8.12.2021

Documentation

4-Player-Chess Predictor

SWEPM GROUP-2 THE-4

Structure

The main program consists of 6 files: CHESS_INTERFACE_TKINTER.py, Chess_Algorithm.py, database.py, chessdatabase, chess_img.jpg and chess_piece_knight.ico.

get_data.py is used to create the database file and is therefore not a main part of the program, since the chessdatabase file should be present in any situation. In case it gets lost, ffa.txt and solo.txt need to be present in the current working directory to make use of this file and recreate the chessdatabase file.

get_data.py:
get_data.py doesn't have any defined class, since the database itself is an object, this file only contains functions to fetch data from solo.txt and ffa.txt and the procedure to create and insert to the database.

1. Import "re", database.py and "datetime"
2. All necessary functions are defined
3. Loop to open txt file, make use of all previously defined functions, close txt file
4. Initialize Database
5. Loop to insert previously retrieved data into database
6. Commit and close database

database.py:
Defines the database object, this file is used by **Chess_Algorithm.**py and **get_data.py**

1. Import "sqlite" and "os"
2. Class to define the database object with all necessary functions to operate with the database and file location

chessdatabase file:
The database file itself, created with database.py with library sqlite

CHESS_INTERFACE_TKINTER.py:
Serves as the main file:

1. Import tkinter, PIL (ImageTk, Image), Chess_Algorithm.py
2. 2 Classes that define the first and second page, the second page includes user input (moves) and calling for the algorithm.

Chess_Algorithm.py:
Defines the Algorithm to make a percentile prediction with entered moves, uses database.py

1. Import database.py, pandas, time, random, operator
2. Preparing database connection and data
3. Defines Algorithm function that returns predictions in form of a list with tuples.

Test_Chess_Algorithm.py, Test_CHESS_INTERFACE_TKINTER.py, Test_database.py:
All 3 test files contain tests written with the help of the library "pytest", they can be found in the "test" folder in the program directory.
The program is tested as much as possible, since programs with interfaces made with such as "tkinter" are nearly impossible to test completely, not every function in the interface is tested, but test examples or suggestions are included.
CHESS_INTERFACE_TKINTER.py is as much tested as possible.
Test_Chess_Algorithm.py containing tests for the Algorithm, Chess_Algorithm.py.
Test_CHESS_INTERFACE_TKINTER.py containing tests for the Interface, CHESS_INTERFACE_TKINTER.py.
Test_database.py containing tests for get_data.py and database.py.

Code Documentation

get_data.py:

This file is used to create the database file from ffa.txt and solo.txt with the library sqlite, it is using the database object from database.py.
Every function goes through the txt files line by line (since every line represents a game) and fetches needed data with the help of the library "re" (regular expression).
After the data has been retrieved, data is inserted into the database object, which is creating the "chessdatabase" file, it only is created if the file isn't present in the current working directory.

get_game_id():
For each line in ffa.txt and solo.txt, it searches game id's with regular expression, "[0-9]{6,10}" represents the game id's pattern, where [0-9] stands for a number between 0 and 9 and {6,10} for a set between 6 and 10 numbers.
Try and except block because there are some lines in the .txt file that don't contain any game data or game id's, when such a line is found, the regular expression function "re.search" throws an error, indicating there is no such pattern in this line and therefore no game id.
Every found Game ID gets appended to list1, the function returns list1.

get_player_id():
This function searches for the player names for each line, every line contains 4 player names.
Similar to get_game_id(), get_player_id() loops through the txt files line by line, searching with regular expression patterns for the player names.
Since there are some areas in each line that are like the player name's pattern, first the function includes an area after the words: Red, Blue, Yellow, Green (For example: red_1).
In this area and again with a regular expression pattern, player names can be retrieved (For example red_2).
Red_3 just serves to remove unnecessary symbols like "\".
Every other colour works the same way, try and except block because, similar to to get_game_id(), some lines in the txt files don't contain any game information, so the pattern cant be found and therefore the function "re.search" throws an error, should an error be caught, pass.
Every name found is appended to list2, where list2 serves as a temporary list, which is appended to list3 after a full iteration.
In the end, list3 contains a list of lists where each list contains 4 player names and represents 1 Game.

get_move_ids():
Function to get every move from each Player each game, “listcomplete” is returned at the end, it contains “roundlist” and “gamelist”, where “roundlists” is representing a list of moves every round and “gamelist” representing a list of “roundlists” for each game, list1 and list2 are temporary.

First the function goes through every line in the txt files and retrieves every move a player made with regular expression, splitting those moves into a list. The Iteration after splitting removes all empty items and unnecessary symbols with the help of regular expression and appends the moves to list1, list1 is appended to list2 and gets deleted.
List2 contains now a list of games that contains every round with moves for each game.

Going though list2, inserting “[99.]” at the end of every “gamelist” “[99.]” represents the end of a “gamelist”.
Iteration: for every “gamelist” in list2 and for every move in in “gamelist”, search for round numbers with the help of regular expression pattern.
If a round number was found (try and except because function “re.search” throws error if pattern isn’t found) and “roundlist” isn’t empty and length of “roundlist” is 4, append the “roundlist” to the “gamelist” and reset “roundlist”.
The check if a roundlist is 4 servers the purpose that only moves from rounds where all 4 players were present are appended, since there is not really a way to determine which player has left the game at any point.
If no number was found (basically it’s a move), append move to “roundlist”.
When the “gamelist” isn’t empty, append it to “listcomplete”, at the end return “listcomplete”.

get_piece_ids(moves):
This function searches for pieces which were moved, it takes the “listcomplete” from the function get_move_ids(): as argument, it is needed to retrieve every piece from each move.
List5 represents the list of all pieces.
Variable “q” is used to check if more than one item was appended for each iteration, if this happens, the last appended move gets removed.
This case happens if a player moves to a starting position of an enemy’s pawn, the value of this move gets saved into the dictionary where the same value is already saved, the second value that is appended is always the wrong one, so its easily removed with “pop()” function.

The first iteration represents all games, here the dictionary’s to numerate the Pawns are created.
Second iteration represents all rounds each game, a new variable “l” gets initialized and is used as a “counter” to keep track of moves made by which player.

The third iteration goes through every move in a “roundlist” and checks:
If move is only a single character of: R, T, S or T#, append the same character to list5, since those are game events.
If no game event is present, check if move was made with a pawn, the pattern of the regular expression searches for all moves that aren’t Pawns: 0-0 for short castle, 0-0-0 for long castle, [A-Z] for capitalised characters.
Try and except block because the function “re.search” throws an error if pattern isn’t found.

If pattern is found, check for: B, N, Q, K, R, 0-0, 0-0-0.
Those symbols and characters are all representing a piece or a short and long castle move, corresponding piece to the character gets appended.

If pattern is not found the piece is for sure a pawn (now the pawn needs to be numerated), get the starting field and the end field from move and remove unnecessary symbols.
Check if starting field is a starting position of a pawn for each player, if yes, append pawn number (left to right) to list5 and insert value(ending field) to the corresponding key.
This check goes for every player(l) for every pawn every round, l+1 to keep track which players turn it is currently.
If currently moving pawn was already moved ones before, check which players turn it is and go through every key in the dictionary, checking if the current starting field is the same as the ending field the turn before.
If yes the corresponding pawn gets appended to list5 and value of the current key in the iteration is replaced with the current ending field.

get_time(moves):
This function find every time stamp for each move and calculates how long a move actually took to make with the library “datetime”.
It takes the “listcomplete” from get_move_ids(): to get every timestamp from the corresponding move.

First loop:
For every line in txt file, get an enclosed area where the time stamps are present, than find every time stamp and append it to timelist with the help of regular expression patterns.

Second loop:
Temp and temp2 serve as temporary lists to save timestamps for each move each game.
Variable v serves as counter, iterating through every “roundlist” each game to count how many moves were made, than get exactly that amount of timestamps (from beginning to end) and append them to temp2, i+1 to get into the next game.
Append temp to temp2 so temp contains a list of lists of timestamps for each move each game.

Third loop:
Here happens the actual calculation, the variables time1 and time2 serve as the current time stamps were time2 is the current stamp and time1 the stamp before.
Timestamps are now transformed into suitable formats for the library datetime with the help of regular expression patterns.
At last the actual calculation happens with dateTimeA – dateTimeB, the result gets appended to “timelistout”.
At the beginning of each iteration, 0 gets appended to “timelistout”, since there is no starting time for each match, the first move can’t be calculated.

Insertion of data into the database:
Starting with a loop in range of 2 to switch between the 2 txt files solo.txt and ffa.txt.

Open the file in only read mode and execute the functions explained above in following order:

- **game_ids = get_game_id()** – print information how many game id's were found, set pointer in txt file to the beginning.
- **player_ids = get_player_id()** – print information how many player id's were found, set pointer in txt file to the beginning.
- **move_ids = get_move_ids()** – print how many moves were found, u used as a counter, u+1 for every move, set pointer in txt file to the beginning.
- **pieces = get_pieces_ids(move_ids)** – print how many pieces were found, set pointer in txt file to the beginning
- **time = get_time(move_ids)** – print how many "calculated time stamps" are found, close txt file

After all data needed is retrieved, create database object and create table, if a database file named "chessdatabase" is already in the working directory, initialize a connection.

The loops for insertion into the database:
For every game, every round, every move, insert:
game id, player id, move id, piece, round, time

z serves as a position counter for the piece and time list.
Commit database operations and close database connection.

<u>database.py:</u>

class Database:
Using "sqlite" library, Initialize the database object, where the variable location gets the current working directory and the name of the database file.
Initialize connection with the database, using location variable.

close():
Close connection to the database, usually called when the database isn't needed anymore.

insert():
Function to insert data into the database, it accepts 6 arguments, where every argument represent one column for each row.
Arguments are used in the "INSERT INTO" query.

create_table():
Creates a table with following columns for each row:
Chess_ID – A automated numeration for each row in the database
Game_id – integer, represents the current game id
Player_id – string, represents the current player
Move_id – string, represent the current move
Piece_id – string, represents the current piece
Time_id – string, represents the time needed to make current move

display():
Used to display data from the database using a SELECT query.

commit():
Used to commit and write the "chessdatabase" file, usually used if database is not needed anymore, right before closing the connection.

Chess_Algorithm.py

Chess_Algorithm.py interacts with database.py for SQL Querries and with CHESS_INTERFACE_TKINTER.py fetching moves entered by the user and give back predictions based on that.

The code snippets before the function def Algorithm(move) ensures that df_raw, a pandas dataframe, is quarried correctly. Df_raw quarries only the column Move_id and the column Player_id to reduce memory costs (however leading already to over 10 million rows).

The actual algorithm function is checking df_raw row by row and if a entered move matches a row the Player_id is saved in predictions dictionary as key and for the first occurrence the value is one.

If another move of the same player is found, "algorithm" is checking if the key is already in dictionary and if so, increase value by one.

Percentage is then calculated by using total amount of moves found and Dreisatz calculation updating every value in the dictionary relatively to its key (Player_id) in percentage.

This "prediction" is on the one hand very simple and therefore it can be estimated that this prediction approach is very poor. However, it has an advantage in predicting more active players because they have more entries in the historical data and therefore more matches.

The original idea was to use pandas and some prediction model like random forest trees or decision trees. Unfortunately, this was discarded due to the lack of experience, expertise, time and organisation.

CHESS_INTERFACE_TKINTER.py (main_file)

The interface is built using tkinter gui and pil module for importing images.
It is a simple beginner friendly interface comprising of **2 pages** for the user to navigate around.
1.Class FirstPage(tk.frame)
2.Class SecondPage(tk.frame)

1. **Class FirstPage(tk.frame):**

Also known as the homepage.

Widgets(self):
This Pages consists of one widget, the **start button** which navigates user to second page.

StartFunc(self):
This function is called when the start button is clicked.

2. **Class SecondPage(tk.frame)**

Widgets(self):
The widget's function consists of all the basic layouts and widgets which the users see on the screen.
Frame 1: The first frame consists of 2 labels: "Player moves" and "Player prediction"
Frame 2: This is the left major frame where user enters the moves in order to receive a prediction. It consists of an **entry box** and a **submit button.**

EnterFunc(self):
This function is called when the submit button is clicked. Its clears the moves in entry box and calls on the **Algorithm function** from the Algorithm module**.**

Frame 3: This is the right major frame which returns the player prediction in percentage after receiving player moves from the user.
Frame 4: This is the lower frame which consists of 3 buttons made for easy navigation between pages for the user.
A restart, end button and help button which takes user to a documentation page.

HelpFunc(self):
This function is called when the Help button is clicked it moves the user to a new frame containing a documentation on how the application works.
The **EndFunc** and **RestartFunc** are both called when the End and Restart buttons are clicked respectfully, which returns user to first page if they wish to end or restart the game.
After the user clicks the help button and is moved to the documentation page we also have a **Return Button** which returns user to the second page to continue prediction.
ReturnToGameFunc(self): this function is called when the Return button is clicked.

Software Requirements & Design Specification Changes

Design Specification Changes:

In the Design Specification it is often mentioned that the program connects to the Chess.com API but since later in the development process it turned out that the API isn't capable to retrieve data for 4 Player Chess, we decided to get the data directly from Chess.com in form of txt files and create a database file, the program than connects to this file to get data.
This circumstance makes a few functions of the program unnecessary, therefore going through the Design Specification the following was updated:

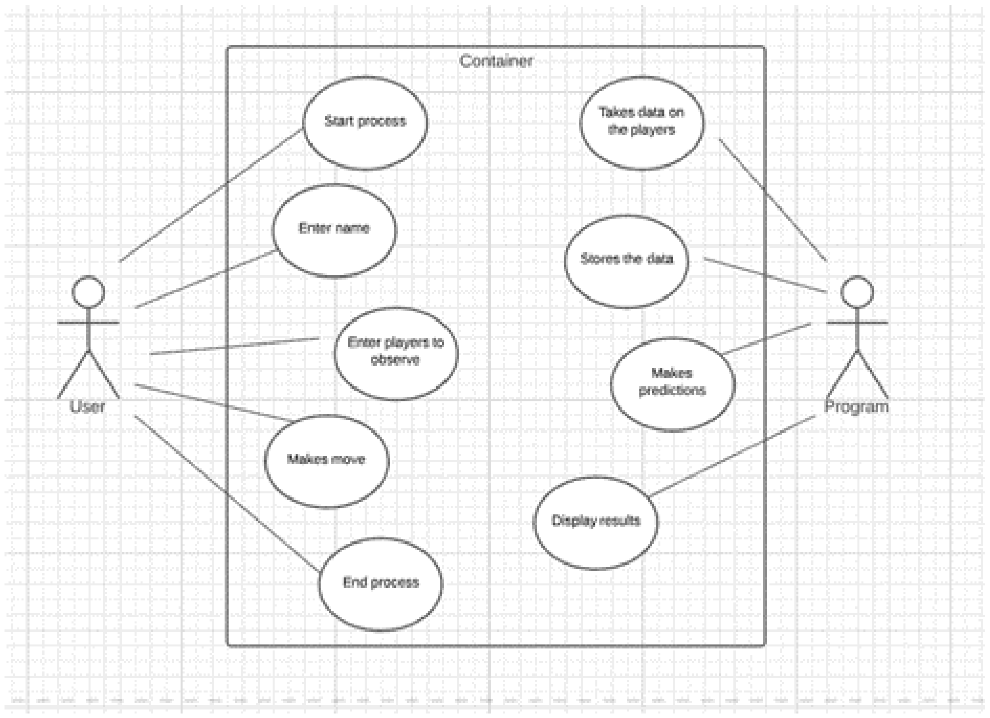
Introduction:

Old:
We are building a program that connects to the API Chess.com. The program provides the client with list of players that is observed. It then predicts which colour each player is. The client can also add players to the list he wishes to observe or wants to predict. Amidst the game it will also show the percentage win each player has. This will help the user understand the play, his opponents and their moves which can help him to improve his game and his understanding.

Updated:
We are building a local program that connects to a previously created database file in the same directory as the program from data given by Chess.com. The user is able to enter current made moves, the program than gives a percentile prediction, which colour which player could be. This will help the user understand the play, his opponents and their moves which can help him to improve his game and his understanding.

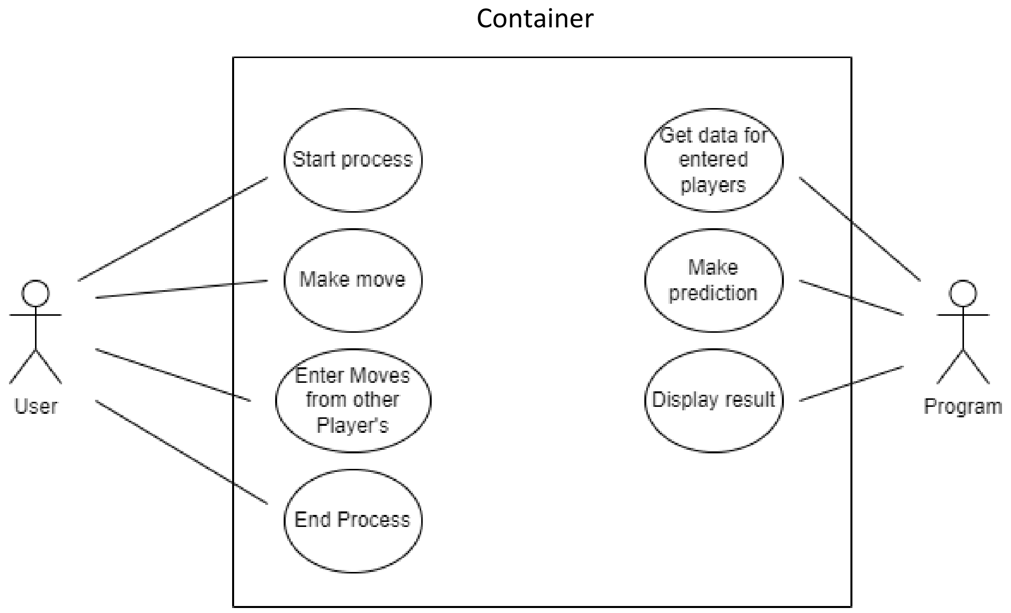
Architectural Design:

Old:
<u>External Perspective:</u>



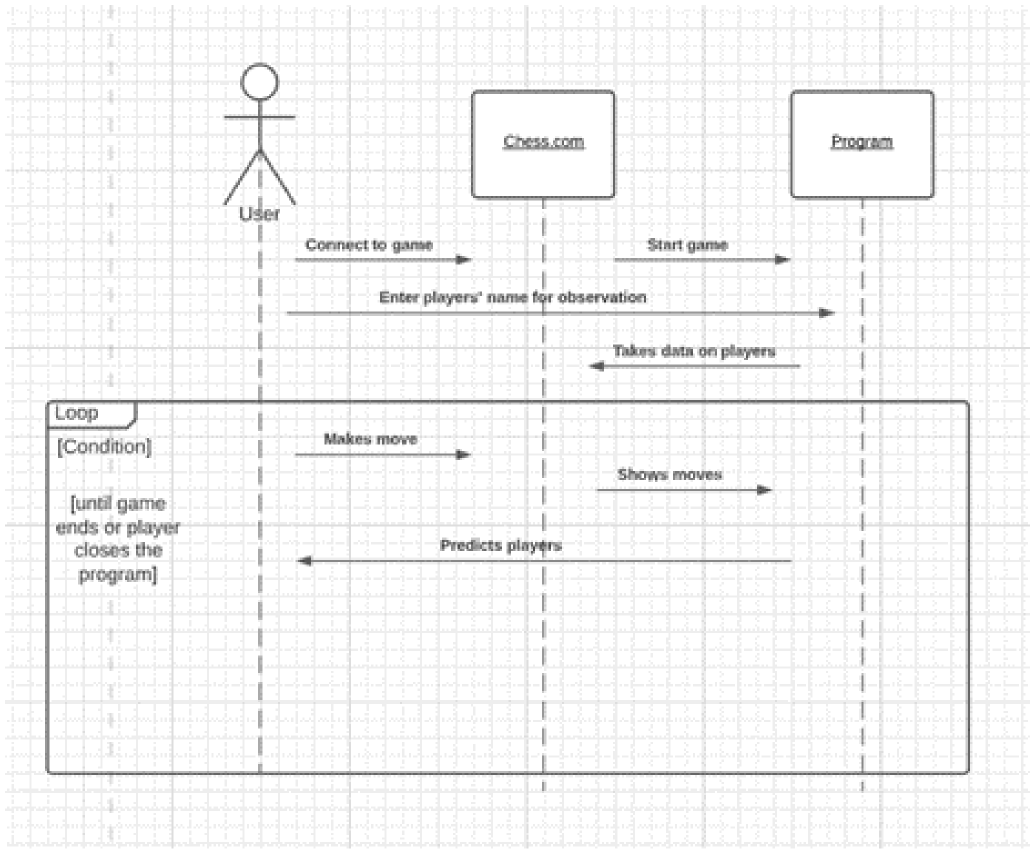
Updated:

External Perspective:



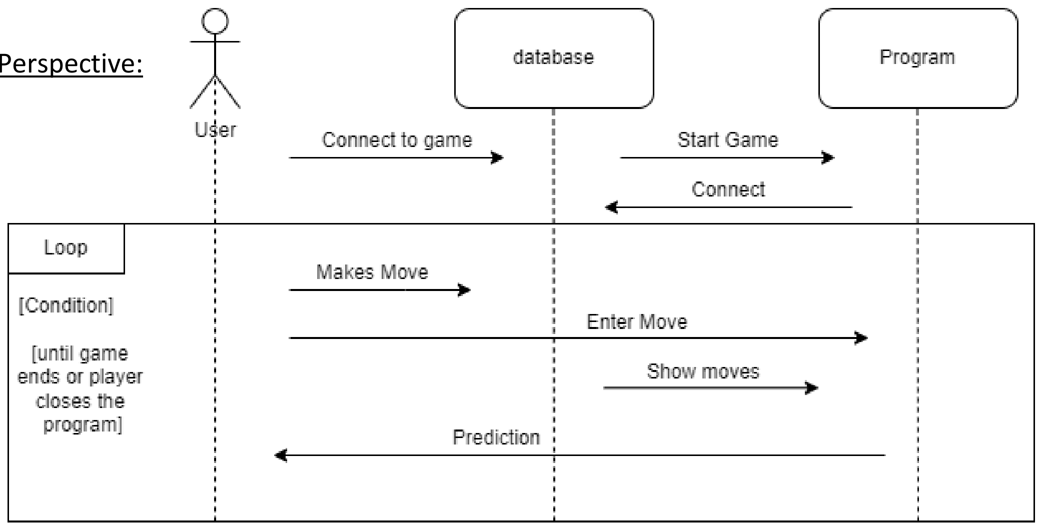
Old:

Interaction Perspective:



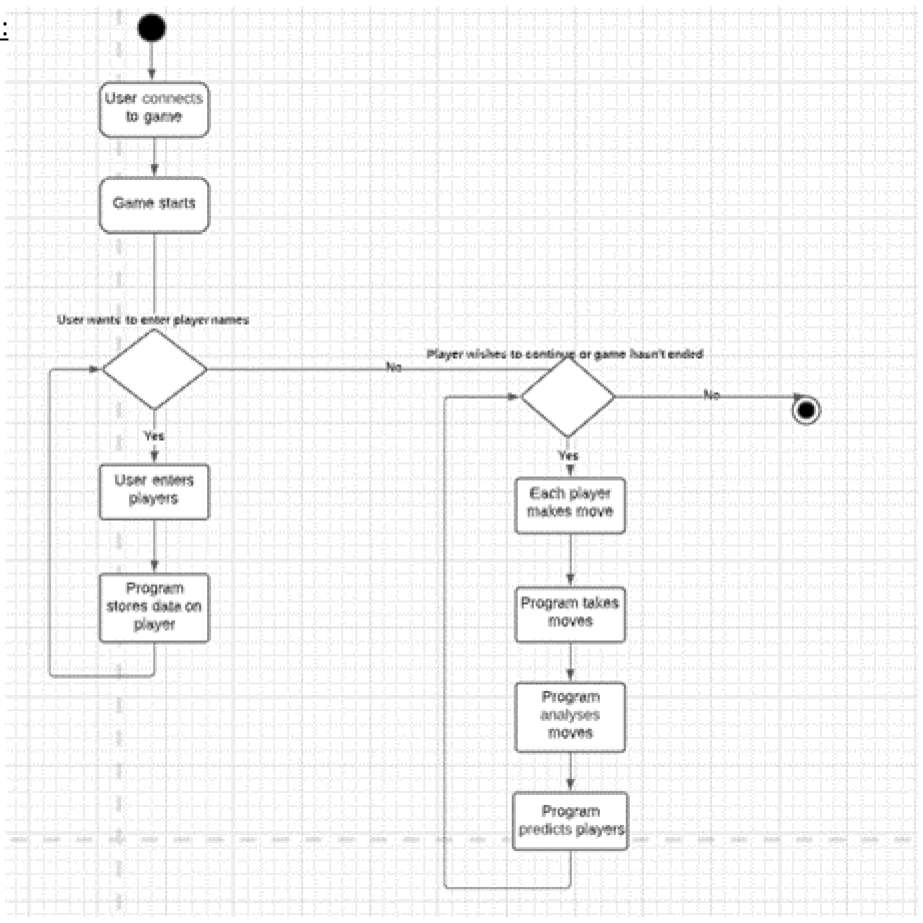
Updated:

<u>Interaction Perspective:</u>

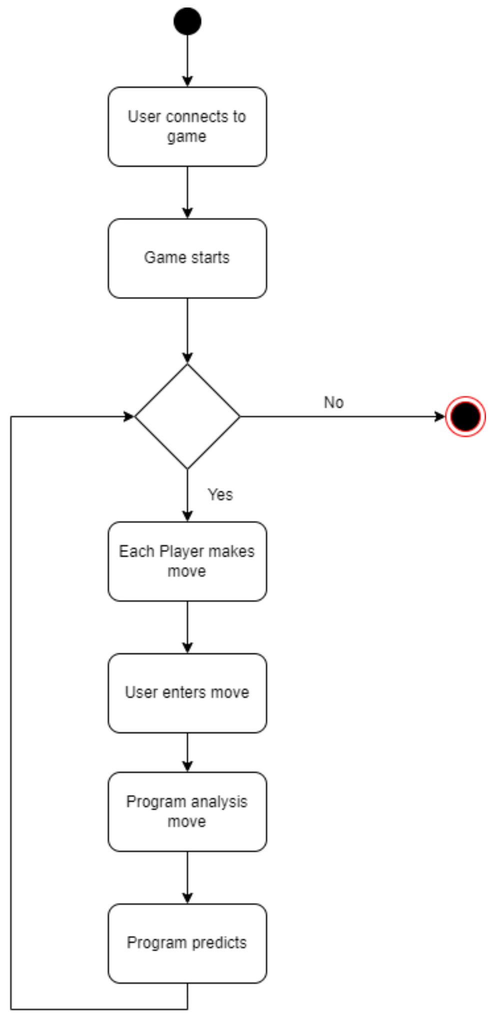


Old:

<u>Behavioural Perspective:</u>



Updated:
<u>Behavioural Perspective:</u>



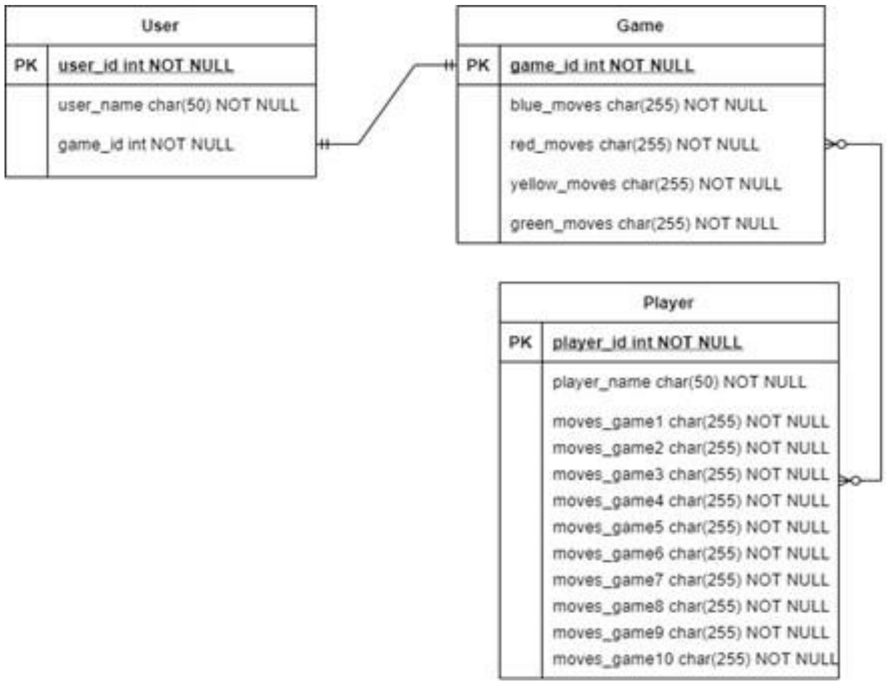
Database Design

Old:
We will use SQL Lite as the Database library. There are many concepts to think through how the database relations are in the end build up for the prediction software. One possible approach is that the program observes every move from every player taken each round and loads it in the database. This pattern is than matched with patterns from players games which were already played and loaded in the database.

Also it is not yet clear how the machine model will look or if a machine model will be used.

Nevertheless final design of database is a matter of change because programming such a prediction software will be a try and error race.

Entity Relationship Diagram:


User
PK
user_id int NOT NULL
user_name char(50) NOT NULL
game_id int NOT NULL
Game
PK
game_id int NOT NULL
blue_moves char(255) NOT NULL
red_moves char(255) NOT NULL
yellow_moves char(255) NOT NULL
green_moves char(255) NOT NULL
Player
PK
player_id int NOT NULL
player_name char(50) NOT NULL
moves_game1 char(255) NOT NULL
moves_game2 char(255) NOT NULL
moves_game3 char(255) NOT NULL
moves_game4 char(255) NOT NULL
moves_game5 char(255) NOT NULL
moves_game6 char(255) NOT NULL
moves_game7 char(255) NOT NULL
moves_game8 char(255) NOT NULL
moves_game9 char(255) NOT NULL
moves_game10 char(255) NOT NULL

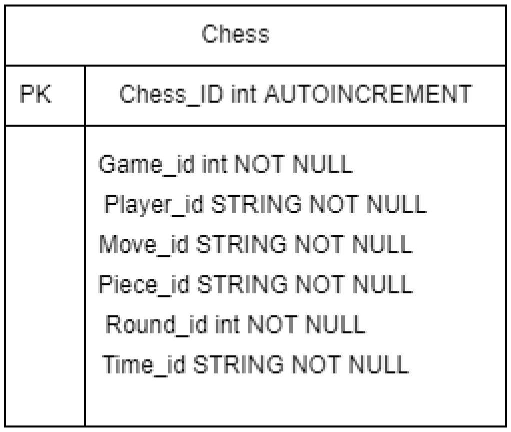
Updated:

We will use SQL Lite as the Database library. There are many concepts to think through how the database relations are in the end build up for the prediction software. One possible approach is that the program observes every move entered by the user and loads it into the prediction algorithm. This pattern is than matched with patterns from players games which were already played and loaded in the database.

Also it is not yet clear how the machine model will look or if a machine model will be used.

Nevertheless final design of database is a matter of change because programming such a prediction software will be a try and error race.

Entity Relationship Diagram:



Components selection, design and interface design

Old:
Chess.com library (connection):

A connection to chess.com is established with the chess.com python library, archive data from each player given by the user will be requested and provided in a json format. This data will then be sent to the Algorithm.

Algorithm (prediction):

Given the users name, and the players he wants to observe, the algorithm compares the anonymous players moves each round from the current game, with the moves from the given players from the past 10 games. The more moves are the same in the same round, the more moves per round that match, the higher the percentile prediction.

For example: 3 rounds in the current match have already passed, if the algorithm finds that move 1 and 3 are matching with one of the past games rounds 1, 2 and 3 there is a 75% prediction.

Interface:

ELEMENTS:

- Information input (user player name, expected opponents player names)
- Algorithm
- Data from chess.com
- Modules

User enters his player name (Information input):

Users logs into system giving their name , the program requests data from chess.com for the current game the user is playing.

User enters players he wants to predict:

When user specifies which player they wish to predict, the system requests past game data of the given players from chess.com and saves it into the database. If data is already saved from player, the system updates the data.
Prediction accuracy increases as players moves increases.
After the specified players move, the system returns the potential predictions in percentage.

Interface consists of 3 parts: “console” log, list of players to observe and the player prediction.

Console Log:

A textual description of what the program is currently doing.

List of players to observe:

A list of players given by the user for prediction. Data is requested for those player to compare them to the saved data.

Player prediction:

The actual percentile prediction of each given player by colour.

Updated:

Chess.com database file:

A connection to a database file created with the python library "sqlite" from txt files, received from Chess.com is established. With "sqlite" querry's data is sent to the Algorithm.

Algorithm (prediction):

Given the move the user enters, the algorithm compares the entered moves to previous games in the database and predicts to which percentile chance the given pattern fits to a specific player. The more moves are the same in the same round, the more per round that match, the higher the percentile prediction.

For example: 3 rounds in the current match have already passed, if the algorithm finds that move 1 and 3 are matching with one of the past games rounds 1, 2 and 3 there is a 75% prediction.

Interface:

ELEMENTS:

- Information input (moves from other players)
- Algorithm output
- Data from chess.com
- Modules

User enters moves from other players:

When user enters the move that was made by another player, the system provides this moves to the algorithm.

The algorithm requests a list of data from the database file containing the moves entered and compares to each player which pattern fits the most and returns a potential percentile prediction for each player fitting.

Prediction accuracy increases as players moves increases.

Interface consists of 2 parts: Field to enter player moves, list of player names with percentile prediction.

Field to enter player moves:
Here the user is able to enter player moves and submit them with submit button below.

List of player names with percentile prediction:
Output of the algorithm, a list with player names and their percentile prediction each is shown.

Prototype plan/outline:

Old:
Start:

When the user opens the program, he needs to enter his player name that he uses on chess.com. The purpose for this is so if it happens that he changes his name on the website or another user uses the program, it is easier accessible. The program searches for the username and saves data into the database, also when the user enters his name again it checks for possible updates on the data. Since the user gave us the freedom if it should be a plugin or a standalone program, we decided to make a standalone program, mainly because it's easier to maintain and to develop.

Player observation:

Next the User is asked to enter the names of players he wishes to observe/predict, the idea behind this is since in tournaments there aren't always the same players and there isn't really a way to know (except for the user) which players are participating, data can be retrieved more easily by searching the archives by player name given by the user.

While observing/predicting:

The program observes each move from each player each round, it also gives the user the information when done, so the user knows exactly what's happening and if the prediction is usable. Also, the user is able to restart the process with the same data already entered or with new data. This restart process should happen every time when the game is over, it gives a better overview to the user and it's easier to get a proper prediction. Console Log: Telling the User what 's currently happening, the purpose here is to keep the user always updated what the program actually does right now, it gives more transparency.

Observed Player List:

List of Players that the Program should observe (Entered by the User). The reason we decided to use this method is so the user can freely decide and also knows which players are actually observed. Again, it gives more transparency and more clarity and also more control.

<u>Player Prediction:</u>

Percentile prediction which colour each Player has, the player is displayed in the predicted colour and with the percentile chance, so the user knows how much chance there actually is if the player has that colour.

Updated:

<u>Start</u>
When the user opens the program and clicks the start button, an input field with a submit button on the left and a list (at the beginning empty) with future predictions and player names with colours on the right is presented.

<u>Player prediction:</u>
If a valid move is entered and submitted with the "submit" button, the algorithm starts to query move lists with player names from the database and returns the player names with the percentile prediction corresponding to it, the interface than displays the names and the predictions with corresponding colours in the right section of the Interface.

<u>While running:</u>
The player needs to enter every single move each round, the Algorithm automatically declares for each move a colour, the first move entered each round is always Red, second Blue, third Yellow, fourth Green. The more rounds are played and therefore more moves are entered, the more precisely is the prediction. Also, the user is able to read a Documentation about the program by clicking on the "Help" button.
Restart button to restart the program representing a new game is starting.
End Button to end the Program.