NTNU
Norwegian University of Science and Technology

TPK4186 - Advanced Tools for Performance Engineering
Spring 2023

Assignment 2: Chess Games

Group 24

Sipan Omar, Morten Husby Sande & Kim-Iver Brevik Blindheimsvik

1.2) Organization of code

Libraries used in our code:

- *numpy*: adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- *matplotlib*: Creates static, animated, and interactive visualizations in Python.
- *re*: Provides regular expression support. Regular expressions are a powerful language for matching text patterns.
- *sys*: This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.
- *graphwiz:*

The python code is divided into eleven different python files. *Main.py* is the file where all of the functions are tested and run. *Game.py is* where the create, get, and set functions for a game are stored. *Reader.py* has functions which writes, reads, and adds information from and to text files. This includes databases and database management. *Plotting.py* uses the *matplotlib.pyplot* library in functions to create graphs and figures to represent different types of data. For example games won, drawn and lost by Stockfish. *Excel.py* writes and reads games from an excel file. *Node.py* contains the class properties for a Node object, as well as supporting functions. *Edge.py* does the same as Node, just for the edges. *Counter.py* counts games won, lost, draws, and counts lengths of games based on color, wins, etc.. It also has the functions for mean values and STDs. *HTMLFile.py* contains the code for the HTML-document (report) created. When creating a document a HTML-file called document.html also automatically gets created. This HTML-file is the report with all the plots and tables specified in the different tasks. *OpeningManagement.py* contains the class of the same name, which looks at the different openings and has functions to retrieve the desired data. Finally *Tree.py* has the Tree-class giving the structure of the encoding of a tree with all of the games, as well as functions to support this endeavor.

2.1) Games

Task 1: Design a data structure to encode games and implement the associated set and get functions. Note this implies managing moves.

The functions for task 1 are all in the game class in *Game.py.* When creating a game the object takes in several parameters such as the event, site, date, round, the player

with black, the player with white, the result, ECO, the opening and variation, the amount of moves, the elo of the black and white player and all the moves in the game. We have also implemented various set and get functions to modify the events, openings and game moves. These get-functions are especially useful since we're later using the Reader_ImportChessDatabase file to create a list (gameList) with all the games from the stockfish file in their own game objects, and then using this gameList in the several plots and tree classes later to analyze the stockfish games.

Task 2: Design functions to import a game from a text file.
This functionality is ensured by the *Reader_ImportChessDatabase(...)* function. When we obtain a list (gameList) of all games in the stockfish file using the function, we can index this list to get the game we want. For example, to get the first game we write gameList[0] and to get the last game (game 2600) we can write gameList[-1].

Task 3: Design functions to export a game from a text file.

For this we have the function, *Reader_ExportGameToFile(game, outputFile, separator)*. The function takes in a game object and writes it to the preferred text file (outputFile) with easy to read formatting. It also uses the *Printer_PrintGame(...)*. The *Printer_PrintGame(...)* uses the get-Functions for all the game variables in the game object to write the variables to the text file on their own line.

Task 4: Design a data structure to manage a database (a list) of games. Extend your reader and your printer so you can manage databases.

We used the game class to manage databases as seen in the Game.py module. Additionally we utilized the *DataBase_ReadChessDatabase* function heavily to retrieve the data from the stockfish-file. In the Reader.py module we have included functions to both import and export game(s).

DataBase_ReadChessDatabase takes an inputFile and the amount of games starting from the first you want to read. The function relies heavily on regular expressions to read the metadata (step 2), store said data in a dictionary (metaDict) and then reads the moves for each game (step 3). The third step also assigns the data to the current game object with functions like *exportMetadataToGame(metaDict)* and *DataBase_ReadMovesInLine(moves, line)*. We simplified the move retrieval process by using regular expressions to "clean up" the string of text, removing the comments in the brackets and [%eval] terms that conflicted with the rest of the code. There is

room for improvement in the function as some brackets seem to slip through, causing some inaccuracies in the results.

In hindsight we probably would have solved the task of importing data from the file a bit differently, should we have reattempted it. Mainly by splitting the function into multiple simpler functions as the debugging eventually became quite tedious in the current format.

Task 5: *Design functions to export and import a game from an Excel spreadsheet.*

All the Excel code is in the Excel.py file. *ImportGameExcel* takes in an excel filename and a gamenumber like ('820.1.529') and prints out a list with two elements. The first element is a dictionary with the metadata, and the second element is a list with the moves. The function uses the openpyxl module to search rows and columns and extract the data written inside the cells.

ExportGameExcel takes in an excel filename and a game in the same format as written above and adds it at the bottom of the excel document. With the *Printer_PrintGameFromDatabase* function you can extract any game from the stockfish database in this format so it can easily be transferred into the excel file.

2.2) Statistics

Task 6: *Design functions to create Word (HTML, LaTex) documents. This document must have a title, some sections, subsections, paragraphs. . . It will include tables and figures (plots of distributions).*

We've chosen to create a HTML-document. The HTMLFile.py contains a Document class that contains several useful functions. The *createFile(gameList)* creates the document, which is a HTML-document called *"document.html"*. It does this by creating the file or by rewriting the contents of the already existing file, then it writes the content to the file using the *createDocument()* function.

CreateDocument() is used for structuring the HTML-document. It makes sure the HTML-document has a head and a body. The *createHead()* function sets the HTML title and the style for tables in the document by using the documentName set in the constructor. The *createBody()* creates the body of the HTML-document using the

contentBody variable created in the constructor. It gradually updates the contentBody string when the different insertion-functions in the file are called.

We have created some general functions except for inserting tables, these are to create small and large heading, to insert a given image, to create a row in a table and to create a paragraph. For example we have a function *createRowInTable()* that contains the rows for both the standard deviation and mean tables, and the tables for stockfish table that contains the wins, draws and losses for stockfish with black and white pieces. This function takes in the three columns you can choose and also a version parameter. The version parameter determines which type of row you want, for example for an std-table and for a stocktable. You can also choose a header in the version parameter that creates the top row of the table.

Task 7: *Design functions to extract the number of games won, drawn and lost by Stockfish in total, then with white pieces, then with black pieces. Design functions to insert a table with these results in your document.*

Reader_CountStockWins takes in the filename and uses regular expressions to find specific keywords and phrases. The keyword in this instance is “Result”. The function then searches for the phrase after this word. If the phrase is “1-0”, then one win is added. If the result is “0-1”, then one loss is added. If none of these are true then there is a draw, and one draw is added to the list. *Reader_CountStockResults* takes this one step further and uses regular expressions to check if stockfish is white or black pieces. This is then used to find how many times stockfish have won with white and black pieces.

From the plotting.py file in the Plots class the *PlotResults* function plots the wins, losses and draws made by stockfish. It takes in an input file and a result type where you can choose between general wins, losses and draws or based on if stockfish was white or black pieces.

The document inserts all these plots using the *insertImage(image, figureText)* function in the HTMLFile class. The images then get added to the body of the document. This function allows you to plot any image specified alongside a text you can determine in the figureText-parameter.

Task 8: *Design functions to plot the proportion of games still on-ongoing after 1, 2, 3, .. moves. Design functions to insert a figure plotting this distribution in your document.*

From the Counter.py file, the *GameLengths* function adds all the lengths of all the games in the database into a list. *PlotOngoning* in plotting.py in the Plots class then creates a dictionary and counts how many times a gamelength occurs and plots it in a graph.

Calculate also the mean and the standard deviation of the number of moves of a game. Design functions to insert a table with these results in your document.

getStandardDeviationMoves uses a simple np.std line to calculate the standard deviation. *getMeanMoves* takes the sum of *GameLengths* and divides it on the length of *GameLengths,* and divides it by 2.

Same questions for games played by Stockfish with white pieces and with black pieces. Note that it may be of interest to plot the three curves on the same figure.

GameLenghtsWhite and *GameLenghtsBlack* does the same as GameLengths but first checks with the *IsStockWhite* function before it adds the game length to the list.

The plotting is the same, and both White and Black gamelengths are shown in the same graph.

Same questions for the games won by Stockfish and those Stockfish lost.

GameLengthsStockfishWin and *GameLengthsStockfishLoss* does the same as GameLengths but first checks with the *IsStockWhite* function and the *game.getResult()* function from the Game class before it adds the game length to the list.

The plotting is the same, and both stockfish wins and stockfish loss gamelengths are shown in the same graph.

Also, using the *createMovesMeanStdTable()* in the HTMLFile class we create a table in the document with the mean and standard deviation for all the ongoing moves in the games where stockfish won, lost, with white and with black pieces. The plots above are also inserted into the document with the already explained *insertImages(...)* function.

2.3) Openings

Task 9: Design a data structure (and all its management functions) to encode a tree such as the one pictured in 2.

To manage our data structure we again used classes and their functionality extensively. The classes for Node, Edge and Tree are all implemented in their own modules. The tree class includes a lot of similar functionality as some of the functions in Node and Edge, as we wanted to include these directly, while still being able to use Node and Edge without relying on the tree class. Functions like: *addEdgeToNode* and *lookForEdgeInList* use the classes' properties to allow the desired functionality.

Task 10: Design functions to parse a database file such as Stockfish 15 64-bit.commented.[2600].pgn.gz and load it into an opening tree.

For this task we use the *DataBase_ReadChessDatabase* function from earlier to generate the list of games from the stockfish-file. From there we used the *buildTree(self, gameList)* function to encode the tree itself. This implements a variety of functions to check whether or not a node or edge already exists as it iterates through the different moves of the 2600 games. These functions: *controlNodeAndEdge, goToNextNode* and *lookForEdgeInNode* uses the objects from the varying classes to

It also has functions to create the root (first node) and leaves (end nodes). These, however, probably would have been better as children of the Node class. Additionally the goToNextNode function transfers the focus from one Node to the next, but there are some issues with the function as it won't always include the potential moves in the Nodes. If we were to attempt this task again a recursive function with some additional classes would have been better. We also faced issues with the duplication of edges, that the function as it stands does not correctly recognize already existing edges and thus creates a list that is of every move in the game, instead of the 5000 or so unique moves (number based off previous code that seemed to work for that functionality, but not others)

Task 11: Design functions to print an opening tree at a given depth, i.e. number of 1/2 moves. The idea here is to get the number of games won by white pieces, by black pieces and the number of games that ended up in a draw in each opening. Design functions to insert your drawing in the report.

NodesAndEdgesToLists() in the tree class takes in the defined nodes which are in a dictionary and strips them of information leaving us with a list of the keys which are numbers: [0,1,2,3,4,5,….]. The defined edges are also formatted in a way that the plotter can handle.

The edges are in lists inside of a list: [[0,1,'fe4'],[1,2,'de5'],.....] where the first value is the startnode, the second value is the end node and the string is the weight of the edge. These two lists are then returned from the function. *plotOneTree()* plots class takes in the two lists and uses the networkx and matplotlib library to present the nodes and the edges in a readable way. The moves are assigned colors based on which turn it is. The graph is stored as a png file in the plots folder for use in the HTML document.

Task 12: The drawing of the previous question is only moderately interesting from a chess point of view: some openings are much more played than others. E.g. the Italian, Spanish and Sicilian defenses on 1. e4. For these openings, we want to go deeper, while on rare openings such as the Bird (1. f4), one or two 1/2 moves suffice.

All the needed functions management functions are in the OpeningManagement.py file in the Opening Management class. The functions for inserting the tables for the openings, and their results with black and white, played more than n times are however located in the HTML-file document.

First *lookAtNOpenings(OpeningDict, n)* function takes in an openingDict created by the *getOpeningCountDict(gameList)* function, and returns a list that contains the names of the openings that's been played more than n times. The *getOpeningCountDict(gameList)* creates an empty dictionary first, and then creates a key with all of the different openings. If an opening is already in the dictionary, the value of the key (opening) gets incremented by one. The *lookAtNOpenings(OpeningDict, n)* then iterates through the dictionary and checks if all of the opening-values are bigger than the specified n and then appends the opening to an empty list if it's bigger. At the end it returns the list with all the openings bigger than N. The *openingWins(gameList, opening, pieces)* gets the results for the chosen openings and for the chosen pieces, white or black. The *getResultForNOpenings(n)* returns the results of black and white pieces for all the openings that's been more than n times in two double lists, one for black results and one for white results. These lists also contain the openings.

For the document part, we've defined three functions in the document, the *createRowOpening(Opening, Colum1, ...), OpeningResultsRows(n) and the createOpeningTable(n)*. The *createRowOpening(Opening, Colum1, ...)* takes in an opening and three columns, and then creates a row that gets added to the contentBody string. The *OpeningResultsRows(n)* iterates through the *getResultForNOpenings(n)* function and then fills two empty lists, one for black rows and one for white rows, using the *createRowOpening(Opening, Colum1, ...)* function with each opening and then the results. The *CreateOpeningTable(n)* function then adds these two lists of white and black rows to the contentBody string by iterating through them and creating a table for each of them. The results are two tables, one for white and one for black, that gets inserted into the document with all the openings that are played more than n times and their results.