CSC111 Project Report: AI Player in Chinese Chess

Junru Lin, Zixiu Meng, Krystal Miao, Jenci Wei

Thursday, April 15, 2021

1   Introduction

Chinese Chess (or *Xiangqi* in Chinese), is a popular 2-player turn-based board game that originated from China. Two sides are involved in a game of Chinese Chess: Red and Black, and Red is the side that goes first. At the beginning of the game, Both sides have an equal number of pieces, and the positions of the pieces are symmetric (i.e. if we switch the colour of all pieces on the board, the board would look exactly the same). Comparing to Western Chess, Chinese Chess has more piece types and more complex rules, which will be explained in the Appendix 1 at the end of this document.

In our childhoods, we often played Chinese Chess against other people since it is a very fun game. With our interest in Chinese Chess, we want to "teach" a computer to play it. Our goal for this project is **to design an AI for Chinese Chess** that can interact with human players. We hope that it could be upgraded to choose a better strategy after training, and in the end, we hope that our program can play Chinese Chess better than all of us.

According to the TA that commented on our project proposal, what we have planned was too advanced for this course, and we were suggested to remove the Pygame interaction feature in order to save time. However, with our hard work, we managed to complete everything we planned, and we made an informative and straightforward Pygame interactive game that allows anyone to play Chinese Chess against our AIs regardless of their familiarity with Chinese Chess.

2   Dataset Used

The first piece of the dataset we used, called `moves.csv` [2], is a csv file containing online Chinese Chess matches (where every move of each game is recorded) played by expert players. This dataset is downloaded through Kaggle, see link in the References section. We originally intended to use those game recordings to train our AIs; however, since the training result using this dataset is not ideal and the decision trees storing all moves in the dataset get very large, we eventually decided to not use this dataset to train our AIs. Here is a sample of dataset, we will use `gameID` to identify different games, use turn and `side` to rearrange the move. This dataset will be used for training later, though we finally don't use it. Here is a sample of it:

gameID	turn	side	move
57380690	1	red	C2.5
57380690	2	red	H2+3
57380690	3	red	R1.2

The second piece of dataset we used, called `tree.xml`, was generated by our program, storing the decision trees containing the moves and their evaluations (i.e. how good/bad a certain move is). In our program, the user can choose to play against an AI that uses this file to guide its starting moves. However, due to the number of possibilities there is for each possible move in Chinese Chess and the fact that we need to restrict the size of this file so that loading the game tree does not take an indefinite time, we had to cut down many branches of the tree, and consequently, this file can only help our AI to make the first two moves.

3 Computational Overview

3.1 Computations

3.1.1 Modelling the Chinese Chess Game

Before making our AIs, we created a class named `ChessGame` (in `chess_game.py`) which represents a state of game in Chinese Chess. This class is responsible for keeping track of the current state of the chessboard, who's turn is it next, what are all the legal moves, and how many moves have been made so far in the game. To represent the chessboard, we used a *list of list* where each distinct index represents a distinct location on the board. Our `ChessGame` class is analogous to the `MinichessGame` class in Assignment 2, and we wrote some of our `ChessGame` methods based on the `MinichessGame` methods. Since by convention, the moves of Chinese Chess are not given in coordinates but rather are given in terms of the location of the piece, the direction of movement, and the magnitude of movement (i.e. the *wxf notation*, see Appendix 2 for more detail) we wrote additional functions to convert between the wxf notation of movement to our list of list index. To represent the pieces on the board, we created a class named `_Piece` (in `chess_game.py` which represents the piece kind (i.e. horse, pawn, etc.) and to which side the piece belongs. We also wrote a function inside `chess_game.py`, called `calculate_absolute_points`, that evaluates the *absolute point* (i.e. which side is in the lead, and by how much) of the given chessboard; the basis of which we calculate the absolute point can be found in its docstring.

3.1.2 Modelling the Decision Tree

As the project theme suggests, the most important aspect of our project is the usage of the data structure *tree*. To represent this data structure, we created a class named `GameTree` (in `game_tree.py`), which represents a decision tree for Chinese Chess moves. Each instance of `GameTree` stores the move it reprsents (in wxf notation), who's move is it next, how good this move is (measured in *relative points*, which is defined in the docstring for `GameTree.reevaluate` inside `game_tree.py`), the probability of which Red will win, and the probability of which Black will win (both probabilities are defined in the docstring for `GameTree.update_win_probabilities` inside `game_tree.py`). As a tree data structure, `GameTree` also stores a list of its subtrees, which represent the possible moves after the current move has been made. To be able to store our `GameTree`s, we wrote functions inside `game_tree.py` that converts between `GameTree` and xml file, utilizing the

library `xml.etree.ElementTree` [7] (Note: the one we actually used is `xml.etree.cElementTree`, which is functionally identical to `xml.etree.ElementTree`, but faster).

3.1.3 Creating and Training AI Players

Now we have all the tools we need to create the AI players. We first created an abstract class `Player`, representing a Chinese Chess player who can make moves. All subclasses of `Player` can be found in `player.py`.

Our first subclass of `Player` is `RandomPlayer`, who selects randomly among all legal moves for a certain board. Of course, `RandomPlayer` is the weakest AI we wrote, and any other subclasses of `Player` with any configuration can beat it.

Our most important subclass of `Player` is `ExploringPlayer`, whose strength can be configured via the `depth` instance attributes, which represents how many moves this player will look into before deciding on which move to make. `ExploringPlayer` makes move based on the *alpha-beta pruning algorithm*, which is the more efficient version of the algorithm we used in Assignment 2 (which is formally known as the Minimax algorithm). Our implementation of the alpha-beta algorithm is inspired by the pseudocode written in an essay [4]. According to Prof. Liu, we only need to explain an algorithm once (either on the code or on the project report); our explanation to most of our algorithms are on the docstring of the function using that algorithm. A detailed explanation of the alpha-beta pruning algorithm can be found in the docstring of `ExploringPlayer._alpha_beta`. To further decrease the time it takes for `ExploringPlayer` to make a move, we employed the usage of *multiprocessing*, via the library `multiprocessing` [1]. With multiprocessing, we can utilize the computational power of all the cores in a computer, thus decreasing the time it takes for `ExploringPlayer` to make move. According to Prof. Liu, using multiprocessing can be risky as the creation of too many processes may cause the computer to crash; however, we successfully circumvented such issue by limiting the number of processes to 9, and that the method using multiprocessing, `ExploringPlayer._alpha_beta_multi`, is never called more than once at any point in time. With a depth of 3, our group members can win against `ExploringPlayer` most of the times. Any depth setting above 3 will result in `ExploringPlayer` thinking for minutes before making a move.

The third subclass of `Player` is `LearningPlayer`. It is an player using an existing tree but will also explore new moves if the possible moves in its subtrees do not reach its expectation. When we initialize a `LearningPlayer`, the larger the tree we use, the more experienced it will be. When it evaluates its moves, it will first check all his moves in subtrees and choose one with highest win probability. We will have a learning rate `EPSILON`. If the win probability of this move is greater than `EPSILON`, then `LearningPlayer` will take this move. Otherwise, it will explore a new move, performing the same as `ExploringPlayer`. If we set `EPSILON = 1`, `LearningPlayer` will be the same as `ExploringPlayer` at the beginning and through the game. We will need to adjust `EPSILON` so that it can have a proper learning rate. `LearningPlayer` will be used to train a tree, with the initial tree generated from dataset with csv file.

Another `Player` subclass we wrote for training purposes is `AIBlack`. This class takes two parameters, `depth` (works the same way as `ExploringPlayer`), and an xml filename. We named it `AIBlack`

to ensure that we only train this player as the Black player, since for our interface, we planned to let the user be the Red player. Through only training as the Black player, we hoped that it would specialize at playing as the Black side, and have more chance of beating the user who is the Red player. At the start of the round, `AIBlack` reads the xml file and converts it into a `GameTree`, then finds the best move based on the stored evaluation of all possible moves. If no tree is available at some point in the game (either used up or never given a tree in the first place), `AIBlack` will behave the same as an `ExploringPlayer` of the same depth. `AIBlack` keeps track of all the moves it searches (how it works is explained in detail inside the docstring for `AIBlack`); and after each game, we can merge the tree stored in the xml file with the tree containing all moves `AIBlack` searched for in the game using the `AIBlack.store_tree` method. `AIBlack` can be trained using the `train_black_ai` function inside `training.py`, which uses a for-loop to match `AIBlack` (with the given tree) against `ExploringPlayer`, then expand the tree by calling `AIBlack.store_tree` after each round.

3.1.4 The Game

After making different AI players who use different strategies, we are now ready to create an interface where the users of this program can play against our AIs. We decided to use *Pygame* [3] to create our Chinese Chess game interface. We wrote a `Game` class (inside `visualization.py`) responsible for simulating the Chinese Chess game via Pygame. The `Game` class gives the user the options to choose which AI player they want to play against, and whether to turn on background music and/or sound effects. When a `Game` class is initialized, it loads all the necessary images (including the board and the chess pieces), audio (background music and sound effects), and the fonts we would use to display messages. The image of the chessboard, the pieces, and the sound effects are all from a website called Xiangqi Cloud Database Query Interface [6]; the background music played by our group member Jenci on the piano. After initializing the `Game` class, the user may call `Game.run` to run the game. Our game is designed to be beginner-friendly - even if the user is unfamiliar with Chinese Chess, they can click on a piece to see all the legal places that piece can go, and click the desired destination to make a move. More will be explained in Section 3.3: Interactive Part.

3.2 Usage of Trees

In this project, the data structure of *tree* is extensively used. A class that uses the tree structure is our `GameTree` class, which stores a move (and the information of how good this move is); also, it stores a list of its subtrees, which represent the valid moves after the move stored in the parent. More detail regarding the `GameTree` is explained in Section 3.1.2: Modelling the Decision Tree.

The `GameTree` class is used by almost all of the `Player` subclasses. When `ExploringPlayer` evaluates how good a move is using the alpha-beta pruning algorithm, it stores the results as a `GameTree`, and then decides which move to make. The classes `LearningPlayer` and `AIBlack` are given a `GameTree` when they are initialized, and they look for the best move in their `GameTree` when making a move.

3.3 Interactive Part

Our interactive part of the project is achieved through *Pygame* [3], which we used to enable the user to play Chinese Chess against our AIs. When designing our interactive part, we assumed that our users do not know how to play Chinese Chess; as a result, anyone can play Chinese Chess using our interface. The user of our program will always be the *Red* player (i.e. the player that moves first), and the AI will always be the *Black* player. To the right of the chessboard, we wrote some instructions to the user regarding how to make move with our program, and a status bar displaying who's turn is it next. During the user's turn, the user can click on one of their pieces to see all valid moves can be made with that certain piece (the spots of which that piece can go are highlighted); then the user can either click anywhere to deselect the piece or click the desired destination to make the move. After the AI makes move, the starting and ending positions of the piece the AI moved are framed, so it is clear which move the AI made. When one of the kings is captured (i.e. the game ends), a winning/losing message will be displayed so that it is clear who the winner is. When no one wins after 200 moves (which is really *a lot*), the game will end as a draw, since it is most likely that after 200 moves, none of the sides have any offensive forces anymore, and neither of the kings can be captured even if the game lasts indefinitely.

3.4 Usage of New Libraries

One new library we used in our project was `xml.etree.ElementTree` [7]. We utilized this library to write functions that convert between xml files and `GameTree`s. This library enabled us to store `GameTree`s as xml files, which allowed us to quickly access results of prior computations (i.e. when training `AIBlack` and `LearningPlayer`). Another usage of this library can be found in our multi-processing version of the alpha-beta pruning used in `ExploringPlayer`: since memory is not shared between processes, we had to convert the resulting `GameTree` of each process to an xml file, then combine all the `GameTree`s stored in an xml file to see which move is the best one (reading xml files is *a lot* faster than generating those `GameTree`s by computation).

Another new library we used in our project was `multiprocessing` [1]. Since there are approximately 40 different options for moves in any Chinese chessboard, evaluation of each move can be very slow. With the `multiprocessing` library, specifically `multiprocessing.Process` to create additional processes, we utilized the computational power of all cores, which resulted in a very significant decrease in the time it takes for our AIs that use the alpha-beta pruning algorithm to make a move.

As our interactive part, we extensively used `Pygame` [3] to allow our user to play Chinese Chess against our AI. More detail can be found in Section 3.3: Interactive Part.

4 Instructions

After downloading the Python files in Markus, go to `https://uoft.me/csc111`, download all four zip files there, unzip them all (you should get 3 folders and 1 xml file), then place those 4 files in the same level as the Python files. Then the user just needs to run `main.py` and follow the prompts.

5 Changed Made

5.1 Not Using the Datasets We Downloaded

Originally, we were planning to train our AIs based on the moves stored in the dataset (i.e. search for all moves then tell the AI that the move made in the dataset is very good). However, due to the existence of too many possible moves given any state of the board, training our AI using our dataset would take Gigabytes of storage; consequently, loading such a big tree would take a huge amount of time. Ultimately, we decided not to train our AIs using the downloaded dataset. In the future, we might do so for fun.

5.2 Multiple AI Training Techniques

In our proposal, we mentioned that we would train an AI. When implementing the `Player` subclasses, we decided to train multiple AIs where each one employs the usage of a different strategy (i.e. `LearningPlayer` and `AIBlack`). This change gave us more flexibility for how to train our AIs.

5.3 User as the Red Player

In our original plan, we wanted our use to be able to choose whether to play as the Red player or as the Black player. However, due to time constraints, we sacrificed this feature in order to save time to work on our Pygame interface. In the current version, the user will always be the Red player (i.e. the player that moves first). In the future, we might implement a feature where the user has the option to play as either sides.

6 Discussion

6.1 Selecting a training model

We used three different training methods (see `training.py`), got one trained xml file for each, and evaluated and compared their performance in practice (see the table below).

item	Model 1	Model 2	Model 3
file size	5.76MB	5.79MB	58.8MB
moves provided in game 1	0	0	2
moves provided in game 2	0	0	0
moves provided in game 3	0	0	1
moves provided in game 4	0	0	2
moves provided in game 5	0	0	1

`Model 1` corresponds to *Training Win Probability for Tree* method in training.py
`Model 2` corresponds to *Training Points for Tree* method in training.py
`Model 3` corresponds to *Training AIBlack* method in training.py
`file size` is the size of the xml file generated after using a certain training method.
We play five games to test each model and record the number of moves directly provided by the tree (the running time for this will be very short).

From the table above, we find that Model 3 has a better performance than Model 1 and Model 2, since it can provide 0 to 2 moves at the beginning of the game, while the other two models could not provide any move. Besides, the size of xml file generated from Model 3 is 58.8 MB, which is acceptable. Therefore, we ultimately decided to choose Model 3 over the rest and use this data as the option for the user to play against one of our trained AIs (the other options being different depths of `ExploringPlayer`).

6.2   Back to our Goal

Our goal of this project is "to design an AI for Chinese Chess", and we wanted our AI to be as strong as possible. In our project, we successfully designed an AI for Chinese Chess; however, our AI does not work the way we originally intended it to work. Initially, we wanted our AI to select moves based on prior training, then after the tree that stores the result of previous training is used up, the AI switches back to the alpha-beta pruning algorithm to search for new moves. Apparently, we underestimated how large a tree need to be in order to instruct the AI which moves to select in the first few turns. With a tree that is half a Gigabyte large when stored as an xml file, our AI can only utilize such tree to make the first *two* moves without the need to search with the alpha-beta algorithm. In our program, though we give the user the option to play with an AI with 50 MB of the trained file, how well such AI plays is not much different from an `ExploringPlayer` (with the same depth setting) who always search for its next move with the alpha-beta algorithm.

On the bright side, the performance of our `ExploringPlayer` is beyond our expectations. Despite the fact that `ExploringPlayer` with a depth setting of 3 would lose to an average Chinese Chess player most of the times (a higher depth setting would result in a very long computation time), it is very good at spotting subtle mistakes made by the users of the program and it will always take advantage of them. We would say that we fulfilled our initial goal as our AI has a chance of winning against us.

6.3   Obstacles We Encountered

One major obstacle we encountered while doing this project is the lack of knowledge in the area of machine learning. The entirety of our computer science knowledge is based on CSC110 and CSC111, through which we did not learn any advanced tools in machine learning. We believe that there are more efficient and effective ways to train the AIs (maybe through neural networks, or something involving much linear algebra), however, we are not at that level yet. After two or three years, we will definitely be more equipped in terms of proficiency in machine learning, and we can revisit this project to make the pieces of training more meaningful (comparing to our current method, which is keeping merging new subtrees representing unexplored moves with an existing tree).

Another obstacle we encountered is the long-running time for 4-depth `ExploringPlayer`. We find that 4-depth `ExploringPlayer` can avoid some unsatisfying moves compared with 3-depth `ExploringPlayer`. However, the running time for 4-depth `ExploringPlayer` is much longer than that for 3-depth `ExploringPlayer`, and thus we need to make a trade-off between good moves and a short running time. The alpha-beta algorithm could not provide a good strategy with short running time, and a long running time is not practical when we design the interaction.

Besides, though we design several training methods, the best one we got requires human participation while training. If we want to get a better training result, we need to play games with our AI,

which is inconvenient and inefficient compared with automatic machine training.

6.4   Next Steps

In a few years from now, we would have accumulated much more computer science-related knowledge comparing to now. With our knowledge in machine learning/deep learning/neural network, we can come back to this project and redesign our training algorithm for a more efficient and more effective training procedure for our existing AIs, `LearningPlayer` and `AIBlack`. Also, with those advanced tools on hand, we can utilize the dataset we downloaded and train our AI to play like the experts whose matches are recorded on the dataset. Specifically, we learn that there is an algorithm called `Q-Learning`, which requires some advanced knowledge in mathematics and computer science. We will try this algorithm in the future to compare the result with the `Alpha-Beta` algorithm. Whenever we have free time, we can implement the option where the user can choose to play Chinese Chess as any side (currently the user can only play as the Red player).

6.5   Conclusion

Through this project, we have successfully designed and created an AI for Chinese Chess. All of our group members learned much throughout the process of completing this project. Beyond utilizing our Python knowledge as acquired in CSC110/111, we also stepped outside of our comfort zone in learning and attempting more complicated topics: we learned how the alpha-beta pruning algorithm work (i.e. correctness and efficiency comparing to the intuitive Minimax algorithm) and how to implement it, how to decrease the computing time of a Python code using multiprocessing, how to store the tree structure as a file, and much more. Doing this project was an enjoyable experience for all of us, and we look forward to pursuing more advanced knowledge in computer science and artificial intelligence.

7   Appendix

7.1   Appendix 1: Rules of Chinese Chess

Board Setup and Board Info:
General description:

- While Western chess have pieces located on the squares, Chinese chess have pieces located on the intersection of lines

- There are two identical parts, each part has 5 horizontal lines and 9 vertical lines (for a total of 10 horizontal lines).

- There is a 'river' between two parts, which lies horizontally in the middle of the board.

  – Elephants cannot cross the river
  – When a pawn crosses a river, it will be able to move horizontally
  – The river has no special effects on any other pieces

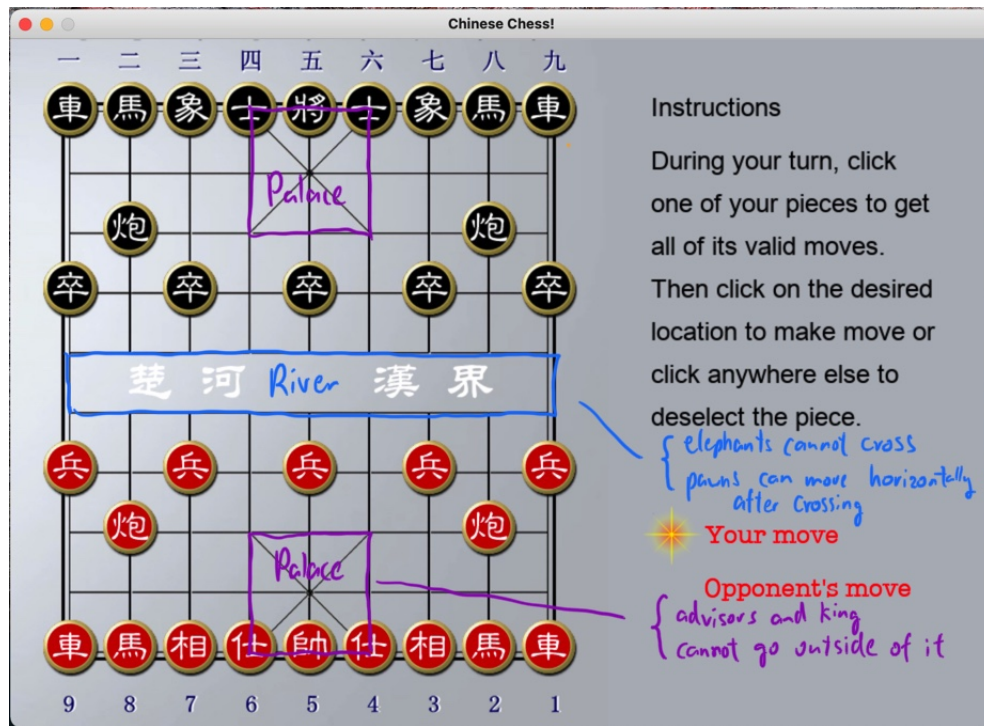- Palace: There are two 3 by 3 palaces in the board, one on each side (as shown in the picture above).

Figure 1: Starting setup from our Pygame interface

    – Advisors and kings cannot go outside of the palace
    – The palace has no special effects on any other pieces
  • There are 7 different pieces in the board. The side that loses the 'General' loses the game.

Rule for different pieces Where 0 represents an empty spot, 1 represents the own piece, 2 represents any other piece, 3 represents any opponent's piece, * represents the place can be either empty or occupied by another piece.

• Pawn: Can only move forward by 1 step per turn before crossing the river and can only move forward, leftward or rightward by 1 step after crossing the river.

• Horse: Can move from $\begin{bmatrix} * & & * \\ 0 & & * \\ 1 & & * \end{bmatrix}$ to $\begin{bmatrix} * & & 1 \\ 0 & & * \\ 0 & & * \end{bmatrix}$ (or similarly in any other directions)

    – Cannot move when being blocked like $\begin{bmatrix} * & & * \\ 2 & & * \\ 1 & & * \end{bmatrix}$

9

- 象 相 Elephant: Can move from $\begin{bmatrix} * & * & * \\ * & 0 & * \\ 1 & * & * \end{bmatrix}$ to $\begin{bmatrix} * & * & 1 \\ * & 0 & * \\ 0 & * & * \end{bmatrix}$ when the destination is not at the other side of the river (or similarly in any other directions).

  – Cannot move when being blocked like $\begin{bmatrix} * & * & * \\ * & 2 & * \\ 1 & * & * \end{bmatrix}$

- 車 車 Chariot: Can move horizontally or vertically by any steps in one move.

  – When blocked in the middle of the move, must either stop before the piece blocking it or "eat" the blocking piece (only if it belongs to the opposite) and take its place.

- 炮 炮 Cannon: Same as Chariot when moving, but cannot "eat" the piece blocking it. Need to jump a single piece when capture the opponent's piece.

  – Can "eat" another piece like this: $\begin{bmatrix} 3 \\ 0 \\ 2 \\ 0 \\ 1 \end{bmatrix}$ to $\begin{bmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 0 \end{bmatrix}$ (or similarly in the horizontal direction)

- 將 帥 General: Can only move vertically or horizontally by 1 step, and cannot exit the palace.

  – Can "eat" the opposite General when there is a clear, straight path between the two generals

  – Game ends when any General is eaten.

- 士 仕 Advisor: Can only move diagonally by 1 step, and cannot exit the palace.

7.2 Appendix 2: WXF Notation

WXF (World Xiang Qi Federation) notation is in the form of "letter/number"+"number"+"symbol"+"number" [5]. For example, "c1+5" is a wxf notation. The WXF notation can only be used to describe one of the sides (i.e. the side that is moving next).

The first letter represents the piece type (the case where the first character is a number will be explained later)

- 'K' stands for king

- 'A' stands for assistance

- 'E' stands for elephant

- 'H' stands for horse/knight

- 'R' stands for rook/chariot

- 'C' stands for cannon

- 'P' stands for pawn

The second character represents the column of the chessboard, which has a range from 0 to 9. For instance, the leftmost line is marked as 0, the rightmost line is marked as 8.

The third character represents the direction of movement, and the fourth character represents the magnitude or destination of movement (depending on the context).

If the third character is "+", then it means moving forward (from own side) vertically. If the piece does not move diagonally, then the fourth character represents by how many spaces the piece moves forward. On the other hand, if the piece does move diagonally, then the fourth character represents to which column the piece moves.

If the symbol is "-", then it means moving backward (from own side) vertically. If the piece does not move diagonally, then the fourth character represents by how many spaces the piece moves backward. On the other hand, if the piece does move diagonally, then the fourth character represents to which column the piece moves.

If the symbol is "." (For example, C1.3), then it means moving from the original column to the new column horizontally. For example, if there is a movement C1.3, then it means move from the first column to the third column horizontally.

In case where two pieces are aligned in the same column, we cannot use a number to represent which column the piece is in (since it creates ambiguity). In this case, the second character will be '+' to represent the piece in the front (from own perspective); and the second character will be '-' to represent the piece in the back (from own perspective). The rules for the other characters hold as usual. For example, "C++1" represents there are two cannon in the column and we move the front cannon forward vertically by 1.

If there are more than three pieces in the same column (the piece must be pawn since no other piece have more than 2 copies), the first character is a number x that represents the x-most-forward piece (from own perspective). The rules for the other characters hold as usual.

References

[1] *multiprocessing — Process-based parallelism.*
     https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.

[2] *Online Chinese Chess (Xiangqi).* https://www.kaggle.com/boyofans/onlinexiangqi.

[3] *Pygame.* `https://www.pygame.org/docs/`.

[4] Shubhendra Pal Singhal and M. Sridevi. "Comparative study of performance of parallel Alpha Beta Pruning for different architectures". In: *CoRR* abs/1908.11660 (2019). arXiv: `1908.11660`. URL: `http://arxiv.org/abs/1908.11660`.

[5] *WXF notation.* `https://xiang-qi.appspot.com/tutorial/listing18.html`.

[6] *Xiangqi Cloud Database Query Interface.* `https://www.chessdb.cn/query_en/`, resources downloaded from the BHGui link at the bottom-left corner.

[7] *xml.etree.ElementTree — The ElementTree XML API.* `https://docs.python.org/3/library/xml.etree.elementtree.html`.