EXPLORING STRATEGIES THROUGH EVOLUTIONARY ALGORITHMS AND NEURAL NETWORKS

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

by

Chiun Lin Lim

Project Advisor: David Delchamps

Degree Date: January 2009

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: Exploring strategies through evolutionary algorithms and neural networks.

Author: Chiun Lin Lim

Abstract:

This project builds a Chinese Chess program with an artificial intelligence that is completely dependent on the position evaluation function. The position evaluation function computes piece values, position values and neural network outputs. Players with different position evaluation function are evolved through single population competitive coevolution, and host-parasite competitive coevolution, which evolves players with strong offensive or strong defensive trait simultaneously to accelerate the learning process. Using hypothesis testing, the evolved players are found to be significantly stronger than the non-evolved players.

Report Approved by

Project Advisor: ____Date:____________

Summary of Accomplishments

The two main tasks of this project are to build a Chinese Chess program and implementing the evolutionary algorithms.

I manage to build a graphically displayed Chinese Chess program that plays according to nearly all rules. The program is equipped with a simple opening book and has the ability to generate all legal given any board position. The program is also able to determine the optimal move for any board position for any search depths. The optimal move is determined using a position evaluation function and efficient search algorithms. The position evaluation function includes piece values, position values and neural networks. Alpha-beta pruning, move ordering, quiescence search and null pruning are implemented to reduce the time taken to find the optimal move.

To evolve the position evaluation function, single population competitive coevolution and host-parasite competitive coevolution are implemented. Each model is initialized, and players undergo competition, selection, crossover, and mutation for 50 generations. The resulting best players are compared to the non-evolved players and analysis is carried out using hypothesis testing.

Table of Contents

1. Background 1
2. Motivation 1
3. What is a strategy? 2
4. Project objective 2
5. Why neural network 3
6. Introduction to Chinese Chess 4
7. Complete Project Specifications 4
   i. Constraints 4
   ii. Complete and functional Chinese Chess program 5
   iii. Evaluation function and search algorithms 5
   iv. Evolutionary algorithm 6
   v. Simulation 6
8. Work completed 6
   i. Meeting the constraint 6
   ii. Graphic display of board 7
   iii. Plays the game according to the rules 8
   iv. Generating all possible moves 9
   v. Opening book 11
   vi. Evaluation 12
      a. Piece value 12
      b. Position value table 13
      c. Neural network 15
   vii. Search algorithms 20
      a. Alpha-beta pruning 20
      b. Move ordering 22
      c. Horizon effect and quiescence search 24
      d. Null pruning 26
      e. Putting it all together 27
   viii. Competition, selection, crossover, mutation 29
9. List of all simulation parameters 29
10. Analysis and result 30
   i. Single population competitive coevolution 30
   ii. Host-parasite competitive coevolution 33
11. Discussion 36
12. References 37
13. Appendix 38

1. Background

Like Chess, Chinese Chess is one of the most popular strategic board games worldwide with a complexity level between that of Chess and Go [1]. After Deep Blue defeated the former chess world champion, Gary Kasparov in 1997, Chinese Chess is projected to be the next strategic board game in which a computer program will defeat the top human player [1].

Currently, Yin-Chuan Xu is widely regarded as the strongest human player in Chinese Chess while the best Chinese Chess program, based on the results of the World Computer Chinese Chess Championship 2007[1], is NEUChess [2]. Yin-Chuan Xu and NEUChess played against each other in August 2006 and the match ended in two draws [4]. The results of the match were largely inconclusive, however. First, it was an exhibition match meant to showcase the program's strength. Second, the base time allocated to each side was a mere 45 minutes while competitive matches typically have a base time of at least 90 minutes.

NEUChess, similar to most other programs, based its strength on its database of stored games and expert knowledge and on efficient search algorithms to find the optimal move. The current focus of most effort to improve the computer programs' strength is to further refine the database, getting more computational power and performing an increasingly complex search strategy to allow deeper searches. In a sense, we are "relying on human expertise to defeat human expertise" [5] and no actual machine learning is done.

2. Motivation

Given the increase in computer processing speed, it should be anticipated that the computer program will defeat the top human players in a few years. After Chess and Chinese Chess, the next challenge will be the more complicated Shogi and Go. At the moment, the best Go program can be beaten by amateur players and are still very far from posing a challenge to human players.

Go is so distinct from Chess and Chinese Chess that a lot of important game programming techniques are not applicable here [6]. Its enormous state space also implies an increase in computer processing speed is unlikely to improve the computer

[1] In the recently concluded World Computer Chinese Chess Championship 2008, Intella placed first placed while NEUChess has dropped to 6th [3].

program by much. Human expertise is also harder to quantify here as human strategies often stretch out over the whole board and over many moves.

All these problems mean that if a program capable of beating human players is ever created, I feel that it is more likely that the program has incorporated machine learning instead of being entirely reliant of encoded human knowledge. This is the reason why I choose to pursue the topic of machine learning.

3. What is a strategy?

A strategy in Chinese Chess consists of position evaluations and long-term goals. Position evaluation takes into account value, position, and mobility of pieces, structure of defensive pieces, control of key points, etc. All these factors are not constant throughput the game or depend solely on the position of the game. Rather, these factors vary according to the presence and position of all other pieces on the board. For instance, a pawn is relatively useless early game but is as powerful as a knight or cannon late game when there are only a few attacking pieces left on the board. Long-term goal is a broad term which refers to what a player wishes to achieve. It may be that the player wishes to draw the game and hence a possible strategy is to start with a more defensive opening. Other long-term goals may include moving several pieces into certain positions on the board, trapping and capturing a piece, etc. As its name implies, formulating long-term goals require seeing far into the game.

The focus of this project is on position evaluation. In particular, a “player” consists of a position evaluation function which translates the aforementioned factors into numerical values. The factors to be included and how they are translated into numerical values are what this project is about.

4. Project Objective

This project develops a self-learning Chinese Chess program using a crossover of the ideas of coevolutionary approaches [7] and neural networks [8].

Fogel’s experiment [8] evolves players which represent position evaluation functions consisting of the values of Chess pieces, the position value tables of each piece and three separate neural networks covering three non-overlapping regions of the board. Each player competes by playing a certain number of games against randomly selected opponents (with replacement and not including self). Selection is

achieved through an elitism scheme which retains the best half of the population to be parents of the next generation. The parents have their piece values, position value tables, and the biases and weights of the neural networks mutated to produce the next generation.

In contrast, Ong et al [7] implemented three models, namely single population competitive coevolution, host-parasite competitive coevolution, and “divide-and-conquer” cooperative coevolution. The models are used to evolve the value of Chinese Chess pieces and their position value tables, which are encoded as a string of bits. Evolution is performed in a similar manner as Fogel’s except that mutation is performed on the string of bits instead of each value separately. The second model consists of two populations – the host and the parasite. Host players are penalized for losing while parasite players are penalized for not winning. This scheme tends to develop host players with strong defensive trait and parasite players with strong offensive trait. The resulting arms race should accelerate the learning process. Lastly, the third model employs “divide-and-conquer” by evolving the piece values and position value tables separately. Each component is evolved independently and evaluation of the fitness function is done by combining the collaborators.

The project will merge the ideas of both papers by implementing the single population competitive coevolution and host-parasite competitive coevolution. Both models will evolve value of the pieces, the position value tables of each piece and the biases and weights of three neural networks.

5. Why neural network

The idea of neural network appeals to me as it appears that it can evaluate the board position by taking into account the interaction between many pieces. The position evaluation function of computer programs has always been about piece values, position values, mobility, etc. Furthermore, all these factors are evaluated by looking at each piece separately, without taking into account the effect of other pieces. However, games like Chess and Chinese Chess are not about the sum of the parts. Neural networks appear to be able to encode some of that information, as its output depends not just on the pieces, but on the positions of all pieces in the network.

6. Introduction to Chinese Chess

Many excellent introductions to Chinese Chess can be found on the web. However, most Chinese Chess terminologies in English are translated from Chinese and the translated terms differ from author to author. If I refer the reader to any of those introductions, ambiguities and confusions are bound to arise when I use a different term in my project. Therefore, I choose to write a short guide of my own which is included in the Appendix. The guide will introduce the conventions of terms that I will stick to throughout this project.

7. Complete Project Specifications

As this is an independent project that I proposed, I am solely responsible for the completion of the whole project. The main components of the projects are listed below and detailed specifications for each component will follow in the later sections.

i) Design a complete and functional Chinese Chess program.
ii) Design an evaluation function that incorporates neural networks and design an efficient search algorithm.
iii) Setting up the evolutionary algorithms.
iv) Running the simulation, analyzing the results and writing up the report.

i. Constraint

The main constraint for the project is a reasonable simulation time. I anticipated the project will involve a significant amount of coding and it is unlikely that my code will be bug free and so the first few simulation runs are likely to produce the wrong result. Taking that as well as the amount of program runs required for analysis of the result into account, I set a constraint of one week as the maximum simulation time allowed. Preferably, the simulation should take less than half a week. A significant portion of the project is on how to meet the constraint.

Before proceeding further, I derived an approximation to the simulation time:

$$\begin{aligned} S = {} & \text{time taken to make a move} \times \text{number of moves per game} \\ & \times \text{number of games per generation} \times \text{number of generations} \end{aligned}$$

I set the number of moves per game to be 50 since that is the number of moves require for most games to have a clear winner. The number of games per generation is taken to be 190 (the number of games needed for 20 players to play against each other in a round-robin fashion). The number of generations is 50. The only undetermined

variable is the time taken to make a move and that depends on how the program is implemented. Plugging in the numbers substituting S with one week, I find that I need the time taken to make a move to be less than **1.27 seconds**. With this benchmark, I can make trial runs with just a half-completed program to determine if my program is on course to achieving the constraint.

To prevent further confusions about the term move, here are the exact definitions:

i) A **move** consists of a **turn** by each player. In a turn, a player shifts the position of one of its piece to another legal position.
ii) A **ply** is one turn taken by a player. So a ply is actually a half-move.

I am going to use the terms turn and ply exactly as defined. The term move will be used as defined or loosely as being the act of a player shifting the position of its piece (making a move). Specifically, generating all possible moves for a player means to generate all the possible legal positions the player can shift his pieces to in a turn.

ii. Complete and Functional Chinese Chess Program

A Chinese Chess program is to be coded in a suitable programming language with the following features:

i) For debugging purposes, a graphic display to show the current board position (board position means the position of all pieces on the board).
ii) Plays the game according to the rules, which include piece movements, checking, victory condition, repetition and chasing, etc.
iii) A function to generate all the possible moves for a given board position.

iii. Evaluation Function and Search Algorithms

The strength of a computer program depends on two main components – position evaluation function and searching heuristic. A position evaluation function translates each possible board position into a value so that different board positions can be compared to determine which position is more advantageous to a player. Searching heuristics are algorithms to efficiently search through the huge number of possible board positions to find the optimal move using the position evaluation function.

The position evaluation function consists of 3 components – piece values, position value tables and 3 non-overlapping neural networks. The efficient search

algorithms used are alpha-beta pruning, move sorting, quiescence search, and null pruning.

iv. Evolutionary Algorithm

The two main components of an evolutionary algorithm are the players and their evolutions. Each player is just a different position evaluation function. The tasks in this section are:

i) Initialization of the pool of players.
ii) A function to perform competition, selection, crossover, and mutation.

The tasks are completed for both evolution models. It should be noted that the main implementation differences between single population competitive coevolution and host-parasite competitive coevolution are only in the number of populations (1 vs. 2) and the fitness function.

v. Simulation

If everything works perfectly, the simulation should run fine and I can proceed to do analysis of the results. However, my program is expected to be long (turns out to be more than 2000 lines of code) and it should not be unexpected that there are quite a few bugs to catch before the simulation will run properly to produce the desired result.

8. Work Completed

In this section, I'll cover the completed design; the reasons behind the design choices, problems faced and unsolved problems.

i. Meeting the constraint

As mentioned before, the simulation needs to be completed within a reasonable amount of time. After some contemplation, I first decided to program in Matlab. I realized that Matlab is slower than programming languages like C++. However, I have not coded in other programming languages in a while. In addition, I have used Matlab to make programs like Sudoku Solver before. My prior experience led me to feel that coding, debugging and analysis of the results will be easier in Matlab with its less strict syntax, stored variable space and ease of plotting figures. If the program was slow, I thought I could have used the time saved to tweak it to meet the constraint.

However, upon completing the Chinese Chess program and implementing a search algorithm on a stub evaluation function, I found that Matlab was taking over 10 seconds to make a move. That was about 1 order of magnitude above benchmark and if the actual evaluation function was implemented, the time taken would have increased by a few folds. No amount of tweaking could bring it down to benchmark so I decided to switch over to C++.

In the following section, I will go in detail about the completed work. All the codes[2] are compiled in C++ using Microsoft Visual Studio 2005 and are included in the appendix. The order of presentation in the following sections is not necessarily the same as the order of completion.

ii. Graphic Display of Board

I programmed the graphical display in Win32 and it is shown in as Fig. 1. The graphical display shows the board, the piece movement, simulation time, position value and game outcome. Note that the piece movement is shown through an unconventional coordinate system which was chosen as it was easier to code, display and check.

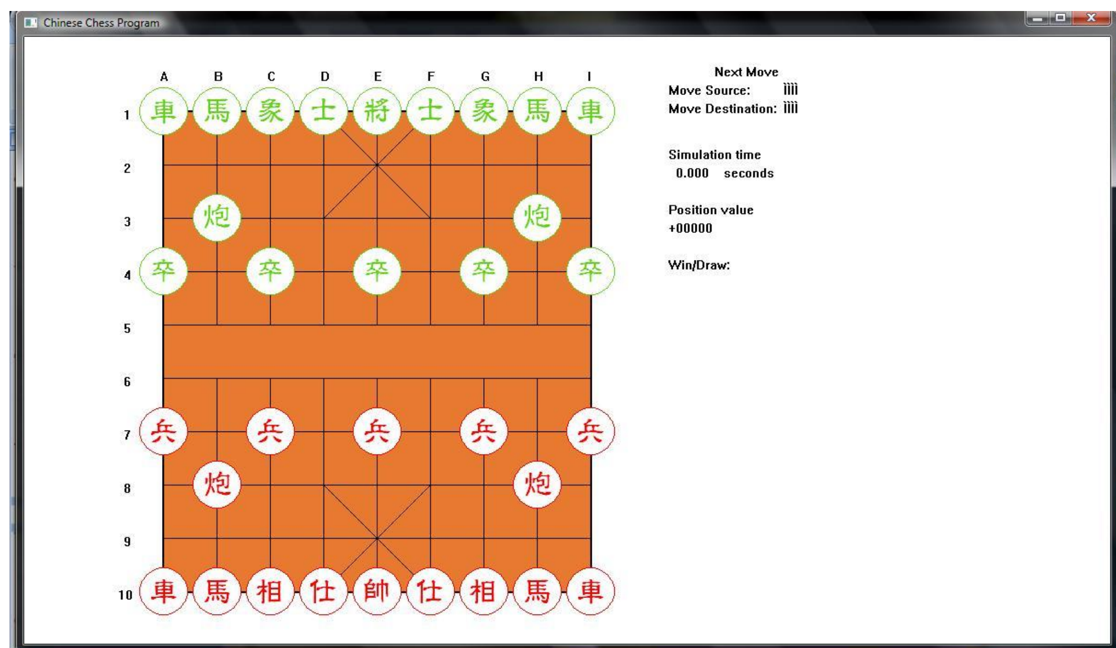

Figure 1. Graphical Display of my Chinese Chess program

[2] The code is not platform independent. I have encountered problems running or compiling it on another computer running a different operating system.

The graphical display is used solely for debugging purposes. Through it, I can check the program functions work as it should such as pieces moving according to the rules and search algorithm producing a reasonable move, etc. It also allows me to get a sense of how smart the computer programs are and how the game typically evolves.

iii. Plays the game according to the rules

The program plays according to all rules except for repetition. The rules are enforced through move generation. All the generated moves are legal except when repetition is involved. There are two types of repetition — perpetual check and perpetual chasing (chasing is defined as threat of capture) of an unprotected piece. The two situations are illustrated in Fig. 2 and 3.

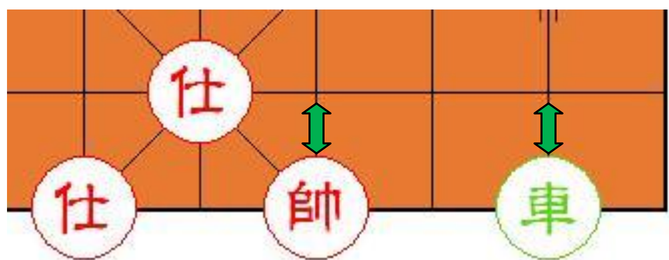

Figure 2. Perpetual checking

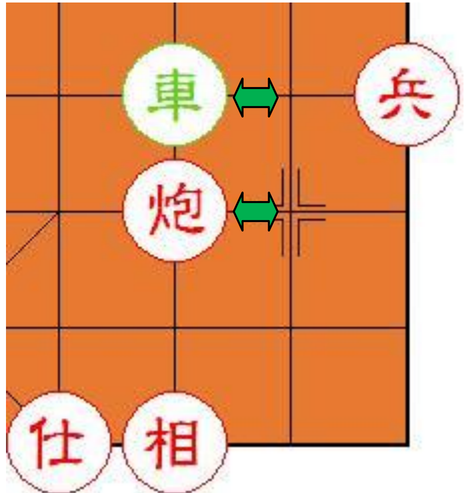

Figure 3. Perpetual chasing of cannon by a rook

The exact rules governing repeated positions are complicated. Both perpetual checking and perpetual chasing by rook are illegal. Perpetual chasing using king or pawn is allowed, however. In situations in which both checking and chasing are involved, whether the moves are illegal or not depends on the pieces involved and the

board positions. The rules are so complicated that it is not uncommon in tournaments that players disagree and the arbiter has to be called on to make the final decision.

Solving the repeated position problem (also known as the repetition detection problem[3]) requires implementing transposition tables using Zobrist key. Zobrist key is an efficient way to represent each board position with a randomly generated bitstring. The number of possible bitstring is smaller than the number of possible board positions but is huge enough such that it can be assumed that the board positions that map to the same bitstring are very different. The transposition table stores the board position after each move as a Zobrist key. Therefore, to check if we are repeating the same position, we just need to look up the transposition table, and check if the current board position's Zobrist key matches with any Zobrist key a few moves back. In theory, all these sound simple and straightforward but I had a hard time integrating it correctly because a complete implementation requires substantial modifications to the searching algorithm. I gave up trying to implement it after two weeks.

Without repetition detection, about 20 to 30% of the games get stuck in repeated positions and end up as draw when the maximum number of moves allowed is reached. With the graphical display, I look through the games and less than half of these are games with a clear winner. Though the majority of the games end without ever getting stuck in repeated position, it should be noted here that since a draw is favorable to the host in the host-parasite coevolution model, it will be interesting to see if any hosts evolve the trait of forcing repeated positions to draw the game.

iv. Generating All Possible Moves

The board position is represented with two arrays. The first is an array with 256(16x16) elements, `square[256]`. All the elements are zero except for positions occupied by a piece, in which case, the element takes on the piece number (see Fig. 4). The piece number is a unique number given to each of the 32 pieces.

There are two reasons behind the length of the array. Chess has 64 = 8x8 squares and it is straightforward to use a 64-element array but as Chinese Chess has 9x10 points, a longer array is needed. 256 is chosen instead of 90 or any other length because it allows easier and faster move generation using hexadecimal notation. For instance, the forward position of a black pawn is just 0x0f + current position.

[3] A good way to check the strength of a computer chess programs on cell phones or handheld computers is to see if it implements repetition detection.

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,39,37,35,33,32,34,36,38,40, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0,41, 0, 0, 0, 0, 0,42, 0, 0, 0, 0, 0,
0, 0, 0,43, 0,44, 0,45, 0,46, 0,47, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,27, 0,28, 0,29, 0,30, 0,31, 0, 0, 0, 0,
0, 0, 0, 0,25, 0, 0, 0, 0, 0,26, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,23,21,19,17,16,18,20,22,24, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4. Array to represent the initial board position. The region enclosed by the red line is the board

The `square` array alone is sufficient to encode all the necessary information about the board but it is more convenient in certain situations to know where the pieces are instead. Therefore, the second array is a 64-element array, `piece[64]` containing the positions of the pieces in the `square` array. The position occupied by each piece in the `piece` array is the same as its piece number. To clarify further, consider the red king, which has the unique piece number 16. `square[piece[16]]` gives the number 16. In Figure 4, the red king is at `square[199]`. So, `piece[square[199]]` gives the number 199.

With these two representations, each ply is represented by two numbers, the `square` position of the piece moved (source) and the `square` position of the piece destination (destination). For instance, the movement of the red king one square forward in Figure 4 is represented by 199 (source) and 183 (destination). All the generated moves will follow this move representation.

Move generation is thus accomplished by taking each piece of the side making the move and generating all the possible moves for each piece according to the rules. Since a lot of games are going to be simulated, for any position on the board, any piece that can reach there is going to occupy it multiple times over the same game or over different games. To save time, instead of generating the moves again each time the piece reaches the same position, all the possible moves can be pre-generated for each piece for each position of the board beforehand.

Such a pre-generation method works fine except for rook and cannon, since their possible moves depend on where other pieces are. Both rook and cannon's mobility are limited if they are blocked by other pieces. The number of moves for each position while including all other pieces is too huge to be stored. Also, at each move, due to their high mobility, the number of possible moves for rook and cannon is about half of the total number of possible moves. Thus, the time savings by pre-generation without rook and cannon is actually minor.

I stumbled upon the solution later in the open source Chinese Chess program, xqwizard. It uses bit-masking to achieve some sort of pre-generation. The method is complicated and since it is not my work, I am not going to go into it here. I should mention that I copied and used the code for pre-generation of rook and cannon as well as other pieces from xqwizard.

v. Opening Book

The first ply, unlike all other plies, uses an opening book. The purpose of an opening book is to introduce variation into the players' moves. Without it, for the first few generations, most of the games played will have the same first move.

The opening book has four variations, the first three (shown in Fig. 5, 6, and 7) of which are opening moves which are most frequently used in tournaments. The fourth option is to let the player decide the opening move instead.

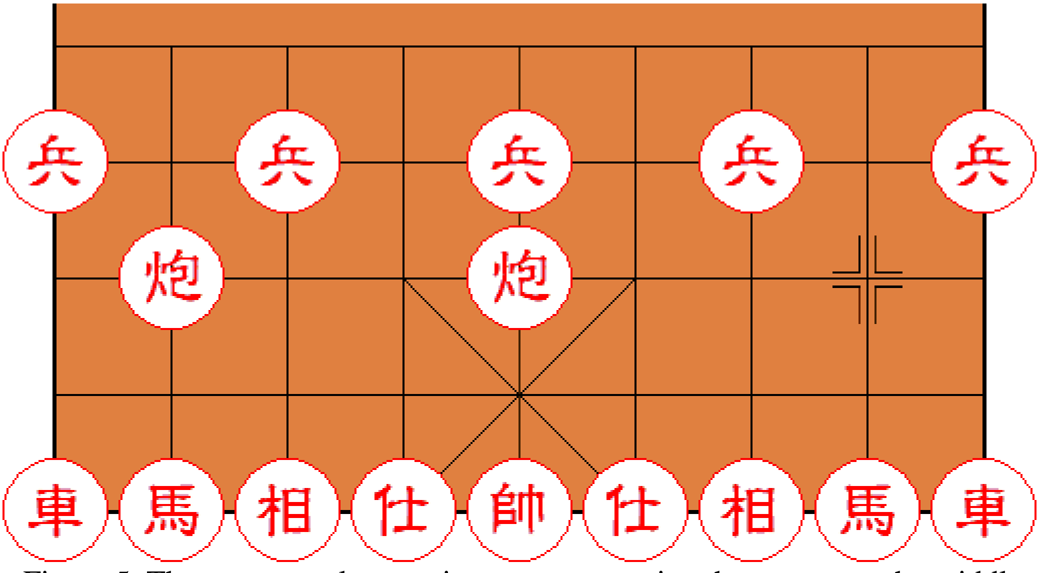

Figure 5. The most popular opening move -- moving the cannon to the middle

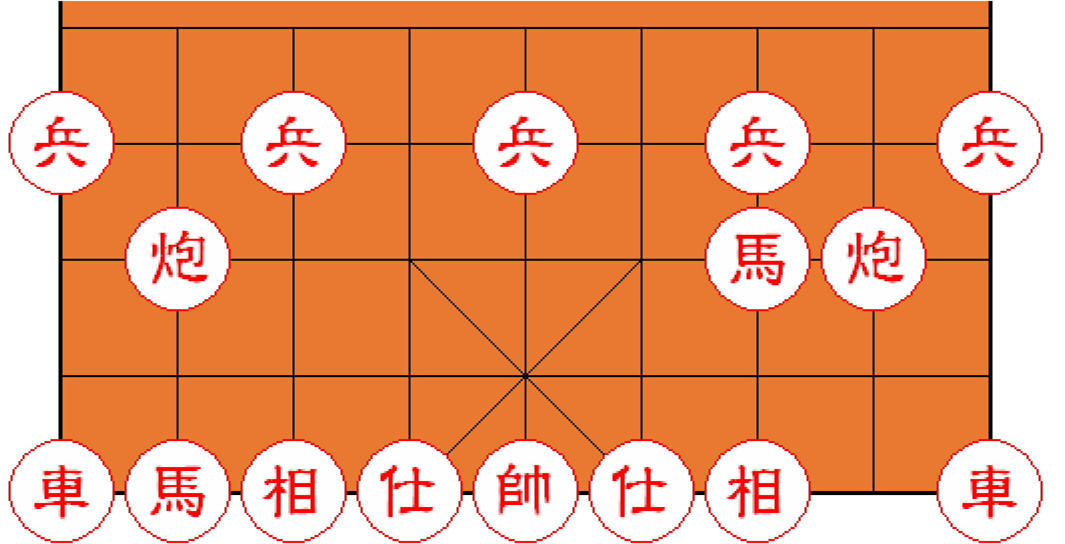

Figure 6. Opening move – usually played by black against the opening move in Fig 5.

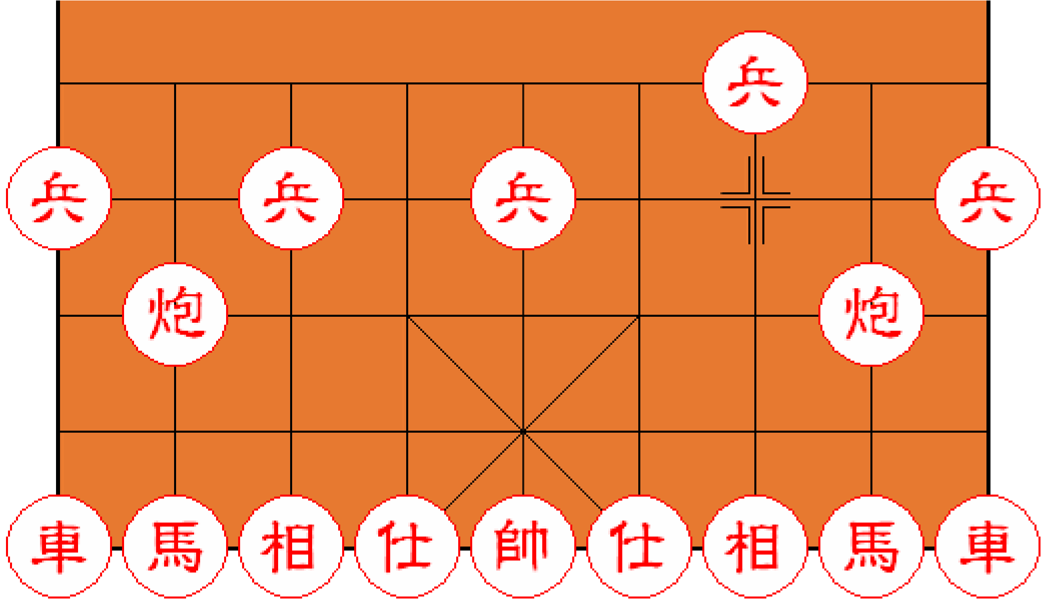

Figure 7. Third opening move

vi. Evaluation

Each board position is evaluated using three components – piece value, position value table and 3 non-overlapping neural networks.

a. Piece Value

Each piece is assigned a value according to importance. According to [1], the values of pieces are:

Piece	King	Advisor	Elephant	Knight	Rook	Cannon	Pawn
Value	6000	120	120	270	600	285	30

Table 1. Value of the pieces

The most important piece after the king is the rook. Its value is roughly equivalent to two knights, two cannons, or a cannon and a knight. A cannon is slightly more valuable than a knight as during early game, the many pieces limit the mobility of knight but serve as platforms for cannon to increase its threat. Pawn is relatively useless since unlike Chess, it cannot be promoted. However, near the end game, it shines when few attacking pieces are left. The evaluation function simply adds up the piece value of all the pieces still on board to get the total piece value (opponent's pieces are assigned negative values).

b. Position Value Table

For the four attacking pieces – knight, rook, cannon and pawn, certain positions on the board are more advantageous to occupy. The relative advantages of the positions are reflected in the position value tables [1]. All the tables below are from red's point of view. To get black's point of view, flip the table upside down. The evaluation function sums up the position value of all attacking pieces (again, opponent's pieces are assigned negative values) to get the total position value.

14	14	12	18	16	18	12	14	14
16	20	18	24	26	24	18	20	16
12	12	12	18	18	18	12	12	12
12	18	16	22	22	22	16	18	12
12	14	12	18	18	18	12	14	12
12	16	14	20	20	20	14	16	12
6	10	8	14	14	14	8	10	6
4	8	6	14	12	14	6	8	4
8	4	8	16	8	16	8	4	8
-2	10	6	14	12	14	6	10	-2

Table 2. Position value table of rook from red's point of view

4	8	16	12	14	12	16	8	4
4	10	28	16	8	16	28	10	4
12	14	16	20	18	20	16	14	12
8	24	18	24	20	24	18	24	8
6	16	14	18	16	18	14	16	6
4	12	16	14	12	14	16	12	4
2	6	8	6	10	6	8	6	2
4	2	8	8	4	8	8	2	4
0	2	4	4	-2	4	4	2	0
0	-4	0	0	0	0	0	-4	0

Table 3. Position value table of knight from red's point of view. Note that positions that check or that are a move from checking are assigned higher values

6	4	0	-10	-12	-10	0	4	6
2	2	0	-4	-14	-4	0	2	2
2	2	0	-10	-8	-10	0	2	2
0	0	-2	4	10	4	-2	0	0
0	0	0	2	8	2	0	0	0
-2	0	4	2	6	2	4	0	-2
0	0	0	2	4	2	0	0	0
4	0	8	6	10	6	8	0	4
0	2	4	6	6	6	4	2	0
0	0	2	6	6	6	2	0	0

Table 4. Position value table of cannon from red's point of view

0	3	6	9	12	9	6	3	0
18	36	56	80	120	80	56	36	18
14	26	42	60	80	60	42	26	14
10	20	30	34	40	34	30	20	10
6	12	18	18	20	18	18	12	6
2	0	8	0	8	0	8	0	2
0	0	-2	0	4	0	-2	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Table 5. Position value table of pawn from red's point of view. A pawn functions somewhat like a rook with limited mobility when it is near the fort.

c. Neural Network

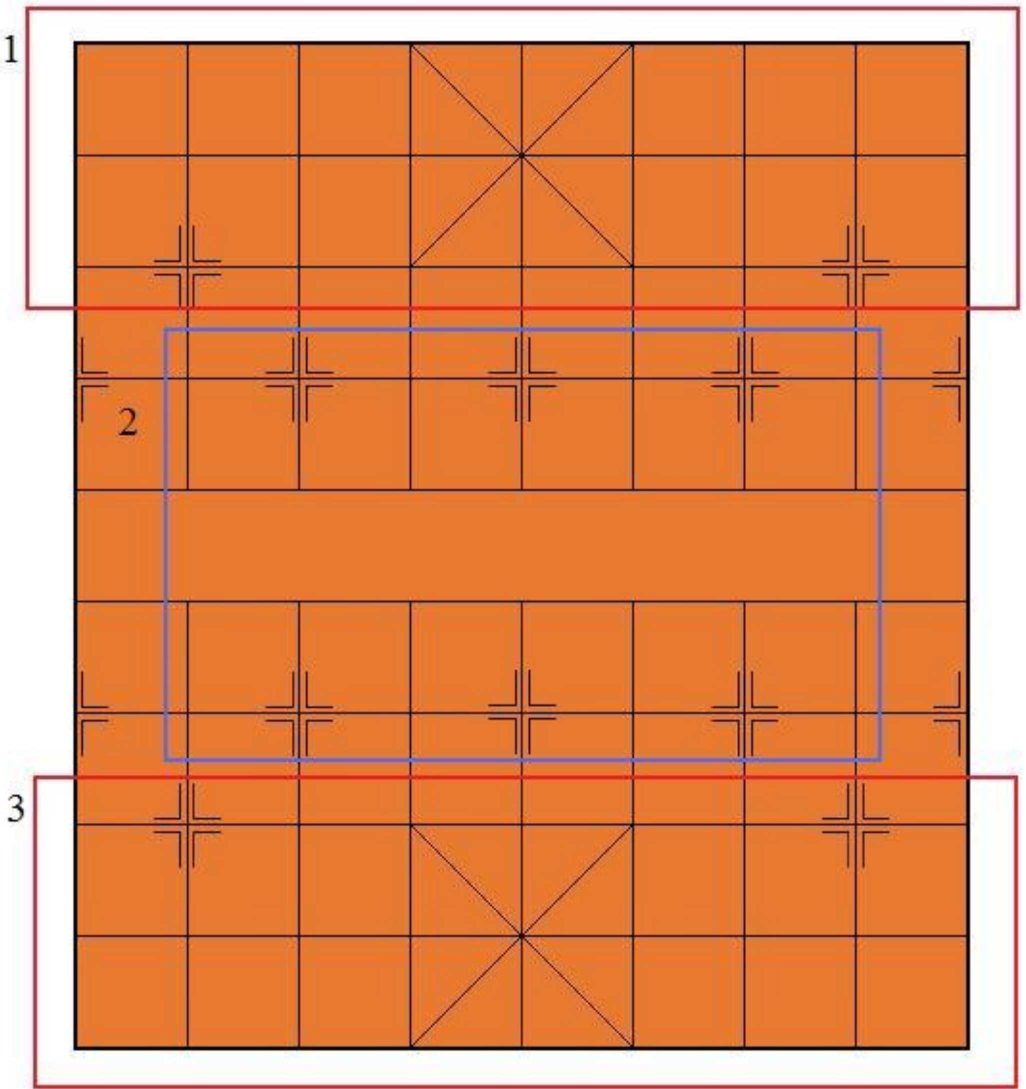

Figure 8. Three non-overlapping neural networks

The last component is the three non-overlapping neural networks. The positions covered by each neural network are determined by activity and importance and are shown in Fig. 8. Region 1 and 3 cover 27 points while region 2 covers 28 points.

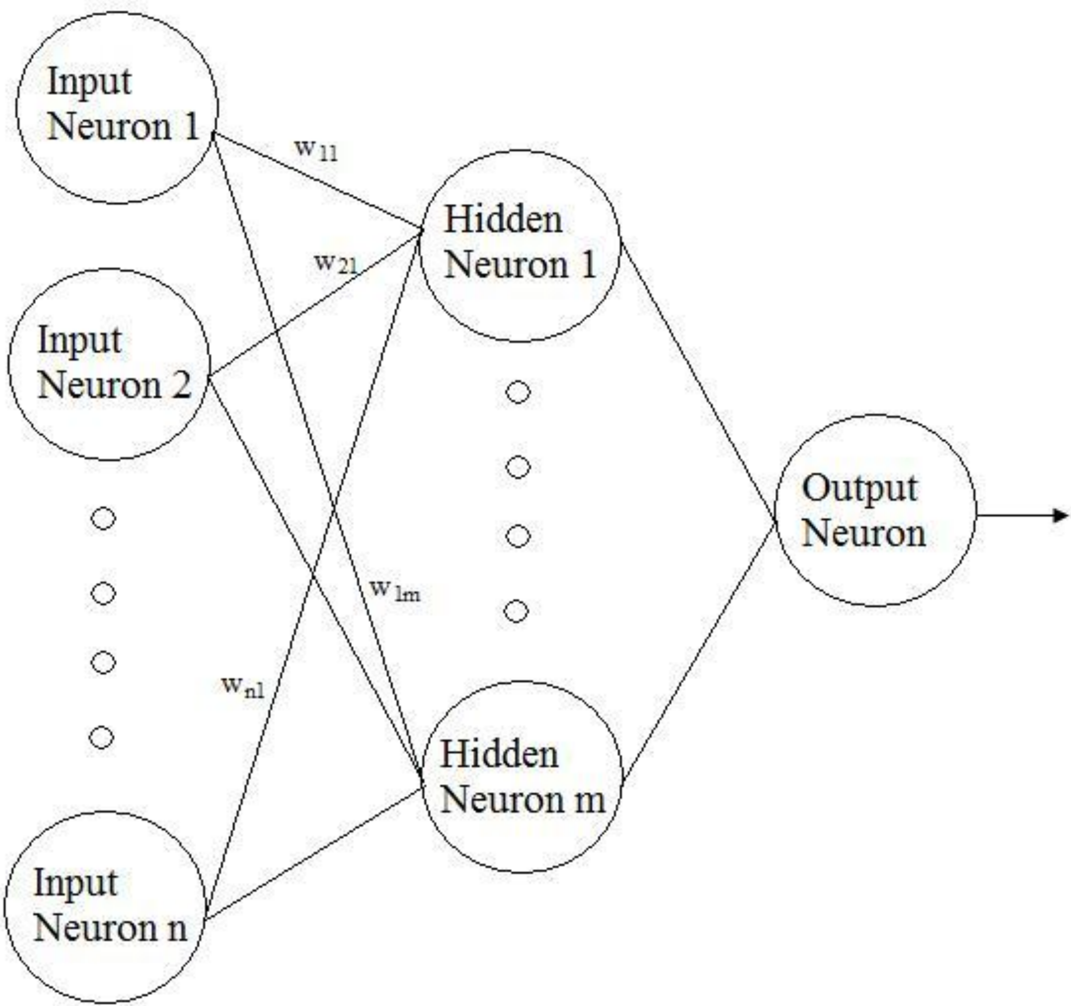

Figure 9. Neural Network Architecture

The neural network architecture adopted is shown in Fig. 9. The network is a fully connected feed forward network with n = 27/28 inputs, m=10 hidden nodes and 1 output node. The neural network works by first mapping each board position covered by the network to an input neuron. Each of the input neuron is in turn connected to the hidden neuron through a variable weight. The hidden neuron computes the activation through a non-linear transformation and sends the result to the output neuron. The result from the hidden neuron is multiplied with a variable weight and the output neuron sums the result from all the hidden neurons, performs a non-linear transformation and outputs a value.

To be precise, the input neuron detects if there is a piece of any side occupying the corresponding board position. If there is, the piece value is taken as the input

neuron value**,** multiplied with the variable weight and the result (or activation) is sent to the hidden neuron. The activation receives at each hidden neuron is thus a dot product of the input neuron values and the weights, which can be expressed as

$$A_i = \sum_{k=1}^{n} p_k w_{ki}$$

where A_i denotes the total activation at hidden neuron i,

p_k denotes the piece value occupying input neuron k, and

w_{ki} denotes the variable weight connecting input neuron k and hidden neuron i

The next step is to compute a non-linear transformation on the activation. Before that, the activation is offset by a bias term, B_i inherent to each hidden neuron. The non-linear transformation chosen here is the standard sigmoid transfer function:

$$f(x) = \frac{1}{1 + \exp(-x)}$$

where $x = A_i - B_i$.

The result multiplied by a variable weight is passed on to the output node, which takes the sum from all the hidden nodes, offset by its bias term and uses the standard sigmoid transfer function to output a value. The output value lies within the range of [-1, 1] and is scaled to be comparable to the values in the position value tables. The higher the scaling factor is, the greater the board position value depends on the relative positions of all pieces.

The position evaluation function sums up the output from the three neural networks to get the total neural network value. Finally, the total piece value, the total position value and the total neural network value are added together to get a value for the board position.

To speed up the computation for the board position value, the idea from pre-generating moves can be applied here by performing pre-evaluation of the board position. Observe that the value before and after a ply differs due to capturing of pieces, change in position values of the piece moved, and change in activation values to all the hidden nodes. Computing these differences is far faster than computing the board position value by taking all the sums. Hence, what I did is to evaluate the board position starting from the initial position and after each move, the board position value is updated through computing the aforementioned differences. Pre-evaluation and

updating are performed for both sides as the players have different position evaluation function parameters.

However, even with the speed-up of pre-evaluation, the evaluation function still takes a considerable amount of time to run. Most of the time taken is used to evaluate the non-linear standard sigmoid transfer function, which involves exponential and division. Also, pre-evaluation updates the activation to each hidden node but not the result from the hidden node. The non-linear function still has to be used 33 times (3 neural networks times 10 hidden nodes plus 3 output nodes) each time the position evaluation function is called. To speed up the computation, I use the following quadratic approximation function which speeds up the evaluation function considerably.

$$g(x) = \begin{cases} 0 & \text{if } x < -4.1 \\ 0.5 + \frac{x}{4.1}\left(1 - \frac{1}{2}\left(1 - \left|\frac{x}{4.1}\right|\right)\right) & \text{if } -4.1 \leq x \leq 4.1 \\ 1 & \text{if } x > 4.1 \end{cases}$$

A plot of the standard sigmoid transfer function and the quadratic approximation is shown in Fig. 10 and the error is shown in Fig. 11. The approximation function is shown to be close to the standard sigmoid transfer function and has an absolute error of less than 0.02.

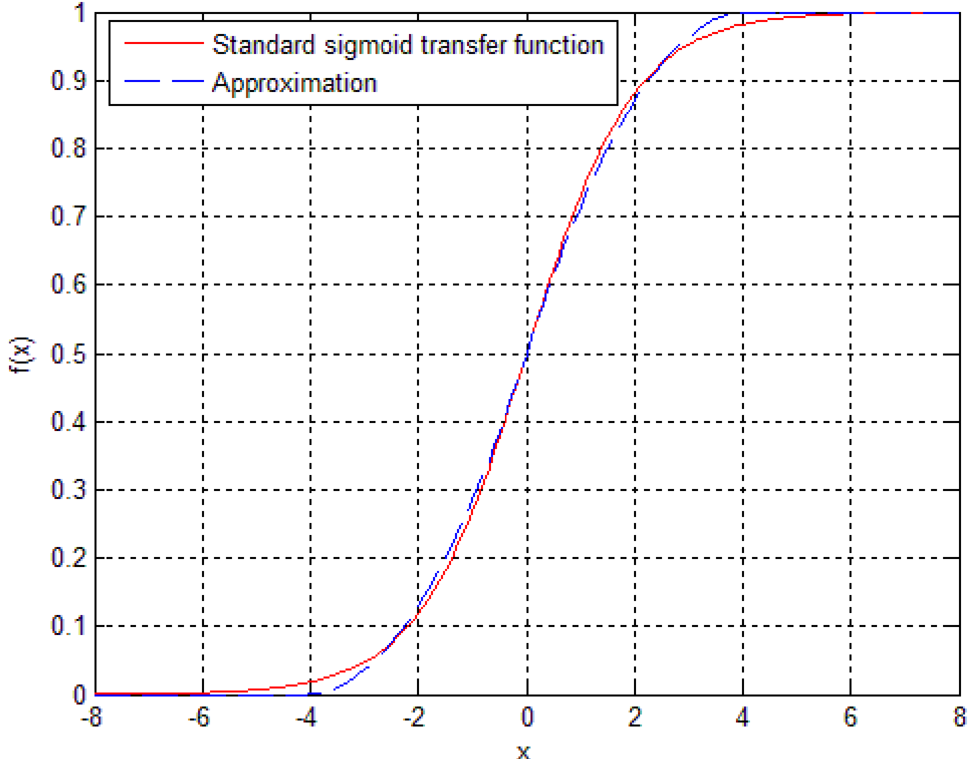

Figure 10. Standard sigmoid transfer and its approximation

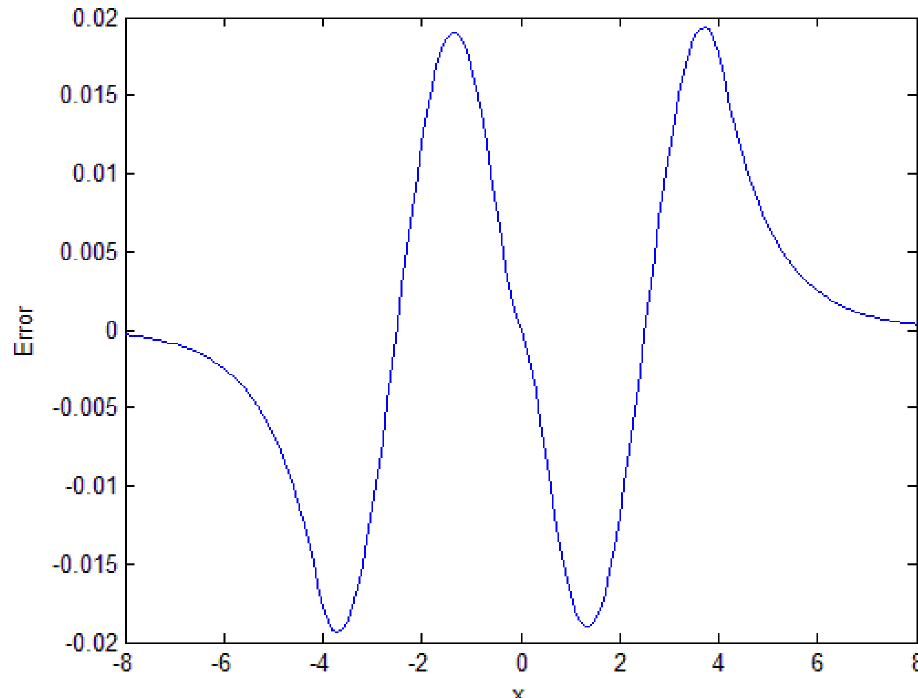

Figure 11. Error of approximation to standard sigmoid transfer function

vii. Search Algorithms

Now that I have an evaluation function to translate each board position into a comparable value, I can search through the game tree to find the optimal move. Chinese Chess has a huge branching factor – at each board position, there is a maximum of 120 possible moves. Due to time constraint, I search only to a depth of 4 plies. Though the search depth is shallow (strong programs sometimes search to a depth of 20 moves!), it still require an examination of more than a million positions and the number increases exponentially as the search depth is increased. To cut down on the number of positions to search, an efficient algorithm to prune away the unnecessary branches is needed. One such algorithm is alpha-beta pruning.

a. Alpha-beta pruning

Alpha-beta pruning is an optimization over the minimax algorithm. It returns the same result as the minimax algorithm but takes as short as one fifth the time it takes with the minimax algorithm. I'll go over the minimax algorithm and then show the modifications of alpha-beta pruning.

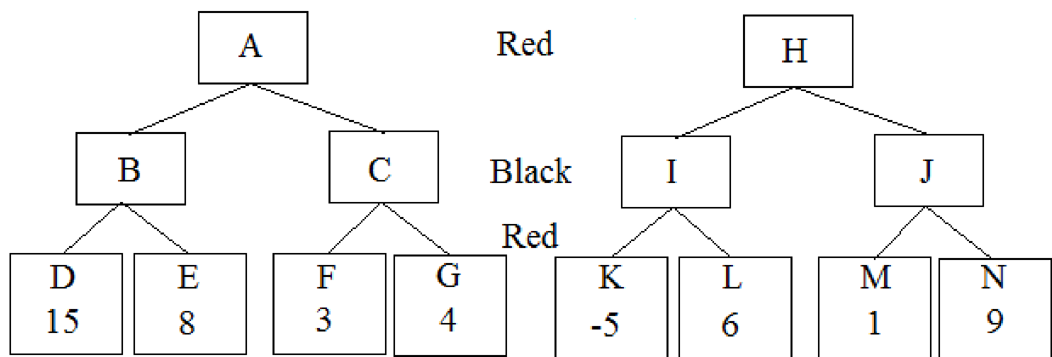

Figure 12. Game tree to illustrate minimax algorithm

Suppose we have the game tree as shown in Fig. 12. Red moves first by choosing between A and H, then black responds, then red plays another move and the board position is evaluated to give a value. At each move, red will attempt to maximize the value while black will attempt to minimize the value red gets, hence the name minimax. If the game is at B, red will choose D to get a maximum payoff of 15 instead of E which gives 8. Similarly, at C, red will choose G. So, at A, black has to choose between B, which will give red a payoff of 15 and C, which will give red a payoff of 4. To minimize the value red gets, black chooses C. The same reasoning is applied to the right half of the game tree and the result is shown in Figure 10. Since H

gives a value of 6 instead of 4 at A, red plays H and the game that results is HIL. HIL is known as the principal variation.

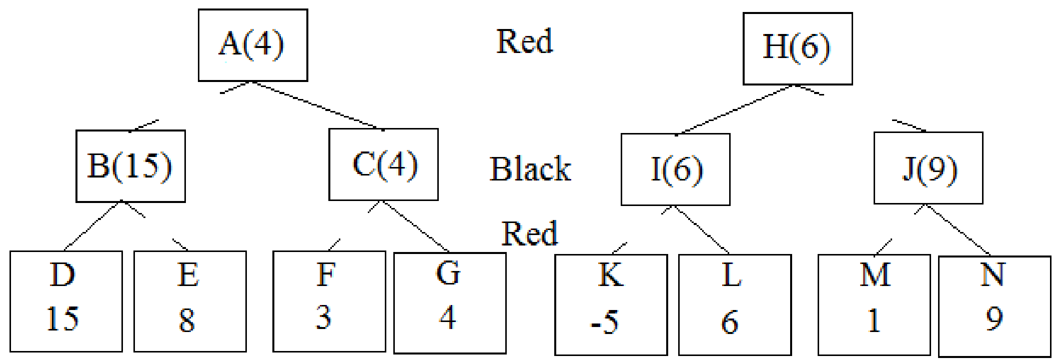

Figure 13. Applying minimax algorithm to game tree. Value of each node in brackets.

The problem here is that minimax algorithm goes through all nodes to find the optimal move. Alpha-beta improves on the minimax algorithm by pruning and thus not searching through all nodes but still returns the correct result under any circumstances. To show how alpha-beta pruning works, consider the same game tree as shown in Figure 12.

Suppose red plays A and now black has to choose between B and C. B and C are known as peer nodes to each other. Black first search down the branches of C, evaluates F and G and determines C to have a value of 4. Then black search down the branches of B, and see E with a value of 8. At this point, without looking at D, black can conclude that B will have a value of at least 8. No matter what the value of D is, if black plays B, red has the option of forcing a value of at least 8 by playing E. B thus has a value of at least 8, which is greater than C's. Therefore, black can stop searching the remaining branches of B and concludes that C is the optimal move. C's value of 4 is known as a cutoff value and if we search down the branches of a peer node and find a value greater than the cutoff, we can prune off all the remaining branches of the peer node.

To put this idea in practice, we need two cutoff values, one for red known as alpha, and one for black, known as beta, hence the name of the algorithm. The pseudocode is as follows.

```
int alphaBeta(int ply, int alpha, int beta) {
     if(ply == 0)
          return evaluatePosition();
     else {
          Array moves = generateAllPossibleMoves();
          for(int i = 0; i < moves.length(); i++) {
               makeMove(moves[i]);
               value = -alphaBeta(ply - 1, -beta, -alpha);
               unmakeMove(moves[i]);

               if(value >= beta)
                    return beta;

               if(value > alpha)
                    alpha = value;

          }
          return alpha;
     }
}
```

The recursive function alphaBeta has 3 arguments, namely ply, which determines the search depth, alpha, a very low number and beta, a very high number. We first check if we have reached the desired search depth, and if we do, we return the board value. If the search depth has not been reached yet, we generate all the possible moves. We make a move and then calls the alphaBeta function recursively. The alpha and beta value passed are swapped and negated as we now move from one player's point of view to another. After the function returns, we unmake the move. Then we compare the returned value to alpha and beta. If the value is greater than the beta cutoff, we know that, as shown in the example, this move guarantees a value of at least the beta cutoff and so we can stop searching the remaining possible moves and returns the function. If the value does not exceed the beta cutoff, we check it with the alpha cutoff, which is the value of the best move. If the current move is better, we replace alpha with value.

b. Move Ordering

The amount of pruning done by alpha-beta depends on how we go through all the possible moves. Consider the following scenario: one move among all the possible moves has a value greater than beta. If that move is examined last, alpha-beta is no

faster than minimax. But if that move is examined first, all the remaining moves do not have to be checked and a huge amount of time has been saved. The example shows the importance of move ordering. If we can somehow order the move according to its "quality" before searching through it, we are likely to produce a cutoff earlier in the search.

One way to determine the quality of the moves is through MVV/LVA (most valuable victim/least valuable attacker). The idea here is that a capturing move is likely to produce a significant change in position values and hence should be searched first. However, we need to check if the capture is worthwhile. For instance, capturing a protected pawn with a rook is a bad move in terms of piece values while capturing a protected knight using a pawn is a good move. The check is achieved by the MVV/LVA scheme which assigns a value of zero to non-capturing moves and for capturing moves, the value is the simple value of the piece captured minus the simple value of the piece capturing if the piece captured is protected. The simple value of a piece is a value which reflects the relative piece value.

Piece	King	Advisor	Elephant	Knight	Rook	Cannon	Pawn
Simple value	5	2	2	3	4	3	1

Table 6. Simple values for pieces

The moves can then be sorted using the MVV/LVA value. The sorting scheme used here is selection sort. Selection sort works by finding the move with the maximum MVV/LVA among the array of possible moves, then swap the best move with the first move in the array of possible moves. Instead of finding the second best move, the best move is used to run the alpha-beta algorithm. The second best move is determined only if the value returned by alpha-beta does not produce a cutoff. The modification to alpha-beta is in the loop, as shown in the pseudocode here:

```
Array moves = generateAllPossibleMoves();
for(int i = 0; i < moves.length(); i++) {
     Find best move among moves[i] to moves[end];
     Swap moves[i] with moves[bestMove];
     makeMove(moves[i]);
     value = -alphaBeta(ply - 1, -beta, -alpha);
     unmakeMove(moves[i]);
...
```

I realize after completing this project that there is a faster and potentially better scheme than MVV/LVA. MVV/LVA is slow as it requires determining whether the

captured piece is protected or not. Protection is determined by looping through all the pieces to see if any of them can reach the position of the piece being captured. Though MVV/LVA is slow in this respect, more computation time is saved by efficient alpha-beta pruning. However, using the pre-evaluation idea earlier, we could have computed the difference in evaluation value due to the moves and use the difference to represent the "quality" of the move. If we disregard the difference due to neural network values and just take into account the piece and position value, the computation is very fast (only a few linear operations) compared to MVV/LVA. The downside of this scheme is that any capturing moves will be of very high quality even though capturing a protected piece using a higher value piece is actually a bad move. However, if we combine both this scheme and MVV/LVA, the move ordering might produce faster cutoffs.

c. Horizon Effect and Quiescence Search

The horizon effect is introduced by search depth limits. Consider the board position as shown in Fig. 14.

Suppose it is red's turn to move. If the search depth is one ply, red will take the cannon, as it gives a huge piece advantage. If the search depth is two plies, red sees that black's rook can capture the red rook that captures the cannon. However, if the search depth is three plies, red sees that after the exchange, it can capture the black rook with the red knight and hence will capture the black cannon. This example clearly illustrates how search horizon affects the moves taken.

Quiescent search solves the horizon effect by selectively extending the depth of the alpha-beta search. Quiescence search works at the leaf nodes of alpha-beta search by extending the search depth if the position is not "quiet", i.e. if it involves a capture or a check. Quiescence search will keep extending the search depth until the position is quiet. My program only extends capturing moves as extending check moves require repetition detection.

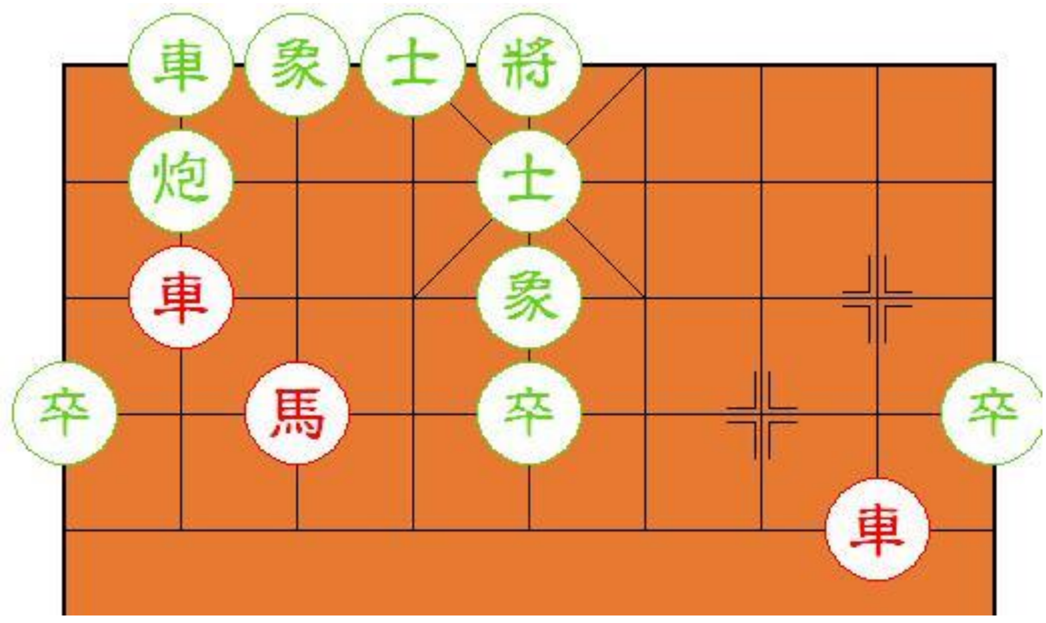

Figure 14. Horizon effect

The pseudocode is as follows:

```
int alphaBeta(int ply, int alpha, int beta) {
...
     for(int i = 0; i < moves.length(); i++) {
          makeMove(moves[i]);
          if(ply == 1 && capture is made)
               value = -quiescenceSearch(-beta, -alpha);
          else
              value = -alphaBeta(ply - 1, -beta, -alpha);
          unmakeMove(moves[i]);
...

int quiescenceSearch(int alpha, int beta){
     Array capture = generateAllCaptures();
     for(int i = 0; i < capture.length(); i++) {
          makeMove(capture[i]);
          value = -quiescenceSearch(-beta, -alpha);
          unmakeMove(capture[i]);
          if(value >= beta)
               return beta;

          if(value > alpha)
               alpha = value;
```

```
    }
    return alpha;
}
```

Alpha-beta search is modified such that at the leaf node, if the move involved is a capturing move, the quiescence search function is invoked instead of alpha-beta. The quiescence search function extends the search further only when the move involves a capture and alpha and beta cutoffs are used to prune away unnecessary searches.

Quiescence search increases the search time but not by much. In complicated situations where a lot of exchanges are involved, quiescence search may extend the search depth by up to 5 or 6 moves and the search time will be lengthen significantly. However, under most situations, quiescence search increases the search time by about 20-30%. In exchange, the result returned by the searching algorithm is now more accurate and the program does not lose pieces easily anymore.

d. Null pruning

Alpha-beta can effectively reduce the search time, but there is a limit to how much time it can save. To further lessen the search time, I implement null pruning, which does not always return the correct result as minimax does.

Null pruning works with the assumption that any move will increase the board position value. So suppose before red makes a move, red evaluates the current position and see that it exceeds the beta cutoff. This means that the move is already too "good" even before red makes a move. There is no need to examine the position further and the function can return.

There are two flaws with null pruning, namely incorrect position evaluation and zugzwang. If in the example when it is red's turn, and the algorithm needs to search two plies deeper, the correct board position value is the minimax value returned from searching down two more plies and not the current board position value. The deeper the remaining search depth, the further off the two values is going to be. Hence, my program only uses null-pruning in quiescence search. The reason is that during quiescence search, the current board position value underestimates the correct board position values as capturing moves (recall that quiescence search only search capturing moves) nearly always improve the board position value.

The second flaw is with the assumption that any move will increase the position value. In zugzwang situations, where a player is forced to make a bad move, the assumption does not hold and null pruning will return an incorrect result. However, zugzwang situations are not that common and usually only happen at the end game.

e. Putting it all together

My search algorithm also extracts the principal variation (the optimal sequence of moves) but that is a matter of passing down an array and storing the correct optimal move (easier said than done actually). There are numerous other search improvement techniques such as killer moves (moves that are good in an immediate previous search are likely to be good now too and so should be searched first), aspiration windows (start with a higher alpha and lower beta cutoff to produce more cutoffs), transposition tables (the same board position can be reached through different order of moves), etc. However, my experience has been that the program was already at a considerable size when implementing the search algorithms and it took more than a month to implement all the search techniques correctly. Implementing any one extra search technique is likely to take longer than a week. Besides, at this point, on average, it takes only about 15 seconds to finish the simulation of a 50-move game. Sometimes it may take as much as 5 seconds to make a ply but most plies take less than 0.1 second to perform. For simulation purposes the time taken is already good enough. Hence the other search techniques will not be implemented.

Putting alpha-beta, quiescent and null pruning together, we have the following pseudocode:

```
int alphaBeta(int ply, int alpha, int beta) {
     if(ply == 0)
          return evaluatePosition();
     else {
          Array moves = generateAllPossibleMoves();
          for(int i = 0; i < moves.length(); i++) {
               // move ordering
               Find    best    move    among    moves[i]    to
moves[end];
               Swap moves[i] with moves[bestMove];
               makeMove(moves[i]);
              value = -alphaBeta(ply - 1, -beta, -alpha);
               unmakeMove(moves[i]);
```

```
                // alpha-beta cutoffs
                if(value >= beta)
                     return beta;

                if(value > alpha)
                     alpha = value;

          }
          return alpha;
     }
}

int quiescenceSearch(int alpha, int beta){
     // null pruning
     value = evaluatePosition();
     if(value >= beta)
          return beta;

     if(value > alpha)
          alpha = value;

     Array capture = generateAllCaptures();
     for(int i = 0; i < capture.length(); i++) {
          // move ordering
          Find   best   capture   among   capture[i]   to
capture[end];
          Swap capture[i] with capture[bestCapture];
          makeMove(capture[i]);
          value = -quiescenceSearch(-beta, -alpha);
          unmakeMove(capture[i]);
          // alpha-beta cutoffs
          if(value >= beta)
               return beta;

          if(value > alpha)
               alpha = value;

     }
     return alpha;
}
```

viii. Competition, Selection, Crossover, Mutation

Competition for single population competitive coevolution is performed by playing each player with every other player in a round robin fashion. The winning player gets 2 points and the losing player gets 0. A draw results in both players getting 1 point. The fitness of a player is the sum of all points from all games played.

For host-parasite competitive coevolution, 6 best parasites from the previous generation play with all 12 hosts and 6 best hosts from the previous generation play with all 12 parasites. The winning player gets 5 points and the losing player gets 0. Hosts get 5 points for drawing while parasites get 0 points for drawing.

After competition, selection is performed through an elitism scheme and fitness proportional selection. The elitism scheme retains the best half of the population to be the first half of the next generation as well as to be the parents for the second half of the next generation. Two parents are selected through fitness proportional selection and an offspring is generated through crossover and mutation of the two parents.

Crossover is achieved by choosing the piece values, position values and neural network biases and weights from either parent with equal probability. After crossover, the values are mutated with a certain probability. Mutation changes the values by adding the original value with the value from a normal distribution.

9. List of all simulation parameters

The players in the two models are initialized using the following list of simulation parameters(which also contain other parameters):

i) Number of players in a generation
   a) Single population competitive coevolution: 20
   b) Host-parasite competitive coevolution: 24 (12 parasites and 12 hosts)

ii) Number of generations: 50

iii) Number of hidden nodes in a neural network: 10

iv) Piece values initialization: Use values in table 1

v) Position value tables initialization: Use values in table 2-5

vi) Neural networks initialization: All weights and biases are initialized to be uniform over [-0.025, 0.025].

vii) Scaling factor for neural network output values: 200

viii) Probability of mutation: 0.5

ix) Distribution for mutation of piece values

Piece	King	Advisor	Elephant	Knight	Rook	Cannon	Pawn
Distribution	Immutable	$N(0,1)$	$N(0,1)$	$N(0,2)$	$N(0,3)$	$N(0,1)$	$N(0,0.25)$

Table 7. Mutation of piece values using normal distribution

x) Distribution for mutation of position value tables: $N(0, 0.25)$

xi) Distribution for mutation of neural network: $N(0, 0.001)$

The output of the neural network is typically in the order of 0.1 to 1. A scaling factor of 200 makes the value comparable to the piece and position values. The variance of the normal distribution for the mutation is chosen to reflect the magnitude of the value being mutated.

10. Analysis and Result

i. Single population competitive coevolution

The best player of generation 50 has the following piece values:

Piece	King	Advisor	Elephant	Knight	Rook	Cannon	Pawn
Value	6000	118(120)	122(120)	281(275)	617(600)	303(300)	30(30)

Table 8. Rounded piece values of the best player of generation 50. The values in brackets are the initial values.

Compared with the initial values, the values of knight and rook has increased by about 3%, the values of elephant and cannon has increased by about 1%, the value of pawn has remained constant while the value of advisor has decreased by about 1%. The value of elephant may have increased as it is frequently ignored when it is about to be captured during middle or early game as there are usually better moves than protecting it. In contrast, the advisors are protected by the king and are harder to be captured. The lost of an elephant is significant during late game and hence the players might have evolved a higher elephant value so that protecting it becomes a higher priority.

The increase in rook values might be due to the shallow search depth. With its high mobility and a shallow search depth of 2 moves, rook is the only piece which can threaten a piece at another corner of the board in 2 moves. Hence, the player sees the rook as a more powerful piece than what its initial value indicates.

The position value tables are shown in Table 9-12. Compared with Table 2-5, The values have varied a bit but the relative advantage of positions for each piece has remained unchanged.

15	14	12	19	16	19	12	14	15
16	19	19	24	27	24	19	19	16
12	10	12	20	19	20	12	10	12
11	18	14	22	23	22	14	18	11
12	14	10	18	20	18	10	14	12
12	16	13	19	21	19	13	16	12
6	8	9	14	15	14	9	8	6
4	10	6	14	11	14	6	10	4
9	5	9	15	8	15	9	5	9
-2	11	6	13	11	13	6	11	-2

Table 9. Rounded position values of rook of the best player of generation 50

6	4	0	-10	-11	-10	0	4	6
1	3	0	-5	-13	-5	0	3	1
1	2	0	-10	-8	-10	0	2	1
0	0	-3	4	11	4	-3	0	0
1	0	1	1	9	1	1	0	1
-3	0	4	2	5	2	4	0	-3
0	0	0	3	3	3	0	0	0
5	-1	8	6	9	6	8	-1	5
1	4	4	7	7	7	4	4	1
0	0	2	7	6	7	2	0	0

Table 10. Rounded position values of cannon of the best player of generation 50

4	8	16	12	14	12	16	8	4
6	10	29	14	9	14	29	10	6
12	13	17	20	17	20	17	13	12
9	23	19	23	20	23	19	23	9
6	16	13	19	18	19	13	16	6
4	11	17	15	11	15	17	11	4
4	6	9	6	10	6	9	6	4
3	1	8	7	5	7	8	1	3
1	4	4	4	-3	4	4	4	1
1	-4	0	0	0	0	0	-4	1

Table 11. Rounded position values of knight of the best player of generation 50

0	4	6	10	11	10	6	4	0
18	34	57	81	119	81	57	34	18
15	26	42	61	80	61	42	26	15
10	19	30	33	41	33	30	19	10
6	13	19	19	20	19	19	13	6
2	1	9	1	8	1	9	1	2
0	0	-3	-1	3	-1	-3	0	0
2	-1	0	1	0	1	0	-1	2
0	1	-1	-1	-2	-1	-1	1	0
1	0	0	-1	1	-1	0	0	1

Table 12. Rounded position values of pawn of the best player of generation 50 (Ignore the bottom three rows).

To determine the evolution of the player's strength, the best player of every 5 generation is taken and played with the first 20 nonevolved players. The best player plays with the nonevolved players for 4 times using the four variations of the opening book to give a total of 80 games.

Generation of player	Wins	Losses	Draws	Win %	Z-score
5	26	16	38	0.61905	1.54303
10	26	19	35	0.57778	1.04350
15	26	20	34	0.56522	0.88465
20	32	23	25	0.58182	1.21356
25	27	17	36	0.61364	1.50756
30	31	16	33	0.65957	2.18797
35	31	16	33	0.65957	2.18797
40	33	18	29	0.64706	2.10042
45	33	18	29	0.64706	2.10042
50	35	19	26	0.64815	2.17732

Table 8. Results of playing the best-evolved player of every 5 generation against the first 20 non-evolved players

The table shows that as the players evolve, the win rate against the non-evolved players increases. Note that the best player of generation 30 and 35 is the same player, and so is the best player of generation 40 and 45. The number of draws also steadily decreases. One possible explanation is that similar players are more likely to draw games. Since players from earlier generations are likely to be more similar to the first generation compared to players from later generations, a greater number of draws is observed in the earlier generations.

To determine whether the best players are stronger than the non-evolved players, statistical hypothesis testing is employed. First, only games with a winner are taken into account. Each such game is modeled as a random variable X with a Bernoulli distribution with parameter p which represents the probability of winning. The sum of n identical and independently distributed Bernoulli distributions give a Binomial distribution with parameters n and p. The parameter n is huge enough here such that the Binomial distribution can be approximated with a normal distribution with mean np and variance np(1-p). Under the null hypothesis, both sides have equal chances of winning, ie $p_0 = 0.5$. The standard or Z-score can thus be computed as

$$Z = \frac{(\sum X) - np_0}{\sqrt{np_0(1 - p_0)}}$$

Note that Z has mean zero and variance 1. As an example, the Z-score for the best player of generation 5 is

$$Z = \frac{26 - (26 + 16)(0.5)}{\sqrt{(26 + 16)(0.5)(0.5)}} = 1.54303$$

For a significance level of $\alpha = 0.05$, which corresponds to $Z^* = \pm 1.96$ (5th or 95th percentile of a standard normal distribution), it can be concluded that the best players from generation 30, 35, 40, 45 and 50 are statistically significantly stronger players than the nonevolved population.

ii. Host-parasite competitive coevolution

Similar to before, the best parasite/host from every 5 generation is played against all 12 non-evolved hosts/parasite. Each best parasite/host plays with each non-evolved host/parasite 4 times using the four variations of the opening book to give a total of 48 games. The results are shown in Table 9 and 10.

For this particular simulation run, it seems that either the parasites are initialized to be extremely strong, or the hosts are initialized to be extremely weak. The parasite and host have a statistically significant difference in strength. If the null hypothesis is parasite and hosts are equally strong, then the Z-score for the best parasite of the first generation is

$$Z_1^p = \frac{16 - 20(0.5)}{\sqrt{20(0.5)(0.5)}} = 2.68328$$

Similarly, the Z-score for the best host of the first generation is

$$Z_1^h = \frac{4 - 25(0.5)}{\sqrt{25(0.5)(0.5)}} = -3.4$$

Both scores reject the null hypothesis with a significance level of $\alpha = 0.05$.

Generation	Wins	Losses	Draws	Win %	Z-score
1	16	4	28	0.33333	0
5	13	9	26	0.27083	-0.9186
10	12	12	24	0.25	-1.2247
15	12	9	27	0.25	-1.2247
20	15	15	18	0.3125	-0.3062
25	13	16	19	0.27083	-0.9186
30	17	15	16	0.35417	0.30619
35	16	16	16	0.33333	0
40	20	8	20	0.41667	1.22474
45	19	6	23	0.39583	0.91856
50	24	0	24	0.5	2.44949

Table 9. Results of playing the best parasite of every 5 generation against the 12 non-evolved hosts.

Generation	Wins	Losses	Draws	Loss %	Z-score
1	4	21	23	0.4375	0
5	18	21	9	0.4375	0
10	7	15	26	0.3125	1.74574
15	5	17	26	0.35417	1.16383
20	14	14	20	0.29167	2.0367
25	11	12	25	0.25	2.61861
30	19	15	14	0.3125	1.74574
35	15	13	20	0.27083	2.32766
40	19	14	15	0.29167	2.0367
45	10	6	32	0.125	4.36436
50	15	6	27	0.125	4.36436

Table 10. Results of playing the best host of every 5 generation against the 12 non-evolved parasites.

To determine whether the hosts or parasites have evolved to be stronger, statistical hypothesis testing is again applied here with some modifications.

For parasites, each game is modeled by X, a Bernoulli random variable with parameter p representing the probability of winning. 1-p now represents the probability of not winning (instead of probability of losing). For hosts, parameter p represents the probability of not losing while 1-p represents the probability of losing.

The null hypothesis is that the parasites/hosts have not evolved to be stronger, i.e. the parameter p has not increased. The parameter p is estimated using maximum likelihood, which for the case of a Binomial distribution, is simply the sample mean of X. From Tables 9 and 10, $\tilde{p}^p = 1/3,\ \tilde{p}^h = 27/48$.

The Z-score can then be computed as before. For instance, the Z-score for the best parasite of generation 15 is

$$Z_{15}^p = \frac{12 - 48\left(\frac{1}{3}\right)}{\sqrt{48\left(\frac{1}{3}\right)\left(\frac{2}{3}\right)}} = -1.2247$$

For parasites, for a significance level of $\alpha = 0.05$, only the best player of generation 50 is significantly stronger than the first generation parasite. It is interesting to note how the number of draws and losses evolve over the generations. Since parasites start out with as few as 4 losses, to increase its win rate, parasites should focus on decreasing the number of draws. Indeed the number of draws gradually decreases. However, the tradeoff is an increase in losses and decrease in wins. At generation 35, the number of draws reaches a minimum of 16. After that, the reverse happens with wins and draws increasing but losses decreasing.

For hosts, for a significance level of $\alpha = 0.05$, all best hosts after generation 20 except for the best host of generation 30 is significantly stronger than the first generation host. Observe that after generation 20, the alternating increase and decrease in the number of losses and wins happens in sync.

I did not get to check and see if hosts evolve the strategy to force repeated moves to draw the game. The problem is that move history alone is insufficient to determine which side forces the repeated move. The only way to determine it is to use the graphical display and judge from the board position, but this takes too much time as there are at least 800 games to look through.

11. Discussion

Though the evolved players improve as expected, a way to quantify how much the players have improved is lacking here. To be able to accomplish that, Chinese Chess software with a known stable rating is required. Also, an interface is required for the two programs to communicate and play with each other.

A lot of the simulation parameters can be played around with. For instance, the number of hidden nodes and the scaling factor of the neural network can be evolved instead of being held constant. The variance of the normal distribution used in the mutation can be adaptive instead of being held constant.

The unresolved problems are repetition detection and implementation of correct repetition rules. More efficient search algorithms should also be implemented, to allow for deeper searches.

One problem with neural networks is the use non-linear functions. Non-linear functions are inherently slower to compute than linear functions and their usage slow down the overall computation time. If a computer program utilizes neural network, for the same computation resources available, it is going to search less depth and than one that only relies on linear functions. Basically, the tradeoff is between complicated strategies and search depth. More study is required to determine the optimal balance between the two.

12. References

[1] S. J. Yen, J. C. Chen, T. N. Yang, and S. C. Hsu, "Computer Chinese Chess," *ICGA Journal*, vol. 27, no. 1, pp. 3-18, Mar 2004.

[2] http://chess.cjcu.edu.tw/en/2007resultA.htm

[3] 13th Computer Olympiad, Chinese Chess – Beijing 2008 (ICGA Tournaments), http://www.grappa.univ-lille3.fr/icga/tournament.php?id=184

[4] http://www.inspur.com/Products/Channer_Promotion/game/index.asp (website is in Chinese)

[5] K. Chellapilla and D. B. Fogel, "Evolving neural networks to play checkers without relying on expert knowledge," *IEEE Transactions onNeural Networks*, vol. 10, no. 6, pp. 1382-1391, Nov 1999.

[6] Bruno Bouzy and Tristan Cazenave, "Computer Go: an AI oriented survey," *Artificial Intelligence*, v.132 n.1, p.39-103, October 2001.

[7] C. S. Ong, H. Y. Quek, K. C. Tan and A. Tay, "Discovering Chinese Chess Strategies through Coevolutionary Approaches," *IEEE Symposium on Computational Intelligence and Games,* April 2007.

[8] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "A self-learning evolutionary chess program," in *Proceedings of the IEEE International Conference*, vol. 92, no. 12, pp. 1947-1954, Dec 2004.

[9] Game programming (1997). http://www.ics.uci.edu/~eppstein/180a/w99.html

13. Appendix

Quick introduction to Chinese Chess

Do note that some of the terminologies mentioned here are actually not part of the standard jargon, but are either terms used in my program or are conventions used in game programming. Throughout the introduction, the bolded words are terms that I will be using throughout the project, and non-standard terms will be indicated with an asterisk.

1. **Players**
   Chinese Chess is a game consisting of a **board**, 16 **red** and 16 **black pieces** and 2 players. The 2 players take turns to move their pieces and the **side** to move first is denoted as **red** while its **opponent** is denoted as **black**.

   I'll stick to the red and black color convention throughout except in the graphical display for black's pieces. Since they are green in color, they are referred to as **green*** specifically when I'm talking about pieces to be displayed.

2. **Board**

   The board is made up of a grid of 10 horizontal and 9 vertical lines. Unlike Chess, the pieces are played on the intersection **points**. There are 10 x 9 = 90 intersection points to play on. The horizontal lines are known as **ranks** while the vertical lines are known as **files**.

   The centers of the first to third rank and the eighth to tenth rank are marked with an X marking the 3 x 3 points constituting the palace or **fort**. In between the fifth and sixth rank is a blank strip known as the **river**. The crosses on the third, fourth, seventh and eighth ranks indicate the initial locations of pawns and cannons.

   The files here are numbered according to the traditional game notation. The rightmost file of each player is numbered 1 and the leftmost, 9. The traditional game notation is used on all Chinese Chess books and on Chinese Chess programs to display the history of moves to the user. However, it is seldom used as the program's internal representation of a move.

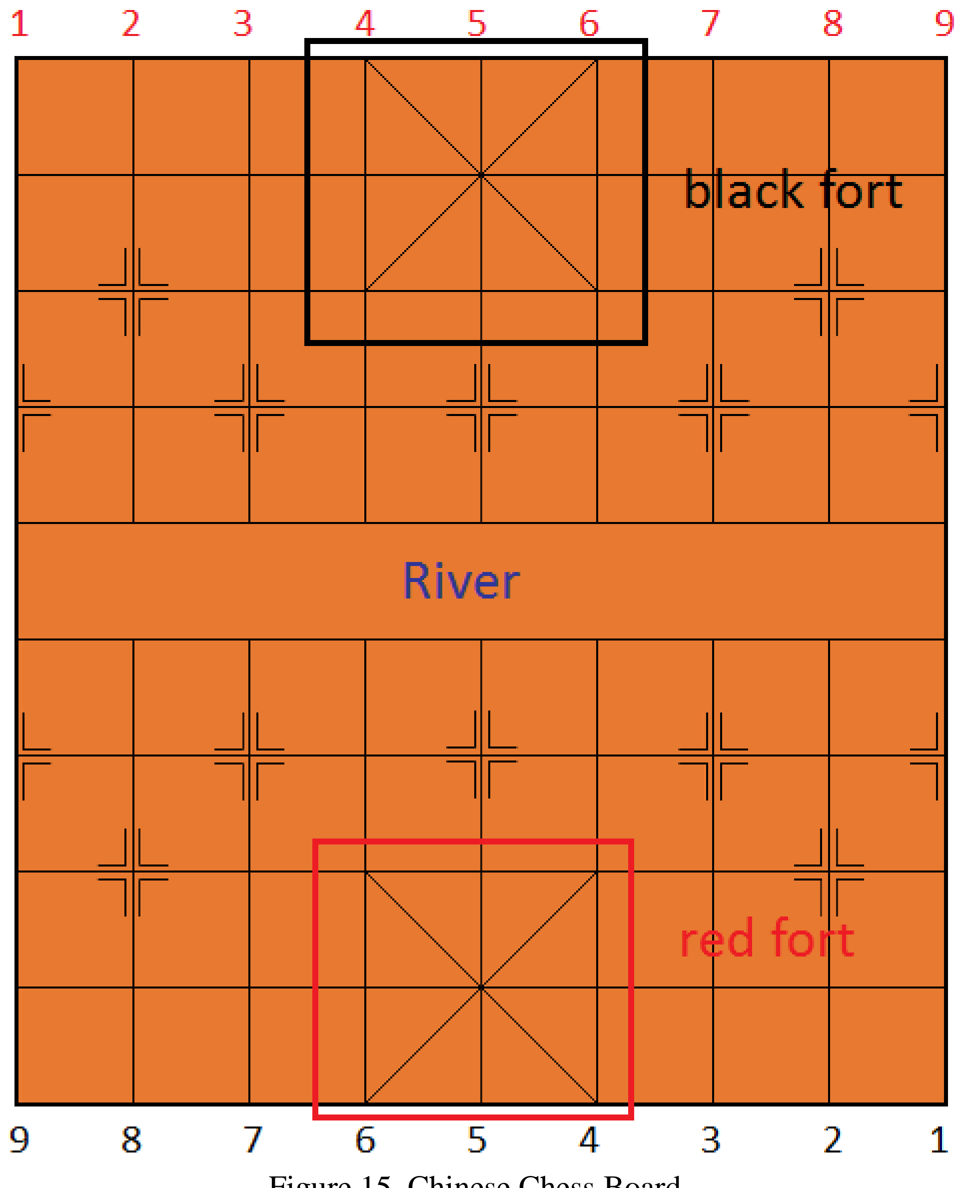

Figure 15. Chinese Chess Board

Terms for board orientation

In my program I will sometimes refer to a region of the board with the term **left***, **right***, **top***, **bottom***, **up*** and **down***. The associated region is as labeled in figure 2. The initial positions of the red pieces will always be at the bottom while the black pieces occupy the top. The bottom half of the board is

the **home side*** for red and the **away side*** is the top half and vice versa for black. The two halves are separated by the river.

The board will also have a **coordinate system [x][y]***. The coordinate system will start at the top left (not necessarily with x = y= 0). x refers to file and x increases in the right direction. y refers to rank and y increases in the down direction. The direction of moving **forward**, however, depends on whose side it is. For red, moving forward means moving up, while for black, it's moving down.

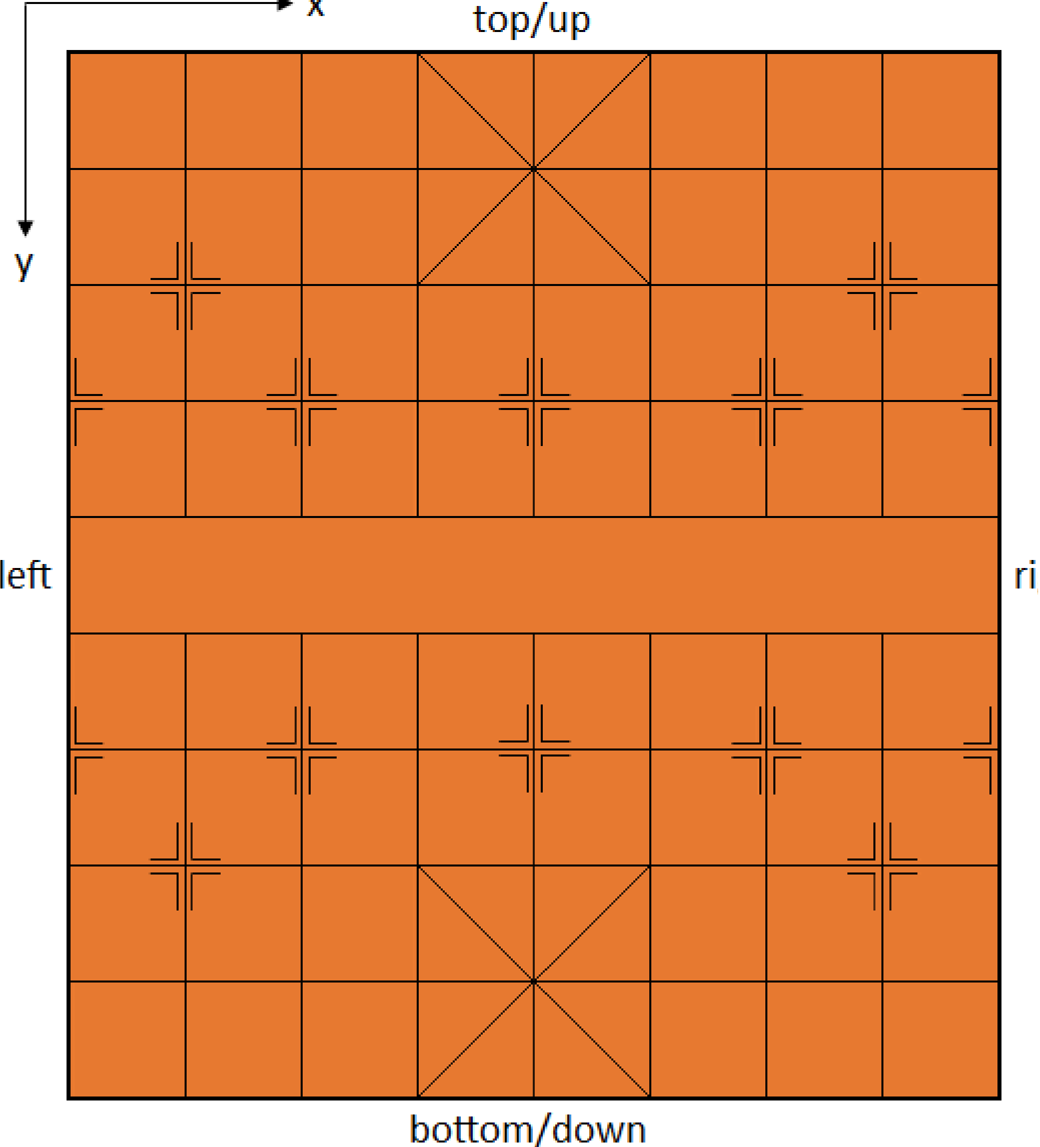

Figure 16. Board Orientation

3. Pieces

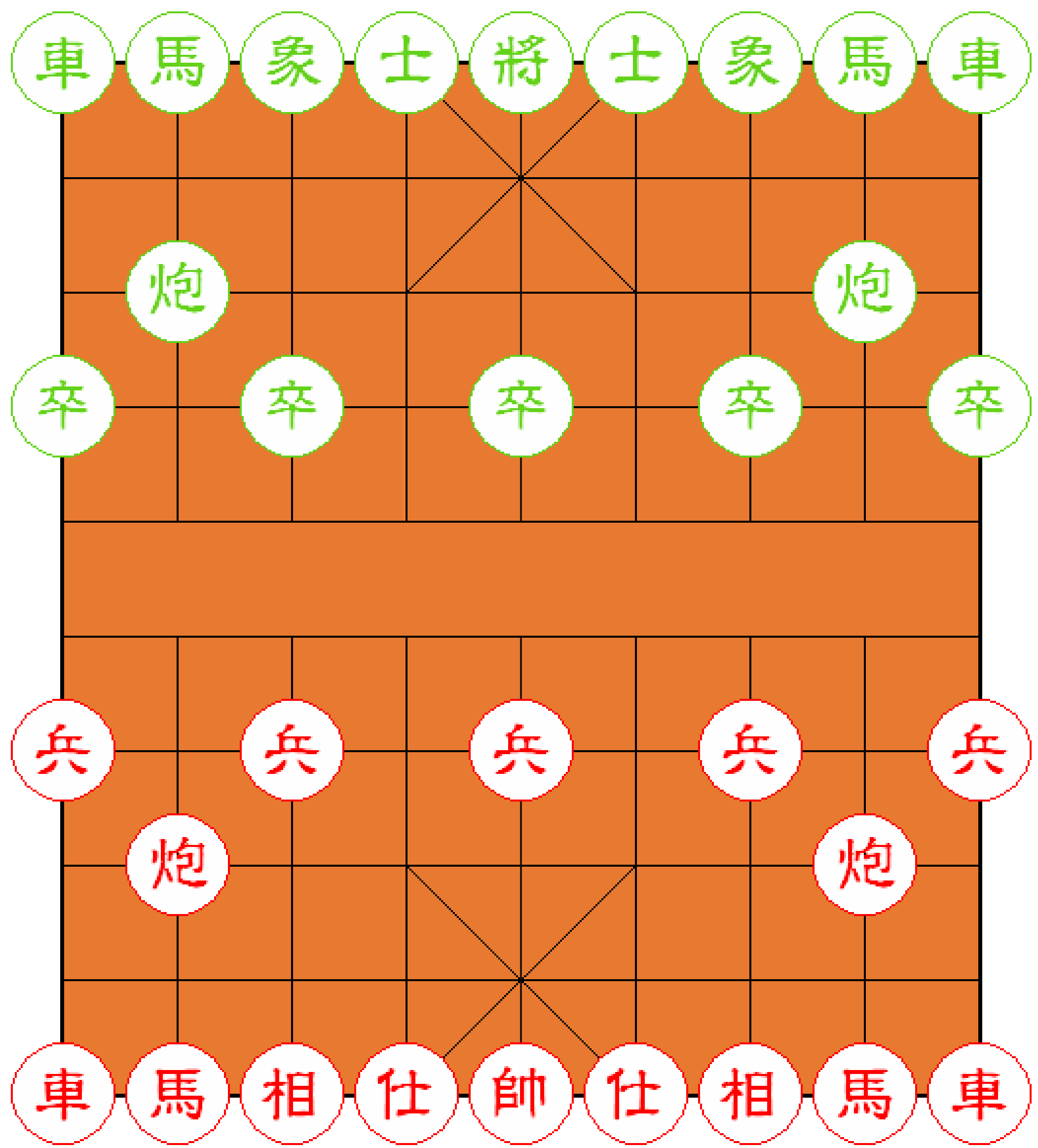

Figure 17. Initial layout of the pieces

Each player has the following 16 pieces (abbreviation for each piece in brackets):

1	General/**King**	**(K)**	帥 將

2	Guards/**Advisors**	**(A)**	仕 士
2	Ministers/**Elephants**	**(E)**	相 象
2	**Knights**/Horses	**(H)**	馬 馬
2	Chariots/**Rooks**	**(R)**	車 車
2	Catapults/**Cannons**	**(C)**	炮 炮
5	Soldiers/**Pawns**	**(P)**	兵 卒

How the pieces move

King. The king can move one point to the left, right, up, or down and its movement is restricted to within the fort.

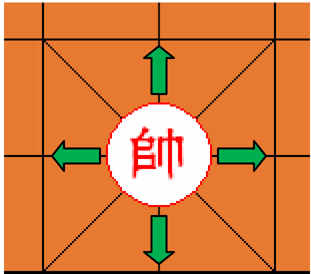

Figure 18. Possible moves for king

One of the unique features of Chinese Chess is that the two kings can never be placed on the same file with no other piece placed in between. This rule is known as the flying king.

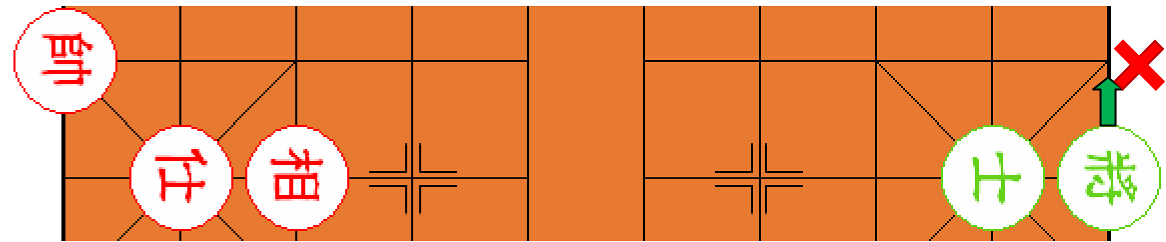

Figure 19. Flying king. Making the two kings face each other

Advisor. The advisor can move one point in any diagonal direction and its movement is restricted to within the fort. The advisor may only occupy the center or the 4 corner points of the fort.

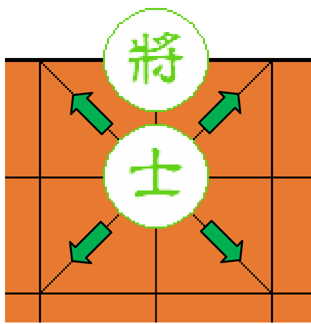

Figure 20. Possible moves for advisor.

Elephant. The elephant may move two points in any diagonal direction. It cannot leap over any piece and the blocking piece is said to be occupying the **elephant's eye** (see figure x). The elephant's movement is restricted to its home side (ie, it cannot cross the river). The elephant can only reach 7 points on the board.

Knight. The knight may move one point in any non-diagonal direction, followed by one-point in a diagonal direction. This L-shaped move is similar to the knight's move in International Chess. However, a knight's move may be blocked if the **knight's leg** is occupied. The elephant's eye and the knight's leg are collectively denoted as **pin***. A knight's leg is any point adjacent(non-diagonal) to the knight and if it is occupied, moves in the direction of the leg is forbidden.

.

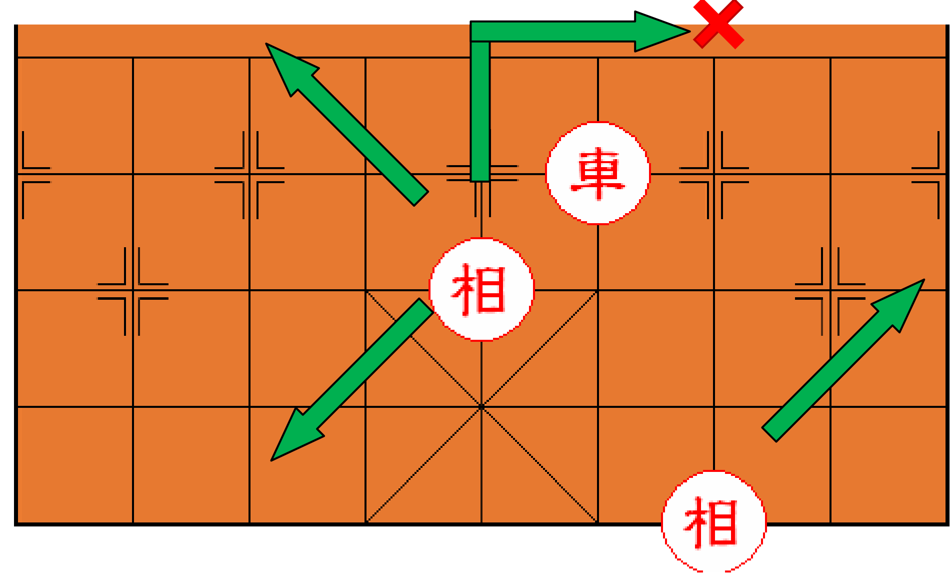

Figure 21. Possible moves for elephant. The rook is occupying one of the elephant eyes

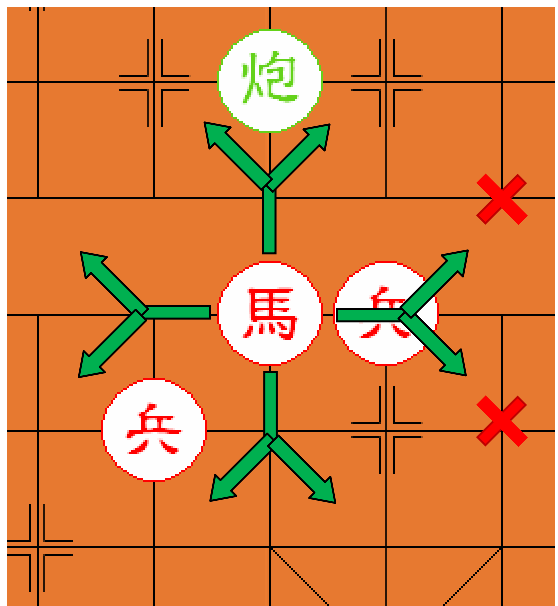

Figure 22. Possible moves for knight. The right red pawn is occupying the knight's right leg

Rook. The rook can move any number of points in any non-diagonal direction just like the rook in International Chess.

Cannon. The cannon, like rook, can move any number of points in any non-diagonal direction. The cannon is also a long-range threat piece but it differs

from the rook on how a capture is made. A cannon can only capture a piece by leaping over exactly one piece of either color. The piece which the cannon leaped over is called the **cannon's platform.** When capturing, the cannon is moved to the point of the captured piece. The cannon may not leap over any piece when not capturing.

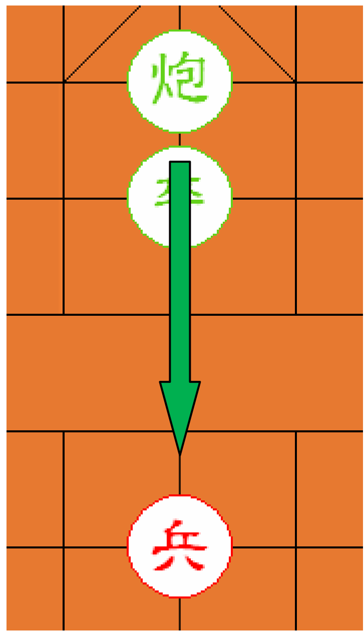

Figure 23. Cannon capturing a piece by leaping over a platform

Pawn. Before crossing the river, the pawn may only move and capture by moving forward by one point. After crossing the river, the pawn may move forward, to the left, or to the right by one point. The pawn may never move backwards and once it reaches the farthest rank, it doesn't promote and its movement is restricted to left and right by one point.

4. Basic rules

(a) The game is won by **checkmating** or **stalemating** the opponent's king.
(b) **Perpetual check** is forbidden.
(c) **Perpetual chasing** (chasing is defined as threat of capture) of any unprotected piece (except rook) is forbidden. Perpetual chasing using the king or the soldier is allowed, however.
(d) There are a number of complex rules governing situations where both perpetual checking and perpetual chasing are involved in complicated ways. We are not going to go into it here.