Final Project: AlphaChess

CS3317

October 11, 2023

1 About this Project

- For this project, students need to work in groups of 3 to implement an AI (or an engine, agent .etc perhaps) of Chinese chess.
- Please use Python for this project.
- Every group will be assigned a ID number such as 1,2,3... Please remember your ID number.
- Please submit a zip file of a folder named Player_[ID], where [ID] is ID number of your group. For example, if your ID is 3, then your folder name should be Player_3.
- Please refer to Section 4 for the details of what to include in the folder.

1.1 Auto-Grading

- Basic scores: If your submission is able to defeat the baseline that we prepared, your group will get a basic score of 60%.
- Additional scores: you will get additional scores according to your ranking among all 10 groups. The details are shown in Table 1.

Table 1: Ranking and Score

Ranking	**Score**
1	+40%
2	+40%
3	+40%
4	+30%
5	+30%
6	+30%
7	+20%
8	+20%
9	+10%
10	+10%

Take it easy! The baseline is easy to defeat.

- Ranking method:
  - Each player has to play against all other players once. The final ranking is proportional to the number of matches you win. If the rankings of two or more groups are the same, additional matches will be launched until sorting out the ranks.

- In each match between two players, there will be 6 games. Each player takes the red side for 3 games and the black side for the other 3 games. The player who wins more games takes the match.
  - If the number of wins for both groups are the same, there will be a playoff. At the playoff, sides (red/black) are assigned randomly. Considering the red side has first-hand advantage, if this game comes to a draw, the black side wins.
  - All of your game history will be recorded and sent to you after the final match. You can replay the games using a provided Python simulator.
- If the following situations occur, you will immediately lose a game:
  - Illegal action: If your method returns an illegal action or an action of incorrect format, you will lose the game directly.
  - Time out: We constrain thinking time for each move for both sides. If your agent fails to make a decision within the time limit, you will immediately lose the game. The thinking time is at max 10 seconds.
- Along with the submisssion, you also need to provide a contribution table based on mutal agreement. The sum of all of contributions must be 300% (if there are 3 members), but individual contributions can exceed 100%. Equal contribution (100%, 100%, 100%) is allowed. In the example shown in Table 2, if the total points obtained by the group is 80, then student 1 will get a score of 80* 100%=80, student 2 will get 80×110%=89, and student 3 will get 80×90%=72. Overflow of scores (the part exceeding 100, if it happens) will be regarded as bonus to the total grading.

Table 2: Example for contributions

Name	**Contribution**
Student 1	100%
Student 2	110%
Student 3	90%

1.2 Presentation

You are required to present your method and results as a group in Week 16. To get higher grades:

- Make the presentation structured and clear.
- Present collaboratively.
- Present your exploration path (different approaches you tried, the iteration process of a single method, heuristics you tried, hyper-parameter tuning, ...) and discuss the insights gained from it.
- Discuss the limitations of your method and possible future work to address them.

1.3 Deadline

This project is due on Friday, Week 15 (**23:59, December 22nd**).

2 Chinese Chess

The chessboard is shown in Figure 1.

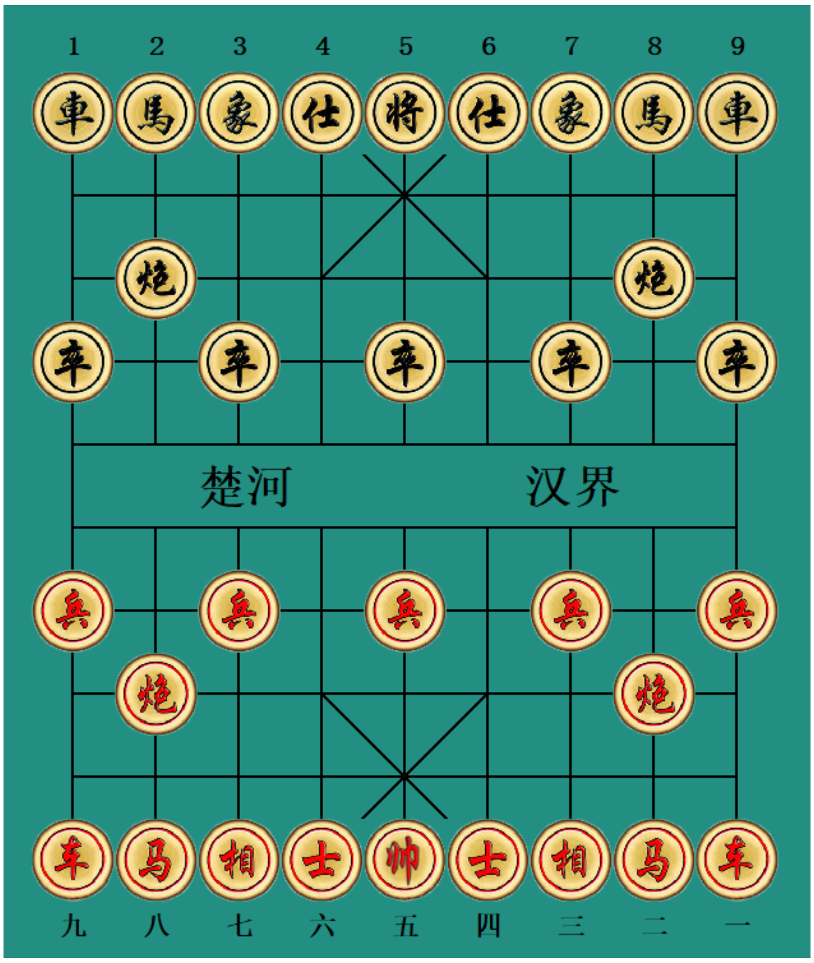
1 2 3 4 5 6 7 8 9
車 馬 象 仕 将 仕 象 馬 車
炮 炮
卒 卒 卒 卒 卒
楚河 汉界
兵 兵 兵 兵 兵
炮 炮
车 马 相 士 帅 士 相 马 车
九 八 七 六 五 四 三 二 一

Figure 1: Chess Board

2.1 Basic Rules

- Chinese chess is a two-player game, where one side is red and the other side is black. The two sides plays in rounds. In each round, the red side takes one move first, followed by the black's move.
- A chessboard is a 10×9 grid. There is a river in the middle, dividing the chessboard into two sides.
- The player of each side has the same 16 pieces at the beginning of a game. They are 5 Pawns (兵、卒), 2 Advisors (士), 2 Elephants (相、象), 2 Horses (马), 2 Cannons (炮), 2 Chariots (车) and only one King (帅、将), respectively.
- At each step, a player can move any of his/her own pieces to capture an opponent's piece by moving to the position of the latter.
- One exception is to move the King (帅、将) and the move exposes the King to a direct attack

(or “in check”, 被将军), then that move is illegal and abandoned.

- When the King (帅、将) of one side is in check and has no legal moves to escape, it is considered “checkmate” (将死), and that side loses the game.

2.2 Chess Pieces

- Pawn (兵、卒): Before crossing the river, a Pawn can only move one unit forward per step; after crossing the river, they can also move one unit horizontally (but not backward).
- Advisor (士): An advisor can only move cell by cell diagonally within the palace (a 3x3 grid at the bottom of the board around the King, or in Chinese “九宫”).
- Elephant (相、象): An elephant can only move two units per step diagonally but can never cross the river. There should not be any piece blocking the Elephant’s path. Ortherwise the move is forbidden, which is known as “堵象眼”.
- Horse (马): In each step, a Horse can either move one unit vertically plus two units horizontally or one unit horizontally plus two units vertically. There should not be any piece blocking the direction of the Horse’s movement. Ortherwise the Horse’s move is forbidden, which is known as “蹩马腿”.
- Cannon (炮): In each step, a Cannon can move vertically or horizontally by any number of units. Or it can use one (but no more than one) piece (of any side) as the springboard to jump over and attack an opponent’s piece, without constaining the moving distance.
- Chariot (车): A Chariot can move vertically or horizontally by any number of units per step.
- King (帅、将): In each step, the King can move one unit vertically or horizontally within the palace (a 3x3 grid at the bottom of the board around the King, or in Chinese “九宫”).

2.3 Draw (平局)

A draw is a situation where there none of the two sides wins the game at the end. There are many situations leading to draws, such as when both sides lose all attacking pieces (Pawn, Horse, Cannon, Chariot). For simplicity and efficiency, we do not detect all draw situations. Instead, we consider it a draw if both sides have not eaten the opponent’s pieces for 60 rounds consecutively.

2.4 Cycles (棋例)

- There are many complicated and hard-to-detect cyclic situations leading to awfully-long games such as “长将”, “长捉”, “一捉一将”. For simplicity and efficiency, we only detect “长将”, the situation that occurs most frequently and can easily lead to a awful game if it occurs. If the same piece consecutively attacks the King for three rounds, it can not attack the King again in the next round.

Our project only considers the above rules. For more details about the full set of rules of Chinese Chess, please refer to online materials.

3 Python APIs

We have provided a package of Python files for you. Please use it as the code base to complete your project.

3.1 Setting up the environment

```
cd alpha_chess
pip install -r requirements.txt
python game_gui.py # validate your environment setup. you should see a chessboard without pieces
```

3.2 Implement your player

- Your player should be implemented in a Python script named player_[ID].py, where [ID] is your group ID.
- You can refer to alpha_chess/player_1.py as an example or as your template. Here is what player_1.py looks like:

```python
import random
from utils import get_legal_actions    # import utilities

class Player: # please do not change the class name
  def __init__(self, side: str):
    self.side = side         # don't change
    self.history = []        # don't change
    self.name = "Player"     # please change to your group name

  def policy(self, board: tuple):    # the core method for you to implement
  '''board is a ×109 matrix, encoding the current game state.
    board[i][j] > 0 means a red piece is on position (i,j)
    board[i][j] < 0 means a black piece is on position (i,j)
    board[i][j] = 0 means position (i,j) is empty.'''

    # get all actions that are legal to choose from
    action_list = get_legal_actions(board, self.side, self.history)
    # here we demonstrate the most basic player
    return random.choice(action_list)

  def move(self, board: tuple, old_x: int, old_y: int, new_x: int, new_y: int):
    # don't change
    '''utility function provided by us: simulate the effect of a movement'''
    ...

  def move_back(self, board: tuple): # don't change
    '''utility function provided by us: restore or reverse the effect of a movement'''
    ...

  def update_history(self, current_game_history: list):
    '''refresh your self.history after each real play; called externally'''
    self.history = current_game_history

  def get_name(self):
    '''used by the external logger'''
    return self.name
```

- Variables:
  - self.side: specifies which side your agent takes. It must be “red” or “black”.
  - self.history: records history actions.
  - self.name: for you to set a name for your player. It is “Player” by default.
- Core methods:
  - policy: the core method for you to implement. It must return a legal action according to the input board configuration. Return values must be a four-element tuple or list in the form of (old_x, old_y, new_x, new_y), with the x coordinate representing the column number and the y coordinate representing the row number.
  - move and move_back: when you do “search” or “rollout”, you can utilize these two methods to simulate the change of the board as the effect of actions and update self.history accordingly. Specifically:
    - move: simulating movement, moving a piece from (old_x, old_y) to (new_x, new_y) and eating a piece when overlap happens.
    - move_back: restoring the last move. You need to use it when backtracing along a path during a search, so that both the board and self.history are reverted correctly.
- You can freely add your own member variables and methods or other attributes. But keep in mind that the “policy” method needs to be called during a game. Its interface, including the name and the format of input arguments and returns, should not be changed.

3.3 Utilities

alpha_chess/utils.py contains utility functions for you to call (**please do not change this file!**). The most important ones are:

- Board encoding (Table 3) and initial board.

Table 3: Encoding

Encoding	**Meaning**	**Encoding**	**Meaning**
1	Red Pawn	-1	Black Pawn
2	Red Advisor	-2	Black Advisor
3	Red Elephant	-3	Black Elephant
4	Red Horse	-4	Black Horse
5	Red Cannon	-5	Black Cannon
6	Red Chariot	-6	Black Chariot
7	Red King	-7	Black King
		0	There is no piece on this position

- So the initial chessboard configuration is encoded as:

```python
    def init_board():    # initialize chessboard
        board = ([-6, -4, -3, -2, -7, -2, -3, -4, -6],
                 [ 0,  0,  0,  0,  0,  0,  0,  0,  0],
                 [ 0, -5,  0,  0,  0,  0,  0, -5,  0],
                 [-1,  0, -1,  0, -1,  0, -1,  0, -1],
                 [ 0,  0,  0,  0,  0,  0,  0,  0,  0],
                 [ 0,  0,  0,  0,  0,  0,  0,  0,  0],
                 [ 1,  0,  1,  0,  1,  0,  1,  0,  1],
                 [ 0,  5,  0,  0,  0,  0,  0,  5,  0],
                 [ 0,  0,  0,  0,  0,  0,  0,  0,  0],
                 [ 6,  4,  3,  2,  7,  2,  3,  4,  6],)
        return board
```

- get_legal_actions: the utility function that you need to use for implementing search:

```python
def get_legal_actions(board: tuple, side: str, history: list):
  '''return the list of all legal actions according to the input board.'''
  return action_list: list
```

  - Args:
    - board: a 10×9 chessboard matrix (a tuple of ten lists). You can use board[x][y] to retrieve the content at column x and row y of the chessboard.
    - side: is a string (“red” or “black”), indicating which side you are querying.
    - history: the list of previous actions taken by both sides. It is necessary because some illegal actions are caused by history.
  - Returns:
    - action_list : a list of actions, with each action being a four-element tuple. An action is encoded as (old_x, old_y, new_x, new_y), which means moving board[old_x][old_y] to (new_x, new_y) if legal and (possibly) eating the piece at (new_x, new_y).
  - In the get_legal_actions function, the order of actions can change according to the input state. If you need a specific ordering, you can reimplement your own version in your own scripts, instead of changing utils.py.

3.4 How do I run a game between my players?

- alpha_chess/game.py contains a function “play_game” for you to start a game between two AI players and watch it via the GUI we provided.

```python
def play_game(red: Player, black: Player, delta_time: float = 1):
  ...
  return None
```

  - Args:
    - red, black: red and black players, both of which are instance objects of the Player class.
    - delta_time: the minimum delay time (in seconds) between rendering two frames. It would be useful when your players think very fast but you wish to play the moves slower.
  - alpha_chess/example.py contains an example:

```python
from game import play_game
from player_1 import player_1
from player_2 import player_2

red = player_1.Player("red")
black = player_2.Player("black")

# start a new game and watch the players play
play_game(red, black, delta_time = 1)
```

3.5 How do I evaluate a player or collect data using an existing player?

- alpha_chess/headless_game.py contains a function for letting your players play games in headless mode (silently, without rendering or GUI). It is the function we use in the final match to evaluate your submitted players.

```python
def play_headless_game(red: Player, black: Player, timeout: bool = False):
  ...
  return winner: str, text: str, history: list
```

  - Args:
    - red, black: the two players, both are instances of the Player class.
    - timeout: the switch on the timing constraint.
      - When timeout is set to true, the player who used out its thinking time (10 seconds) will lose the game immediately. This is the setup we will use during the final match.
      - When timeout is set to false (default), it means there is no limitation on the thinking time. This mode is useful for collecting learning data. It also has lower overhead due to the removal of the timing thread.
  - Returns:
    - winner: “red”, “black” and “draw”.
    - text: explanations of the result.
    - history: the record of actions taken by both sides during the game.
- alpha_chess/example.py provides a demonstration:

```python
from headless_game import play_headless_game
from player_1 import player_1
from player_2 import player_2

if __name__ == "__main__":     # for Windows OS
  red = player_1.Player("red")
  black = player_2.Player("black")

  # you can start a game for testing or collect game data by this function.
  winner, text, history = play_headless_game(red, black, timeout = True)

  print(winner, text, history) # print the result in the console window
```

- Notice: we used multiprocessing to implement the timer. So If you work on Windows OS, remember to call play_headless_game(timeout=True) inside the main function (under if __name__ == "__main__"). For example:
  - Correct:

```python
if __name__ == "__main__":     # for Windows OS
  red = player_1.Player("red")
  black = player_2.Player("black")
  winner, text, history = play_headless_game(red, black, timeout = True)
```

  - Incorrect:

```python
red = player_1.Player("red")
black = player_2.Player("black")
winner, text, history = play_headless_game(red, black, timeout = True)
if __name__ == "__main__":
   pass
```

3.6 How do I replay a past game in my dataset?

- alpha_chess/replay.py provides two functions "replay_history_game" and "replay_final_game" for you to replay a game history using our rendered GUI.

```python
def replay_history_game(history: list, delta_time: float = 1):
  ...
  return None
```

  - Args:
    - history: history recorded in a past game, a list of actions.
    - delta_time: the minimum delay time (in seconds) between rendering two frames.
- alpha_chess/example.py provides an example of usage:

```python
from replay import replay_history_game
from headless_game import play_headless_game
from player_1 import player_1
from player_2 import player_2

red = player_1.Player("red")
black = player_2.Player("black")

# get history of a game
winner, text, history = play_headless_game(red, black, timeout = False)

# replay history
replay_history_game(history, delta_time = 1)
```

- After the **final match**, we will send you files recording the history of your 10 matches. The recordings will be arranged in folders named "Player id VS Player id". For example, the match between group 1 and group 3 will be recorded in the folder named "Player 1 VS Player 3". In this folder, a file named "summary.txt" summarizes the game results. The other files are recorded game histories (for Game 1 to Game 6 or Game 7). You can then use the "replay_final_game" function provided in alpha_chess/replay.py to watch these recorded games.

```python
def replay_final_game(player_1: int, player_2: int,
                      game: int, delay_time: float = 1):
  ...
  return None
```

- Args:
  - player_1, player_2: player_id (1∼10) of the two groups.
  - game: specifies which game you want to watch (1∼6/1∼7).
  - delta_time: the time gap between rendering two frames.
- alpha_chess/example.py contains a usage example:

```python
from replay import replay_final_game
# replay the fourth game between group 2 and group 3 with delay time 0.1s.
replay_final_game(player_1 = 2, player_2 = 3, game = 4, delta_time = 0.1)
# this will automatically load the record of Game 4 in the "Player 2 VS Player
                                        3" folder
# replay_final_game(player_1 = 3, player_2 = 2, game = 4, delta_tiem = 0.1) is
                                        equivilent.
```

3.7 What if I myself want to play against my player?

- alpha_chess/pve_game.py provides a function "play_pve_game" for you to play a game against AI.

```python
def play_pve_game(side: str = "red", player: Player = None):
  ...
  return None
```

  - Args:
    - side: the side ("red" or "black") you want to take.
    - player: an instance of your Player class.
  - Note that your side must be different from player.side.
- Example from alpha_chess/example.py

```python
from player_1 import player_1
from pve_game import play_pve_game
player = player_1.Player("red")    # AI takes red side
play_pve_game("black", player)    # you take black side
```

3.8 What if I want to mock the final match?

- In alpha_chess/final_game_example.py, we have provided a fight function:

```python
def fight(player_i: int, player_j: int):
  ...
  return None
```

- Args:
  - player_i, player_j: id (1∼10) of two groups.
- This function requires the existence of "player_[ID]" folders, for example:
  - alpha_chess/player_1 folder:
    - player_1.py (you implementation of player 1)
    - Your own utility scripts (scripts that your Player class depends on)

4 Submission

You need to submit the following:

- player_[ID] folder (zip)
  - requirements.txt (the python packages that your program depends on, such as "numpy", etc..)
  - player_[ID].py (your implementation of the Player class)
  - Your own utility scripts (scripts that your Player class depends on)

5 Tips

- You are obligated to ensure that your program is bug-free and exception-free.
- Please do not touch the files we provide except example scripts.
- We will contact you if and only if your program has dependency issues during the final games.
- If you have any questions or have found any bug in the code base we provide, contact us as soon as possible.

Good Luck to All Groups!