Open Chess Software Architecture

William Stenberg

2019-06-10
Version 0.1

Introduction

This document outlines the software architecture for the web-based application *Open Chess* that offers an interactive chess board through a React Application served with Flask. Using the application (playing a game of chess through mouse interaction), the user is encouraged to explore the early-game strategies of the game of chess. The backend of the application (written in Flask, with a SQLAlchemy database) internally evaluates positions and assigns them scores. The evaluation process is performed by Stockfish, a chess computer engine. Stockfish is used to assign scores to board positions fed to it, representing a move's soundness in the game. The scores for the initial and resulting board states in a move form the weight of the move, taking into account other avaliable moves in the initial position.

The application defines *book moves*, which are moves in certain positions of the board that are known to be good. These can be shown as arrows on the graphical board representation in the frontend, and express their weights through their visual representation.

In the future, the viewing of book moves will be a feature for the player to manage their known chess moves with the application, through exploring variations of chess openings and adding new moves as they learn.

The app is intended for use as a training tool, prompting the player to make good moves in pre-determined board positions. This is meant to be gamified through assigning the a *rank* to each move, a value that varies with the player's prompted actions. The player evaluation could be expanded to define an "experience point levelling" type of gamification system, or a chess-style "Elo ranking" type system.

The Open Chess source code is on `https://github.com/WilliamStenberg/open-chess`, and is Free Software.

Database layout

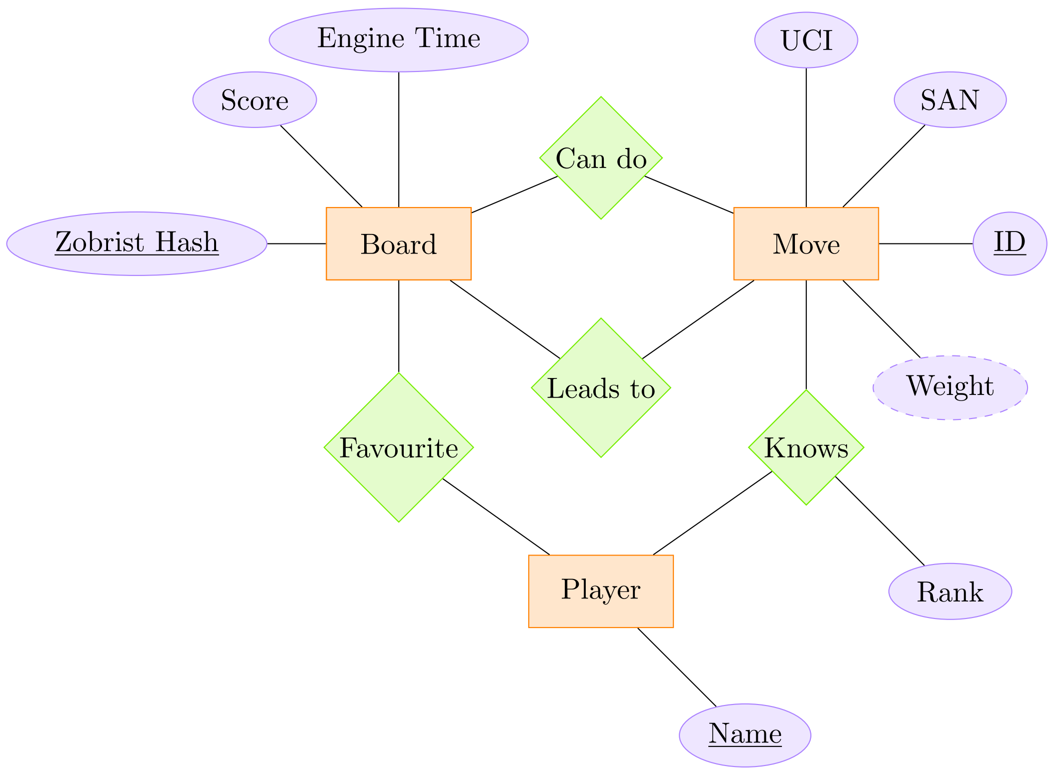

Figure 1: Entity-Relationship (ER) diagram for *open-chess* database

The entity-relationship diagram in Figure 1 outlines the structure of the database, implemented in SQLAlchemy in the backend of *open-chess*. The method of drawing the diagram comes from `https://www.guitex.org/home/images/ArsTeXnica/AT015/drawing-ER-diagrams-with-TikZ.pdf`. Below, each of the elements are explained in terms of their intended use in the application:

- **Board**: Representing a chess board, i.e. a configuration of pieces. The pieces themselves are not modelled in the database, but during a session an active *python-chess* Board object will hold such information.
- **Zobrist Hash**: A chess board is uniquely defined by its Zobrist hash (TODO: cite Zobrist theory), which is computed by taking into account piece positions, castling rights etc, resulting in a hash string.
- **Score**: A board's score is its numeric evaluation by a chess engine - who's winning. A score is computed from a player's perspective to a

custom format, but this can be converted to a numeric value, using negative values for black's advantage and positive for white's.

- **Engine Time**: Since the score of a position is computed by an engine (Stockfish) and this score is more precise the longer the engine was allowed to think, the time that the engine has been run is kept so that scores can be improved. When simply loading in boards from an external source, this is set to zero indicating that any score cannot be relied upon.
- **Can do**: From a board there is a set of legal moves that can be made from that position. Since the board state implicitly convey whose turn it is to move, a board will only be connected to moves for a certain side (white or black).
- **Move**: Representing a legal move in a game of chess.
- **UCI**: The UCI representation of the move, comprised of the square the moved piece starts and lands on, concatenated to a four-letter string, e.g. "e2e4", "b1c3".
- **SAN**: The SAN representation of the move, which is more commonly used by humans when talking about moves, e.g. "e4", "Nc3". The SAN representation implicitly holds board state information.
- **ID**: Since two moves may have the same SAN and UCI representations, but pertain to different board states, a unique ID is required. Typically a numerical index.
- **Weight**: A derived property conveying how good a move is for the side making it, computed from the scores of the initial and resulting boards. A negative value would indicate a bad move, while a positive move is better since it improves the board score for the move-making side.
- **Leads to**: The relationship between a move and the board it leads to. While a board *can do* many moves, a move only *leads to* one board.
- **Player**: A registered user of the system, created when they enter the application interface for the first time.
- **Name**: A player is uniquely identified by a string.
- **Favourite**: A player may record board positions to be remembered, to quickly place the interface's board in this state.
- **Knows**: A player is said to know a move if they have performed it in the interface. When a player knows a move, the interface's *game*

mode expects the player to be able to perform it if the session's current board *can do* it.

- **Rank**: Each time the game mode prompts the player for a move in a board where the player knows at least one move, the player's action will influence the rank of one or several moves: If the player performs a known move, the rank is improved. If the player performs another move (not known by the player), the rank of all the known moves for that board will decrease. It is thus a measure of the player's familiarity the the known move.