation involved defining basic game operations and implementing algorithms with specific functions and models. Algorithm strength was measured based on the number of wins and search time. A GUI interface enabled user interaction, allowing users to select difficulty levels and play against the AI.Result analysis showed that Alpha-Beta Pruning was slower, while Greedy was the fastest algorithm. The self-implemented CNN algorithm was the easiest for beginners.This project provides an AI Chinese checkers game with different difficulty levels, utilizing various algorithms to challenge players.**

*Index Terms*— Chinese Checker, AI, algorithms

# INTRODUCTION

Chinese Checkers is a board game that can be played by two to six players. It was invented in 1893 and has since then been released by various publishers under different names. Chinese Checkers is played on a star-shaped board. The most common board contains 121 fields, where each player starts with ten pieces. The goal of each player is to move the own pieces to the own base at the other side of the board. Pieces may move to one of the adjacent squares or they may jump over another piece to an empty field. A player may also make multiple jumps with one piece in one turn. It is possible to create a setup that allows pieces to jump over a large distance. The first player who manages to fill the home base wins the game. To avoid blocking behavior, the player wins the game when the home base is filled and the player owns at least one of the pieces in the home base.

In this project, we developed an AI-based Chinese checkers game and implemented four algorithms: Minmax, Greedy, Alpha-Beta Pruning, and a self-implemented Residual Convolutional Neural Network (CNN) algorithm. These algorithms were chosen to provide different levels of difficulty and challenge for players.

The Minmax algorithm is a basic game algorithm commonly used in zero-sum games. It assumes the opponent to be perfect and aims to minimize potential damage and maximize winning chances. By considering all possible moves and their outcomes, Minmax identifies the best possible move based on worst-case scenarios.

The Greedy algorithm, also known as the mountaineering algorithm, aims to find the optimal result step by step. It makes locally optimal choices at each step, without considering the overall situation. While it may not guarantee the optimal solution, it performs well under strict conditions.The Alpha-Beta Pruning algorithm is an improvement over Minmax, using pruning techniques to reduce the search space. By eliminating branches that are not promising, Alpha-Beta Pruning saves computational resources and time. However, it may terminate prematurely and heavily relies on the order of search.

The self-implemented Residual CNN algorithm utilizes a convolutional neural network model. It predicts opponent moves and evaluates the likelihood of winning based on the current board state. This algorithm combines the power of neural networks with the game's dynamics to make informed predictions and decisions.

In terms of implementation, the game's basic operations were defined, including rules, move functions, and board updates. Each algorithm was implemented with its specific functions and models. To measure algorithm strength, metrics such as the number of wins and search time were recorded and analyzed. This helped determine the effectiveness and efficiency of each algorithm.

A GUI interface was designed to facilitate user interaction. Players could select the difficulty level, and the AI opponent would employ the corresponding algorithm. The interface provided a seamless experience, allowing users to play against the AI and enjoy the game.

Overall, this project offers an AI Chinese checkers game with varying difficulty levels, thanks to the inclusion of different algorithms. It enables players to challenge themselves against AI opponents with diverse strategies and decision-making capabilities. The implementation of a user-friendly interface enhances the gaming experience and encourages user engagement.

# Literature Review

Chinese Checkers, also known as Halma, is a strategy board game played on a star-shaped board with 121 holes. The objective is to be the first to move all of one's pieces from one's home triangle to the opposite triangle. The game can be played by two, three, four, or six players.

In terms of algorithms, various approaches have been developed to create computer programs capable of playing Chinese Checkers effectively. One common algorithm used is the Minimax search with Alpha-Beta pruning. [1] This algorithm explores the game tree by recursively evaluating possible moves and selecting the best move based on a minimax strategy. Alpha-Beta pruning helps to reduce the number of unnecessary evaluations by eliminating branches that are guaranteed to be worse than previously examined moves.

Another algorithm often applied is Monte Carlo Tree Search (MCTS). [2] MCTS uses random simulations to estimate the value of each possible move and then expands the search tree accordingly. By balancing exploration and exploitation, MCTS can effectively handle the large branching factor of Chinese Checkers.

Several implementations of Chinese Checkers algorithms can be found in programming languages like Python, Java, and C++. [3] These implementations typically combine search algorithms with heuristics that evaluate the current game state. The heuristics consider factors such as piece positions, distance to the target triangle, and potential moves to make informed decisions.

Overall, algorithms for Chinese Checkers aim to balance exploration and exploitation while efficiently searching the game tree, allowing computer programs to play the game at a high level and provide challenging gameplay.

## Min-Max search and Alpha-Beta pruning

The research [4] presents a game system for checkers based on research in computer game algorithms. It introduces the Min-Max search tree algorithm and utilizes Alpha-Beta pruning to optimize the game tree search. The system addresses game formation, depth valuation, and time-consuming search issues, resulting in improved search efficiency and performance. The experimental results demonstrate the feasibility, practicality, and effectiveness of the game search model.

The paper also discusses the representation of the chessboard using a two-dimensional array and the generation of steps through layout formation. Valuation is performed using static and dynamic evaluation functions. The search algorithms employed include Min-Max search and Alpha-Beta pruning. Additionally, the paper briefly mentions the introduction of self-learning techniques, such as unsupervised learning using a neural network. Overall, the paper provides insights into the design and application of computer game algorithms for checkers.

## Heuristics, Monte Carlo Tree Search, and Deep Reinforcement Learning

This article [5] presents an innovative approach to tackle the challenges of Chinese Checkers using a combination of heuristics, Monte Carlo Tree Search (MCTS), and deep reinforcement learning. Unlike Chess or Go, Chinese Checkers poses unique difficulties due to the indefinite presence of checkers on the board and the allowance of repetitions and backward movements.

The authors address these challenges by proposing a two-stage training pipeline. In the first stage, the agent is guided and initialized using a greedy heuristic, enabling it to quickly learn basic strategies and reduce the depth of search trees. The second stage focuses on reinforcement learning, where the agent generates self-play games using MCTS and trains a deep residual convolutional neural network (CNN) to improve its performance.

Experimental results demonstrate the effectiveness of this approach. The agent performs competently in various scenarios and reaches the level of experienced human players. In comparison to alternative methods, such as MCTS-guided self-play reinforcement learning and Deep Q-Learning, the proposed approach consistently outperforms them.

This research sheds light on the application of machine learning techniques to Chinese Checkers, an area that has received limited attention thus far. By combining heuristics, MCTS, and deep reinforcement learning, the authors have developed a Chinese Checkers agent without relying on human gameplay data.

In summary, this study contributes to the understanding of Chinese Checkers by introducing an effective approach that leverages heuristics, MCTS, and deep reinforcement learning. The results highlight the agent's competency and demonstrate its potential for advancing the field of machine learning in traditional board games.

## N-Grams and the Last-Good-Reply Policy Applied in General Game Playing

This article [6] focuses on the improvements made to the application of the greedy algorithm in General Game Playing (GGP). GGP aims to develop programs that can play various games at an expert level without specific game rules.

The most successful GGP programs currently employ Monte Carlo Tree Search (MCTS) algorithm based on simulations. The performance of MCTS heavily relies on the simulation strategy employed. The article proposes enhanced simulation strategies and implements them in CADIAPLAYER, a GGP agent program that won the International GGP Competition in 2007 and 2008.

The improvements primarily involve two aspects: firstly, the authors found that a simple greedy exploration strategy outperforms the current SoftMax-based Gibbs measure used by CADIAPLAYER during simulations; secondly, a promising framework for learning N-gram-based move sequences is introduced.

These enhancements significantly improve CADIAPLAYER's performance, achieving an average win rate of around 70% in a test suite consisting of five different two-player turn-based games. These improved strategies are also validated in multiplayer and simultaneous-move games. Additionally, the article experiments with the Best Response to Opponent Strategy (LGRP) and tests the integration of N-grams into LGRP. LGRP has already proven successful in Go programs, and the authors demonstrate its potential in GGP as well.

# Methodology

## Minmax Algorithm

We applied the Minmax algorithm in our Chinese checker game. Minmax is one of the basic game algorithms and is usually employed in zero-sum game in which two sides contest and two sides will execute their steps in turn, if one side win, then another side fails, like board games. Minmax is a kind of pessimistic algorithm, it always considers its opponent to be perfect and will lead it to the worst situation, then all it can do is to minimize the damage and manage to be in the best situation among all worst ones. Therefore, if its rival does not reach perfect in one decision, then the result will be better than the predicted worst. The basic principle of the algorithm is to find and reach the best situation in worst cases, which means Minmax algorithm always consider its opponent to win, and its goal is to makes the probability of its opponent’s winning the lowest. The efficiency is not high since it checks all possibilities in the situation to find the optimal one.

## Greedy Algorithm

The second algorithm we applied in our game is Greedy algorithm, which is also called mountaineering algorithm because of its working principles. Its goal is to get the optimal result step by step, just like climbing a mountain, and it finds the global best solution by finding local best solutions, namely, it does the current optimal decision each time it chooses. However, because it always chooses the local best solution rather than consider the overall situation and each time the choice influences the final result without chances to withdraw, the situation may be ‘trapped’ into local best solutions, which means Greedy algorithm cannot guarantee the results to be optimal. Nonetheless, it will work well if strict conditions are defined.

The main objective of the Greedy algorithm is to reach the overall best solution by iteratively making locally optimal choices. At each step, the algorithm evaluates the available options and selects the one that appears to be the most beneficial or promising, solely based on the information known at that particular moment. It does not consider the future implications of that decision or explore alternative paths extensively.

The strength of the Greedy algorithm lies in its simplicity and efficiency. It often performs well when applied to problems that exhibit the greedy-choice property, which means that making a locally optimal choice at each step will eventually lead to the global optimum. In such cases, the Greedy algorithm can provide a reasonably good solution with significantly reduced computational complexity compared to other optimization techniques.

## Alpha-Beta Pruning Algorithm

The third algorithm we applied in our Chinese checker game is Alpha-Beta pruning algorithm. It is in fact a promotion version of Minmax algorithm, whose efficiency is greatly increased because of the application of pruning, which reduces the search space and therefore, less resources and time are needed for searching. In the algorithm, alpha is the value that the best solution our side can find, beta is the worst situation we can make our opponent in. If the current step we take is less than or equal to alpha, or is bigger than or equal to beta, then the choice is abandoned, since it is not good enough, and this process is called pruning. If the current step we take is bigger than alpha and less than beta, then the step is available and useful, it will be added to consideration. However, the pruning method may also lead to the premature termination of the whole program, and as a result, the searching stops before the optimal solution is found. Besides, the algorithm depends severely on the order of the search, if the order is not appropriate, the result could be much troublesome and disappointing.

## Residual Convolutional Neural Network Algorithm

The fourth algorithm we apply is the Residual Convolutional Neural Network, which is also known as Residual CNN, it is an algorithm we wrote on our own after use some neural networks for reference and improved based on them. It mainly applied the convolutional neural network to do the calculation and prediction, and it predicts the probability of a movement or step the opponent will take and the evaluation of whether we will win. The algorithm forecasts the best next step and when invoked, the best next step is carried out then.、

It's important to note that while CNNs are commonly used for image-related tasks, they can also be adapted and applied to other types of data, such as time series, text, or even graph data. The underlying principles of convolution and feature extraction remain similar, but the architecture and input representations may vary based on the specific problem domain.

Overall, CNNs have proven to be highly effective in various domains, including computer vision, natural language processing, and even games. Their ability to automatically learn hierarchical representations from data makes them a powerful tool for prediction, classification, and decision-making tasks.

# Implementation

We have a System implementation process for project, our implementation is divided to five parts, we will show some detail related to them

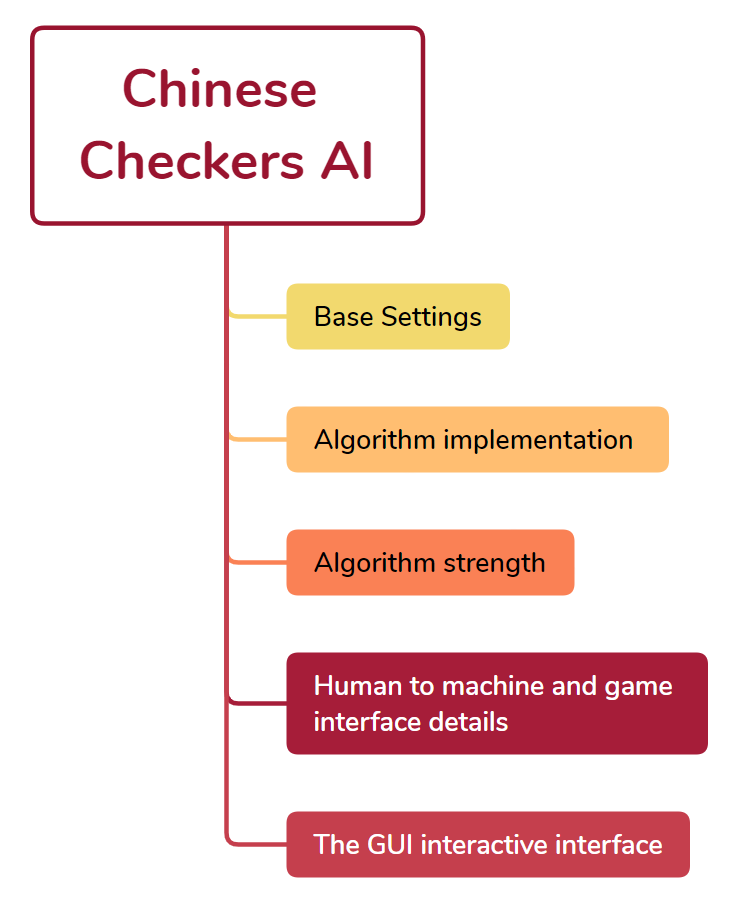


Fig. 1. System implementation process

## Base settings

With Chinese checkers as the theme, our group designed an AI intelligent checkers game, which can realize the purpose of human-computer interaction. The first task to complete this project is to implement some of the basic operations of the checkers game, including allowing the machine to fully understand the rules of the game and let them play according to the rules. Rules are at the heart of a game, and the game revolves around them. So, our project system implementation must first implement the rules of checkers. In addition to this, some of the most basic move, design, and update functions are essential. In this process, we need to define functions to set the initial position, target position, and unreasonable target position of each player's piece. Next, we need functions to update the position of each player's pieces. In order to ensure that the movement of the chess pieces is in line with the rules of checkers, we also need to set up a series of functions to find the situation of other pieces around the pieces that need to be moved, so that we can define the movement restrictions of the pieces according to the rules of the checkers game, including the operation of moving and jumping, through the restrictions, we can find all the positions that each piece can reach (in line with the rules). It can also determine whether a piece can move.

## Algorithm implementation

In our AI checkers game, we used four algorithms, which are alphabeta, greedy, minmax and an algorithm that trained and defined by our group. The usefulness of these four algorithms is to enable the machine to calculate the optimal solution of the movement of the chess piece. because different algorithms use different calculation methods, and the depth of their search may also be different, although the aim is similar, the effect is different. So based on this feature, for the whole game, we set four difficulties, when people and machines play against each other, they will face four different algorithms in different difficulty levels, and we need to implement these four different algorithms.

In the implementation of the first three algorithms, we call the basic function that has been defined in the first module. According to the different principles of each algorithm, you can find the optimal movement with code and the model defined by each algorithm. In subsequent implementations, we will call the completed algorithms to implement other modules and functions.

The last algorithm uses our self-implemented algorithm. it defines a class of models based on residual convolutional neural networks (Residual CNN), which can be used to predict the action and value of checkers. In order to realize this algorithm, we give the computer a state of the board (whether each coordinate has pieces, and which player one pieces are belong to). Then by using the Residual CNN model and some self-defined functions, we can get two values, one is to predict the probability of the piece action in the board state, the other is to predict the estimated value in the board state. Finally, we can get the optimal movement based on the two values by calculate.

## Algorithm strength

When we implement all four algorithms, we need to get their respective difficulty levels, and for this we need to use some metrics to measure, the indicators we use in the project are the number of wins and the time used for search.

For the first indicators, we realize machine-to-machine battles to achieve the goal. we define a class to implement the game which is machine-to-machine. In the previous steps, we have set the rules for the entire game to move the pieces to a reasonable position, in addition to implementing four different algorithms on how to find the optimal move. So, in this class that implements machine-to-machine battle, most of the functions we can call directly have been defined before. While calling the function, we first determine the state of the initial position (according to the coordinates and number; 1-6 means have pieces, 0 means no pieces). Next, we need to define the function to correspond different position numbers (1-6) to different algorithms. For example, position 135 corresponds to greedy, position 246 corresponds to minmax. Then output of the best movement found by call function. Finally, define a function in this class that can determine who wins and return the result information. At this point, all the functions required for machine-to-machine battles have been defined. We run them in the master file and set the number of rounds to 100. After the end, compare the number of winning rounds of different algorithms, the more wins, the higher the strength of the algorithm.

In addition to the above methods, we also record the time it takes for different algorithms at different depths to complete a search to measure algorithm strength. Specific results and analyses will be presented in subsequent result analyses.

## Human to machine and game interface details

This is the most important and crucial step in the game of AI checkers, and this module can help realize the game confrontation between man and machine. Since there are people involved, we need to first set the player position for the person and the machine to be able to interact. Then similar to machine-to-machine confrontation, we also define a class. This class can implement the following functions:

each game has a user and a machine in other five positions, we start the game with the player in the position which is the white chess piece. When the machine plays, call the functions defined earlier in other classes to calculate the optimal movement, and move pieces to the same position as they were found.

When the user operates the piece, they can move the mouse to the piece the user want to move. than the computer displays where the piece can reach. When the mouse clicks on the position that is not marked, the computer will return the prompt of "cannot move”. when the mouse clicks the marked position, the computer can obtain and record the coordinates corresponding to the position, so that the chess pieces move to this place. It should be noted that we have defined a function that allows the user to operate when the machine is paused.

## The GUI interactive interface

We use Pychame to define a GUI interface to enable interaction between the user and the computer. When the user wants to play a checkers game, the user can see the initial interface after running the main program (Fig. 2.). There are four difficulty selection buttons in the initial screen, corresponding to four checkers opponents of different difficulty. After the user selects the difficulty, The computer can connect to the algorithm corresponding to the difficulty based on whether the button is pressed. When entering the game interface, the game begins, at this time the user corresponds to the lowest character, the user needs to face five robot opponents for checkers, and the game will end when the first player reaches the end. At this time, we set up a GUI interface, this interface is used to display the result of the game. when the user clicks the end button, the GUI interface will automatically jump to the interface of selecting the difficulty, so that the user can adjust the difficulty and restart the game.

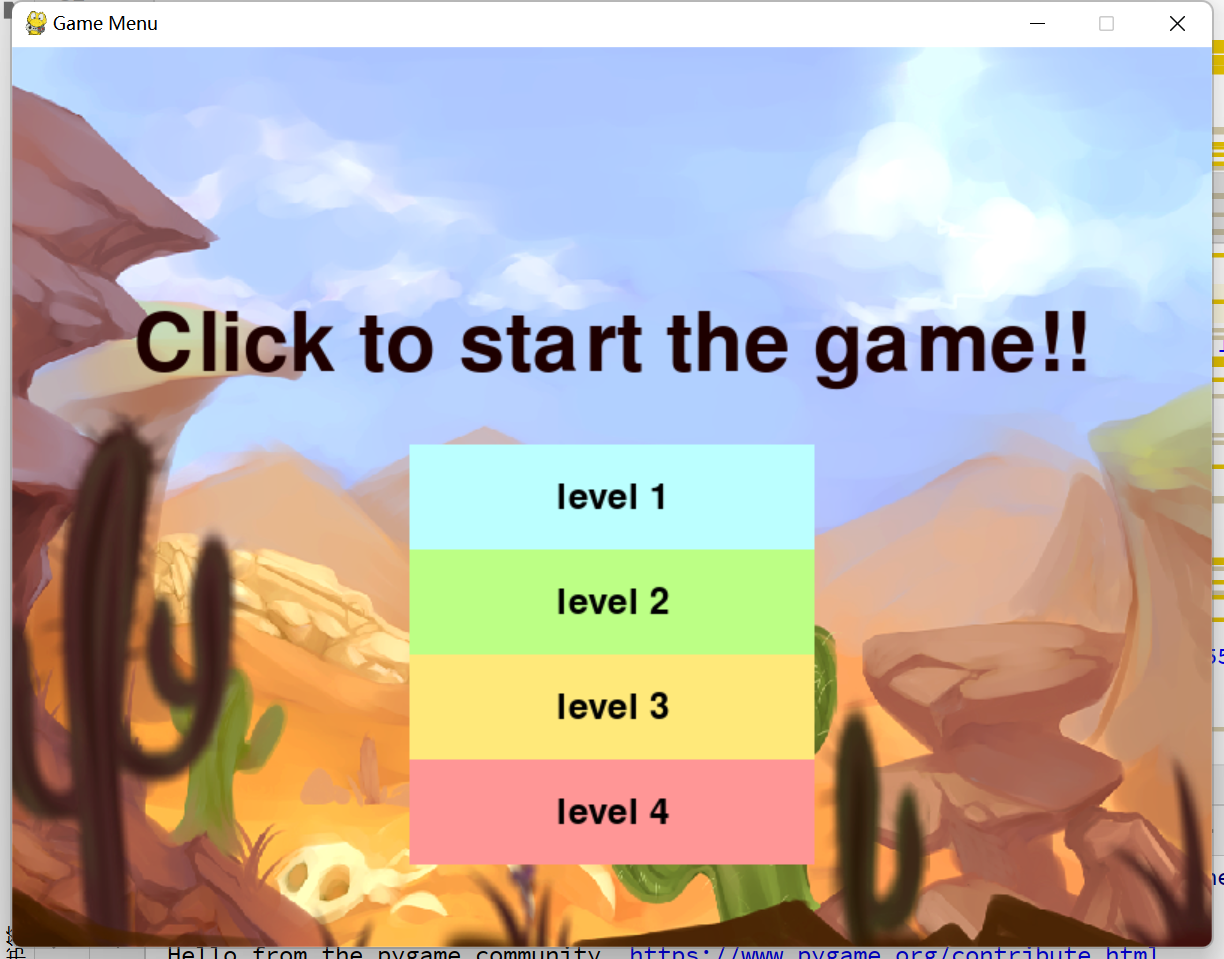


Fig. 2. Initial Interface

# Result Analysis

In M2M, we added timer to count the time spent by each kind of algorithms including greedy, minmax, and alphabeta. The timer is in file engine.py and we count each player and their steps’ time and put them into different list. At the end of a game, we use sum() function to sum up times then divide the number of round of the game. So that the average time of each player (one player use one algorithm, some players use the same algorithm but with a little difference in parameters) We can draw the conclusion that alphabeta is slower than greedy and minmax. Greedy is faster than minmax. Greedy is the fastest algorithm. And with the number of depth growing, the algorithm will become much slower.

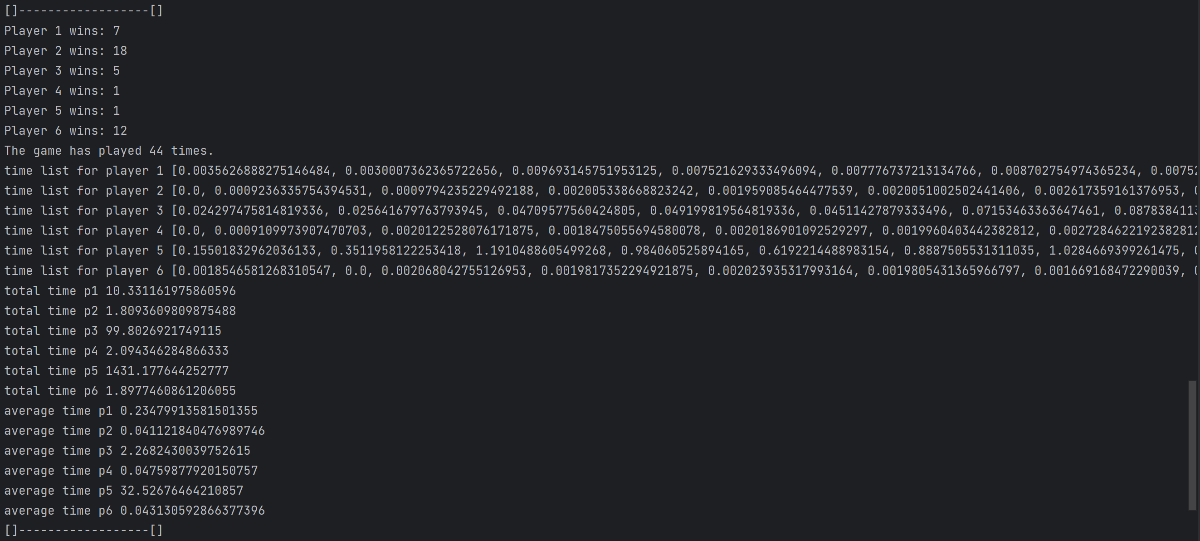


Fig. 3. 135 alphabeta; 246 greedy

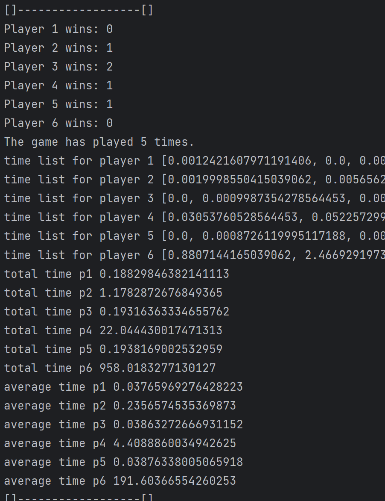


Fig. 4. 135 Greedy; 246 Minmax

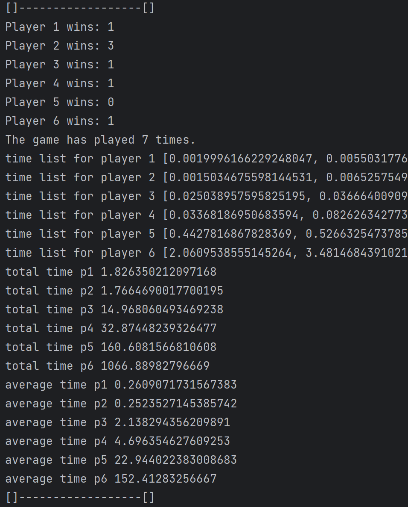


Fig. 5. 135 Alphabeta; 246 Minmax

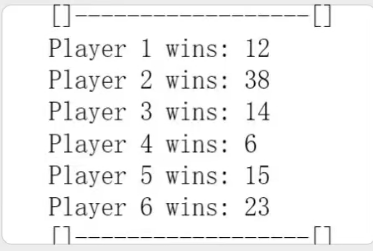


Fig. 5. Greedy (won 67) V.S. Alphabeta(won 41)

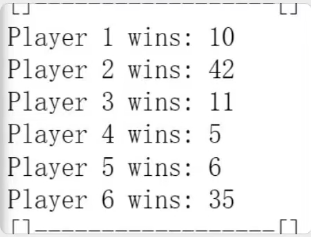


Fig. 6. Greedy (won 27) V.S. Minmax (won 82)

For the number of M2M winning times, each two algorithms we tested about 100 round of games. At the end of a game, winning counter will record each player’s score.

Accoding to figure 4 & 5, minmax is smarter than greedy, and greedy is smarter than alphabeta.

According to figure 6 & 7, minmax is smarter than greedy, and greedy is smarter than alphabeta.

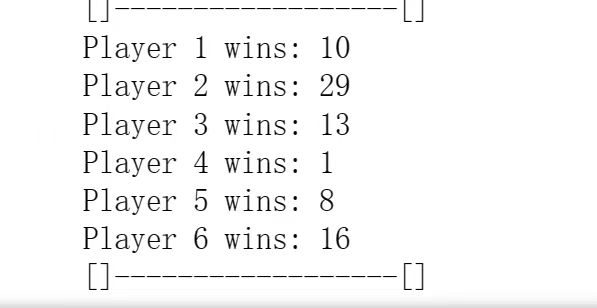


Fig. 7. 135 CNN (won28); 246 minmax：(won46)

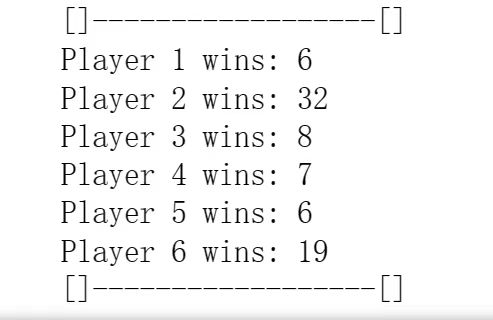


Fig. 8. 135cnn: 20; 246greedy:58

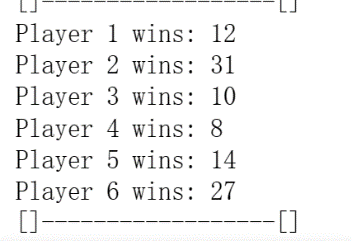


Fig. 9. 135 CNN; 246 Alphabeta

We added out own algorithm CNN. We can draw the conclusion that, CNN is the easiest algorithm for Chinese Checker beginners. So, the level of difficulty we settle is CNN, alphabeta, greedy, minmax.

The reason why CNN does not have a good performance is that the amount of our dataset is not large enough, and it is expensive to train and interpret model. Because the interpretability is limited, so it is difficult for us to figure out the knowledge that network have learned [7]

# ­­Conclusion

In conclusion, our project focused on developing an AI Chinese checkers game with multiple difficulty levels, achieved through the implementation of four distinct algorithms: Minmax, Greedy, Alpha-Beta Pruning, and a self-implemented Residual Convolutional Neural Network (CNN) algorithm. Each algorithm brings a unique approach to decision-making, providing players with different challenges and gameplay experiences.

The Minmax algorithm, as a fundamental game algorithm, aims to minimize potential damage by considering worst-case scenarios. It assumes the opponent to be perfect and strategically selects moves that minimize the opponent's chances of winning. Although it checks all possibilities to find the optimal move, its efficiency is not high due to exhaustive search.

The Greedy algorithm, inspired by mountaineering, focuses on finding locally optimal solutions at each step. It gradually climbs towards the best outcome but may fall into local best solutions. While not guaranteeing the optimal result, it performs well under certain conditions and can provide a challenging gameplay experience.

The Alpha-Beta Pruning algorithm is an improvement over Minmax, incorporating pruning techniques to reduce the search space. By eliminating unpromising branches, it optimizes efficiency and saves computational resources and time. However, its performance heavily relies on the order of search and may prematurely terminate the search, potentially missing the optimal solution.

Our self-implemented Residual CNN algorithm utilizes convolutional neural networks to predict opponent moves and evaluate the probability of winning. By leveraging the power of neural networks and the dynamics of the game, it provides informed predictions and decisions, offering a unique gameplay experience for players.

Throughout the implementation process, we defined the game's basic operations, including rules, move functions, and board updates. We conducted thorough testing and analysis to measure the strength of each algorithm based on metrics such as the number of wins and search time. This enabled us to determine the effectiveness and efficiency of the algorithms and assign appropriate difficulty levels to provide a challenging experience for players.

To enhance user interaction and enjoyment, we designed a user-friendly GUI interface that allows players to select the desired difficulty level and engage in gameplay against AI opponents. The interface provides a seamless and immersive experience, encouraging users to play and explore the game.

In conclusion, our AI Chinese checkers game offers a diverse range of difficulty levels and gameplay experiences. The inclusion of four algorithms allows players to challenge themselves against AI opponents with varying strategies and decision-making capabilities. The project's success lies in the effective implementation of these algorithms, the creation of a user-friendly interface, and the thorough analysis of algorithm strength. Overall, this project showcases the potential of AI in board games and provides an enjoyable and engaging experience for players of different skill levels.

References

[1] Yin, S. H. W. H. H. (2016). Playing Chinese Checkers with Reinforcement Learning.

[2] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science, 362, 1140–1144. doi:10.1126/science. aar6404.

[3] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. Nature, 550.

[4] Tao, J., Wu, G., & Pan, X. (2020, August). Design and Application of Computer Games Algorithm of Checkers. In 2020 Chinese Control And Decision Conference (CCDC) (pp. 231-234). IEEE.

[5] Liu, Z., Zhou, M., Cao, W., Qu, Q., Yeung, H. W. F., & Chung, V. Y. Y. (2019). Towards Understanding Chinese Checkers with Heuristics, Monte Carlo Tree Search, and Deep Reinforcement Learning. arXiv preprint arXiv:1903.01747.

[6] Tak, M. J., Winands, M. H., & Bjornsson, Y. (2012). N-grams and the last-good-reply policy applied in general game playing. IEEE Transactions on Computational Intelligence and AI in Games, 4(2), 73-83.

[7] GeeksforGeeks. n.d. Introduction to Convolution Neural Network. https://www.geeksforgeeks.org/introduction-convolution-neural-network/

1. . [↑](#footnote-ref-0)