Presentation - Positional analysis of chess games

Jake Moss

May 3, 2021

Difficulties and issues

- Pawn capture on 8th rank
- Timezones and my ignorance of them
- Performance
- KDE plots and axes

Some pawn capture heat maps show a capture on the 8th rank (furthest row away from the starting position). According to the FIDE

> *3.7e) When a pawn reaches the rank furthest from its starting position it must be exchanged as part of the same move on the same square for a new queen, rook, bishop or knight of the same colour. The player's choice is not restricted to pieces that have been captured previously. This exchange of a pawn for another piece is called 'promotion' and the effect of the new piece is immediate. [?]*

Thus a pawn can never be captured on the 8th rank.

This bug likely occurs due to how the detection of captures and handling of positions works. While promotions are accounted for there is a specific edge case that persists.

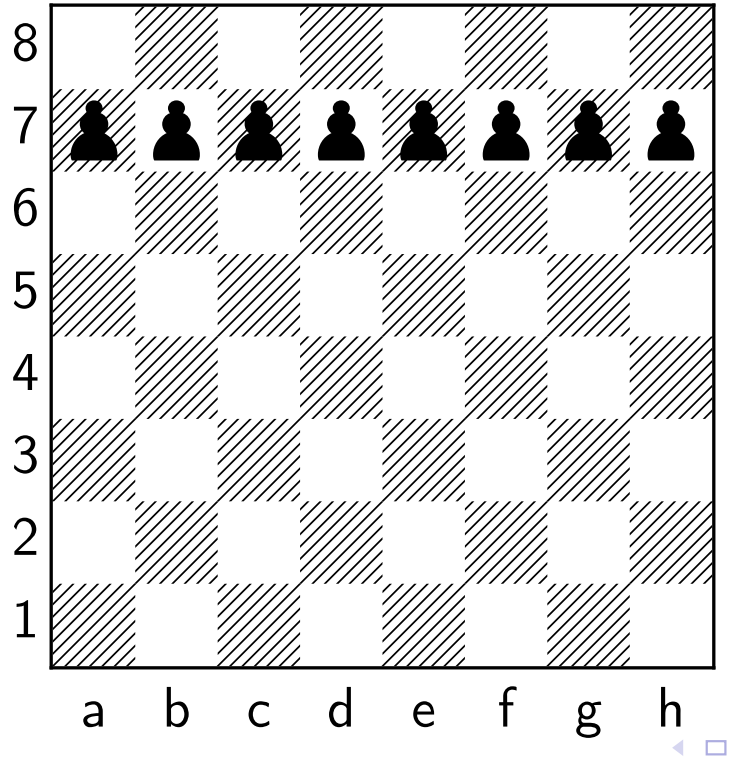
```
def piece_delta(board: chess.Board, count: int,
↪   piece_count: Dict[int, int],
                colour: bool) -> Tuple[int, int, int]:
    piece_position = (0, 0, 0)
    for key, value in piece_count.items():
        current_count = bin(board.pieces_mask(key,
        ↪   colour)).count('1')
        if current_count < value: # Detects lost based
            on previous state
                        piece_position = (key,
        ↪   uci_to_1d_array_index(board.peek().uci()),
        ↪   count)
            piece_count[key] = current_count # Modify by
                object-reference
                            break
```

```
            elif current_count > value: # Accounts for
                promotion
                                piece_count[key] = current_count
                # Modify by object-reference
                                piece_count[chess.PAWN] =
                bin(board.pieces_mask(chess.PAWN,
                colour)).count('1')  # Account for pawn
                count change
                                break
    return piece_position # piece id, position, count
```

This code block works by making mask of the current board state. This returns a boolean bitboard consisting only of the piece requested. For example,

```
board.pieces_mask(chess.PAWN, chess.BLACK)
```

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0



This is then counted and used as the number of current pieces of the piece type present. As this method only keeps track of the number of pieces on the board at once and relies on the previous state to detect a capture it can be easily broken by a change in the piece count other than a capture. For example promotions, as promotions exchange a pawn in favour of another piece type the piece change is not negative. Although this is accounted for there is still a specific edge case which is not caught.

After a pawn promotes, if it is immediately captured the lost piece is still attributed as a pawn. I was unable to squash this bug. In 1918, Russia switched from the Julian calendar to the Gregorian calendar. In the switch the dates from 1st–13th of February [?]. In doing so breaking any naive date comparison implementation from before the switch to after. As Tom Scott put it "What you learn after dealing with time zones, is that what you do is put away from code and you don't try and write anything to deal with this. You look at the

people who have been there before you. You look at the first people, the people who have dealt with this before, the people who have built the spagetti code, and you thank them very much for making it open source.". Rather than dealing with time zones and calendar changes, the `pd.to_datetime()` method and `pd.DateTime` class were employed to correctly handle dates. The database used for demonstration here is the `2000 Standard (all ratings)` FICS Games Database. Coming in at `134.63MB` it is the smallest of all the years, and the only one capable of being analysed due to RAM limitation. It has $3,502,985$ lines and approximately $170,000$ games. The entire database is `17GB`, $452,107,755$ lines, and an unknown number of games.

Processing this subset of the database takes approximately `12min` and `15GB` of RAM, it is a `12th` the size of the largest PGN file and `130x` smaller than years 1999 to 2020.

Initially processing speed was a huge concern, taking `17sec` to load `1000` games was unacceptable. This was optimised down

to `0.6sec` through smarter garbage collection and vectorisation of the `game` objects and data frames.
Unfortunately this implementation is not $O(n)$ and does not scale.
The largest down fall of this program is the hard dependency on `python-chess`, while an amazing feature full library, it is biggest source of possible optimisation. Note it is **not** the slowest component, `pandas dataframes` are notoriously slow.
One such optimisation would be a custom game parse that doesn't check move validity (this is not required as it is fair to assume all games follow the rules) in a very high level compiled language such as Haskell, Rust, or C++.
Although the this program is dependent on `python-chess` it is not dependent on any specific competent of the library that would be unreasonable to port to custom library. This is because it attempts to avoid custom objects and instead favours builtin types. This does add some complexity however it was believed to be the best option.

Optimisation and profiling were conducted through the use of `cProfile` and `snakeviz` to see the time taken by each function call. Originally a second KDE plot was produced to provided a visually appealing histogram variant. However as the density calcinations where handled in matplotlibs back-end there was no clean way to standardise the axes. This lead to misleading plots where although everything looked nice, no conclusion could be drawn from these plots. On solution was to set the `y-max` to 1, while this was an easy fix it produced equivalently unreadable plots due to scaling.