Project Light Blue

Armaghan Behlum **(TF)**: armaghan23@gmail.com
Phillip Huang: philliphuang@college.harvard.edu
Christopher Chen: cchen02@college.harvard.edu
Javier Cuan-Martinez: cuanmartinez@college.harvard.edu
Collin Styring: collinstyring@college.harvard.edu

Purpose Overview

We seek to create a move-recommender for Chess using Python. A user can input each move in a game of Chess using the standardized algebraic chess notation, and the program will return a recommended move for the given configuration of the board. Move recommendations will be based on recorded game history from past games played by winners and competitors in the World Chess Championship.

Name inspired by IBM Deep Blue.

Final Technical Specification

General Overview

Light Blue's primary functionality will take a Chess move input in Standard Algebraic Notation (SAN) from the user signifying the move that was just made on a real-life board. The move input alters a virtual simulation of the current board. The resulting configuration of the board is matched in a data structure, which then provides a recommended move based on previously stored information placed into the data structure. If no information exists for a given configuration, the user will be prompted to make a random move and input the random move for the next turn.

We implement this data structure as graph consisting of nodes and edges. Each node represents a possible configuration of the pieces on a chess board, while each edge connecting two nodes represents the move to change the board from one configuration to the other. Each edge will have a given "weight." The edge from a particular node that has the highest "weight" compared to the other edges from that node will be the move recommended to the user. Recommendations will consist of displaying the recommended move to the user in SAN and also displaying the board configuration resulting from taking the recommended move in a semi-graphical format constructed using text-based characters. Move details about the graph module can be found below.

The graph will be constructed by importing recorded game history from players who have participated in the World Chess Championship. This game history is provided in the form of a Portable Game Notation (PGN) format, a standard in the Chess world for recording games. PGN files can consist of multiple games. We will parse files consisting of hundreds of games, typically representing the full careers of Chess players, into separate moves and the resulting configurations from those moves. We will use each move and the configuration before and after the move to build the nodes and edges of the graph. This way, we ensure every move made from a given node is legal and (hopefully) a particularly good move.

The Graph Module and Weighting System

BASIC PROCEDURE

I. Each instance the recommender is run, the user's move and the configurations before and after the move are fed into the graph module.

II. The module will search for the pre-move configuration in the existing graph. The first search will always be successful since all games begin with the same starting position, and all the previous moves and configurations in the game will be added in real time to the graph.

III. The module will then check the node if the user-input move already exists as an edge.

    1. If it does exist, the module will follow that edge to the destination node (which must match the post-move configuration passed into the module because Chess is deterministic) and return the highest-weighted edge of that node. This is the recommended move
    2. If it does not exist, the module searches for the post-move configuration in the graph.
        a) If the post-move configuration exists, an edge representing the move input will be made connecting the pre-move node to the post-move node. The highest

weighted edge from the post-move node is then returned as the recommended move.

b) If the post-move configuration does not exist in the graph, a new node is created, with an edge representing the move input connecting the pre-move node to the newly created node. The user is informed that no data exists for the configuration and is prompted to make a random move.

WEIGHTING SYSTEMS (EACH A DIFFERENT GRAPH MODULE)

1. The most basic implementation of the graph will weight each edge simply by the number of times it has been visited. Essentially, the most “popular” moves for a given configuration will be recommended to the user. Each time an edge is visited, we will increment the edge weight by one. This approach is the most simple to implement and will be used to test the rest of the application. However, it does not differentiate between “good” and “bad” moves.
2. The second implementation will weight each move based on the Elo Rating of the player making the move. In addition to the input move and the respective pre-move and post-move configurations, the Elo rating of the player making the move will be fed into the graph module. The module will use the Elo rating as a multiplier of how much to increment the weight of the move. Exactly values determining how potent this multiplier will be have yet to be determined, and will be optimized through testing and real-world experimentation. Using this module, users will input their own Elo rating and the Elo rating of their opponent, allowing their game to be automatically incorporated into the graph.
3. The third implementation is to weight moves by who wins or loses (or ties) the game in question. When importing past data, a char value indicating if the move was made by the winning player will be passed in along with the move and its respective configurations. This will act as a multiplier for the weight in a similar fashion to the Elo rating (exact values to be determined).
   - Unfortunately, when users input moves, the winner of the game is uncertain. To accommodate for this, we will keep a list of all the moves made in the game, and once the game is completed, we will automatically detect which player won the match and backtrack through the path made in the graph and retrospectively edit each of the weights based on whether the winner or loser made them. This is a more advanced feature and will only be implemented given ample time at the end. If we run out of time, user moves will be weighted as a tie.

INTERFACE

- **Recommend** - General functionality of the graph module: takes in a move input in SAN, a pre-move configuration, a post-move configuration. Outputs a recommended move.
  - The Elo weighted module will additionally take in the Elo rating of the player who makes the move.
  - When importing World Championship data, the win/loss weighted module will also take in a char value indicating if the move was made by the winner ('w' for winner, 'l' for loser, 't' for tie). When the user plays the game, as the winner is still uncertain, a 'u' will be passed instead. When 'u' is received, Recommend will still trace the graph, add new nodes, and recommend moves, but will make no change to the weight of each edge. If the edge did not exist beforehand, an arbitrary, typical value will be used (in case the user ends the program before completing the game, so our graph remains usable). See Backtrack function of the main module below.
- **Reset** - Clears the graph.
- **Save** - Saves the current graph via file I/O.
- **Load** - Loads the specified pre-built graph from a file.

The Chess Module by Will McGugan

Light Blue heavily utilizes the Python Chess Module written by Will McGugan and edited by Armaghan Behlum (our TF). The module parses PGN files in SAN format and simulates board configurations. We will use the datatypes defined in this module to represent moves and board configurations. The Chess Module also has abilities to detect check, checkmate, and tie results, as well as the function to check if a given move is legal by generating a list of all possible legal moves. We will use these functions to implement automatic backtracking for the win/loss module, and to test Light Blue's overall functionality.

INTERFACE

- **Chess.Game()** - Contains information to describe a chess game. Including the board, all the moves and external information, such as player info.
  - **setup** - initializes a new game
  - **reset** - clears existing game
  - **initial_board** - returns the initial board configuration

    - **import_pgn** - imports a PGN file into the created game and begins parsing
    - **export_pgn** - creates a PGN based on the result of a game
    - **move** - creates a virtual board move based on the next move recorded in the imported PGN file
- **check_legal** - checks to see in a given move is legal
- **get_legal_moves** - checks to see if there are any legal moves
- **check_result** - returns status of board: ONGOING, MATE, STALEMATE

The Main Module

The main module serves to run the command line interface for the user, as well as facilitate all the value passing between the Chess Module and the Graph Module. The main module will allow the user to switch between using the Popularity weighting system, the Elo weighting system, and the win/loss rating system.

INTERFACE

- **Move** - Parses user move input and alters the simulation of the board using the Chess Module. Then, passes the necessary inputs to the Recommend function of the graph module, and prints the output to the user.
    - When using the win/loss weighting system, also keeps a list of the moves and configurations made throughout the game for use in Backtrack. Upon detecting a win, loss, or tie, goes through all the moves to edit the char value to indicate which player won the game.
- **Backtrack** - Win/loss weighting module only. Takes in the list of moves and configurations made during the game, with each move also containing a char value indicating if the winner made the move. Runs the Recommend function for each item in the list, effectively “finishing the job.” The end result resembles the case in which the user’s game was merely imported normally along with all the historical data.
- **Import** - Parses through a bulk PGN file using the Chess Module and feeds each game move-by-move into the move function to construct a graph.
- **Switch** - Switches to the specified weighting system.
- **Reset** - Calls the reset function of the graph module.
- **Load** - Calls the load function of the graph module.

- **Save** - Calls the save function of the graph module.

Timeline

WEEK 1: APRIL 12 - APRIL 18

- Finish Python Learning
- Finish conceptual planning
- Finish setting up GitHub, Python, Asana, SublimeText, etc.
- Prototype parsing functionality

WEEK 2: APRIL 19 - 25

- Implement Popularity Graph Module
- Implement Main Module
- Finish parsing functionality
- Have working functionality for Popularity weighting system
- Prototype Elo weighting system
- Prototype Win/Loss weighting system

WEEK 3: APRIL 26 - MAY 1

- Implement Elo weighting system
- Implement Win/Loss weighting system
- Extension: Combination weighting system using Popularity, Elo, and Win/Loss
- Extension: Simple GUI
- Develop Demo Video
- Write Documentation

Progress Report

Thus far, we feel like we have a solid conceptual grasp on what each part of the program needs to do. We recognize the amount of work needed to actually implement these ideas, and anticipate any possible obstacles that may appear. A few parts of the code have begun to materialize, though much of our team is still reviewing the ins and outs of Python. Attached are snippets of code currently being written. Asana is set up, a few confusions with GitHub are being ironed out (will consult TF soon for advice), and the Python Chess Module is being reviewed in depth for any more useful functions we may utilize. Each of us feels increasingly confident about the project and its progress.