Chinese Chess AI Player

FINAL REPORT

Author: Mingrui Fang

Supervisor: Yukun Lai

Moderator: Oktay Karakus

Module: CM3203 – One Semester Individual Project (40 Credits)

Institution: School of Computer Science and Informatics, Cardiff University

Abstract

This project’s objective is to investigate different AI techniques for building an automatic game player for Chinese chess. A variety of existing AI algorithms such as the Negamax Search and so on are discussed in the report.

The report in the early stage covers the whole process of AI Chinese chess game design including the UI (User Interface) design and the implementation of AI algorithms of Chinese chess, including Negamax, Alpha-beta Search and its optimization algorithms such as fail-soft Alpha-beta Search, Aspiration Search, MTD(f) and Minimal Window Search (PVS). In the later stage, these AI algorithms which are successfully implemented will be experimented and tested, and the report will show a comparison of operating efficiency of different AI algorithms and how well the AI players are playing against the human players, so as to obtain the overall results and evaluation to these different AI techniques.

Acknowledgements

First of all, I would like to acknowledge yukun, my supervisor, for his support of my project. During the weekly online meetings, Yukun was very patient in answering my questions and gave some very positive and constructive feedback. Then I would like to acknowledge my friends who helped me all the time. In the experiment part of my project, they were very patient to help me test my project, which allowed me to get more experiment data.

Table of Contents

Introduction

Artificial intelligence (AI), as a branch of computer science, has more and more widespread concern in the computer field. AI systems are part of many of the world’s devices and technologies, such as personal assistants, smart cars, content generation systems, and more [1]. With the improvement of AI theory and technique, the application field of AI is also expanding. Adversarial games like board games such as Othello, Gomoku, Checkers and Go can also be applied in AI technique. An AI player can be created in the board games by the AI technique of analysing board game situations and evaluating the possible moves an opponent can make to choose the most optimal moves on each game step. With the success of computer games in Othello, Checker and Chess, scholars all over the world have focused on more complex Chinese Chess and Shogi Go [2].

My project aims to create an AI player for Chinese chess. The main goal of my project is to implement multiple AI algorithms and evaluate their performance through a series of tests and data comparisons. To achieve this goal, I firstly need to implement a UI (User Interface) of Chinese chess so that users can easily interact and play with AI players or other human players. Before designing a UI, it is necessary for me to fully understand the game logic and rules of Chinese chess to ensure the feasibility and accuracy of the game. Then, I will start to create an AI player for Chinese chess. I will create a set of AI classes in the project that contain different AI algorithms. Human players will be able to randomly choose the AI algorithms which are implemented in the UI to play against AI players. In the duration of the game, the relevant data such as operating efficiency of the AI player will be recorded on the console for each move to performance evaluation later.

Though the AI techniques of Chinese chess have been researched and solved to a certain extent, I still think that it is meaningful and interesting to create the Chinese chess game UI and implement and evaluate the performance of the AI algorithms. There are still many potential AI algorithms worth studying and exploring in the field of AI Chinese chess.

Background and Research

Before the beginning of designing and implementing my project, I first gather a range of information which is related to my project to start my research. This section contains all the information I have gathered including the background and rules of Chinese chess and relevant potential AI algorithms. This research will help me better to make the approach to implement the game logic and AI algorithms.

Chinese Chess

Game Background

Chinese chess, also called Xiangqi, is one of the most popular and historical board games in China, which dates back to the Song Dynasty (1127−1279 A.D.). It is a two-player, zero-sum game with complete information [3]. The board represents the battle field between two armies, with one river across in the middle [3]. Each team has one King, two

Horses, two Elephants, two Advisors, two Rooks, two Canons, and five Pawns [3]. The team who captures the opponent king first wins the game [3].

Chinese chess is played on a board with 9 lines wide and 10 lines long. The pieces are placed on the intersections, which are known as points. The vertical lines are known as files and the horizontal lines are known as ranks [4]. Centred at the first to third and eighth to tenth ranks of the board are two zones, each three points by three points, demarcated by two diagonal lines connecting opposite corners and intersecting at the centre point [4]. Each of these areas is known as a castle. Dividing the two opposing sides, between the fifth and sixth ranks, is a river. Although the river provides a visual division between the two sides, only two pieces are affected by its presence: soldiers have an enhanced move after crossing the river, and elephants cannot cross it [4]. The starting points of the soldiers and cannons are usually, but not always, marked with small crosses [4].

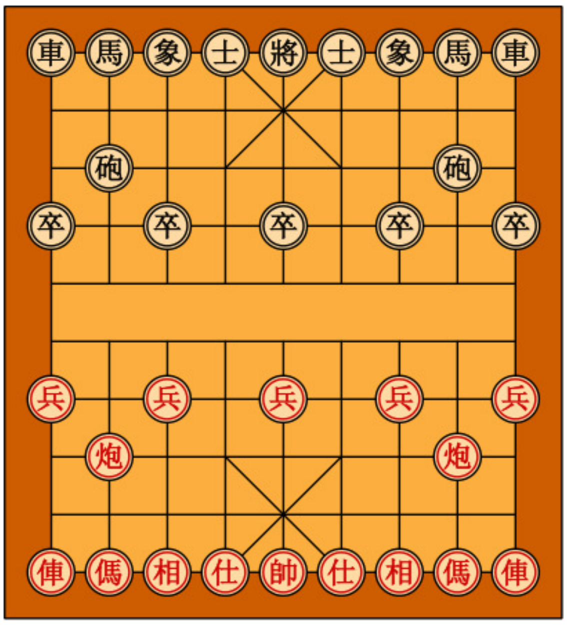


Figure 1: A 9x10 Chinese chess board

Name	Pieces
Advisor	仕 士
Cannon	炮 砲
Rook	俥 車
Elephant	相 象
King	帥 將
Horse	傌 馬
Pawn	兵 卒

Figure 2: A table which shows the meaning of each pieces

Game rules

The pieces start in the position shown in the diagram of Figure 1 above. Each player can move the pieces according to the following rules:

The King moves only one space at a time, either horizontally or vertically. Furthermore, the King must always stay within the palace, which is a square marked with an X.

The Advisor moves only one space at a time diagonally. Similar to the King, the Advisor must stay within the palace.

The Elephants move two spaces at a time diagonally (i.e. 2 spaces left/right and 2 spaces up/down in a move). They must stay within their own side of the river. If there is a piece midway between the original and final intended position of an Elephant, the Elephant is blocked and the move is not allowed.

The Rooks move one or more spaces horizontally or vertically provided that all positions between the original and final positions are empty.

The Horses move two spaces horizontally and one space vertically (or respectively 2 spaces vertically and one space horizontally). If there is a piece next to the Horse in the horizontal (vertical) direction, the horse is blocked and the move is not allowed.

The Cannons move one or more spaces horizontally or vertically like a Rook. However, in a capture move, there must be exactly one non-empty space in between the original and final position. In a non-capture move, all spaces in between must be empty.

The Pawns move one space at a time. If a Pawn does not cross the river yet, it can only move forward vertically. Once crossing the river, the Pawn can also move horizontally.

Capture: When a piece moves to a position currently held by an opponent's piece, it captures that opponent's piece. The captured piece is removed from the board.

King's line of sight: The two Kings in the board must never be on the same file (vertical line) without any pieces in between them. A move that puts the two Kings in such a setting is illegal.

King safety: One must never leave the King to be captured by the opponent in the next move. Any moves that put the King in such a setting is illegal.

The game ends when one of these two situations happens: (1) If one threatens to capture the opponent's King and the opponent has no way to resolve the threat, one wins. (2) If one does not have any valid move, one loses.

Game and AI

As a two-player, zero-sum game with complete information, Chinese chess is very suitable to be researched in the field of artificial intelligence. Compared to Chess, which has had the great improvement on the research of AI, the field of artificial intelligence in Chinese chess still needs to be further researched. In recent years, the AI Chinese chess has gradually attracted the attention of many researchers. Therefore, the field of artificial intelligence in Chinese chess has high research value and potential in the future.

AI Algorithms Research

The research into AI algorithms is quite significant to choose and implement the creation of Chinese AI players. Therefore, I will show as much information as possible I have gathered about AI algorithms and their relevant pseudocode so as to help me more easily and quickly to understand the principle of their approach before I begin to implement them.

Minimax & Negamax Search

The minimax theorem was first proven and published in 1928 by John von Neumann [5], which was considered the starting point of game theory. In the mathematical area of game theory, the minimax theorem provides conditions that guarantee that the max–min inequality is also an equality.

In game theory, the Minimax algorithm is to address the problem of n-players playing against each other in a zero-sum based game. It is a decision rule for minimising the possible loss for a worst case (maximum loss) scenario. When dealing with gains, it is to maximise the minimum gain. Originally formulated for n-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision-making in the presence of uncertainty [6]. Chinese chess, as a two-player, zero-sum game with complete information, is a great fit to be applied in the Minimax algorithm.

The Minimax algorithm is used in Chinese chess to denote minimising the opponent's maximum payoff. In a zero-sum game, this is identical to minimising one's own maximum loss, and to maximising one's own minimum gain. For example, in Figure 3, there is a tree based on Minimax. The circles represent the moves of the maximising player running the algorithm, and squares represent the moves of the opponent who is as a minimising player. Because of the deficiency of computation efficiency on computers, the tree can just search ahead of 4 moves. Suppose the search depth is 4. Then when the AI player moves a step (it is thought to be the best, record it as the number of steps 1, and the search depth is 4), it will first consider that if the opponent moves this step 1, then the AI player will definitely move the worst step 2 which is the most relevant to this step (search depth 3), and then AI player will assume the best move steps 3 (search depth 2) according to move steps 2, and continues to consider the worst move step 4 according to the move step 3 (search depth 1) Then, the search depth is 0. At this time, the evaluation function of this situation will be given.

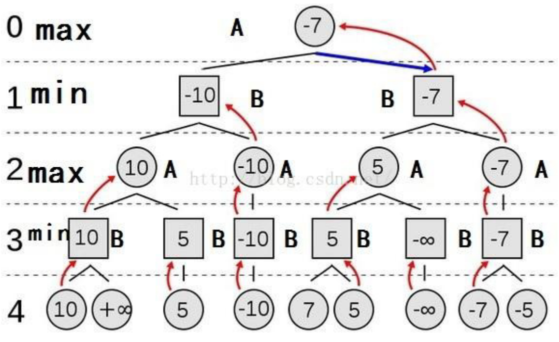


Figure 3: A tree based on Minimax

As for the program code of Minimax, the algorithm is based on recursion. Figure 4 is the pseudocode of the Minimax algorithm.

```
int Max(int depth) {
  int best = -INFINITY;
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();       //generate all reasonable moves
  while (MovesLeft()) {
    MakeNextMove();            //when moves this step
    val = Min(depth - 1);      //accepts a relevant minimum
    UnmakeMove();
    if (val > best) {
      best = val;
    }
  }
  return best;                //returns a relevant maximum value (the AI thinks)
}

int Min(int depth) {
  int best = INFINITY;
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = Max(depth - 1);   ///accepts a relevant maximum
    UnmakeMove();
    if (val < best) {
      best = val;
    }
  }
  return best;      //returns a relevant minimum value (the opponent thinks)
}
```

Figure 4: Pseudocode of the Minimax algorithm

The code of Figure 4 is very long. This algorithm requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor. There is an optimal algorithm called Negamax algorithm which is a variant form of Minimax algorithm. Negamax was invented by Knuth and Moore in 1975. This algorithm relies on the fact that $\max(a,b) = -\min(-a,-b)$ to simplify the implementation of the minimax algorithm. Here is a pseudocode of the Negamax algorithm in Figure 7 as follows.

```
int NegaMax(int depth) {
  int best = -INFINITY;
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -NegaMax(depth - 1); // there is a negative sign here.
    UnmakeMove();
    if (val > best) {           //the best value all the time
      best = val;
    }
  }
  return best;
}
```
Figure 5: Pseudocode of the Negamax algorithm

We can see from Figure 5 that the little of this code makes the algorithm make two less judgments. This function always finds the optimal value of the current node (always finds the best way to move the current node). But when the node is transformed (when the AI is transformed to a human), the function result becomes negative. This change saves the step of finding the min function and reduces the amount of code.

Alpha-beta Search and its optimisation & improvement

The Minimax algorithm needs to search the entire game tree and choose the route as best as possible. However, because the branch factor of the game tree is too large, the efficiency is very low and a deep search cannot be done. Therefore, I will show some optimization algorithms – Alpha-beta Search and its optimization & improvement.

Alpha-beta Search

Alpha–beta pruning is also a search algorithm that seeks to decrease the number of nodes that are evaluated by the Minimax algorithm in its search tree. The benefit of it is that the branch factors of the search tree can be eliminated. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. The Alpha–beta pruning reduces the effective depth to slightly more than half that of the original Minimax algorithm if the nodes are evaluated in an optimal order. Effective use of pruning will allow an AI player to evaluate more game states within the allotted processing time so as to improve the quality of its decisions. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision [7].

The principle of Alpha-beta Search can also be described in a game tree. For example, in Figure 6, making use of the Alpha-beta pruning, some part of sub-trees such as E, F and G need not to be completely searched and some of their leaf nodes can be eliminated.

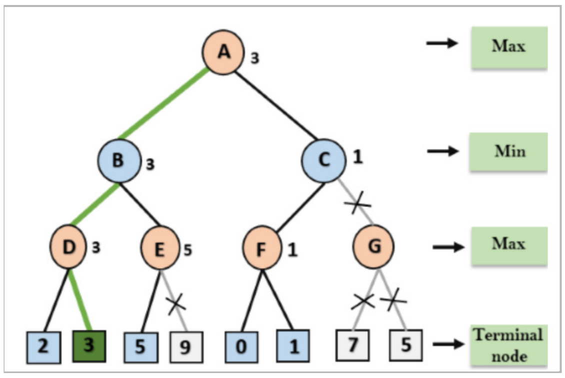


Figure 6: A game tree based on alpha-beta pruning

There are two values in this algorithm including alpha and beta. Alpha player is the maximising player which needs to ensure the minimum score. A beta player is the minimising player which needs to ensure the maximum score. In the start, both players start with their worst possible score. Whenever the maximum score that the beta player is assured of becomes less than the minimum score that the alpha player is assured of (alpha > beta), the alpha player will need not to consider further descendants of this node because their scores will never be reached in the actual play. Here is a pseudocode of the Alpha-beta Search algorithm in Figure 7 as follows.

```
int AlphaBeta(int depth, int alpha, int beta) {
  if (depth == 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -AlphaBeta(depth - 1, -beta, -alpha);  //recurse to search new node
    UnmakeMove();
    if (score >= alpha) {
      alpha = score;   //get the biggest value
      bestmove = m;    //make the best move
    }
    if (alpha >= beta) {
      break;    //occur to the beta pruning
    }
  }
  return alpha;    // get the best value
}
```

Figure 7: Pseudocode of the Alpha-beta Search algorithm

Fail-Soft Alpha-beta Search

The Fail-Soft Alpha-beta Search algorithm is based on Alpha-beta Search algorithm. The idea of this algorithm is to limit the alpha-beta value of the Alpha-Beta search so as to subtract more nodes. When the search fails, return the current value. This is available to avoid the situation when the search fails and cannot obtain any information leading to searching again. Here is a pseudocode of the Fail-soft Alpha-beta Search algorithm in Figure 8 as follows.

```
int  FAlpah(int depth,int alpha,int beta){
    int current =-INFINITY;   //current=负无穷
     if(game over or depth <=0)
         return eval();
     if(depth <=0)
          return eval();
     for(each  possible move m)  {
             make move m;
             score =-FAlpahBeta(depth-1,-beta,-alpha) //recurse to search new node
             unmake move m;
                        if(score >current){
                                     current =score;   //retain the maximum value
                                     if(score >=alpha)
                                                alpha=score;  //change the boundary of alpha
                                       if(score>=beta)
                                                    break;  //beta pruning
                         }
    }
       return current;  //get the best value
}
```
Figure 8: Pseudocode of the Fail-Soft Alpha-beta Search algorithm

Aspiration Search

The Aspiration Search algorithm can be also based on Alpha-beta Search algorithm or Fail-Soft Alpha-beta Search algorithm. The idea of this algorithm is to assume that the search results are near a certain value, so as to use Alpha-Beta search within a given small range, and adjust the range when fail high and fail low, which belongs to a narrow window search. Although this may cause the search to not return the correct value, it is also possible to find the correct value faster. When it is found that the returned value is not within the predicted range, it needs to be searched again. If the cost saved by narrowing the window is greater than the cost increased by searching again, the search speed will be improved. Here is a pseudocode of the Aspiration Search in Figure 9 as follows.

```
int alpha = previous - WINDOW;
int beta = previous + WINDOW;
for ( ; ; ) {
  score = alphabeta(depth, alpha, beta);    //use Alpha-beta algorithm
  if (score <= alpha) {    //lower than the fail low of window
    alpha  = -WIN;
  } else if (score >= beta) {    //higher than the fail high of window
    beta = WIN;
  } else {
    break;
  }
}
```
Figure 9: Pseudocode of the Aspiration Search algorithm

Minimal Window Search (PVS)

The Minimal Window Search is also called NegaScout or Principal Variation Search (PVS). The idea of this algorithm is that in a strongly ordered game tree, the first move on each node is likely to be the best and proving a subtree is inferior takes less time than calculating the game value of this subtree.

According to this thinking, this algorithm assumes that the first move at each node is the best, and tries to keep proving that the following moves are relatively inferior until this assumption is proved wrong. For the first subtree of the node, use the complete search window [α, β] to perform an exhaustive search to obtain the game value V of the subtree. For the remaining subtrees of this node, use the smallest search window [V, V + 1] to search so as to quickly find out whether the subtree is relatively inferior. If it proves that the current subtree is not inferior, the subtree needs to be searched again and the game value V needs to be modified. When the search depth is 5, the efficiency of this algorithm is about 250% better than Alpha-beta Search. This minimum window is also called the Null Window. Here is a pseudocode of the Principal Variation Search in Figure 10 as follows.

```
int PVS(int depth, int alpha, int beta) {
  move bestmove, current;
  if (game over or depth <= 0) {
    return eval();
  }
  make move m;
  current = -PVS(depth - 1, -beta, -alpha);  //make the first move and assume it is the best move
  unmake move m;
  for (other possible move m) {   //make other moves
    make move m;
    score = -PVS(depth - 1, -alpha - 1, -alpha);
    if (score > alpha && score < beta) {
          //indicate the assumption of that the first move is the best move is not check out
          //search again and find the best move
      score = -PVS(depth - 1, -beta, -alpha);
    }
    unmake move m;
    if (score >= current) {
      current = score;
      bestmove = m;
      if (score >= alpha) {
        alpha = score;
      }
      if (score >= beta) {
        break;
      }
    }
  }
  return current;
}
```

Figure 10: Pseudocode of the Minimal Window Search algorithm

MTD(f)

MTD(f) was first described in a University of Alberta Technical Report authored by Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin [8]. This algorithm is like the Aspiration search. The difference is that the initial value is adjusted when using the Alpha-beta Search. The narrower the search window is, the faster the search is. The idea of this algorithm is to make the search window as narrow as possible. It always uses "beta = alpha + 1" to use Alpha-Beta Search. The effect of using such a "zero-width" search is to compare the Alpha value to the exact value. If the search returns a value that is at most Alpha value, the exact value itself is at most an Alpha value. otherwise, the exact value is greater than Alpha value. Here is a pseudocode of MTD(f) in Figure 11 as follows.

```
int alpha = -WIN;
int beta = +WIN;
while (beta > alpha + 1) {
  int test = (alpha + beta) / 2;
  if (alphabeta(depth, test, test + 1) <= test) {
    beta = test;
  } else {
    alpha = test + 1;
  }
}
```

Figure 11: Pseudocode of the MTD(f)

Evaluation Function

An evaluation function, also known as a heuristic evaluation function or static evaluation function, is a function used by game-playing computer programs to evaluate the value or goodness of a position (usually at a leaf or terminal node) in a game tree [9]. The evaluation function is the “brain” of AI Chinese chess. It is used for evaluating which moves might be relevant to the AI Player at the current game state. Each possible move will be provided a score as the evaluation. The evaluation function is flexible and variable using the Negamax algorithm. The evaluation results from the same game board may be different, relying on whose turn the player is. There are mainly three factors for evaluation function that need to be considered in AI Chinese chess including the piece value, piece position and piece flexibility.

Piece Value

The piece value refers to the value of a piece itself. For example, if the rook is worth 500, it might be 300 for the horse, 80 for the pawn, etc. Hence, when evaluating the situation, we first need to consider the comparison of the sum of the sub-forces of the two sides. For example, if one player has full Rooks, Horses and cannons, and another player only has one Rook and two horses, then the former player has an obvious advantage.

Piece Position

The piece position refers to the position occupied by a piece of a player on the chessboard. For example, Cannons moving to the bottom of the opponent, Pawns passing river, and are the good pieces of chess position status, while kings leaving the bottom, etc, are bad pieces of chess position status.

Piece Flexibility

The piece flexibility refers to the movability of the pieces. For example, the Rook at the starting position has poor mobility, so we pay attention to moving Rook away from the starting position as soon as possible in Chinese chess. Similarly, Horses at their starting position with their legs held back on all sides have poor mobility (for a piece that cannot move any step, it can be considered that its mobility is 0).

Approach

For my project, the approach taken to create a Chinese chess AI player is based on the background information I gathered in the previous section. According to my research into Chinese chess and its relevant AI techniques development, I think the project about creating a Chinese chess AI player is feasible and meaningful. To achieve this goal, firstly, the design of UI (User Interface) of Chinese chess is needed. This is to visualise the game so that human players can play Chinese chess on this interface. Then, I will start to implement the creation of Chinese chess AI players according to the AI approach I have gathered in the previous section. The implementation of the AI approach includes two parts: search algorithm and situation evaluation function. The search algorithm will use different types to create multiple different AI players such as Negamax Search, Alpha-beta Search and its relevant optimization Search algorithm. Finally, I will test and analyse different AI players through human-AI playing and data comparison between different AI players. The console will output relevant data, such as time spent, the number of nodes searched, etc, for each move of AI players.

I will use the Java programming language to implement these approaches. The reason why I choose this programming language is because I am familiar with it. During my study in Cardiff University, many of my courses were based on Java language, and I completed many relevant coursework using Java, which made me accumulate a lot of experience on Java. Hence, I think choosing Java is able to improve my work efficiency to a certain extent.

What is more, as an object-oriented programming language, Java can make it easier for me to divide the functions I need to implement into various modules, such as UI module and AI module, for management. This will make my project code have a better extendability and be easier to debug and modify. Additionally, the Java programming language contains abundant packages that can be imported directly. These packages can help me implement some functions quickly. For example, through the swing package, I will be able to design the UI of Chinese chess more quickly and easily. This also improves my work efficiency.

To provide myself with a clear design idea to follow in the implementation stage, I created an overall design flow diagram shown in Figure 12. I will further implement these functions according to this flow diagram.

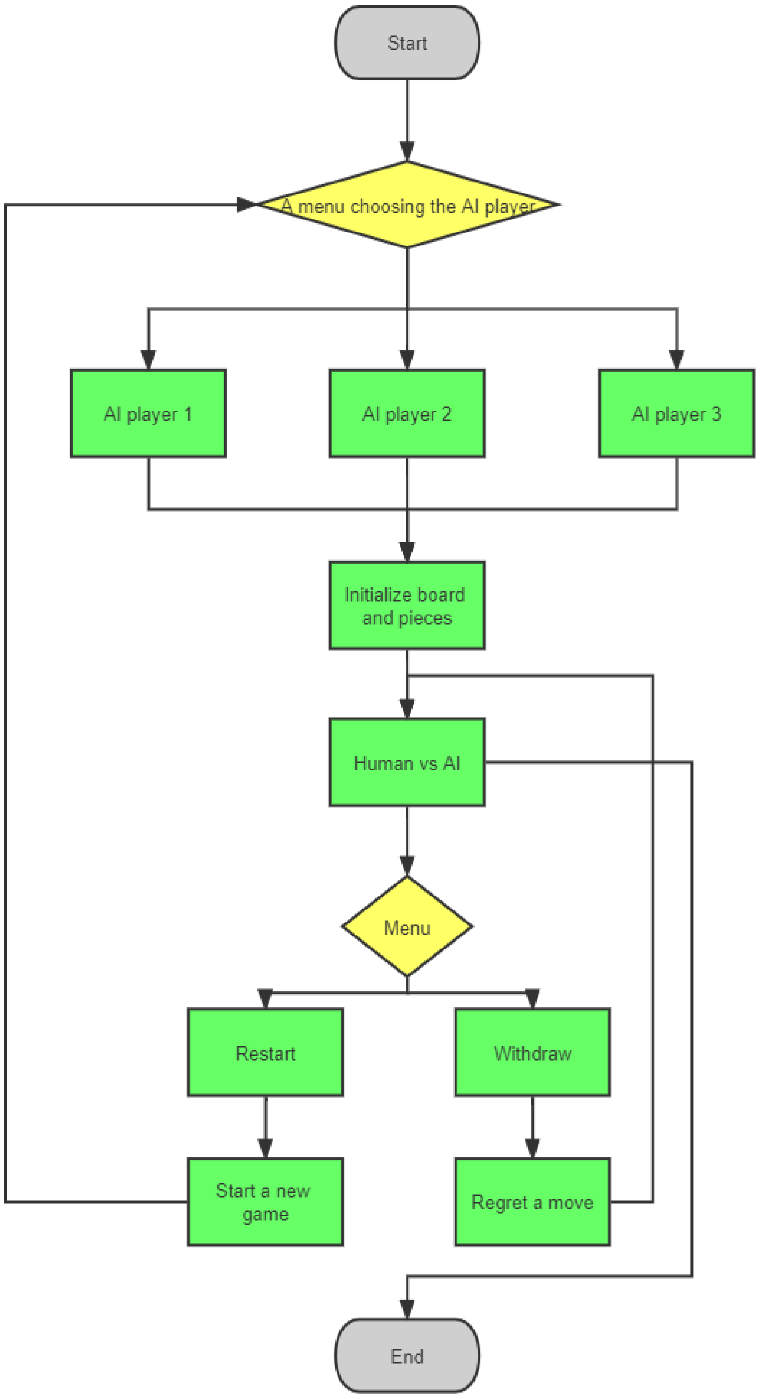


Figure 12: An overall design flow diagram

As for the specific design and implementation of UI and AI approaches, I will describe them in detail in the next section.

Design and Implementation

In this section, I will detailly show the design and implementation of Chinese chess UI and AI. Some relevant uml diagrams and source code implementation will be directly attached so as to help readers easier to understand the processing of design and implementation.

Uml Class Diagram

The structure of the program is described in a Uml class diagram and I will explain these classes in detail.

Overall Structure

Figure 13 is the Uml class diagram showing the general structure of the Chinese chess project.

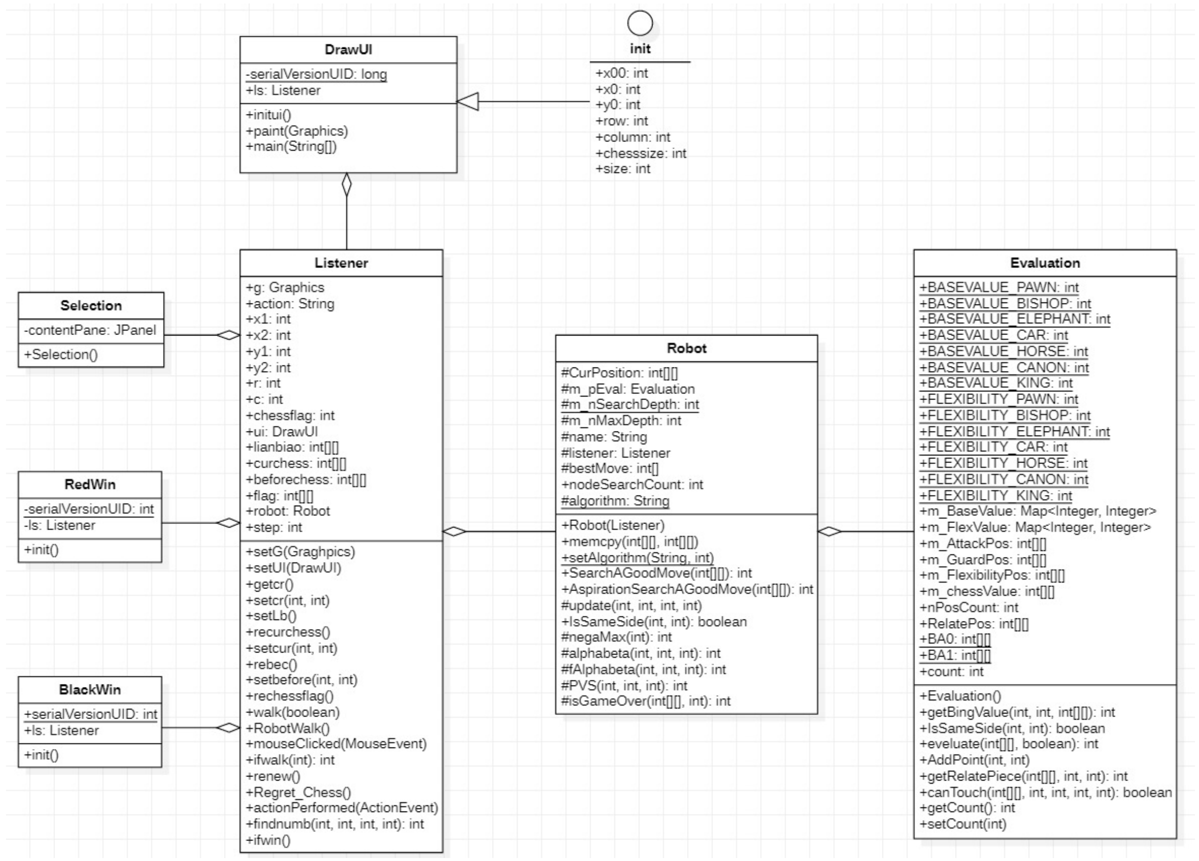


Figure 13: Overall Uml class diagram of Chinese chess

UI Module

The UI module is made up of DrawUI class, Selection class and Listener class, which is to create the graphical user interface, create the game rules and create an operating function in the button.

DrawUI class

DrawUI class is created to define the variables and instance objects used in the game, initialise the graphical user interface through the constructor, add components and chess pieces, and register event components.

There are three buttons created in the DrawUI class including the “Start” button, “Restart” button and “Withdraw” button. The ”Start” button is to initialise the game state to start the game. When clicking the “Restart” button, the game board will be initialised again. This will clear the current operating status and recreate a new initial game board. When clicking the “Withdraw” button, the program will go back to the last step game situation.

Selection class

Selection class is to create a prompt box to let users choose the AI player and the depth of search in two drop-down menus. This class will be used when clicking the “Start” button or the “Restart” button. And there is also a ”Confirm” button in this class so as to send the action information of users.

Listener class

Listener class is created to set the game rules and do some action events on the mouse. Many functions such as starting the game, restarting the game and pieces withdrawing are implemented in this class.

What is more, this class is a significant “pivot” to combine the UI module and AI module. All calls of AI algorithms initially start in this class. This makes the game playing between Human players and AI players can be implemented.

AI Module

The AI module is made up of Robot class and Evaluation class, which is to implement the AI algorithms and analyse the game board situation so as to create AI players.

Robot class

Robot class is created to implement a variety of AI algorithms and the Evaluation class will be called when an AI player searches for possible moves. The Robot class is called by the Listener class.

Evaluation class

Evaluation class is created to evaluate which moves might be relevant to the AI Player at the current game state. Each possible move will be provided a score as the evaluation. Each piece is given a variant of their piece value, piece position and piece flexibility. Some methods referring to piece relations use these variants to obtain the values of searching in this class.

Implementation

The project implementation contains the UI and AI implementation of Chinese chess. To clearly show the process of implementation, the screenshot of the whole process of interface interaction and some key source codes will be shown in the report. I will explain these interface screenshots and source codes in detail.

UI Implementation

The UI implementation is via Java's Swing package. The main frame is created in the main interface page using the JFrame class. This interface has three panes containing the chessboard pane, button pane and logo pane, which all extend the JPanel class.

Logo pane contains an instance of the JLabel class. It uses the paint() method of JPanel to place the Chinese chess image logo into the pane. Figure 14 shows the Chinese chess logo placed in the interface.

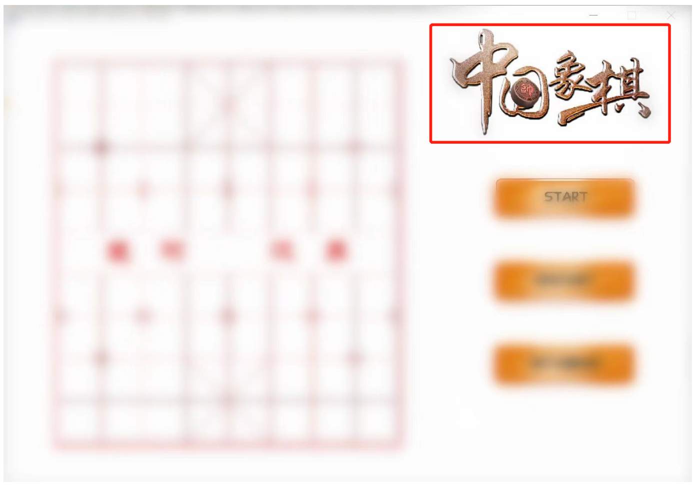


Figure 14: Chinese chess logo

Chessboard pane is to place the Chinese chess board. The image is from the internet. I also use the paint() method of JPane to implement. Here is the chessboard placed in the interface in Figure 15.

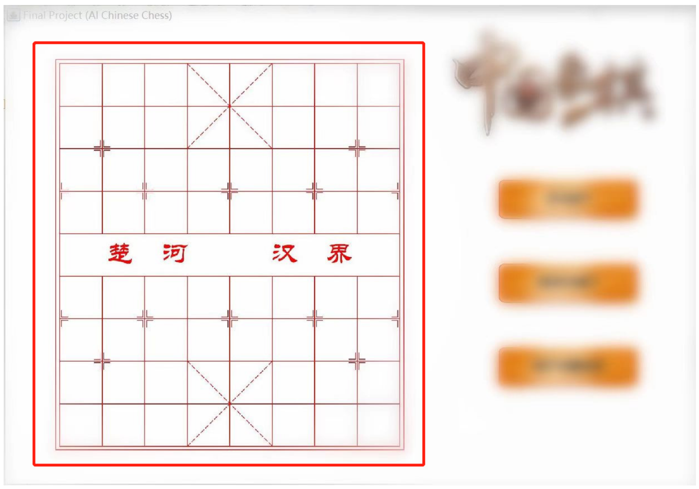


Figure 15: Chessboard

Button pane has three buttons containing the “Start” button, “Restart” button and “Withdraw” button. I use the paint() method of JButton class to add an image to the button, and set the position of the button by the setBounds() method. A JPane is created to place these buttons. Figure 15 shows the Buttons placed in the interface.

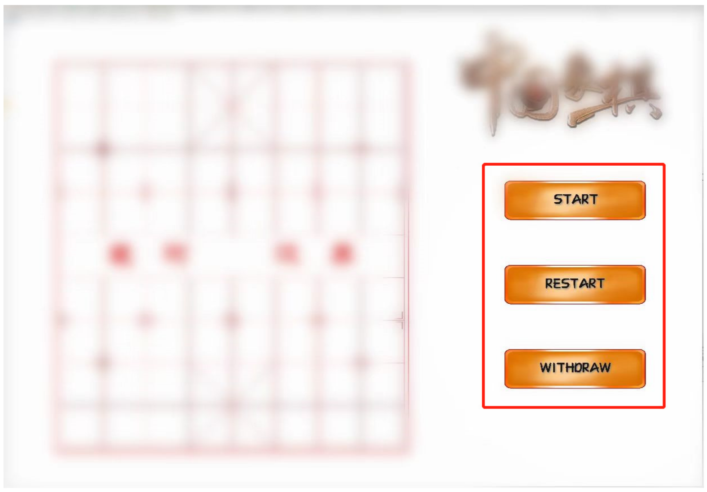


Figure 15: Buttons in the interface

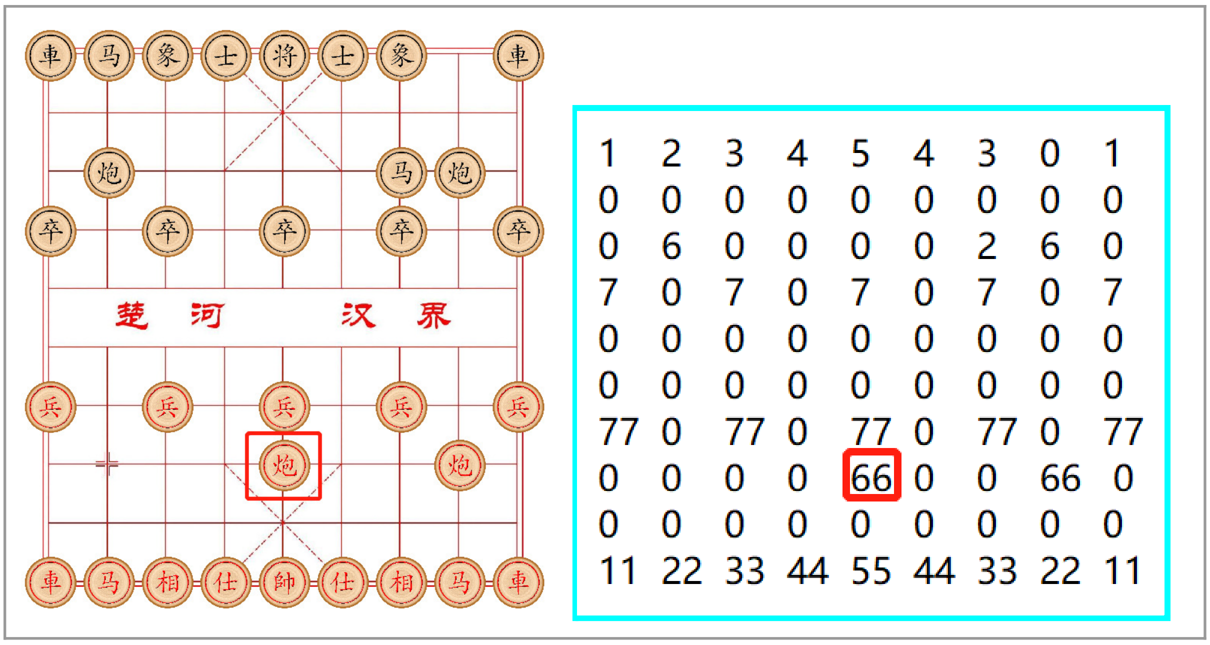
These panes above consist of the original and basic static interface. To implement the creation and control of the pieces in the chessboard, a Listener class is created. In this class, there are some methods created to control the pieces.

I created a 2-dimensional array named flag. Each piece can be signified by a number on the chess board. For example, the Rooks are signified as 1 or 11 on the chess board, the Elephants are signified as 3 or 33 on the chess board. If there are no pieces in a position on the chess board, this position can be signified as 0. The detailed number signs of pieces are as follows.

Black pieces		Red pieces	
Piece	Sign	Piece	Sign
Rook	1	Rook	11
Horse	2	Horse	12
Elephant	3	Elephant	13
Advisor	4	Advisor	14
King	5	King	15
Cannon	6	Cannon	16
Pawn	7	Pawn	17

Figure 16: Number signs of pieces in 2-dimensional array

Here is an example of the piece Cannon signed in the chessboard in Figure17 as follows.



```
1  2  3  4  5  4  3  0  1
0  0  0  0  0  0  0  0  0
0  6  0  0  0  0  2  6  0
7  0  7  0  7  0  7  0  7
0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0
77 0  77 0  77 0  77 0  77
0  0  0  0  66 0  0  66 0
0  0  0  0  0  0  0  0  0
11 22 33 44 55 44 33 22 11
```

Figure 17: An example of the piece Cannon signed in the chessboard

Now, the design of the interface is finished. Figure 18 shows the whole main interface.

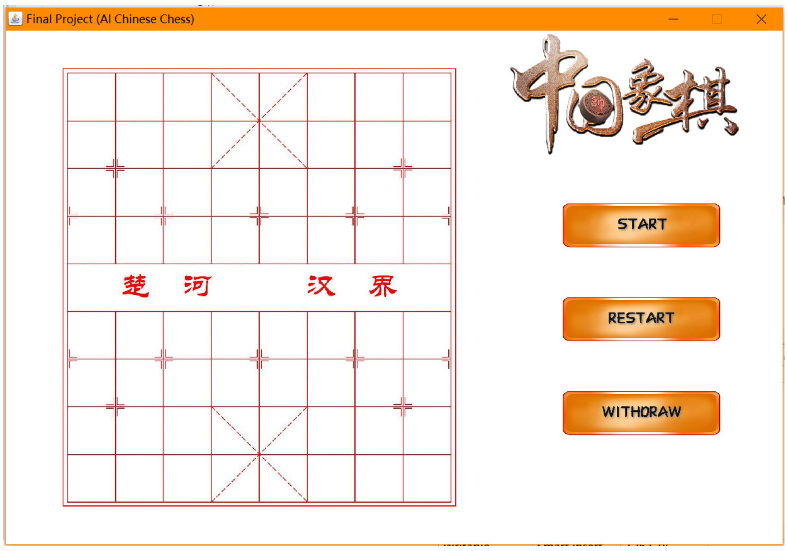


Figure 18: The whole interface

Next, I will introduce the implementation of the functions of moving pieces, withdrawing pieces and restarting. These functions are implemented in the Listener class.

I have mentioned above I create a 2-dimensional array named flag to place the pieces on the chessboard. If we hope that the pieces can move, just need to change the values of this array. Firstly, use the mouseclick() method to obtain the current click position. Then, use the getcr() method to the coordinate of row and column of the current click position.

When I click a piece, and then click another position, the piece can move to this new position. There are two array variants including curchess and beforechess created to save each click position.

The move of pieces needs to follow the game rules. Hence, the findnumb() method is created to get the number of pieces in the middle of the start position and the click position when they are on a straight line, so as to judge whether the Cannon and the Rook can move. The ifwalk() method contains the rule of all piece move.

Now, the piece can move following the game rules. Figure 19 shows an example of the Horse move. We can see from Figure 19 that the red Horse moved from (9, 1) to (7, 2).

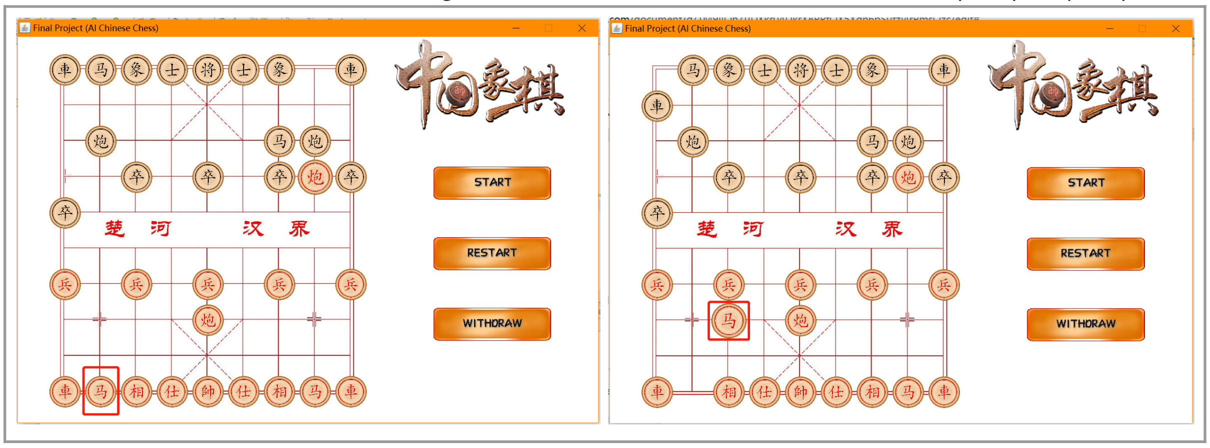


Figure 19: An example of the Horse move

Additionally, in order to allow human players to freely select the AI algorithm and search depth, I created a Selection class. This Selection class is to create a prompt box which has two drop- down menus and a “Confirm” button. This prompt box will be called when clicking the “Start” button or the “Restart” button in the main interface page.

Selection class extends the JFrame class. There is a private variant contentPane which is an instance of JPane. There are two instances containing algorithmBox and depthBox which are instances of JComboBox. One is to select the algorithm and another is to select the search depth. These two boxes will be added in the contentPane. And there is also a ”Confirm” button in this class so as to send the selected action of users. Figure 20 shows this prompt box.

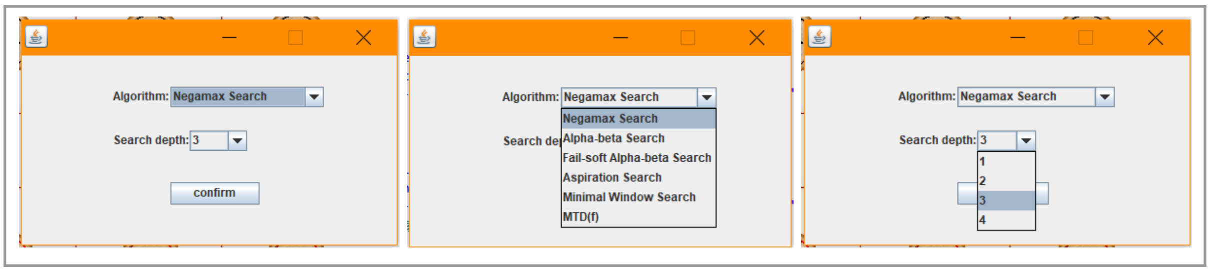


Figure 20: An prompt box using for selecting the algorithm and search depth

Besides, there is also a function of 'regretting‘ to withdraw a move. This function is implemented by creating an array(int[][] lianbiao) to save a last move. When clicking the “Withdraw” button, the Regret_Chess() method in Listener class will be called to let current move back to the last move which is saved in the array(int[][] lianbiao).

All UI requirements have been implemented so far. I will start to create a Chinese chess AI player.

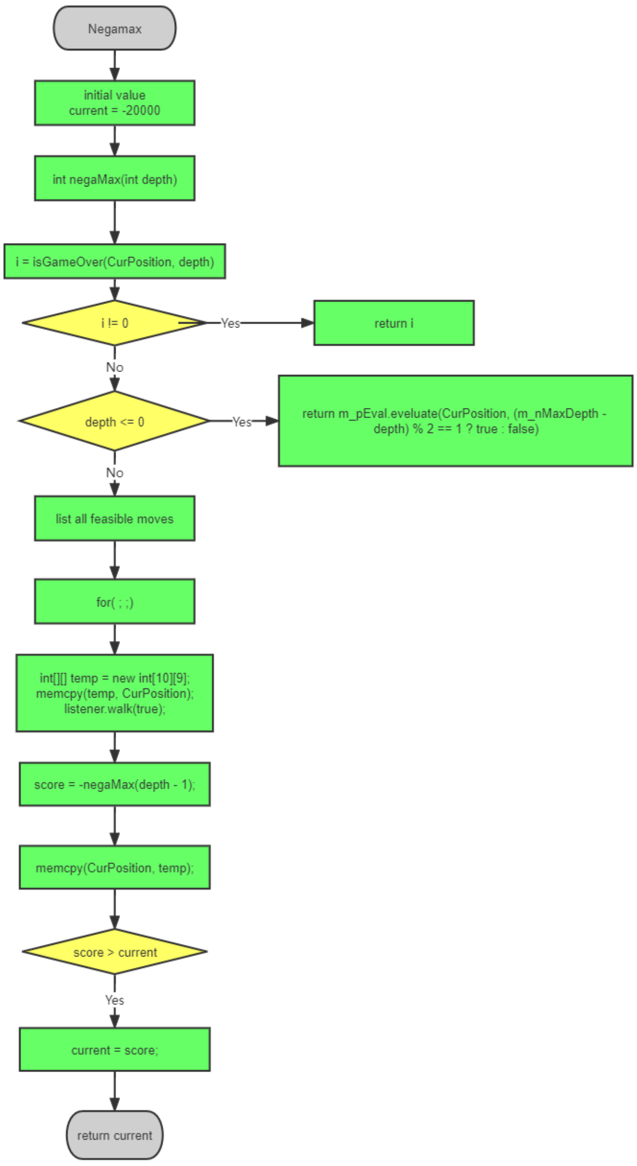
AI Implementation

AI implementation means to successfully create Chinese chess AI players. In my java project, I create two classes including a Robot class which implements the AI algorithm and an Evaluation class which implements the evaluation function. In the report, I will show the flow diagrams of these algorithms to explain their principle process and some relevant source code I wrote in the project will be attached so as to make it easy for the reader to understand.

Algorithm Implementation

Negamax Search implementation

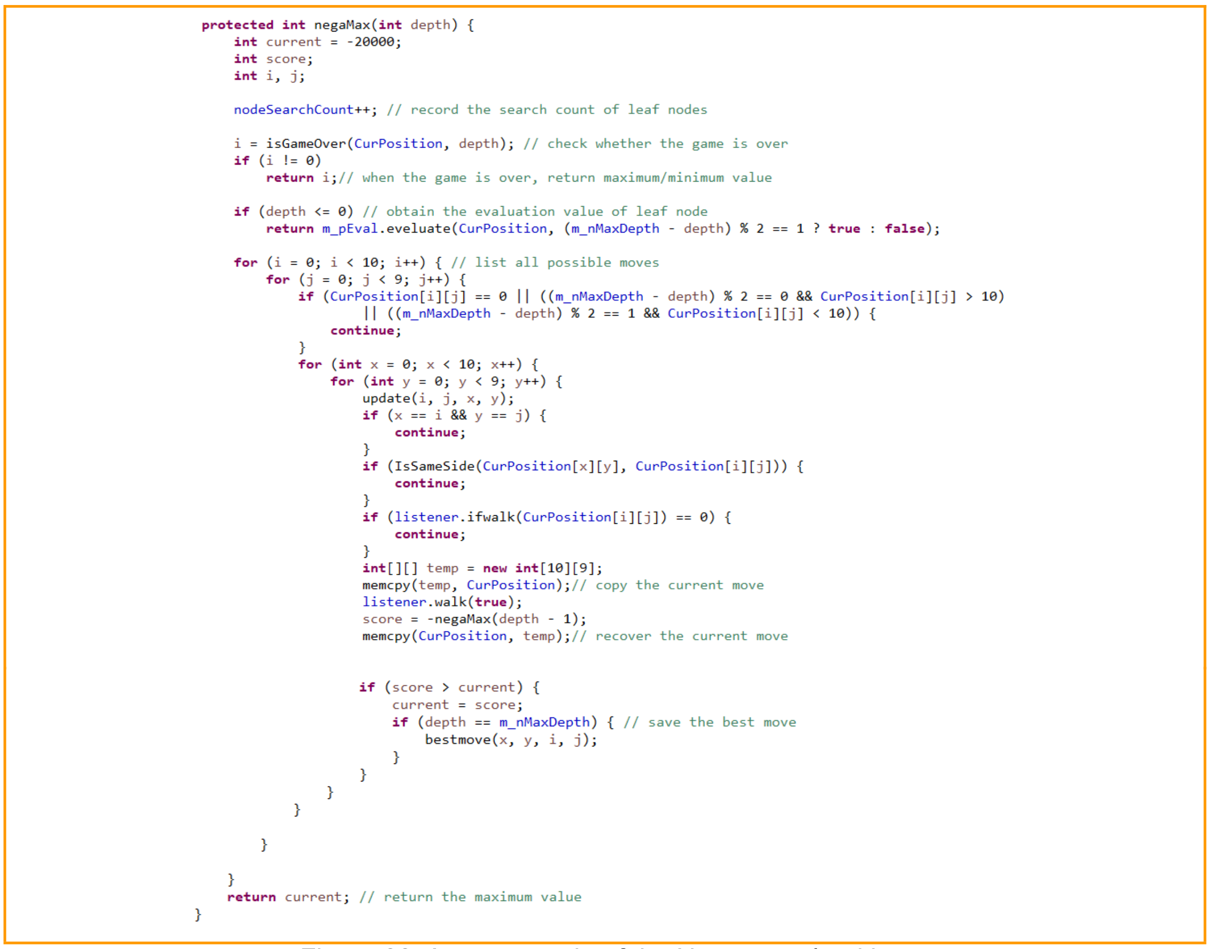
Due to the Negamax algorithm having the same algorithm logic as the Minimax algorithm, I decided to just implement the Negamax algorithm. To implement the Negamax algorithm, I created a NegaMax(int depth) method in the Robot class. The flow diagram of the NegaMax(int depth) method is shown in Figure 21.



Negamax
initial value
current = -20000
int negaMax(int depth)
i = isGameOver(CurPosition, depth)
i != 0
Yes
return i
No
depth <= 0
Yes
return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false)
No
list all feasible moves
for(; ;)
int[][] temp = new int[10][9];
memcpy(temp, CurPosition);
listener.walk(true);
score = -negaMax(depth - 1);
memcpy(CurPosition, temp);
score > current
Yes
current = score;
return current

Figure 21: A flow diagram of NegaMax(int depth) method

We can see from Figure 21 that the Negamax algorithm has an initial variant “current” and current = -20000. Firstly, use the isGameOver(CurPosition, depth) method to check whether the game is over. when the game is over (1 != 0), return maximum/minimum value. Then, if the depth < 0, obtain the evaluation value of leaf node by calling the evaluate(int[][] position, boolean bIsRedTurn) method. Later, search and list all possible moves. For each possible move, make a temporary move. Call score = -NegaMax(nDepth-1) using recursion to search the next depth. Then recover the current move. If score > current, modify the variant current to score. Finally, return the variant current. The source code of the Negamax algorithm is shown in Figure 22.

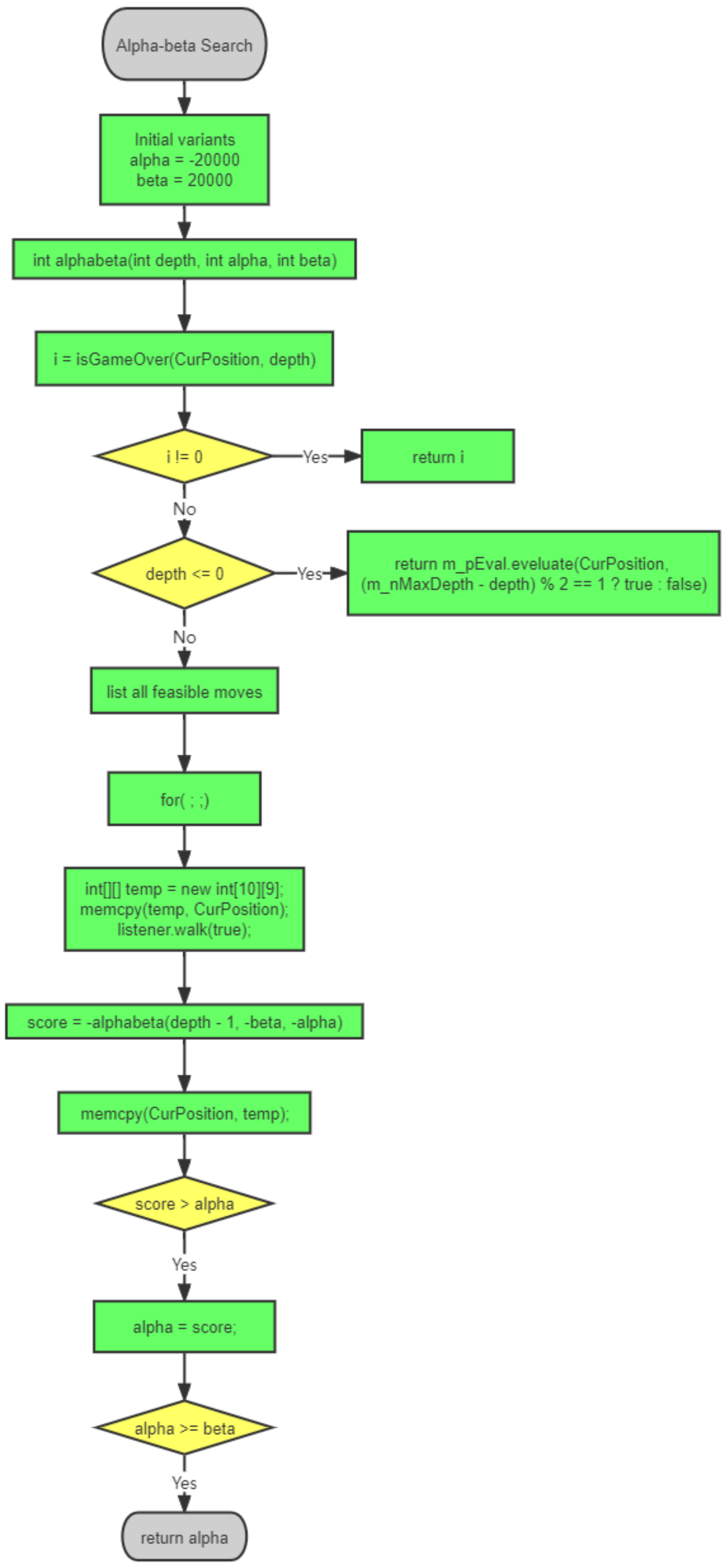


```
protected int negaMax(int depth) {
    int current = -20000;
    int score;
    int i, j;

    nodeSearchCount++; // record the search count of leaf nodes

    i = isGameOver(CurPosition, depth); // check whether the game is over
    if (i != 0)
        return i;// when the game is over, return maximum/minimum value

    if (depth <= 0) // obtain the evaluation value of leaf node
        return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false);

    for (i = 0; i < 10; i++) { // list all possible moves
        for (j = 0; j < 9; j++) {
            if (CurPosition[i][j] == 0 || ((m_nMaxDepth - depth) % 2 == 0 && CurPosition[i][j] > 10)
                    || ((m_nMaxDepth - depth) % 2 == 1 && CurPosition[i][j] < 10)) {
                continue;
            }
            for (int x = 0; x < 10; x++) {
                for (int y = 0; y < 9; y++) {
                    update(i, j, x, y);
                    if (x == i && y == j) {
                        continue;
                    }
                    if (IsSameSide(CurPosition[x][y], CurPosition[i][j])) {
                        continue;
                    }
                    if (listener.ifwalk(CurPosition[i][j]) == 0) {
                        continue;
                    }
                    int[][] temp = new int[10][9];
                    memcpy(temp, CurPosition);// copy the current move
                    listener.walk(true);
                    score = -negaMax(depth - 1);
                    memcpy(CurPosition, temp);// recover the current move


                   if (score > current) {
                       current = score;
                       if (depth == m_nMaxDepth) { // save the best move
                           bestmove(x, y, i, j);
                       }
                   }
                }
            }
        }

    }

   }
   return current; // return the maximum value
}
```

Figure 22: A source code of the Negamax algorithm

Alpha-beta Search implementation

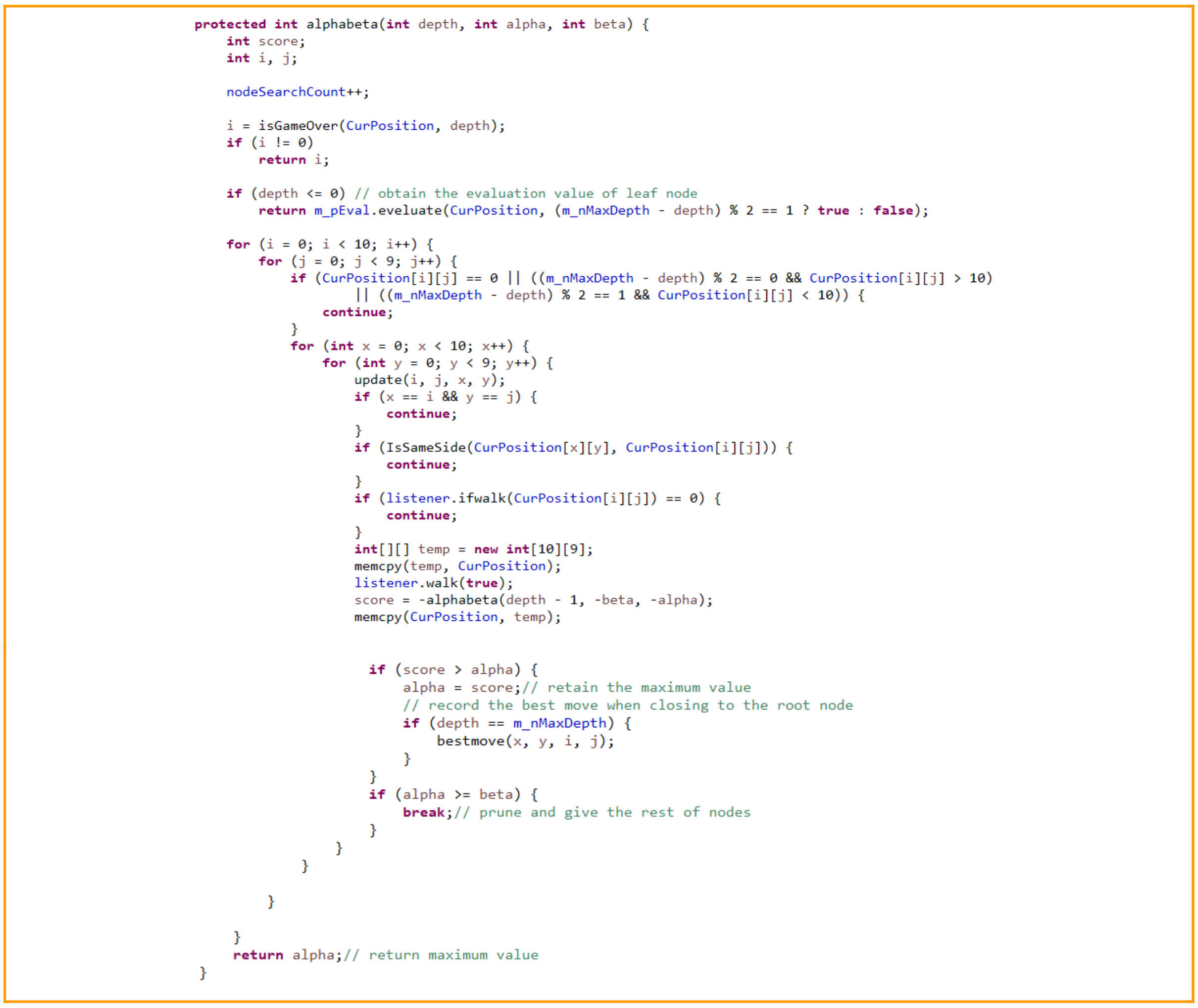
Alpha-beta Search algorithm is to “prune” the branches of the game tree under certain conditions. To implement the Alpha-beta Search algorithm, I created an alphabeta(int depth, int alpha, int beta) method in the Robot class. In this method, the value alpha is to maximum and the value beta is to minimum. The flow diagram of the alphabeta(int depth, int alpha, int beta) method is shown in Figure 23.



Alpha-beta Search
Initial variants
alpha = -20000
beta = 20000
int alphabeta(int depth, int alpha, int beta)
i = isGameOver(CurPosition, depth)
i != 0
Yes
return i
No
depth <= 0
Yes
return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false)
No
list all feasible moves
for(; ;)
int[][] temp = new int[10][9];
memcpy(temp, CurPosition);
listener.walk(true);
score = -alphabeta(depth - 1, -beta, -alpha)
memcpy(CurPosition, temp);
score > alpha
Yes
alpha = score;
alpha >= beta
Yes
return alpha

Figure 23: A flow diagram of alphabeta(int depth, int alpha, int beta) method

We can see from Figure 22 that Alpha-beta Search algorithm has initial variants “alpha” and “beta”. The “alpha” is given a value -20000 and the ”beta” is given a value 20000. Most of the steps of this algorithm are the same as the Negamax algorithm. The difference is that for each possible move, maybe there is a pruning process. If score > alpha, let alpha = score to retain the maximum value. If alpha > beta, the node can be pruned. Finally, return the maximum alpha. The source code of the Alpha-beta Search algorithm is shown in Figure 24.

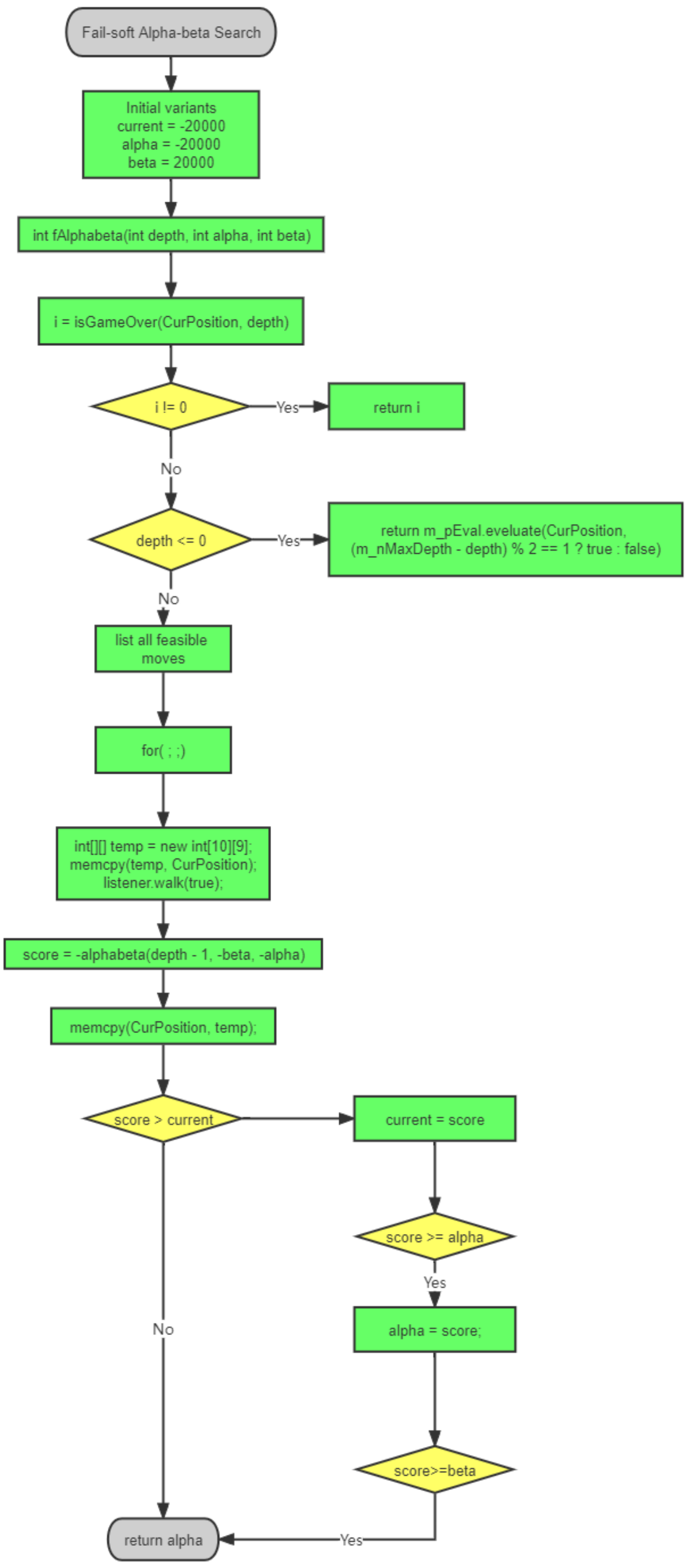


```
protected int alphabeta(int depth, int alpha, int beta) {
    int score;
    int i, j;

    nodeSearchCount++;

    i = isGameOver(CurPosition, depth);
    if (i != 0)
        return i;

    if (depth <= 0) // obtain the evaluation value of leaf node
        return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false);

    for (i = 0; i < 10; i++) {
        for (j = 0; j < 9; j++) {
            if (CurPosition[i][j] == 0 || ((m_nMaxDepth - depth) % 2 == 0 && CurPosition[i][j] > 10)
                    || ((m_nMaxDepth - depth) % 2 == 1 && CurPosition[i][j] < 10)) {
                continue;
            }
            for (int x = 0; x < 10; x++) {
                for (int y = 0; y < 9; y++) {
                    update(i, j, x, y);
                    if (x == i && y == j) {
                        continue;
                    }
                    if (IsSameSide(CurPosition[x][y], CurPosition[i][j])) {
                        continue;
                    }
                    if (listener.ifwalk(CurPosition[i][j]) == 0) {
                        continue;
                    }
                    int[][] temp = new int[10][9];
                    memcpy(temp, CurPosition);
                    listener.walk(true);
                    score = -alphabeta(depth - 1, -beta, -alpha);
                    memcpy(CurPosition, temp);


                      if (score > alpha) {
                          alpha = score;// retain the maximum value
                          // record the best move when closing to the root node
                          if (depth == m_nMaxDepth) {
                              bestmove(x, y, i, j);
                          }
                      }
                      if (alpha >= beta) {
                          break;// prune and give the rest of nodes
                      }
                  }
             }

         }

     }
     return alpha;// return maximum value
 }
```

Figure 24: A source code of the Alpha-beta Search algorithm

Fail-soft Alpha-beta Search implementation

Fail-soft Alpha-beta Search algorithm is one of the optimizations of Alpha-beta Search algorithm. This difference is that the values ”current” and “alpha” are saved separately. This modification can let computers get the “fail-soft” information. To implement the Fail-soft Alpha-beta Search algorithm, I created an fAlphabeta(int depth, int alpha, int beta) method in the Robot class. The flow diagram of the fAlphabeta(int depth, int alpha, int beta) method is shown in Figure 25.

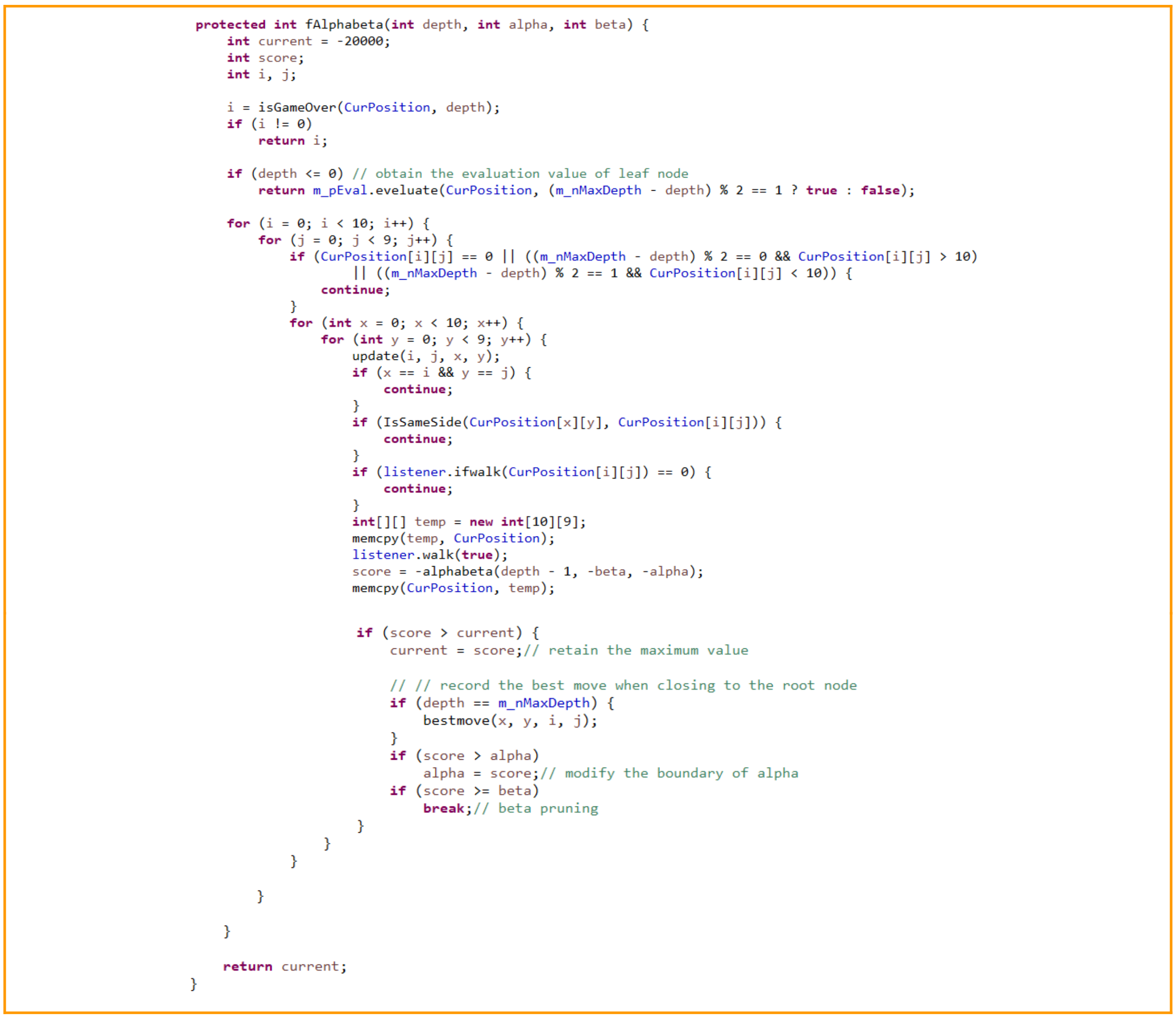


Fail-soft Alpha-beta Search
Initial variants
current = -20000
alpha = -20000
beta = 20000
int fAlphabeta(int depth, int alpha, int beta)
i = isGameOver(CurPosition, depth)
i != 0
Yes
return i
No
depth <= 0
Yes
return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false)
No
list all feasible moves
for(; ;)
int[][] temp = new int[10][9]; memcpy(temp, CurPosition); listener.walk(true);
score = -alphabeta(depth - 1, -beta, -alpha)
memcpy(CurPosition, temp);
score > current
current = score
score >= alpha
Yes
alpha = score;
No
score>=beta
Yes
return alpha

Figure 25: A flow diagram of fAlphabeta(int depth, int alpha, int beta) method

We can see from Figure 25 that Fail-soft Alpha-beta Search algorithm has initial variants “current”, “alpha” and “beta”. The “current” is given a value -20000. The “alpha” is given a value -20000 and the ”beta” is given a value 20000. Different from Alpha-beta Search algorithm, for each possible move, if score > current, retain the maximum value (current =

score). When score > alpha, modify the alpha boundary. When score >= beta, prune beta. Finally, return the value “current”. The source code of the Fail-soft Alpha-beta Search algorithm is shown in Figure 26.



```
protected int fAlphabeta(int depth, int alpha, int beta) {
    int current = -20000;
    int score;
    int i, j;

    i = isGameOver(CurPosition, depth);
    if (i != 0)
        return i;

    if (depth <= 0) // obtain the evaluation value of leaf node
        return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false);

    for (i = 0; i < 10; i++) {
        for (j = 0; j < 9; j++) {
            if (CurPosition[i][j] == 0 || ((m_nMaxDepth - depth) % 2 == 0 && CurPosition[i][j] > 10)
                    || ((m_nMaxDepth - depth) % 2 == 1 && CurPosition[i][j] < 10)) {
                continue;
            }
            for (int x = 0; x < 10; x++) {
                for (int y = 0; y < 9; y++) {
                    update(i, j, x, y);
                    if (x == i && y == j) {
                        continue;
                    }
                    if (IsSameSide(CurPosition[x][y], CurPosition[i][j])) {
                        continue;
                    }
                    if (listener.ifwalk(CurPosition[i][j]) == 0) {
                        continue;
                    }
                    int[][] temp = new int[10][9];
                    memcpy(temp, CurPosition);
                    listener.walk(true);
                    score = -alphabeta(depth - 1, -beta, -alpha);
                    memcpy(CurPosition, temp);

                    if (score > current) {
                        current = score;// retain the maximum value

                        // // record the best move when closing to the root node
                        if (depth == m_nMaxDepth) {
                            bestmove(x, y, i, j);
                        }
                        if (score > alpha)
                            alpha = score;// modify the boundary of alpha
                        if (score >= beta)
                            break;// beta pruning
                    }
                }
            }

        }

    }

    return current;
}
```

Figure 26: A source code of the Fail-soft Alpha-beta Search algorithm

Aspiration Search implementation

The Aspiration Search algorithm is not to modify the Alpha-beta Search algorithm, it is just to change a way to call the search process from the outside. The feature of Aspiration Search is that We can directly set a narrow window centred around the previous search value, which is often useful for searching. If your search fails, increase the width of the window and search again. To implement the Aspiration Search algorithm, I created an AspirationSearch(int[][] position) method in the Robot class. The flow diagram of the AspirationSearch(int[][] position) method is shown in Figure 27.

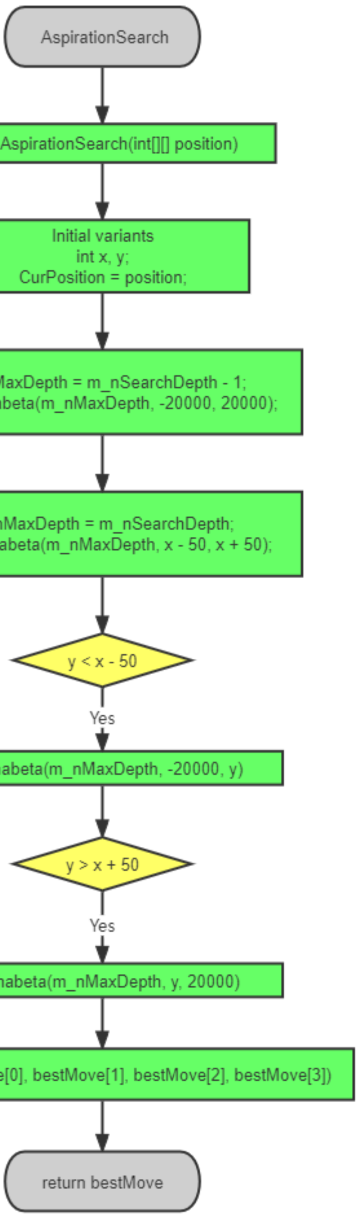
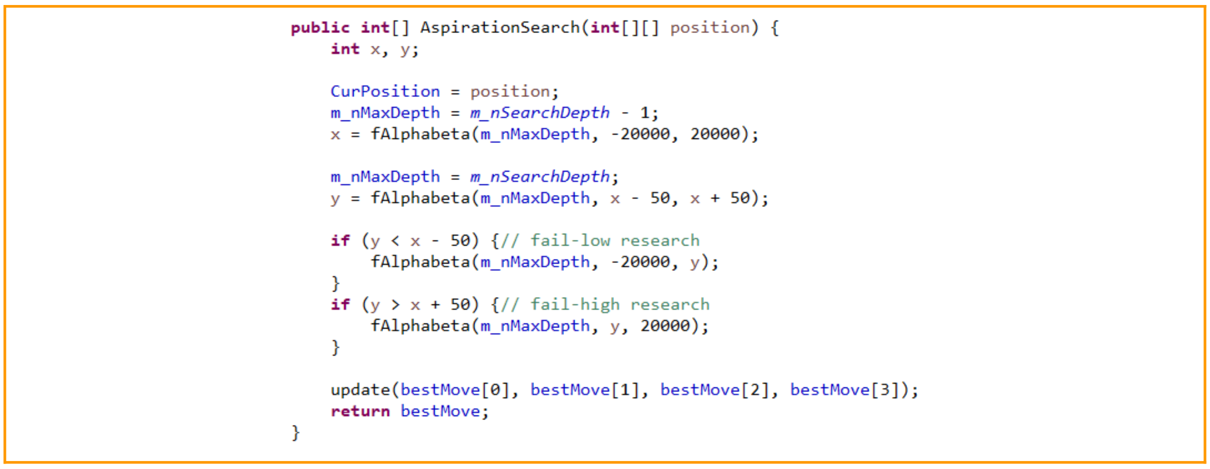

AspirationSearch
int[] AspirationSearch(int[][] position)
Initial variants
int x, y;
CurPosition = position;
m_nMaxDepth = m_nSearchDepth - 1;
x = fAlphabeta(m_nMaxDepth, -20000, 20000);
m_nMaxDepth = m_nSearchDepth;
y = fAlphabeta(m_nMaxDepth, x - 50, x + 50);
y < x - 50
Yes
fAlphabeta(m_nMaxDepth, -20000, y)
y > x + 50
Yes
fAlphabeta(m_nMaxDepth, y, 20000)
update(bestMove[0], bestMove[1], bestMove[2], bestMove[3])
return bestMove

Figure 27: A flow diagram of AspirationSearch(int[][] position) method

We can see from Figure 27 that the Aspiration Search algorithm has initial x and y, which will be set as the window range. The value x and y will be assumed by calling the fAlphabeta(m_nMaxDepth, -20000, 20000) method. Then, if y < x - 50, this means the fail-low research and need to call fAlphabeta(m_nMaxDepth, -20000, y) method again. If y > x + 50, this means the fail-high research and need to call fAlphabeta(m_nMaxDepth, y, 20000) method again. Finally, update the best move and return it. The source code of the Aspiration Search algorithm is shown in Figure 28.



```java
public int[] AspirationSearch(int[][] position) {
    int x, y;

    CurPosition = position;
    m_nMaxDepth = m_nSearchDepth - 1;
    x = fAlphabeta(m_nMaxDepth, -20000, 20000);

    m_nMaxDepth = m_nSearchDepth;
    y = fAlphabeta(m_nMaxDepth, x - 50, x + 50);

    if (y < x - 50) {// fail-low research
        fAlphabeta(m_nMaxDepth, -20000, y);
    }
    if (y > x + 50) {// fail-high research
        fAlphabeta(m_nMaxDepth, y, 20000);
    }

    update(bestMove[0], bestMove[1], bestMove[2], bestMove[3]);
    return bestMove;
}
```

Figure 28: A source code of the Aspiration Search algorithm

Minimal Window Search implementation

The idea of the Minimal Window Search algorithm (PVS) is that In a game tree, assume the first move is the best. Then, the rest of the search is to build a very small search tree with a narrow window (v, v+1) each time. To implement the Minimal Window Search algorithm, I created an PVS(int depth, int alpha, int beta) method in the Robot class. The flow diagram of the PVS(int depth, int alpha, int beta) method is shown in Figure 29.

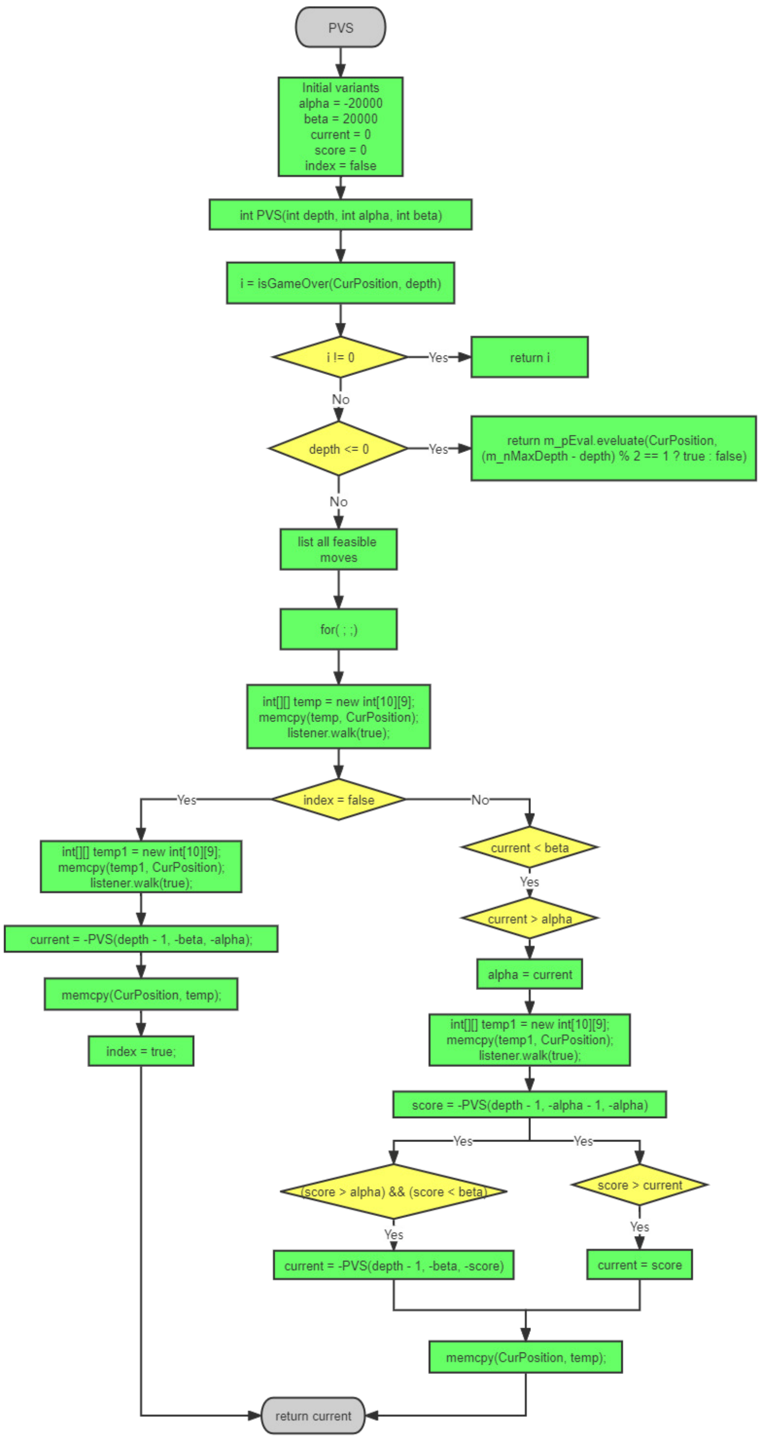
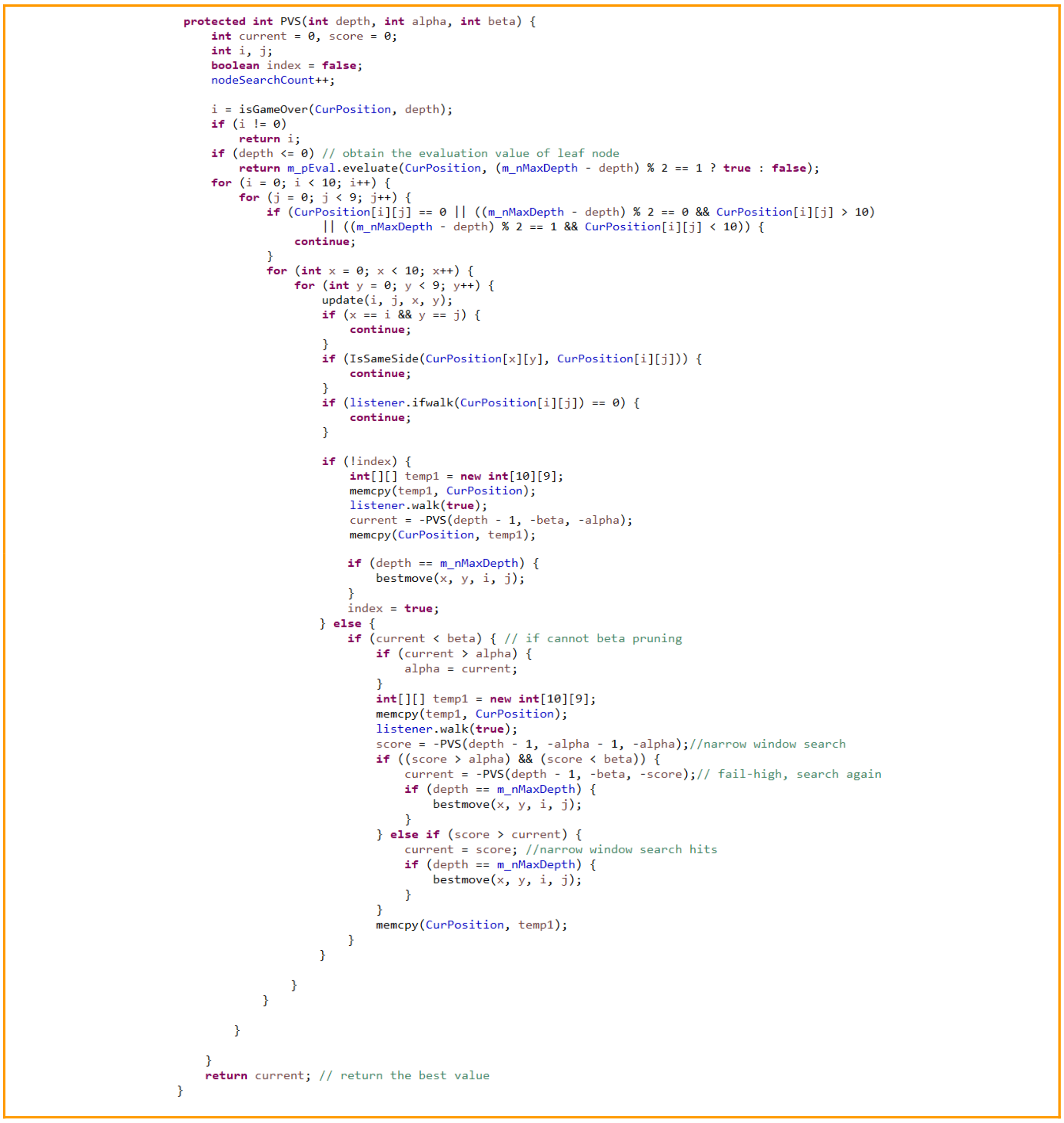

PVS
Initial variants
alpha = -20000
beta = 20000
current = 0
score = 0
index = false
int PVS(int depth, int alpha, int beta)
i = isGameOver(CurPosition, depth)
i != 0
Yes
return i
No
depth <= 0
Yes
return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false)
No
list all feasible moves
for(; ;)
int[][] temp = new int[10][9]; memcpy(temp, CurPosition); listener.walk(true);
index = false
Yes
No
int[][] temp1 = new int[10][9]; memcpy(temp1, CurPosition); listener.walk(true);
current = -PVS(depth - 1, -beta, -alpha);
memcpy(CurPosition, temp);
index = true;
current < beta
Yes
current > alpha
alpha = current
int[][] temp1 = new int[10][9]; memcpy(temp1, CurPosition); listener.walk(true);
score = -PVS(depth - 1, -alpha - 1, -alpha)
Yes
Yes
(score > alpha) && (score < beta)
score > current
Yes
Yes
current = -PVS(depth - 1, -beta, -score)
current = score
memcpy(CurPosition, temp);
return current

Figure 29: A flow diagram of PVS(int depth, int alpha, int beta) method

We can see from Figure 29 that the Minimal Window Search algorithm has initial variants alpha = -20000, beta = 20000, current, score and index. The boolean value “index” is to recognize the first move and the rest of moves. The first possible move is called the current = -PVS(depth - 1, -beta, -alpha) and let index = true. Then, the rest of the possible moves will use the narrow window to search. If it fails high, search again. The source code of the Minimal Window Search algorithm is shown in Figure 30.



```
protected int PVS(int depth, int alpha, int beta) {
    int current = 0, score = 0;
    int i, j;
    boolean index = false;
    nodeSearchCount++;

    i = isGameOver(CurPosition, depth);
    if (i != 0)
        return i;
    if (depth <= 0) // obtain the evaluation value of leaf node
        return m_pEval.eveluate(CurPosition, (m_nMaxDepth - depth) % 2 == 1 ? true : false);
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 9; j++) {
            if (CurPosition[i][j] == 0 || ((m_nMaxDepth - depth) % 2 == 0 && CurPosition[i][j] > 10)
                    || ((m_nMaxDepth - depth) % 2 == 1 && CurPosition[i][j] < 10)) {
                continue;
            }
            for (int x = 0; x < 10; x++) {
                for (int y = 0; y < 9; y++) {
                    update(i, j, x, y);
                    if (x == i && y == j) {
                        continue;
                    }
                    if (IsSameSide(CurPosition[x][y], CurPosition[i][j])) {
                        continue;
                    }
                    if (listener.ifwalk(CurPosition[i][j]) == 0) {
                        continue;
                    }

                    if (!index) {
                        int[][] temp1 = new int[10][9];
                        memcpy(temp1, CurPosition);
                        listener.walk(true);
                        current = -PVS(depth - 1, -beta, -alpha);
                        memcpy(CurPosition, temp1);

                        if (depth == m_nMaxDepth) {
                            bestmove(x, y, i, j);
                        }
                        index = true;
                    } else {
                        if (current < beta) { // if cannot beta pruning
                            if (current > alpha) {
                                alpha = current;
                            }
                            int[][] temp1 = new int[10][9];
                            memcpy(temp1, CurPosition);
                            listener.walk(true);
                            score = -PVS(depth - 1, -alpha - 1, -alpha);//narrow window search
                            if ((score > alpha) && (score < beta)) {
                                current = -PVS(depth - 1, -beta, -score);// fail-high, search again
                                if (depth == m_nMaxDepth) {
                                    bestmove(x, y, i, j);
                                }
                            } else if (score > current) {
                                current = score; //narrow window search hits
                                if (depth == m_nMaxDepth) {
                                    bestmove(x, y, i, j);
                                }
                            }
                            memcpy(CurPosition, temp1);
                        }
                    }

                }
            }

        }

    }
    return current; // return the best value
}
```

Figure 30: A source code of the Minimal Window Search algorithm

MTD(f) implementation

MTD(f) algorithm is to adjust the initial value when calling the Alpha-beta Search algorithm. MTD(f) searches the move using an empty window and adjusts the position of the empty window according to the search result. The narrower the search window, the faster the search. To implement the MTD(f) algorithm, I created an MTDf(int firstguess, int nDepth) method in the Robot class. The flow diagram of the MTDf(int firstguess, int nDepth) method is shown in Figure 31.

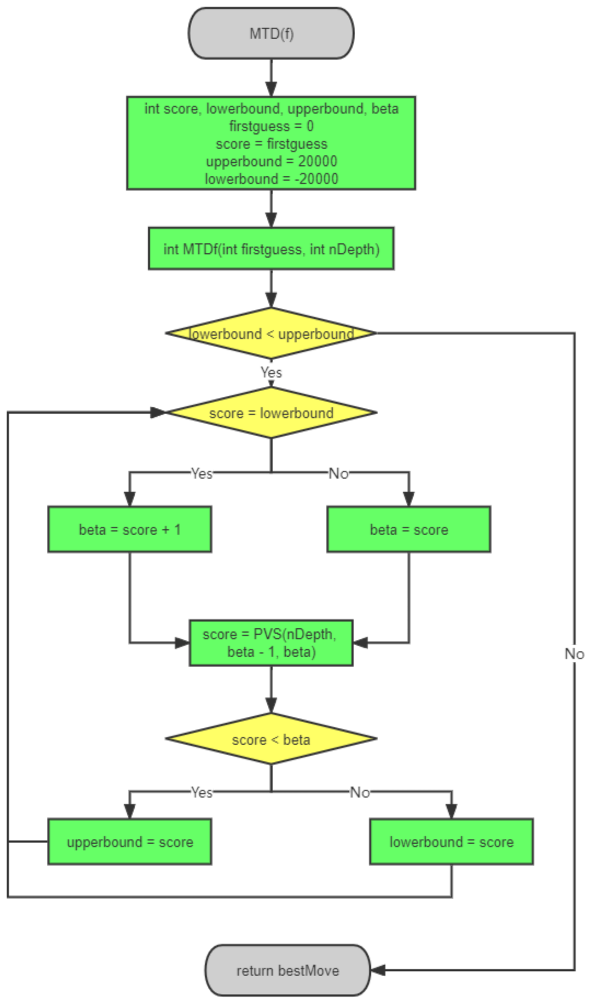
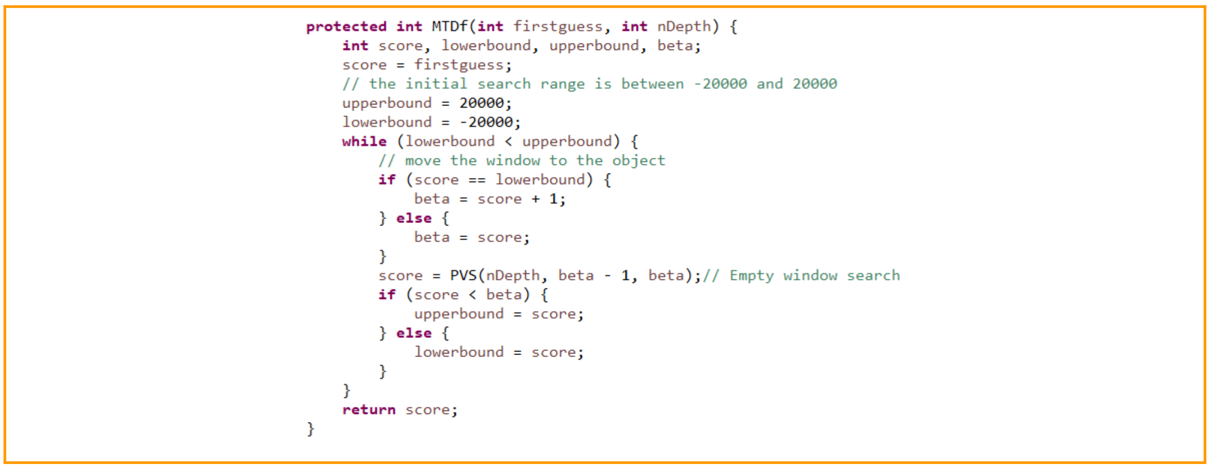


Figure 31: A flow diagram of MTDf(int firstguess, int nDepth) method

We can see from Figure 31 that the MTD(f) algorithm has initial variants lowerbound = -20000, upperbound = 20000, firstguess, score and beta. I set the initial guess value “firstguess” = 0. Then, use a while loop to narrow the window. While lowerbound < upperbound, use empty window search to adjust the range of the window. Finally, return the score. The source code of the MTD(f) algorithm is shown in Figure 32.

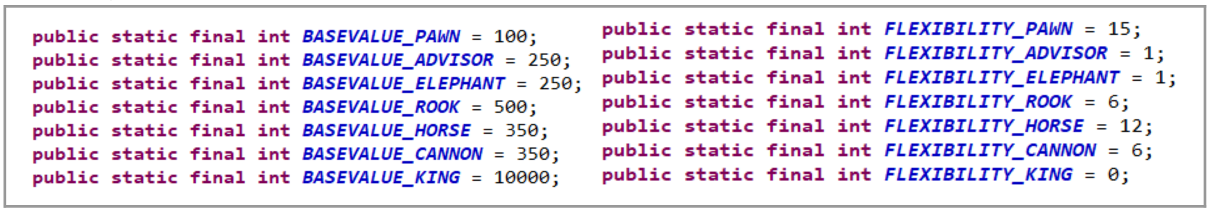


```
protected int MTDf(int firstguess, int nDepth) {
    int score, lowerbound, upperbound, beta;
    score = firstguess;
    // the initial search range is between -20000 and 20000
    upperbound = 20000;
    lowerbound = -20000;
    while (lowerbound < upperbound) {
        // move the window to the object
        if (score == lowerbound) {
            beta = score + 1;
        } else {
            beta = score;
        }
        score = PVS(nDepth, beta - 1, beta);// Empty window search
        if (score < beta) {
            upperbound = score;
        } else {
            lowerbound = score;
        }
    }
    return score;
}
```

Figure 32: A source code of the MTD(f) algorithm

Evaluation Function Implementation

To implement the evaluation function of AI Chinese chess, I created a separate Evaluation class in my Java project. In this class, I firstly define the variants which define the basic value and flexibility of each piece. Figure 33 shows the definition of the basic value and the flexibility value of each piece.



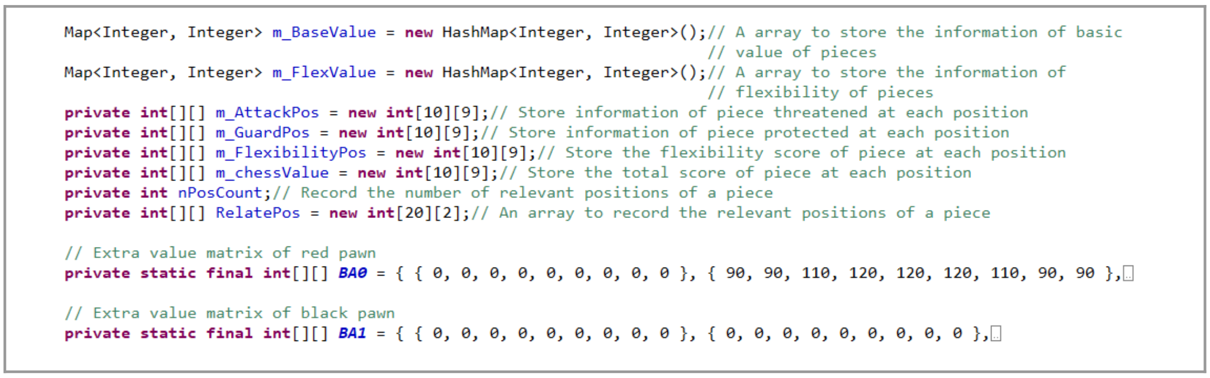
```java
public static final int BASEVALUE_PAWN = 100;
public static final int BASEVALUE_ADVISOR = 250;
public static final int BASEVALUE_ELEPHANT = 250;
public static final int BASEVALUE_ROOK = 500;
public static final int BASEVALUE_HORSE = 350;
public static final int BASEVALUE_CANNON = 350;
public static final int BASEVALUE_KING = 10000;
```

```java
public static final int FLEXIBILITY_PAWN = 15;
public static final int FLEXIBILITY_ADVISOR = 1;
public static final int FLEXIBILITY_ELEPHANT = 1;
public static final int FLEXIBILITY_ROOK = 6;
public static final int FLEXIBILITY_HORSE = 12;
public static final int FLEXIBILITY_CANNON = 6;
public static final int FLEXIBILITY_KING = 0;
```

Figure 33: The definition of the basic value and the flexibility value of each piece

Then, there are some arrays created to store these data including the basic value of pieces, flexibility value of pieces, piece threatened and protected at each position. All of these variants will be initialised in the Constructor Evaluation().

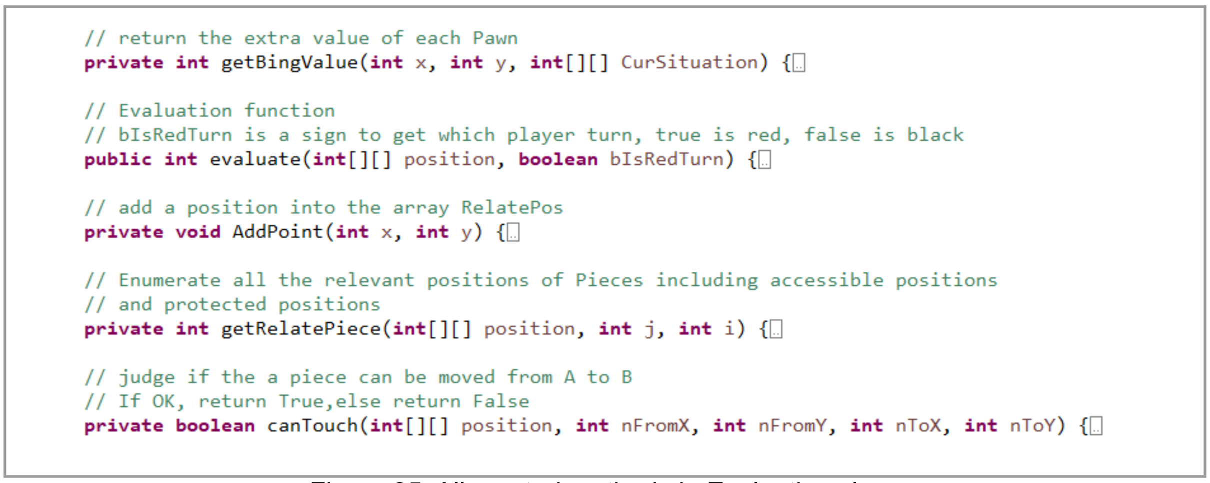
In the game board, the piece “Pawn” has extra value when it ”passes the river”. Hence, I create two 2-dimensional arrays “BA0” and “BA1” to store the ”Pawn” extra value. Figure 34 shows the array variants which are to store information.



```java
Map<Integer, Integer> m_BaseValue = new HashMap<Integer, Integer>();// A array to store the information of basic
                                                                    // value of pieces
Map<Integer, Integer> m_FlexValue = new HashMap<Integer, Integer>();// A array to store the information of
                                                                    // flexibility of pieces
private int[][] m_AttackPos = new int[10][9];// Store information of piece threatened at each position
private int[][] m_GuardPos = new int[10][9];// Store information of piece protected at each position
private int[][] m_FlexibilityPos = new int[10][9];// Store the flexibility score of piece at each position
private int[][] m_chessValue = new int[10][9];// Store the total score of piece at each position
private int nPosCount;// Record the number of relevant positions of a piece
private int[][] RelatePos = new int[20][2];// An array to record the relevant positions of a piece

// Extra value matrix of red pawn
private static final int[][] BA0 = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, { 90, 90, 110, 120, 120, 120, 110, 90, 90 },□

// Extra value matrix of black pawn
private static final int[][] BA1 = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, { 0, 0, 0, 0, 0, 0, 0, 0, 0 },□
```

Figure 34: The array variants which are to store information

A evaluate(int[][] position, boolean bIsRedTurn) method is created to implement the evaluation function. Other relevant methods are also needed in this class to be called. Figure 35 shows all created methods in Evaluation class and the function of these methods are described in the annotation.



```java
// return the extra value of each Pawn
private int getBingValue(int x, int y, int[][] CurSituation) {□

// Evaluation function
// bIsRedTurn is a sign to get which player turn, true is red, false is black
public int evaluate(int[][] position, boolean bIsRedTurn) {□

// add a position into the array RelatePos
private void AddPoint(int x, int y) {□

// Enumerate all the relevant positions of Pieces including accessible positions
// and protected positions
private int getRelatePiece(int[][] position, int j, int i) {□

// judge if the a piece can be moved from A to B
// If OK, return True,else return False
private boolean canTouch(int[][] position, int nFromX, int nFromY, int nToX, int nToY) {□
```

Figure 35: All created methods in Evaluation class

Experiments

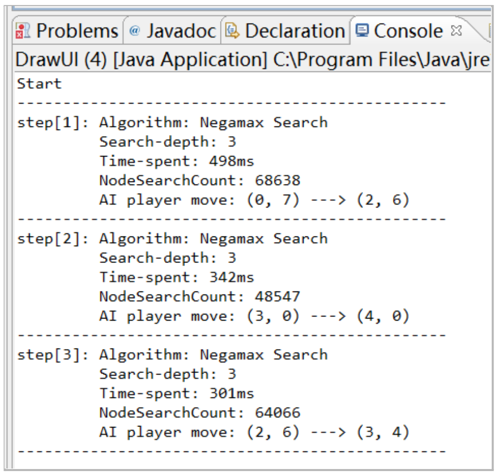
Experiments approach

In this section, experiments and tests for different AI techniques will be done to collect relevant data. And then evaluate and analyse the strength and feasibility of these AI approaches according to the experimental data. The experiment evaluation of AI approaches includes two aspects: 1. Comparison of search efficiencies between different AI approaches (AI & AI). 2 Evaluation of the effect of playing games against each other between human players and AI players (Human & AI).

Comparison of search efficiencies between different AI approaches (AI & AI) has two aspects including the average number of search nodes and the search time of AI players. In order to collect the average number of search nodes of AI players, I created a variable "nodesearchcount" in Robot class. Every time before calling the search algorithm, the variable will be set to 0, and then each time one node is searched, the value of this variant will be increased by 1. When the search ends, the value of this variant is the final number of search nodes. The data collection of average search time spent is also obtained in robot class. I use the currentTimeMillis() method in the Java System package to get the start time and end time of the search. Then, the search time spent can be obtained by "endtime - starttime".

Additionally, by playing games against each other between human players and AI players and then statistics on the winning percentage of AI players, it will be more intuitive to show the strength and effect of AI approaches. In the test, human players will not be allowed to use the "Withdraw" function to ensure the accuracy and fairness of the experiment.

All experiment data will be output on the console. In each step of the game, the data including selected algorithm, search depth, time spent, the number of search nodes and the move of the AI player will be shown on the console. Figure 36 shows an example of the result of console output.



```
DrawUI (4) [Java Application] C:\Program Files\Java\jre
Start
-----------------------------------------------
step[1]: Algorithm: Negamax Search
         Search-depth: 3
         Time-spent: 498ms
         NodeSearchCount: 68638
         AI player move: (0, 7) ---> (2, 6)
-----------------------------------------------
step[2]: Algorithm: Negamax Search
         Search-depth: 3
         Time-spent: 342ms
         NodeSearchCount: 48547
         AI player move: (3, 0) ---> (4, 0)
-----------------------------------------------
step[3]: Algorithm: Negamax Search
         Search-depth: 3
         Time-spent: 301ms
         NodeSearchCount: 64066
         AI player move: (2, 6) ---> (3, 4)
-----------------------------------------------
```

Figure 36: An example of the result of console output

Finally, I will organise these experiment data and show them in the form of tables so as to clearly show the comparison of different AI approaches, which will help me more conveniently evaluate and analyse the data.

Results and Evaluation

The Comparison of search efficiency (AI & AI)

Negamax Search & Alpha-beta Search

In this experiment, I compare the search efficiency between Negamax Search and Alpha-beta Search. The result is as follow:

The comparison of the average number of search nodes;

Algorithm \ Search Depth	1	2	3	4	5
Negamax Search	44	1637	67244	2294524	87270310
Alpha-beta Search	44	544	20549	119156	2068190

The comparison of the average search time spent;

Algorithm \ Search Depth	1	2	3	4	5
Negamax Search	1	81	4316	163481	6715029
Alpha-beta Search	3	29	392	4305	57286

The result shows that the average number of search nodes in Alpha-beta Search is obviously more than Negamax Search while the average search time spent is converse, which indicates that the search efficiency of Alpha-beta Search is completely better than Negamax. This results as I expected. Because Negamax Search needs to search the whole game tree, which leads to the low search efficiency while Alpha-beta Search to some extent prunes part of redundant children nodes, which improves the search efficiency, especially when the search depth is higher than 3.

Alpha-beta Search & Fail-Soft Alpha-beta Search

In this experiment, I compare the search efficiency between Alpha-beta Search and Fail-soft Alpha-beta Search. The result is as follow:

The comparison of the average number of search nodes;

Algorithm \ Search Depth	1	2	3	4	5
Alpha-beta Search	44	544	20549	119156	2068190
Fail-Soft Alpha-beta Search	44	552	21865	108327	1903691

The comparison of the average search time spent;

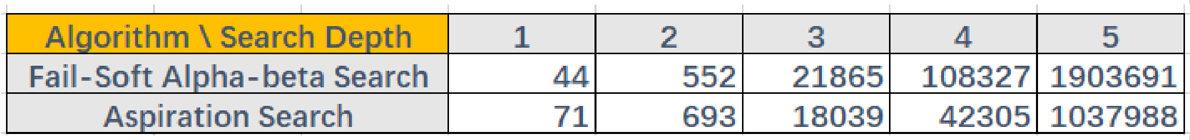
Algorithm \ Search Depth	1	2	3	4	5
Alpha-beta Search	3	29	392	4305	57286
Fail-Soft Alpha-beta Search	3	29	375	4162	51837

In this result, we can see that Alpha-beta Search and Fail-soft Alpha-beta Search have similar average number of search nodes and search time spent, which proves that their search efficiency is similar. When the search depth is higher than 4, the search efficiency of Fail-soft Alpha-beta search is slightly better.

Fail-Soft Alpha-beta Search & Aspiration Search

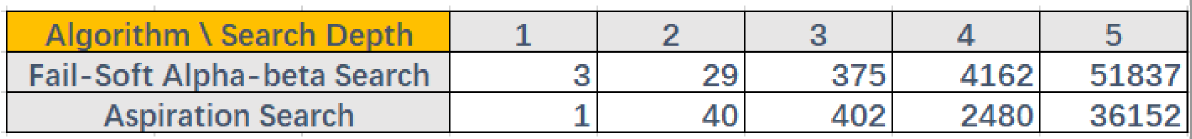
In this experiment, I compare the search efficiency between Fail-soft Alpha-beta Search and Aspiration Search. The result is as follow:

The comparison of the average number of search nodes;



Algorithm \ Search Depth	1	2	3	4	5
Fail-Soft Alpha-beta Search	44	552	21865	108327	1903691
Aspiration Search	71	693	18039	42305	1037988

The comparison of the average search time spent;



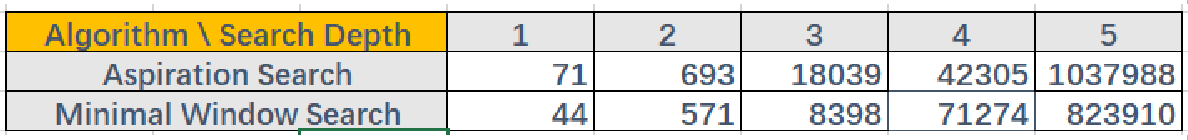
Algorithm \ Search Depth	1	2	3	4	5
Fail-Soft Alpha-beta Search	3	29	375	4162	51837
Aspiration Search	1	40	402	2480	36152

The result shows that the search efficiency of Aspiration Search is better than Fail-soft Alpha-beta Search. As the search depth increases, The advantages of Aspiration Search are more obvious. Because the assumed value in Aspiration Search can always be in the expected range, which saves nearly half time.

Aspiration Search & Minimal Window Search

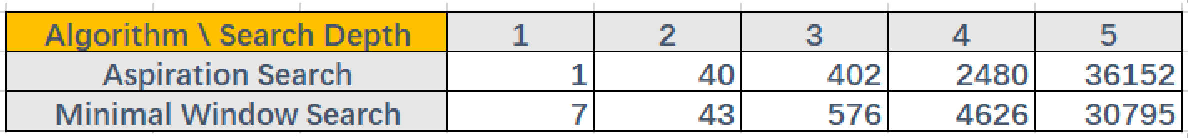
In this experiment, I compare the search efficiency between Aspiration Search and Minimal Window Search. The result is as follow:

The comparison of the average number of search nodes;



Algorithm \ Search Depth	1	2	3	4	5
Aspiration Search	71	693	18039	42305	1037988
Minimal Window Search	44	571	8398	71274	823910

The comparison of the average search time spent;



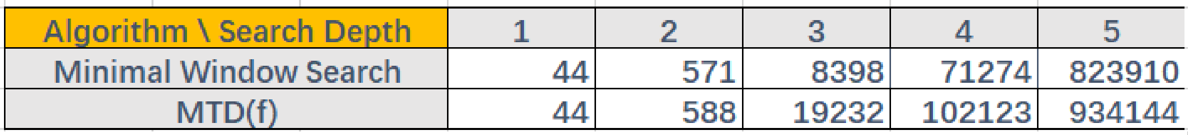
Algorithm \ Search Depth	1	2	3	4	5
Aspiration Search	1	40	402	2480	36152
Minimal Window Search	7	43	576	4626	30795

This comparison indicates that the search efficiency of Minimal Window Search is better than Aspiration in general. This is mainly because the setting of the narrow window of Minimal Window Search makes the search range become smaller. However, this setting of narrow windows exists a risk and accident because if the returned value of Minimal Window Search is out of bounds, it is necessary to search again, which leads to more time spent. We can see from the table that this risk occurs when the search depth is 3 and 4.

Minimal Window Search & MTD(f)

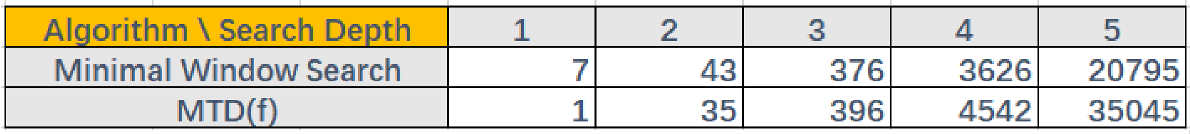
In this experiment, I compare the search efficiency between Minimal Window Search and MTD(f). The result is as follow:

The comparison of the average number of search nodes;



Algorithm \ Search Depth	1	2	3	4	5
Minimal Window Search	44	571	8398	71274	823910
MTD(f)	44	588	19232	102123	934144

The comparison of the average search time spent;

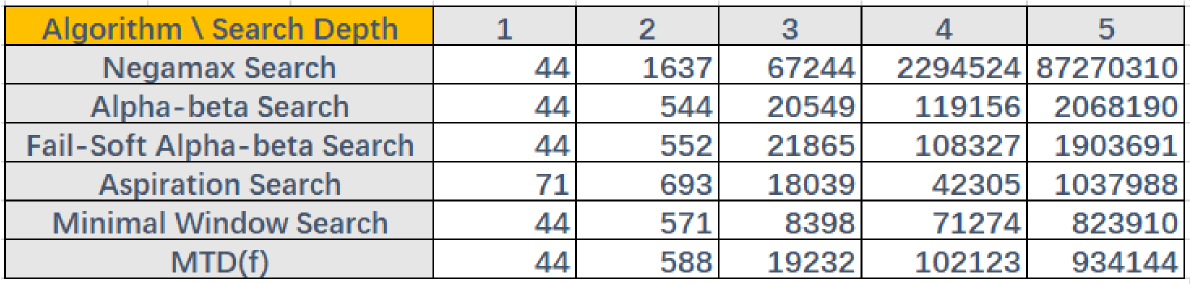


Algorithm \ Search Depth	1	2	3	4	5
Minimal Window Search	7	43	376	3626	20795
MTD(f)	1	35	396	4542	35045

The result shows that the search efficiency of MTF(f) is not better than Minimal Window Search and even not better than Aspiration Search. I think the reason is that the MTD(f) needs the Transposition Table to coordinate so as to save the time of searching again.

Summary

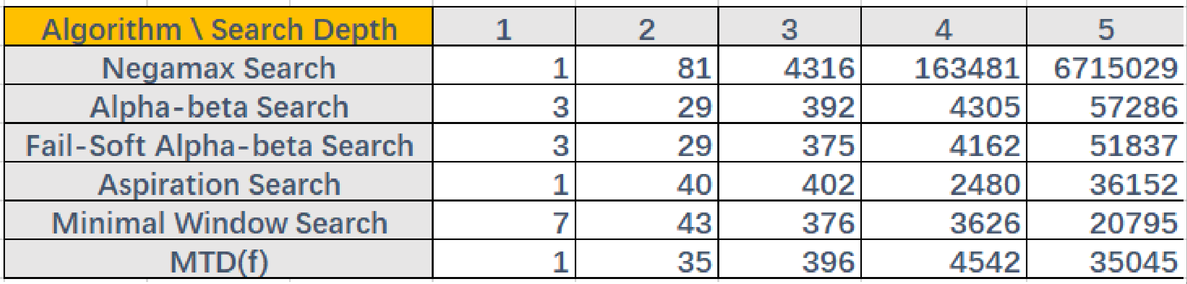
The comparison of the average number of search nodes;



Algorithm \ Search Depth	1	2	3	4	5
Negamax Search	44	1637	67244	2294524	87270310
Alpha-beta Search	44	544	20549	119156	2068190
Fail-Soft Alpha-beta Search	44	552	21865	108327	1903691
Aspiration Search	71	693	18039	42305	1037988
Minimal Window Search	44	571	8398	71274	823910
MTD(f)	44	588	19232	102123	934144

Figure 37: The comparison of the average number of search nodes of all algorithms

The comparison of the average search time spent;



Algorithm \ Search Depth	1	2	3	4	5
Negamax Search	1	81	4316	163481	6715029
Alpha-beta Search	3	29	392	4305	57286
Fail-Soft Alpha-beta Search	3	29	375	4162	51837
Aspiration Search	1	40	402	2480	36152
Minimal Window Search	7	43	376	3626	20795
MTD(f)	1	35	396	4542	35045

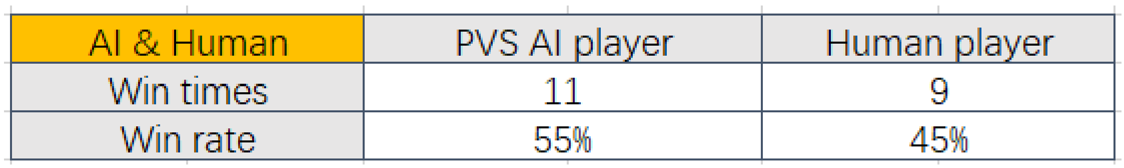
Figure 38: The comparison of search time spent of all algorithms

In Figure 37 and Figure 38, it is clear to see the comparison of the search efficiency of all AI algorithms. According to the data from these two tables, the Negamax Search is shown the worst search efficiency due to its feature of complete search game tree. Apart from the Negamax search, other five algorithms have the optimisation to some extent. Among these optimised algorithms, Minimal Window Search has better search efficiency than other algorithms. The experiment data still has occasionality, but in general, the result reached my expectation.

Human Player vs AI player Evaluation (Human & AI)

In this experiment, I selected the best AI player – Minimal Window Search(PVS) which is proved above to test and evaluate its performance by playing games against each other between human players and PVS AI players. There will be 20 matches played of this experiment between two players. I will statistics on the winning rate of the AI player in the end.

Minimal Window Search(PVS) AI Player vs Human Player



AI & Human	PVS AI player	Human player
Win times	11	9
Win rate	55%	45%

Figure 39: PVS AI player vs Human player

In Figure 39, we can see that in the 20 matches played between PVS AI player and Human player, the PVS wins 11 times and the Human player wins 9 times, which means that the winning rate of PVS AI players is more than 50%. This experiment result indicates that the PVS AI player is competitive and can give challenges to the Human player. However, the winning rate which is just over 50 % also indicates that the strength and performance of the AI algorithm still need to be further improved.

Future Work for the project

In general, I have achieved the original goal of the project. I succeeded in creating Chinese chess AI players. Multiple AI algorithms have been implemented, including all optimization algorithms based on the alpha-beta pruning algorithm. I am very pleased with my implementation of this Chinese chess project.

However, as can be seen from the experimental results and analysis, the AI players I have created are not the most perfect, and there are the deficiencies on performance to some extent. This is because the algorithms I have implemented so far are relatively Independent and cannot combine the advantages of multiple algorithms together to enhance the performance and strength of the algorithm.

Additionally, the existing AI players I have implemented are all based on the same set of evaluation functions, which makes these AI players essentially only have the difference in search efficiency but without the difference of strength and level among them.
Hence, in the future, I will go on trying to create new AI players that are completely different from the current ones, such as the AI players based on Monte Carlo search algorithm and reinforcement learning.

Improvements to existing AI algorithms

To address the problem that my existing AI algorithms are too independent to combine their respective advantages, I think it is necessary to research some enhancement algorithms that can cooperate with the Alpha-beta Search algorithm. In the background information I gathered before, I found that the Transposition Table algorithm is a very efficient enhancement algorithm. Transposition Table Search is a cache of previously seen positions and associated evaluations in a game tree generated in the game of Chinese chess. If a position recurs via a different sequence of moves, the value of the position is retrieved from the table, avoiding re-searching the game tree below that position [10]. Transposition tables are primarily useful in perfect-information games. This algorithm can be called before calling the Alpha-beta Search algorithm. If the Transposition Table finds the best move which is stored previously, the Alpha-beta Search algorithm will not need to be called again. The higher depth AI player search, the more obvious the effect of the Transposition Table algorithm.

In addition, although the existing pruning algorithm reduces the amount of search tree nodes, which improves the search efficiency, the efficiency is still not high enough. The search depth of existing AI players is 5 at most. When the search depth is higher than 5, the number of search nodes will be very huge, which will lead to too long search time being spent. Iterative Deepening Search algorithm overcomes this issue to some extent. The idea of the Iterative Deepening Search algorithm is to continue the search based on the best move found in the first search. This allows the search depth to be higher. With each iteration, the search depth is doubled. Iterative Deepening is not just a simple search, but also writes heuristic information for the implementation of the Transposition Table. The iteration can help the AI player to find the good search sequence of the next iteration. When iterative deepening mixing with Alpha-beta Search algorithm, the efficiency of iterative deepening will be further improved. Therefore, the Iterative Deepening Search algorithm plays an important role in improving AI algorithm performance.

Implementation of New AI player

Monte Carlo Tree Searching (MCTS)

Monte Carlo Tree Searching algorithm is the latest algorithm which is applied in the artificial intelligence of Chinese chess. The Monte Carlo method, which uses random sampling for deterministic problems which are difficult or impossible to solve using other approaches, dates back to the 1940s [11].

The idea of Monte Carlo Tree Searching is based on the analysis of the most promising moves, expanding the search tree based on random sampling of the search space. The most basic way to use playouts is to apply the same number of playouts after each legal move of the current player, then choose the move which led to the most victories [12].The implementation of MCTS of each step in Chinese chess contains four parts: Selection, Expansion, Simulation and Backpropagation.

Reinforcement Learning

Reinforcement learning is also an interesting AI approach which is applied in AI Chinese chess. This skill has been successfully applied in the game of Go. In 2016, the AI player AlphaGo, which is based on reinforcement learning beated the human champion Lee Sedol in Go. So, I think this skill can be good to apply to AI Chinese chess.

The principle of reinforcement learning is to learn a large amount of game cases in a big database to train the model to become an advanced AI player.

Conclusion

The project aims to investigate different AI techniques for creating multiple Chinese chess AI players and evaluate their performance through a series of tests and data comparisons. To achieve this goal, I divide my work into two parts: UI implementation and AI implementation.

UI implementation includes the implementation of creating the chessboard, moving the pieces, and setting the rules of the pieces. Besides, there is also the function of "withdraw"(back the chess move to the previous step). The successful creation of the UI is to provide a visual interface to the user, so as to let users better view the state of the game.game state.

AI implementation is a core and key part of this project. In order to achieve the creation of Chinese chessAI players, I have created two classes in JAVA project including Robot class and Evaluation class. Robot class contains all the algorithms I have implemented, including Negamax Search, Alpha-beta Search, Fail-soft Alpha-beta Search, Aspiration Search, Minimal Window Search and MTD(d). In this part, I spent a lot of time researching how to implement these AI algorithms. Evaluation class is to implement the evaluation function, which is used to give a score to each possible move to find the best move. The two classes, Robot class and evaluation class, work together to allow me to successfully create AI players.

Finally, I will evaluate the performance of each created AI player, so as to know the strength and level of these AI players.

Overall, I am very satisfied with the work I have achieved. Different AI players have been successfully created.

Reflection

In order to achieve the goal of the project, in the early stage of work, I spent a lot of time gathering background information related to AI Chinese chess. This is to let me have a better understanding of how this project is achieved and I can decide which AI approach I want to implement. This has given a good heuristic for later writing code to create Chinese chess AI players.

However, I think there are still many aspects of my project that can be further optimised and improved. First of all, although a number of AI players have been successfully created, these AI players are not strong enough and are only suitable for confrontation with amateur Chinese chess Human players, and cannot form a threat to top Chinese chess Human players so far. Hence, I think I need to further research some enhancement algorithms, such as the Transposition Table and Iterative Deepening algorithm, so as to improve the strength of Chinese chess AI players.

In addition, the experiment evaluation for AI players needs to be more comprehensive and systematic. At this stage, the sample size of the experimental data is relatively small, which leads to the possible deviation of the experiment data and affects the accuracy of the evaluation.

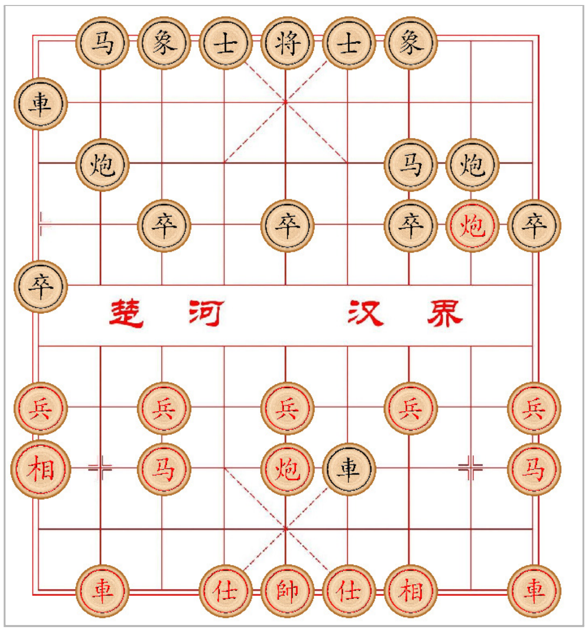
Finally, I also hope to implement more advanced AI techniques such as Monte Carlo search and Neural Networks. If these can be achieved, the level and strength of AI players will be greatly improved.

Table of Figures

Appendices

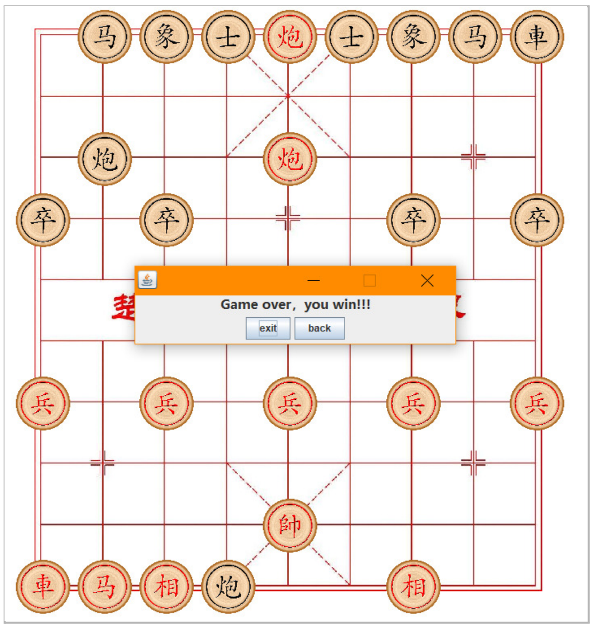
Appendix 1

Example of Chinese chess game states



Appendix 2

Example of the game state when the red player wins



Game over, you win!!!

References

[1] Albright, Daniel. 26th September 2016. 10 Examples of Artificial Intelligence You're Using in Daily Life [Online]. Available at: http://beebom.com/examples-of-artificial-intelligence/. Last accessed 31st January 2017.

[2] J.E. Laird, "Using a Computer Game to Develop Advanced AI," Computer, vol. 34, no. 7, pp. 70-75, July 2001.

[3] Deng, Li. "AI Agent for Chinese Chess."

[4] Wikipedia. Xiangqi, April 2016. URL https://en.wikipedia.org/wiki/Xiangqi.

[5] Von Neumann, J. (1928). "Zur Theorie der Gesellschaftsspiele". Math. Ann. 100: 295–320. doi:10.1007/BF01448847.

[6] Wikipedia. Xiangqi, April 2016. URL https://en.wikipedia.org/wiki/Minimax.

[7] Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc. p. 167. ISBN 978-0-13-604259-4.

[8] "Adaptive Strategies of MTD-f for Actual Games". Tokyo University of Agriculture and Technology. K SHIBAHARA et al

[9] Shannon, Claude (1950), Programming a Computer for Playing Chess (PDF), Ser. 7, vol. 41, Philosophical Magazine, retrieved 12 December 2021

[10] Wikipedia. Transposition_table, April 2016. URL https://en.wikipedia.org/wiki/Transposition_table.

[11] Nicholas, Metropolis; Stanislaw, Ulam (1949). "The monte carlo method". Journal of the American Statistical Association. 44 (247): 335–341. doi:10.1080/01621459.1949.10483310. PMID 18139350.

[12] Brügmann, Bernd (1993). Monte Carlo Go (PDF). Technical report, Department of Physics, Syracuse University.

[13] Xiaochun Wang. PC game programming (game theory)[M]. Public of Chongqing University, 2002.

[14] CSDN. Implementation of Chinese chess using Java, August 2019. URL https://blog.csdn.net/weixin_44547562/article/details/99711390.