Planning Document II

Project Scotch

James Parkington

There has never been a more exciting time to be a chess enthusiast. In recent years, more and more prominent chess thinkers, grandmasters, and commentators have taken to platforms like YouTube and Twitch to stream their live games, talk about interesting positions, and even create experiments with the many bots being developed and released to bend our notions of how the game works. Streams where prominent players like Hikaru Nakamura, Daniel Naroditzky, and even top-ranked player Magnus Carlsen provide detailed descriptions of their techniques attract similar viewership to some of the top gamers in the world.

That huge increase in readily-available chess content presents individuals from all skill levels with the opportunity to improve their play, but it can be challenging to find commentary on a particular position without knowing exactly where to look.

My vision is to meet that challenge with a free online chess tool that would provide users with immediate access to top-level commentary on any position you enter into the tool. After supplying a board orientation, the tool would then crawl its database of YouTube videos, Twitch VODs, and any other free source of video-based chess material to return a queue of relevant embedded videos. This way, users would see how a professional would continue from their position and learn why.

However, rather than supplying content on just a specific position, it would be great if the tool could proactively try to understand how a player arrived at that position, so that the analysis can be even more contextual.

Project Description

To build this program, it'll be to have a performant way to not only store thousands of sequences of chess positions, but to also render them quickly on a chess board.

Fortunately, all of today's major chess-playing platforms offer something called a PGN game export for every public game played on their system. Later on in this proposal, I explain what the PGN format is and why it's useful for this project.

In short, with this project I'll be attempting to build a Python program that:

1. Parses PGN files and breaks them down into each "move" in a game from start to completion
2. Converts each of those moves into a performant data type that contains the state of the 64-square board (*my first thought is a 64-bit string, but there will be some discovery for this step)*
3. Store each bit string in a sequential way for easy retrieval
4. Render each bit string on a chess board that can be navigated left and right, ideally be graphical interface or user input
5. Loops through each of those bit strings and compares them to a database of exemplary games to try to detect if a known opening pattern has been followed
6. Based on how close the sequence of bit strings matches exemplary games, recommend analysis content to the user

Classes/Objects

Position

The **`Position`** class represents a specific chess position, including the current state of the board, move history, and player turn.

It provides methods to apply moves, check the validity of a position, and access or modify the position's attributes. This class is essential for the project as it enables the representation and manipulation of chess positions, which are then used to find and display relevant commentary from professional games.

```python
class Position:
    """
    Attributes:
        board_state (str):   A 64-bit string representing the current state of the chess board.
        move_history (list): A list or stack storing the sequence of moves made in the game.
        player_turn (str):   A string indicating which player's turn it is ('white' or 'black').

    Methods:
        apply_move(move):     Updates the board state, move history, and player turn accordingly.
        is_valid():           Checks if the current position is valid according to the rules of chess.
        board_to_bitstring(): Converts the board state to a 64-bit string representation.
    """

    def __init__(self, board_state, move_history, player_turn):
            self.board_state  = board_state
            self.move_history = move_history
            self.player_turn  = player_turn

    # Accessors
    def get_board_state(self):
        return self.board_state

    def get_move_history(self):
        return self.move_history

    def get_player_turn(self):
        return self.player_turn

    # Mutators
    def set_board_state(self, board_state):
        self.board_state = board_state

    def set_move_history(self, move_history):
        self.move_history = move_history

    def set_player_turn(self, player_turn):
        self.player_turn = player_turn

    # Other Methods
    def apply_move(self, move):
        """
        Applies a given move to the current position and updates the board state,
        move history, and player turn accordingly.
        """
        pass

    def is_valid(self):
        """
        Checks if the current position is valid according to the rules of chess.
        """
        pass

    def board_to_bitstring(self):
        """
        Converts the board state to a 64-bit string representation.
        """
        pass
```

PGNParser

The **PGNParser** class is responsible for parsing PGN (Portable Game Notation) files, which are the standard format for representing chess games. This class is necessary for the project because it enables the chess game analysis tool to read and interpret PGN files and extract relevant information such as the board state, move history, and game metadata.

By converting PGN files into a sequence of **Position** objects, the **PGNParser** allows the analysis tool to work with chess positions in a structured and organized way, making it easier to process, analyze, and compare games.

```
class PGNParser:
    """
    Attributes:
        pgn (str): The file path of the PGN file to parse or a string containing the PGN data.

    Methods:
        parse_pgn():           Reads the PGN file or string and generates a list of Position objects for each game in
the file.
        parse_metadata(pgn):   Extracts and returns metadata (e.g., event, site, date, players) from a given PGN string.
        pgn_to_positions(pgn): Converts a given PGN string into a sequence of Position objects, each representing a
distinct position in the game.
    """

    def __init__(self, pgn):
        self.pgn = pgn

    # Accessors
    def get_pgn(self):
        return self.pgn

    # Mutators
    def set_pgn(self, pgn):
        self.pgn = pgn

    # Other Methods
    def parse_pgn(self):
        """
        Reads the PGN file or string and generates a list of Position objects for each game in the file or string.
        """
        pass

    def parse_metadata(self, pgn):
        """
        Extracts and returns metadata (e.g., event, site, date, players) from a given PGN string.
        """
        pass

    def pgn_to_positions(self, pgn):
        """
        Converts a given PGN string into a sequence of Position objects, each representing a distinct position in the
game.
        """
        pass
```

Matcher

Matcher is a class designed for comparing user-inputted chess positions with a database of existing games in order to provide the user with valuable insights, such as potential openings, plausible continuations, and other relevant information.

This class plays the most important role in the chess game analysis tool by enabling users to find similar games and learn from them, ultimately improving their understanding of the game and enhancing their decision-making abilities on the chessboard. It will achieve this by efficiently searching and matching positions within its database and returning the best matching games based on a similarity score calculated from the board states and move histories of the positions being compared.

```
class Matcher:
    """
    Attributes:
        position_database (dict or DataFrame): A data structure storing existing games and their corresponding board states for efficient searching and matching.

    Methods:
        add_position(position):                         Adds a given Position object to the position_database.
        find_best_matches(user_position, num_matches): Compares a user-inputted position with positions in the position_database and returns the num_matches best matching games and their relevant information (e.g., opening names, number of moves, plausible continuations).
        find_partial_matches(user_position):           Searches the position_database for positions with similar board states and move histories to the user-inputted position.
        compute_similarity(position1, position2):      Calculates a similarity score between two positions based on their board states and move histories.
    """

    def __init__(self, position_database):
        self.position_database = position_database

    # Accessors
    def get_position_database(self):
        return self.position_database

    # Mutators
    def set_position_database(self, position_database):
        self.position_database = position_database

    def add_position(self, position):
        """
        Adds a given Position object to the position_database.
        """
        pass

    def find_best_matches(self, user_position, num_matches):
        """
        Compares a user-inputted position with positions in the position_database
        and returns the num_matches best matching games and their relevant information
        (e.g., opening names, number of moves, plausible continuations).
        """
        pass

    def find_partial_matches(self, user_position):
        """
        Searches the position_database for positions with similar board states and
        move histories to the user-inputted position.
        """
        pass

    def compute_similarity(self, position1, position2):
        """
        Calculates a similarity score between two positions based on their board
        states and move histories.
        """
        pass
```

File I/O

I plan to organize and store the data using the Parquet file format. This approach is especially beneficial when processing a large number of chess games and their moves, and I can leverage the benefits of Parquet while working within the Python ecosystem using the **`pandas`** library.

The primary benefits of using the Parquet format for processing numerous chess moves across many games include:

1. **Efficient storage**: Parquet's columnar storage and data compression techniques help reduce the storage space required for data, allowing more chess games to be stored within a limited storage capacity.
2. **Improved performance**: Columnar storage enables faster query execution, as only the necessary columns are read from the disk. This is particularly important when analyzing a large number of chess games, as it minimizes the time spent on I/O operations.
3. **Schema evolution**: Parquet supports schema evolution, which facilitates the adding, removing, or modification of columns in a dataset without the need to rewrite the entire file. This is particularly useful for feature development.

With **`pandas`**, I'll read the data from Parquet files into DataFrames for analysis, manipulation, and matching. This choice of I/O ensures that the tool can handle large volumes of chess games without compromising on performance, storage efficiency, or adaptability.

Basic Programming Skills

The following programming skills from the course are some of the many that will be essential to implementing this project:

1. **Control structures**: Conditionals (if-elif-else) and loops (for, while) will be required to control the flow of the program, process data, and iterate over various data structures (e.g., lists, dictionaries, DataFrames).
2. **Decomposed functions & methods**: Defining functions and methods will be necessary for organizing code, encapsulating functionality, and promoting code reusability. The custom classes (**`Position`**, **`PGNParser`**, **`Matcher`**) will require clear, effective methods for processing and interacting with the data.
3. **"Ask forgiveness" error handling**: Implementing error handling using **`try–except`** blocks will ensure that the program can gracefully handle unexpected situations, such as invalid input data, missing files, or incorrect PGN strings.
4. **File I/O**: Reading and writing data from and to files is essential for processing chess game data. As discussed earlier, we'll use the **`pandas`** library and Parquet file format for efficient file I/O.

5. **Classes & objects**: Creating custom classes (**`Position`**, **`PGNParser`**, **`Matcher`**) and working with objects will be crucial for organizing and managing the data and functionality of the chess game analysis tool.
6. **Importing libraries**: Importing and using external libraries, such as **`pandas`**, **`lumpy`**, and potentially **`python-chess`**, will be essential for handling data manipulation, file I/O, and chess-specific functionality.

Testing Plan

To ensure the chess game analysis tool is functioning correctly and efficiently, it is essential to develop a comprehensive testing plan. Here are some types of tests and edge cases to consider:

1. **Unit**: Create unit tests for each method and function within the custom classes (**`Position`**, **`PGNParser`**, **`Matcher`**) to validate that they are working correctly in isolation. This includes testing the accessors, mutators, and other methods for expected behavior and results.

2. **Integration**: Develop tests that focus on the interaction between different components of the tool, such as the integration between **`PGNParser`** and **`Position`** or **`Matcher`** and **`Position`**. Verify that the components work correctly together and produce the expected output.

3. **Functionality**: Simulate real-world scenarios, such as loading a PGN file, processing it, comparing user-inputted positions, and displaying the results. Ensure the tool meets the project requirements and performs as expected.

4. **Performance**: Measure the efficiency and speed of the tool when handling large datasets, such as processing thousands of chess games or comparing positions in a massive database. Optimize the tool to ensure it can handle large-scale operations.

5. **Edge cases**: Identify and test edge cases to ensure the tool can handle unexpected or extreme situations. Some to consider:
    - Empty or corrupted PGN files
    - Invalid PGN strings (e.g. missing or incorrect move notation, incomplete game data)
    - Incorrect file paths or inaccessible files
    - Unusual or non-standard chess positions (e.g. positions that violate the rules of chess)
    - User-inputted positions with no matching or similar positions in the database

Additional Background Information

PGN stands for Portable Game Notation and is a standard format for representing chess games. A PGN file contains a series of moves and other game data, such as the names of the players, the date of the game, and the result of the game. PGN files can be used to replay and analyze chess games.

```
[Event "Tilburg Interpolis"]
[Site "Tilburg NED"]
[Date "1985.10.13"]
[Round "1"]
[White "Kasparov, Garry"]
[Black "Sosonko, Gennadi"]
[Result "1-0"]

1.e4 e6 2.d4 d5 3.Nc3 Bb4 4.e5 c5 5.a3 Bxc3+ 6.bxc3 Qc7 7.Qg4 f5 8.Qg3 Ne7
9.Ne2 O-O 10.h4 cxd4 11.cxd4 Qxc2 12.h5 Bd7 13.h6 g6 14.Bg5 Nbc6 15.Rc1 Qa4 16.Qh4 Rae8
17.Rh3 Rf7 18.Rhc3 Nc8 19.Bf6 Nb6 20.Nf4 Rxf6 21.Qxf6 Re7 22.Nxg6 hxg6 23.Rg3 Be8 24.h7+
1-0
```

This is an example of the standard PGN format used to record chess games. Each file contains the following metadata at the start:

- The name of the event
- the date it was played
- the players' names
- result of the game

The moves of the game are then recorded in sequential notation, where each move consists of the piece moved and the square it is moved to. In this format, moves are numbered and any capture is signified with an **x** marker. Handling this type of notation in a way that returns an accurate board state will be the brunt of the project. As you can see, each move consists of a turn for both black and white, which specifies which piece arrived at the square. You do not receive any context about the rest of the board. Therefore accurately presenting the board state after each move requires some knowledge of how each piece can legally take another.

Lingering Questions

Necessary for MVP

- Are there existing, clean, accessible PGN games from an open-source database that I can retrieve via API to start building my own database with?
- What's the best way to create a read/write database within the confines of Jupyter Notebook that stores the results of these conversions? Ultimately, I'd like to take this a step further than a file and use dataframes, so future users can write SQL queries off of my data.
- What's the most simple, but inclusive format I should use for the notation that includes all 64 squares and the remaining pieces?
- Ultimately, I'd like for the program to find the longest string of matches against the full database of games for both move number and board state. (e.g. Moves 11 to 18 from your game matched this other famous game from 1992—here's more about it). I'm not sure what the most performant way is to cut through all of that data.

Nice-to-Haves

- Explore OCR possibilities for video content
    - Ideally there's a tool that can recognize the shapes of a board and its pieces on-screen, which I can then render in the same notation for the embedding idea

Resources

Game Databases

Fortunately, there are plenty of open-source databases available for retrieving PGN games, in order to build out this database. In fact, instead of building my own, if the conversion algorithm is strong enough, I might be able to do the conversion in-place by querying one of these sources in an efficient way repeatedly.

1. **The Chess Games Database** - This is a free online database that provides PGN files for over 500,000 games, including games from some of the top tournaments and players.
2. **The Million Base** - The Million Base is a collection of over 2 million PGN games, which can be downloaded for free from several sources online.
3. **Lichess** - Lichess is a free online chess platform that maintains a database of millions of PGN games, which can be accessed and downloaded by their users.

Python Resources

1. **`chess`**: This Python library is not essential for the project, but I'd like to review its documentation to see how other programmers have handled chess positions, moves, and game rules. It may be useful in processing sequences within PGN files.
    - https://python-chess.readthedocs.io/en/latest/
2. **`pandas`**: I'll use this library to handle I/O operations with parquet files, enabling me to efficiently store and access the growing position database.
    - https://pandas.pydata.org/pandas-docs/stable/index.html
3. **`parquet`**: The parquet file format will be crucial, because it offers efficient storage and fast querying capabilities, which will be necessary when dealing with a large number of chess positions.
    - https://arrow.apache.org/docs/python/parquet.html
4. **`DataFrames`**: Part of the **`pandas`** library, these will help us efficiently organize and manipulate the position data, making it easier to perform matching and analysis tasks. If I find these difficult to manipulate, I may use dictionaries instead.
    - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html