CS440/ECE448 Spring 2024

MP07: Two-Player Games

The first thing you need to do is to download this file: mp07.zip (mp07.zip). It has the following content:

- `main.py` . This file plays the game (python3 main.py).
- `submitted.py` : Your homework. Edit, and then submit to Gradescope (https://www.gradescope.com/courses/486387).
- `mp07_notebook.ipynb` : This is a Jupyter (https://anaconda.org/anaconda/jupyter) notebook to help you debug. You can completely ignore it if you want, although you might find that it gives you useful instructions.
- `grade.py` : Once your homework seems to be working, you can test it by typing `python grade.py` , which will run the tests in `tests/tests_visible.py` .
- `tests/test_visible.py` : This file contains about half of the unit tests (https://docs.python.org/3/library/unittest.html) that Gradescope will run in order to grade your homework. If you can get a perfect score on these tests, then you should also get a perfect score on the additional hidden tests that Gradescope uses.
- `grading_examples/` . This directory contains the JSON answer keys on which your grade is based (the visible ones). You are strongly encouraged to read these, to see what format your code should produce.
- `chess/` , `res/` , `tools/` . These directories contain code and resources from PyChess that are necessary to run the assignment.
- `requirements.txt` : This tells you which python packages you need to have installed, in order to run `grade.py` . You can install all of those packages by typing `pip install -r requirements.txt` or `pip3 install -r requirements.txt` .
- `extracredit*` : These files are for extra credit only, check the ec section for details.

This file (`mp07_notebook.ipynb`) will walk you through the whole MP, giving you instructions and debugging tips as you go.

Table of Contents

I. Getting Started

The `main.py` file will be the primary entry point for this assignment. Let’s start by running it as follows:

In [1]:
```
!python3 main.py --help
```

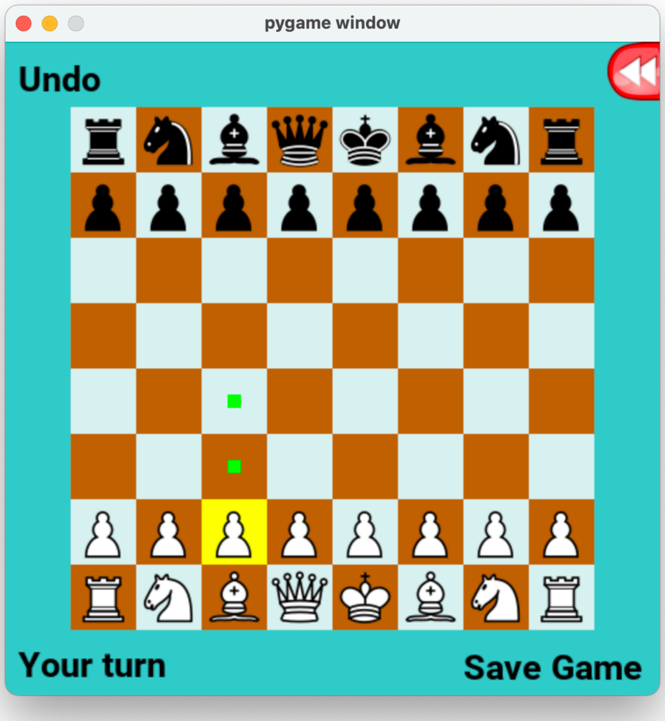
```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
usage: main.py [-h] [--player0 {random,human,minimax,alphabeta,stochas
tic}]
               [--player1 {random,human,minimax,alphabeta,stochastic}]
               [--depth0 DEPTH0] [--depth1 DEPTH1] [--breadth0 BREADTH
0]
               [--breadth1 BREADTH1] [--loadgame LOADGAME]

CS440 MP7 Chess

optional arguments:
  -h, --help            show this help message and exit
  --player0 {random,human,minimax,alphabeta,stochastic}
                        Is player 0 a human, a random player, or some
type of
                        AI? (default: human)
  --player1 {random,human,minimax,alphabeta,stochastic}
                        Is player 1 a human, a random player, or some
type of
                        AI? (default: random)
  --depth0 DEPTH0       Depth to which player 0 should search, if play
er 0 is
                        an AI. (default: 2)
  --depth1 DEPTH1       Depth to which player 1 should search, if play
er 1 is
                        an AI. (default: 2)
  --breadth0 BREADTH0   Breadth to which player 0 should search, if pl
ayer 0
                        is stochastic. (default: 2)
  --breadth1 BREADTH1   Breadth to which player 1 should search, if pl
ayer 1
                        is stochastic. (default: 2)
  --loadgame LOADGAME   Load a saved game from res/savedGames (defaul
t: None)
```

This will list the available options. You will see that both player0 and player1 can be human, or one of four types of AI: random, minimax, alphabeta, or stochastic. The default is `--player0 human --player1 random` , because the random player is the only one already implemented. In order to play against an "AI" that makes moves at random, type

In [2]:
```
!python3 main.py
```

```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
```

You should see a chess board pop up. When you click on any white piece (you may need to double-click), you should see bright neon green dots centered in all of the squares to which that piece can legally move, like this:



When you click (or double-click) on one of those green dots, your piece will move there. Then the computer will move one of the black pieces, and it will be your turn again.

If you have trouble using the mouse to play, you can debug your code by watching the computer play against itself. For example,

```
python3 main.py --player0 random --player1 random
```

If you want to start from one of the stored game positions, you can load them as, for example:

```
python3 main.py --loadgame game1.txt
```

We will grade your submissions using `grade.py`. This file is available to you, so that you can understand how this assignment will be graded.

Let's see what happens when we run this script:

In [3]:
```
!python3 grade.py
```

```
.......
----------------------------------------------------------------------
Ran 7 tests in 6.327s

OK
```

As you can see, all of the tests raise `NotImplementedError`, because we have not yet implemented the functions `minimax` or `alphabeta` in `submitted.py`. We will do this in the next few sections.

II. the PyChess API

The chess-playing interface that we're using is based on **PyChess** (https://github.com/pychess/pychess). All the components of PyChess that you need are included in the assignment5.zip file, but if you want to learn more about PyChess, you are welcome to download and install it. The standard distribution of PyChess includes a game-playing AI using the alphabeta search algorithm. You are welcome to read their implementation to get hints for how to write your own, but note that we have changed the function signature so that if you simply cut and paste their code into your own, it will not work.

You do not need to know how to play chess in order to do this assignment. You need to know that chess is a game between two players, one with white pieces, one with black pieces. White goes first. Players alternate making moves until white wins, black wins, or there is a tie. You don't need to know anything else about chess to do the assignment, though you may have more fun if you learn just a little (e.g., by playing against the computer).

Though you don't need to know anything about chess, you do need to understand a few key concepts, and a few key functions, from the PyChess API. The most important concepts are:

1. **player**. There are two players: Player 0, and Player 1. Player 0 plays white pieces, Player 1 black. Player 0 goes first.
    - **side**. PyChess keeps track of whose turn it is by using a boolean called **side**: `side==False` if Player 0 should play next.
2. **move**. A move is a 3-list: `move==[fro,to,promote]`. `fro` is a 2-list: `fro==[from_x,from_y]`, where `from_x` and `from_y` are each numbers between 1 and 8, specifying the starting x and y positions. `to` is also a 2-list: `to==[to_x,to_y]`. `promote` is either `None` or `"q"`, where `q` means that you are trying to promote your piece to a queen.
3. **board**. A board is a 2-tuple of lists of pieces: `board==([white_piece0, white_piece1, ...], [black_piece0, black_piece1, ...])`. Each piece is a 3-list: `piece=[x,y,type]`. `x` is the x position of the piece (left-to-right, 1 to 8). `y` is the y position of the piece (top-to-bottom, 1 to 8). `type` is a letter indicating the type of piece, which can be (`p`=pawn, `r`=rook, `n`=knight, `b`=bishop, `q`=queen, or `k`=king).

To get some better understanding, let's look at the way the board is initialized at the start of the game. The function `chess.lib.convertMoves` starts with an initialized board, then runs forward through a series of specified moves, and gives us the resulting board. If the series of specified moves is the empty string, then `chess.lib.convertMoves` gives us the opening board position:

```
In [4]: import chess.lib.utils, pprint

        side, board, flags = chess.lib.convertMoves("")

        print("Should we start with the White player?", side)

        print("")

        print("The starting board position is:")

        pprint.PrettyPrinter(compact=True,width=40).pprint(board)
```

```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
Should we start with the White player? False

The starting board position is:
[[[1, 7, 'p'], [2, 7, 'p'], [3, 7, 'p'],
  [4, 7, 'p'], [5, 7, 'p'], [6, 7, 'p'],
  [7, 7, 'p'], [8, 7, 'p'], [1, 8, 'r'],
  [2, 8, 'n'], [3, 8, 'b'], [4, 8, 'q'],
  [5, 8, 'k'], [6, 8, 'b'], [7, 8, 'n'],
  [8, 8, 'r']],
 [[1, 2, 'p'], [2, 2, 'p'], [3, 2, 'p'],
  [4, 2, 'p'], [5, 2, 'p'], [6, 2, 'p'],
  [7, 2, 'p'], [8, 2, 'p'], [1, 1, 'r'],
  [2, 1, 'n'], [3, 1, 'b'], [4, 1, 'q'],
  [5, 1, 'k'], [6, 1, 'b'], [7, 1, 'n'],
  [8, 1, 'r']]]
```

II.A evaluate

The function `value=evaluate(board)` returns the heuristic value of the board for the white player (thus, in the textbook's terminology, the white player is Max, the black player is Min).

For example, you can find the numerical value of a board by typing:

In [5]:
```
import chess.lib.heuristics

# Try the default board
value = chess.lib.heuristics.evaluate(board)
print("The value of the default board is",value)

# Try a board where Black is missing a rook
board2 = [ board[0], board[1][:-1] ]
value = chess.lib.heuristics.evaluate(board2)
print("If we eliminate one of black's rooks, the value is ",value)

# Try a board where White is missing a rook
board3 = [ board[0][:-1], board[1] ]
value = chess.lib.heuristics.evaluate(board3)
print("If we eliminate one of white's rooks, the value is ",value)

# Eliminate one piece from each player
board4 = [ board[0][:-1], board[1][:-1] ]
value = chess.lib.heuristics.evaluate(board4)
print("If the players are each missing a rook, the value is ",value)
```

```
The value of the default board is 0.0
If we eliminate one of black's rooks, the value is  14.0
If we eliminate one of white's rooks, the value is  -14.0
If the players are each missing a rook, the value is  0.0
```

II.B encode and decode

Lists cannot be used as keys in a dict, therefore, in order to give your `moveTree` to the autograder, you will need some way to encode the moves. `encoded=encode(*move)` converts a `move` into a string representing its standard chess encoding. The **decode** function reverses the processing of `encode`. For example:

In [6]:
```
from chess.lib.utils import encode, decode

# This statement evaluates to True
move1 = encode([7,2],[7,4],None)
print("The move [7,2]->[7,4] encodes as",move1)

# This statement also evaluates to True
move2=encode([5,7],[5,8],"q")
print("The move [5,7]->[5,8] with promotion to queen is encoded as",mo
ve2)

# This statement evaluates to True
move3 = decode("g7g5")
print("The move g7g5 is decoded to",move3)
```

```
The move [7,2]->[7,4] encodes as g7g5
The move [5,7]->[5,8] with promotion to queen is encoded as e2e1q
The move g7g5 is decoded to [[7, 2], [7, 4], None]
```

II.C generateMoves, convertMoves, makeMove

The function **generateMoves** is a generator (https://docs.python.org/3/glossary.html#term-generator) that generates all moves that are legal on the current board. The function **convertMoves** generates a starting board. The function **makeMove** implements a move, and returns the resulting board (and side and flags). For example, the following code prints all of the moves that white can legally make, starting from the beginning board:

In [7]:
```
import submitted, importlib
import chess.lib

# Create an initial board
side, board, flags = chess.lib.convertMoves("")

# Iterate over all moves that are legal from the current  board positi
on.
for move in submitted.generateMoves(board, side, flags):
    newside, newboard, newflags = chess.lib.makeMove(side, board, move
[0], move[1], flags, move[2])
    print("This move is legal now:",move, newflags)
```

```
This move is legal now: [[1, 7], [1, 5], None] ([True, True, True, Tru
e], [1, 6])
This move is legal now: [[1, 7], [1, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[2, 7], [2, 5], None] ([True, True, True, Tru
e], [2, 6])
This move is legal now: [[2, 7], [2, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[3, 7], [3, 5], None] ([True, True, True, Tru
e], [3, 6])
This move is legal now: [[3, 7], [3, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[4, 7], [4, 5], None] ([True, True, True, Tru
e], [4, 6])
This move is legal now: [[4, 7], [4, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[5, 7], [5, 5], None] ([True, True, True, Tru
e], [5, 6])
This move is legal now: [[5, 7], [5, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[6, 7], [6, 5], None] ([True, True, True, Tru
e], [6, 6])
This move is legal now: [[6, 7], [6, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[7, 7], [7, 5], None] ([True, True, True, Tru
e], [7, 6])
This move is legal now: [[7, 7], [7, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[8, 7], [8, 5], None] ([True, True, True, Tru
e], [8, 6])
This move is legal now: [[8, 7], [8, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[2, 8], [3, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[2, 8], [1, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[7, 8], [8, 6], None] ([True, True, True, Tru
e], None)
This move is legal now: [[7, 8], [6, 6], None] ([True, True, True, Tru
e], None)
```

The **flags** and **newflags** variables specify whether or not it has become legal for black to make certain specialized types of moves. For more information, see `chess/docs.txt` .

II.D random

In order to help you understand the API, the file `submitted.py` contains a function from which you can copy any useful code. The function `moveList, moveTree, value  = random(side, board, flags, chooser)` takes the same input as the functions you will write, and generates the same type of output, but instead of choosing a smart move, it chooses a move at random.

Here, the input parameter `chooser` is set to `chooser=` random.choice (https://docs.python.org/3/library/random.html#functions-for-sequences) during normal game play, but during grading, it will be set to some other function that selects a move in a non-random fashion. Use this function as if it were equivalent to `random.choice` .

In [8]:
```python
import submitted, importlib
importlib.reload(submitted)
help(submitted.random)
```

```
Help on function random in module submitted:

random(board, side, flags, chooser)
    Return a random move, resulting board, and value of the resulting
board.
    Return: (value, moveList, boardList)
      value (int or float): value of the board after making the chosen
move
      moveList (list): list with one element, the chosen move
      moveTree (dict: encode(*move)->dict): a tree of moves that were
evaluated in the search process
    Input:
      board (2-tuple of lists): current board layout, used by generate
Moves and makeMove
      side (boolean): True if player1 (Min) plays next, otherwise Fals
e
      flags (list of flags): list of flags, used by generateMoves and
makeMove
      chooser: a function similar to random.choice, but during autogra
ding, might not be random.
```

III. Assignment

For this assignment, you will need to write three functions: `minimax` and `alphabeta` . The content of these functions is described in the sections that follow.

III.A minimax search

For Part 1 of this assignment, you will implement minimax search. Specifically, you will implement a function `minimax(side, board, flags, depth)` in `search.py` with the following docstring:

```
In [9]: importlib.reload(submitted)
        help(submitted.minimax)
```

```
Help on function minimax in module submitted:

minimax(board, side, flags, depth)
    Return a minimax-optimal move sequence, tree of all boards evaluat
ed, and value of best path.
    Return: (moveList, moveTree, value)
      moveList (list): the minimax-optimal move sequence, as a list of
moves
      moveTree (dict: encode(*move)->dict): a tree of moves that were
evaluated in the search process
      value (float): value of the final board in the minimax-optimal m
ove sequence
    Input:
      board (2-tuple of lists): current board layout, used by generate
Moves and makeMove
      side (boolean): True if player1 (Min) plays next, otherwise Fals
e
      flags (list of flags): list of flags, used by generateMoves and
makeMove
      depth (int >=0): depth of the search (number of moves)
```

As you can see, the function accepts `side`, `board`, and `flags` variables, and a non-negative integer, `depth`. It should perform minimax search over all possible move sequences of length `depth`, and return the complete tree of evaluated moves as `moveTree`. If `side==True`, you should choose a path through this tree that minimizes the heuristic value of the final board, knowing that your opponent will be trying to maximize value; conversely if `side==False`. Return the resulting optimal list of moves (including moves by both white and black) as `moveList`, and the numerical value of the final board as `value`.

A note about `depth`: The `depth` parameter specifies the total number of moves, including moves by both white and black. If `depth==1` and `side==False`, then you should just find one move, from the current board, that maximizes the value of the resulting board. If `depth==2` and `side==False`, then you should find a white move, and the immediate following black move. If `depth==3` and `side==False`, then you should find a white, black, white sequence of moves. For example, see [wikipedia's page on minimax (https://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves)](https://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves) for examples and pseudo-code.

You are strongly encouraged to look at the grading examples in the `grading_examples` folder, to get a better understanding of what the `minimax` function outputs should look like. For example, the board game in `more grading_examples/minimax_game0_depth2.json` contains `value` on the first line, `moveList` on the second line, and `moveTree` on the third line:

In [10]:
```
import json, pprint
with open("grading_examples/minimax_game0_depth2.json","r") as f:
    value=json.loads(f.readline())
    print("value is",value,"\n")
    moveList=json.loads(f.readline())
    print("moveList is",moveList,"\n")
    moveTree=json.loads(f.readline())
    print("moveTree is:",moveTree,"\n")
```

```
value is 10.0

moveList is [[[2, 1], [3, 3], None], [[4, 7], [4, 5], None]]

moveTree is: {'a7a5': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {},
'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3':
{}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b
5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f
3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'a7a6': {'a2a4': {}, 'a2a3': {}, 'b
2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {},
'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5':
{}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g
5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'b7b5': {'a2a
4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e
2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {},
'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4':
{}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g
1': {}}, 'b7b6': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d
4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h
2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {},
'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4':
{}, 'f3d4': {}, 'h1g1': {}}, 'c7c5': {'a2a4': {}, 'a2a3': {}, 'b2b4':
{}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g
4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c
3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {},
'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'c7c6': {'a2a4': {},
'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4':
{}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b
1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f
3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}},
'd7d5': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {},
'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4':
{}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e
4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f
3d4': {}, 'h1g1': {}}, 'd7d6': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b
2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {},
'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1':
{}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e
5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'e7e5': {'a2a4': {}, 'a2a
3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e
2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {},
'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1':
{}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'e7e
6': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d
3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h
2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {},
'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4':
{}, 'h1g1': {}}, 'f7f5': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3':
{}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g
3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c
3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {},
'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'f7f6': {'a2a4': {}, 'a2a3': {},
'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3':
{}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d
5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f
3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'g5g4': {'a
2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {},
'e2e4': {}, 'e2e3': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1':
{}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g
1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}},
'h7h5': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {},
'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4':
{}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e
4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f
3d4': {}, 'h1g1': {}}, 'h7h6': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b
2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {},
'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1':
{}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e
5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'b8c6': {'a2a4': {}, 'a2a
3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e
2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {},
'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1':
{}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'b8a
6': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d
3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h
2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {},
'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4':
{}, 'h1g1': {}}, 'f8g7': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3':
{}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g
3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c
3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {},
'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'f8h6': {'a2a4': {}, 'a2a3': {},
'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3':
{}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d
5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f
3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h1g1': {}}, 'g8h6': {'a
2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd2d4': {}, 'd2d3': {},
'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {}, 'h2h4': {}, 'h2h3':
{}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5': {}, 'c3e4': {}, 'c3a
4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h4': {}, 'f3d4': {}, 'h
1g1': {}}, 'g8f6': {'a2a4': {}, 'a2a3': {}, 'b2b4': {}, 'b2b3': {}, 'd
2d4': {}, 'd2d3': {}, 'e2e4': {}, 'e2e3': {}, 'g2g4': {}, 'g2g3': {},
'h2h4': {}, 'h2h3': {}, 'a1b1': {}, 'c3d5': {}, 'c3b1': {}, 'c3b5':
{}, 'c3e4': {}, 'c3a4': {}, 'f3g1': {}, 'f3g5': {}, 'f3e5': {}, 'f3h
4': {}, 'f3d4': {}, 'h1g1': {}}}
```

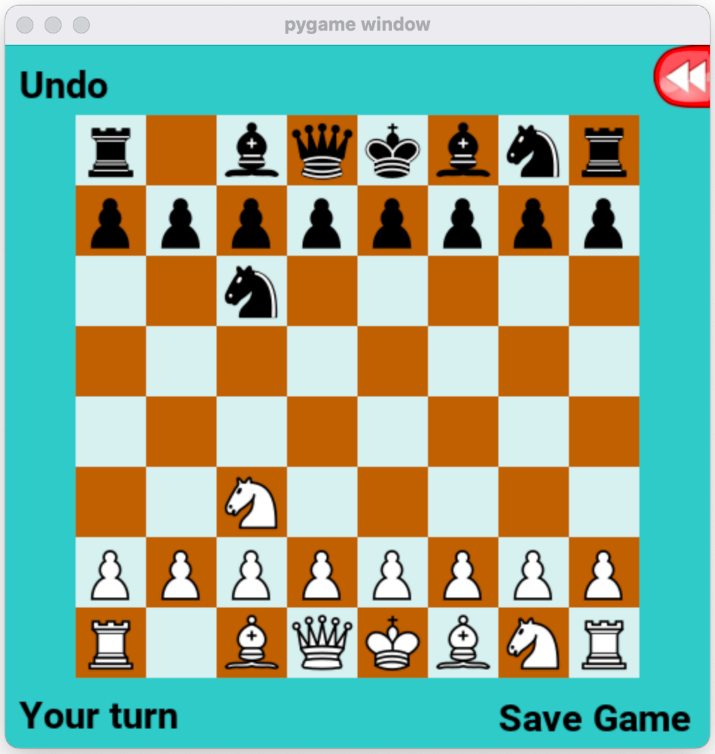
You will certainly want to implement `minimax` as a recursive function. You will certainly want to use the function `generateMoves` to generate all moves that are legal in the current game state, and you will certainly want to use `makeMove` to find the newside, newboard, and newflags that result from making each move. When you get to `depth==0` , you will certainly want to use `evaluate(board)` in order to compute the heuristic value of the resulting board.

Once you have implemented minimax, you can test it by playing against it. If you have pygame installed, the following line should pop up a board on which you can play against your own minimax player. If you have not yet written `minimax` , the game will throw a `NotImplementedError` when it is white's turn to move.

In [11]:
```
!python3 main.py --player1 minimax
```

```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
```

Test that it is working correctly by moving one of your knights forward. The computer should respond by moving one of its knights foward, as shown here:



If you want to watch a minimax agent win against a random-move agent, you can type

In [12]:
```
!python3 main.py --player0 minimax --player1 random
```

```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
2024-03-02 16:39:26.133 Python[1012:17745797] TSM AdjustCapsLockLEDFor
KeyTransitionHandling - _ISSetPhysicalKeyboardCapsLockLED Inhibit
```

III.B alphabeta search

For Part 2 of this assignment, you will implement alphabeta search. Specifically, you will implement a function `alphabeta(side, board, flags, depth)` in `search.py` with the following docstring:

In [13]:
```
import submitted
help(submitted.alphabeta)
```

```
Help on function alphabeta in module submitted:

alphabeta(board, side, flags, depth, alpha=-inf, beta=inf)
    Return minimax-optimal move sequence, and a tree that exhibits alp
habeta pruning.
    Return: (moveList, moveTree, value)
      moveList (list): the minimax-optimal move sequence, as a list of
moves
      moveTree (dict: encode(*move)->dict): a tree of moves that were
evaluated in the search process
      value (float): value of the final board in the minimax-optimal m
ove sequence
    Input:
      board (2-tuple of lists): current board layout, used by generate
Moves and makeMove
      side (boolean): True if player1 (Min) plays next, otherwise Fals
e
      flags (list of flags): list of flags, used by generateMoves and
makeMove
      depth (int >=0): depth of the search (number of moves)
```

For any given input board, this function should return exactly the same value and moveList as `minimax` ; the only difference between the two functions will be the returned `moveTree` . The tree returned by `alphabeta` should have fewer leaf nodes than the one returned by `minimax` , because alphabeta pruning should make it unnecessary to evaluate some of the leaf nodes.

You can test this using:

In [14]:
```
!python3 main.py --player0 random --player1 alphabeta
```

```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
```

IV. Extra Credit

The heuristic we've been using, until now, is the default PyChess heuristic: it assigns a value to each piece, with extra points added or subtracted depending on the piece's location. For extra credit, if you wish, you can try to train a neural network to compute a better heuristic.

IV.A. The Game

The extra credit assignment will be graded based on how often your heuristic beats the default PhChess heuristic in a two-player game.

Ideally, the game would be chess. Unfortunately, grading your heuristic based on complete chess games would take too much time. Instead, the function `extracredit_grade.py` plays a very simple game:

1. Each of the two players is given the same chess board. Each of you assigns a numerical value to the board.
2. Then, the scoring program finds the depth-two minimax value of the same board. This value is provided in the lists called "values" in the data files `extracredit_train.txt` and `extracredit_validation.txt`; but if you have already completed the main assignment, it should be the same value that you'd get by running your `minimax` or `alphabeta` search with `depth=2`.
3. The winner of the game is the player whose computed value is closest to the reference value.

Notice that what we're asking you to do, basically, is to create a neural network that can guess the value of the PyChess heuristic two steps ahead.

Notice that, if you can design a funny neural net architecture that, instead of being trained to solve this problem, solves it without training by exactly computing a two-step minimax operation, then you're done. This is explicitly allowed, because we think it would be a very effective and very interesting solution.

Most of you, we guess, will choose a more general neural net architecture, and train it so that it imitates the results of two-step minimax.

IV.B. Distributed Code: Exactly Reproduce the PyChess Heuristic

You will find the following files for the extra credit:

- extracredit.py: Trains the model. This is the code you will edit and submit.
- extracredit_embedding.py: Embeds a chess board into a (15x8x8) binary pytorch tensor.
- extracredit_train.txt: Training data: sequences of moves, and corresponding values.
- extracredit_validation.txt: Validation data: sequences of moves, and corresponding values.
- extracredit_grade.py: The grading script.

Try the following:

In [15]:
```
!python3 extracredit.py
```

```
pygame 2.3.0.dev3 (SDL 2.0.14, Python 3.8.3)
Hello from the pygame community. https://www.pygame.org/contribute.htm
l
extracredit_train.txt loaded board number 10000...
/Users/fanxulin/Downloads/mp07/extracredit_embedding.py:107: UserWarni
ng: Creating a tensor from a list of numpy.ndarrays is extremely slow.
Please consider converting the list to a single numpy.ndarray with num
py.array() before converting to a tensor. (Triggered internally at /Us
ers/runner/work/pytorch/pytorch/pytorch/torch/csrc/utils/tensor_new.cp
p:233.)
  self.embeddings = torch.tensor(embeddings,dtype=DTYPE,device=DEVICE)
initial model, validationloss = 5.9855
epoch 0, trainloss = 5.16544, validationloss = 5.98202
epoch 1, trainloss = 5.15432, validationloss = 5.97888
epoch 2, trainloss = 5.12396, validationloss = 5.97579
epoch 3, trainloss = 5.11742, validationloss = 5.97256
epoch 4, trainloss = 5.14128, validationloss = 5.96929
epoch 5, trainloss = 5.1591, validationloss = 5.96601
epoch 6, trainloss = 5.16684, validationloss = 5.96271
epoch 7, trainloss = 5.13396, validationloss = 5.95942
epoch 8, trainloss = 5.15179, validationloss = 5.95617
epoch 9, trainloss = 5.11968, validationloss = 5.95456
epoch 10, trainloss = 5.13249, validationloss = 5.95414
epoch 11, trainloss = 5.14369, validationloss = 5.95374
epoch 12, trainloss = 5.11408, validationloss = 5.9534
epoch 13, trainloss = 5.12687, validationloss = 5.95305
epoch 14, trainloss = 5.1237, validationloss = 5.95269
epoch 15, trainloss = 5.14522, validationloss = 5.95233
epoch 16, trainloss = 5.11821, validationloss = 5.95198
epoch 17, trainloss = 5.14071, validationloss = 5.95163
epoch 18, trainloss = 5.13708, validationloss = 5.95128
epoch 19, trainloss = 5.1357, validationloss = 5.95091
epoch 20, trainloss = 5.12911, validationloss = 5.95059
epoch 21, trainloss = 5.13696, validationloss = 5.95024
epoch 22, trainloss = 5.13054, validationloss = 5.94986
epoch 23, trainloss = 5.11434, validationloss = 5.94952
epoch 24, trainloss = 5.1417, validationloss = 5.94915
epoch 25, trainloss = 5.14091, validationloss = 5.94877
epoch 26, trainloss = 5.13796, validationloss = 5.94844
epoch 27, trainloss = 5.14031, validationloss = 5.94807
epoch 28, trainloss = 5.1213, validationloss = 5.94776
epoch 29, trainloss = 5.13525, validationloss = 5.94742
epoch 30, trainloss = 5.12676, validationloss = 5.94707
epoch 31, trainloss = 5.12602, validationloss = 5.94674
epoch 32, trainloss = 5.12823, validationloss = 5.94636
epoch 33, trainloss = 5.13483, validationloss = 5.94601
epoch 34, trainloss = 5.11632, validationloss = 5.94566
epoch 35, trainloss = 5.1131, validationloss = 5.94534
epoch 36, trainloss = 5.14313, validationloss = 5.94496
epoch 37, trainloss = 5.10264, validationloss = 5.94461
epoch 38, trainloss = 5.12092, validationloss = 5.94427
epoch 39, trainloss = 5.11243, validationloss = 5.9439
epoch 40, trainloss = 5.1033, validationloss = 5.94355
epoch 41, trainloss = 5.10993, validationloss = 5.94317
epoch 42, trainloss = 5.10801, validationloss = 5.94286
epoch 43, trainloss = 5.12233, validationloss = 5.94254
epoch 44, trainloss = 5.12883, validationloss = 5.9422
epoch 45, trainloss = 5.11561, validationloss = 5.94182
epoch 46, trainloss = 5.14705, validationloss = 5.9415
epoch 47, trainloss = 5.14881, validationloss = 5.94113
epoch 48, trainloss = 5.1223, validationloss = 5.94088
epoch 49, trainloss = 5.11798, validationloss = 5.94073
```

What just happened? Well, if you open `extracredit.py` you will find these lines:

```
# Well, you might want to create a model a little better than this...
    model = torch.nn.Sequential(torch.nn.Flatten(),torch.nn.Linear(in_featur
es=8*8*15, out_features=1))

    # ... and if you do, this initialization might not be relevant any more
...
    model[1].weight.data = initialize_weights()
    model[1].bias.data = torch.zeros(1)

    # ... and you might want to put some code here to train your model:
    trainset = ChessDataset(filename='extracredit_train.txt')
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=1000, shu
ffle=True)
    for epoch in range(100):
        for x,y in trainloader:
            pass # Replace this line with some code that actually does the t
raining

    # ... after which, you should save it as "model_ckpt.pkl":
    torch.save(model, 'model_ckpt.pkl')
```

The `torch.nn.Sequential` line has flattened the input board embedding, and then multiplied it by a matrix. The function `initialize_weights()` initializes that matrix to be exactly equal to the weights used in the PyChess linear heuristic. If you look closely at the training part of the code, you will see that it has done nothing at all; this part of the code is only here to show you how the ChessDataset and DataLoader can be used. After the `pass` , you see that the model, initialized but not trained, has been saved to `model_ckpt.pkl` .

IV.C. Leaderboard

Now that you've saved `model_ckpt.pkl` , you can score it using `extracredit_grade.py` :

```
In [16]: !python3 extracredit_grade.py
```

```
/Users/fanxulin/Downloads/mp07/extracredit_embedding.py:107: UserWarni
ng: Creating a tensor from a list of numpy.ndarrays is extremely slow.
Please consider converting the list to a single numpy.ndarray with num
py.array() before converting to a tensor. (Triggered internally at /Us
ers/runner/work/pytorch/pytorch/pytorch/torch/csrc/utils/tensor_new.cp
p:233.)
  self.embeddings = torch.tensor(embeddings,dtype=DTYPE,device=DEVICE)
{
  "tests": [
    {
      "name": "validoreval_winratio_gt_0.5",
      "score": 10.0,
      "max_score": 10
    }
  ],
  "visibility": "visible",
  "execution_time": 0.3578357696533203,
  "score": 10.0,
  "leaderboard": [
    {
      "name": "winratio_evaluation",
      "value": -1
    },
    {
      "name": "winratio_validation",
      "value": 0.591
    },
    {
      "name": "Time",
      "value": 0.3578357696533203
    }
  ]
}
```

The `leaderboard` section shows two separate scores:

- `winratio_validation` is the fraction, of the 1000 boards in `extracredit_validation.txt` , in which your neural net beat the PyChess heuristic.
- `winratio_evaluation` is the same thing, but for the boards in `extracredit_evaluation.txt` which you shouldn't have access to. This is used for final grading with the autograder and will always be -1 locally.

IV.D. Extra Credit Grade

Your extra credit grade is given by the variable `score` in the output from `extracredit_grade.py`. It is calculated as

```
score =(2 * (grade_ratio >= 0.5)) + 100 * min(0.08, max(grade_ratio - 0.5, 
0))
```

where grade_ratio is either winratio_validation or winratio_evaluation depending on whether you are grading locally or on gradescope. You will get 2 points for getting winratio = 0.5 and full 10 points if you can get winratio higher than 0.58.

Note: Final grade is determined by the score on gradescope which will be different from local score due to the difference in validation and evaluation set. Local score is for your reference only.

IV.E. Submission Instructions

Your file `extracredit.py` must create a pytorch `torch.nn.Module` object, and then save it in a file called `model_ckpt.pkl`.

Pre-trained models (of any interestingly large size) cannot be uploaded to Gradescope, so your model will have to be created, trained, and saved by the file `extracredit.py`.

You are strongly encouraged to load `extracredit_train.txt` and/or `extracredit_validation.txt` in your `extracredit.py` function.

In order to allow you to train interesting neural nets, we've set up Gradescope to allow you up to 40 minutes of CPU time (on one CPU). It is possible to get full points for this extra credit assignment in three or four minutes of training, but some of you may want to experiment with bigger models.

Warning: Hard to beat alpha-beta!

The extra credit thresholds are set so that you can get all of the points if you do well enough on the training data, even if your result is massively overtrained and doesn't generalize well to validation or evaluation data. That's because it's actually really hard to find a neural model that beats alpha-beta by any significant margin. Deep Blue (https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)) did not use a neural network explicitly, though it had a number of parameters in its evaluation function that were trained using machine-learning-like techniques.

V. Grade your homework

Submit the main part of this assignment by uploading `submitted.py` to Gradescope. You can upload other files with it, but only `submitted.py` will be retained by the autograder.

Submit the extra credit part of this assignment by uploading `extracredit.py` to Gradescope. You can upload other files with it, but only `extracredit.py` will be retained by the autograder.

```
In [ ]:
```