327 Object-oriented programming

Homework 5

Due Tuesday 12/6/2020

Description

This assignment picks up where homework 4 left off. You will build on the code you wrote for Checkers in order to implement Chess as well. You will then write a smarter AI player for both games using minimax search.

Chess rules and modifications

You can find the rules with some nice examples here.

https://www.chess.com/learn-how-to-play-chess

We will make the following changes to the rules to simplify the game slightly.

- When a pawn is promoted, instead of giving the player a choice, automatically replace it with a queen.
- Do not implement castling
- Do not implement "en passant"
- You do not need to enforce check or checkmate. A player is allowed to make a move that puts themselves in check and if they are in check they are not required to get out of check. Instead of ending the game with a checkmate, a player wins by capturing the opponent's king. This has less impact on that game than it seems at first, since a good player (or a good AI) will not put leave themselves in check unless they have no other moves, in which case they would be in checkmate anyway.
- For draws we will only worry about the case where a player hasn't lost, but has no moves available. This is the same as the previous assignment.

Minimax search

Finding moves that lead to a win is a search problem. Consider a tree of game states where the root is the initial game set up and then every move is a branch to a child game state. Leaf nodes in this tree represent a loss, draw, or win. You want to explore this tree until you find a leaf corresponding to a win, then you make the moves along the path to that leaf. There are a couple problems though. First, each player can control their own moves, but not those of their opponent, so we can't take the easiest path to victory. Instead we assume that the opponent is also optimizing their moves. This is the idea behind minimax search. Second, even for a fairly simple game like Checkers, the size of the search space is 5×10^{20}. This is the largest game that has been solved and it took 18 years of calculations to do so. We want to play the game in real time without a super computer so we can't afford to search the tree all the way to a leaf at the start of a game. We can only look ahead a certain number of moves and we won't know at that point who has won. Instead we need a heuristic to score the state of the game. This is a best guess as to who is winning and by how much. By repeatedly taking moves that maximize our score as far out as we can predict, eventually we get near the bottom of the tree where we can see if it is a win, loss, or draw.

Carefully read this web page about minimax search. This includes a more complete explanation, some figures, and pseudocode that you can translate into your application.

https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm

Requirements

Implementing Chess

Throughout this assignment your goal is to re-use as much code as possible from the previous assignment. We expect Checkers to continue working properly after any necessary refactoring and we will re-use test cases from the previous assignment to verify this. Code that is specific to Chess or Checkers should be encapsulated. As a reference point, my solution to both assignments comes out to 1030 lines of code. 660 lines are generic and shared between both games. 220 lines are specific to Chess and 150 are specific to Checkers.

- **10 points** Construct, set up, and display a Chess board. We will only use 8x8 boards for Chess. If another value is given for size alongside the chess option, you may simply exit.

```
1 ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
3 ■ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ □ ■ □ ■
7 ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
  a b c d e f g h
```

- **30 points** Enable two human players (or one playing both sides) to play the game via command line inputs. This should work the same way it did in the previous homework in terms of selecting a piece and displaying its available moves. Keep in mind the rule simplifications above. While there are more pieces with more movement options, overall this is less complicated than Checkers because there is no branching recursion and no overarching restriction on movement (since we aren't concerned with check).

- **5 points** Check for victory at the start of a turn. If the current player has lost their King, then the other player wins. You should also check whether the game has ended in a draw by the same rules as Checkers.
- **15 points** Your Random player from the previous assignment should be able to play Chess in the same way. Modify the Simple player so that instead of counting the number of captures for a move, it takes the move that captures the greatest total value of pieces. For Checkers, count a man as 1 and a king as 2 and sum any pieces captured. For chess there is only ever one piece captured at a time, but their values have a wider range. Treat a pawn as 1, bishops and knights as 3, rooks as 5, queens as 9, and kings as 100. When you have finished this step, your AI players should be generic with no need to know which game they are playing. Additionally, make the AI players print their selected move after printing the game turn and current player (see examples below).
- **5 points** Undo/redo should work for Chess the same way it did in the previous assignment.
- **15 points** Create a UML diagram demonstrating how your design and use of patterns enabled sharing/re-use of generic code.

Smarter AI players

- **20 points** Create a new type of player that applies a fixed-depth minimax search to find its next move. This player should be able to play both Checkers and Chess. For our heuristic we will use the piece values mentioned above and score a game state by summing the values of all pieces for each side and taking the difference. The depth chosen tells the algorithm how many turns into the future it should look. The deeper you go the more likely you are to find the best move, but the longer the computation will take. Since the runtime is exponential in the chosen depth, you should make the depth configurable to give us flexibility when running your code on different hardware. Specify the depth for the player in the CLI arguments as shown below. Alpha-beta pruning is not specifically required, but you may implement it along with other optional improvements...

Potential extra credit

For those of you looking for an extra challenge I am considering running a tournament for your AI players. There are some technical details that need to be worked out before I can guarantee anything. I will provide an update on this later on, but in the meantime you can be thinking about potential improvements to make your AI smarter.

- You can apply alpha-beta pruning to reduce unnecessary work in the minimax search.
- You can parallelize the search to make use of multiple CPU cores.
- You can save and re-use the evaluation of game states you have seen before (wrote learning).
- You can create a more wholistic heuristic that takes into account more than piece values. This could include control of certain parts of the board or flexibility to move backline pieces. In general, this can be manually specified or derived from training data using machine learning techniques.

Running your code

Your code should take in arguments from the command line to configure the game, types of players, the size of the board, and whether history (undo/redo) should be enabled or not. Note that you will want history off when running two computers against each other so that you don't have to type next every turn. Up to five arguments may be passed in to `main.py` from the command line. You cannot give an argument without supplying the ones before it. The arguments that are omitted from the end will have the default values. Therefore `python3 main.py` is equivalent to `python3 main.py chess human human 8`. Note that the minimax player specifies the depth as an integer after the word `minimax`.

arg	allowed values	default
argv[1]	checkers, chess	chess
argv[2]	human, simple, random, minimax#	human
argv[3]	human, simple, random, minimax#	human
argv[4]	8-26 (only 8 for chess)	8
argv[5]	history	None

Other examples...

```
 python3 main.py checkers simple random
python3 main.py chess random random 12
python3 main.py chess human random 8 history
python3 main.py checkers minimax6 random 8 history
python3 main.py chess minimax2 minimax3 8
```

Additional guidance

- If you were successful in the previous homework then you will find that this isn't much additional work, but it will put the flexibility of your design to the test. If you were unsuccessful in the previous homework, I will make a solution available on Nov. 30th that you can use as a starting point for this assignment. While technically anyone can use my solution, you should be more familiar with your own code and it should be a better learning experience to refactor your own and improve on your design.
- Think about how you can re-use code for movements of chess pieces. Rooks, Bishops, and Queens are all nearly identical in terms of movement.
- When writing the minimax search you need to be careful about how you apply moves. You can think of it like you are exploring multiple alternate timelines and you don't want a move in one branch to affect another branch. You also don't want to change the original game state until you have chosen which move you actually want to make. Think about how you implemented undo/redo and that should help with this. It can be done by copying game states for each branch or by making moves reversible if you used a command pattern.
- When testing your minimax player, you can expect it to beat a random player at Chess almost every time even with a depth of only 2 or 3. For Checkers you will

find that you are able to search at a greater depth because the game is simpler, but you may still run into draws frequently. This is partly because the jump requirement forces the random player to make moves that often aren't too bad. There are also many more ways to end up in a draw. In fact, now that Checkers has been solved, we know that it ends in a draw if both players play perfectly (similar to Tic-Tac-Toe).

Testing

You are strongly encouraged to write unittests for your code. Even some simple test cases written early on (setting up the board, printing it out) will be helpful as you go. However, we will not specifically check your tests in this assignment.

Submissions

Submit your project to gradescope with whatever files are necessary to run `main.py`. In the same submission, include your UML diagram in PDF format. Public tests are coming soon. You can expect very similar tests to those in the previous assignment. We will also test your minimax player to see that it runs in a reasonable time and is able to beat random players consistently.

Example input/outputs

```
 $ python3 main.py chess
1 ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
3 ■ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ □ ■ □ ■
7 ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
  a b c d e f g h
Turn: 1, white
Select a piece to move
e7
0: move: e7->e6
1: move: e7->e5
Select a move by entering the corresponding index
1
1 ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
3 ■ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ♙ □ ■ □
6 □ ■ □ ■ □ ■ □ ■
7 ♙ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
  a b c d e f g h
Turn: 2, black
Select a piece to move
g1
0: move: g1->h3
1: move: g1->f3
Select a move by entering the corresponding index
1
1 ♜ ♞ ♝ ♛ ♚ ♝ ■ ♜
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
3 ■ □ ■ □ ■ ♞ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ♙ □ ■ □
6 □ ■ □ ■ □ ■ □ ■
7 ♙ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖
  a b c d e f g h
Turn: 3, white
Select a piece to move
b8
0: move: b8->c6
1: move: b8->a6
Select a move by entering the corresponding index
0
```

```
1 ♜ ♞ ♝ ♛ ♚ ♝ ■ ♜
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
3 ■ □ ■ □ ■ ♞ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ♙ □ ■ □
6 □ ■ ♘ ■ □ ■ □ ■
7 ♙ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ■ ♗ ♕ ♔ ♗ ♘ ♖
  a b c d e f g h
Turn: 4, black
Select a piece to move
f3
0: move: f3->g1
1: move: f3->h4
2: move: f3->d4
3: move: f3->g5
4: move: f3->e5
Select a move by entering the corresponding index
4
1 ♜ ♞ ♝ ♛ ♚ ♝ ■ ♜
2 ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
3 ■ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ♞ □ ■ □
6 □ ■ ♘ ■ □ ■ □ ■
7 ♙ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ■ ♗ ♕ ♔ ♗ ♘ ♖
  a b c d e f g h
Turn: 5, white
Select a piece to move
```

```
 $ python3 main.py chess minimax2 random
...
1 ♜ □ ♝ ♛ ♚ ♝ ■ ♜
2 ♟ ■ ♟ ♟ □ ♟ □ ♞
3 ■ □ ♗ □ ■ □ ■ □
4 ♙ ■ □ ♕ ♟ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 16, black
move: f1->c4
1 ♜ □ ♝ ♛ ♚ □ ■ ♜
2 ♟ ■ ♟ ♟ □ ♟ □ ♞
3 ■ □ ♗ □ ■ □ ■ □
4 ♙ ■ ♝ ♕ ♟ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 17, white
move: c3->a1
1 ♗ □ ♝ ♛ ♚ □ ■ ♜
2 ♟ ■ ♟ ♟ □ ♟ □ ♞
3 ■ □ ■ □ ■ □ ■ □
4 ♙ ■ ♝ ♕ ♟ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 18, black
```

```
Turn: 18, black
move: c2->c3
1 ♗ □ ♝ ♛ ♚ □ ■ ♜
2 ♟ ■ □ ♟ □ ♟ □ ♞
3 ■ □ ♟ □ ■ □ ■ □
4 ♙ ■ ♝ ♕ ♟ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 19, white
move: d4->e4
1 ♗ □ ♝ ♛ ♚ □ ■ ♜
2 ♟ ■ □ ♟ □ ♟ □ ♞
3 ■ □ ♟ □ ■ □ ■ □
4 ♙ ■ ♝ ■ ♕ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 20, black
move: a2->a3
1 ♗ □ ♝ ♛ ♚ □ ■ ♜
2 □ ■ □ ♟ □ ♟ □ ♞
3 ♟ □ ♟ □ ■ □ ■ □
4 ♙ ■ ♝ ■ ♕ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 21, white
move: e4->e1
1 ♗ □ ♝ ♛ ♕ □ ■ ♜
2 □ ■ □ ♟ □ ♟ □ ♞
3 ♟ □ ♟ □ ■ □ ■ □
4 ♙ ■ ♝ ■ □ ■ ♟ ■
5 ■ □ ■ □ ■ □ ■ □
6 □ ■ □ ■ ♙ ■ □ ■
7 ■ ♙ ♙ ♙ ■ ♙ ♙ ♙
8 ♖ ♘ ♗ ■ ♔ ■ ♘ ♖
  a b c d e f g h
Turn: 22, black
white has won
```

```
 $python3 main.py checkers minimax2 random
...
1 ■ □ ■ □ ■ □ ■ □
2 □ ■ □ ■ □ ⚇ □ ■
3 ⚈ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ⚆ □ ■ □ ■ □
6 □ ■ □ ■ □ ⚈ □ ■
7 ■ □ ■ □ ⚇ □ ⚆ □
8 □ ■ □ ⚆ □ ⚆ □ ⚆
  a b c d e f g h
Turn: 45, white
jump move: e7->g5, capturing [f6]
1 ■ □ ■ □ ■ □ ■ □
2 □ ■ □ ■ □ ⚇ □ ■
3 ⚈ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ⚆ □ ■ □ ⚇ □
6 □ ■ □ ■ □ ■ □ ■
7 ■ □ ■ □ ■ □ ⚆ □
8 □ ■ □ ⚆ □ ⚆ □ ⚆
  a b c d e f g h
Turn: 46, black
basic move: a3->b4
1 ■ □ ■ □ ■ □ ■ □
2 □ ■ □ ■ □ ⚇ □ ■
3 ■ □ ■ □ ■ □ ■ □
4 □ ⚈ □ ■ □ ■ □ ■
5 ■ □ ⚆ □ ■ □ ⚇ □
6 □ ■ □ ■ □ ■ □ ■
7 ■ □ ■ □ ■ □ ⚆ □
8 □ ■ □ ⚆ □ ⚆ □ ⚆
  a b c d e f g h
Turn: 47, white
jump move: c5->a3, capturing [b4]
1 ■ □ ■ □ ■ □ ■ □
2 □ ■ □ ■ □ ⚇ □ ■
3 ⚆ □ ■ □ ■ □ ■ □
4 □ ■ □ ■ □ ■ □ ■
5 ■ □ ■ □ ■ □ ⚇ □
6 □ ■ □ ■ □ ■ □ ■
7 ■ □ ■ □ ■ □ ⚆ □
8 □ ■ □ ⚆ □ ⚆ □ ⚆
  a b c d e f g h
Turn: 48, black
white has won
```