University of Victoria

CSC 501

Report (Assignment-0)

Submitted by:

Name	Student ID
ABHISHEK KATHURIA	V00959831
DHRUVRAJ SINGH	V00970352
SIDDHARTH CHADDA	V00947906

Section-1
Data Modelling And Design Decisions

If we think closely then chess can be defined as game made up of sequence of positions of chess pieces. These positions of the chess pieces change in sequential fashion by the players playing the game. Now going by this logic, to maintain an index of all the possible moves in one game we will be required to represent the chess data in to the below 4 dimensions:

Total number of type chess pieces: 6 (Queen, King, Bishop, Knight, Rook and Pawn)
Total number of colours: 2 (Black and White)
Total number of Ranks on the chess board: 8
Total number of Files on the chess board: 8

#Data Dimensionality Reduction Hack:
But instead of representing the chess positions data in the above 4 dimensions we can use below equation to decrease the number of dimensions from 4 into 1 single dimension with a max value of 768.

The total number of possible game positions can be given by: 6x2x8x8 = 768 positions
Now we can maintain a simple 1D index of all the possible chess board positions in the game, thereby saving memory and processing time.

Data Modelling Approach And Design Decisions:

1) **Parsing the PGN Data:**
   For parsing 5the chess data awe will be using the “chess.pgn”, python library. Using this library, we will first load the chess pgn data. The data inside the pgn file is in FEN (Forsyth-Edwards-Notation) format, FEN is a descriptive string listing the location of each piece on the board. For analysis, we want to transform this game format into one that we can process numerically. To implement this, we will use the chess.pgn library to iterate through each move of each game and export its pieces’ board position.

2) **One-Hot Encoding:**
   To further reduce the memory footprint of the data file and save on processing we will use Arrays to first build a “one hot encoded matrix” , to encode the string of the different chess pieces like below:

k	1	0	0	0	0	0	0	0	0	0	0	0
q	0	1	0	0	0	0	0	0	0	0	0	0
b	0	0	1	0	0	0	0	0	0	0	0	0
n	0	0	0	1	0	0	0	0	0	0	0	0
r	0	0	0	0	1	0	0	0	0	0	0	0

p	0	0	0	0	0	1	0	0	0	0	0	0
K	0	0	0	0	0	0	1	0	0	0	0	0
Q	0	0	0	0	0	0	0	1	0	0	0	0
B	0	0	0	0	0	0	0	0	1	0	0	0
N	0	0	0	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	0	0	0	0	1	0
P	0	0	0	0	0	0	0	0	0	0	0	1

3) **Designing the games matrix:**

In a similar fashion to the above we will construct a one hot encoded matrix representation of each of the chess games. To store all this information, we will build an Array matrix of size (768, X). Here 768 refers to the total number of possible positions/ moves that can be made in 1 chess game and “X” here refers to the number of games. For example, if there are 100 games then X= 100. Now by doing this we can essentially stack the 1d position matrix of X games on top of each other. Now since the size of arrays are fixed at initialization, we will populate the above discussed array matrix with zeros and then as we iterate through the games, we will keep on replacing the zeros with chess data.

4) **Saving chess game properties using HashMap/Dictionary:**

Each game of chess not only contains chess piece positional data but we can also use this data to calculate other important data like the probability of a particular chess piece (For example Black Queen) to appear in a particular place (Eg: e4). We can also the data derive answers to questions like: “What is the most used / least used chess piece?”, “What is the most commonly used chess board position?” etc.

Now to save such a type of data, we will use HashMap or Dictionary because now we can directly associate the property data of the game with that particular chess game. We will use the game number as the “key” and the “value” of HashMap to store game properties like chess positional data and count of pieces used etc.

5) Lastly, we will run experiments by using different types of data structures like Numpy Arrays, Lists, HashMaps etc. to store the chess data and then compare their perfomance against each other and our data model.

Section-2

Transforming Raw Data into Insights (Data Visualization)

Plot-1: Most and Least Traversed Chess Position in ~3,26,000 chess games

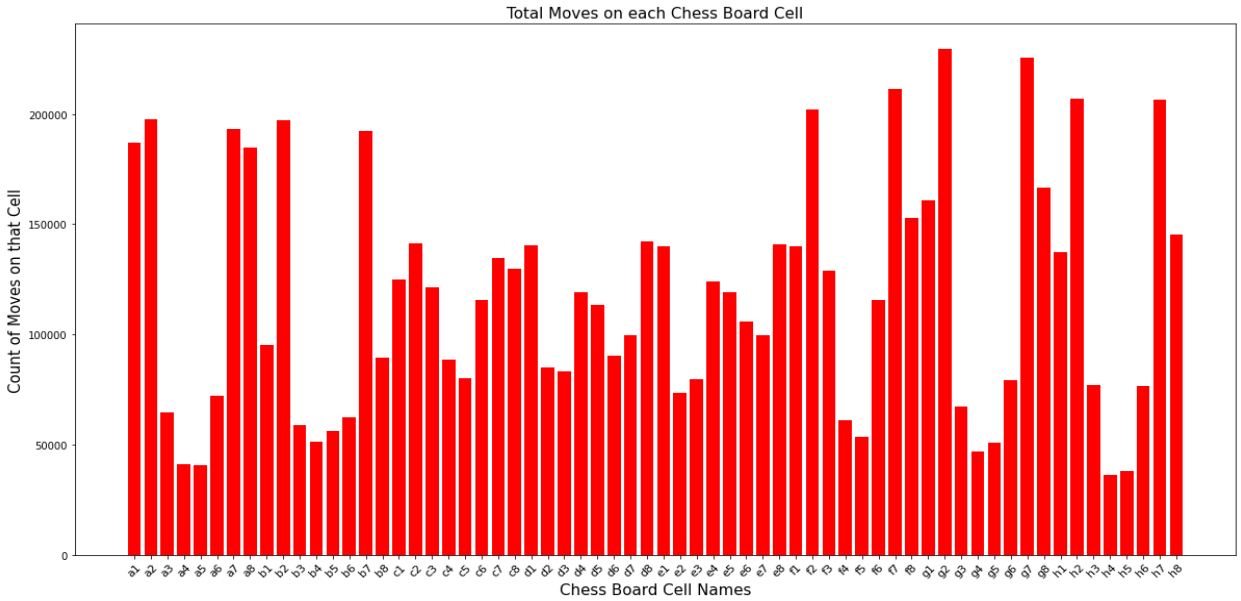

Fig. 1

The bar graph (Fig. 1) visualizes the realtionship between the “Number of Moves“ and “the Chess Positions”. We can observe from the Fig.1 graph that position ‘g2’ had the highest number of counts in terms of Moves.

Question: Which was the most traversed and least traversed chess position throughout the games?

Data Modelling Process:
Converted the data stored in the above discussed chess data properties dictionary in to a pandas dataframe, then we used pandas operations like: groupby, sort and sum to get the count of moves that occurred on that chess board cell. Used the matplotlib library to construct the above Bar plot.

Insights:
By finding the Most and Least traversed chess board positions, we can develop strategies to counter our opponents moves in advance.
From the above graph we can clearly see that, position ‘g2’ was the most commonly used position (i.e., most traversed position) from which a chess piece had made a move in a total of ~3,26,000 chess games. Similarly, we can observe that position ‘h4’ was the least common chess position throughout the games.

Plot-2: Overall Result from the first 5,000- games

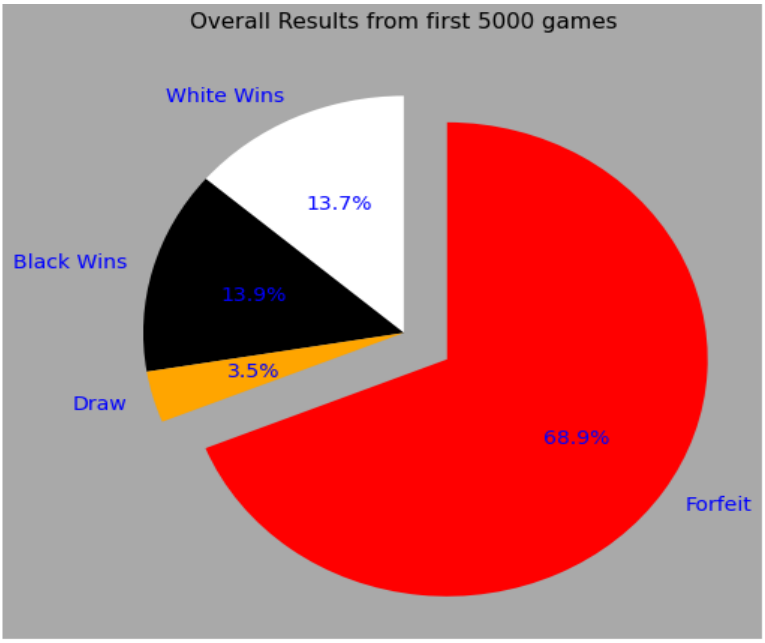

Fig. 2

This pie chart shows the outcomes of first 5000 chess games.

Question: What is the overall result? Which side won how many matches and how many of the matches were forfeited?

Data Modelling Process:
For this visualization, we used pandas to extract the information of just the first 5,000 matches and extrapolated the overall results as a pie chart expressing the percentage of matches won, lost, ended up as a draw and forfeited.

Insights: The pie chart visualization (Fig. 2) represents the overall results from the first 5,000 chess matches. We can clearly see that ~69% of the matches ended up in forfeit. This is a huge proportion when compared to the other categories on the pie chart. White and Black side won approximately the same number of times hence each of them occupied ~14% of the overall proportion while only 3.5% of these matches ended up in a draw.

Plot-3: Count of Moves for each Chess Pieces

	piece	count
0	B	375631
1	K	325900
2	N	368374
3	P	1920837
4	Q	225690
5	R	524441
6	b	387058
7	k	325900
8	n	361230
9	p	1930315
10	q	224959
11	r	523359

Fig. 3

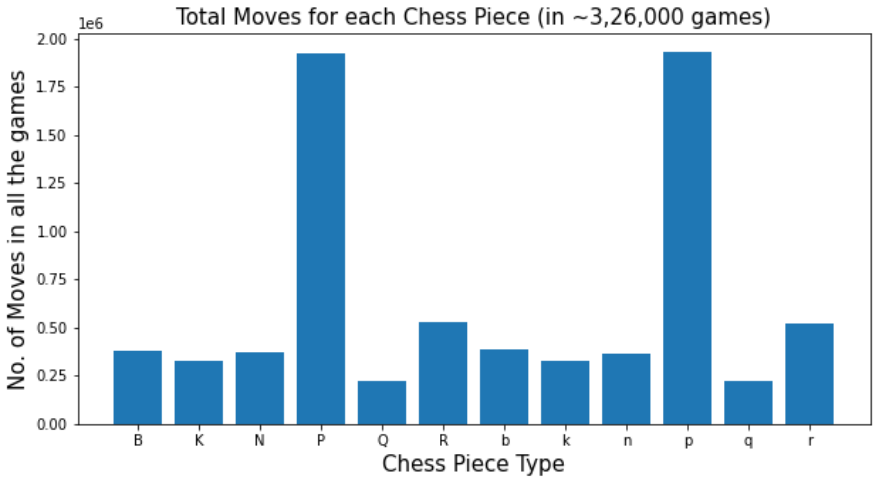

Fig. 4

In this visualization, we have tried to derive the number of moves for each chess piece in a total of ~3,26,000 games.

Questions: Which chess piece has Most and Least number of moves in all the games combined?

Data Modelling Process:
Used pandas operations like: groupby, sort and sum to get the count of moves that occurred using a particular chess piece. Used the matplotlib library to construct the above plot.

Insights:
Every player prefers to use one chess piece more than the other pieces, if we can identify that chess piece then we can decode the playing style of that player.

From Fig. 4, we can observe that the Pawn for both Black and White side had the highest number of moves. Even though we can't observe the subtle difference in moves from Fig. 4, we know from Fig. 3 that the Black Pawn ('p') had 9K additional moves than the White Pawn which makes it the highest moved Chess Piece in the game.

In addition to this, we can also derive the insight that Queen was the least moved chess piece for both Black and White side. If we observe the dataframe in detail, we can see that amongst the two Queens, Black Queen ('q') had the least number of moves in the overall games.

Plot-4: Probability of Chess Piece Positions

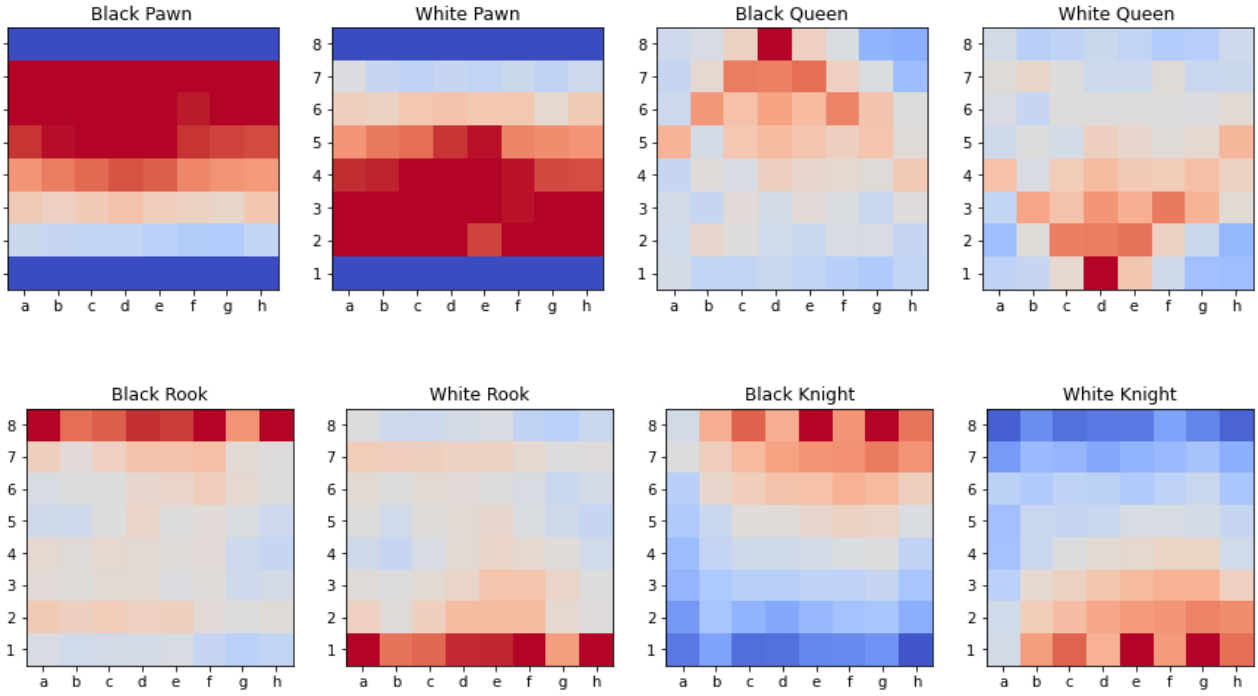

Fig. 5

The heatmaps in Fig. 5 depict the most probable positions for particular chess pieces throughout the games. For this visualization, we have chosen the ‘Pawn’ which is the most commonly used chess piece, ‘Queen’ the least used chess piece, ‘Rook’ with the most symmetrically probable positions and ‘Knight’ with the most unsymmetrically probable positions.

Questions: What are the most probable positions for the chess pieces? What can be the next move for a specific chess piece?

Insights:
If we can predict beforehand where the opponent might be be placing his chess piece, then we can not only decipher our opponent’s strategy but we can also develop counter measures.

Here we have drawn a comparison between the white and black chess pieces of the same type, to understand the probability positions of these chess pieces. As we can observe from Fig. 5, we are able to identify the most probable positions for these chess pieces by looking at the heatmaps. Different shades of red represent the positions the chess piece would likely take. Darker the shade of red, more probable the position. Conversely, darker the shade of blue, least probable the position.

Section-3

Scalability

Processor - Intel i5 11th generation
RAM - 8 Gigabytes
Software used – Jupyter Notebook (Anaconda Individual Edition 2020.11)
Programming Language – Python 3.6

Experiments and time complexity

Alternatives Considered

For the process of experimentation, we compared three data structures namely, NumPy Arrays, Lists and HashMaps. For each game, we generated the one-hot encoded matrix containing the total moves for every game.

From our experiments we find that NumPy arrays give a superior multidimensional array and devices for working with these array clusters.

The Experiment

For each game, we generated the one-hot encoded matrix containing the total moves. The main agenda for our experimentation was to observe the time taken by each data structure to store the encoded matrix for numerous games and study the variation in time complexity.

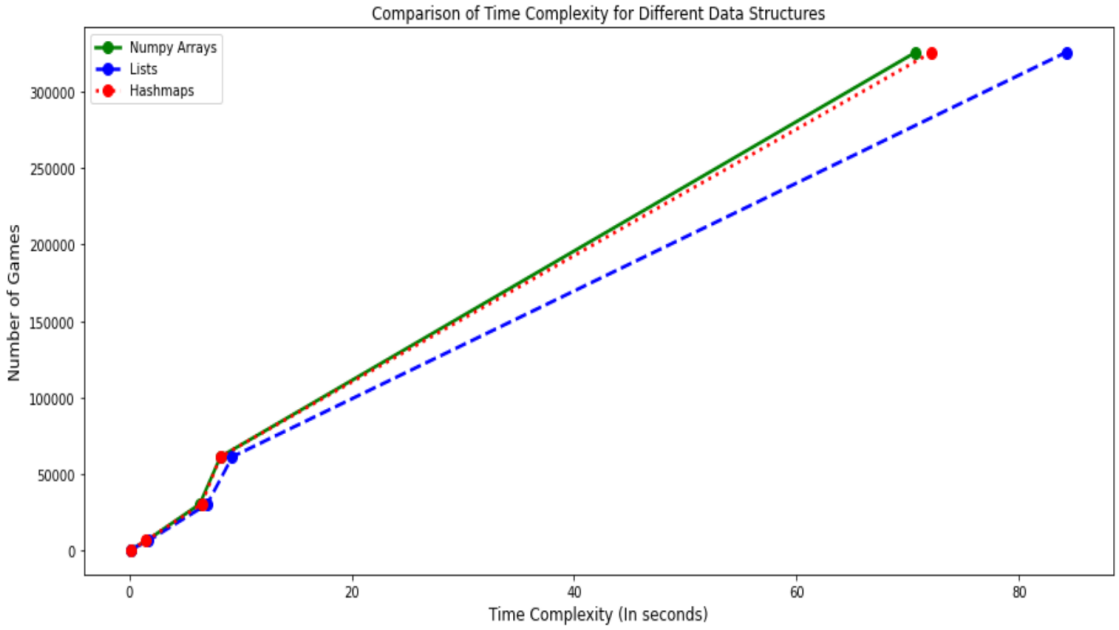

Fig. 7

Fig. 7 depicts the comparison of time complexity for different data structures with the number of games. In the above line graph, the green colour depicts the time complexity for NumPy arrays, the blue colour depicts time complexity for lists and the red colour shows the time complexity for HashMaps.

It can be clearly observed that till the number of games were less than 10,000, the time taken to store the matrices is nearly equal for all three data structures. But, as the number of records or games increases, the line depicting time complexity for lists starts to deviate which showcases that it starts to perform poorly with respect to NumPy arrays and HashMaps.

Number of Games	Time Complexity for Numpy Arrays (In seconds)	Time Complexity for Lists (In seconds)	Time Complexity for Hashmaps (in seconds)
540	0.164182	0.150887	0.143936
6193	1.532843	1.694492	1.487601
30166	6.326215	6.998555	6.505342
61169	8.199140	9.263883	8.183428
325900	70.730473	84.268777	72.165584

Fig. 6

As we can see from the following figure 6 that when the number of games are 540, the time taken to store the matrix data is maximum in NumPy arrays with the value of 0.164 seconds and minimum for HashMaps with the value of 0.14 seconds. But as the number of games increases, the time complexity of NumPy arrays starts to get better than lists and HashMaps.

We can see from the fact that when the number of games were 3,25,900, the time complexity of NumPy arrays was 70.73 seconds whereas the time complexity of lists and HashMaps was 84.26 and 72.165 seconds respectively. This table in figure 6 also depicts that when the number of games starts to increase, the time complexity of lists starts to decrease drastically as compared to Numpy arrays and HashMaps.

How NumPy based Arrays outshine HashMaps:
Numpy arrays provides faster indexing of large data as compared to HashMaps, as the volume of data grows the deviation between the performance of Numpy arrays and HashMap also grows.

How NumPy based Arrays outshine Lists:
As we know that for creating a new object in a list, we require 8 bytes in order to reference that object. Also, if we just want to store an integer, 28 bytes are required by the list. Hence the total size of the list, in this case, can be calculated by:

Total memory size of list = 64 + [(28 bytes)*(length of list)] + [(8 bytes)*(length of list)]

Whereas storing an integer array in NumPy takes up 8 bytes only. Hence the total size of the numpy array is given as:

The total memory size of NumPy array= 96+ (8 bytes)*(length of the array)

Hence, we can clearly see that the total memory used by the NumPy arrays is much less than that of lists

Conclusion
From the experiment it can also be inferred that when the records reach nearly 3,00,000, the NumPy array starts performing the best and as the number of games increases, the deviation between the lines of time complexity for NumPy arrays and HashMaps will increase further.

This clearly indicates that the best kind of data structure that can be used for handling this dataset, is NumPy arrays and also proves that Numpy arrays are scalable for a large number of records.