

Henry Oldroyd

A level computer
science NEA

Chess Program

Candidate No: 7145

Centre No: 17535

My Final Product:



At the end of this project I was able to create a fully working chess game with a web based GUI and click interface. I created the interface and AI that the user plays against completely from scratch, without using any chess libraries. My chess AI can be told to look X many moves ahead to decide the best move. I implemented a database to allow the AI to learn from previous games it has played. I also added various difficulty level from easy to very hard. The pieces taken and move history are also displayed.

This project covers an implementation of the “minimax” search algorithm with alpha beta pruning. I also added my own optimisation including a caching function so that the engine doesn’t have to re-analyse positions that it has seen before. I also added a variable search depth (like the famous chess program Deep Blue). I even added a feature where multiple instances of the searching algorithm can run at once in different python processes in order to maximise use of a multi-core CPU.

Contents

My Final Product:.....	2
Glossary of Key Terms:.....	10
Analysis Section.....	11
Problem definition:	11
What are the benefits to solving this problem with a computational approach?.....	11
My computational approach:	11
Thinking ahead:.....	12
Inputs:	12
Outputs:	13
Preconditions:	14
Thinking abstractly:.....	14
Thinking procedurally / decomposition.....	15
Thinking concurrently:	17
Stake Holders:	17
Research:.....	19
Essential Features:.....	26
Limitations:	27
Solution Requirements:	28
Success Criteria:	29
Design:	33
Initial Design for the system's features and user interface:	33
Breaking down the problem into smaller components as well as explaining and justifying the structure of the solution.	34
The title / menu page:.....	44
The pregame config form page:.....	44
The chess game play page:	44
The game over pop-up:.....	44
Usability features to be used in the solution:.....	45
Effective	45
Efficient	46
Error tolerant:	47
Easy to learn.....	47
Engaging	48
Further Problem Analysis At a Technical Level:.....	49

Design Objectives:	51
Some changes to the design:	53
Usability Features to be Used in the Solution:	56
Effective	56
Efficient:	56
Error Tolerant:.....	56
Easy to Learn and Engaging:	56
Proposed Solution Description:	57
Front End:.....	57
Server Side API Connection and Webserver:	58
GUI to communicate.....	58
Chess Computer:.....	59
Benefit of Decomposition:	59
Diagrams of propose solution:.....	60
Decomposition of Chess Computer (not algorithms just functions, classes and data structures)...	63
Decomposition of GUI logic and API client server connection:	64
Chess Computer: All major algorithms	66
The Chess Engine / Chess Functions:	66
The Move Engine algorithm:.....	75
Game Manager:	80
GUI logic and API client server connection:.....	82
Describe database use (save games and minimax cache) and other persistent storage:	90
Variables and key data structures used in these key algorithms:	94
Complete trace tables and dry runs of key algorithms:.....	95
Validation:.....	104
Client side validation needed:.....	104
Server side validation:.....	106
Functionality that each prototype will have:.....	106
Prototype 1:	106
Prototype 2:	107
Prototype 3:	107
Test Plans	108
Test Plan for Prototype 1:.....	108
Test Plan for Prototype 2:	108
Test Plan for prototype 3:	110
Post development testing against development criteria (black box alpha testing):	112

Acceptance testing for success criteria (beta testing):.....	115
Iterative Development – Prototype 1	115
Aims for this iteration	115
The main aim is to create a working prototype for a system to solve a simpler problem that can be tackled in a similar way to chess.....	115
Functionality that the prototype will have	116
Annotated code screenshots with description	118
Libraries, Languages and other assets:	119
The code:.....	120
Here are some screenshots of the final product:.....	133
Problems throughout and how I tackled them:.....	139
Test Plan for this version.....	148
Changes to code made after testing:.....	155
Validation:.....	158
Feedback from Stakeholder	158
Planned changes after stakeholder feedback.....	159
Evaluation	160
Iterative Development – Prototype 2	162
Aims for this iteration	162
Functionality that the prototype will have	162
Annotated code screenshots with description	162
Testing.....	244
Validation:.....	268
Feedback from Stakeholders	268
Changes/Fixes that I now plan to make to the design or code as a result of testing and feedback	269
Evaluation	269
Iterative Development – Prototype 3:.....	271
Functionality that the prototype should have:.....	271
Annotated code screenshots with description:.....	271
Assorted Module:.....	275
General.py.....	276
chess_exceptions.py	277
Chess Functions Module (Chess Engine):.....	278
Move Engine:	279
Minimax_parallel.py	293

Game Manager	312
Database module:.....	328
Create_database.py	328
Models.py.....	330
handle_games.py	332
Schema Module:	333
Socket_shemas.py:	337
Website Module:	342
app.py	342
Chess_game.html:.....	356
Main.js.....	360
Demonstrating the end result.....	385
Example Play	392
Testing:.....	412
Automated unit tests:	412
Here are the main issues that were detected by the automated unit tests:	433
Timed Minimax not working:	433
Tests completed on final product:	436
Knights cannot take pieces	442
Pawns cannot move forward 2 places:	445
Concurrent workers not caching.....	447
Erroneous Results:	455
Stakeholder Feedback:.....	460
GUI:	460
Special Features:.....	461
Reliability (after the fixes from my testing):.....	461
Suggestions for improvement:.....	461
Evaluation (just of prototype 3):.....	461
Evaluation: Evaluative Testing:	461
Post-Development Testing.....	461
Usability Survey:.....	465
Usability Survey Results	467
Breakdown of success criteria (what has been achieved):.....	473
Limitations and Maintenance:	477
Limitations with the program itself include:.....	477
As part of maintenance, certain issues may arise:	478

Some of the key ways in which this chess program could be improved:	478
Appendix:	479
Sources:.....	479
Code Listing:.....	480
Prototype 1:	480
directory structure:.....	480
app.py	481
logger.py	482
game_engine.py.....	484
index.html	487
main.js.....	489
Style.css.....	496
Prototype 2: Vue page	497
ChessGame.vue.....	497
external.js.....	501
Prototype 2: Final Code	502
Directory Structure:	502
assorted.py.....	503
vector.py	504
pieces.py	505
board_state.py.....	516
game.py.....	521
console_chess.py	525
test_vector.py	527
test_pieces.py	528
test_board_state.py.....	530
test_minimax.py	535
test_data\vector\from_square.yaml	545
test_data\vector\vector_add.yaml	546
test_data\vector\vector_in_board.yaml	547
test_data\vector\vector_multiply.yaml	548
test_data\vector\vector_to_square.yaml	548
test_data\pieces\test_board_populated.yaml.....	549
test_data\pieces\test_empty_board.yaml.....	551
test_data\board_state\color_in_check.yaml	552
test_data\board_state\game_over.yaml	555

test_data\board_state\generate_all_pieces.yaml.....	557
test_data\board_state\generate_legal_moves.yaml.....	558
test_data\board_state\generate_pieces_of_color.yaml	560
test_data\board_state\piece_at_vector.yaml	561
Prototype 3: Final code	562
Directory Structure	562
__init__.py	565
app.py	565
assorted__init__.py	566
assorted\chess_exceptions.py.....	566
assorted\general.py	567
assorted\safe_hash.py.....	567
chess_functions__init__.py.....	568
chess_functions\vector.py.....	568
chess_functions\pieces.py.....	570
chess_functions\board_state.py	585
chess_functions\test_fast_vector.py.....	594
chess_functions\test_fast_pieces.py.....	595
chess_functions\test_fast_board_state.py	597
chess_functions\test_data\vector\from_square.yaml	607
chess_functions\test_data\vector\vector_add.yaml.....	608
chess_functions\test_data\vector\vector_in_board.yaml	608
chess_functions\test_data\vector\vector_multiply.yaml.....	609
chess_functions\test_data\vector\vector_to_square.yaml.....	610
chess_functions\test_data\pieces\test_board_populated.yaml	610
chess_functions\test_data\pieces\test_empty_board.yaml	613
chess_functions\test_data\board_state\color_in_check.yaml.....	614
chess_functions\test_data\board_state\game_over.yaml.....	617
chess_functions\test_data\board_state\generate_all_pieces.yaml	619
chess_functions\test_data\board_state\generate_legal_moves.yaml	619
chess_functions\test_data\board_state\generate_pieces_of_color.yaml	622
chess_functions\test_data\board_state\piece_at_vector.yaml	623
chess_game__init__.py	624
chess_game\console_chess.py.....	624
chess_game\game_web.py	627
chess_game\game.py	634

chess_game\test_fast_game.py	641
database__init__.py	645
database\create_database.py	645
database\handle_games.py.....	646
database\models.py	648
move_engine__init__.py.....	649
move_engine\cache_managers.py.....	650
move_engine\minimax_parallel.py	659
move_engine\minimax.py	678
move_engine\parallel_minimax_testing.py	689
move_engine\test_slow_timed_minimax_engine.py	695
schemas__init__.py	712
schemas\cache_item_schema.py	712
schemas\socket_schemas.py.....	715
website__init__.py	718
website\flask_server.py.....	718
website\templates\chess_game.html	729
website\static\main.js	733
website\static\chess_class.js	741
website\static\vector.js	751
website\static\initial_game_data.js	752
website\static\style.css.....	752
database\database.db	757

Glossary of Key Terms:

I will use a variety of terms in my course work to describe my approach and I will aim to define them here for maximum clarity.

Board state: This is a general term that I am using to describe the details of the current chess board that are relevant to the game and making a decision about the next move. This is a set of information about which pieces are still on the board for each player and where they are. In addition other information including which player's turn it is currently and the **utility score** for both sides given the current game state.

Move engine: This is a module that will be a component of my finished system and is responsible for making the best move given the current **board state**. It will be used to make the moves for the computer / AI opponent

Chess engine: This is a module that will be a component of my finished system and is responsible for making inferences such as the legal moves, check and is the game over? From the current **board state**.

Decision Tree: This is a general term for a tree where each node represents a decision and the child nodes are the possible options. In this context, by a decision tree I mean a tree that describes possible paths the board can take from the given **board state**. In this case the nodes are **board states** and the edges are moves. Each **board state** has many branches to show its many children. This idea of a decision tree is used by the **Move Engine**.

British Museum Algorithm: This is a variant of the minimax algorithm. Instead of using heuristics, this algorithm keeps exploring the **decision tree** until it reaches the terminal node where the game is over. With this fully explored tree, the best possible move can be calculated. This is feasible for a game like Tic Tac Toe but not for Chess.

Utility score (for a given player): This is a integer value that aims to evaluate and measure the total value of all the pieces on a given player's **board state**. Using player white's utility score and subtracting player black's utility score will give a score that describes how advantageous the current board state is to player white. An example of how the utility score could be calculated could be giving each piece a value (e.g. a queen is worth 9, a castle 3 and a pawn 1) and then totalling these values. This score is useful to the **chess engine** as it can only look so many moves ahead. As such it will need to have a way to compare the **board states** that result from each move in order to determine the best move.

AI and its use in the chess engine: While the term AI is predominantly used to refer to neural network my planned approach of using the **Minimax algorithm** to recursively evaluate a tree in order to determine the best possible move is considered a type of symbolic AI. I aim to looks a few moves ahead be creating a tree where the current board state is the root node. Branching of the root node are many arcs that represent the many possible **legal moves**. The connect to nodes that represent the resulting game state after this move. This process will continue to some given depth and then the **chess engine** will use the **utility scores** to calculate how much the leaf node game state favours the computer in order to select the best possible move. More advanced implementations of the algorithm in late prototypes will use alpha beta pruning in order to improve time efficiency and search to a greater depth of moves.

Analysis Section

Problem definition:

605 million adults play chess regularly around the world according to the UN (<https://www.un.org/en/observances/world-chess-day>). While some play chess at a competitive level with the aim of improving their skills, many of those who play chess do so casually with the primary goal of entertainment and the secondary goal of getting better. Online chess programs are growing increasingly popular as a pastime. Chess.com is one such website that predominantly is used to play chess online against AI opponents and other users online. It has 77 million users.

My aim is to make a website that will allow users to play chess against an AI opponent. The program will be aimed at casual chess players and as such it will include features to help it specifically cater to this group of chess players. The website will have an interactive chess board that users will use to make their moves and see the computer's move. When either the computer or the user wins the game will end.

What are the benefits to solving this problem with a computational approach?

This approach allows users to play chess without a chess board. This enables users who don't own a chess board to still play chess. A mobile device that can access my website is also more portable than a chess board. As such users will be able to play chess on the go for example on a bus or train. In these scenarios it would not be practical to use a physical chess board as there may be limited space and if the journey is bumpy (for example when walking or using a bus) the physical pieces would fall off the board. Additionally, a computational solution like a website would allow users to play even when they cannot easily access another person to be their opponent. For example on the go, if they don't know anyone that enjoys chess or at odd times of day like early in the morning. A chess game on a phone against the AI opponent can also be passed and picked up again at any time which would not be appropriate when playing against another human. This allows games to be played even when there is not a large enough consecutive block of time in which to play them.

For these reasons a computational approach would make playing chess more accessible to more people and allow more games to be played at the convenience of the user.

My computational approach:

In order to determine if a computational solution is appropriate I have analysed the problem using a variety of computational approaches. This has helped me to imagine how a computer may be used to solve this problem even before the detailed planning stage.

It is important to clarify that in this section I am considering the minimum viable prototype that only contains essential features. For example I may end up adding additional features from my wish list. For example allowing users to log in in order to earn trophies for winning and earn a place on a leader board or to load and continue games.

This would greatly increase the number of:

- inputs: details for login pages as well as new menus for user exclusive features
- outputs: for example the leader board or an email confirmation in order to make an account
- preconditions: some features will only work if a user is logged in, in order to manage this issue I may need to make use of sessions or cookies in order to keep track of whether or not a user is logged in

Thinking ahead:

Inputs:

One set of inputs needed will be the configurations for the game. These may include whether or not taking back moves is permitted. This could be implemented as a tick box in order to ensure that a Boolean value is given and to make validation simpler. These configurations may also include the difficulty setting for the chess match (this could for example affect how many moves ahead the chess engine looks affecting the quality of the selected move). This input could be collected with a set of radio buttons. This would ensure that only one option is selected. I could have the normal difficulty be selected by default to reduce the validation needed as this way exactly one choice will always be selected.

One of the important considerations here is that my target user will have likely already used a chess program before. It will make the transition to using my user interface and my chess program simpler if my user interface follows the conventions of other online menus and chess programs. This is because users are already likely to be familiar with radio buttons and tick boxes

Another important input is the users inputting their move. I intend to follow convention in how this is done (the benefits are discussed above). As part of my research I identified that how the user inputs are collected was similar across many chess programs. In many chess programs the user selects a move by first selecting a square and then selecting another square. Assuming the user gives valid input the square they select first should contain a piece that they own. At this point dots will show that possible squares that piece can move to (the set of all possible moves is shown visually). The user then selects one of these marked squares to indicate that this is where they would like this piece to move to. The chess piece then moves to this new square

Chess board is presented and the user is told it is their turn:



The user selects one of their pieces:



The user then selects the square for it to move to



(computers move also shown but this is to be ignored for this explanation)

Outputs:

The first important output is the computer's moves in the game. This is to be shown to the user visually using the convention of lichess.org and chess.com by moving one of the computer's pieces and highlighting the squares it has moved from and into. The move will also be added to a list of the moves in the game. This will be displayed on the side and an example of how a move could be recorded could be "black bishop moves from F6 to G7".

The other main output is who has won the game. It must be outputted once the game has finished. I will display that the game is over and who has won. It must also disable the chess board widget to prevent the user making any more moves. This information may be displayed with a popup window but it may be simpler to have a banner appear above the chess board that states who has won.

Preconditions:

The main precondition is that all inputs must be valid. If this isn't the case then the chess engine will not be able to function.

In order to solve this problem I will try to make many of the inputs forms on any menus feature limited options to reduce the chance of giving an invalid input. For example the user of radio buttons or checkboxes to ensure the only inputs that can be given are valid.

I also need to ensure that the users move is valid. In keeping with the conventions laid out by chess.com and lichess.org I will not display an error message if the user gives an invalid input. Instead nothing will happen and the program will continue to await the users move. This is detailed above in the inputs section. An example of an invalid input that may not be allowed (and so ignored) is the user selecting an empty square or one with an opponent's piece in when selecting one of their pieces to move. Another may be the user selecting an invalid square for one of their pieces to move to (the selected square is not one of the possible legal moves).

Thinking abstractly:

Irrelevant details and properties of the chess board and chess pieces are abstracted and removed to reduce the complexity of the problem. The only details that will be kept will be those that are relevant to the object in game. I will use objects to represent the abstracted chess board and pieces servers side. To clarify these details we be abstracted 'under the hood' in the server side chess engine and the client side validation. The user will still be able to see the colour and general shape of the pieces and the colours of the squares on the chess board when on the website

A chess piece will have these attributes abstracted:

- its specific shade and colour. Instead it will be assigned an owner to ensure the user's and computer's pieces are distinguished
- The exact physical dimensions of each piece e.g. height and width
- The exact physical shape / profile of the piece

Attributes that will be kept include:

- Which player owns the piece
- The set of possible ways in which the piece can be moved
- The relative value of each piece (to be used by the chess engine)
- Where is it on the board (which square) or is it off the board
- Is the piece a king? Due to handling check the king will be a special case with special properties

The ways in which a piece can move will be represented by a set of vectors.

The chess board will have these details abstracted:

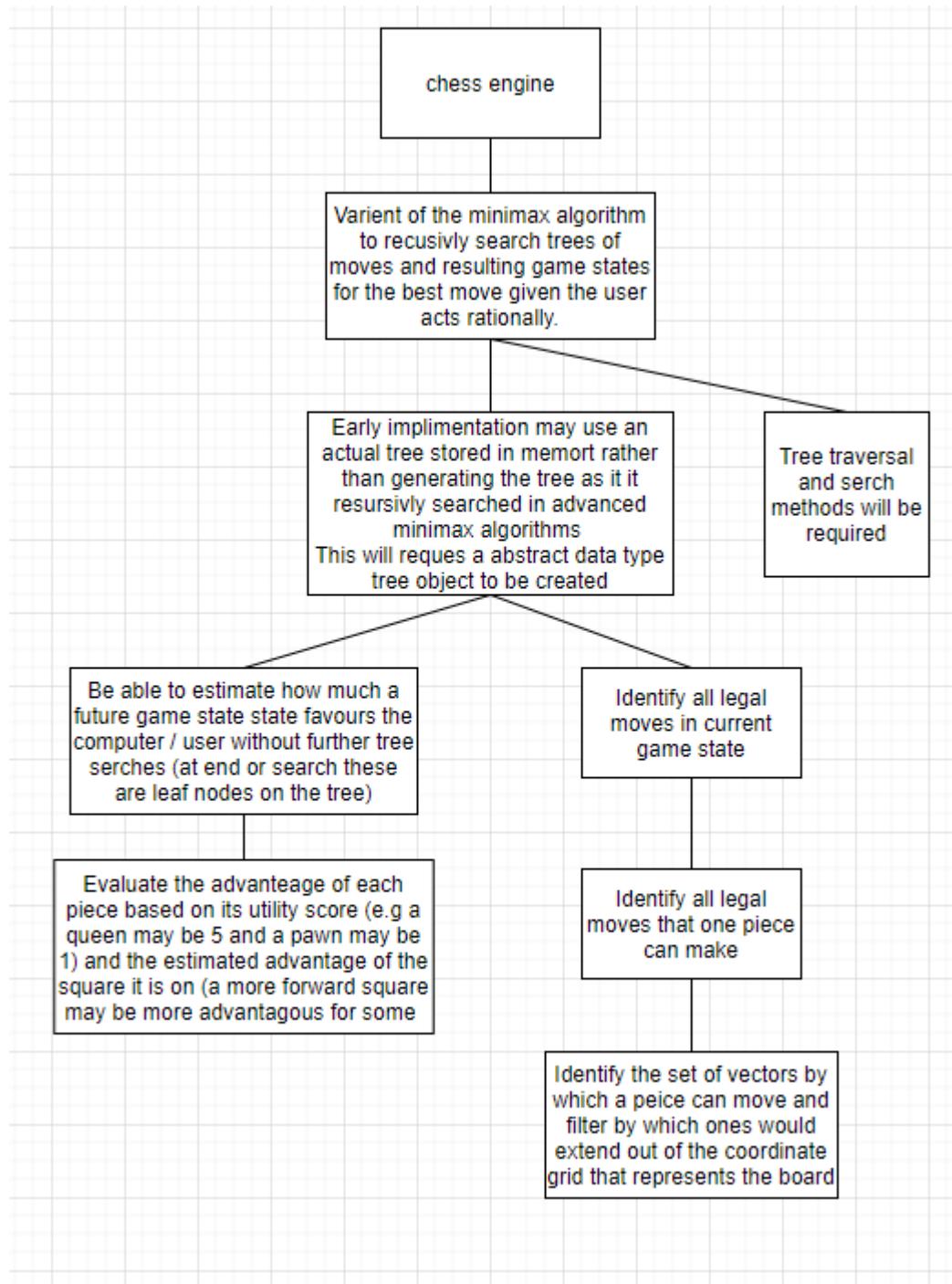
- The colour of each square: it makes no difference in game if a square is black or white
- The physical dimensions of the board e.g. the width and height

The board will be represented server side as an 8*8 coordinate grid. Pieces that are knocked out may be stored in a separate list. Where the pieces are on the board is a detail that must also be stored

Thinking procedurally / decomposition

I aim to break this problem down into subproblems that are more manageable and can be solved independently. The 2 main modules will be the chess engine and the front end website

The chess engine will be responsible for determining a good move to make given the current board state. I intend to implement it by analysing decision trees using the minimax algorithm and symbolic ai. I will go into more detail about this in the design section. An example of how the chess engine may be broken down into sub function is shown



The chess engine will be written in Python. The frontend website user interface will be created using HTML, CSS, JavaScript and Flask to act as a Python webserver. The function of the website is to provide an interface to the user and validation of the user's inputs. The frontend and backend will be connected via a type of API. This could be a REST API that uses HTTP or a type of web sockets as appropriate. HTTP based connection is simpler but sockets allow for a full-duplex low latency connection which is advantageous.

The website will be broken down into pages each of which will serve a function. For example a home page, a user login page and a puzzle game page. Each page will be composed of multiple interactive and static widgets. Each of these will be made up of many HTML tags.

Thinking concurrently:

The main component of this problem which can be solved with concurrent processing is the method of finding the adversarial AI's move. This will involve using the Minimax algorithm to search a decision tree.

This will involve looking at each move from the current board state and recursively applying the minimax algorithm to score that board state. Then selecting the move that gives results in the board state with the best score. It would be possible to break up the legal moves that you need to recursively search (work to be done) into chunks. These chunks could then be processed concurrently to increase the algorithm's speed.

For example, if the board state that must be searched has 24 legal moves I could break this array of legal moves into 4 subarrays of 6 moves. Then I could make full use all 4 cores in my processor to execute the algorithm on these subarray concurrently and then select the best move once all the moves have been evaluated.

This would have the benefit of significantly increasing the speed at which I would be able to perform a deep search (many moves ahead) of the target board state.

A drawback of this approach is that the minimax algorithm is already complex and so programming it to work concurrently will be difficult. This will mean that this feature will take up a significant amount of time and so will have a significant opportunity cost in terms of other features not implemented.

In addition, there will be a significant overhead (time delay) in setting up the various threads and workers needed to run the algorithm concurrently. This means that this approach will only provide an improvement to performance for large minimax calls (searching to a high depth / looking many moves ahead).

This means that making use of concurrency in my project could give a significant benefit to performance of the minimax function if used in the right circumstances. As the minimax algorithm is the main algorithm needed to program the adversarial AI, this should mean that with the same amount of time, the AI would be able to make better moves. This should make chess games more challenging and enjoyable for the stakeholder.

Stake Holders:

Stake holders are those who will be affected by the quality of chess program that is produced

The only stake holder for this game is the end user as its function is purely to provide entertainment. If the chess program doesn't meet the end user's needs or is not created then they will be affected as they will not have the opportunity to get the same level of enjoyment from playing with my chess program.

My target audience is a casual chess player who plays primarily for entertainment with the goal of improving at chess being secondary. For this reason the target user is someone who is already a casual chess player or is interested in giving chess a try. They should enjoy problem solving and be of

age 8+ as chess is a highly complex game that involves high level strategy and problem solving that many younger children are unlikely to enjoy.

I have identified my parents as specific stake holders. Roger is a casual chess player who regularly plays chess against the computer on his phone at the start of the morning for entertainment. He is familiar with the rules of chess and has the technical knowledge to provide specific and useful feedback about which features he may require. My mum is interested in starting to play chess and doesn't have a technical background. This allows her to represent the majority of my users and give feedback that is more focused on usability of the interface and features she would want. This makes my parents ideal to help with beta testing of prototypes. It is important to note that unlike other games there are likely to be many older adults (60+) who want to play chess. This demographic may not play other games like shooters but may enjoy games like chess, solitaire, backgammon for example.

As mentioned earlier I aim to ensure my chess program specifically caters to my stakeholders (in this case my target audience) in the features that it offers.

I am aiming my program at casual chess players. I have tried to plan the features that I will use accordingly as shown. Further details about each feature will follow in later sections

Because my stake holder is a casual chess player (entertainment first / improvement in ability second) I have decided to implement these features:

- The chess games against the AI will be more casual and will allow you to backtrack and see the board state before the most recent moves to see what is done (a feature on chess.com and one Roger recommended as one he wishes for). A tick box could also be selected before hand to allow for moves to be reversed. This would obviously be an advantage but could be a feature that very new players who are learning may demand
- Puzzles will be made available that aim to be a challenge that can exercise problem solving skills. This challenge will be in the form of a board state that is 1 or 2 moves away from checkmate (the user winning) their goal is to find the path to winning. This should be entertaining and hopefully I will be able to create a variety of different challenges of different difficulty
- As the games are casual there will be no clock that limits how long the user has to play. This is standard in tournaments but I am trying to emulate casual chess that is played on a kitchen table between friends so these features serve no purpose.
- The most important one as mentioned in interview: The user will be able to dictate the starting conditions of a match in order to have a fun off the wall match. For example where they have an unfair advantage over the hardest AI by starting with 1 king and 8 pawns as normal but all other 7 pieces are queens. This should be a fun and different game for the user to play.
- Casual chess players are less likely to be completely committed to chess in comparison to competitive players. This means that they may appreciate other games being made available that are similar. For example draughts. These games are similar to chess in terms of the board and pieces but may be a refreshing change for users.

My older stakeholders (e.g. 60+) may be less familiar with mobile devices and technology

In order to ensure my chess program is suitable for people of all ages I will ensure that the final product is distributed as a website and as a mobile app if possible. I believe that older people are less likely to own a smart phone or be familiar with touch screens. I know that my own grandfather plays solitaire on his PC computer on a website as he is more comfortable with this. This way older users should still be able to make use of the website. I aim to prioritise usability over aesthetics in the user interface design. I will also aim to ask my mum for feedback in detail about how the user interface can be improved to ensure the program is assessable to everyone.

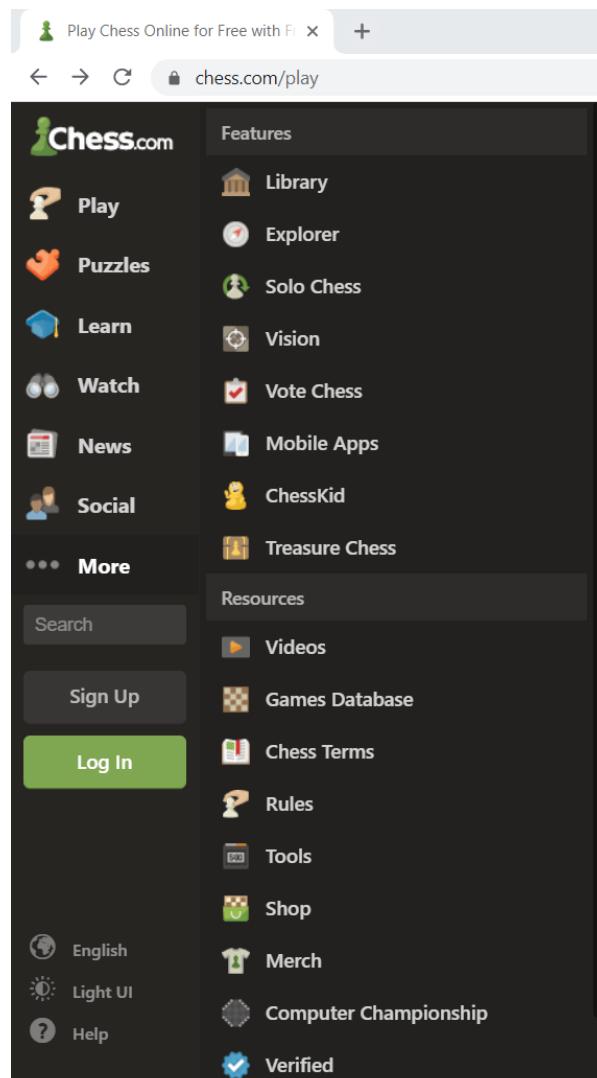
Research:

For my research I have:

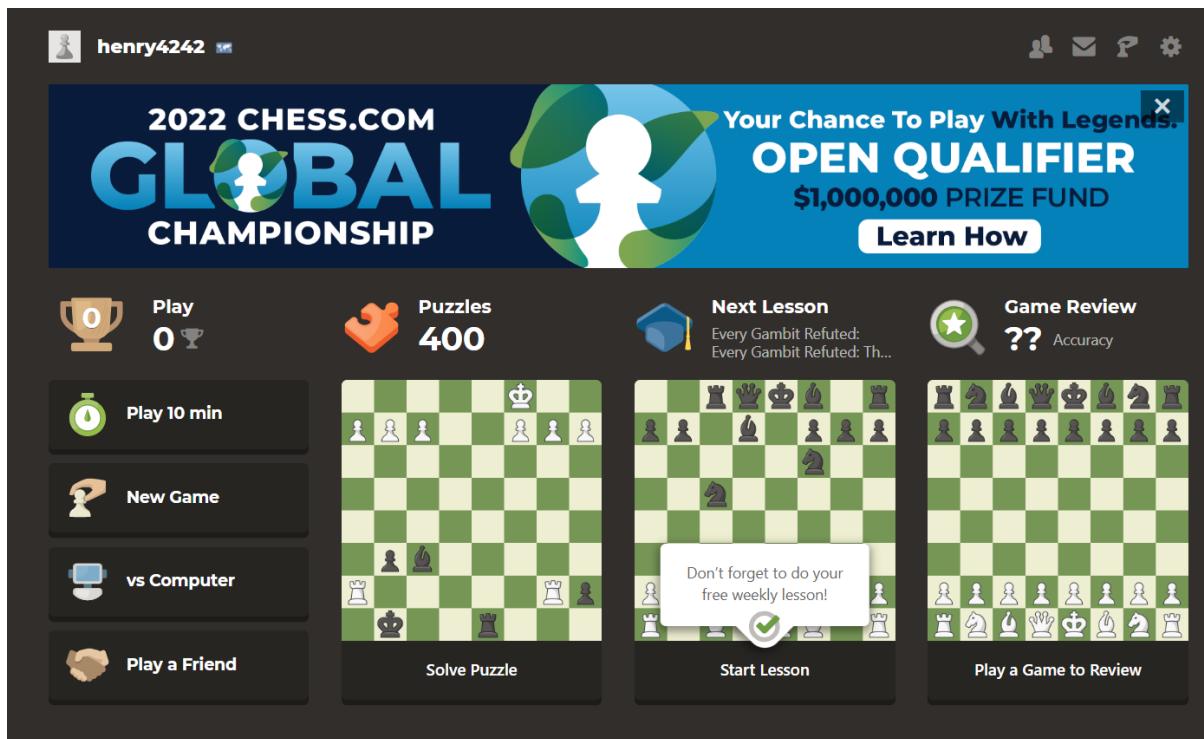
- Looked into 2 online chess games to see their features and functionality: chess.com and lichess.org
- Interviewed Roger who is a casual chess player who plays chess daily on a mobile app

The first existing solution I researched was <https://www.chess.com/>

It is a more broad chess website that features a variety of chess content



I will specifically focus on the playing chess against a computer features of the website.



The user interface shows many features including the opportunity to play puzzles, collect trophies, review my gameplay or play another user. This site was clearly developed by a large team with lots of resources. While there are many sites where you can play chess this site seems to have a massive scope which make sense considering its 77 million users.

This site also allows for logging in which allows for users to access other features and upgrade to the premium mode. The login process was smooth as I could log in with Google preventing the need for a verification email.

Once I began to play a game I was asked to select a few pre-game configurations

Play vs...



Jimmy (600) 

Jimmy wants to make sure you enjoy the game. He'll adapt to make it a little easier, or a little harder, depending on how you play.



Nisha (900) 

Nisha is a friendly person who plays a little defensively and enjoys playing with people at all levels.

ADAPTIVE 



BEGINNER

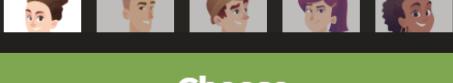



Choose

ADAPTIVE 



BEGINNER

Choose

Play vs...


Play vs...



Jimmy (600) 

I PLAY AS



MODE

Challenge
No help of any kind

Friendly
3 takebacks & 3 hints allowed

Assisted
All the tools available

Play

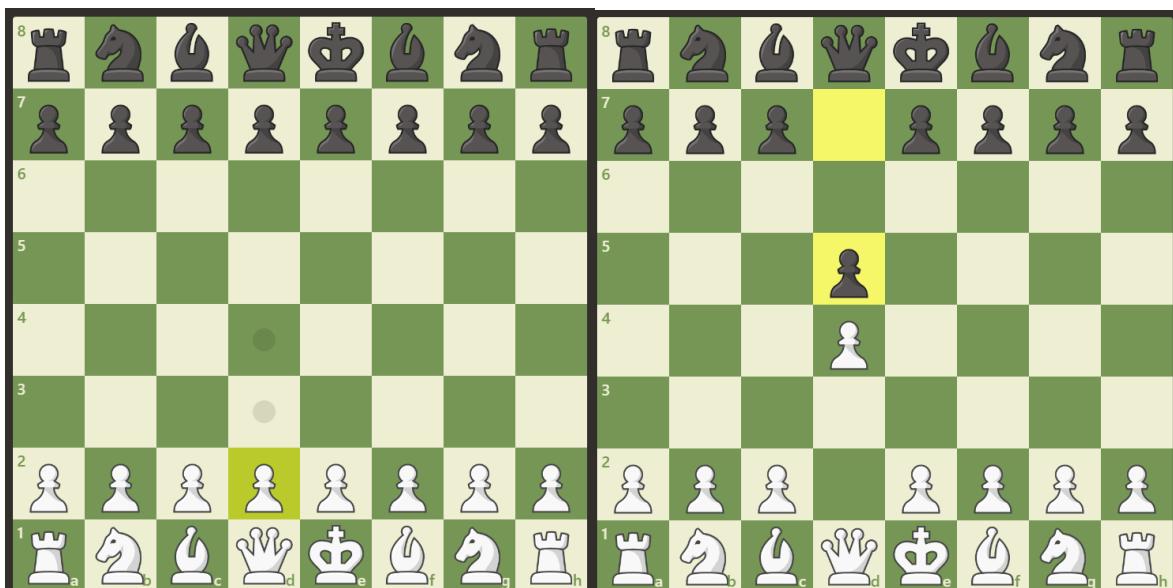
Interestingly they offered more than just different difficulty settings. They seemed to have different chess engine configurations represented by personas like Jimmy and Nisha with different skill levels. This seemed to dictate difficulty and the play style of the AI opponent.

The user then had the option to select how much help they wanted in the form of hints and takebacks.

Then the game begins and an interactive widget is used to represent the board visually to the user.



The user can then select a piece to move and the square to move it to. Then the computer will make its move. Selecting an invalid square or piece does nothing and the program still waits for you to make a valid move. Once you do your computer opponent immediately makes their move. The move they made is highlighted by yellow squares to make it clear to the user what the computer has done.

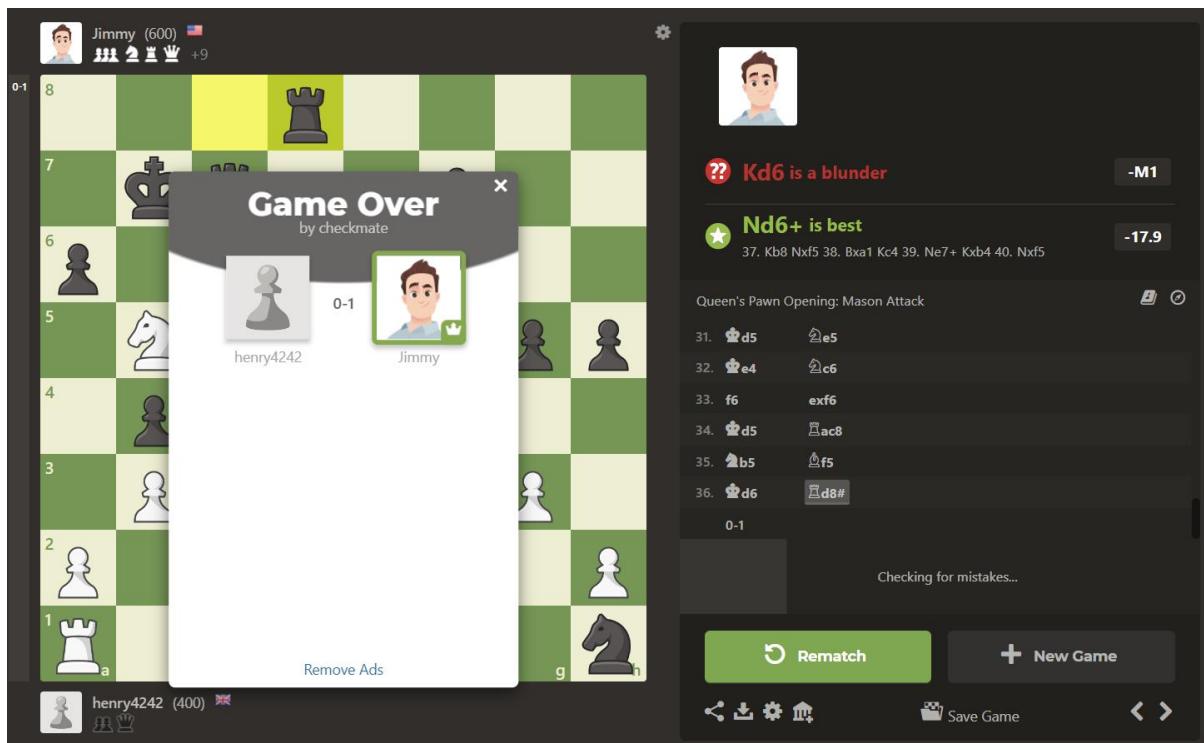


For the users benefit there is a list of the moves made by each player



This list is interactive and clicking on a move will revert the board to the game state immediately after that move. This is shown above.

I ultimately find this list confusing and not descriptive without using the step back feature. Some moves have an icon next the square they moved into to show which piece moved but there is no icon for pawns. Additionally without stepping back through you cannot tell which square the piece came from originally.



Once checkmate is reached the game ends as the chess board prevents new moves from being made. A popup window show that the game is over and who won. The rematch button is highlighted to attract the user's eye. Taken pieces are shown next to the player along with name, picture and trophies.

Overall chess.com is a highly functional website with incredible scope and many other features including allowing 2 users to play chess against each other over the internet. I thought that for the most part the user interfaces were clear however sometimes there was information overload due to the amount of options and statistics on each page.

I like how the difficulty setting and the help offered in game settings were distinct and different. I am likely to use this in my program.

I thought that some of the features may have been more suited to more competitive players. For example trophies and match feedback e.g. "Kd6 is a blunder". In addition some functionality such as some of the puzzles and ai opponent personas were locked until you had played more on their website. I will not lock and features from logged in users based on trophies etc as such a feature seems wouldn't meet the needs of my casual chess player stakeholders who may not have time to play regularly.

I then conducted an interview with Roger:

Roger is a casual chess player who plays chess on a mobile app on his phone most days. The app he uses is called Lichess and is available both as a mobile app and as a website (<https://lichess.org/>).

He mostly uses Lichess to play games vs the ai (it like chess.com has many options of how to play)

I asked him about what he liked about Lichess:

- He liked the fact that he could choose a difficulty level. This makes sense as it will allow different players of different abilities to still play at the challenge level they want. It also allows one person to have an easier or harder game if they want
- He liked the interface as it was user friendly. He particularly likes how you can drag and drop or click to move pieces rather than type in what move you want to make. He found this easier to use and more intuitive
- He especially mentioned the satisfying sounds that this site uses as being an attractive feature, there is one sound for moving a piece and one for taking it
- He also likes how the moves are recorded and he can see the last move visually with highlighting and see a list of moves made in the game. It is also possible to step back through the moves like chess.com

I had a look myself at Lichess afterwards and noted how its interface seemed to be very similar in its features and design to chess.com suggesting that a convention is followed

Lichess also allows for 2 users to play against each other online but I am more interested in the chess vs a computer component

The same interface for getting the move input and stepping back through the games is used



One of the features he likes about Li Chess is how configurable a game is. This is perfect for a casual player as it allows for colours to be changed (for example to brown and cream) and the colour that the user plays as (white which goes first, or black). In addition the starting positions of the pieces can be configured. This has allowed him to play variations of chess against the strongest AI where he plays with an unfair advantage. For example him having extra pawns of the opponent having no queen.

The second feature he likes that chess.com doesn't have and I am now considering adding is an undo button that allows the last move to be taken back. He is a casual player and uses it occasionally

when he miss clicks and selects the wrong move. He pointed out that players know that it is a kind of cheat and can always chose not to use it.

The third feature he likes was the puzzles. Some of these puzzles are about finding the path to checkmate in 2 moves and others are about maximising your utility score over the computer by seeing the advantageous move. For example one of the challenges was to see how you could make 2 moves to sacrifice a knight in order to take a queen. He wasn't told that this is what he was aiming to do. If he selected the wrong move then he was asked to try again and given the option to see the solution. Then the computer made it's counter move and he was asked for a move for the second time. He said he enjoys this as even though he doesn't take chess to seriously he still sometimes wants to improve. He also plays a puzzle when he doesn't have time for a full game.

We then discussed some other features and he gave me his opinions.

He said that it would be really useful if the server would store the game as it is currently after each of the users moves. This way if the website was closed or Wi-Fi is lost then the game can be continues when the user logs back in to the website. He said that this wasn't a feature that he was aware of on Lichess.

He said that he wouldn't be interested in a trophy reward system for winning matches or a leader board for ranking players this way.

He also said that he thought that if I add in a move clock / timer is should be optional. He said that while some players may enjoy the tension it creates, he enjoys having the time to fully consider each move. He viewed this as more of a feature for competitive players (who are not the target audience).

He also said it is important that difficulty settings are discrete not continuous (e.g. not a slider) so users can ensure they are playing on the same difficult.

Essential Features:

One of the desirable properties of this project was how simple the most basic chess program could be. For this reason the essential features are relatively basic. The goal is to then add in features form the success criteria on top of this.

How I aim to implement this is described in decomposition section

The essential features are:

- The user interface must have a menu where the user can select the difficult of the game that they would like to play.
- It must then present them with a fully functional chess board widget as is discussed at length in the research and thinking ahead section to allow the user to enter moves and see the computers move
- Proper validation must be in place in client order to ensure that the new game state resulting from the user's move is only sent to the backend chess engine once a valid move is inputted

- The client side JavaScript must be able to recognise when the game is won, stopping the game and announcing the winner accordingly.
- The server must be able to keep track of the current board state by representing the chess board with an instance of a dedicated class or with a data structure such as a 2 dimensional array
- The server must be able to determine a good move given the current board state. The goal is to use the minimax algorithm in later versions. At its most basic looking one move ahead and picking the one that gives the greatest utility score advantage over the user will suffice (so a move that takes a piece for example)
- The chess engine must be able to pick moves that give it an overall utility score advantage over an opponent in the early and mid-game
- In the late game the chess engine must aim to get to checkmate over taking other pieces to ensure that it is able to actually win the game

Limitations:

Here are some of the possible limitation of my current approach and how I aim to overcome them.

Firstly, if I am not careful, the graphical user interface may not be intuitive to older users. As mentioned some of the stakeholders will be older (60+) and it is important to try to make the solution as accessible to them as possible. Older users may be less confident using a touch screen as an input device and may not have access to a smartphone. For this reason I think it is important that the website can also be accessed through traditional laptops and PCs. Older users may find it easier to use a mouse and/or keyboard. However it is clear that this solution will not be suitable for some older people who enjoy chess if they lack basic computer skills (which is not uncommon).

Secondly, it would be difficult to add in specialist moves like castling and en passant. This is because these moves are special cases and so are more than just a possible vector that a piece can move. As such they will be difficult to implement if I do indeed decide to use a vector space and set of vectors to determine the legal moves of a piece. Perhaps later versions of my program could include specialised code to add these moves as a feature but this will not be present in earlier versions. While this means that the program will fail to meet the needs of competitive players who need these moves in their repertoire, I hope that this will be less of an issue for the casual players at which this chess program is aimed.

The third potential limitation of the full stack website approach is to do with the use of server side code that includes my chess engine. The work load of my server program will be proportional to the number of users as they will all use my server side chess engine. As a result costs will be proportional to the number of users and will not be recouped as there is no current plan to monetise the application. Hopefully this will not be an issue as it is unlikely that the program will have many users but in a real life scenario this would be an issue. In addition having a web application as opposed to a downloadable executable file creates complexity in the form of web hosting. For example I may need to host with a cloud provider like AWS. This would mean paying for computation and storage space used by the database. It would also mean that I would need to purchase a domain name for the site. I would need to ensure my server was secure against attack (for example SQL injection or

stealing users data). I may also require specialist API services such as an email api to allow me to verify the emails of new users that sign up.

The forth issue is that I will have to ensure that I comply with all relevant laws and regulation if I decide to allow users to login. This could be complex to check I am following as I don't have access to a legal expert. For an example, I would need to ensure the minimal amount of personal data I store on each user is in compliant with the data protection act. For example this means I would need their permission. One possible solution to both this problem and the email API problem is to use a login with google feature. This would allow google to handle obtaining the users details in advance and asking for their consent for this site to use them. However this would be highly complex to implement.

Solution Requirements:

In order to implement my solution I will need to rely on a variety of libraries, pieces of specialised software and online services.

I will require online resources:

- A domain name
- Online web hosting and compute to run my full stack python application and the chess engine
- An online storage solution for the database as appropriate
- API keys relevant to allow me to log users in. For example I may need to register a new project on my Google developer dashboard and fill in relevant forms the enable users to login with google giving me access to basic information like name and email (and importantly a mechanism for logging users in). Alternatively I could implement this myself with an emailing service and corresponding API

I will require specialised software:

I will need to host my python full stack flask application. For this I will need a WSGI standard webserver such as waitress and a virtual machine running ubuntu in order to run this python program.

I will require programming and scripting libraries, languages and corresponding compilers / interpreters:

- JavaScript libraries: for example I am likely to use VUE.js for its superior component system and server side rendering
- I will use typescript, the JavaScript superset that is type sensitive. This will make programming and debugging client code easier but will require the appropriate compiler to convert the typescript to JavaScript
- I will require SQL database software like MySQL server in order to securely store user data if I decide to allow users to log in.
- I will require the python interpreter and pip package manager in order to install dependencies and libraries that I

- will use.

Success Criteria:

Number	Feature name	Importance	Description	Justification
1	Difficulty settings	essential	The user is able to select from at least 2 different difficulty settings before the game	Interview, this allows a greater range of casual players to play.
2	Validation client side	Essential	The program must continue to await the users move until they use the devised system to input a valid to and from square	is needed for chess program to function at all
3	Validation client side	Essential	When it isn't the user's turn, appropriate validation should be able to decide to ignore any of the user's square clicks	is needed for chess program to function at all
4	Validation client side	Essential	When in check the user must not be able to make a move that results in them still being in check. This is a difficult part of the chess rules but essential to replicating a proper game.	is needed for chess program to function at all
5	Intuitive Move Input with dot markings	High priority	When the user clicks on a square that contains a piece they own, this should be registered in software and in the visuals of the GUI they see. They have completed the first part of the move by selecting the from square. Invalid from squares will be ignored. Once the user selects a piece to move, dots should indicate the legal moves that piece can make. Once a marked square is clicked the piece should move into it	Interview and research
6	Graphically show the move	Essential	Once the user has successfully inputted a from and to square, they should see the piece they tried to move complete the desired move. This includes the piece moving and any taken piece being removed from the board. In essence, the board must be visually updated after both the computer's and user's move	

7	Computer Move output	Essential	Following each move by the user the computer must make a move shown by their piece moving	is needed for chess program to function at all
8	Highlighting	High priority	After the computer makes a move the squares that this piece moved form and to should be highlighted for a short time. This will help show the user what is happening by catching their eye. This should ensure that they don't miss the computer's move by	Research of chess programs
9	Chess engine mid game	Essential	To at least some extent, the chess engine should be able to select moves in the mid and early game that take the opponents pieces and provide a utility advantage. This means that even though none of the board states n moves ahead (where n is search depth) involve the game being over, it should still produce a good move.	is needed for chess program to function at all
10	Chess engine late game	Essential	In the late game, when given the opportunity, the chess engine must at least aim to win by checkmate in order to ensure it can close out the game	is needed for chess program to function at all
11	Alert after each move for check	Essential	When the user or the computer are in check, some kind of alert should be produced to show this. This will explain to the user why their legal moves are reduced (it is important information to convey)	Needed to create an adequate user interface (it is a necessary output)
12	Proper recognition of game over and its properties	Essential	The chess game that the user plays should end when the game is over. Additionally, it should be displayed to the user why the game is over. By this I mean the main outcomes such as "stalemate" and "checkmate". Additionally, the program should be able to identify 2 different kinds of stalemate: No legal moves and not in check, and the game state has been repeated 3 times. It must be recognised when the game is won and, the game must end (no more moves), it must be outputted who has won the game.	Needed to properly handle when a game is over (essential part of the user experience with my chess game)
13	Chess engine aware of	Essential	The chess engine should be able to see when games are over as it looks	This is needed to ensure that

	different game over outcomes		ahead, it should be able to recognise wins and losses due to checkmate as well as both kinds of stalemate. It should give each one a score to appropriate guide its behaviour (e.g. aim for checkmate, avoid stalemate unless loosing etc)	the move engine models a human opponent in order to avoid irrational moves and to create the best chess games for the user
14	Undo button for last user move	Low priority	The user should be able to undo their last move and make a different one, then continue playing the game	Interview
15	Log of history of moves in game	Ideal / desirable	When a move is made it should be added to a list of the moves in the game for later reference by the user: e.g. "white bishop from F4 to G5"	Interview and research of online chess programs
16	Step back through log	Low priority	The ability to see the board state before a certain move was made in order to see what has happened in the last few turns	Interview and research of online chess programs
17	Fast and reliable connection between front and backend	Essential	The API interface between the frontend and backend must be robust and it must be efficient in order to not compromise the websites performance or cause functionality to fail	Essential in order for a web interface to work
18	Allowing users to login & sign up	Low priority	Allow users to create an account to allow access to other features	Research into other chess sites
19	Reloading unfinished game	Ideal / Desirable	Allow a logged in user to continue a game from where they left off if they login again	Interview
20	Puzzles	Low priority	Allow users to engage in puzzles to try to see the best moves in some preloaded game state	Interview
21	Draughts	Last add, very low priority	Allow users to play draughts vs the computer	N/A low importance
22	Allow users to start with a custom game layout	Low priority	For example users could start a game with a second queen or deprive the ai opponent of a queen	interview
23	Leader board and trophy system	After interview I no longer want to implement this (lowest)	Logged in users can earn trophies by winning games and completing challenges. Players are on a public leader board by trophies. The justification for not doing this is that it has a high cost due to its	

		possible priority)	complexity (I must have a user management and database system as well as a multipage website). Additionally this feature is less likely to be appreciated by casual chess players as they are less competitive. This means that it is not the most suitable feature considering my users.	
24	Accessibility through website	Essential	This will make the end product more user friendly to older users. I would define this a factor which leads to decisions to maximise usability for those who are not "tech savvy" or are differently abled. This includes making that all the text and widgets are readable (appropriate colours and sizes)	Thinking about stakeholders needs
25	Mobile access	Desirable	I could make the website work on the aspect ratio of a phone. This makes the program more accessible for those on the go	Thinking about stakeholders needs (e.g. the use of an IPAD)
26	Efficient adversarial AI	Highly desirable	The AI which the user plays against should use efficient algorithms so that a reasonable quality of move can be produced in a reasonable response time (minimise he user waiting)	Needed to create a positive user experience when playing against the AI and to differentiate the different difficulty levels.
27	The AI adversary feels as organic and real as possible	Highly desirable	It should change or learn over time or make purposeful mistakes in order to remain unpredictable and keep games fresh and challenging	Needed to create a positive user experience when playing against the AI
28	Validation of users move	Essential	The user should not be able to input an invalid move (an illegal move)	Essential to have a proper chess game
29	Valid computer move	Essential	The computers move should not be able to make an invalid move	Essential to have a proper chess game
30	Pieces Taken output	Desirable	The use should be able to see at a glance which pieces have been taken	My research of similar sites

			from both black and white to allow them to see who is winning.	showed that this was standard practice
31	Restart game button	Essential	A button should allow the user to reset the chess game in order to play a new game from scratch or play again after the game has ended	My research of similar sites showed that this was standard practice
32	Concede game button	Essential	There should be a button for conceding the game. This will allow the user to still see all the information relating to the game (piece layout, pieces taken, move history) without the game continuing	My research of similar sites showed that this was standard practice

Design:

Following my analysis where I identified the stakeholder and their needs, I will begin my design process by looking at the final product that users will interact with.

This means that I will start by designing the user interface and determining the core features that the program will have. Then once I have done this I will create a set of design objectives. These design objectives will closely follow the high priority item in the success criteria. They should act as specific and achievable technical goals that, if achieved, will ensure that I meet the essential, high priority and desirable elements of the success criteria. If I have more time at the end of the project, I may go on to implement lower priority items from the success criteria.

I will then lay down a concrete plan of how I will structure and implement the proposed solution and clearly define the subcomponents and algorithms within each component. I will then determine the functionality that each of my 3 prototypes should have.

I will then create a test plan for the whole system including testing during development as well as alpha and beta testing post development.

Initial Design for the system's features and user interface:

I would like to note that I have decided not to include the login system initially in my plan. This was because I determined that the number of features that required a user to be logged in was low. Therefore the benefit is small when considering the significant added complexity. So I have decided that my initial plan will not include a login. I will then add detail about the potential structure of a login system later.

Breaking down the problem into smaller components as well as explaining and justifying the structure of the solution.

Due to the lack of a login system the initial proposed user interface is rather simplistic. This is an advantage as it will be easier to program and easier for the user to use. I can potentially add to it later.

The problem that this simplified version of my solution is trying to tackle: is to simply and easily allow my users to play chess.

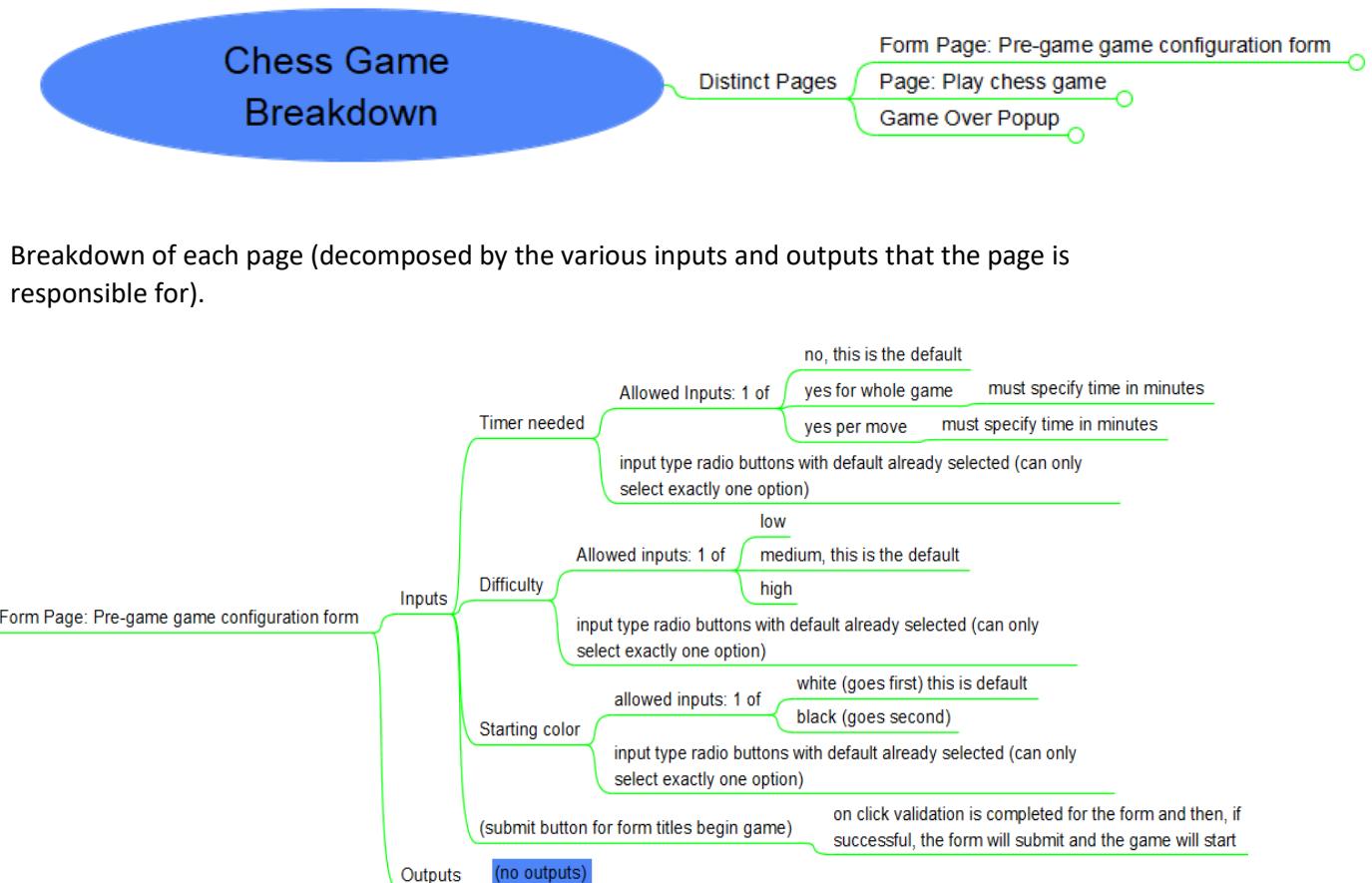
In order to allow the user to play chess I have decomposed the problem in this way.

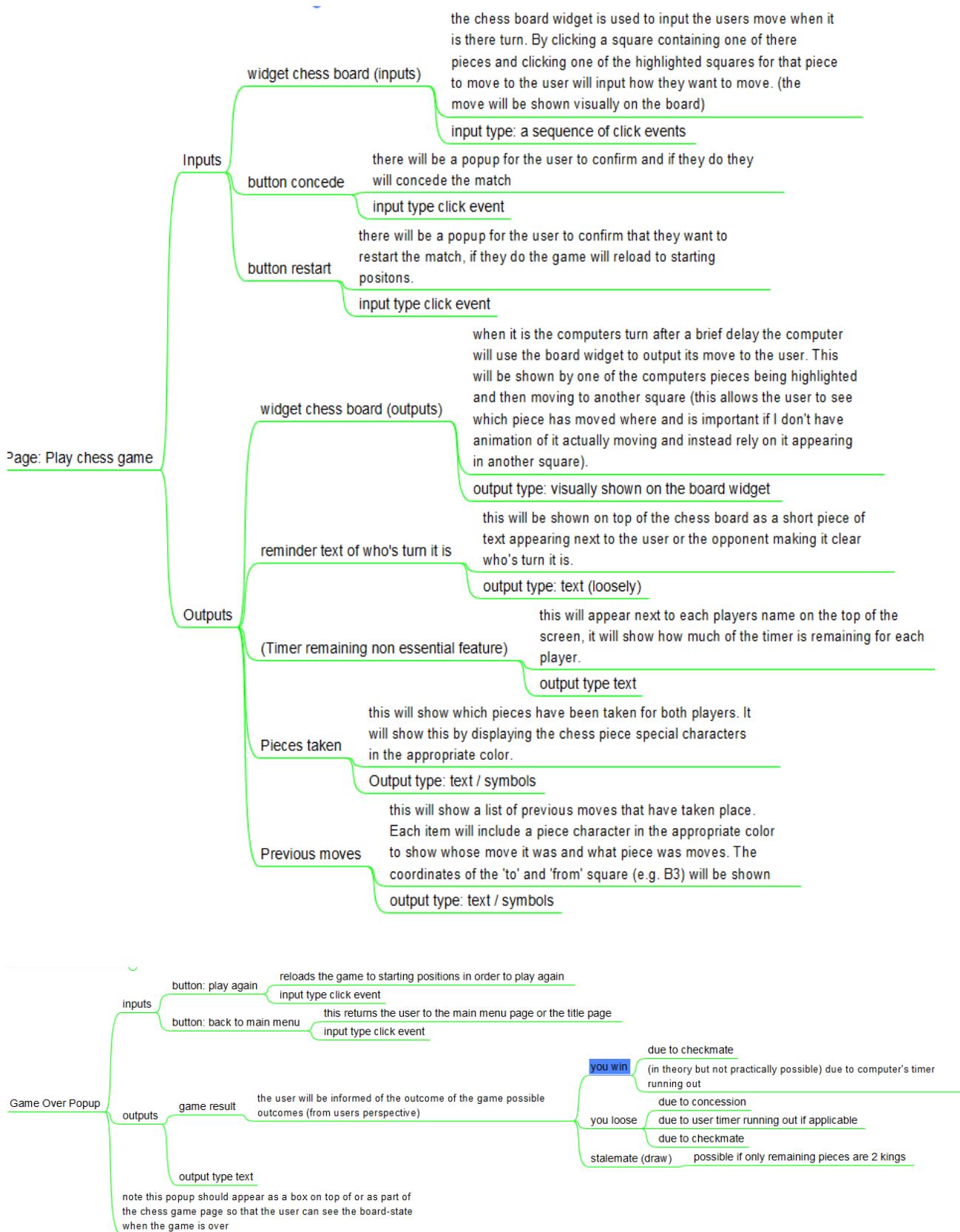
The problem consist of:

- Getting the users input for the pregame configurations
- Providing a way to get input moves and give output moves in order to facilitate a chess game
- Informing the user when a game ends and how it has ended.

Here is a mind map of how I decomposed this problem into separate pages and decomposed each page into a set of different inputs and outputs that had to be gathered. This method aims to use decomposition to break the problem into a collection of pages and widgets and thinking ahead to determine the nature and requirements of each of these widgets / subcomponents.

Pages needed:

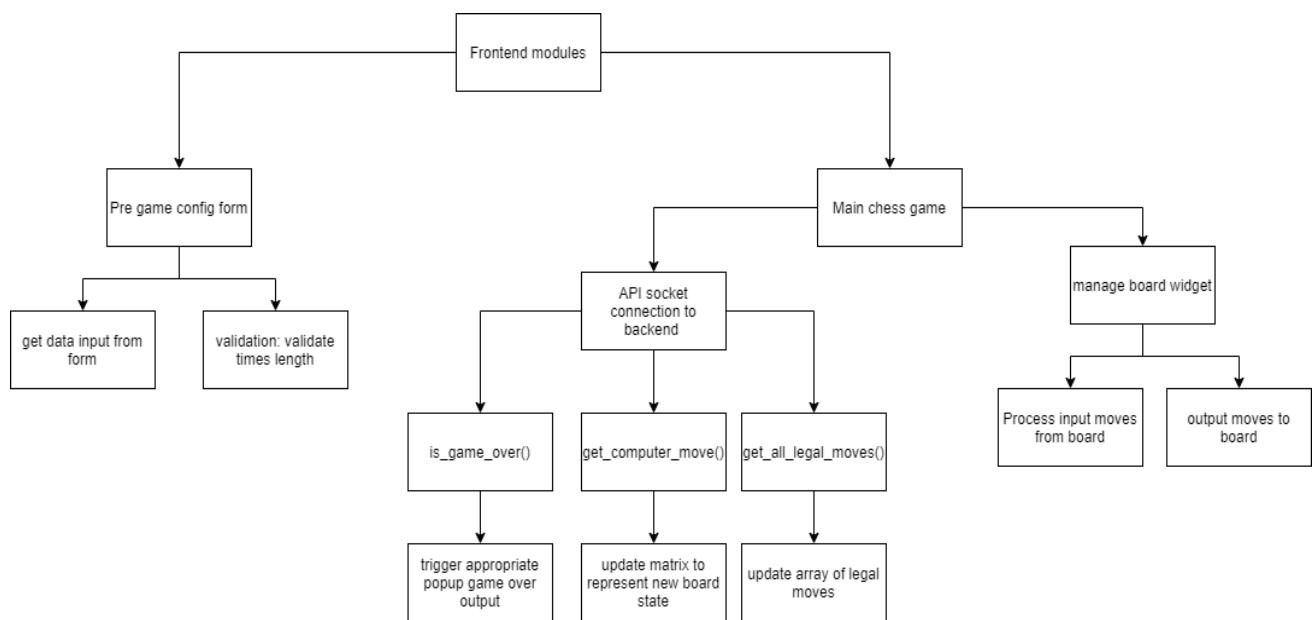




Many decisions were in the decomposition of the problem into various inputs and outputs. here are my justifications:

- For one I decided to reduce the initial problem size by removing the login and user part from my design. This could be added in later but I would first like to complete this design without. This greatly reduced complexity and reduced the number of webpages needed.
- For another I decided that I should include default options for many inputs in the pre-game form to ensure that no matter what the user does a valid input will be given (e.g. using radio buttons). This will reduce work needed for validation and make my program more error resistant.
- I also decided to approach receiving input moves and giving output moves in the way I did as this is a standard among chess apps and websites. Ensuring my program meets the convention will make it feel more intuitive to those who have used other similar programs before and will reduce the learning curve. This was inspired by the research completed in my analysis step on websites such as chess.com.

I have also included a breakdown of the modules that will be needed in the client side JavaScript

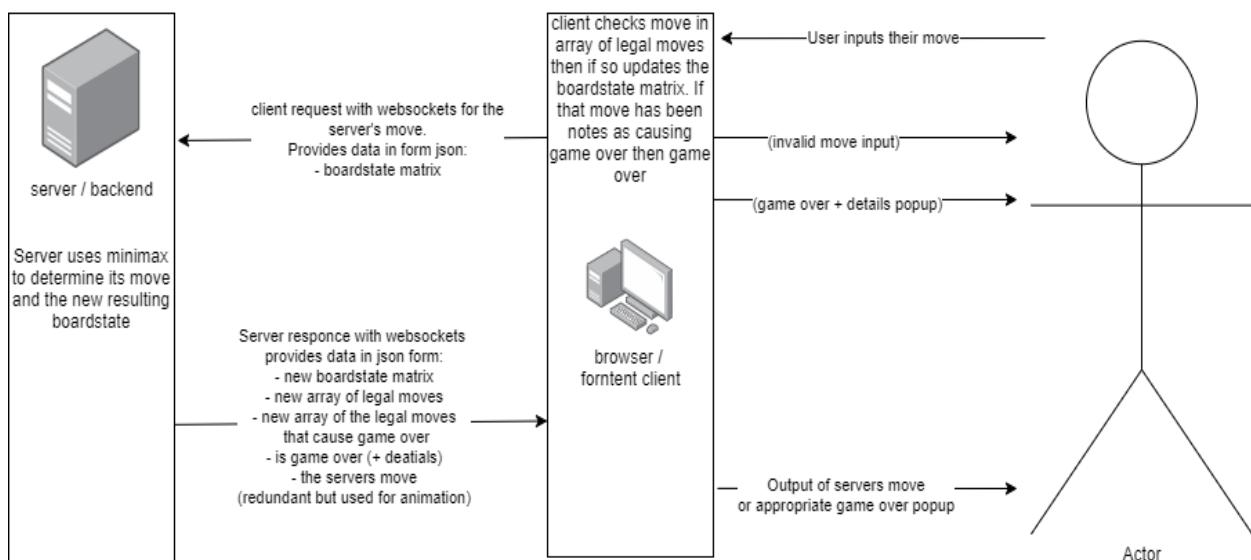


I should explain why the frontend has so few modules and why it doesn't need a way to understand a chess game such as where a bishop can move or if a given game state is check mate.

The reason for this is that I intend to process all that information on the backend and leave only validation to the frontend.

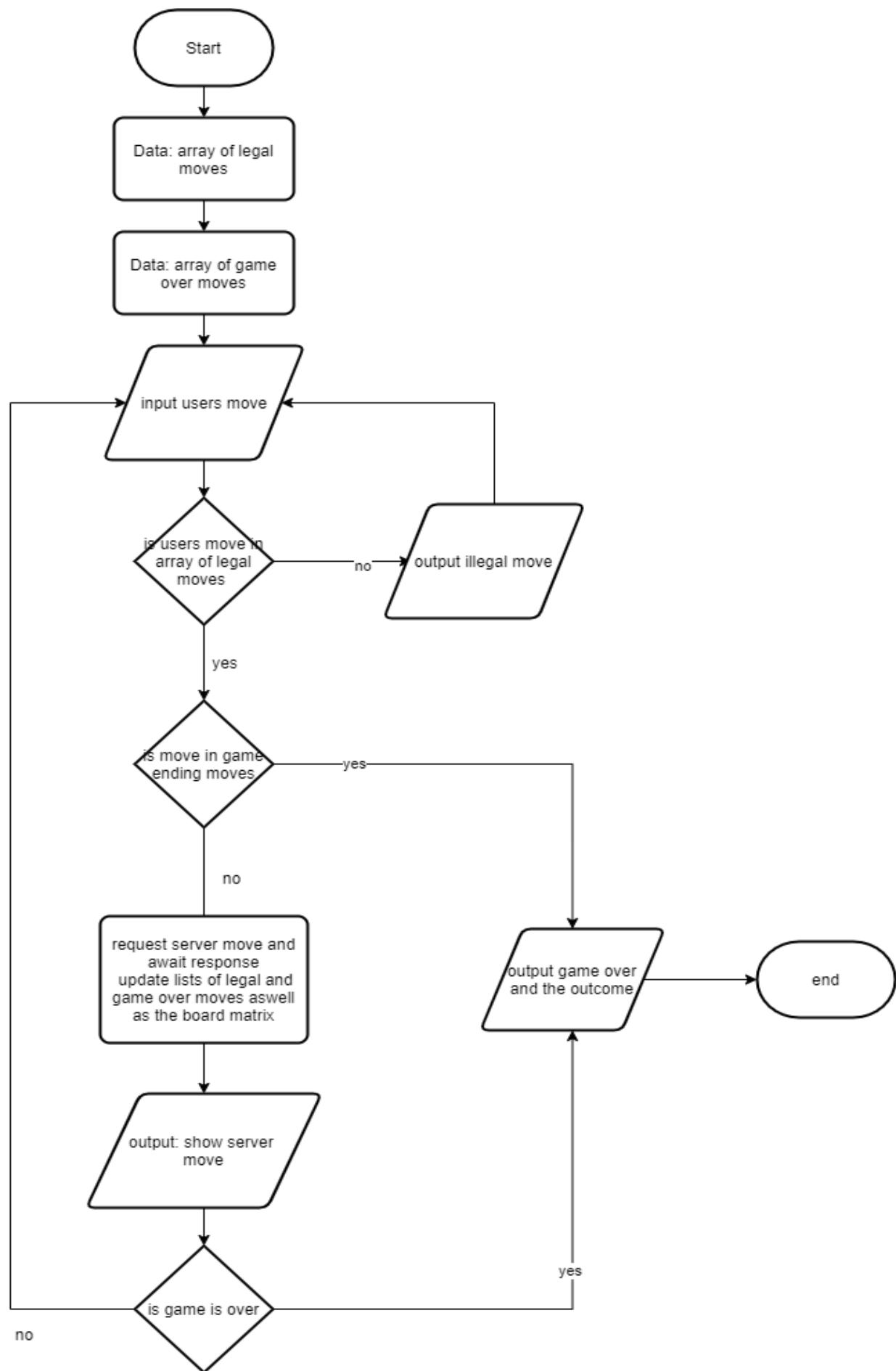
API socket connection to backend is a module that connects to the backend and request information using methods such as `is_game_over()`. It then uses another module to handle the response, in this case a small submodule responsible for causing the appropriate popup to occur to notify the user that the game is over and to state the result.

Here is a diagram of the API connection between my frontend and backend



It shows how the front end client will be equipped with an array of moves that are legal and an array of moves that are legal and end the game + how the game ends.

Here is a flow chart to act as another way to show the logical flow of handling input to produce output in the frontend



The point of this is to explain that the frontend is relatively dumb as it is unable to make inferences about the game ion its own. However by using one request per user move the server can send the client all the data it needs to validate the users next move. This is done by sending a new array of all the legal moves the user can make for future validation along with the server's move.

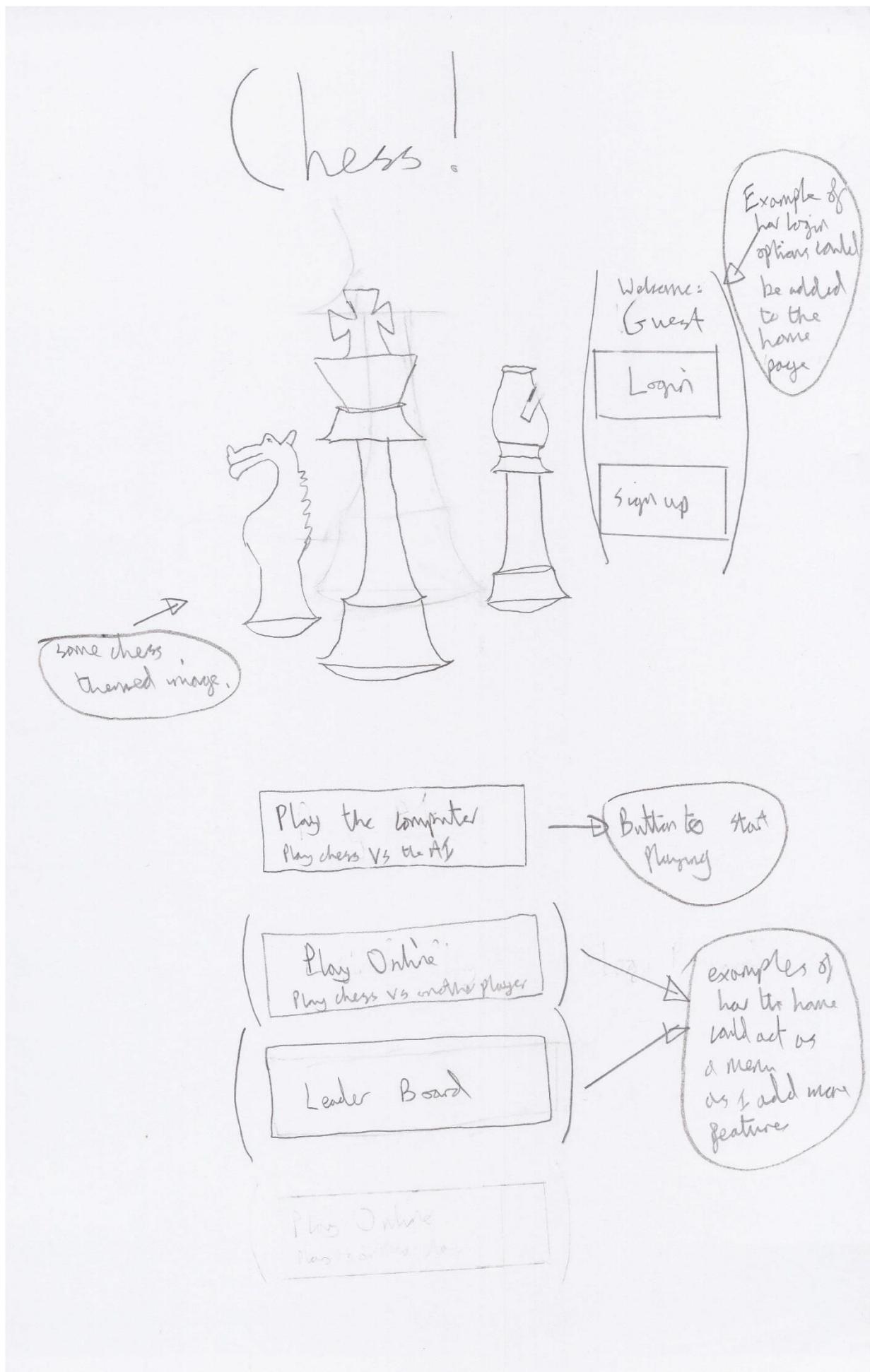
This means that the main responsibility of the frontend (and therefore the purpose of most of its modules) is to validate the users move and process it into a request for the server as well as to process the servers move and turn it into the appropriate output for the user (e.g. know when to display game over) This reduces the complexity of the frontend greatly as it reduces the number of methods and classes needed. For instance no method is needed to determine if a game is over and no class is needed to represent the moving patterns of a knight.

I have used decomposition and thinking ahead to identify the various pages needed and the inputs and outputs that each should provide. Using this I have created on paper some designs for the various pages that would make up the user interface

The scans include arrows with annotations of the various features, some features in brackets are there to show where such a feature could fit if implemented in future.

My user interface will be made up of these pages:

- A menu page / title page
- A pre-game configurations form
- The main page in which a chess game is played
- A popup menu providing information about how the user has won.



home page link

 Game Configuration

Difficulty:

low medium high

Starting Colour
(white goes first)

white black

Allow take-backs

yes no

Timer

no timer

timer for whole game: min

timer per go: min

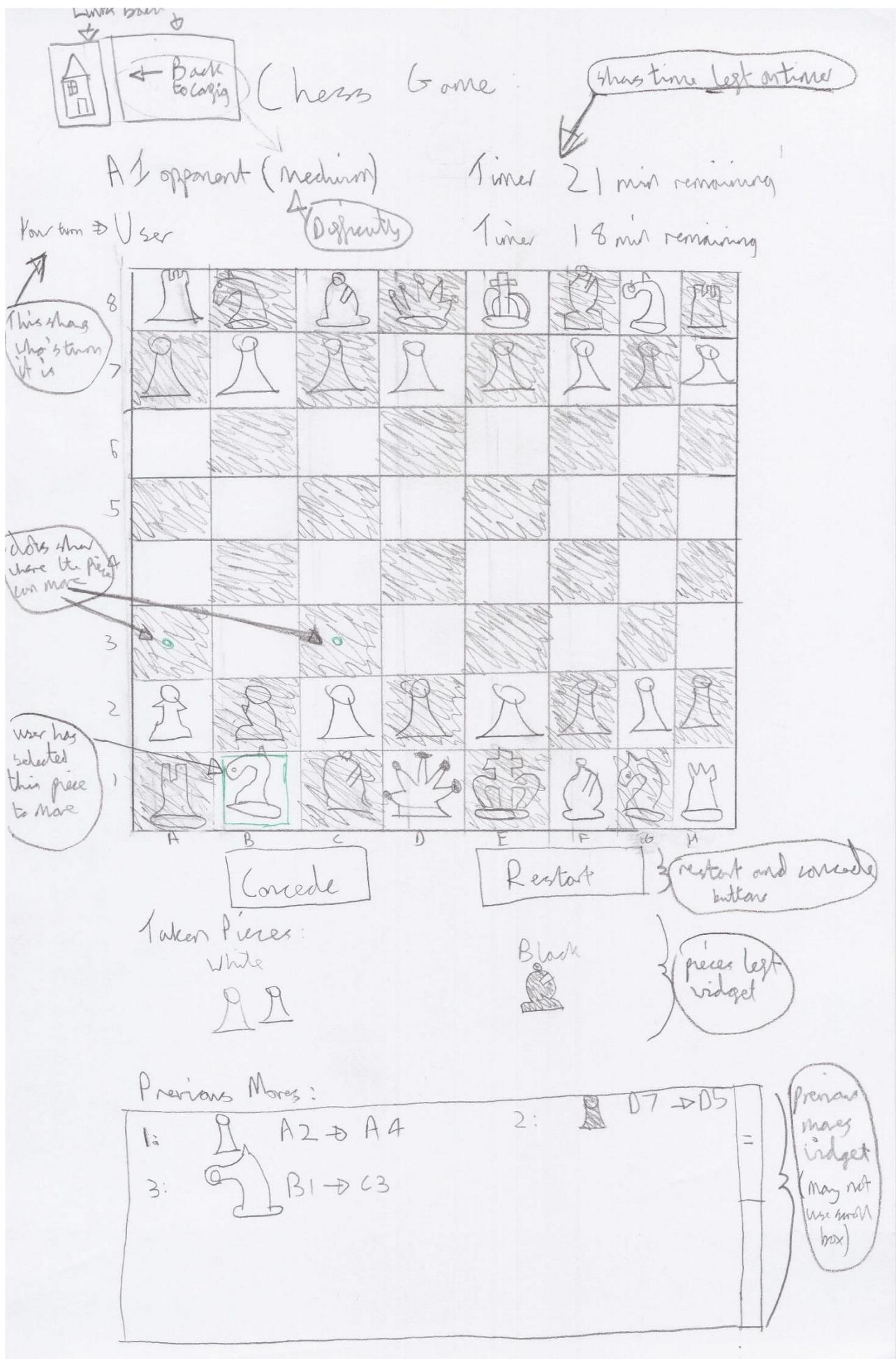
Begin!

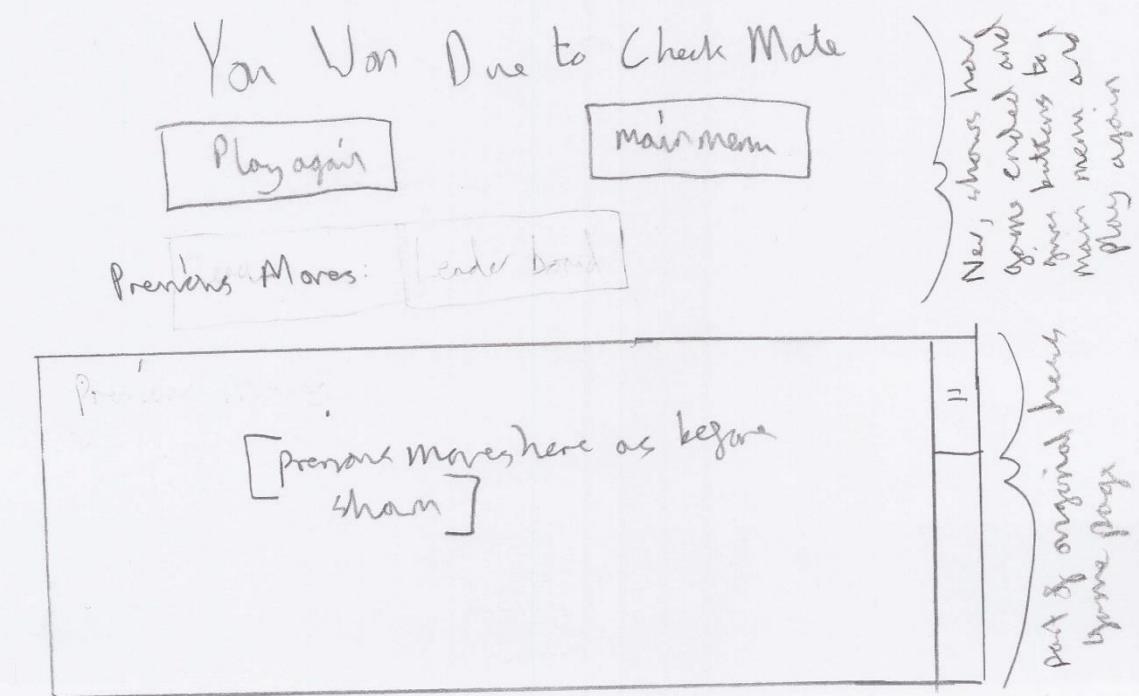
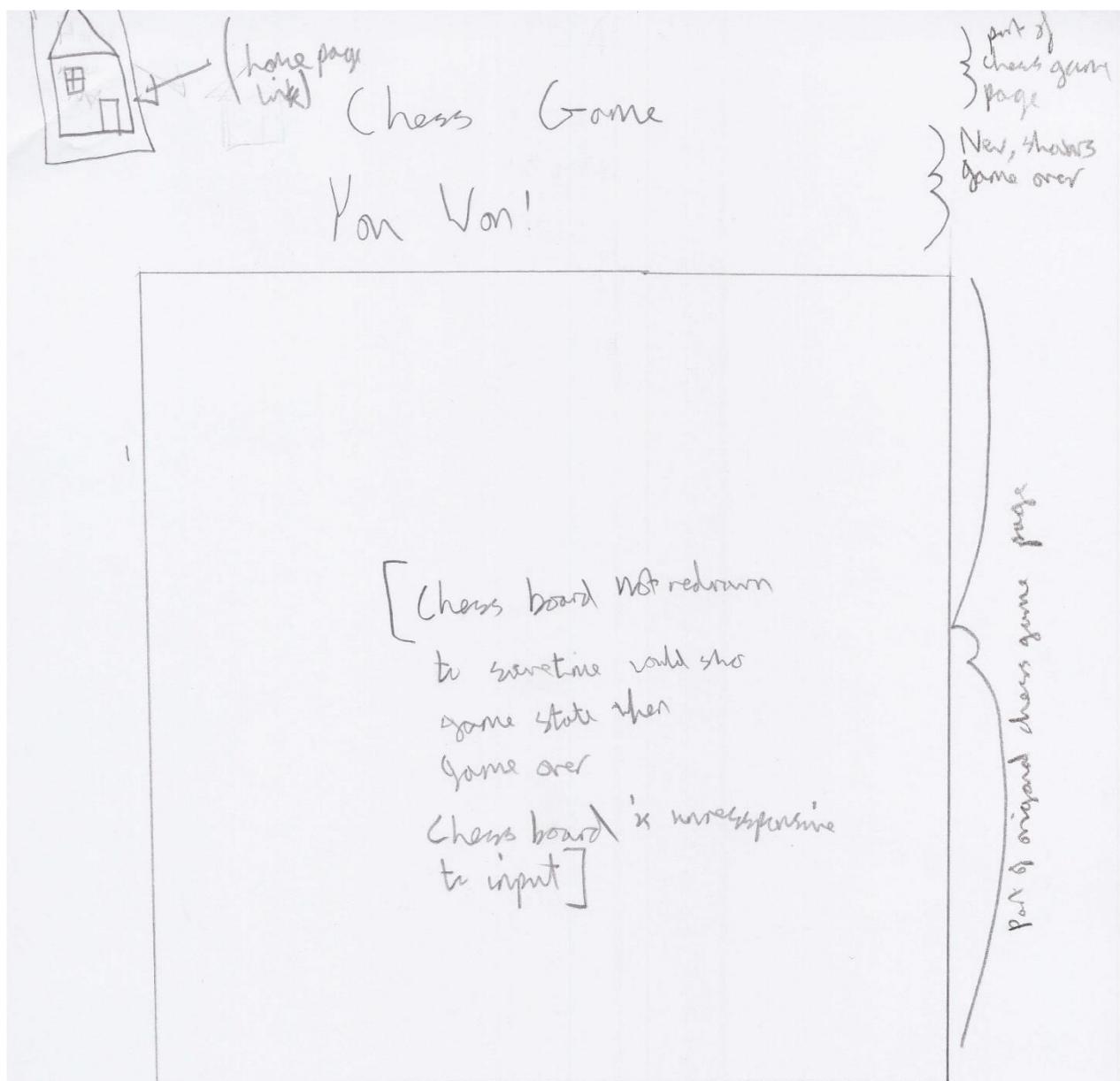
radio buttons with default options

example of other features that could be added to this form

timer length input.
Potential for invalid input so validation required

acts as submit button





I will outline and explain some of the key features and decisions made when creating each page.

Note the aesthetics such as the background have been abstracted in order to reduce the problem. Implementations of these user interfaces will not have plain monochrome backgrounds and features.

I also choose to centre align all my pages in order to catch the users attention (they are most likely to look in the centre)

I would also like to mention here that while I intend to make my website work for multiple resolutions, these sketches are for the 820x1180 resolution of an iPad Air in portrait

The title / menu page:

This page will initially be a simply page with only a title, a chess related image and a button to begin playing vs the computer. As more features are added it will be expanded to become a home page and menu for the website. Examples are included.

The pregame config form page:

This page will be what the user sees after they click to begin a game vs the computer. I will use radio buttons with default options to ensure that the user cannot provide an invalid input for fields like difficulty and starting colour. I will need to add validation to the times length inputs (as an example). This would likely include a presence check, type check (integer) and a range check (e.g. $15 \leq \text{time} < 120$).

The chess game play page:

As earlier stated the method of inputting and outputting is standardised to make the program easier to use.

Outputs like previous moves and pieces taken will auto update after each move. How the timer works will depend on implementation. It will only appear in the top corner if requested in the game configuration.

The game over pop-up:

The main decision here was how the information should be conveyed. I decided against a different webpage or a popup square that obscures the chess game. This was because I thought that users would appreciate still being able to see the state of the current chess board. The current implementation will work by using the same webpage as the chess game play page. When the game is over some components like timers, whose turn and the pieces taken will be hidden and another widget that announces that the game is over unhidden. I may have the pieces taken component not disappear but this would mean that all of the widgets would not fit on a mobile phone screen without scrolling down. This may not be a bad thing if there is enough space left as users may not mind scrolling down to see all the previous moves. This decision depends on how the widgets fit together once the GUI is created and what feels natural.

Here is a menu flow diagram to show how the website is composed of these webpages to form the website.



Usability features to be used in the solution:

I have fully read this article which introduced me to the breakdown of usability into these components. I have integrated its tips with my project.

<http://www.wqusability.com/articles/more-than-ease-of-use.html>

Effective

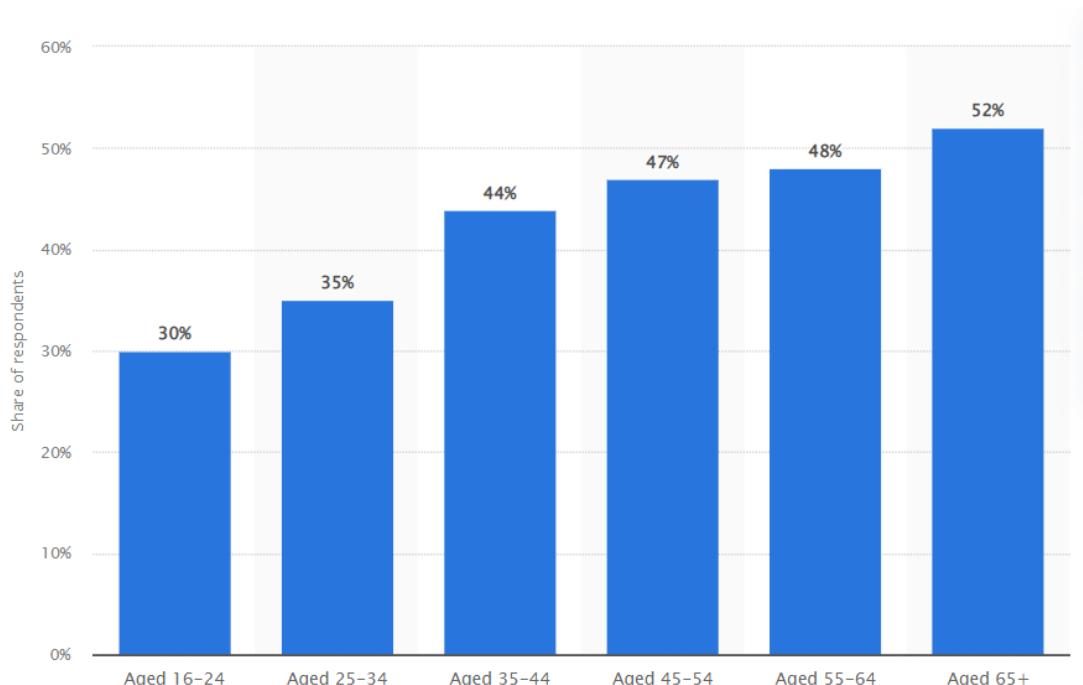
For a user interface (UI) to be effective it must meet the expectations of its users with regard to the utility it offers and also how effectively an average user could use this site to meet their goals.

I have determined in my analysis step that my target users are casual chess players of all ages. As such I have tried to meet their needs specifically by adding features like take backs (if implemented) and an easy difficulty.

To help meet users of all ages I intend to utilise my discussions with my grandparents and my research. I realised that many older people enjoy playing games like scrabble and chess on an app or website. They also commonly use iPads and tablets as they are bigger and so require less finger dexterity. Here is a breakdown of hardware usage by age: (<https://www.statista.com/statistics>)

Proportion of people in different age brackets who use a tablet to go online:

<https://www.statista.com/statistics/300257/tablet-pc-usage-to-go-online-by-age-uk/>



52% of people aged 65+ have a tablet that they use to access the internet. I found this surprising and useful. For this reason one of the other ways in which I aim to ensure that all my target users are able to effectively make use of this product is by ensuring that it has specialist layout that are compatible with large resolutions of laptops and PCs (resolutions vary) as well as tablets (baseline could be iPad Air which is 820px * 1180px) and mobile phones (a baseline could be the iPhone SE with a resolution of 375px * 667px). (My source for resolutions is the chrome web browser in its dev-tools section). This way I can ensure that my website can be accessed by commuters on a mobile, and by people of all ages at home with laptops, PCs and tablets.

Due to the target users I am prioritising some features over others. For one example I am prioritising creating a range of difficulty settings over making a mobile chess clock component in order to best meet my casual user's needs.

Efficient

I would like to divide efficiency into two distinct parts: user interface efficiency and overall program time efficiency.

I aim to make the user interface efficient as I believe that this goes hand in hand with being easy to learn. For example I have minimised the number of clicks that the user will need to make in order to begin playing chess. They can click play, change 1 or 2 configurations by clicking radio buttons and begin the game unhindered. They only need to make 2 intuitive clicks in order to move a piece. Additionally conformation popups are kept to a minimum.

The program has also had multiple considerations factored in to maximise efficiency. For instance I am using web sockets for the super-low latency (as well as the highly practical full duplex connection and ability to implement user vs user games is needed). I am also factoring in time complexity with all of the algorithms I employ. For instance I am using pruning and caching to improve the efficiency of my minimax variant algorithm. Additionally the computer is only given a limited time to think (for example 5 seconds max, if needed, difficulty dependant) as the user may become bored if the computer took as long as an average user to make a move. This allows the user to always be playing the game.

I may in future add keyboard shortcuts but I don't currently deem it necessary or a priority.

Error tolerant:

With regard to error tolerance I have used some of the tips provided in the article to passively improve my application error tolerance without more validation and input sanitation. For example I decided to implement the difficulty input field in the program config form using radio buttons with medium as a default. This means that it is impossible to select an invalid difficulty or not provide a difficulty. I also took the tip of making everything easily reversible by adding a 'back to config' hyperlink in the chess game page so that difficulty etc. can be easily changed without going back to the main menu and restarting.

With some fields like the duration of the timer if a timer is preferred by the user I will need validation as mentioned earlier. I will use a presence check, a type check to ensure the input is of type integer and a range check such as timer in range $15 \leq t \leq 120$. When the submit button on the form is clicked, if this validation fails I will use the industry standard and give the box with invalid input a red outline along with red text to indicate the input requirements.

With regards to moves made by the user on my chess widget. I will talk about it in more depth later but validation is needed. For instance I must ensure that the first square that the user clicks on contains one of their pieces, then I must ensure that the second square is one that the user could move to. I will highlight the legal squares a piece, one selected, can move to with a green dot to eliminate ambiguity.

This validation will be quiet failing as if the user clicks on an invalid first square nothing will happen and if the user clicks on an invalid second square then the piece will only become unselected. No error messages will be used as I believe that this way is most intuitive due to convention.

Easy to learn

One of the reasons that this user interface is easy to learn is that it follows the convention of other chess programs with regards to how the user inputs a move. For example the same method, including highlighting legal moves, is used on Lichess and chess.com which I examined as part of my research. I was impressed by how intuitive it felt when I experimented with these websites.

Additionally by using a chess program convention in my interface design I will reduce the learning curve for new users that wish to start using my program. In this way I am providing an interface in

the way that the user expects it. This interface also maps well onto both PC and mobile touch screen device.

Engaging

I have aimed to make my user interface pleasant and satisfying to use by not making it too complicated. I have tried to reduce the level of noise and needless complexity on the screen and aimed to logically block together information for the user. For instance I put the submit button at the end of the form as users complete forms top to bottom and are less likely to read them properly if the button is at the top. I also used this principle of distinct blocks on information in my chess game play page as information such as previous moves and pieces left was grouped together with clearly allocated, non-overlapping space. They were also at the bottom as they are less important than the chess board and the concede button which should catch the users full attention.

I also plan to make my website visually appealing. I have already in the plan included an image in the home page and centred all the content to look appealing. In the final version I intend to have a different background colours for the webpages (this detail was abstracted away in my plans and so left white). I also want to give the chess board more of a wooden look or even better give the user the ability to alter the look in the pre-game configuration or their account setting. Here is an example of the wooden look I would like to add.



Future plans to maximise how engaging the website is include:

- Adding sound effects for moving a piece and taking an opponents piece (inspired by interview and Lichess research)
- Adding proper animation to show the piece actually moving from one square to another rather than just appearing in the selected square. This would help make the computer's move more clear as it would show what piece had moves and where as a moving object is more catching to the eye.
- The image on the title slide could be replaced with a chess board showing a rapid replay of a series of example matches on loop.

Further Problem Analysis At a Technical Level:

I thought it would be prudent to investigate the technical aspects and details of the problem of making a computer that can play chess. Without doing this I will struggle to decide the appropriate algorithms to use as well as the limitations of my solution.

Chess is a game with complete information as there are no hidden zones (e.g. hands in poker). This is important as it means that both players can use game theory and assume that the opponent is rational in order to try to predict how the opponent will react to each of the moves that they are considering. This thinking ahead is what human players do to identify the best move.

Chess is a game that uses 6 types of pieces:

- Pawn
- Rook,
- Knight
- Bishop
- King
- Queen

Each of these pieces come in 2 varieties: black and white. I will need to incorporate this into my program, both to mark which player owns and controls each piece as well as to properly display the chess board to the end user.

A game of chess is played on a board that is 8*8 and starts with each player having 16 pieces.

Most chess moves obey a set of rules that can be effectively modelled with vectors. Each chess piece can be attributed a position vector and then a series of movement vector if the chess board is treated as a vector space. This is a useful abstraction as it provides a mathematical and easy to program way of describing how a piece can move from wherever they are.

Each piece has a set of movement vectors by which it can move. Ignoring the effect of check on legal the legal moves of a given piece for now, each piece can make usually perform each of these moves so long as the square that the piece would move to is empty or contains an enemy piece and so long as this vector doesn't cause the piece to jump over an allied piece.

This is not always true as there are exceptions. For instance the knight can jump over pieces, in addition the pawn is able to move forward 2 places on its first move only and takes pieces in a different direction to how it moves.

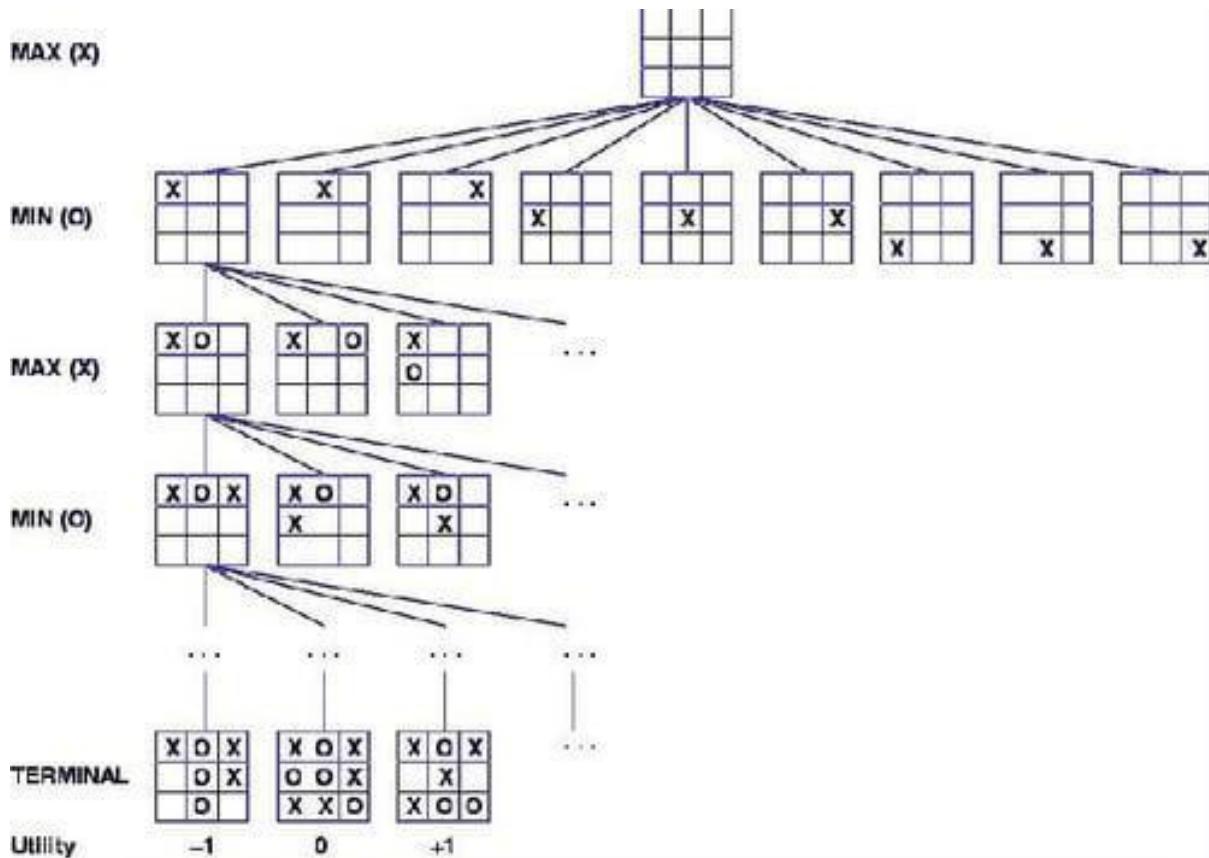
It is a useful generalisation to say that a chess move causes the square where the moving piece was to be empty and the square which it is moving to, to now contain the new piece. Assuming that every move has this simple structure and that no other squares are affected is an oversimplification but makes the problem easier to tackle

As such I don't aim to add some more complex moves which violate this general idea of a move. These include:

- En passant
- Castling
- Ascension (a pawn becoming a queen)

In addition to analysing the problem of representing a chess game, I must try to investigate how a computer could play chess. In 1913 a man named Ernst Zermelo said: "As long as we are able to describe a game as a finite tree, we can find the optimal play". Describing a game as a tree is a useful abstraction for an algorithm as the current game state can be

A game like tic tac toe can easily be described as a finite tree, if the user goes first then this narrows down the number of possible games from $9!$ ($360,000$) to $8!$ ($40,000$).



(diagram source: <https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/>)

This means that all the possible games that could result from a tic tac toe game can be described as a small finite tree that can easily be searched in its entirety by the computer. This process is a recursive and involves exploring the decision tree down to the terminal nodes (games that are won, lost or drawn). These nodes are then given a utility and the algorithm propagates back up the tree. By assuming that the user will act rationally it can take the move that maximises the best outcome that it can guarantee even if the user plays optimally. This brute force process which explores all the paths a game could take in its entirety is called the British Museum Algorithm

This will not work for chess. There are around 10^{123} possible games of chess. This number is far too large for any super computer to apply the British Museum Algorithm. For context there are only 10^{80} atoms in the universe. This means that if every atom in the universe acted as a chess computer and processes chess board states at a rate of 1 per millisecond since the Big Bang, only 1 100 trillionth of

the work (10^{-14}) would have been completed today. This highlights the issue with the exponential nature of possible chess games. Due to this combinatorial explosion, the problem is intractable.

As a result of this, I will discuss a more feasible algorithm that can tackle this problem later in this document called the minimax algorithm.

Design Objectives:

Here are my design objectives. These are based on my success criteria and should serve as a series of specific technical requirements for my solution in order to meet the success criteria.

As the size of the problem is significant, I have focused on the aspects of the success criteria that are of a high priority or are essential. This should ensure that a high quality product is produced at the end of the iterative development process that is realistic considering time constraints.

Number:	Name	Description	Success Criteria Link
1	GUI: Chess board	I need a graphical user interface that includes a visual representation of a chess board	2-8 inclusive
2	GUI: Chess Board	This chess board should be able to register click events and run a handler function in order to allow the user input their move	2-8 inclusive
3	GUI: Chess Board	The square click handler function should be able to keep track of the state of the chess board with some key variables in order to decide how to action the users click input	2-8 inclusive
4	GUI: chess board	The program should do nothing when an invalid square is clicked. Relevant validation should decide to disregard the click event rather than raise an error.	2-8 inclusive
5	GUI: Chess Board	The chess board should be able to respond to inputs as appropriate by visually changing how it appears to the user to add highlighting and move pieces. I will aim to ensure that every click causes a reaction (visual feedback)	2-8 inclusive
6	GUI: Chess Board	The chess board should be able to display any layout of pieces overlayed onto the squares using either characters or images to show the pieces	2-8 inclusive
7	GUI: Chess Board	The chess board and the pieces must be sufficiently different in color so that white pieces can be seen on a black background	2-8 inclusive
8	GUI: Chess Board	The chess board should be able to show the computers move but adding highlighting and then moving the pieces.	2-8 inclusive
9	GUI: Difficulty Radio Buttons	Radio buttons should be used for the difficulty with the medium button already selected. This will ensure that only valid inputs for the difficulty are given. It will also ensure that at all times, at least one input is given.	1
10	Difficulty Of AI	The move engine (AI adversary) should be able to be limited in how well it can explore the board (measured in either depth or time) to create different difficulty levels.	1
11	Move Engine AI	The move engine should be able to perform a type of tree search to determine scores of child board states and use these scores to decide and return both a move and a score	9, 10
12	Move Engine AI	The move engine should feature some optimisations to allow it produce the same quality of move faster or produce a better move in the same	26

		time. The new optimisations should not change the output of a given search (this is deterministic) but should just improve speed	
13	Move Engine AI	The move engine should be able to learn from its past games in order to improve its decisions and select new moves.	27
14	Move Engine AI	Persistent data storage should be used to enable the AI to retain calculations and cache results from previous games in order to improve and learn.	27
15	Move Engine AI	The move engine should be able to make a good move decision even if none of the terminal nodes are game which are over by using heuristics.	9
16	Move Engine AI	The move engine should be able to understand when a given node represents a board state where the game is over and value this node accordingly. In the late game it should aim to try to steer towards outcomes where it wins.	10
17	GUI: Main Title	Above the chess board there should be a main title which displays a message which changes over time. This should be in large font. It will indicate whose go it is and whether or not a given player is in check. This should also act as the prompt for users to input another move, to make moving a clearer process. This title will also be used to explain when the game is over and who has won.	5, 11, 12
18	Board State Analysis	The chess engine should be able to determine if a given player is in check. This will affect legal moves and therefore the move engine as well as validation of the user's move	11, 28, 29
19	Board State Analysis	The chess engine should be able to determine if the game is over, and the nature of the end to the game (e.g. is it a stalemate or who the winner is)	12, 28, 29
20	Board State Analysis	The chess engine should also be able to accurately determine all the legal moves available to a player for a given board state. This functionality will be relied on to validate the user's move input and for the move engine's search of the decision tree.	28, 29
21	Move engine can access board state analysis	The move engine (adversarial AI) should be able to access functions to recognise the legal moves of a given board state or whether or not the game is over in order to properly construct and traverse its decision tree.	9, 10, 26, 29
22	Chess game: A data structure should be used to display previous moves	An array should be used to hold as vectors all the previous moves made in the game.	15
23	GUI: A widget to output previous moves	A table should be used to display the previous moves in plain English and using chess squares (not vectors).	15
24	GUI: buttons	I will need buttons on my user interface to concede and restart the game as well as functions to facilitate this	31, 32
25	GUI: Pieces taken function and widget	A function should be able to use the piece layout for a chess board state in order to determine which pieces are missing (have been taken). There should be a widget in the user interface to provide this as a graphical output to the user.	30
26	API connection	I should have a fast and efficient way to convey data between the server and the client to allow for real time inputs and outputs and to prevent	17

		lags. The connection structure should also allow all the relevant data to be exchanged while not being too complex.	
27	GUI: Reloading unfinished chess games	I have decided that this feature can be completed without a login system. When loading up, the website should make a request to the server for the up to date chess game rather than load data for a starting chess game. This should allow the user to continue a chess game where they left off if they close and then reopen the tab	19
28	Reloading unfinished chess games	The server should be able to read the cookies within a browser when client server session starts. If an unfinished game exists in the cookie (or the cookie contains an id to lookup in a database) then this game will be loaded into memory for the duration of the client and servers session. When the session ends, a cookie should be created in the users browser that contains or links to the unfinished game for future use by the server.	19
29	GUI: Mobile device access	The GUI should be designed so that the core components (main title and chess board) can be correctly displayed in a variety of aspect ratios and orientations to allow for the use of tablets and phones (maximise accessibility and convenience)	25
30	GUI Accessibility	The website will aim to practise clarity and accessibility of the content over visual appeal when deciding colour schemes and layouts. This means that there should be sufficient color contrast to identify individual widgets and text from the background as well as highlighting and pieces on the board. This includes ensuring that the colours used for highlight are not a common color pair for color blindness.	24
31	Move Engine Heuristics	The move engine (adversarial AI) should be able to perform a fast heuristic analysis of the board. This will provide a score that one player aims to minimise and the other maximise. It should be based on pieces remaining and piece layout. It will be used by the move engine to play chess in the early and mid-game.	9

Some changes to the design:

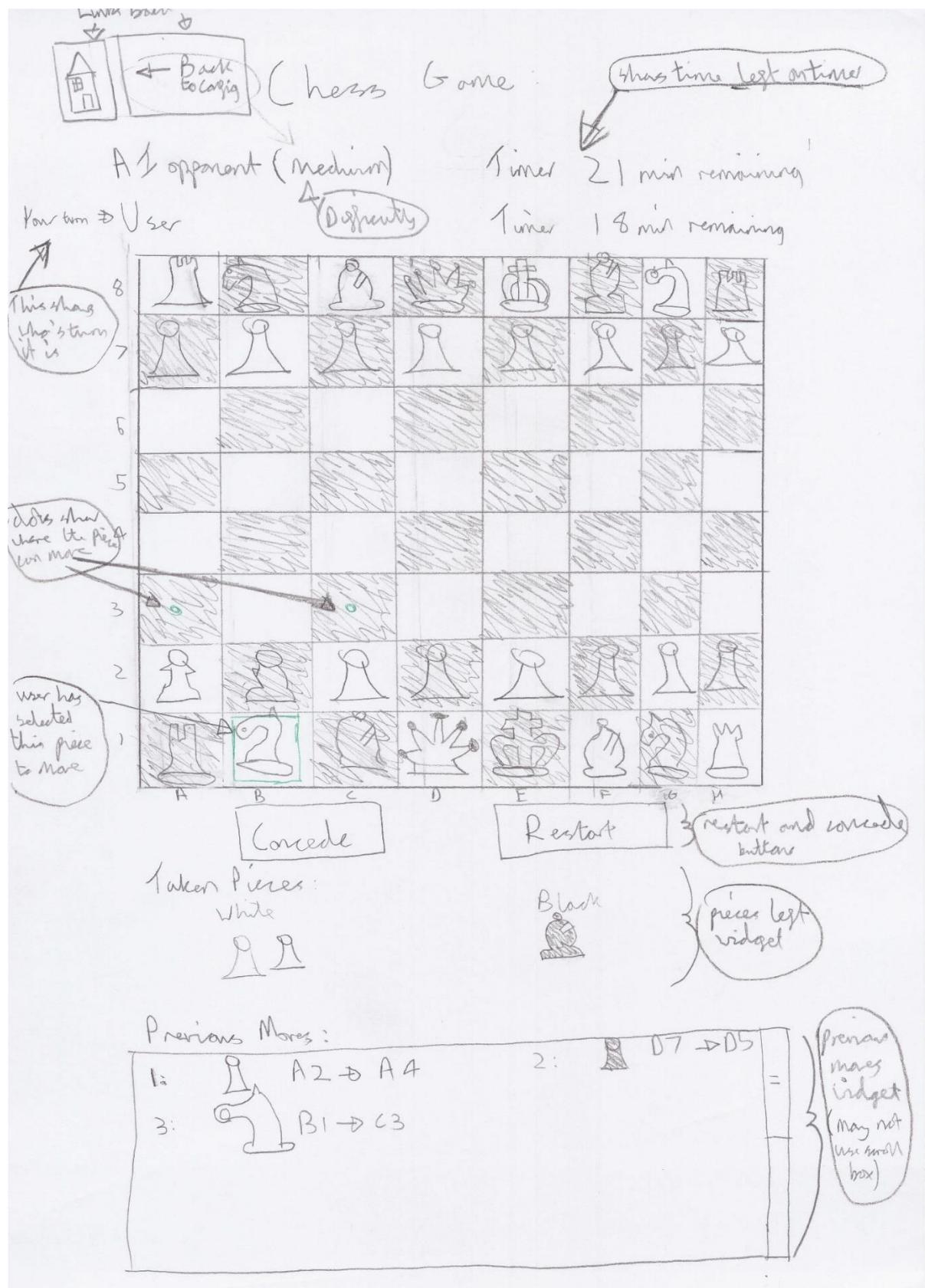
After deciding my technical design goals to achieve the high priority part of the success criteria I have made some changes to my design for the proposed solution.

I found the initial attempt to design the program very productive as it help me understand how the problem was trackable by computational techniques. My initial attempt to break down the problem helped me see which components would be needed to tackle the problem.

I will now use the new ideas from my design objectives to revise to structure of the proposed solution and how it will be implemented. One reason for doing this is that laying out all the design goals in a table has helped me see that there are more feature to implement than I first thought. In order to achieve the most important components of the success criteria and deliver a working final prototype that meet the stakeholder needs I must be realistic about the time available.

For example, my first design iteration included a multi webpage user interface that included login capabilities. This was not a high priority feature in the success criteria and so it is not realistic to implement in the limited time frame.

However, the process of determining how the problem was solvable by computational techniques and designing the user interface has been useful. For instance I have decided that I will use the minimax function to implement the chess AI. I have also decided that of all my drawings of the user interface, I like the following design the most. As a result I will model my single webpage user interface on it:



Usability Features to be Used in the Solution:

I made use of my research into usability and how to ensure that it is maximised within my program within the first part of my design. I liked these ideas and will not change them. I have integrated them into my design objectives.

Effective

I have aimed to make my proposed solution effective by trying to maximise the range of aspect ratios that the final website will work with. This will include tablets as my research shows that they are more popular among older stakeholders. I will aim to make the final product work on phones. This will allow me to meet the needs of stakeholders who are on the go or otherwise play chess on their phones.

This links to design objective 29 and 30

Efficient:

I will maximise the efficiency of my program by using a fast API connection and by making improvements to the minimax function. This should help avoid symptoms of inefficiency like lag which could negatively effect the user experience.

This corresponds to design objectives 26 and 12

Error Tolerant:

I will ensure that my final product is error tolerant by adding validation to avoid invalid inputs causing invalid outputs. I will also use automated unit tests to ensure that the final product is robust and has as few logic errors as possible.

This corresponds to design objective 4

Easy to Learn and Engaging:

I have used my research from the analysis section and decided that I will try to emulate the highlighting feature as well from competitor websites like chess.com. This should make the program easier to use and more intuitive as it should allow me to show the user visually the legal moves that are available to them. I will also use the highlighting to show the computer's move.

This should make the program easier to learn and use for both new player (who may not know where all the pieces can move) and players who are used to the user interfaces of competitors like chess.com. Features such as the game being saved for later should also provide convenience and make the game easier to use.

I believe that if I can make the sight easy to use, then there should be nothing to get in the user's way of enjoying and engaging with the chess game. Addition different difficulty setting to allow user to set the write level of challenge should also make the chess games more engaging.

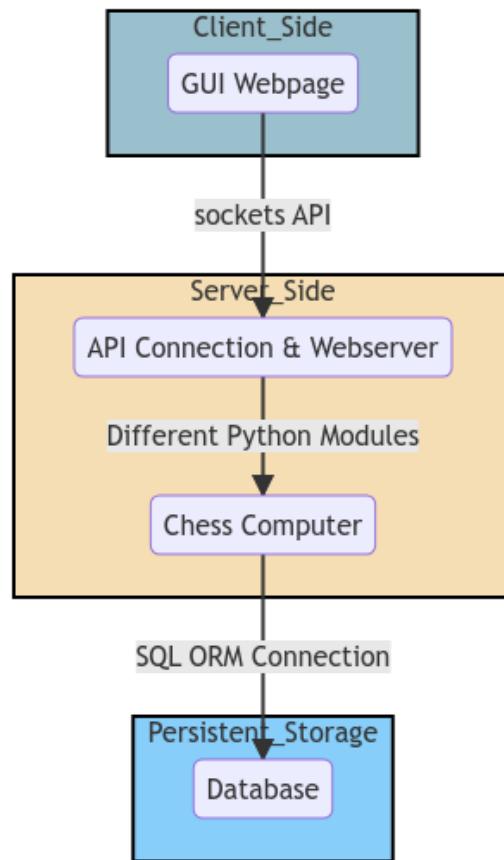
This corresponds to design criteria items:

- 1, 10, and 12-16 for different difficulties and AI challenge
- 5, 23-25 for highlighting as well as other features common to many chess sites

- 27 and 28 for reloading chess games

Proposed Solution Description:

My proposed solution will be a full stack web application. It will have this overall structure:



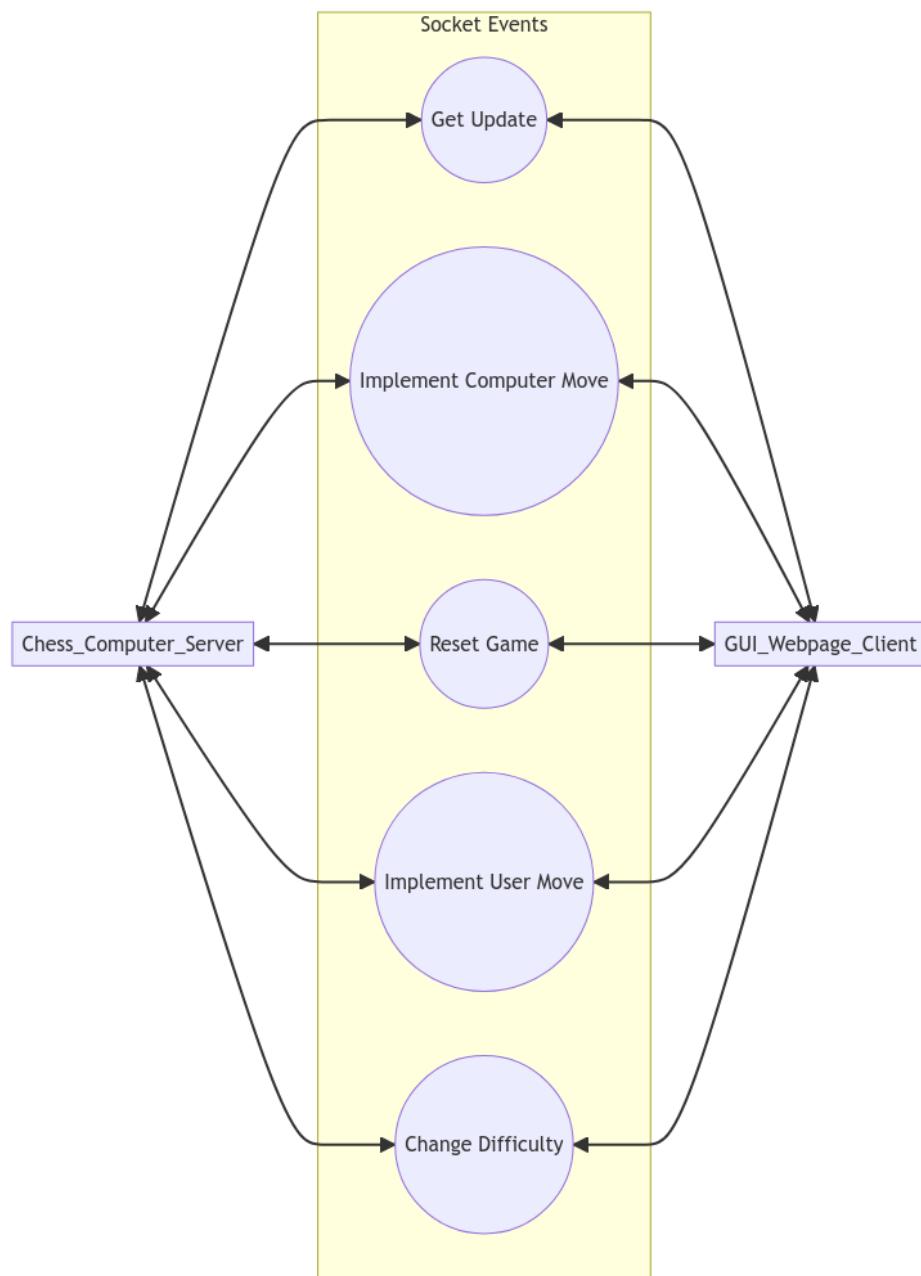
As shown in the above diagram, the solution is a full stack web application. This means that it includes code and logic at both the client and server side.

Front End:

The front end or client side component of the solution will be a combination of HTML, CSS and JavaScript. It will be responsible taking in user inputs and providing outputs. This should be done through a purely graphical user interface (GUI). The client side JavaScript should also perform client side validation of the users inputs in order to ensure that only legal moves are made. The client side JavaScript should also use asynchronous programming by using JavaScript promises to connect to the server and exchange data. The client side JavaScript code is intended to be “dumb”. By this I mean that it won’t have any algorithms that can help it understand what is happening in this game of chess (e.g. if the game is over or what the legal moves are). Instead it will request act only as a wrapper around the logic already created in the chess computer, providing a User Interface on top of this.

Server Side API Connection and Webserver:

The next layer in my application is the server's webserver and API gateway. This server side code will setup the HTTP webserver that will send the relevant HTML, CSS and JS files to the browser. In addition, appropriate WebSocket routes will be created to allow the server and the client side chess



GUI to communicate.

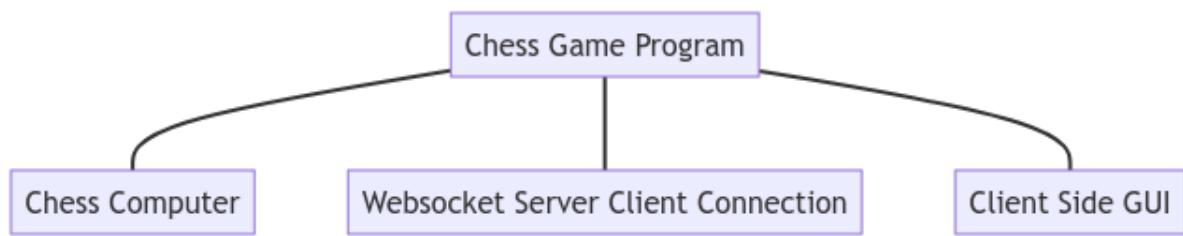
The webserver and API gateway should act as an intermediary between the client side JavaScript and the functions of the chess engine. It should include server side validation to ensure that the data sent to the server by the client is valid (so for instance the user move must be valid). As part of

defensive design to minimise unexpected behaviour and validation needed I will aim to minimise the amount of data that the client and server exchange.

Chess Computer:

The chess engine will take the form of a server side python program that is able to keep track of a chess game, provide the legal moves, and recognise check and game over as well as determine the adversarial AI's move. This will be completed by dividing the chess computer into various sub programs.

Benefit of Decomposition:



The benefit of this decomposition of these main parts is that each part can be tackled separately and using different means.

The chess computer can be tackled with pure python

The client side user interface can be tackled with HTML and JavaScript

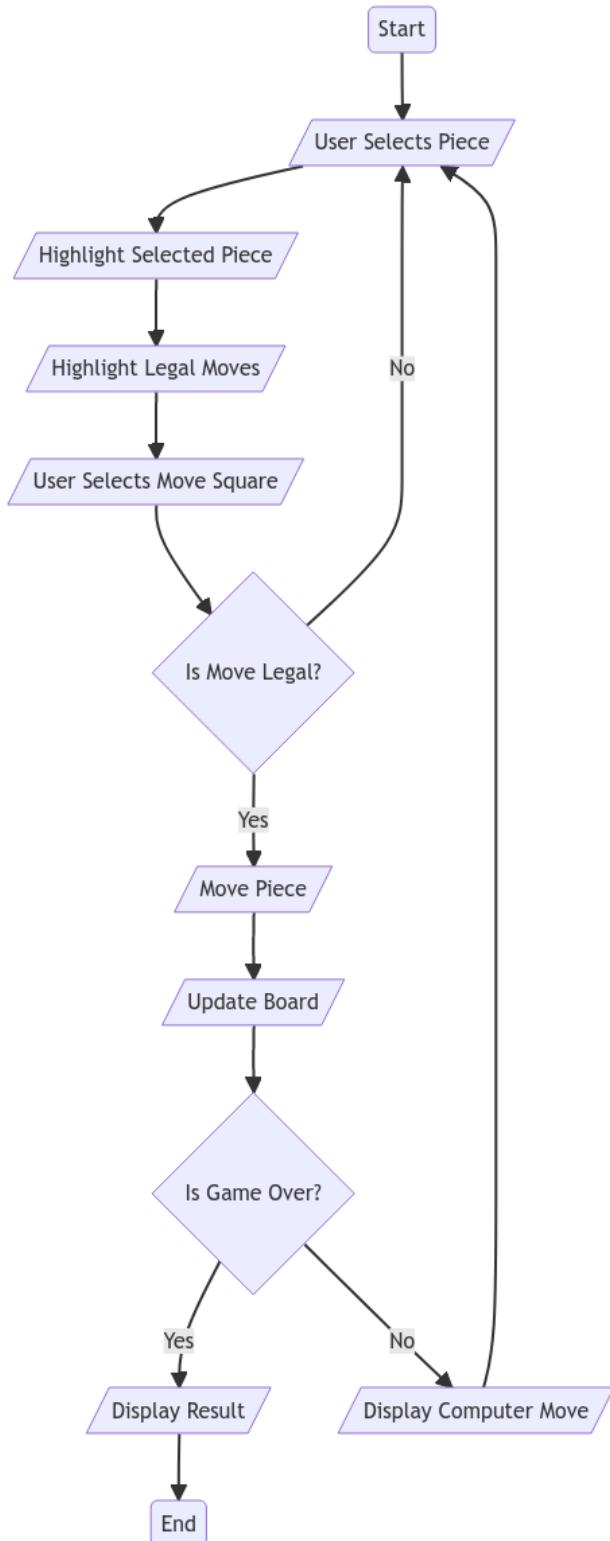
The API connection and webserver can be tackled using the Flask and WebSocket frameworks

Dividing up the program into these parts allows for specialist languages and frameworks to be used to tackle the sub-problem. Additionally different people could work on different sub problems independently if this was a professional project. Because of this, the above decomposition makes the final product easier to implement and should improve its quality in terms of how well it meets the design goals.

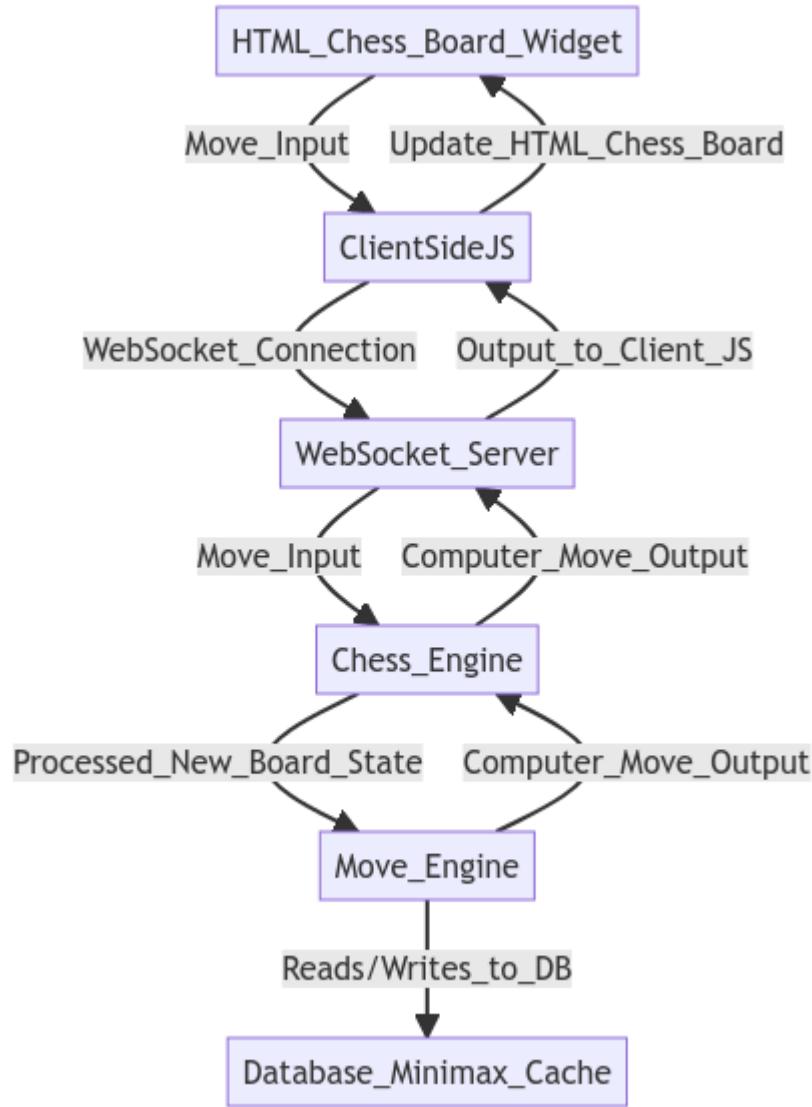
Diagrams of proposed solution:

Here is a flowchart to describe how the user will input a move and play chess against the computer.

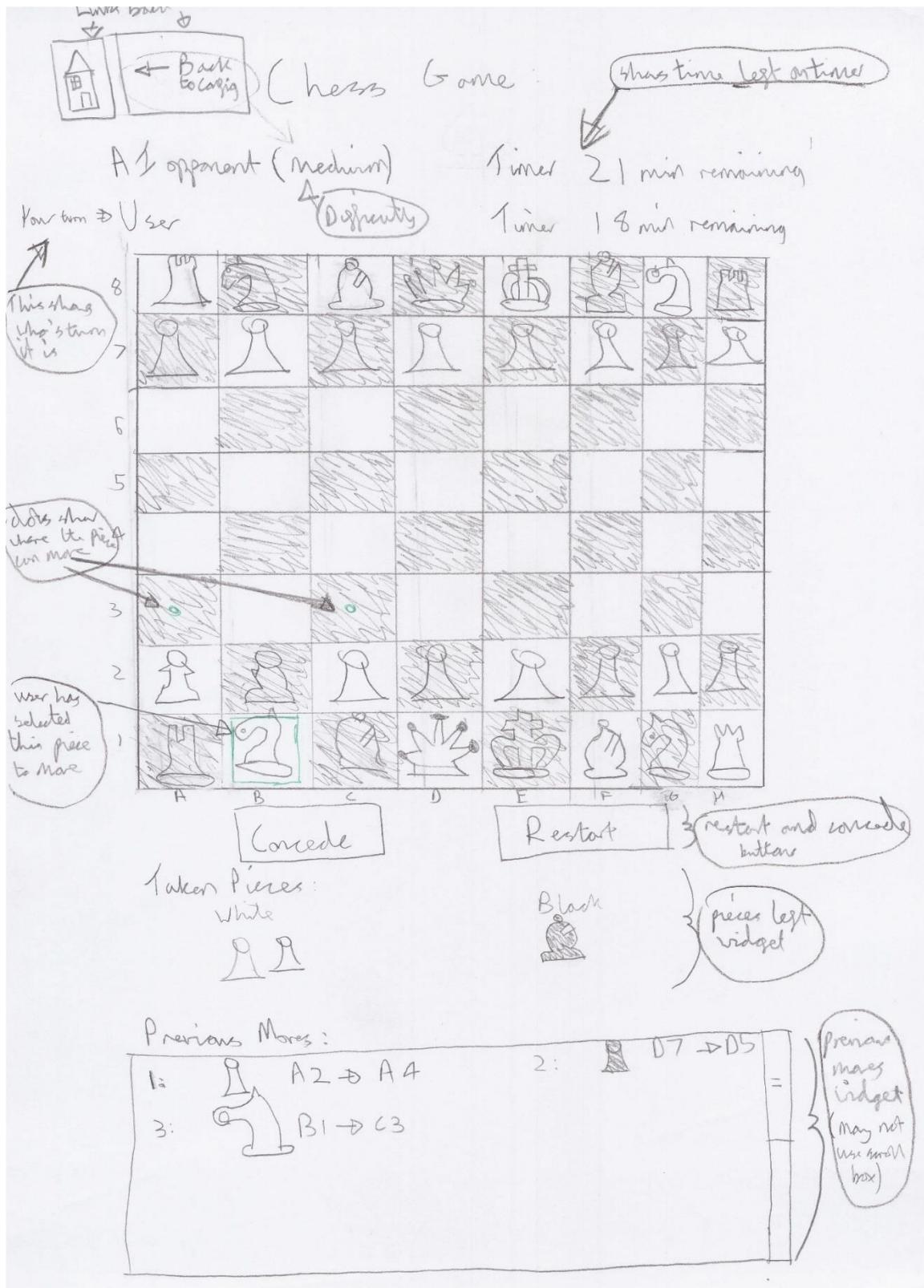
This will all take place on the same page as the application is only a single web page (reduces complex menus that the elderly can struggle to navigate):



And here is a data flow diagram to describe the flow of data within the program when making a move. It shows the layers of the application that sit between the GUI and the database. It shows how a user move input is processed and then a computer move output is produced.



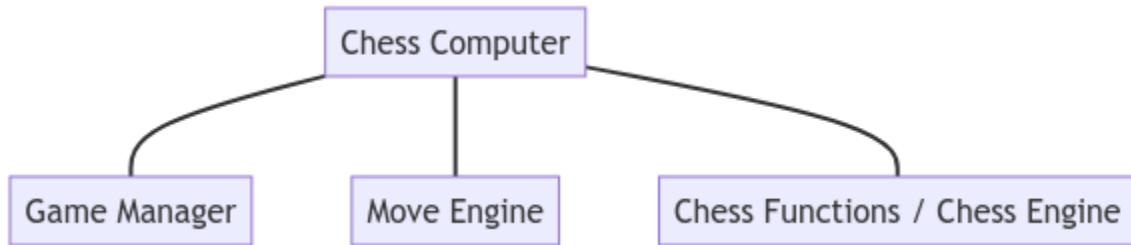
I will use the interface design from my first design iteration as I believe it addresses the various usability requirements that I identified in the first part of my design:



Decomposition of Chess Computer (not algorithms just functions, classes and data structures)

The chess computer is a broad term that I am using to describe the backend logic that runs on the server to allow it to manage a game of chess, including generating the computer's moves.

The chess engine is a large program that can be further decomposed into multiple modules.



The Chess Functions Library or Chess Engine us used to analyse a given chess board state. It will contain functions to use a given board state to determine the possible legal moves, or if the game is over.

The Move Engine component will be responsible for performing a minimax search of a given board state in order to determine a suitable move for the computer to make. As part of this it may interact with the database in order to make use of caching to improve efficiency.

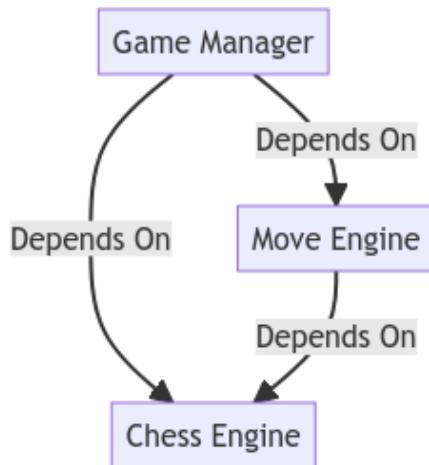
The Game Manager module will act as a wrapper around all of the logic and functionality of the other 2 modules. It will be able to keep track of a whole chess game where as the chess engine can only keep track of a single board state which is a snapshot of a whole game. It will be able to keep track of which players go it is as well as implement moves. It will include validation to ensure that only the right player is able to move on their turn and that they can only make a legal move.

It will make use of the Chess Engine module to keep track of the board state throughout the game. It will use the Move Engine module to decide the move of the adversarial AI that the user will play against. It may also make use of a database in order to save and restore games for the user.

This means that:

- The Chess Engine module has no dependency on the other modules
- The Move Engine module is dependent on only the chess engine module

- The Game Manager module is dependent on both of these modules. It is the way in which the functionality of the other modules is accessed and choreographed together to produce useful functionality.



This decomposition should allow me to more easily achieve my design objectives as a big problem is being tackled with multiple separate sub components with their own responsibilities.

Additionally, the use of a Game Manager module should provide abstraction as it allows other components that facilitate a chess game to access shared logic rather than each interacting with board states and keeping track of turns individually. This should make other components such as the Server Side API Handlers easier to implement as less logic will need to be repeated. In addition the Game Manager module can be tested to show that it is reliable whereas lots of different instances of similar code to facilitate a chess game cannot be easily tested.

Each section should be able to achieve the following design criteria (by numbers):

- Chess Engine: 18, 19, 20
- Move Engine: 10, 11, 12, 13, 14, 15, 16, 21, 31
- Game Manager: 22, 18. Also ties in with requirements around game: (part of objectives 4, 5, 6, 7, 80)

Decomposition of GUI logic and API client server connection:

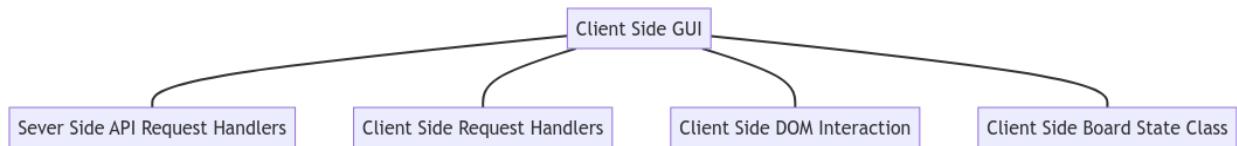
This component will include some logic on both the server and client side. Logic on the sever side will facilitate the API connection between the client and the server. Relevant handler function will call the appropriate functions of the Game Manager module when events are received from the client.

The majority of the code will be client side. It will include HTML and CSS to define the relevant components (tags) and there layout / style on the webpage.

JavaScript will also be used to provide interactivity, logic and connect to the server:

- It will include logic to affect the DOM object. The Document Object Model is an object in JavaScript that allows for the HTML content that the browser renders to be interacted with. This will allow inputs to be received and outputs to be graphically displayed.

- JavaScript will also be used to create a class to represent all the data and logic from a given board state. This will include all the logic to decide how user inputs will be processed, when server requests will be made and how the returned value will be outputted.
- JavaScript will also be used to request data from the server. For example this could allow for a user move to be implemented and the new board data to be used to update the board object and the front end. This will involve the use of a framework like WebSocket that will provide an API: Application Program Interface with which the client and server can exchange data.



Each component should be integrated together to achieve the specified design objectives.

For instance, when a user clicks a piece, the DOM interaction components should register the square click and then send he data about which square is clicked to the Board State Class. This will decide what output are appropriate.

For example it could decide to change the highlighting in response. The DOM interaction components would then use a corresponding function to update the visual highlighting colours that the user sees.

Alternatively that Board State class could decide that the user has inputted a valid move. If so both client and server side API handlers would be used to communicate this input to the Game Manager module in the Chess Computer. Then the new board state and all its relevant data would be sent back to the client side Board State class via both client and server side API handlers. The new data contained within the Board State class (e.g. a moved piece) would then be provided as a visual output to the user using the DOM interaction component to affect the HTML.

HTML and CSS will be used to dictate the content of the webpage (the tags that JavaScript will interact with) and its layout. I will keep styling to a minimum and focus instead on the interfaces usability. I will also use CSS to ensure that all the text is big enough and clear to read. The layout of the webpage, as defined by CSS, should also be such that it should be compatible with PCs, tablets and mobile phones.

Each module will be used together to achieve the design objectives that are relevant to the GUI. Here are the design objectives that this component should be able to achieve:

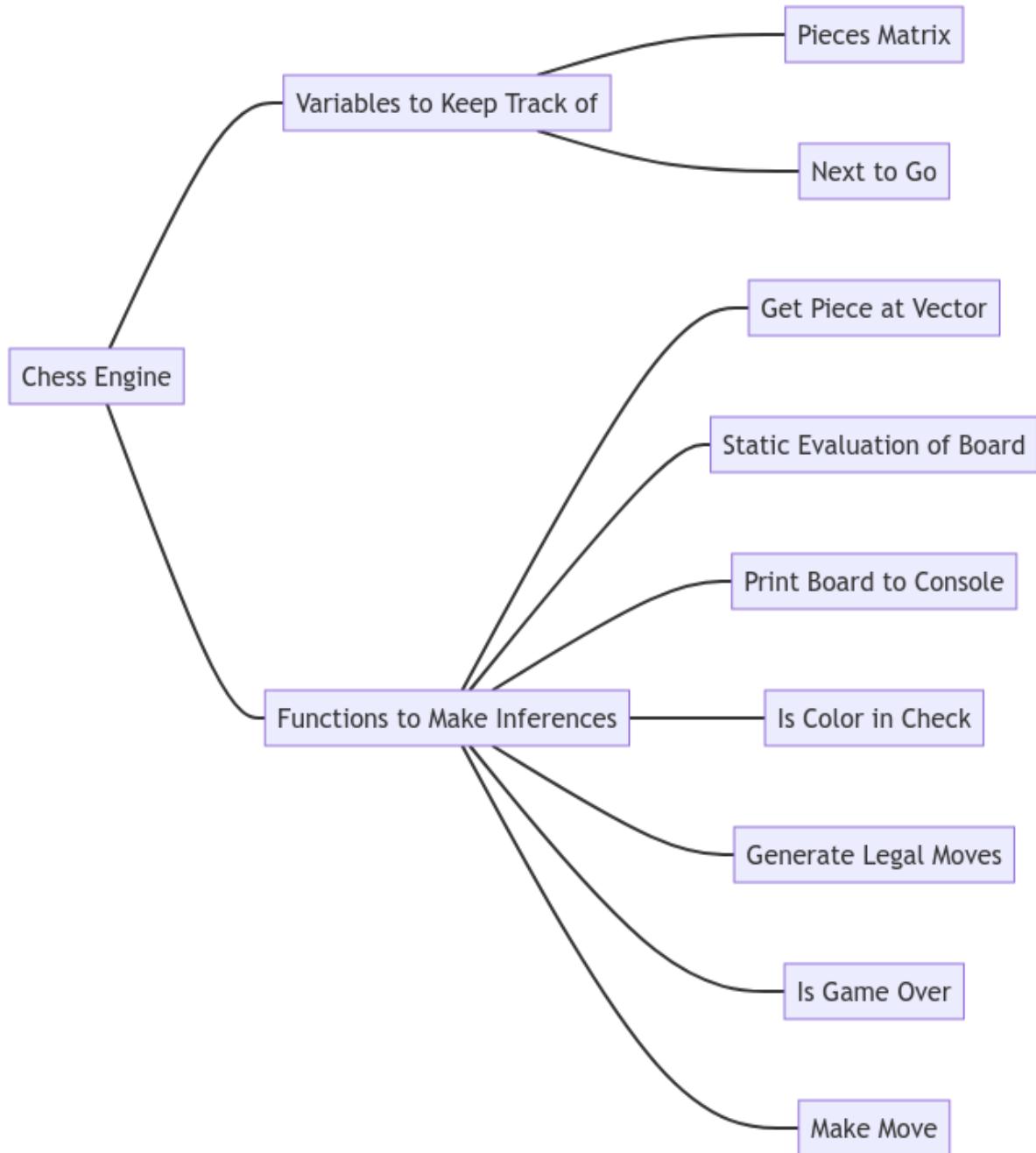
1, 2, 3, 4, 5, 6, 7, 8, 9, 17, 23, 24, 25, 26, 27, 29, 30.

Chess Computer: All major algorithms

The Chess Engine / Chess Functions:

The Chess Engine component should be able to make various inference to determine information about a chess board. It should also be possible to implement moves to get a new resultant board state. This will be needed in the minimax search completed by the Move Engine as well as to implement user's moves.

Here is a further decomposition of the Chess Engine to identify its functionality and properties.



Due to the nature of the problem, there are various reasons why OOP (object oriented programming) is needed. These include:

- I need to keep track of many different board states at once in a minimax search
- I have already grouped the responsibilities of the module into variables and function that use these variables. This can be easily translated to properties of a Board_State class and method that reference these properties
- The module will be easier to develop and test if OOP is used as variables and methods form different board states will be grouped withing self-contained objects and so will not interfere with each other (as global variables might).

So here is the design for a class that will possess this functionality:

Board_State
<pre>-next_to_go: string -pieces_matrix: list of lists -pieces_matrix_frequency: dictionary -check_encountered: bool -three_repeat_stalemates_enabled: bool +add_pieces_matrix_to_frequency_table() : None +is_3_board_repeats_in_game_history() : bool +generate_pieces_taken_by_color(color: string) : list of strings +print_board() : None +get_piece_at_vector(vector: Vector) : string or None +generate_all_pieces() : list of strings +generate_pieces_of_color(color: string = None) : list of strings +color_in_check(color: string = None) : bool +generate_legal_moves() : list of tuples +is_game_over_for_next_to_go() : tuple of bool and string or None +static_evaluation() : int +make_move(from_position_vector: Vector, movement_vector: Vector) : None +database_hash() : int</pre>

Due to the fact that I need to keep track of multiple different board states and child board states at once when performing a minimax search, I decided that it would be best to make the objects produced by the Board_State class immutable. This means that the make_move will return a new child board state object rather than mutate the existing object. This prevents me needing to make copies of the board state before making moves whenever I want to preserve the original board state. It also prevents any properties being set erroneously so long as the make_move and constructor methods are used correctly.

The pieces_matrix variable will be a 2 dimensional array of pieces to represent the board layout. It can be accessed by a position vector by converting the position vector to a row and column in the 2d array:

Row = 7 - vector.j

Column = vector.i

Piece = pieces_matrix[row][column]

The most important function in this module is the generate legal moves function which is essential for determining if the game is over (no legal moves) as well as performing the minimax search and validating the users move.

It returns an array of moves. Each move is encoded as a position vector (for the piece moving) and a movement vector (the direction it moves in).

In order to represent these vectors, I will create a vector class which will include both the vector's data: integers for the i and j components as well as functions relevant to vectors.

Here is the pseudocode and UML diagram for my vector class:

```

FUNCTION sqrt(x):
    RETURN x**(1/2)
END FUNCTOIN

FUNCTION square(x):
    RETURN x**2
END FUNCTOIN

CLASS Vector():
    // 2d vector has properties i and j
    i: int
    j: int

    // code to allow for + - and * operators to be used with vectors
    FUNCTION add_vectors(self, other):
        ASSERT isinstance(other, Vector), "both objects must be instances
        of the Vector class"
        RETURN Vector(
            i=self.i + other.i,
            j=self.j + other.j
        )
    END FUNCTOIN

    FUNCTION subtract_vectors(self, other):

```

```

        ASSERT isinstance(other, Vector), "both objects must be instances
of the Vector class"
        RETURN Vector(
            i=self.i - other.i,
            j=self.j - other.j
        )
END FUNCTOIN

FUNCTION multiply_vectors(self, multiplier: int | float):
    RETURN Vector(
        i=int(self.i * multiplier),
        j=int(self.j * multiplier)
    )
END FUNCTOIN

// check if a vector is in board
FUNCTION in_board(self):
    // Assumes that the current represented vector is a position
vector
    // checks if it points to a square that isn't in the chess board
    RETURN self.i in range(8) and self.j in range(8)
END FUNCTOIN

// alternative way to create instance, construct from chess square
@classmethod
FUNCTION construct_from_square(This_Class, to_sqr):
    // Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1,
3)
    to_sqr = to_sqr.upper()
    letter, number = to_sqr

    // map letters and numbers to 0 to 7 and create new vector object
    RETURN This_Class(
        i=ord(letter.upper()) - ord("A"),
        j=int(number)-1
    )
END FUNCTOIN

// this function is the reverse and converts a position vector to a
square
FUNCTION to_square(self) -> str:
    letter = chr(self.i + ord("A"))
    number = self.j+1
    RETURN f"{letter}{number}"
END FUNCTOIN

FUNCTION magnitude_and_unit_vector(self):

```

```

        magnitude: float = sqrt(square(self.i) + square(self.j))
        // unit_vector: Vector = self.__mul__(1/magnitude)
        // RETURN magnitude, unit_vector

    RETURN magnitude
END FUNCTOIN
END CLASS

```

Vector
+i: float +j: float
+add_vectors(other: Vector) : Vector +subtract_vectors(other: Vector) : Vector +multiply_vectors(multiplier: float) : Vector +in_board() : bool +magnitude_and_unit_vector() : float +construct_from_square(to_sqr: string) : Vector +to_square() : string

I will use Vector objects to describe the position of each piece and how it is moving. These objects are used as parameters for some methods like make_move and as return values for generate_legal_moves.

In order to implement the static_evaluation and generate_legal_moves functions I am going to further decompose the problem by using a pieces class.

I will aim to make a class for each chess piece. The resulting objects will be able to represent both black and white pieces. Each object will have 2 main responsibilities:

- Given where it is on the board, determine its own utility
- Given where it is on the board, ignoring check, determine the movement vectors by which it can move.

Each piece object will contain a starting utility and a 8*8 matrix of additional utility. Depending on where it is on the board, the value matrix may increase or decrease the pieces utility. The value matrix is meant to describe how some pieces are more valuable if they are in a certain area of the board (e.g. knight near the centre of the board). The sum of all these utility values will be used by the board state class to find the static evaluation of the board as a whole.

In addition each piece will contain a function that will return a list of movement vectors. Each piece will use the pieces matrix (board positions) and its position vector (where it is) to identify where it can move to. All the different pieces will have different ways of doing this:

- For instance the knight will simply need to check the contents of the square it would move to for each of its 8 possible vectors. The king is similar

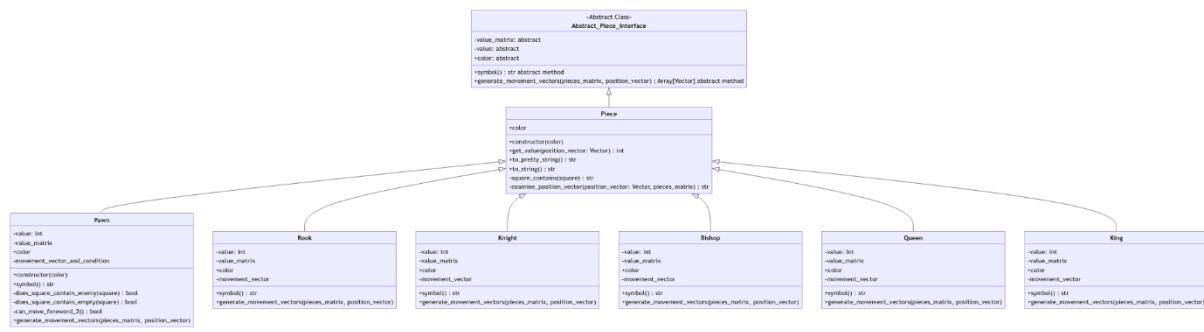
- The pawn needs to use rules around the contents of its neighbouring squares (e.g. are they empty or do they contain an enemy) to decide if it can move forward or diagonally. It additionally may need to overload the constructor of the pieces class to include a flag to determine if it has moved yet. This will affect it can move forward 2
- Pieces like the rook, bishop and queen must iteratively test increasing length vectors in a given direction until a non-empty square is hit. Then they cannot go any further in that direction and should start iteratively checking vectors in other directions.

Due to the fact that each piece contains some logic that is different (where it can move) and some logic that is shared (calculating its utility score) I have decided to use parent classes. All logic that is generic and shared will be implemented in a parent Piece class. In addition, an abstract class called Abstract_Piece_Interface will be used to ensure that every child piece has the relevant methods and properties.

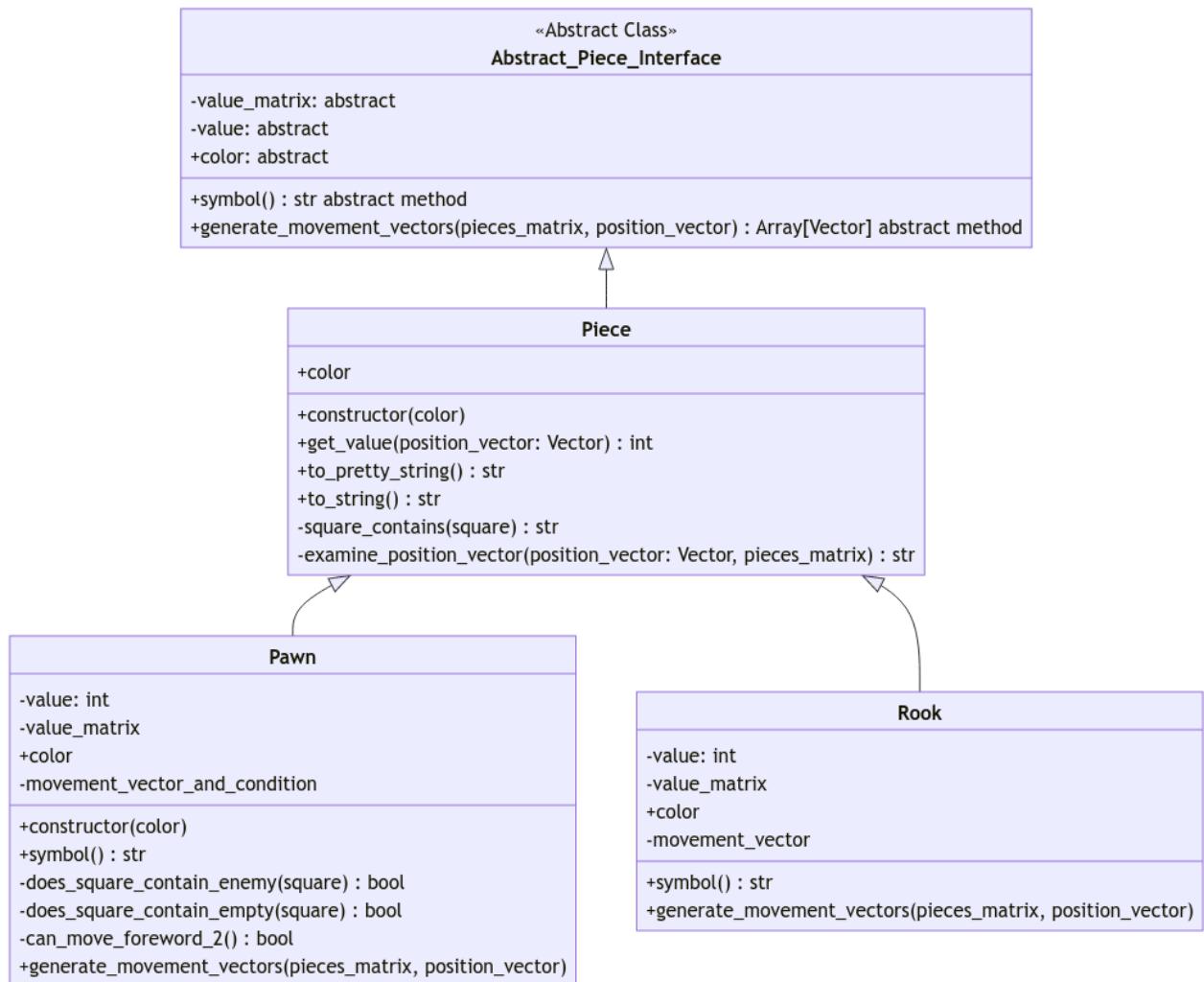
By having the Piece class Inherit from the Abstract_Piece_Interface class and by having each individual piece class (e.g. Pawn) inherit form Piece I will ensure that:

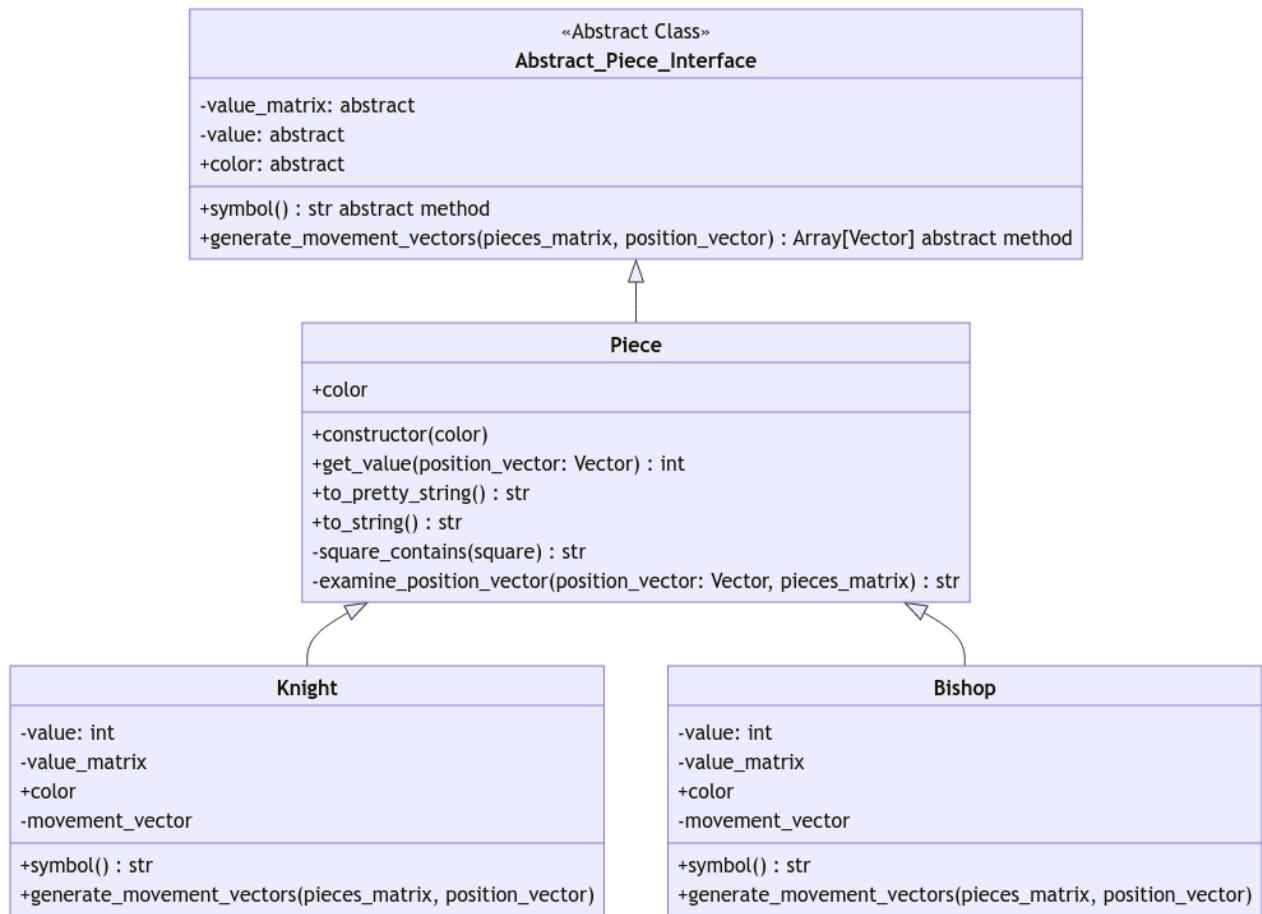
- I keep repeated logic to a minimum
- Every piece class has the same external interface and so can be used in the same way.

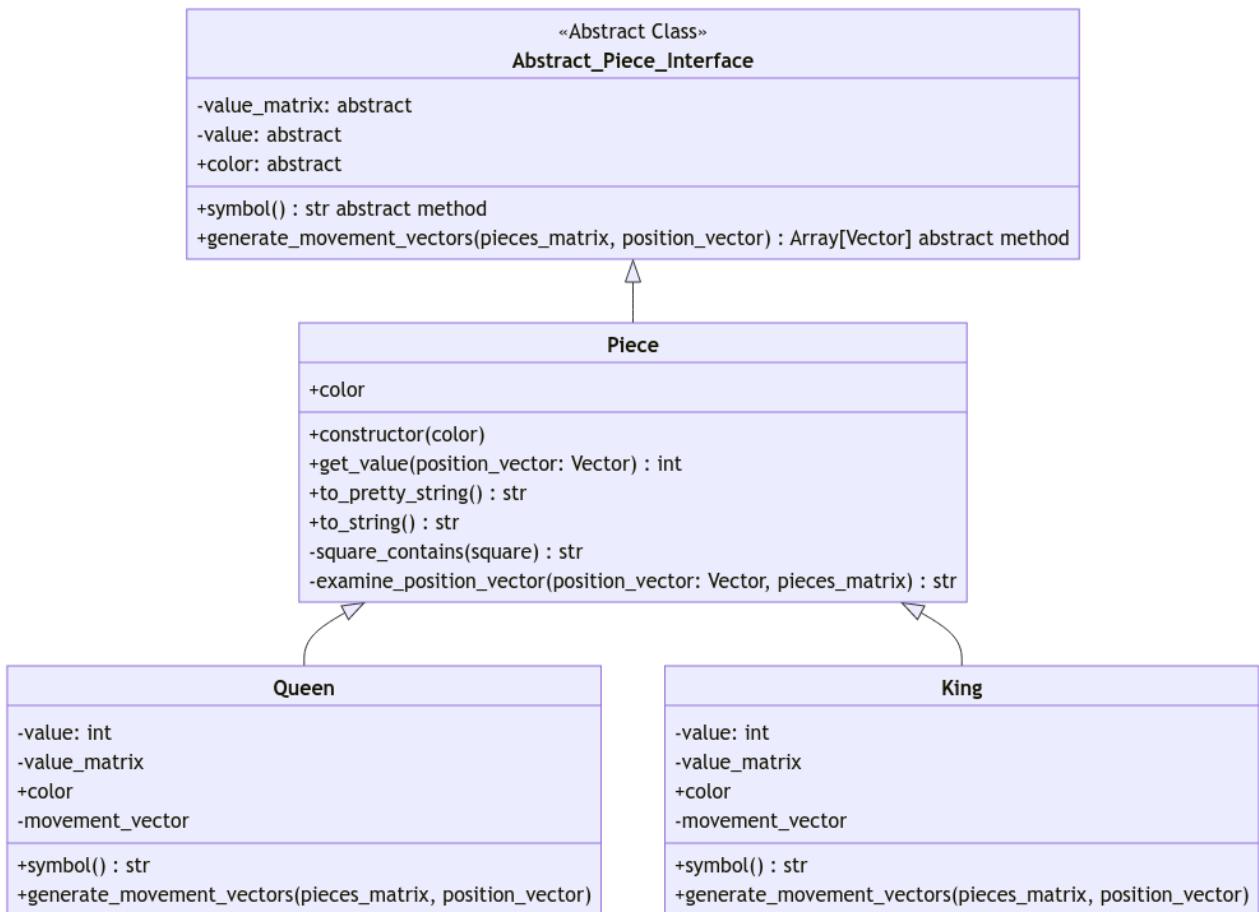
Here the whole UML diagram for these classes:



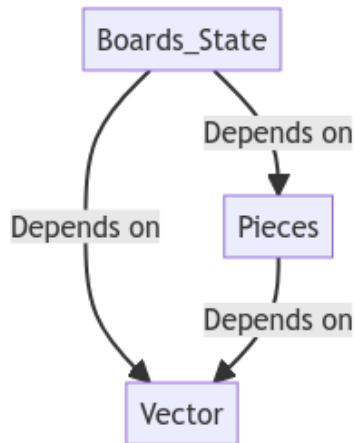
This is too small to see but should illustrate the structure I am trying to show. I will now show 3 sub-diagrams to make the details easier to read:







Using the abstract base class will mean that an error will be raised if any child class is initialised without all the necessary attributes and methods. This should ensure that the interface between the classes is the same and should help prevent logic errors later.



In addition, the use of the pieces and vector sub components allows me to further break down the problem through layers of abstraction. At the bottom is the Vector class, this is at the lowest level and is performing individual vector calculations. This module can be tested to ensure it is robust. Then the Pieces component can abstract all that logic and simply deal with vectors to try to determine the moves a given piece can make. Then at the highest level, all of the previous detail is

abstracted. All the board state class needs to do is collate all of these individual pieces moves together into a big array and filter it to remove child board states that cause check. This reduces the complexity of a function like generate legal moves within the board state class. This is because the details for each which way each piece can move are abstracted as decomposition has been used to create a subcomponent to solve this issue.

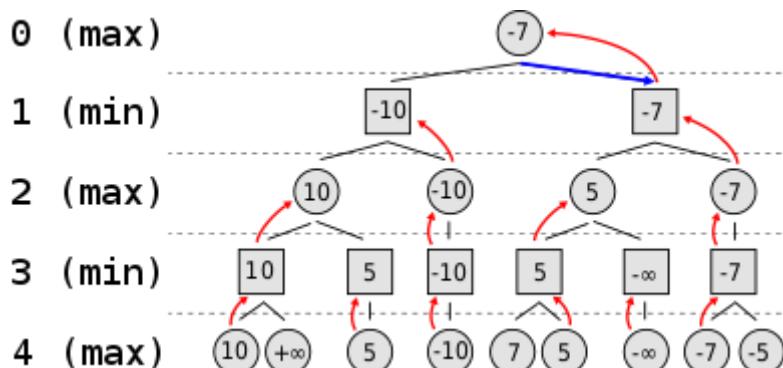
This should make writing the otherwise tricky legal moves and static evaluation functions easy. All the other functions in the Board_State class should either be easy to write (e.g. get piece at vector) or should able to make use of the generate legal moves function (e.g. the is_game_over function).

The Move Engine algorithm:

The move engine should act as a large and complex function that is composed of many sub functions. It takes as input a board state and returns as output a best move and a score for how good that move is estimated to be.

As discussed earlier, I cannot perform the British Museum Algorithm on a chess decision tree as the number of computations needed is so high that the problem is intractable. Instead I will use a recursive function called the minimax function to explore the decision tree to a given depth, use heuristics to estimate each players advantage at the terminal nodes and then back propagate to find the best move. I am calling the tree that I am using to represent the possible chess games from a given board state a decision tree as each node represents the decision of which move to pick.

Here is an example of this process. I have here a decision tree which I created to represent the problem with all the details of chess abstracted. The tree has been explored to a given depth using the legal moves of parent board states. Then the terminal nodes have been given a score:



(source: <https://en.wikipedia.org/wiki/Minimax>)

We can see that in the above decision tree we have simplified the problem by illustrating each node as only having 2 children.

The terminal nodes are given a score, this could be infinity if the game is over or a number estimating how favourable the game is using a heuristic. One player aims to maximise this score and the other aims to minimise it. In this way, chess is a zero-sum game as one players gains are another

players losses. We can see that all chess board where the maximising player can decide a move are represented with circular nodes and the minimisers moves are represented with square nodes.

As complete information is assumed, the idea of picking a move instead becomes deciding which of the child board states is most favourable for you to create. Because of this, we can see that at depth 3, the values assigned to the rectangular nodes are the lowest scores of the child nodes. This is because these are the moves that are best for the maximiser.

Because the assumption is made that each player is rational and as there is complete information, the maximising player can anticipate that when presented with these board states, the minimising player will act in these ways. As a result, the maximising player at depth 2 can attribute nodes at depth 3 with the value that the minimising player will rationally pick. Then the maximising player picks the highest score among these nodes.

In doing this, the final score and move is the best that can be guaranteed, assuming the opponent plays optimally. This means that when a player makes a sub optimal play, this leads to a better outcome than expected for the opponent.

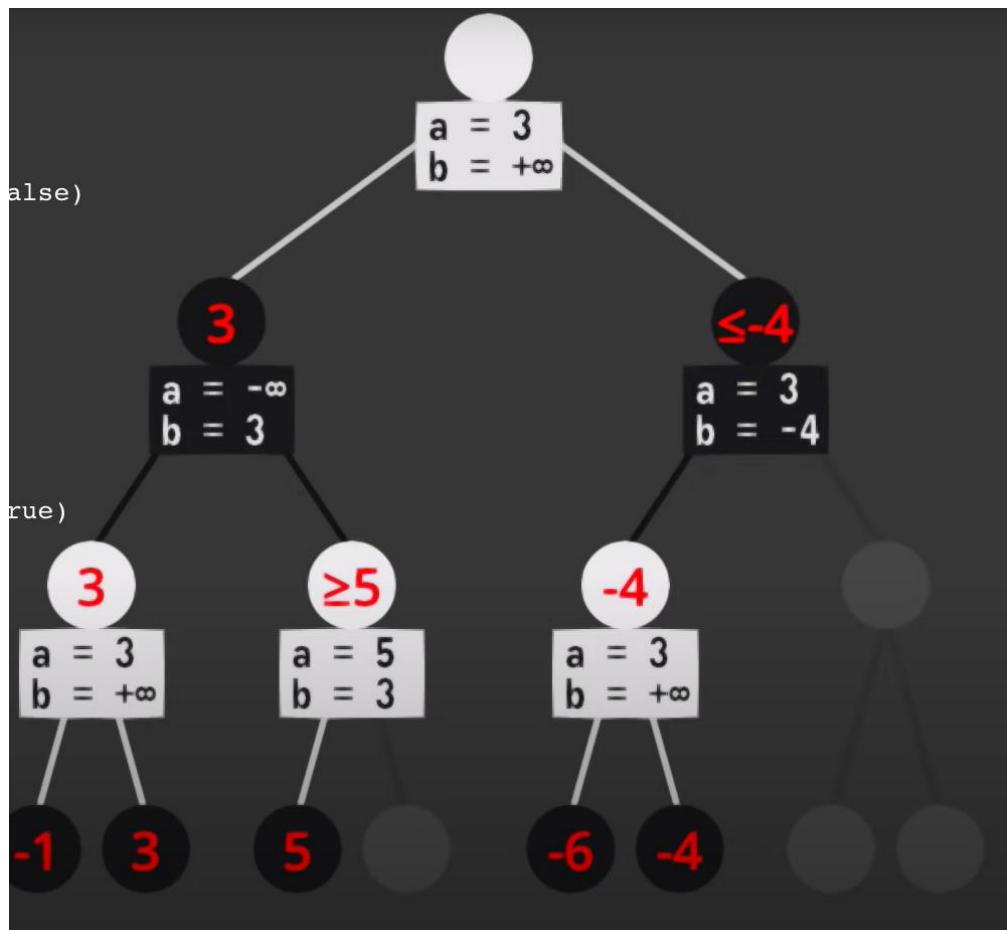
In my program, the static evaluation function from the Chess Engine module will be used to provide the heuristic that will evaluate the terminal nodes. The computer will be the minimiser and the user will be the maximiser.

This algorithm will allow me to tackle the game of chess as it provides a heuristic solution that is good enough. The algorithm is still slow as it has an exponential time complexity. We can define the branching factor at a given stage in the game to be the number of legal moves available from a given board state and the depth to be the number of moves ahead that we look. Now the number of static evaluation of terminal nodes is proportional to the branching factor to the power of depth. This means that the function has an exponential time complexity $O(2^n)$. This means that the solution algorithm cannot be used to a high depth as it has a non-polynomial time complexity.

I will add a series of optimisation to my program. The simplest of optimisation to add. It works to reduce redundant calculations by “pruning” parts of the tree to prevent unnecessary computation.

Here is a screenshot from the following video that illustrates this concept well:

<https://www.youtube.com/watch?v=l-hh51ncgDl>



The minimax function now uses 2 additional arguments that are passed on to recursive calls. Alpha represents the best that the maximiser can already achieve and beta represents the best that the minimiser can already achieve. Another way of looking at this is to say that alpha is what the maximising player can achieve without further exploration and beta is the best that the minimiser has to allow the maximiser to achieve. Because of this, if alpha is greater than or equal to beta, it means that further exploration of child nodes is not needed as the opponent minimiser would never let you the maximiser get to this path as it is already too good for them to allow.

This is shown in the above diagram. Nodes are labelled with their alpha and beta values. In this case, white is the maximiser and black is the minimiser. We can see from the inequalities on some nodes how the alpha beta pruning indicated that a certain branch of the tree can be “pruned” and no longer explored.

I also intend to add caching to improve the efficiency of my minimax algorithm. To do this I will store the parameters and return value of each minimax function call as an entry in a database (structure explained later). This will allow me to prevent redundant searches in future. Cached results could be retrieved from the database if they match the search being performed exactly or if they match a recursive function call within a search (e.g. cached depth 1 results could be used as part of a depth 2 search later).

To maximise efficiency I will not store the whole board state in my database, instead I will store its SHA256 hash as a hexadecimal string. Due to the size of 2^{256} I can make the safe assumption that there is a 0 chance of collision. Therefore each board state can be encoded as a small binary string.

To write the result of a minmax search to the cache I will delete all previous cached results that are at a lesser depth and then add the new search to the database.

To retrieve data from the cache (for a search of a given board state at a given depth) I will hash the board state and then query the table for matching hashes. This will be efficient as the table will be indexed by hash. The depth of any stored records will then be checked, if it is insufficient then it will be ignored. Otherwise, if the cached search has a depth greater than or equal to the needed depth, the database entry will be returned.

Further optimisation that I intend to add to the algorithm to improve efficiency or efficacy include:

- Variable depth search. This would model the variable depth feature of Deep Blue by allowing certain parts of the tree to be searched to an additional depth if check is encountered. These branches are likely to be more important to search as they could help to cause or avoid checkmate.
- Pre-sorting the moves best to worst (using a heuristic or a recursive depth n-2 search) before performing a recursive minimax search of each move to improve pruning
- Concurrency: The minimax function can be made concurrent by breaking up the legal moves to examine into sub arrays. Then many threads could be used to concurrently perform minimax searches of these segments of the legal moves at once.
- Insurance policy + self-interrupting: This is the idea that a minimax call at a lesser depth takes exponentially less work. This means that it only takes around 1/30th more work to do a depth 1 then 2 then 3 search of a board state than to just start with a depth 3 search. This means that the algorithm could be structured to keep searching for a given amount of time and then stop itself and return the best result. Unfinished searches could be added to the cache allowing for additional search time in future to improve the quality of the result.

The implementation of the self-interpreting feature will allow me to create a minimax function that will search the decision tree for a given amount of time.

- This will allow for more than 2 difficulty settings: depth 1 and depth 2 as a greater depth will likely be infeasible in the context of the limited thinking time of a chess AI.
- This will remove the inconsistent thinking time of setting the AI to a given depth. This problem arises as the number of legal moves can vary greatly from 1 to 50. This can cause great variation in how long a minimax search to a given depth can take
- This idea of a certain amount of time to explore will also allow me to achieve my design objective (13 and 14) of making a learning adversarial AI. This will be possible by combining caching with a set exploration time. This will allow the AI to continue where it left off when it encounters a board state again (like the opening positions) and search in greater depth to improve the quality of its move.

Here is the pseudocode for a basic minimax function. It will be able to calculate the computers best move from a given board state. It features alpha beta pruning to improve its efficiency.

```
// default arguments allow for the function to be provided only the board
state and be able to find the computer move
FUNCTION minimax_search(board_state: Board_State, depth: 2, alpha=-
infinity, beta=+infinity, is_maximizer=False)

    // base case
    over, winner = board_state.is_game_over_for_next_to_go()
    IF depth == 0 or over THEN
        // use static_evaluation for score
        best_score = board_state.static_evaluation()
        best_move = None

        RETURN best_score, best_move
    END IF

    // recursive case
    // set initial variables
    best_move = None
    IF is_maximizer:
        best_score = - infinity
    ELSE:
        best_score = + infinity
    END IF

    // iterate through legal moves and get child board states
    legal_moves = board_state.generate_legal_moves()
    FOR EACH move in legal_moves
        // get child board state
        from_v, move_v = move
        child_board_state = board_state.make_move(from_v, move_v)

        // recursive call to evaluate child board state
        child_score, child_move = minimax_search(child, depth - 1, alpha,
beta, not is_maximizer)

        // depending on if player is maximizer affect if a lower or higher
score is better
        // update alpha / beta and update best score and move variables as
necessary
        IF is_maximizer:
            alpha = max(alpha, child_score)
            IF child_score > best_score:
                best_score = child_score
                best_move = child_move
            END IF
        ELSE:
            beta = min(beta, child_score)
            IF child_score < best_score:
```

```

        best_score = child_score
        best_move = child_move
    END IF
END IF

// pruning, stop exploring more legal moves
if beta <= alpha:
    break
END FOR

// return the score and move corresponding to the best child
RETURN best_score, best_move

END FUNCTION

```

Game Manager:

This module is to be in charge of managing a chess game. It should be able to keep track of whose turn it is and include validation to ensure that only valid moves are made.

Due to the fact that a web server may have many sessions with clients at once, I will need to be able to store the data for multiple games at once. Additionally I want to be able to save all data relevant to the game as binary to be restored later. These factors mean that using global variables, that can be accessed by all function in the game manager module, to keep track of the game is not feasible. Instead I will again use OOP.

Here the game class that I will use when creating my console chess prototype:

Game
<pre> -player_color_key:dict[str, int] -time:int -echo:bool -board_state:Board_State -move_counter:int -move_engine:Move_Engine_Timed -game_history_output:tuple +_init__(time:int=10, user_color:str="W", echo:bool=False) : void +create_row(move_data: dict) : str +print_game_history() : void +make_move(move:str, time_delta:int, future_utility:int) : void +implement_user_move(from_square:str, to_square:str, time_delta:int=None, estimated_utility:int=None) : void +check_game_over() : list[bool, str, str] +implement_computer_move(best_move_function:Callable=None) : void </pre>

And here is the game class that I will use to facilitate the chess games that are played on my webpage:

Game_Website
<pre>-difficulty: str -player_color_key: dict -board_state: Board_State -move_counter: int -move_engine: Move_Engine_Timed -move_history_output: list +__init__(difficulty: str, user_color: str) +get_time() : int +add_move_to_history(move: list[Vector, Vector]) : void +make_move(move: list[Vector, Vector]) : void +implement_user_move(move: list[Vector]) : void +check_game_over() : list[bool, str, str] +best_move_function(time: int) : list[int, list[Vector, Vector]] +generate_computer_move_description(move, previous_board_state) : dict[str: Vector, str: Piece] +implement_computer_move_and_report(time=None) : dict[str: Vector, str: Piece]</pre>

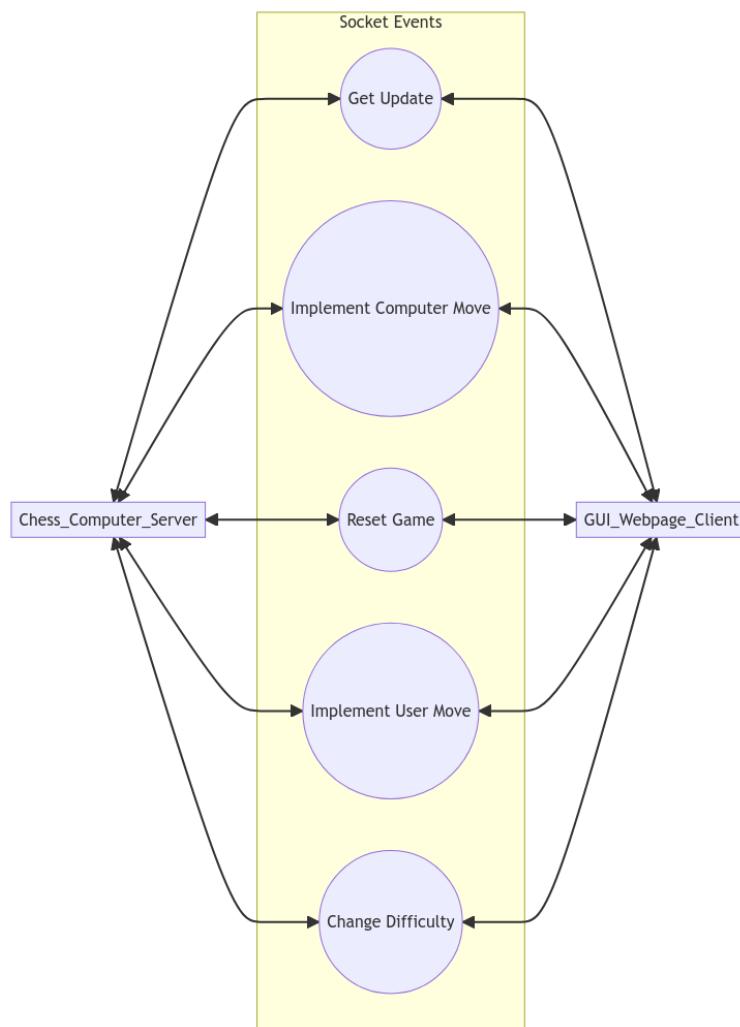
The main difference between them is that the website game class has the capacity to play at different difficulties. In addition it doesn't need to build a table of data to give running commentary of the game in the console. Instead it creates a dictionary of data about the computers move that is sent to the webpage to allow for the computer's move to be displayed.

The implement user move and computer move methods in both classes contain validation to ensure that it is that player's turn to go. In addition, the user move is checked to ensure that it is a valid move.

It is also important that this object can be pickled as this will allow it to be serialized to binary data which can be stored in a database. The game object's binary can then be read from the database and deserialized in order to reload the game for the user to finish.

GUI logic and API client server connection:

As previously mentioned, I will use a system where data is exchanged between the client and server using the WebSocket framework.



There will be 5 main WebSocket events that will be used to exchange data between the client and the server.

After the server updates its python Game_Website object, a standardised block of JSON data will be sent to the client to provide the client all the data needed to update the GUI.

Here is an example of this json data for the start of a chess game (opening positions):

```
{
  "difficulty": "medium",
  "next_to_go": "W",
  "game_over_data": {"over": false, "winning_player": null, "victory_classification": null},
  "legal_moves": [[[0, 1], [0, 1]], [[0, 1], [0, 2]], [[1, 0], [-1, 2]],
                 [[1, 0], [1, 2]], [[1, 1], [0, 1]], [[1, 1], [0, 2]], [[2, 1], [0, 1]],
                 [[2, 1], [0, 2]], [[3, 1], [0, 1]], [[3, 1], [0, 2]], [[4, 1], [0, 1]],
                 [[4, 1], [0, 2]], [[5, 1], [0, 1]], [[5, 1], [0, 2]], [[6, 0], [-1, 2]],
                 [[6, 0], [1, 2]], [[6, 1], [0, 1]], [[6, 1], [0, 2]], [[7, 1], [0, 1]],
                 [[7, 1], [0, 2]]],
  "pieces_matrix": [[[{"B", "\u265c"}, ["B", "\u265e"]], [{"B", "\u265c"}, ["B", "\u265e"]]]}
}
```

```
[ "B", "\u265d"], [ "B", "\u265b"], [ "B", "\u265a"], [ "B", "\u265d"], [ "B",
"\u265e"], [ "B", "\u265c"]], [[ "B", "\u265f\ufe0e"], [ "B",
"\u265f\ufe0e"], [ "B", "\u265f\ufe0e"], [ "B", "\u265f\ufe0e"], [ "B",
"\u265f\ufe0e"], [ "B", "\u265f\ufe0e"], [ "B", "\u265f\ufe0e"], [ "B",
"\u265f\ufe0e"]], [[null, null], [null, null], [null, null], [null, null],
[null, null], [null, null], [null, null], [null, null]], [[null, null],
[null, null], [null, null], [null, null], [null, null], [null, null],
[null, null], [null, null]], [[null, null], [null, null], [null, null],
[null, null], [null, null], [null, null], [null, null], [null, null]],
[[null, null], [null, null], [null, null], [null, null], [null, null],
[null, null], [null, null], [null, null]], [[ "W", "\u265f\ufe0e"], [ "W",
"\u265f\ufe0e"], [ "W", "\u265f\ufe0e"], [ "W", "\u265f\ufe0e"], [ "W",
"\u265f\ufe0e"], [ "W", "\u265f\ufe0e"], [ "W", "\u265f\ufe0e"], [ "W",
"\u265f\ufe0e"]], [{"check": false, "pieces_taken": {"B": [], "W": []},
"move_history": []}]
```

Here we can see that many data points have been pre-calculated and provided to the client UI. This includes an array of legal moves as vector pairs and an 8*8 2d array of the pieces matrix. Sub strings like this "\u265f\ufe0e" correspond to a chess pieces character that is not in the standard ASCII character set but instead in the larger UTF-8. To ensure compatibility, it has been encoded by the python json module so that the JSON is ASCII compatible as it is sent between the client and the server. JSON is a useful abstract data structure to use as modules exist to convert python objects to JSON and JSON to JavaScript objects. This means that this standard makes transferring data between the client and the server easier.

On the server side I will need the following functions handler functions to process data receives on each of these 5 events. These handler functions will cause some mutation to the game object held within the server client session and then return JSON data to update the client side about the game.

Here is the pseudocode for this:

```
FUNCTION generate_game_update_data()
    game = get_game_in_session()

    next_to_go = game.board_state.next_to_go
    difficulty = game.difficulty

    legal_moves = list(game.board_state.generate_legal_moves())
    legal_moves_serialised = serialize_legal_moves(legal_moves)

    pieces_matrix = game.board_state.pieces_matrix
    pieces_matrix_serialised = serialize_pieces_matrix(pieces_matrix)
```

```

next_to_go_in_check = game.board_state.color_in_check()

over, winning_player, victory_classification = game.check_game_over()

game_over_data = {
    "over": over,
    "winning_player": winning_player,
    "victory_classification": victory_classification,
}

pieces_missing = {}
FOR EACH color IN ("B", "W")
    pieces_missing[color] = list(map(
        serialize_piece,
        game.board_state.generate_pieces_taken_by_color(color)
    ))
END FOR

move_history = game.move_history_output

payload = {
    "difficulty": difficulty,
    "next_to_go": next_to_go,
    "game_over_data": game_over_data,
    "legal_moves": legal_moves_serialised,
    "pieces_matrix": pieces_matrix_serialised,
    "check": next_to_go_in_check,
    "pieces_taken": pieces_missing,
    "move_history": move_history,
}
}

RETURN payload
END FUNCTION

@bind_socket_handler("get_update")
FUNCTION get_update()
    result = generate_game_update_data()
    RETURN result
END FUNCTION

@bind_socket_handler("implement_computer_move")
FUNCTION implement_computer_move()
    game = get_game_in_session()
    assert game.board_state.next_to_go == "B"
    assert not game.board_state.is_game_over_for_next_to_go()[0]

```

```
move_description = game.implement_computer_move_and_report()

set_game_in_session(game)

outgoing_payload = generate_game_update_data()
outgoing_payload["computer_move_description"] = move_description

RETURN outgoing_payload
END FUNCTION

@bind_socket_handler("reset_game", respond=False)
FUNCTION reset_game()
    old_game = get_game_in_session()
    difficulty = old_game.difficulty
    new_game = Game_Website(difficulty=difficulty)

    set_game_in_session(new_game)
END FUNCTION

@bind_socket_handler("implement_user_move")
FUNCTION implement_user_move(incoming_payload)
    game = get_game_in_session()

    assert game.board_state.next_to_go == "W"
    assert not game.board_state.is_game_over_for_next_to_go()[0]

    user_move = tuple(
        deserialize_move(
            incoming_payload["user_move"]
        )
    )
    game.implement_user_move(user_move)

    set_game_in_session(game)

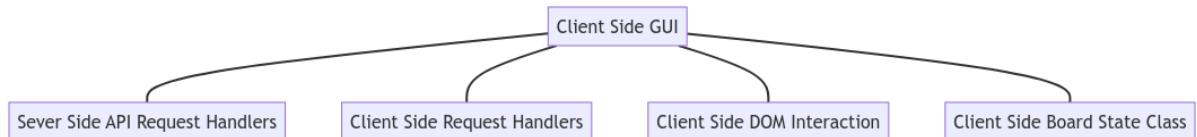
    RETURN generate_game_update_data()
END FUNCTION

@bind_socket_handler("change_difficulty", respond=False)
FUNCTION change_difficulty(incoming_payload)
    game = get_game_in_session()

    new_difficulty = incoming_payload["new_difficulty"]
    game.difficulty = new_difficulty

    set_game_in_session(game)
```

```
END FUNCTION
```



(diagram repeated)

The client side JavaScript need 3 main sets of functions. As preciously discussed, these are:

- Client side request handlers, these functions should handle making requests to the sever and receiving the servers response to return.
- Client side DOM interaction, these functions will affect the DOM (HTML displayed) to display outputs
- Client side board state class, this will contain all the data from the server about the current bord state and will manage the logic or receiving user inputs and making the relevant requests to the server vie the handlers

The Client side request handler will consist of the following functions:

```

ASYNC FUNCTION external_get_update()
  PASS
END FUNCTION

ASYNC FUNCTION external_implement_computer_move_and_update()
  PASS
END FUNCTION

ASYNC FUNCTION external_reset_game()
  PASS
END FUNCTION

ASYNC FUNCTION external_implement_user_move_and_update(from_vector,
  to_vector]()
  PASS
END FUNCTION

ASYNC FUNCTION external_change_difficulty(new_difficulty)
  PASS
END FUNCTION
  
```

Each of these asynchronous functions will send the relevant data to the server and then wait for a response which will then be returned. The functions will be asynchronous as this allows other code to be executed while this function is idle (waiting for server response).

I will need the following functions to affect the DOM and display outputs to the user:

```
FUNCTION create_board_widget()
    PASS
END FUNCTION

FUNCTION get_square_at_vector(v)
    PASS
END FUNCTION

FUNCTION add_pieces(board)
    PASS
END FUNCTION

FUNCTION add_highlighting(board)
    PASS
END FUNCTION

FUNCTION clear_board()
    PASS
END FUNCTION

FUNCTION update_main_title(board)
    PASS
END FUNCTION

FUNCTION update_pieces_taken(board)
    PASS
END FUNCTION

FUNCTION set_selected_difficulty(board)
    PASS
END FUNCTION

FUNCTION set_widget_move_history(board)
    PASS
END FUNCTION

FUNCTION update_board_widget(board)
    update_main_title(board)
    add_pieces(board)
    add_highlighting(board)
    set_selected_difficulty(board)
    update_pieces_taken(board)
```

```
set_widget_move_history(board)
END FUNCTION

create_board_widget()

let board = new Chess_Board(
    update_board_widget,
    update_main_title
)

FUNCTION handle_radio_button_click(radio_button)
    PASS
END FUNCTION

FUNCTION handle_square_click(i, j)
    board.handle_square_click([i, j])
    update_board_widget(board)
END FUNCTION

FUNCTION reset_button_click()
    board.reset_game()
END FUNCTION

FUNCTION concede_button_click()
    // update_main_title(null, "You Conceded The Game")
    board.concede_game()
END FUNCTION
```

The chess class will have the following methods. It must be able to handle each of the 4 inputs. These include board square click, difficulty radio button click, reset button click and concede button click. It must make the appropriate decisions of how to handle these inputs. This may involve affecting the DOM using or making a request to the server using the above function.

Here is the UML diagram for the client side Chess_Board class



This UML diagram shows only one class. This it's the **Chess_Board** class that will be used to represent data about the current board state client side. The properties and methods given have appropriate types included. Some of the types are piece types or vector types. These will in practice be implemented as a standardised format of array containing 2 values. A vector will be an array of 2 integers and a piece will be an array of 2 strings. I have named these types in the UML diagram to make it clearer.

Describe database use (save games and minimax cache) and other persistent storage:

In my programs, most of the data stored will be contained within objects. I will make use of some key data structures in addition to these objects. These will be highlighted below.

In the client side JavaScript I will use a dictionary data structure which uses string keys and contains a list for each value. This list will contain a hexadecimal hash and a function.

Here is the pseudocode:

```

previous_data_hashes = {
    "pieces_taken": [None, (board) =>
get_hash_of_data(board.pieces_taken)],
    "move_history": [None, (board) =>
get_hash_of_data(board.move_history)],
    "piece_layout": [None, (board) =>
get_hash_of_data([board.pieces_matrix, board.possible_to_vectors,
board.selected_from_vector])],
    "highlighting": [None, (board) =>
get_hash_of_data([board.possible_to_vectors,
board.selected_from_vector])],
    "difficulty": [None, (board) => get_hash_of_data(board.difficulty,
stringify=False)]
};

FUNCTION update_as_necessary(board, update_function, hashes_table_key)
old_hash, compute_hash = previous_data_hashes[hashes_table_key];
let new_hash = compute_hash(board);

IF new_hash != old_hash THEN
    // hashes are different, updating table
    previous_data_hashes[hashes_table_key][0] = new_hash
    update_function(board)
END IF
// ELSE hashes are the same so don't update

END FUNCTION

FUNCTION update_board_widget(board)
update_main_title(board);
update_as_necessary(board, add_pieces, "piece_layout")
update_as_necessary(board, add_highlighting, "highlighting")
update_as_necessary(board, set_selected_difficulty, "difficulty")
update_as_necessary(board, update_pieces_taken, "pieces_taken")
update_as_necessary(board, set_widget_move_history, "move_history")

```

```
END FUNCTION
```

The function get_hash_of_data represents a generic SHA2 hash function.

Each entry contains a hash of the data relevant to a certain widget on the board and a function to generate the updated hash.

The function update as necessary checks the hash of the current data against the old hash in order to decide if the relevant DOM update function must be run. If the hashes are the same, the function is not run which improves performance. If the hashes are different, the function is run and the hashes are updated.

I felt that this table is necessary as some of the DOM update methods can be visually noticeable to the user and so I don't want to be recreating my HTML widgets every time if there is not change in output to display.

I used a dictionary data structure to organise the hashed and functions into the widget / data category that they corresponded to. I used a list as the 2 items that I wanted to store were of different types.

In my program I also used a 2 dimensional array of pieces. This was useful as it allowed me to very naturally represent that tabular layout to the chess board in an 8*8 array. I will use a 2 dimensional array called pieces matrix in both the frontend and the backend.

The backend 2d array may contain values that are either None or a Piece object. The classes that will represent pieces were defined above.

Here is an example of how the pieces matrix would in the backend, using pieces objects (showing starting positions):

```
STARTING_POSITIONS: array[array[Piece | None]] = [
    [
        Rook(color="B"),
        Knight(color="B"),
        Bishop(color="B"),
        Queen(color="B"),
        King(color="B"),
        Bishop(color="B"),
        Knight(color="B"),
        Rook(color="B")
    ],
    [Pawn(color="B"),]*8,
    [None,]*8,
    [None,]*8,
    [None,]*8,
    [None,]*8,
    [Pawn(color="W"),]*8,
```

```
[  
    Rook(color="W"),  
    Knight(color="W"),  
    Bishop(color="W"),  
    Queen(color="W"),  
    King(color="W"),  
    Bishop(color="W"),  
    Knight(color="W"),  
    Rook(color="W")  
]
```

The frontend pieces matrix will use an array of 2 characters to represent a piece. These character would correspond to the piece's color and its symbol / type. This array pair of characters will likely be the easiest way to represent the chess board and will definitely be used in JavaScript and JSON data representations of the board. This would means that the pieces matrix will use a 3 dimensional 8*8*2 array of characters as appose to the 2 dimensional 8*8 array of Piece objects

Here are some examples of describing pieces in this way:

['B', ' ♜ ']

[‘W’, ‘▲’]

And here is how I would then use this principle to describe a whole chess board (starting positions shown):

I will also make use of a database in my program. This database will have 2 main uses:

- It will use entries in a table to store cached minimax function results for reuse
 - It will use an entries in a different table to store saves games and a corresponding cookie ID contained within the user's browser

I will aim to make use of an external library like SQL-Alchemy or SQLite3 to implement my database. This will allow me to defend my program against accidentally running malicious SQL queries as a result of SQL injections attack. This is needed as the value held in a cookie in the users browser will

be used to reference the saves chess game binary in the database. If this value were to be manually altered to be a harmful SQL query, I want to ensure that no deletion of data can be allowed.

Using a library should also make the process of saving and reading data to and from the database earlier. If I use SQL-Alchemy I can make use if an ORM (Object Relational Mapper) to directly describe how my objects can be converted to and entries in my database.

I will be able to create my 2 tables and the 2 indexes I want with the following SQL commands:

```
CREATE TABLE "Minimax_Cache" (
    primary_key INTEGER NOT NULL,
    board_state_hash VARCHAR,
    depth INTEGER,
    score INTEGER,
    move VARCHAR,
    PRIMARY KEY (primary_key)
);
CREATE TABLE "Saved_Games" (
    primary_key INTEGER NOT NULL,
    cookie_key VARCHAR,
    raw_game_data BINARY,
    PRIMARY KEY (primary_key)
);
CREATE INDEX board_state_hash ON "Minimax_Cache" (board_state_hash);
CREATE INDEX cookie_key ON "Saved_Games" (cookie_key);
```

This should result in a database with the following structure being created:

Minimax_Cache	
int	primary_key
varchar	board_state_hash
int	depth
int	score
varchar	move

Saved_Games	
int	primary_key
varchar	cookie_key
binary	raw_game_data

The above entity relationship diagram shows that database structure that I aim to create. There is not relationship between the tables.

In addition, It could be argued that the use of integer primary keys is unnecessary as I am effectively using the board state hash and cookie key as primary keys. This is because I intend to use these values to search for rows in each table, this is why I have created indexes by these values. In many ways I think it would make sense to make these columns the primary keys, however many ORM libraries (including SQL-Alchemy) do not allow custom primary keys. This means that I must have an incrementing integer primary key, even if it isn't used, due to the limitations of the framework with which I intend to implement my database.

Due to the exponential nature of the minimax algorithm. A single depth 2 search will cause around 900 depth 0 searches (static evaluation) to be completed, assuming a branching factor of 30. This will increase exponentially at a greater depth and could pose scalability problems for the database. In addition, depth 0 cache (stored depth 0 searches in the minimax cache table) is the least valuable of cached minimax results. This is because it can be calculated the quickest and so the cached item prevents the least computation. If the depth 0 cache was stored in a database in secondary storage, the slow read times (relative to time to compute) and large space requirement could result in no overall efficiency gain from the cache.

As a result I intend to implement the above database structure in both a local ".db" file in secondary storage and in separate logical database that will exist in RAM. This will allow me to store depth 0 cache in the volatile database. This will improve read write speed and prevent the persistent database from becoming overfull. This should still present efficiency gains as static evaluation are likely to be reduced within a game. It is not necessary to store these permanently as the chance of encountering the same chess board again is sufficiently low that only high value cache is worth storing.

With this explained redundancy aside, I believe that my database design is already normalised. This is because my 2 tables have no relationship and store no duplicated data. Therefore there is no redundancy and so this design is already in third normal form due to its simplicity.

The use of indexes will increase the amount of space needed to store each entry, as well as increasing write times. However, the reduction in retrieval times should make indexes on certain columns a worthwhile trade off as I intended to query the tables by these columns every time.

Going forward, if further tables needed to be added to the database (e.g. containing user data) then normalisation would need to take place again to ensure that the database remained as efficient as possible.

Variables and key data structures used in these key algorithms:

Do to the heavy use of OOP in my prosed design, global variable will be rare. I outlined the properties of each of my classes using UML diagrams earlier. I have also outlined key data structures that were used earlier.

In addition to this, I intend to use a variable to keep track of the client server session on the server side. This should be able to contain the serialised binary for the specific game object being used in

each of the browsers games, while the user has the website open. There is already a way of achieving this within the Flask framework using a flask session object. This uses temporary cookies on the client's browser to store some data (the Game object) while the tab is open.

I intend to read the cookies of the client browser, when the session is first created. If a cookie has already been created by this website then I will query the database for the relevant game object and restore it into the session.

When the session closes I will save the game object to the database and create a cookie on the clients browser containing the cookie key of that saved game's database entry. This will allow the game to be reloaded if the user visits the website.

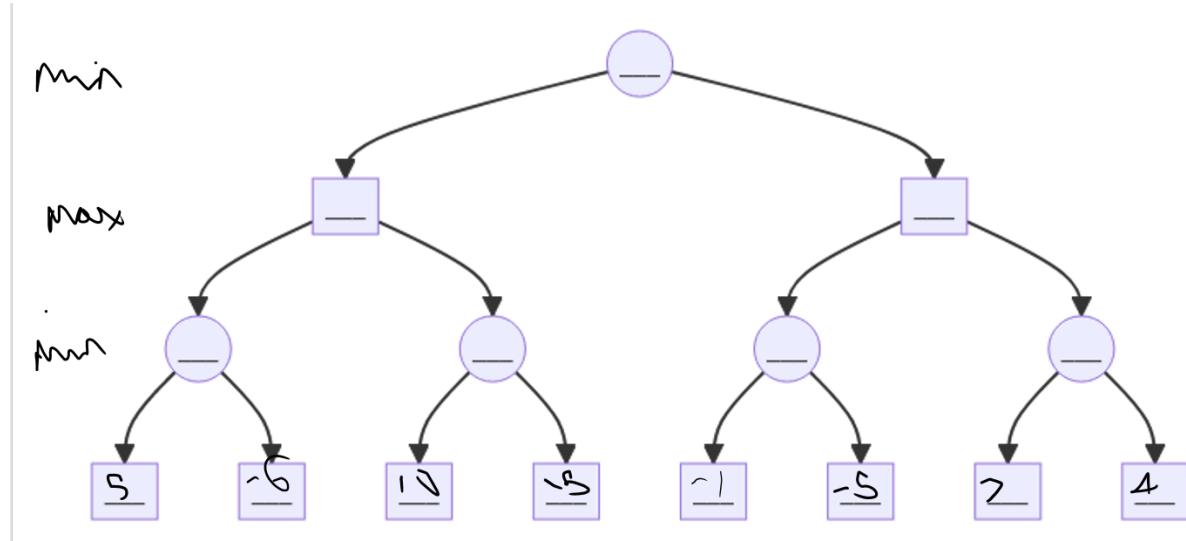
Complete trace tables and dry runs of key algorithms:

I will now aim to trace though some of my main algorithms. As this system uses layers of abstraction, high level functions such as the minimax function would be very difficult for a human to trace in its entirety. This is because a single minimax search needs to run many board state functions, even move pieces functions and a huge volume of vector calculations. As such I will aim to simplify where appropriate.

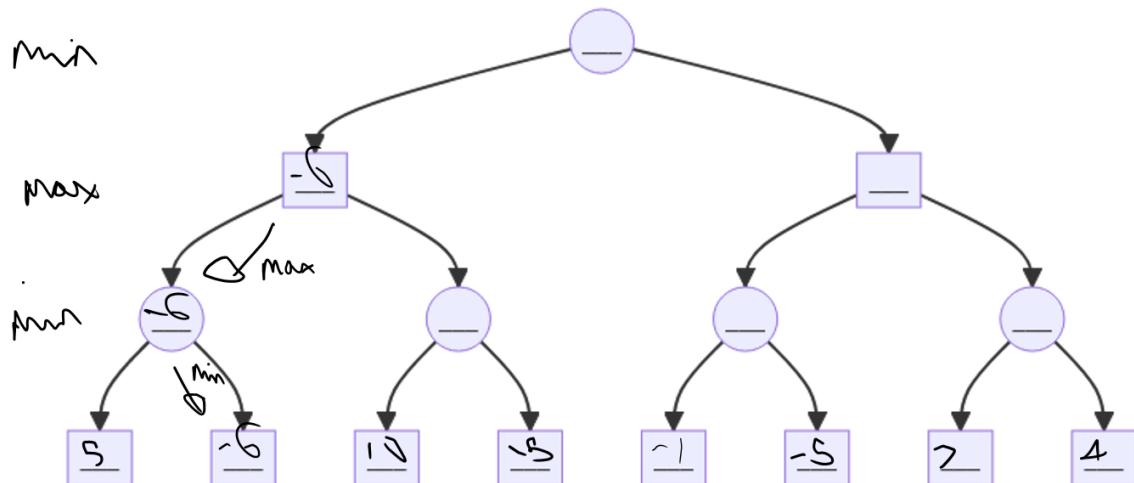
In the case of the minimax function, I will use a general decision tree and abstract away the process of determining static evaluations. I will focus on traversing the tree and then picking the appropriate score.

I will use a diagram of a tree that I have created to trace though the minimax algorithm. I have traced the algorithm though by annotating the diagram at various points.

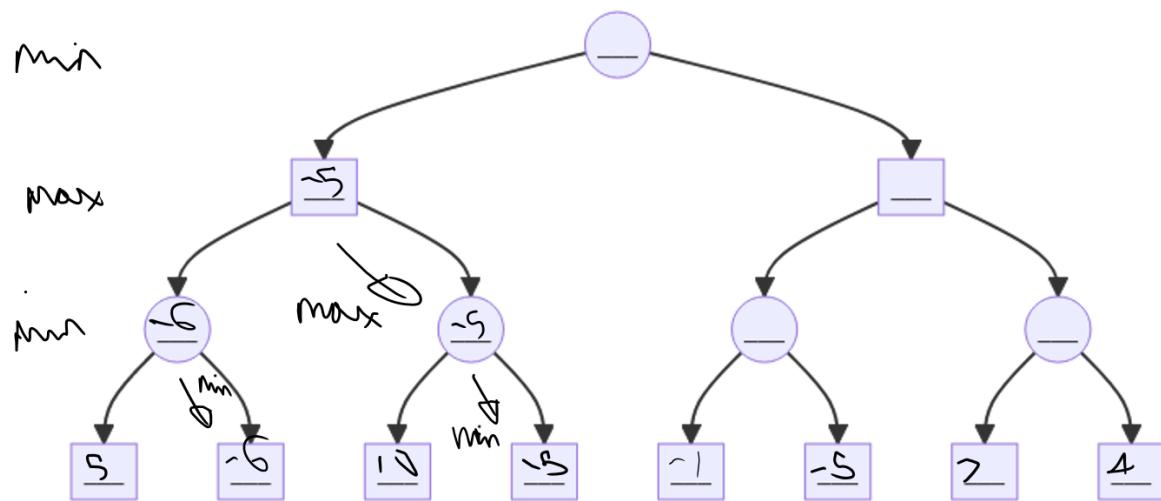
Firstly I traced though a standard minimax function without any pruning.



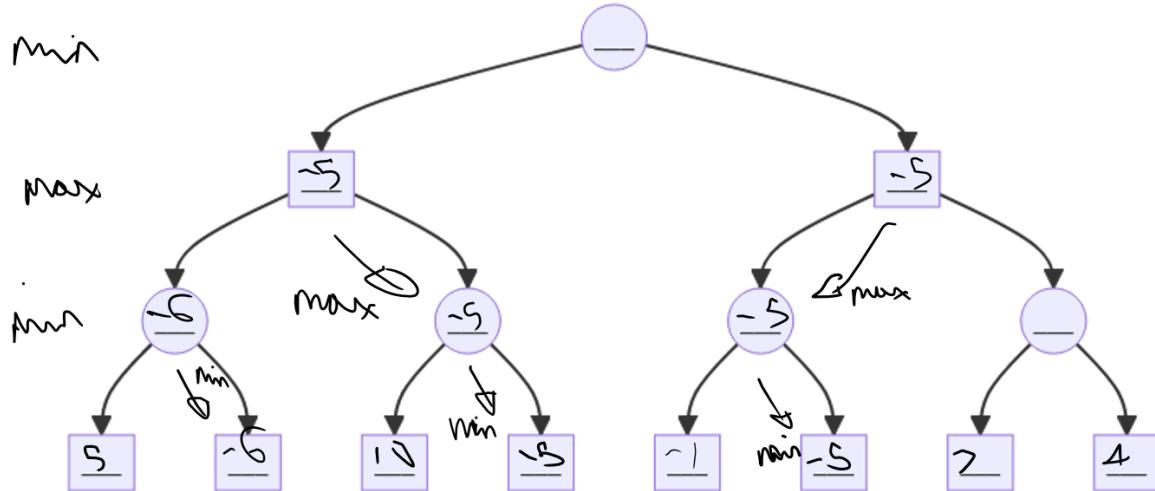
Here we can see a simplified illustration of a depth 3 search by the minimiser. The tree has been simplified by only showing 2 of the children of each node. The static evaluations of the terminal nodes are written at the bottom. Circular nodes represent a decision by the minimiser whereas rectangles are for the maximiser.



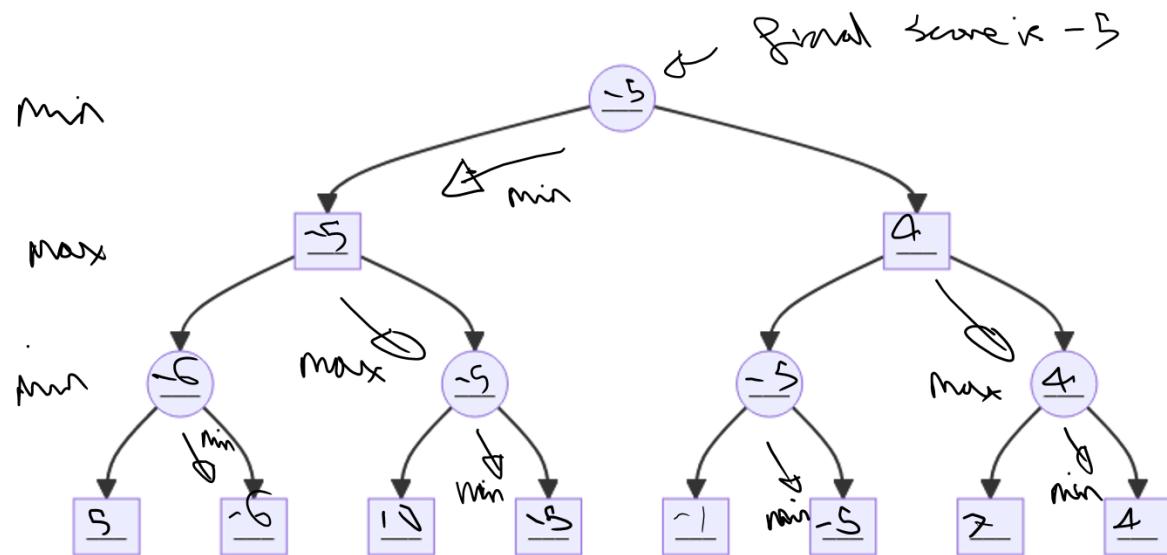
Here we can see that -6 is the lowest score among 5 and -6 and so the minimiser choose -6. As a result, the current best score of the parent maximiser move is also now -6.



We can see that this process is repeated with the minimiser picking -5. As there is complete information and the maximiser anticipates that the 2 child nodes, the minimiser can get a score of -6 and -5 respectively, therefore to maximise its score, the maximiser picks the second child with a current best score of -5.

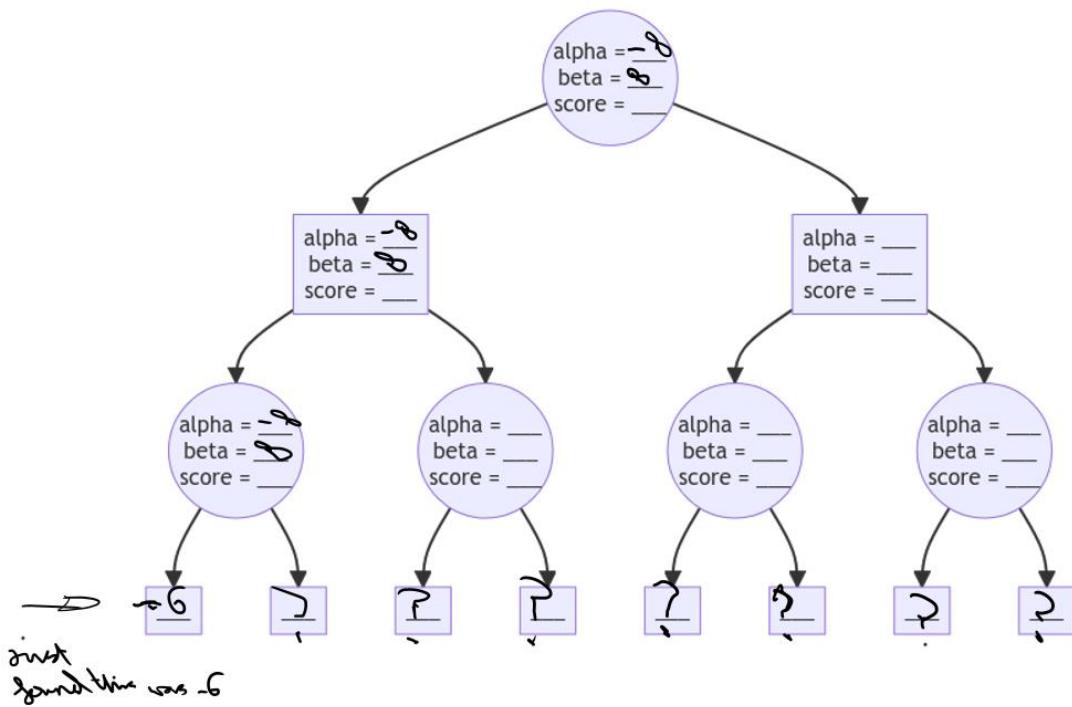


This process repeats again.



then finally, all parts of the tree have been searched and the minimiser decides that the best score that they can guarantee is -5 (at a depth of 3).

Now I will trace through the minimax function more slowly to show its method when using alpha beta pruning.

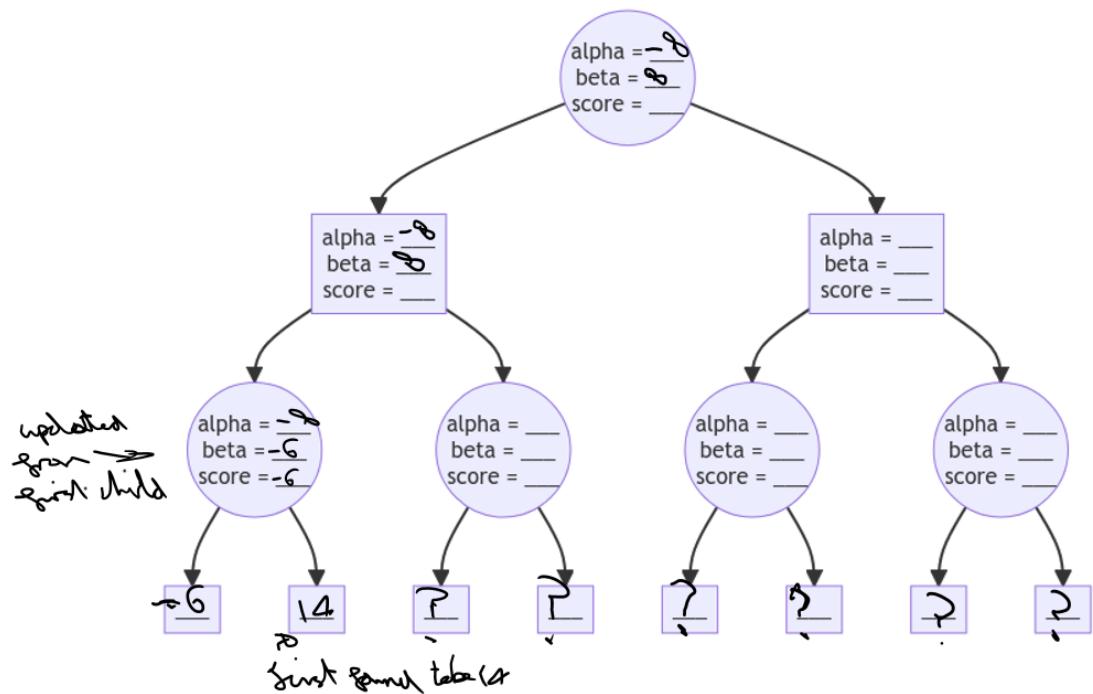


Here we see again a recursive search of depth 3 by the minimiser. I have only filled in the values of nodes once they are reached by the algorithm. Above we see the recursive call has passed its starting values of

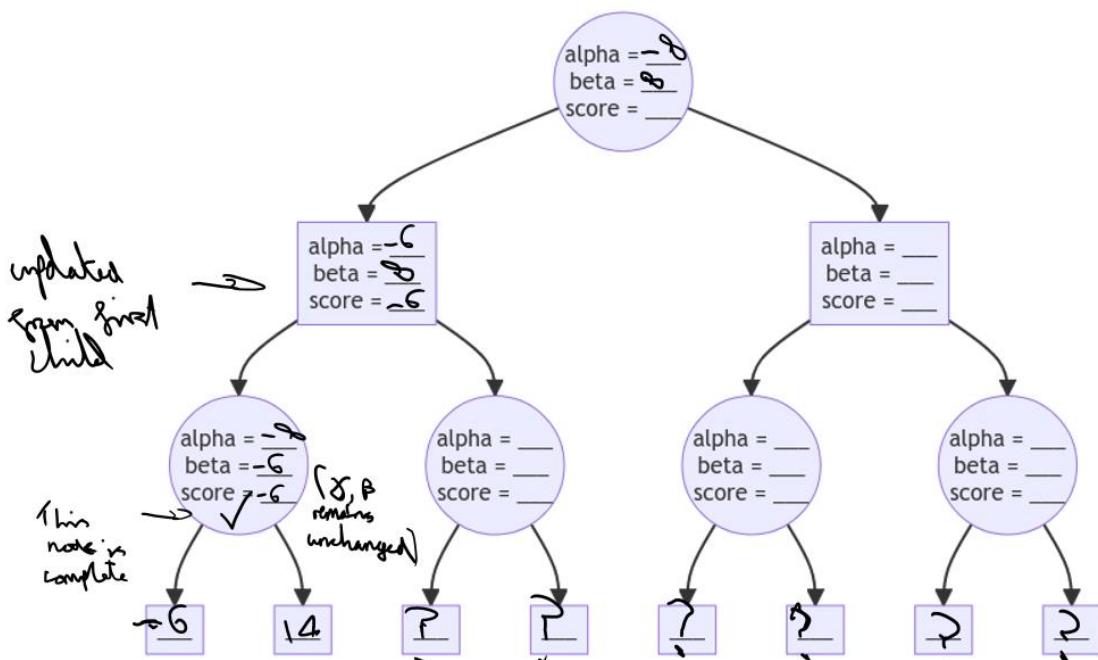
alpha = -infinity

beta = infinity

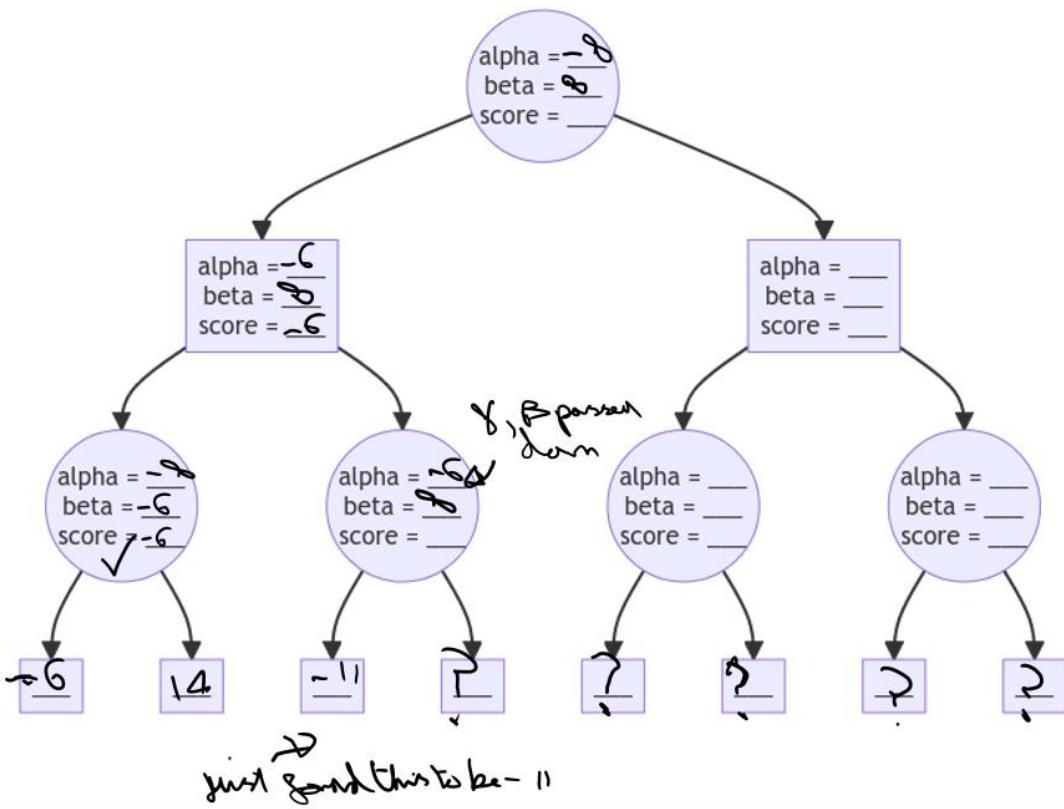
down in all its recursive calls until the base case is reached and a static evaluation is performed to get a value of -6.



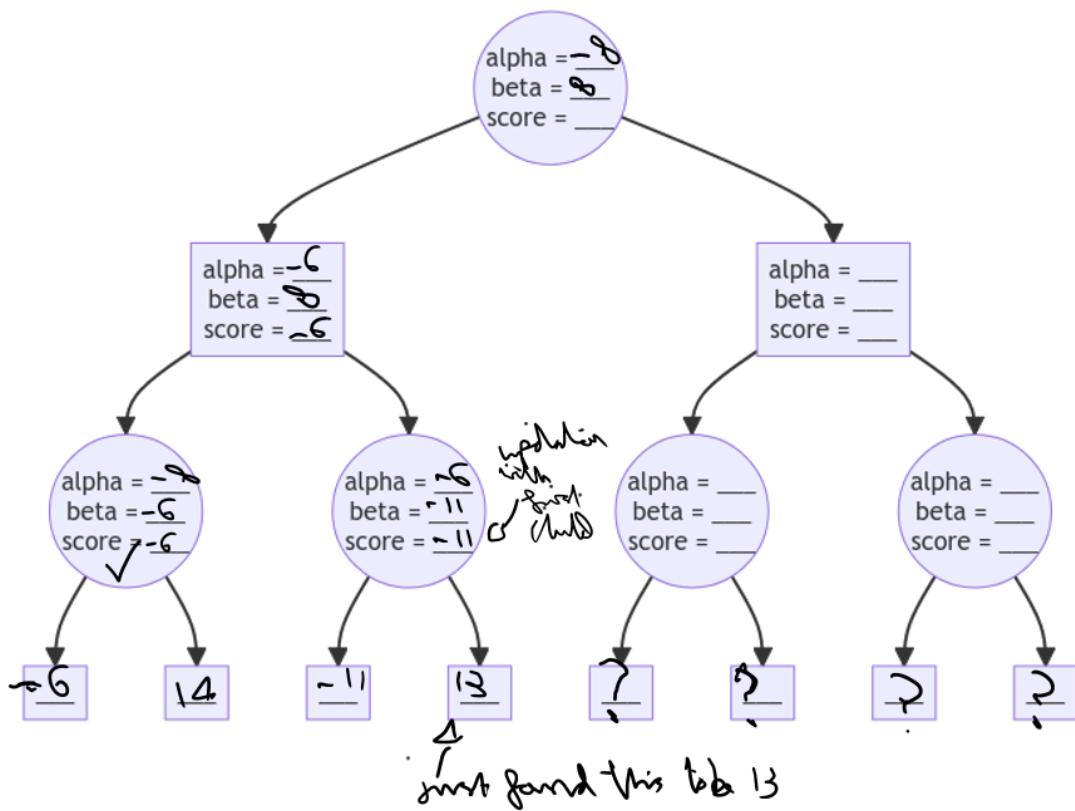
We then see how after examining the first child, the indicated minimiser node (minimisers are circles and maximisers are rectangles) has updated its beta value and its current best score to -6. It then continues to search the children and the next static evaluation is 14.



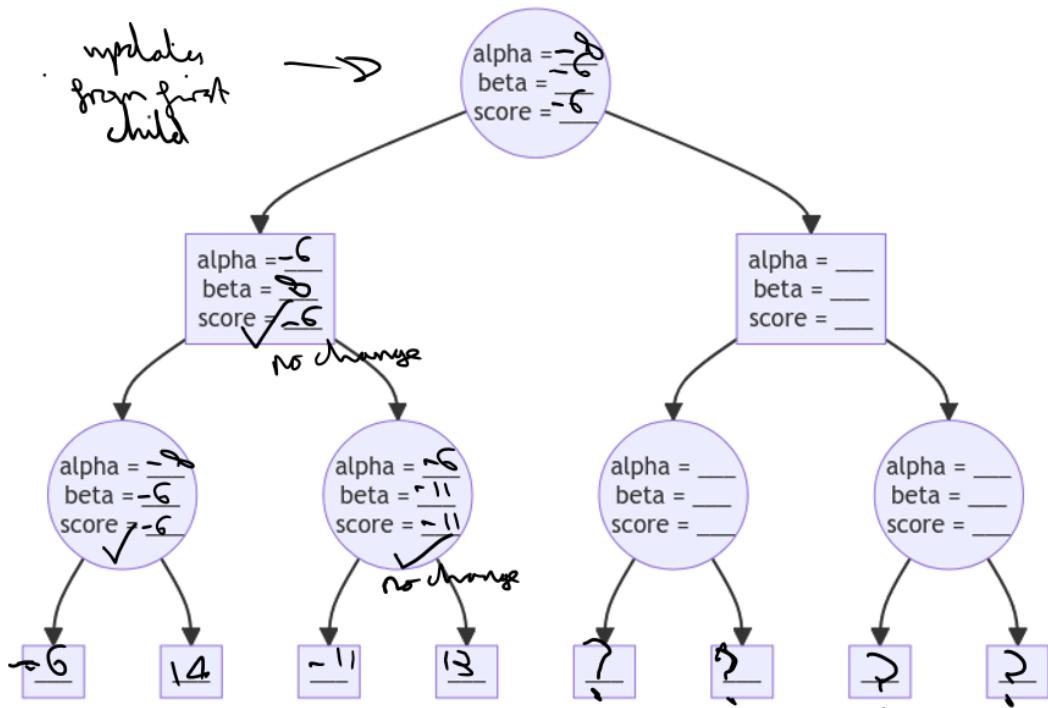
We see how, because the previous value of -6 is better than 14 for the maximiser, the values of beta and best score remain unchanged in the minimiser node (indicated with a tick). Its parent maximiser node has not successfully searched its first child and so updates its alpha and best score values to -6.



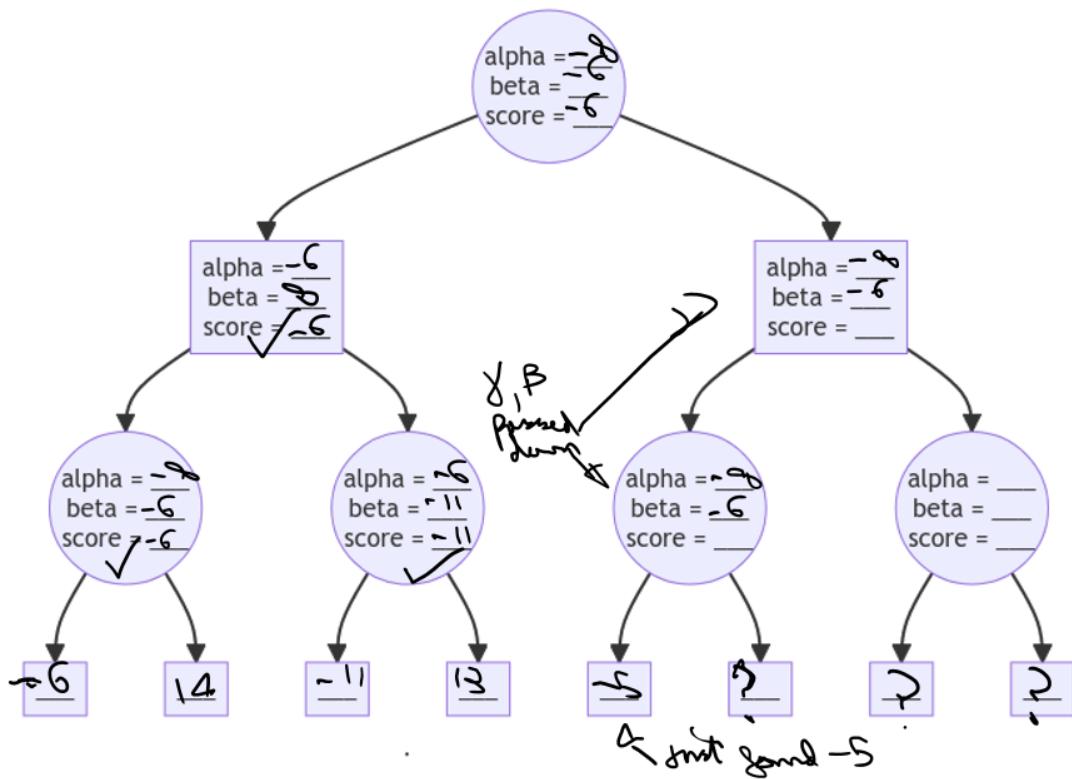
We then see this process repeat recursively. Each time, the alpha and beta values form the parent node are passed down to the search at a lesser depth. (they are never returned up the tree)



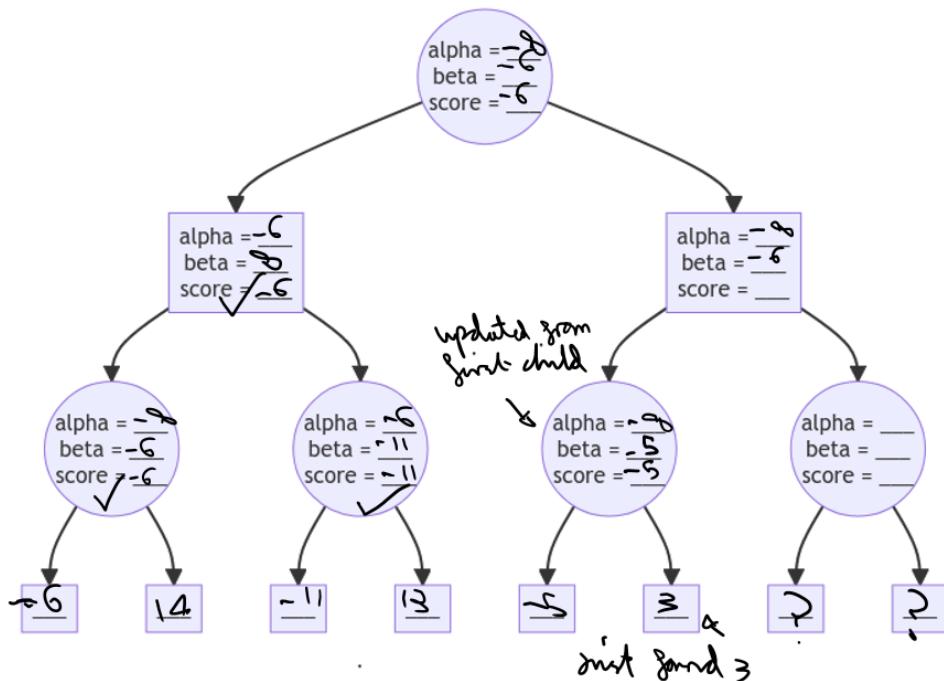
The minimiser node is updated with a new beta and best score value of -11. It should be noted at this point that the pruning condition: $\alpha \geq \beta$ has been met. This is not relevant in this example, but if the minimiser node had more than 2 children, it could stop examining them now.

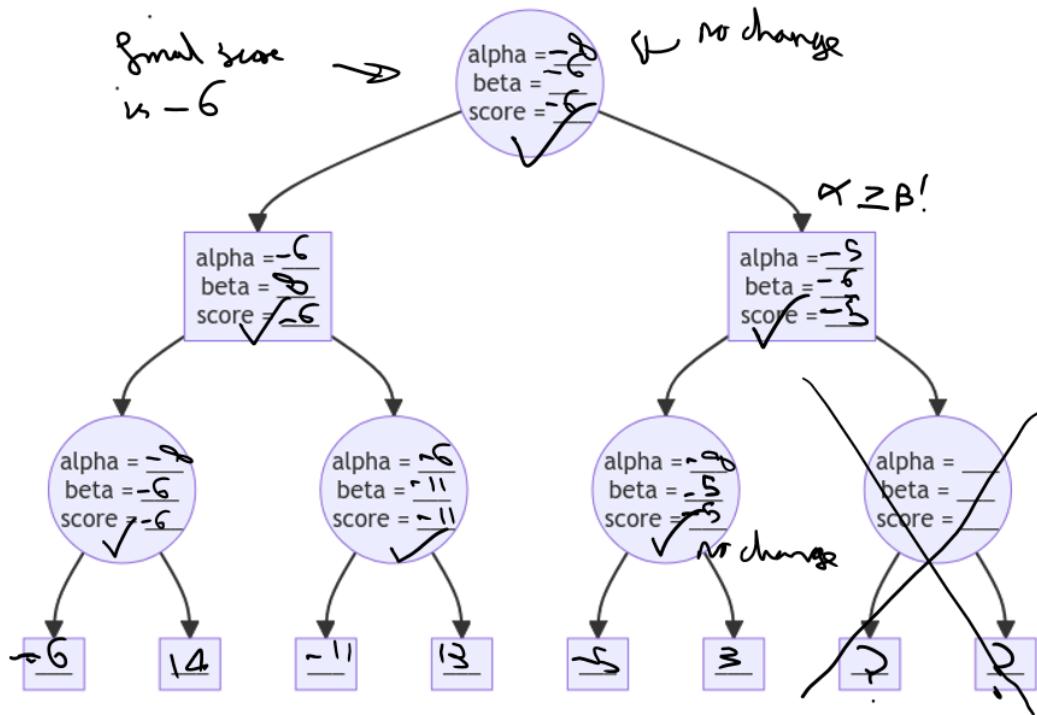


We then see that the minimiser 2nd minimiser node on the third level has finished its search. This means that the 1st maximiser node on the 2nd level has also finished its search with no change to its variables. This causes the variables of the root node to update as its first child has been searched.



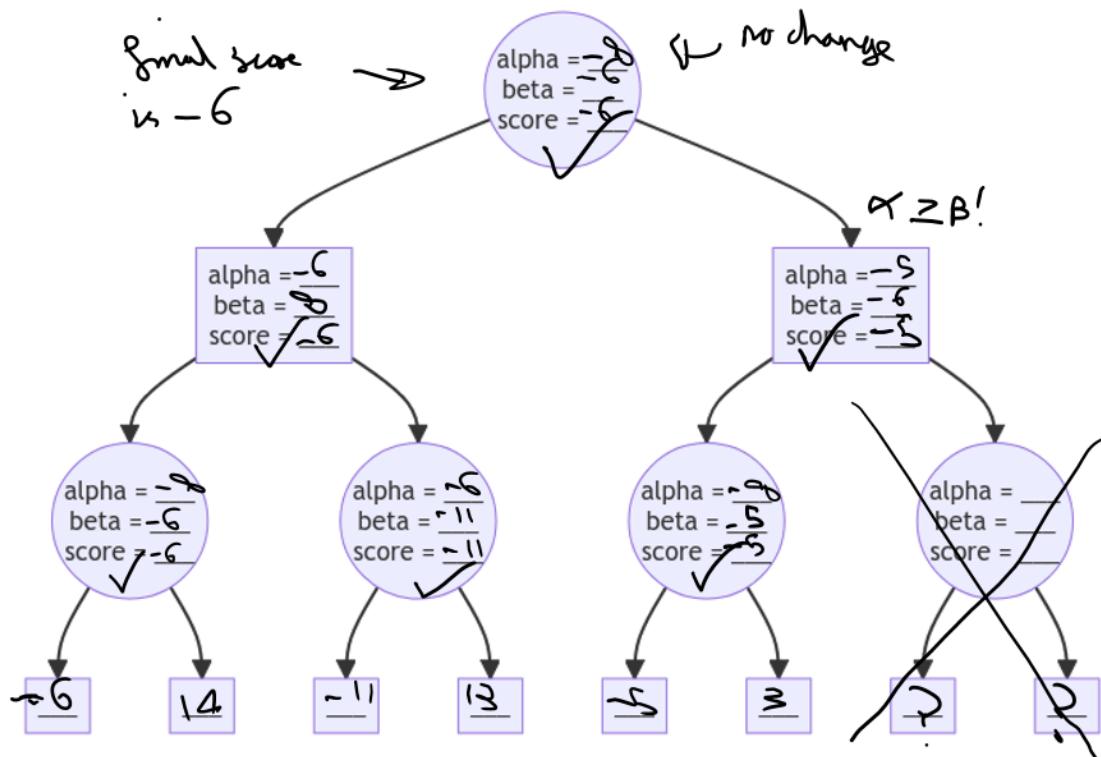
We then see the right side of the tree get recursively searched with alpha and beta values passed down by the root.





The 3rd minimiser node on the 3rd layer has now been completed. The alpha and best score values of the 2nd maximising node of the 2nd layer are now updated.

Again we see that $\alpha \geq \beta$. This means that the 2 static evaluations on the right were prevented as they were not necessary. This can be logically explained as the minimiser at the root node has to decide which of the 2 maximising nodes on the 2nd layer are the worst for the maximiser. Without exploring the last part of the tree, we can see that the alpha value for the right maximiser node is -5 which is already higher than the -6 alpha value of the left node. This means that no matter what the scores are in the unexplored part of the tree, the right half of the tree is already better for the maximiser and so the minimiser at the root node will always pick the left maximising child node.



This means that the depth 3 search is completed with a final score of -6. If no pruning was used, the final score would still remain the same.

I have now accurately traced the minimax algorithm through in a graphical manner. This traceback exemplified the order in which it searches nodes. It also shows how the values of alpha and beta are updated and passed to child nodes. With the previously defined pseudocode, I can apply this searching method to chess. The use of a best move variable in the pseudocode will also ensure that the move is returned and not just the score.

Validation:

The final program would feature both client side and server side validation. This should allow invalid inputs to be detected so that they do not cause invalid outputs.

My final program will have 4 main inputs:

- The user can click any square on the chess board
- The user can click the reset button
- The user can click the concede button
- The user can click the radio buttons that control difficulty

Client side validation needed:

It should not be possible for the user to provide an invalid input to the concede and reset buttons. The button should be able to be used as any time. This includes when the game is over but also during both the user and computer's go. The only input the user should be able to provide is a

Boolean, did they click the button? yes or no? Both of these inputs (pressing and not pressing the button at any given stage in the game) are valid.

It should also be impossible for the user to select an invalid difficulty. The difficulty settings will be changed using radio buttons. This means that when the user select one option, all previous options should be unselected. Using radio buttons (e.g. as opposed to a text input) will also reduce the range of inputs that can be given as only valid difficulties can be selected. I will make the program automatically select one of the radio buttons (e.g. default of medium) when the page loads. This should ensure that at all points in the program, exactly one of the possible difficulties will be selected.

I will also use client side validation to decide when the client should ignore erroneous and invalid click events from the user. To input a move the user will click 2 squares on the chessboard: a move from square and a move to square. I will use highlighting to make this intuitive.

The client side `Chess_Board` class should contain an array of legal moves that the user can make at any one point. After the user move these should be updated by the server. The `Chess_Board` class will use an additional 2 properties to allow the user to input their move and add validation in the process:

- `Selected_from_vector`: This variable will contain none of a position vector in the form `[i, j]` where i and j are integers
- `possible_to_vectors`: This variable will be an array that will contain position vectors (in the form `[i, j]` where i and j are integers)

Before the user first clicks a square, the `selected_form_vector` will be set to None and the `possible_to_vectors` will contain an empty array.

When the user clicks a square, the position vector of that square will be checked to see if it is an item in the `possible_to_vectors` variable. If the clicked square's position vector is a in the `possible_to_vectors` property then the user move will be made. If not then the user has not selected a valid to square.

I will then check if the clicked square contains a white piece. If it does then it will be treated as the user's input for their move from square (which piece they want to move). The position vector for the clicked square will be stored in the property `selected_from_vector`. The `possible_to_vectors` will then be updated by iterating through the array of legal moves and adding all possible legal moves that start by moving the selected piece.

This method will allow me to:

- Ensure that the user cannot move the opponent's pieces. This is because only clicking one of the users own pieces will allow them to input a from square. Clicking other squares will cause the `selected_vector` property to be set to none.
- Ensures that the once the user has selected one of their pieces to move, they cannot select an invalid movement square move to. Only squares in the position vector array `possible_to_vectors` (a subset of legal moves for where the selected piece can move) can be selected as a to square

All invalid square click events will be ignored.

Server side validation:

I will also add server side validation to prevent invalid inputs.

If the user tries to bypass the JavaScript and make a move that they should be able to an error will be caused on the server. Due to the use of the Flask framework, this error will be caught. While it will be logged to the console and it may prevent that client from continuing with their chess game, it will not cause the server to crash or disrupt other users.

My server side validation should only be used when users deliberately bypass of tamper with the client JavaScript. As a result I will focus on preventing these invalid inputs from causing unexpected behaviour. I will not produce an error message that will be sent to the client to explain the issue as it will have been caused deliberately.

Below is a list of the various exceptions that should be raised on the server to top it and the relevant invalid input they represent. I will validate the input and then deliberately raise and exception to prevent the server using the invalid input which could result in unexpected behaviour. I will create the custom exceptions below myself by creating a python class that inherits form Exception.

Invalid input to server	Resulting Exception
Invalid difficulty provided to the change difficulty handler	ValueError("difficulty not recognised")
Not the user's turn, and yet the client has tried to implement a user move	NotUserTurn()
Not the computer's turn and yet the client has tried to implement a computer move.	NotComputerTurn()
Illegal move: it is the users turn but the provided move isn't legal	InvalidMove()

Functionality that each prototype will have:

Prototype 1:

The aim for this prototype is to successfully tackle a simpler problem with a similar approach. I will aim to create a website that will allow uses to play tic tac toe against the computer.

The users will have access to a 3 by 3 board with which to play the game. A title will declare that it is their turn to move first. The board's squares will respond to their mouse clicks, allowing them to put a cross in any of the 9 squares. There will also be a reset button to allow for the game to be restarted.

I will create an API connection between the server and the client using either WebSocket HTTP. This will allow the client to request the user's move. The server will then respond with a move which the client will implement on the board. The game will continue until the computer wins or the game is a draw.

The computer will use alpha beta pruning and the minimax algorithm to use a decide the computer move. The approach will be to use the British Museum algorithm to look ahead and search the

whole decision tree, through to the 9th turn where all games end. This will allow the computer to perform a search and decide the best move to use.

This prototype will help me work towards chess and will help me partially achieve a variety of success criteria. These include:

- Working towards items 2, 3, and 4 by adding client side validation to ignore clicks when it is not the user's turn
- Working towards items 6 and 7 as I will need to graphically show the computer and user move
- Working towards items 9, 10 and 26 as I will create a first iteration of a working minimax algorithm that can then be later adapted to play chess.
- Working towards item 17 as I will need to create a similar client server API connection for this simpler program
- Working towards item 31 as I will need to create a reset button in this game

Prototype 2:

This prototype should begin to address the complexities of chess. It will include a working version of the 3 main components of the chess computer:

- It will include a fully finished Chess Engine / Chess Functions module that can make inferences about a chess board. This must include detecting if the game is over and identifying all legal moves
- I will have created a working prototype of the Move Engine Component. This version of the minimax function will include heuristics and search to a given depth instead of using the approach of the British Museum Algorithm. The move engine should also feature some more optimisation including variable depth checking and pre-sorting moves
- I will also create a working prototype of the Game Manager module that is able to keep track of and manage an individual chess game

I will also aim to create a console based chess game what uses my game manager module. It should provide a text based interface with which the user can play chess against the computer

I will also aim to begin exploring how best to implement final chess webpage GUI. I will attempt to use a framework called VUE to achieve this. I will then build upon my progress with the user interface in prototype 3.

Prototype 3:

This will be the final prototype of my iterative development cycle. It will aim to create a program that meets all the design criteria. Doing this will enable me to create a solution that meets all essential, high priority and desirable success criteria items. I can then add more featured from the success criteria if I have move time later.

This means that the third prototype should feature a webpage GUI that the user can access to play a game of chess.

The Move Engine should be fully optimised to maximise efficiency and efficacy.

The Game Manager should be fully implemented and include multiple different difficulty setting.

Test Plans

Functional white box testing during development:

Test Plan for Prototype 1:

For this iteration, I will test a variety of aspects of the final webpage and game to ensure that the chess AI, validation and GUI behave as anticipated.

To do this I will perform the following tests:

- Valid: Try clicking a square when the game loads and see that it now contains an “X” and I should see that another square fills with a “O”.
- Invalid: I will then test what happens when I click the 2 filled squares, nothing happens.
- Valid: I will test that when I click another square, I am able to make my second move and put an “X” there
- Valid: The game should be able to successfully identify when the user has lost. It should freeze the board and display this
- Valid: The game should be able to successfully identify when the game is a draw and produce the appropriate output.
- Valid Extreme: The game should be able to successfully identify when the user wins and handle this. The chess AI should prevent this ever happening so I will perform the test by loading a board state into the board so that the user is a move away from winning.
- Valid: The reset button should work in resetting the board in all 3 game states: the user’s move, the computer’s move and when the game is over.

Test Plan for Prototype 2:

The testing for this prototype will come in 2 main parts:

Firstly, I will create automated test for the Chess Engine and Move Engine to ensure that these components are working correctly.

To test the Chess Engine I will test the various functions that I have created for a variety of inputs and expected outputs. I will store the test data in external files and use the python unit test library to complete the tests.

It is important that the functions in the Chess Engine are robust as they will be used by many of the other modules. For instance the Move Engine Module and Client Side validation rely on the generate legal moves and game over functions. If these have logic errors, this could produce knock on logic errors in the wider program.

I will create automated tests to test the following functions for a variety of inputs and expected outputs:

I will test the method of the Vector class:

- Vectors are equal
- Add vectors
- Subtract Vectors
- Multiply Vectors
- Check position vector in chess board
- Check the vector can be converted to a square (e.g. F5 or A2)

I will then test the generate movement vectors method of the piece class for each pieces.

To do this I will check that the piece can move where I expect on an empty chess board.

I will then create a populated chess board that is filled with other pieces. This will allow me to test the vectors by which a piece can move are affected by the pieces around it, resulting in the expected legal moves (ignoring check).

I will then test various methods of the Board State class:

- I will test the generate all piece method for various piece layouts
- I will test the get piece at vector method for various pieces at different places in various boards
- I will then test that the generate pieces of a specific color method works. I will do this by checking that it can give me all the black and white pieces for a variety of chess board states.
- I will then test that the is color in check method works for a variety of board states and edge cases (including checkmate).
- I will test that the game over function is able to correctly recognise when the game is over in a variety of board states as well as say the nature: “checkmate / stalemate” a winner “none / w / b”
- I will test that the generate legal moves function is able to correctly identify all the legal moves available to a given player for a variety of board states. This will include testing extreme data such as game over situation where there aren’t any legal moves.

I will then test the Move Engine’s Minimax function.

To do this I will create a variety of tests where 2 different computer bots play against each other. All my tests will have a “good bot” that I expect to win and a “bad bot” that should lose. I will use my Game class to have the 2 bots play against each other. I will output the moves that they made and the running score to a file. By ensuring that the good bot beats the bad bot and by checking the graph of the game to see that the win was progressive (gradually took pieces then won, not was losing and the bad bot then made a mistake) I can test that the good bot is better.

Being able to create a test like this where I can confirm that one bot is better than another will enable me to:

- Confirm that my minimax algorithm is making intelligent moves by having it beat a randomly moving bot
- Confirm that a minimax search at a greater depth gives me a better move by testing that bots with a greater depth beat bots with a lesser depth.

- Confirm that variable depth checking produce better moves. To do this I will test 2 depth n bots against each other to see that the good bot with a variable depth of 1 beats the bad bot.

I will create the following test to test my minmax algorithm: (good bot on the left, bad bot on the right)

- Depth 1 vs random mover
- Depth 2 vs random mover
- Depth 3 vs random mover
- Depth 2 vs depth 1
- Depth 3 vs depth 1
- Depth 3 vs depth 2
- Depth 1 + 1 variable depth vs depth 1
- Depth 2 + 1 variable depth vs depth 2

I will not test the console chess game as this is not a significant module. I created it to allow users to give feedback on the program. The focus of this iteration (and therefore also the focus of the testing) was to test that the chess engine module is robust.

Test Plan for prototype 3:

To test this module I will use some automated testing:

- I will add to the “bot vs bot” test of prototype 2 to test that bots that are given more time to explore the decision tree do better. I will perform multiple tests where a bot with X seconds of time should be beaten by a bot with 1.4X seconds of time. I will perform these tests for the following values of X
 - 2
 - 5
 - 10
 - 15
 - 20
- I will create a test to verify that the output of the parallel minimax algorithm is the same as the output of the standard minimax algorithm for a variety of board states
- I will create a test to check that the cache enabled minimax function performs much faster when performing a search on a board state that it has already seen and searched before
- I will test that my game object can be serialised to binary and then restored without any properties being overwritten

I will then manually perform the below test using the website:

Test No.	Aspect Being Tested		Type of test	Test Data Used	Expected Result
1	The chess board and website renders correctly		Valid	I loaded the page in my browser	The webpage load and the chess board is drawn
	Actual Result	Evidence		Success?	

Test No.	Aspect Being Tested		Type of test	Test Data Used	Expected Result
2	I can, the user can move my pieces and will the computer respond with a move		Valid	I clicked a piece to move it forward	I should be able to move my pieces and I should see the computer move its pieces
	Actual Result	Evidence		Success?	

Test No.	Aspect Being Tested		Type of test	Test Data Used	Expected Result
3	I should be able to make all legal moves and shouldn't be able to make any illegal moves		Valid (I can make legal moves) and then Invalid (I can't make illegal moves)	I will try to move various pieces	I should be able to try to move a variety of pieces and they should move as expected, no missing legal moves and not extra legal moves.
	Actual Result	Evidence		Success?	

Test No.	Aspect Being Tested		Type of test	Test Data Used	Expected Result
4			Valid extreme	I tried to loose and then I tried to win	When the game is over (for both winners), the board should be disabled and the title message should explain that the game is over.
	Actual Result	Evidence		Success?	

Test No.	Aspect Being Tested		Type of test	Test Data Used	Expected Result

5	The database cache is being build up after each minimax search the computer completes	Valid	I played the AI on various difficulties and checked that database as I did	I should add a small amount of depth 1 cache to the database on a low difficulty. It should produce a large amount of depth 1 cache with some depth 2 cache at a high difficulty
	Actual Result			Evidence
				Success?

Post development testing against development criteria (black box alpha testing):

I will then perform the following tests in post development. These test will check the functionality of the final program. Each test will be justified with a link to the success criteria

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
1	Different Difficulty Settings	Needed for stakeholder (casual chess player). Success criteria item 1	Valid	Clicked radio buttons then looked for a change in response time for the computer move	The response time should change so that it is approximately the same as the listed time (e.g. medium is 5 seconds) plus an extra second delay.
	Actual Result			Evidence	Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
2	Client Side Validation	Test that I cannot input an invalid move or move the opponents pieces on my turn.: Success Criteria items: 2 and 28	Invalid	Clicked on my piece than an invalid square, then tried to click and move the opponent's pieces	The invalid click events are ignored, they will only affect the highlighting.

	Actual Result	Evidence	Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
3	Client Side Validation	To meet success criteria items 3 and 29, the user cannot move when it is not the user's turn. The move that the computer produces should then be valid	Invalid	Set difficulty to high and then try to move while it is the opponents turn. The computer then made a move which was valid	These clicks will be ignored and there will be no corresponding output. The computer's move should be a valid legal move
	Actual Result	Evidence			Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
4	Check: testing that a message is used to show check and testing that the user cannot input invalid moves that cause their king to be in check	To meet success criteria items 4 and 11 and to ensure that the program has the relevant and does not allow invalid inputs	Valid (for message) And Invalid (try to make invalid move)	I play the game until I had moved my king into check	The computer should make use of the main title to show that it is my go and I am in check. It should show with highlighting that I cannot make moves that violate the rules of check. When I try to make these moves anyway, they should be ignored
	Actual Result	Evidence			Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
5	All aspects of highlighting in both the user and computer move. As well as checking that the user and computer can move	This will indicate if my usability feature to show the moves and legal moves with highlighting is working. I will also see if the computer can make a move and if it shown with highlighting This corresponds to items: 5,6,7,8 from the success criteria	Valid	I implemented user moves step by step to see the highlighting at each part of the process.	I can click one of my pieces, it will be highlighted red and everywhere it can move to will be highlighted green. The computer move will be highlighted before it happens. The moving piece will be red and the square it move to will be green.
	Actual Result	Evidence			Success?

	Exactly as expected	See clip 5	Full success		
--	---------------------	-------------------	--------------	--	--

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
6	The Chess AI should be capable of making smart moves, it should also be able to learn over time. I will also try to show that I have improved the algorithms efficiency by using multiple codes	Tests success criteria item 9,26, 27	Valid	I will make a series of moves to lay a trap where it trades a knight for a pawn	on trivial difficulty it should fall for the trap. Then on extreme difficulty and all later games it should avoid the trap. I should be able to show this with the database.
	Actual Result	Evidence			Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
7	Computer's ability to put me in checkmate. And the appropriate game over output	This will test that the AI is trying to actually win and will test how the program uses outputs to clearly show when the game is over. This corresponds to success criteria items 10, 12, 13	valid	I will try to put myself in checkmate	The computer should be able to put my king in checkmate. The main title should then show that I am in check mate. The board should then be disabled
	Actual Result	Evidence			Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
8	Bulk testing of many features. I will test: The move history and pieces lost tables work; the reset and concede buttons work and that reloading the webpage restores the game. I will also check that the server connection is fast enough	This will test various features are working as intended. It will test that the program meets success criteria item: 15, 30, 31, 32	Valid	Checking the contents of the tables and clicking the buttons	The tables should be able to accurately keep track of the chess game (moves and pieces taken) as I make moves. The buttons should be able to end the game and then reset it. This includes resetting the move and pieces tables
	Actual Result	Evidence			Success?

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
9	Bulk testing of many features. I will test: that reloading the webpage restores the game. I will also check that the server connection is fast enough. I will also verify that the program works in different aspect ratios.	This will test various features are working as intended. It will test that the program meets success criteria item: 17, 19, 25	Valid	I started a new game, reloaded the tab. Then show the response speed and then different aspect ratios	I expect the response speed to be fast. The game should be reloaded when the tab is reopened. The website should work on: <ul style="list-style-type: none"> • PC / Laptop • Tablet Landscape • Phone portrait
	Actual Result	Evidence			Success?

Acceptance testing for success criteria (beta testing):

For each item in the success criteria I will justify whether I believe that I have or have not met that item. I will reference specific post development tests as evidence when I believe that I have achieved a success criteria item. I will also provide evidence for and success criteria items that I feel I met with partial success.

I will add a comment for each item of the success criteria to comment on how important I thought that criteria was, how I met it or why I didn't meet it.

I will also perform a questionnaire of my stakeholders to try to determine how well they think I met the success criteria. I will focus on usability features and ease of use.

Iterative Development – Prototype 1

Aims for this iteration

The main aim is to create a working prototype for a system to solve a simpler problem that can be tackled in a similar way to chess.

I intend to create some code to run as a server and some code to run in the user's browser. They must be able to communicate using some form of API.

This prototype should act as a stepping stone to help me tackle the larger problem of chess.

It should use a simple version of minimax with alpha beta pruning to determine the best move in tic tac toe. This is over-engineering for tic tac toe as a simple set of 'if-then' rules would suffice. However if I use the approach of the British Museum Algorithm to make a working version of minimax, then this will help me with chess in future.

Another main aim is to create a simple skeleton architecture that can be built upon. This involves:

- using python to write the code for the webserver
- writing the JavaScript code to validate input and manage the game client side
- Using a REST API or WebSocket to create a connection between the frontend and the server.

In terms of the aims of the prototype from general perspective:

The prototype will have only one page/window and will allow the user to play tic tac toe vs the computer. This is a simpler problem to tackle as tic tac toe is a solvable game. This means that there is a best strategy that leads to a draw or a win and it can be determined. I will go into more specifics on why I have chosen tic tac toe and why it is a simpler problem to tackle in the research section.

The primary goal of this prototype is not to perfect the frontend or to perfect the backend for my final chess program. Instead It will focus on developing the backend and the connection between the frontend and backend. Multiple prototypes will provide a stepping stone toward the completion of each element of the project. In this case the primary goal will be met so long as the user can play noughts and crosses, even if the user interface is not the most easy to use. This should be a test that I have successfully implemented the webserver, minimax algorithms and the connection between the frontend and backend.

With regards for the macro plan form my prototypes:

The aim for prototype 2 is to create a working interface and game engine to play a simplified version of chess where there are only pawns. I will then add in other pieces and the idea of checkmate in subsequent iterations. I will use as many prototypes as need be to get to full chess. I will then refine by adding a database and difficulty levels. I will then add additional features if I have time.

Functionality that the prototype will have

In terms of the prototypes functionality:

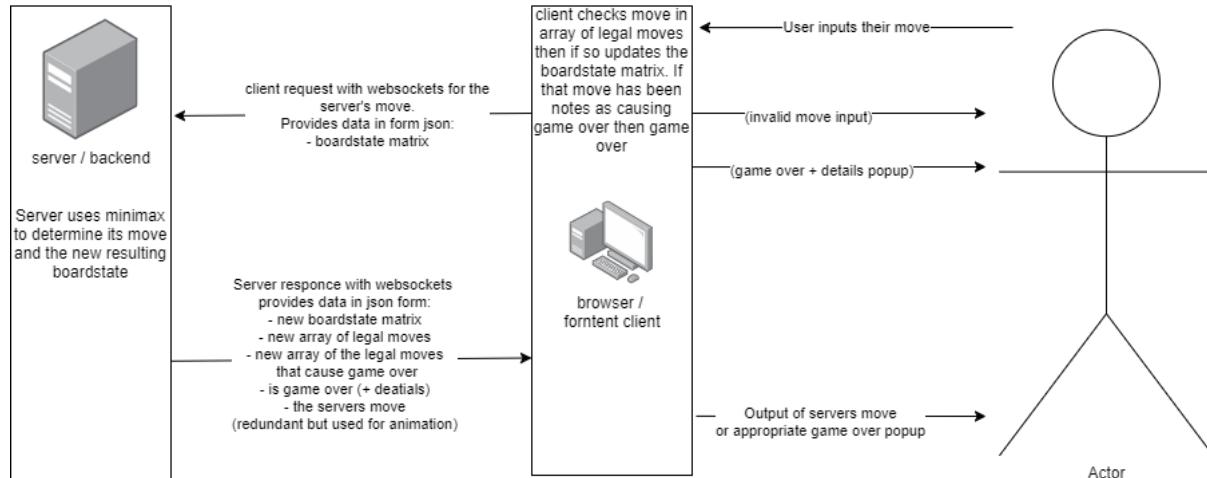
- It should be a single webpage that consists of a tic tac toe board, a reset button as well as a title and some text.
- Text should prompt the user that it is their turn.
- The user should then be able to click a square to put a cross 'X' there.
- The computer should then respond (near instantaneously) by putting a naught 'O' in another square.
- The user should then be able to play the game out until a draw or loss (or in theory win).
- Then an alert should appear to explain how the game has ended.
- The user should be able to reset the game at any point to play again.

I recognise that either I will fail to correctly implement the minimax function, in which case I will not meet all my technical goals for this prototype, or the user will never be able to win. This shouldn't be a problem with chess as the computer cannot play perfectly and there will be different difficulty settings available.

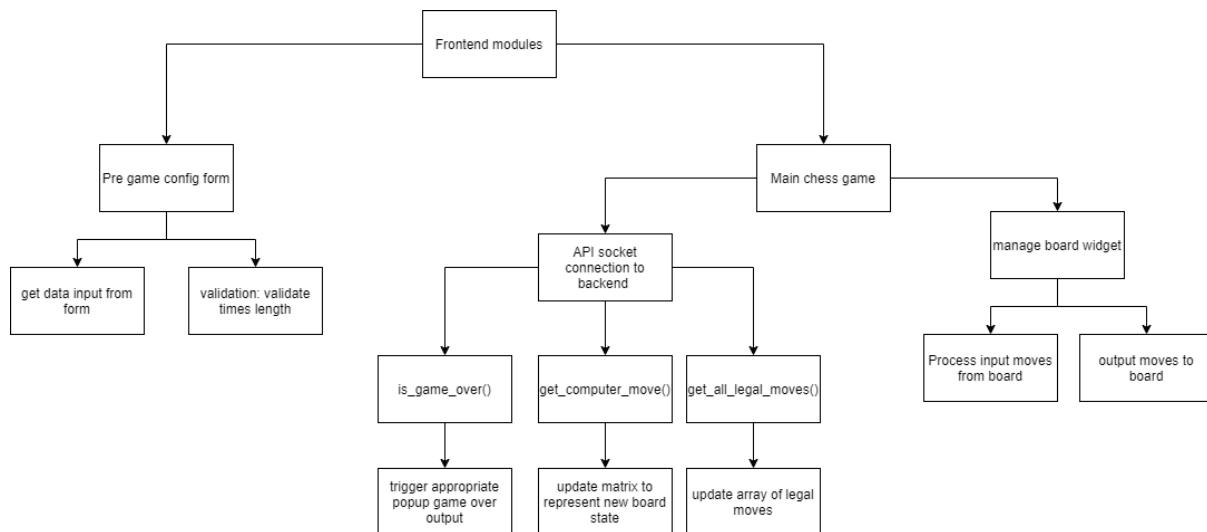
I believe that the added difficulty of making the computer play sub optimally is greater than the benefit to the user experience in this prototype if the user can win. This is after all only a prototype and the user experience is not its primary goal. This sounds silly but my end product isn't tic tac toe so refinement of the user experience would experience diminished returns.

Due to the simplicity of this game compared with chess, not all of the modules, in my plan for the final product, are needed / applicable. I have only in a loose sense, created a portion of my modules for the front and backend. Instead tic tac toe has some different modules that are simpler.

From by overall plan this iteration includes the following functionality



The API connection is similar. The main difference is that the frontend determines if a move is legal on its own



Of the frontend module plan, I have created the get computer move function and the board widget (a simpler version) but not the rest. The API connection has been simplified and the menu wasn't needed for this prototype

However the functions developed on the backend (including the api connection) are the bulk of this project. I have created:

- simpler version of the minimax function (no static evaluation)
- a simple game state class with child generator and other methods
- I have created a WSGI webserver that host the webpage (simpler as only one webpage)
- I have created the socket connection (simpler as less data transmitted between)

Annotated code screenshots with description

First I will show screenshots of the code with explanation beyond comments when relevant (but comments are extensive). Then I will show screenshots of it working

Code Editor:

I used visual studio code for the editor and IDE. I didn't 'set it up' as I used my normal programming configuration. The relevant extensions I am using in vs code include:

- Microsoft IntelliCode
- Microsoft Jupyter
- Microsoft Python
- Sourcery
- Ventur
- VUE language features

These extensions are mostly for syntax highlighting, 'intelli-sence' and jupyter notebook compatibility.

I used the console to run my program mostly but made some use of breakpoints for debugging.

My main tool in debugging was jupyter notebook which allowed me to import the function from a python file I was developing and see how the functions behaved with sample inputs.

The reason I used Visual Studio Code is that it was the editor I was familiar with and had already configured. Over time I had tried other editors like 'Notepad ++' and 'Atom' but I preferred VS code. This was for the following reasons:

- It provides easy navigation of a folder full of files by allowing you to access them with the keyboard only and have multiple windows at once
- It allows you to run your code using the integrated PowerShell terminal (which I rely on as I later discover that I need to use the terminal to run my code successfully)
- It has integration with git which I will try to setup for version 2 to have some version control
- It has a myriad of different available extensions to help with language syntax
- It has direct support for python and jupyter notebooks unlike other editors.
- All settings and keyboard shortcuts can be changed.

Below are some screenshot examples of me using VS Code.

```

# style.css      main.js      index.html      app.py      game_engine.py
prototype 1 > app.py > handle_server_move_request
19 # issue solved by running with 'python -m flask run'
20
21
22 # define socket handler for move request
23 # decorator on move request sent from client
24 @socketio.on("server_move_request")
25 def handle_server_move_request(msg):
26     # unpack json data and load into object
27     game_state: ge.Game_State = msg["game_state"]
28     board_positions:msg["board_positions"],
29     to_go_next:msg["next_to_go"],
30     moves_left:msg["moves_left"]
31
32     # use game engine module's main function to determine the best child state ()
33     best_child: ge.Game_State = ge.main(game_state)
34
35     # code used for debugging to print out the move
36     # print('selecting move:')
37     # for row in best_child.board_positions:
38     #     print(row)
39
40     # now use emit to give a response in the form of an updated board matrix
41     socketio.emit(
42         "server_move_response",
43         {"board_positions": best_child.board_positions}
44 )

```

```

prototype 1 > game_engine > game_engine.py > main
1 # the data classes library is used to minimize boiler plate code in defining __init__
2 from dataclasses import dataclass
3 # this is made to complete and distinct copy of a 2d array so changes to
4 from copy import deepcopy
5
6 # this is a class to represent the game state
7 # its attributes cannot be altered after it is initialized (immutable).
8 # instead changes to the game state, such as a move, are handled as new game states
9 @dataclass(frozen=True)
10 class Game_State:
11     # these variables hold the data relevant to a game state
12     board_positions: list[list]
13     moves_left: int
14     to_go_next: str
15
16     # used for debugging so that the game state can be represented in the console
17     def print_board(self):
18         for row in self.board_positions:
19             print("|".join(["." if e == "" else e for e in row]))
20
21
22 > def is_game_over(self):
23
24     # this function iterates through and yields the game states that could result
25     def gen_child_game_states(self):
26         """precondition of game not being over"""
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
InvalSession XEtck_HFKGRYi7AAA (further occurrences of this error will be logged with level INFO)
127.0.0.1 - [07/Oct/2022 22:11:17] "POST /socket.io/?EIO=4&transport=polling&t=0615208ksid=XEtck_HFKGRYi7AAA HTTP/1.1" 400 -
The WebSocket transport is not available, you must install a WebSocket server that is compatible with your async mode to enable it. See the documentation for details. (further occurrences of this error will be logged with level INFO)
127.0.0.1 - [07/Oct/2022 22:11:17] "GET /socket.io/?EIO=4&transport=polling&t=0EqISqFe HTTP/1.1" 200 -
127.0.0.1 - [07/Oct/2022 22:11:17] "POST /socket.io/?EIO=4&transport=polling&t=0EqISqFe&id=VjGhMnt6Czb-wdAAA HTTP/1.1" 200 -
127.0.0.1 - [07/Oct/2022 22:11:17] "GET /socket.io/?EIO=4&transport=polling&t=0EqISqFe&id=VjGhMnt6Czb-wdAAA HTTP/1.1" 200 -

Python 3.10.0 64-bit

File Edit Selection View Go Run Terminal Help <- > prototype 1 (Workspace)

main.js game_engine.py debug_minimaxipybnb

prototype 1 > game_engine > debug_minimaxipybnb > original function

+ Code | Markdown | Run All | Clear Outputs of All Cells | Restart | Variables | Outline ...

i will now try to make a tree to better visualize what is happening

```

@dataclass
class Tree:
    root_node_text: str
    sub_trees: list

    def print_tree(self, indent=0):
        # print(f"printing tree {self.root_node_text} with indent {indent}")
        print("  " * indent + "→" + self.root_node_text)
        for sub_tree in self.sub_trees:
            sub_tree.print_tree(indent=indent+1)

[38]
Tree("A", [
    Tree("B", [
        Tree("E", []),
        Tree("F", [])
    ]),
    Tree("C", [
        Tree("D", [
            Tree("G", []),
            Tree("H", [])
        ])
    ])
]).print_tree()

[39]
|-->A
|----->B
|----->C
|----->D
|----->E
|----->F
|----->G

```

Python 3.10.0 64-bit

Jupyter Server: Local Ln 33, Col 17 | Spaces: 4 | CRLF | Cell 3 of 31 | Go Live | Spell |

Libraries, Languages and other assets:

I decided to use python and JavaScript for this project as I am already familiar with both. I know how to code many more complex technical tasks in python that I will need to complete in later prototypes. For example maintaining a SQL database. I also think that there is no real alternative to JavaScript for client side logic and validation. Therefore these languages were familiar and a good fit for the project, so I choose to use them.

For this iteration I use very few libraries and external assets. I only used VS Code for the editor and only used code I had written. No elements of my solution (including the html and UI) were produced using a forms designer.

I used libraries minimally, here are all the libraries I used:

Python:

- Flask: used to create a WSGI HTTP web server
- Flask_socketio: used to create a WebSocket server to connect the front and backend

- Dataclasses: used to avoid boiler plate code (which can be quite a lot) in class definitions for the `__init__` and `__eq__` (comparison) method. It also easily allowed me to make my classes immutable
 - Copy: I used the `deepcopy` method to make a duplicate copy of a 2 dimensional array in python. Without it changes to the original array change the new copy as they use the same memory location

JavaScript:

- `Socket.io.js`: used to allow the client side JavaScript to connect and communicate with the `socket.io` server

WSGI stands for Web Server Gateway Interface. It allows the python framework that I am using (flask) to give instructions in a standardised way to specialist webserver software like waitress. I can still run the program locally for debugging without this software in development mode. When I am finished I will need to use Waitress as an additional piece of necessary software to deploy my project.

I decided to use WebSocket to make my API over a traditional REST approach using HTTP. This is because WebSocket creates a full duplex TCP/IP connection. This leads to lower latency (allowing for more complex validation to be server side without delays for the user) and it allows for messages to be 'broadcast' to all clients or just 'emitted' to one. This is important if I want to allow for a leader board or player vs player matches in future.

The code:

Starting with the HTML

Here is my head tag. It contains boilerplate code to provide browsers with basic metadata. It also fetches the WebSocket library which is how the frontend and the backend will communicate. A CSS file and JavaScript file are also loaded in from the server.

```
<body>
    <!-- i used the center tag to make the CSS simpler -->
    <center>
        <h1>Tic Tac Toe</h1>
        <h2>You go first:</h2>
        <!-- this table will be the 3x3 board -->
        <table id="board">
            <tbody>
                <tr>
                    <!-- each square has an ID so I can see which one is clicked -->
                    <td id="sq_11"></td>
                    <td id="sq_12"></td>
                    <td id="sq_13"></td>
                </tr>
                <tr>
                    <td id="sq_21"></td>
                    <td id="sq_22"></td>
                    <td id="sq_23"></td>
                </tr>
                <tr>
                    <td id="sq_31"></td>
                    <td id="sq_32"></td>
                    <td id="sq_33"></td>
                </tr>
            </tbody>
        </table>

        <!-- this is the button for resetting the game -->
        <button id="reset_btn">Reset Game</button>
    </center>

</body>
```

Here is the body tag and its contents. The individual squares of the board and the reset button are given ids so that I can give them a click event in JavaScript later.

The JavaScript then consists of one file with many functions which I will add screenshots of here. I will only add explanation on top of the comments if I think it is necessary. I am not necessarily explaining the JavaScript in which it appears in the file as the order how the functions are defined is not relevant or the best way to explain it.

Here I define my global variables and setup WebSocket. I initially tried to make a class to represent the noughts and crosses game that contained attributes and methods that used and changes these attributes. I then realised that the class would be all the code and that it didn't make sense as I would only ever have one instance. Due to the small size of the code, this lead me to decide to use global variables to contain the data about the current game.

```
// create a socket object from the online socket library
let socket = io();

// global constants to show game state
let moves_left = 9;
let next_to_go= "X";
let board_positions= [
    ['', '', ''],
    ['', '', ''],
    ['', '', '']
];
```

I then I add click events to the squares and the reset button

```
// code to be executed when the page loads
window.addEventListener('load', function(){
    // for testing (if loading with a non blank game state)
    // update_widget();

    // the following blocks of code bind the appropriate handler method to each square
    // I tried to use iteration but it results in a logic error that I struggled to debug
    // I therefore adopted the less elegant manual approach
    // the looping code is still here commented out as I have tried to make it work

    // for(i=1; i<=3; i++){
    //     for(j=1; j<=3; j++){
    //         get_square(i, j).addEventListener('click', function () { square_click(i, j) });
    //     }
    // }
    get_square(1,1).addEventListener('click', function(){square_click(1, 1)});
    get_square(1,2).addEventListener('click', function(){square_click(1, 2)});
    get_square(1,3).addEventListener('click', function(){square_click(1, 3)});
    get_square(2,1).addEventListener('click', function(){square_click(2, 1)});
    get_square(2,2).addEventListener('click', function(){square_click(2, 2)});
    get_square(2,3).addEventListener('click', function(){square_click(2, 3)});
    get_square(3,1).addEventListener('click', function(){square_click(3, 1)});
    get_square(3,2).addEventListener('click', function(){square_click(3, 2)});
    get_square(3,3).addEventListener('click', function(){square_click(3, 3)});

    // tie the reset the game handler function to the reset button on click
    document.getElementById('reset_btn').addEventListener('click', reset_game)
});
```

I will not show you the functions that I defined. In theory many of them are actually procedures as they don't return anything. However they behave in a similar way as they use global variables.

```
// this function takes a row and column 1 to 3 and gives the html element
function get_square(row, col){
    // console.log('get_square called');
    // return document.querySelector(`#sq_${row.toString() + col.toString()}`);
    return document.getElementById(`sq_${row.toString() + col.toString()}`);
};
```

```

// this is the method that is run whenever a square is clicked
function square_click(row, col){
    // console.log("square_click called");

    // validate that the square clicked is a legal move for the user to make
    // if the game isn't over
    if (!(is_game_over())){
        // and if the next to go is X, the user
        if(next_to_go === "X"){
            // and if clicked square is a legal move
            if(is_legal_move(row, col)){
                // run the make move function
                make_move(row, col);
            };
        }
        // else do nothing and ignore the click input as if the square tag were disabled.
        // else {
        //     console.log({next_to_go})
        //     console.log("user move ignored as server next to go")
        // }
    };
}

// checks if the target square is empty and returns a boolean
function is_legal_move(row, col){
    // console.log("is_legal_move called");
    // console.log({row})
    // console.log({col})
    i=row-1; j=col-1;
    return board_positions[i][j] === '';
};

// this function is responsible for determining if the current game is over.
// this is done by examining moves left to detect draws and triplet sequences to detect wins
// this function only returns a boolean value and not how the game is over
function is_game_over(){
    // console.log("is_game_over called");
    // if no moves left then the game must be over
    if(moves_left === 0){
        return true;
    };

    // get the relevant triplet sequences
    triplets = get_triplets();
    console.log({triplets});

    // iterate through the 8 triplets
    // for(i=0; i<8; i++){
    for(i=0; i<7; i++){
        triplet = triplets[i];
        // using 2 equals comparison on purpose
        // i am using JSON.stringify to compare arrays to see if they are 3 in a row
        if (JSON.stringify(triplet) == JSON.stringify(["X", "X", "X"]) || JSON.stringify(triplet) == JSON.stringify(["0", "0", "0"])){
            return true;
        };
    };
    // if not already determined that the game is over then it must not be over
    return false;
};

```

```
// this returns an array of all rows, columns and diagonals on the board for examination
function get_triplets(){
    triplets = []
    // push all rows
    for(i=0; i<=2; i++){
        triplets.push(board_positions[i])
    }
    // push all columns
    for(j=0; j<=2; j++){
        column = []
        for (i = 0; i <= 2; i++) {
            column.push(board_positions[i][j])
        }
        triplets.push(column);
    }
    // add diagonals
    triplets.push([board_positions[0][0], board_positions[1][1], board_positions[2][2]])
    triplets.push([board_positions[0][2], board_positions[1][1], board_positions[2][0]])
    return triplets
}

// this function is run once the users move has been validated and is responsible for executing
// it orchestrates the various functions that must be called in the process of making a move
function make_move(row, col){
    // console.log("make_move called");
    i=row-1; j=col-1;
    // alter board position array to make the move
    board_positions[i][j] = next_to_go;
    // change who is next to go
    toggle_next_to_go();
    // console.log({next_to_go})
    // there is now one less move left in the game
    moves_left += -1;
    // update the visual widget in html
    update_widget();

    // if the game is over handle else request the server's move
    if(is_game_over()){
        handle_game_over();
        // alert("game over")
    }
    else {
        request_server_move();
    };
}
```

```
// changes the next to go to the next player based on current player
function toggle_next_to_go(){
    // console.log('toggle_next_to_go called');
    if(next_to_go === "X"){
        next_to_go = "0"
    }
    else {
        next_to_go = "X"
    }
};

// this function updates the table tag in html so the contents reflects the board positions array.
function update_widget(){
    // console.log("update_widget called");
    // console.log({board_positions})
    var row; var col;
    for (let i = 0; i <= 2; i++) {
        for (let j = 0; j <= 2; j++) {
            row=i+1; col=j+1;
            square = get_square(row, col);
            // console.log("changing content of this tag")
            // console.log(`square ${row.toString()} ${col.toString()}`)
            // console.log({square})
            // console.log(`to ${board_positions[i][j]}`)
            // square.innerText = board_positions[i][j];
            square.textContent = board_positions[i][j];
            // square.textContent = "Test";
        }
    }
};
```

```

// this function is run once the game is over
// it determines how the game is over and takes displays the appropriate message
function handle_game_over(){
    // precondition game is over so function is_game_over returned true
    // determine who won or if it was a draw
    // 1 is user (X) won, 0 is draw, -1 is user (X) lost
    // console.log("handle_game_over called");

    // variable initialized with 0 to represent draw
    winner = 0;

    // get triplets and iterate through them
    triplets = get_triplets();
    // for (i = 0; i <= 8; i++) {
    for (i = 0; i <= 7; i++) {
        triplet = triplets[i];
        // for each triplet make a comparison to check if the triplet means that the user has won or lost
        // using 2 equals comparrison on purpose
        if (JSON.stringify(triplet) == JSON.stringify(["X", "X", "X"])) {
            winner = 1;
            // no real need to break out of the loop as their can only be one or zero winning or loosing sequence
            // but it makes the code more efficient and readable
            break;
        }
        else if (JSON.stringify(triplet) == JSON.stringify(["0", "0", "0"])){
            winner = -1
            break;
        };
    };
    // if there were no winning sequences then the outcome will remain the default: 0 for draw

    // display the appropriate message to the user in each case
    if(winner === 1){
        alert("congratulations you won");
    }
    else if(winner === 0){
        alert("the game was a draw");
    }
    else{
        alert("unfortunately you lost");
    };
    // no need to disable the game as the is_game_over check will fail whenever the user clicks a square
};

function reset_game(){
    // reset globals
    // if the user has just in the last second moved and then clicked reset before the computers move
    // | a logic error could occur where the game resets and then the server responds with its move
    // this is prevented here by ensuring that a server move isn't pending
    if(next_to_go === "0"){
        return false
    }

    moves_left = 9;
    next_to_go = "X";
    board_positions = [
        ['', '', ''],
        ['', '', ''],
        ['', '', '']
    ];
    // update the html to reflect the board position matrix
    update_widget();
}

```

The following functions are used to request and handle the server's move.
They require a little more explanation

```
// this function emits a request to the server for its move
function request_server_move(){
    // console.log("request_server_move called");
    // it uses socket emit
    // is provides the server with the relevant game data
    socket.emit(
        'server_move_request',
        {
            board_positions,
            next_to_go,
            moves_left
        }
    );
}
```

This function sends a request to the server for its move using the `socket.emit` function. The server will process the request and then respond with a new game state, that is where its move is added.

The following event handler ties a relevant handler function to run when this happens.

```
// tie the handler method to the socket event.
socket.on('server_move_response', function(msg){
    // console.log({msg})
    // update global for board positions with the new one that the server has sent
    board_positions = msg.board_positions
    // call handler function
    handle_server_move();
    // i believe all the handler function have to return false
    return false;
})
```

Here is the `handle_server_move` function

```
// this function is called to handle a response from the server and execute the server's move
// the global board positions will have already been updated with the ones returned by the server
function handle_server_move(){
    // update the html to show the user the updated game state
    update_widget();
    // decrement the moves left
    moves_left += -1;
    // toggle who is next to go to the user
    toggle_next_to_go();
    // check if the servers move is a winning one and if so handle appropriately
    if (is_game_over()) {
        handle_game_over();
        // alert("game over")
    };
    // no else needed to enable the user to have a turn as game over validation included before user's turn
}
```

This is all the client side JavaScript that I used to make the interface work. The rest of the program is python based. It is responsible for hosting the website and determining the best move.

Here is the code for app.py:

This is a python program that acts as the webserver and socket server for the frontend website. It is a relatively small program

```
# imports
# built in libraries
import flask
# import os
from flask_socketio import SocketIO
|
# my local code
# import logger as logger_module
from game_engine import game_engine as ge
```

To start with I Imported 2 libraries to help provide the WSGI and WebSocket server functionality. I also imported the functions from a program I write called game_engine. This contains the code for my minimax function.

```
# setup flask app
app = flask.Flask(__name__)

# setup sockets
socketio = SocketIO(app, async_mode=None)
# RuntimeError: You need to use the gevent-websocket server
# issue solved by running with 'python -m flask run'
```

I then setup my app object and socketio object. This is boilerplate code to use the relevant libraries and setup my WSGI and socket server.

```
# define flask api route handlers
# this endpoint responds with the index.html page which will then load and request files form the static folder.
# this is the main endpoint that hosts the
@app.route('/', methods=['GET'])
def index():
    return flask.render_template('index.html')
```

I then used my app object to setup a handler for what happens if a HTTP get request is made to the root. This method returns the html page. I use the flask render template method so that URLs to external CSS and JS files are substituted into the html code where {{ }} double curly brackets are used.

```
# define socket handler for move request
# decorator on move request sent from client
@socketio.on("server_move_request")
def handle_server_move_request(msg):
    # unpack json data and load into object
    game_state: ge.Game_State = ge.Game_State(
        board_positions=msg['board_positions'],
        to_go_next=msg["next_to_go"],
        moves_left=msg['moves_left']
    )
    # use game engine module's main function to determine the best child state (game state following best move)
    best_child: ge.Game_State = ge.main(game_state)

    # code used for debugging to print out the move
    # print('selecting move:')
    # for row in best_child.board_positions:
    #     print(row)

    # now use emit to give a response in the form of an updated board matrix
    socketio.emit(
        "server_move_response",
        {"board_positions": best_child.board_positions}
    )
```

I then used my socketio object to define a handler function for the server_move_request.

It unpacks data provided into a Game_State class from the game_engine module. It then gets the best child game state using the minimax function. This represents the game state after the server has made the best possible move. The resulting board positions matrix is then sent back to the client by making an emit with the name 'server_move_response'.

```
# this function simply runs the app and configures the port that is used.
def run_app():
    app.run(host='127.0.0.1', port=5000, debug=True)

# this code runs the main app function only if this file is run directly and not if it is imported
if __name__ == '__main__':
    run_app()
```

The above code is then responsible for starting the server when this python file is run directly. It used port 5000 on the local host.

Game engine file

This file contains code that is used to determine the server's best move.

```
# the data class library is used to minimize boiler plate code in defining __init__ and __eq__ methods for classes
from dataclasses import dataclass
# this is used to make a complete and distinct copy of a 2d array so changes to the original don't affect the copy
from copy import deepcopy
```

Here is imported external libraries and gave the reasons for why

```
# this is a class to represent the game state
# its attributes cannot be altered after it is initialized (immutable).
# instead changes to the game state, such as a move, are handled as new game state objects
@dataclass(frozen=True)
class Game_State:
    # there variables hold the data relevant to a game state
    board_positions: list[list]
    moves_left: int
    to_go_next: str

    # used for debugging so that the game state can be represented in the console.
    def print_board(self):
        for row in self.board_positions:
            print("|".join(["." if e == "" else e for e in row]))

    def is_game_over(self):...

    # this function iterates through and yields the game states that could result from all possible moves on the current game state
    def gen_child_game_states(self):...
        # else continue to next iteration
```

I then use a class to represent a tix tac toe game in a specific state.

```
def is_game_over(self):
    """Returns False, None for still going and True, 1/0/-1 for over.
    1 is user wins, 0 is draw, -1 is user loses"""
    # internal function as it only used here
    # yields all of the sequences of 3 squares
    def gen_triplets():
        # yield rows
        yield from self.board_positions
        # yield columns
        for col in range(3):
            yield [self.board_positions[row][col] for row in range(3)]
        # yield diagonals
        yield [
            self.board_positions[0][0],
            self.board_positions[1][1],
            self.board_positions[2][2],
        ]
        yield [
            self.board_positions[2][0],
            self.board_positions[1][1],
            self.board_positions[0][2],
        ]
    # if there is a sequence that is 3 in a row return True and 1 or -1 for win or loss (user perspective)
    if ["X"]*3 in gen_triplets():
        return True, 1
    if ["O"]*3 in gen_triplets():
        return True, -1
    # note so long as user takes all odd turns 9,7,...3,1 they will have last go so redundant in theory
    # but I left it in incase the code needs to be extended in future
    if self.moves_left == 0:
        return True, 0
    # if not already determined otherwise then the game isn't over
    return False, None
```

I then produced a function to determine if the game is over and if so what the state is

```
# this function iterates through and yields the game states that could result from all possible moves on the current game state
def gen_child_game_states(self):
    """precondition of game not being over"""
    # for each square
    for i, row in enumerate(self.board_positions):
        for j, square in enumerate(row):
            # if the square is empty
            if square == "":
                # make a copy of the boards state where the next to move moved there
                new_board_positions = deepcopy(self.board_positions)
                new_board_positions[i][j] = self.to_go_next

                # yield this as a new game state where the user is next to move
                yield Game_State(
                    board_positions=new_board_positions,
                    moves_left=self.moves_left-1,
                    to_go_next="X" if self.to_go_next == "0" else "0"
                )
            # else continue to next iteration
```

I then produced a function to iterate through all empty squares on the board and yield the resulting game state if the server had moved there. This function returned an iterator of child Game_State objects

```
# this is so called as it used the minimax algorithm along with alpha beta pruning to navigate the decision tree
# the british museum algorithm is an approach where you explore the tree fully
# until all terminal nodes are game states where the game is over
# this function returns 2 values, the score and game state object for the best child game state (corresponds to best move)
def british_museum_minimax(game_state: Game_State, maximizing_player:bool, alpha, beta):
    # computer want 0 or -1 so minimizer
    # this is the base case to the recursive function (stop at terminal nodes)
    over, outcome = game_state.is_game_over()
    if over:
        return outcome, game_state

    # recursive case
    best_child= None

    # even though the initial call is for the servers move (minimizer), recursive calls may be for maximizer
    # I will explain the logic in more depth for the maximizing player
    if maximizing_player:
        # similar for minimizer
        else:
```

```

if maximizing_player:
    # meant to represent negative infinity (arbitrary bad so it is replaced by the best evaluation)
    max_evaluation = -100
    # for each possible game state
    for child in game_state.gen_child_game_states():
        # recursive class to function to evaluate child node
        # now minimizing player
        # index for first element as I only care about score not the actual game state
        evaluation = british_museum_minimax(
            game_state=child,
            maximizing_player=not maximizing_player,
            alpha=alpha,
            beta=beta,
        )[0]
        # max_evaluation = max(max_evaluation, evaluation)
        # if this evaluation is the best so far
        if evaluation > max_evaluation:
            # update the max evaluation and best child (only for this call)
            max_evaluation = evaluation
            best_child = child
            # update the alpha value to represent the best that the maximizing player can get
            alpha = max(alpha, evaluation)

        # if the minimizing player can do better (for them less) than alpha then they have a better option
        # this means that this won't be the best child of the earlier call
        # (the minimizer wouldn't need to give the maximizer such a good score)
        if beta <= alpha:
            break
    # return the best child game state and its score
    return max_evaluation, best_child

```

```

# similar for minimizer
else:
    min_evaluation = 100
    for child in game_state.gen_child_game_states():
        evaluation = british_museum_minimax(
            game_state=child,
            maximizing_player=not maximizing_player,
            alpha=alpha,
            beta=beta,
        )[0]
        # min_evaluation = min(min_evaluation, evaluation)
        if evaluation < min_evaluation:
            min_evaluation = evaluation
            best_child = child
            beta = min(beta, evaluation)
        if beta <= alpha:
            break
    return min_evaluation, best_child

```

The function is explained by the comments. It explores the decision tree down to the terminal nodes where games are over and then decides what the best move is. It returns the score and the object for the best child of the current game state object

```
# this main function is an easier wrapper for the app module to use
# it only determines and returns the best child (servers best move)
def main(game_state: Game_State):
    # output for debugging
    print("selecting best move from this game state:")
    game_state.print_board()

    # call the recursive function to determine the best child (and score for debugging)
    # start with arbitrarily low alpha value and high beta value
    score, best_child = british_museum_minimax(
        game_state=game_state,
        maximizing_player=False,
        alpha=-100,
        beta=+100
    )
    # more output for debugging
    print("best move is:")
    best_child.print_board()
    print(f"with a guaranteed score of {score} or better (minimizer)")

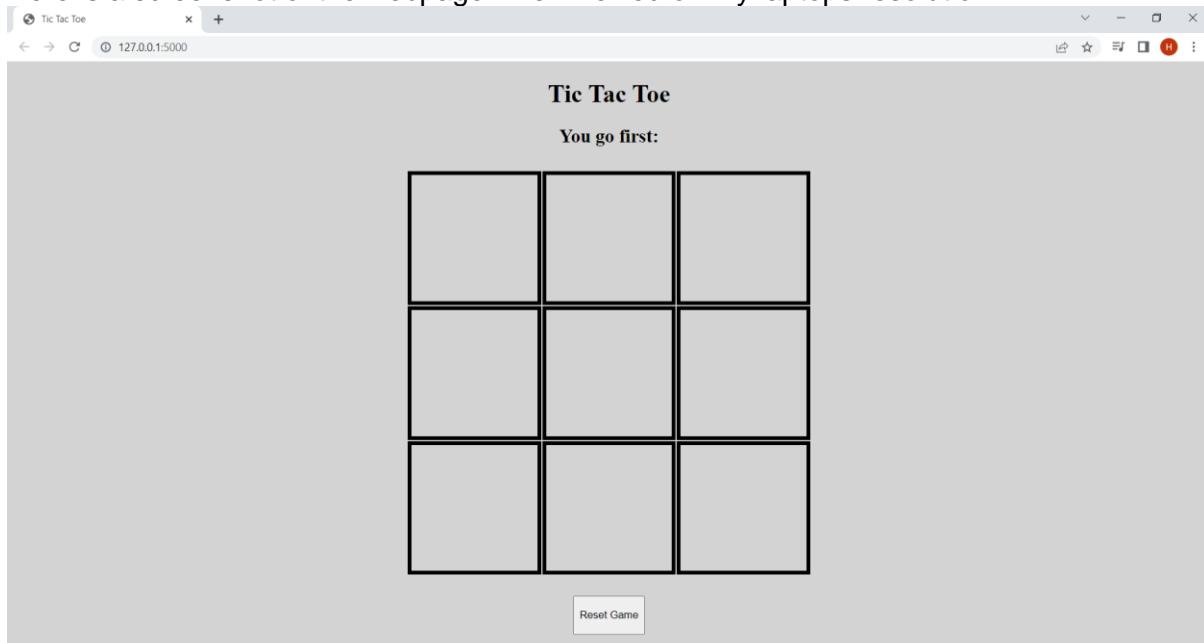
    # return the best child
    return best_child
```

I then made a main function for the whole module that provides abstraction from the minimax function. It takes a Game_State and returns the child game state corresponding to the best move. This is the function that the app.py program uses.

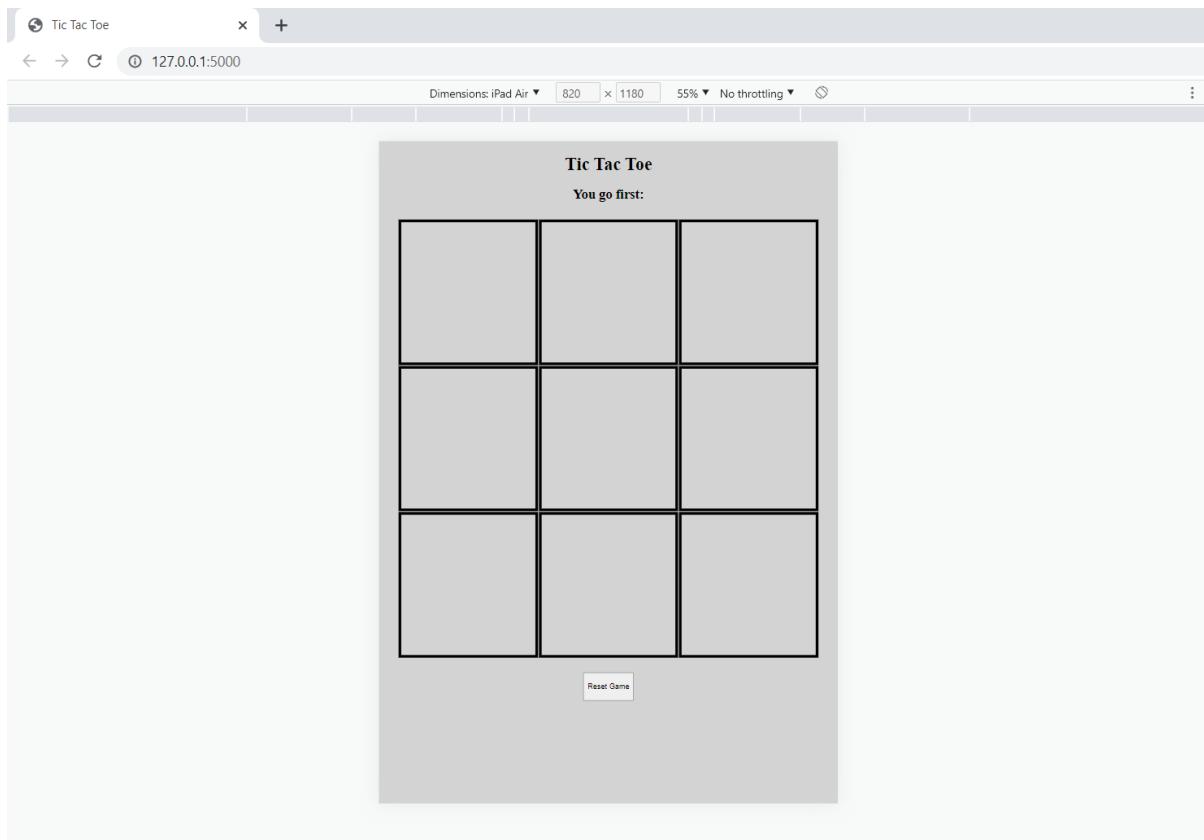
This is all the code used in the project.

Here are some screenshots of the final product:

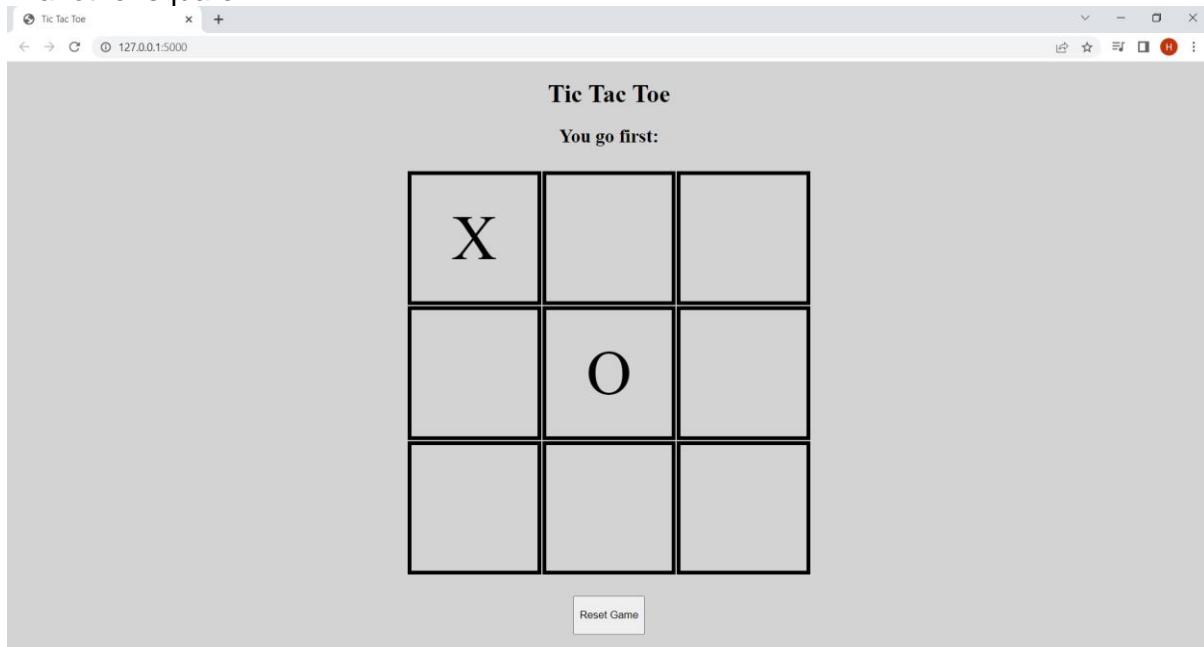
Here is a screenshot of the webpage when viewed on my laptops resolution



And here is a screenshot of the website from the resolution of the ipad

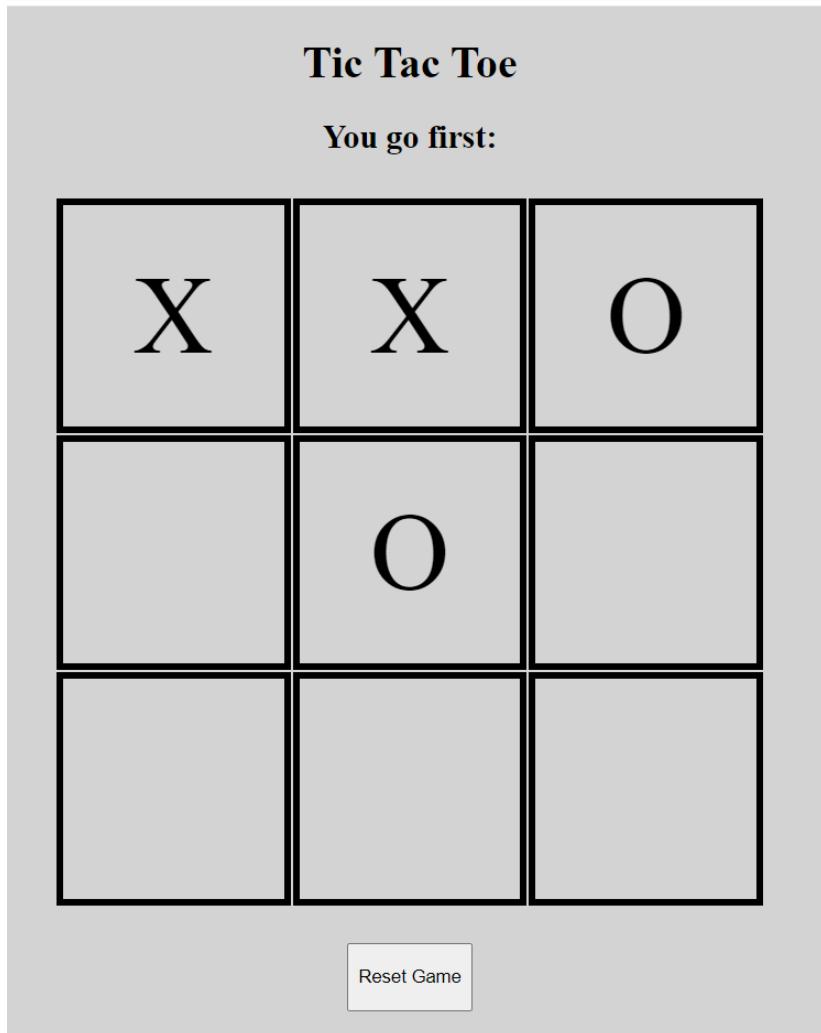


When I click on a square a 'X' Appears in it. Then a fraction of a second later a 'O' appears in another square

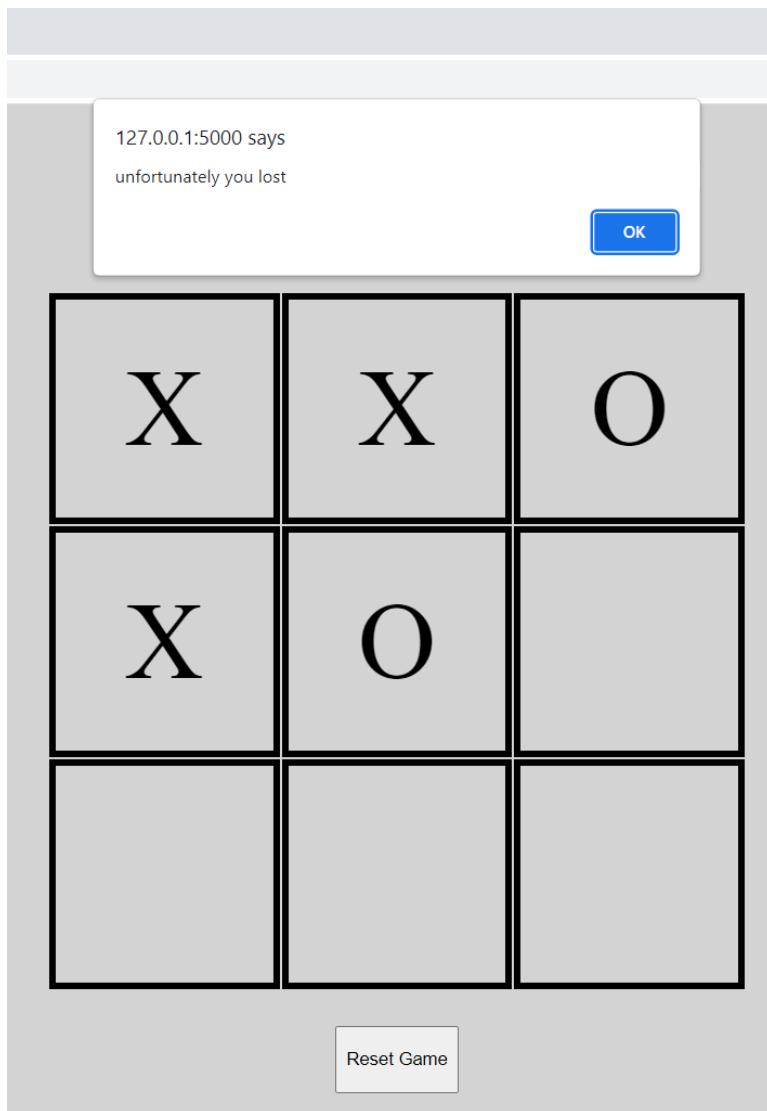


If I click one of the squares containing the 'X' or 'O', nothing happens due to client side validation

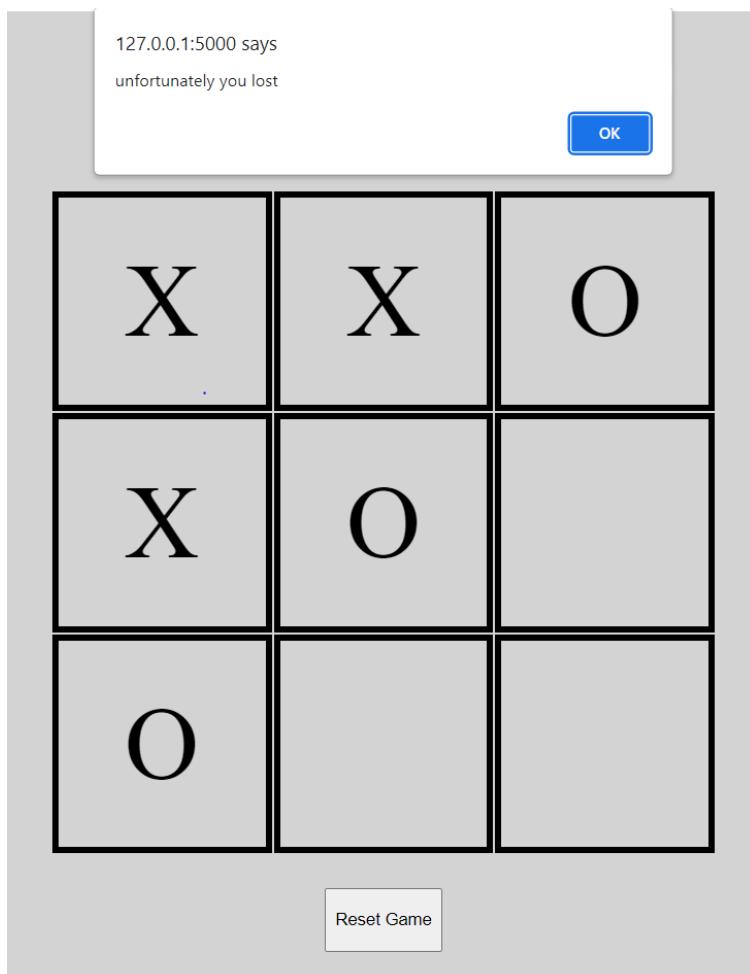
Below are a series of screenshots of a game where I allow the computer to win. (I will not show all of the webpage just the relevant part like the board as the rest is clear)



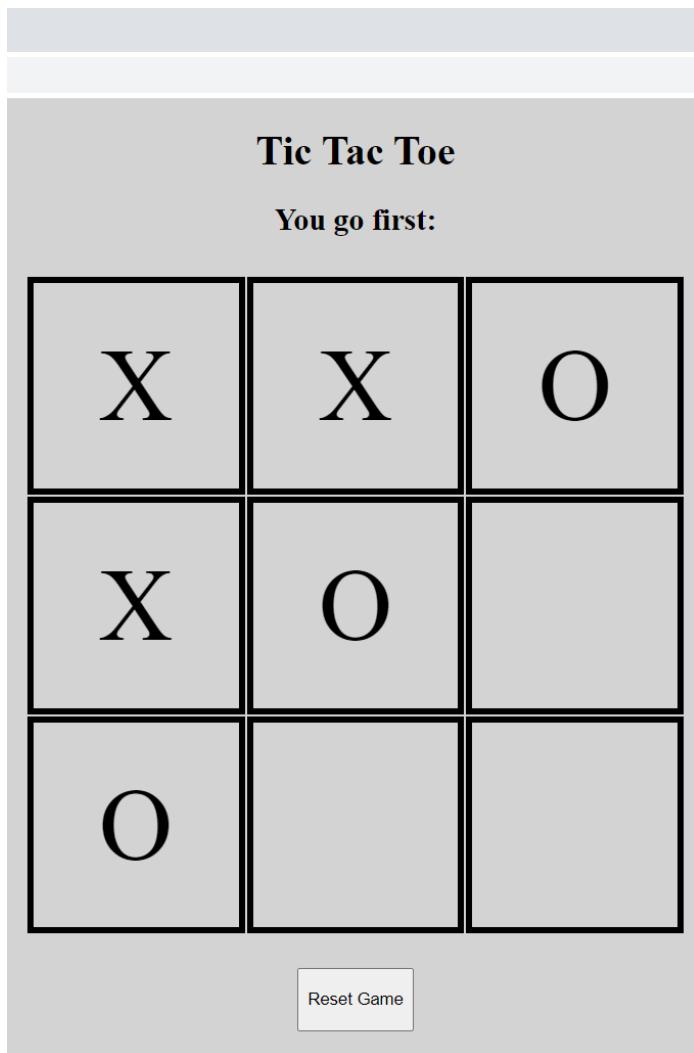
I then clicked the square in the 1st column and second row and this was the result



This shows a logic error that is minor but hard to deal with. It is where the alert to say that the game is over comes before the winning move (in this case a 'O' in the bottom left).



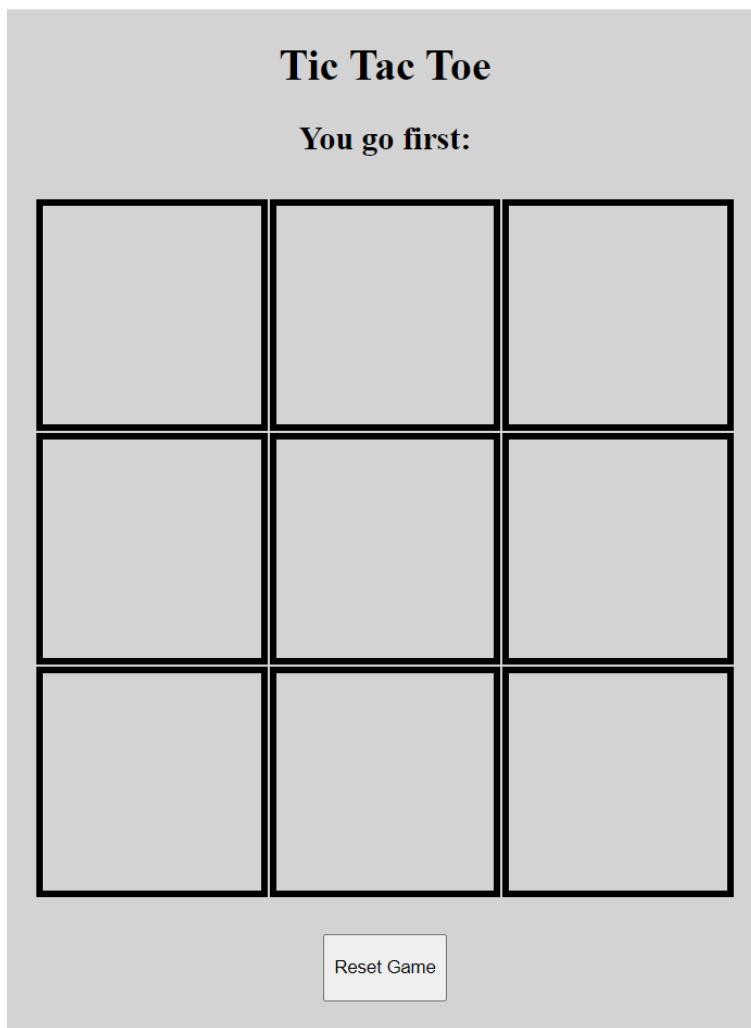
I noticed that if the window is minimised and then clicked again the winning move is shown



The winning move is also shown either way when the user clicks 'ok' on the alert. This is an issue but it doesn't prevent the program from functioning, it is more of a ruff edge.

Now if I click any square nothing happens as the game is over

If I click the reset button the board resets as shown



Problems throughout and how I tackled them:

While I did have to use a bit of trial and error to get my CSS to work the main problem I had in development was with my minimax function.

The process of developing this prototype was to make a rudimentary server that couldn't decide the best move. I then created the frontend and tested the program. The server would use command line input to allow me to input a move and simulate the minimax process before I wrote it. I then wrote my minimax function and found that it wasn't working.

I found that it was returning a move but not always the best move. It was strangely inconsistent and missed opportunities to win or stop the user winning at times. This was a challenge to debug as the minimax function is complex and has many recursive calls. For instance, for the first computer move it could have (without alpha beta pruning) $8!$ or 40320 recursive calls (total calls not depth). The challenge therefore was to try to make the function more transparent so I could see how it was behaving.

To do this I tried using breakpoints and stepping but this was not fruitful as it was hard to keep track of what the correct behaviour of each recursive call should be and I was unsure where the problem was. So I created a jupyter notebook which is a python environment where I can define cells which are blocks of code I can run manually. This allowed me to import the function and to see what the results were for a variety of outputs. Here is what I did:

I started by importing some libraries and making a copy of the invalid code (which I ran)

```
▶ from dataclasses import dataclass
  from copy import deepcopy

[26]                                         Python

@dataclass(frozen=True)
> class Game_State: ...

[27]                                         Python

# original function
> def british_museum_minimax(game_state: Game_State, maximizing_player: bool, alpha, beta): ...

[28]                                         Python
```

(code is hidden to save space as it is repeated from earlier)

I did this rather than import from the file so that my initial tests wouldn't change if I tried to fix the version in the file.

I then picked an example game state to try to capture the issue

```
▶ v score, best_child = british_museum_minimax(  
    game_state=Game_State(  
        moves_left=6,  
        to_go_next="0",  
        board_positions=[  
            ["0", "X", ""],  
            ["", "X", ""],  
            ["", "", ""],  
        ]  
    ),  
    maximizing_player=False,  
    alpha=-100,  
    beta=+100  
)  
[29] Python
```



```
score  
[30] Python
```



```
... 1  
▶ v best_child.print_board()  
[31] Python
```



```
... 0|X|0  
..|X|..  
..|.|.
```

This shows a game state where the computer could prevent the user from winning by placing an O in the bottom row and middle column. As we can see there is an issue as it fails to do this. The score of 1 shows that the minimax function believes that no matter what it does, from here if the user plays optimally it has lost (computer is minimiser).

```
> v
    score, best_child = british_museum_minimax(
        game_state=Game_State(
            moves_left=2,
            to_go_next="0",
            board_positions=[
                ["0", "X", "X"],
                ["0", "X", "0"],
                ["", "", "X"],
            ],
        ),
        maximizing_player=False,
        alpha=-100,
        beta=+100
    )
32]                                         Python

score
33]                                         Python
.. -1

best_child.print_board()
34]                                         Python
.. 0|X|X
  0|X|0
  0|.|X
```

A similar test showed that sometimes the algorithm works. Here it successfully determines that it can win by putting an O in the bottom left

This was helping me to build up a picture of when it fails

Perhaps the issue is with the game over function incorrectly determining who wins

+ Code + Markdown

```
Game_State(  
    moves_left=0,  
    to_go_next="0",  
    board_positions=[  
        ["0", "X", "X"],  
        ["0", "X", "0"],  
        ["X", "0", "X"],  
    ]  
).is_game_over()
```

[35]

Python

```
... (True, 1)
```

Thats correct

```
Game_State(  
    moves_left=0,  
    to_go_next="0",  
    board_positions=[  
        ["0", "X", "X"],  
        ["X", "0", "0"],  
        ["X", "0", "X"],  
    ]  
).is_game_over()
```

[36]

Python

```
... (True, 0)
```

thats also correct

```
Game_State(  
    moves_left=0,  
    to_go_next="0",  
    board_positions=[  
        ["X", "0", "0"],  
        ["X", "0", "X"],  
        ["0", "X", "0"],  
    ]  
).is_game_over()
```

Python

```
(True, -1)
```

this is also correct

I then investigated to check that the game over function was working as it should or if it was getting confused as to who won. It was working.

I then tried to visualize the decision tree that the minimax function was exploring

I will now try to make a tree to better visualize what is happening

```
@dataclass
class Tree:
    root_node_text: str
    sub_trees: list

    def print_tree(self, indent=0):
        # print(f"printing tree {self.root_node_text} with indent {indent}")
        print("|" + "---" * indent + f"-->{self.root_node_text}")
        for sub_tree in self.sub_trees:
            sub_tree.print_tree(indent=indent+1)
```

[38]

Python

```
> v Tree("A", [
    Tree("B", [
        Tree("E", [])
    ]),
    Tree("C", [
        Tree("D", [
            Tree("F", []),
            Tree("G", [])
        ])
    ]),
])
print_tree()
```

[39]

Python

```
.. |-->A
|----->B
|----->E
|---->C
|----->D
|----->F
|----->G
```

that isn't a great tree

This didn't work out and so I tried a different approach

maybe I could just add a load of print statements

```
[40] def print_dec(func):
    func_name = func.__name__
    def wrapper(*args, **kwargs):
        print(f"started: {func_name}(args={args}, kwargs={kwargs})")
        result = func(*args, **kwargs)
        print(f"finished: {func_name}(args={args}, kwargs={kwargs})  return value: {result}")
        return result

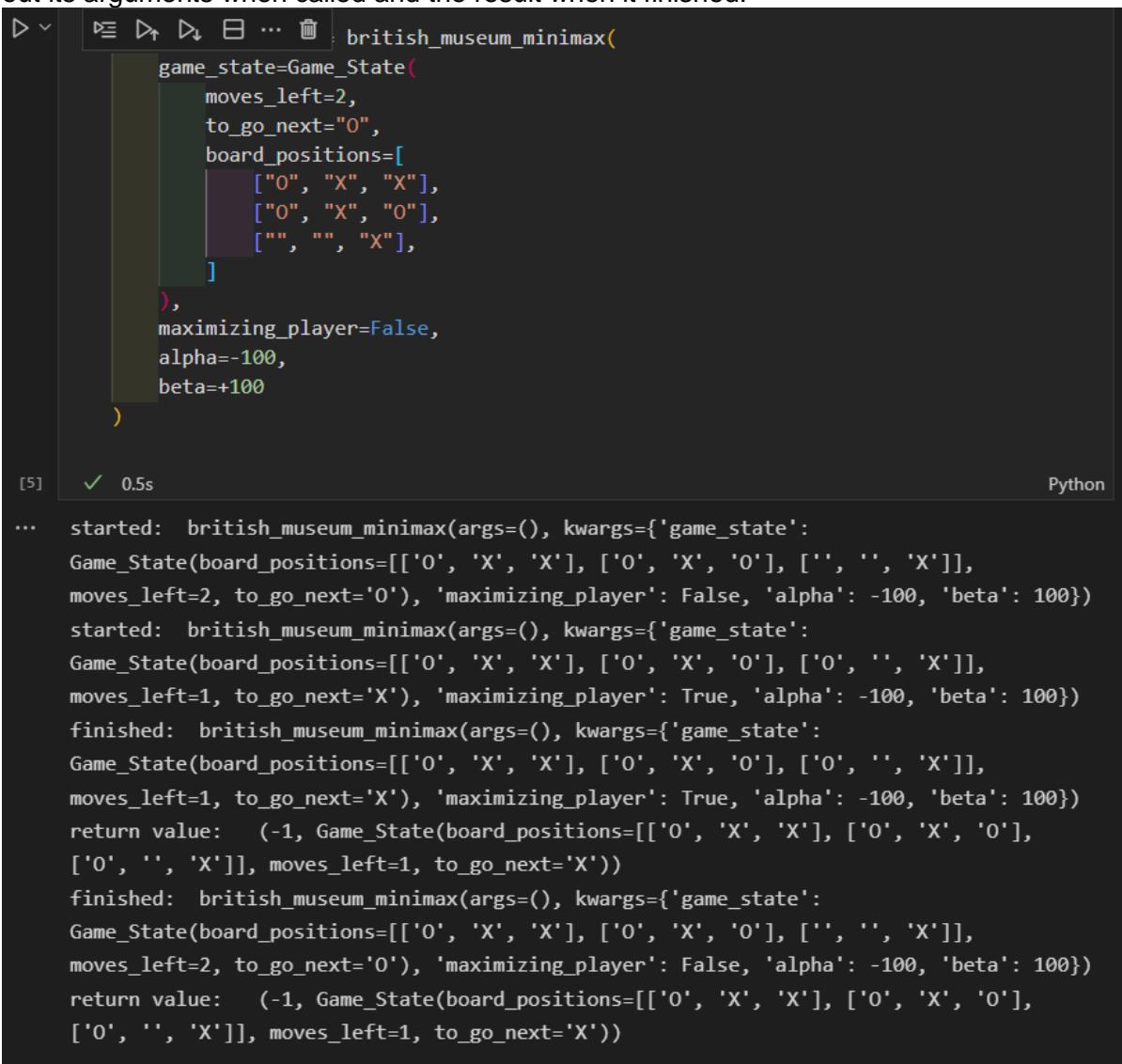
    wrapper.__name__ = func_name
    return wrapper
```

Python

```
[41] # new transparent version
@print_dec
> def british_museum_minimax(game_state: Game_State, maximizing_player: bool, alpha, beta): ...
```

Python

I then created a decorator to augment the function's behaviour. It caused the function to print out its arguments when called and the result when it finished.



The screenshot shows a Python code editor interface. At the top, there are several icons: a dropdown arrow, a file icon, a run icon, a search icon, a help icon, and a trash bin icon. Below these is the function definition:

```
british_museum_minimax(
    game_state=Game_State(
        moves_left=2,
        to_go_next="0",
        board_positions=[
            ["0", "X", "X"],
            ["0", "X", "0"],
            ["", "", "X"]
        ],
        maximizing_player=False,
        alpha=-100,
        beta=+100
    )
)
```

At the bottom left, there is a status bar with a green checkmark icon and the text "0.5s". On the right side, the word "Python" is displayed. The main pane below the code editor shows the execution output:

```
... started: british_museum_minimax(args=(), kwargs={'game_state': Game_State(board_positions=[['0', 'X', 'X'], ['0', 'X', '0'], ['', '', 'X']], moves_left=2, to_go_next='0'), 'maximizing_player': False, 'alpha': -100, 'beta': 100})
started: british_museum_minimax(args=(), kwargs={'game_state': Game_State(board_positions=[['0', 'X', 'X'], ['0', 'X', '0'], ['0', '', 'X']], moves_left=1, to_go_next='X'), 'maximizing_player': True, 'alpha': -100, 'beta': 100})
finished: british_museum_minimax(args=(), kwargs={'game_state': Game_State(board_positions=[['0', 'X', 'X'], ['0', 'X', '0'], ['0', '', 'X']], moves_left=1, to_go_next='X'), 'maximizing_player': True, 'alpha': -100, 'beta': 100})
return value: (-1, Game_State(board_positions=[['0', 'X', 'X'], ['0', 'X', '0'], ['0', '', 'X']], moves_left=1, to_go_next='X'))
finished: british_museum_minimax(args=(), kwargs={'game_state': Game_State(board_positions=[['0', 'X', 'X'], ['0', 'X', '0'], ['', '', 'X']], moves_left=2, to_go_next='0'), 'maximizing_player': False, 'alpha': -100, 'beta': 100})
return value: (-1, Game_State(board_positions=[['0', 'X', 'X'], ['0', 'X', '0'], ['', '', 'X']], moves_left=1, to_go_next='X'))
```

This worked as I used a near completion game so that there were fewer recursive calls. Due to pruning the there is only one recursive call as it successfully identifies the winning move.

```
[43] best_child.print_board()
Python
...
... 0|X|X
  0|X|0
  0|.|X

[44] score
Python
...
... -1
```

that makes sense, lets try to do this for one that doesn't work

This example seemed to all be in order

```
D ▾
score, best_child = british_museum_minimax(
    game_state=Game_State(
        moves_left=6,
        to_go_next="0",
        board_positions=[[
            ["0", "X", ""],
            ["", "X", ""],
            ["", "", ""]
        ]],
        maximizing_player=False,
        alpha=-100,
        beta=+100
    )
)

best_child.print_board()
print(score)
Python

[45]
...
... Output exceeds the size limit. Open the full output data in a text editor
started: british_museum_minimax(args=(), kwargs={'game_state':
Game_State(board_positions=[['0', 'X', ''], ['', 'X', ''], ['', '', '']], moves_left=6,
to_go_next='0'), 'maximizing_player': False, 'alpha': -100, 'beta': 100})
started: british_museum_minimax(args=(), kwargs={'game_state':
Game_State(board_positions=[['0', 'X', '0'], ['', 'X', ''], ['', '', '']], moves_left=5,
to_go_next='X'), 'maximizing_player': True, 'alpha': -100, 'beta': 100})
started: british_museum_minimax(args=(), kwargs={'game_state':
Game_State(board_positions=[['0', 'X', '0'], ['X', 'X', ''], ['', '', '']], moves_left=4,
to_go_next='0'), 'maximizing_player': False, 'alpha': -100, 'beta': 100})
started: british_museum_minimax(args=(), kwargs={'game_state':
Game_State(board_positions=[['0', 'X', '0'], ['X', 'X', '0'], ['', '', '']], moves_left=3,
to_go_next='X'), 'maximizing_player': True, 'alpha': -100, 'beta': 100})
started: british_museum_minimax(args=(), kwargs={'game_state':
Game_State(board_positions=[['0', 'X', '0'], ['X', 'X', '0'], ['X', '', '']], moves_left=2,
to_go_next='0'), 'maximizing_player': False, 'alpha': -100, 'beta': 100})
```

I then tried to look at the output for a game where the function had failed earlier. This was not fruitful as the output was too large to successfully analyse (6! Or 720 calls without pruning).

This is the end of my use of Jupyter notebook. It did however help me understand a little more what was happening when the program failed.

From coming back to the problem at a later date I realised that the client side JavaScript couldn't recognise a win when it was a diagonal in a row. I then fixed this on the frontend and backend by adding to the 'triplets' function. This still didn't account for some of the errors and it still wasn't working. From looking at my research into the minimax function I realised my mistake.

```
beta = min(alpha, evaluation)
```

the above line is the source of my logic error. It is contained in the else (so if minimiser) part of the algorithm. It is supposed to update beta with the best result the minimiser can so far obtain (to allow for pruning of clearly worse branches of the decision tree in future).

The correction is this:

```
beta = min(beta, evaluation)
```

With the help of my debugging and jupyter notebook this unassuming logic error was found. The correction allowed the program to work in all scenarios and play tic tac toe optimally. It is worth mentioning that I, nor my stakeholders, never beat this version of the minimax function at tic tac toe.

The other significant error I encountered was the following

```
PS C:\Users\henry\Documents\computing coursework\prototype 1> python app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 713-354-091
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - [07/Oct/2022 22:03:01] "GET /socket.io/?EIO=4&transport=polling&t=0EqGZPT HTTP/1.1" 200 -
127.0.0.1 - [07/Oct/2022 22:03:01] "POST /socket.io/?EIO=4&transport=polling&t=0EqGZQ_&sid=JgJR_iMrLEGYccaQAAAA HTTP/1.1" 200 -
127.0.0.1 - [07/Oct/2022 22:03:01] "GET /socket.io/?EIO=4&transport=polling&t=0EqGZR2&sid=JgJR_iMrLEGYccaQAAAA HTTP/1.1" 200 -
127.0.0.1 - [07/Oct/2022 22:03:01] "GET /socket.io/?EIO=4&transport=websocket&sid=JgJR_iMrLEGYccaQAAAA HTTP/1.1" 500 -
Traceback (most recent call last):
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/flask/app.py", line 2091, in __call__
    return self.wsgi_app(environ, start_response)
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/flask_socketio/__init__.py", line 43, in __call__
    return super(_SocketIOMiddleware, self).__call__(environ,
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/engineio/middleware.py", line 63, in __call__
    return self.engineio_app.handle_request(environ, start_response)
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/socketio/server.py", line 604, in handle_request
    return self.eio.handle_request(environ, start_response)
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/engineio/server.py", line 411, in handle_request
    packets = socket.handle_get_request()
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/engineio/socket.py", line 103, in handle_get_request
    return getattr(self, '_upgrade_' + transport)(environ,
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/engineio/socket.py", line 158, in _upgrade_websocket
    return ws(environ, start_response)
  File "C:/Users/henry/AppData/Local/Programs/Python/Python310/Lib/site-packages/engineio/async_drivers/gevent.py", line 35, in __call__
    raise RuntimeError('You need to use the gevent-websocket server.')
RuntimeError: You need to use the gevent-websocket server. See the Deployment section of the documentation for more information.
```

I struggled to solve this using Stackoverflow as their solution didn't work for me. I found that the program works if I run it in a different way using the following command

```
PS C:\Users\henry\Documents\computing coursework\prototype 1> python -m flask run
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
The WebSocket transport is not available, you must install a WebSocket server that is compatible with your async mode to enable it.
will be logged with level INFO)
127.0.0.1 - - [07/Oct/2022 22:05:05] "GET /socket.io/?EIO=4&transport=polling&t=0EqH1RB HTTP/1.1" 200 -
127.0.0.1 - - [07/Oct/2022 22:05:05] "POST /socket.io/?EIO=4&transport=polling&t=0EqH1gw&sid=XEtck_HfKGKRYIy7AAAA HTTP/1.1" 200 -
127.0.0.1 - - [07/Oct/2022 22:05:05] "GET /socket.io/?EIO=4&transport=polling&t=0EqH1gx&sid=XEtck_HfKGKRYIy7AAAA HTTP/1.1" 200 -
```

In future, rather than circumvent the error (as this still gives a warning), I will try to alter the async parameter of my socket object to fix this.

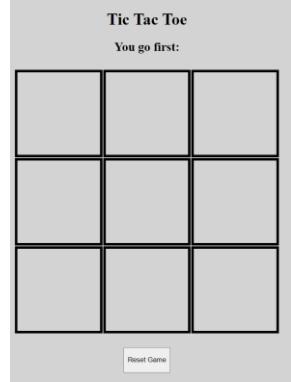
```
# setup sockets
socketio = SocketIO(app, async_mode=None)
# RuntimeError: You need to use the gevent-websocket server
# issue solved by running with 'python -m flask run'
```

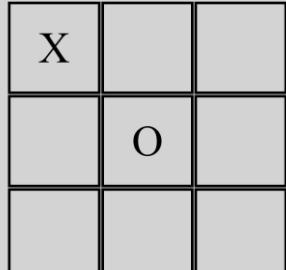
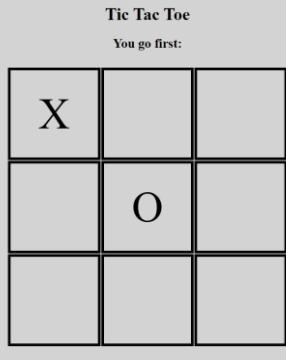
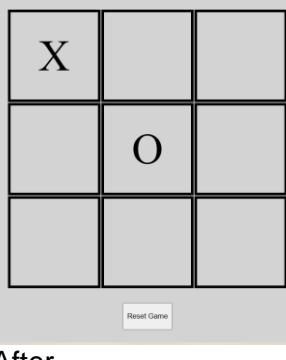
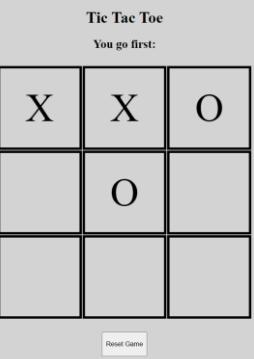
This does mean that I use an integrated terminal to run my program as running the python file directly doesn't work.

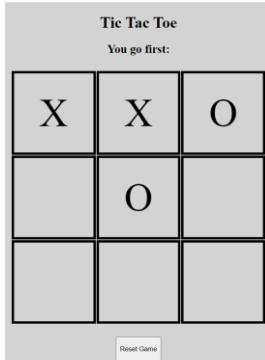
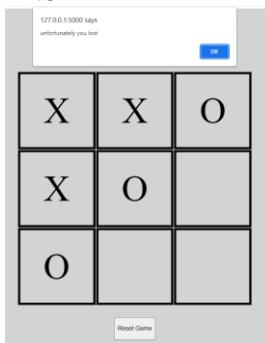
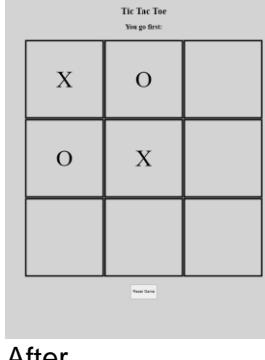
Test Plan for this version

The test plan for this program has 2 prongs. One set of tests will be testing that when I click on a specific square or button on the website, the expected behaviour happens. Another set of tests will be that in specific cases the tic tac program can see the correct move to win. These tests will be non-exhaustive as there are 9! Possible games which is too many to test.

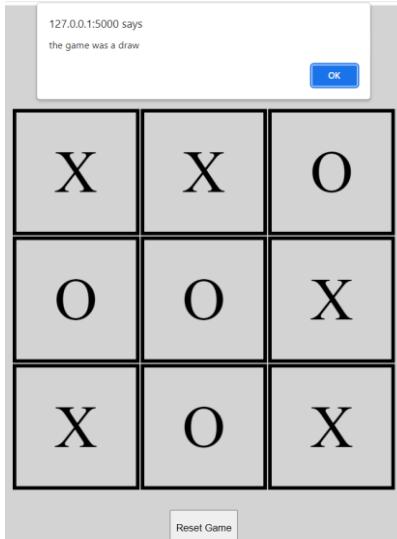
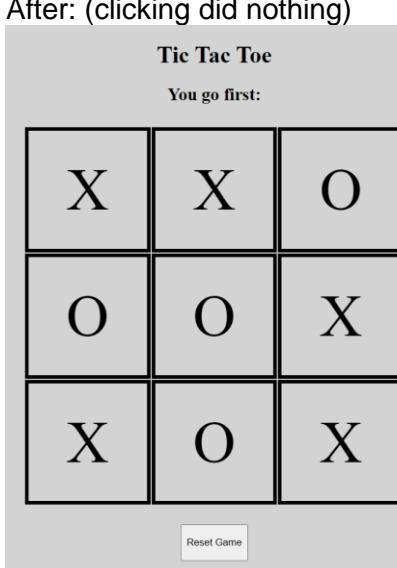
The below tests are shown in a table. I use numbers to indicate what screenshots are evidence to what test (usually 2 for before and after)

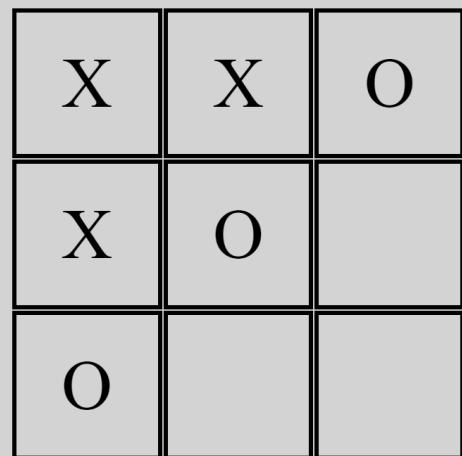
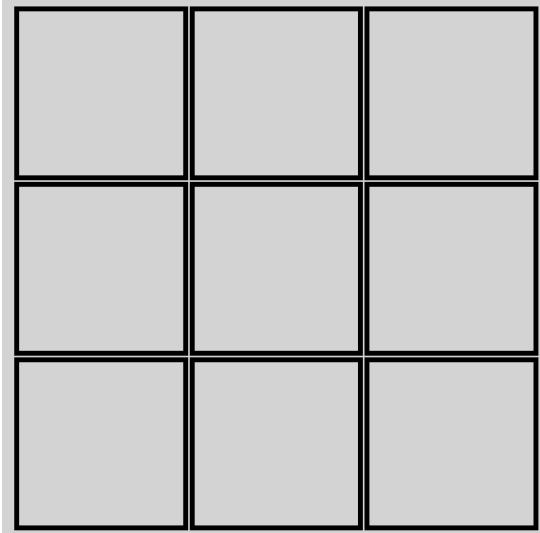
Description	Expected behaviour	Success ?	Evidence
When the game starts I will click one of the squares	That square should fill with a 'X' then another square should fill with a 'O'	Y	<p>Tried top left square. Before</p>  <p>After</p>

			<p>Tic Tac Toe You go first:</p>  <p>Reset Game</p>
2: I will then try to click both of the filled squares	Nothing should happen as validation should stop me moving there	Y	<p>Nothing to evidence but nothing did happen when I clicked the top left X or the centre O. the webpage still looked like this</p>  <p>Reset Game</p>
3: If I click on another empty square I can move there	An X should appear in that square and then an O should appear in another square	Y	<p>I moved to the top middle Before:</p>  <p>Reset Game</p> <p>After</p>  <p>Reset Game</p>

4: It should successfully recognise a loss	I should be able to click on another square and then the computer should move into the winning square. An alert should show that the computer has won	Y (but sometimes the alert happens before the move as already discussed)	<p>Before:</p>  <p>After:</p> 
5: The program should successfully recognise when the user wins. (Note this doesn't ever happen so I changed the JavaScript so that the game starts with the user a move away from winning and the user moves next)	The user should be able to move into the winning position. There should be no subsequent server move. An alert should show that the game is over	y	<p>Global variables changed:</p> <pre>// global constants to show game state // let moves_left = 9; // let next_to_go= "X"; // let board_positions= [// [',', ',', ''], // ['', ',', ''], // ['', ',', ''] //]; let moves_left = 5; let next_to_go= "X"; let board_positions= [['x', 'o', ''], ['o', 'x', ''], ['', '', '']];</pre> <p>Update widget to array added on load</p> <pre>// code to be executed when the page loads window.addEventListener('load', function(){} // for testing (if loading with a non blank game state) update_widget());</pre> <p>Before:</p>  <p>After</p>

6: The program should handle a draw correctly	If the user plays appropriately the game should continue until the user fills the 9 th square (none left empty). Then the game should end with an appropriate alert	y	<p>Before:</p> <p>After:</p>

7: once the game is over the board should be unresponsive	Once the game ends the user can click 'ok' on the alert and then clicking any square on the board should do nothing	Y	<p>Before:</p>  <p>After: (clicking did nothing)</p> 
8: the reset button should work once the game is over or halfway through	Clicking reset button should in effect reload the page to an empty board.	Fail (only worked in some cases). I address the [redacted] correction as a result after the tests	Case 1: the game is over loss Before:

Tic Tac Toe**You go first:**[Reset Game](#)**After:****Tic Tac Toe****You go first:**[Reset Game](#)**Before:**

Tic Tac Toe

You go first:

X	X	O
O	O	X
X	O	X

After:

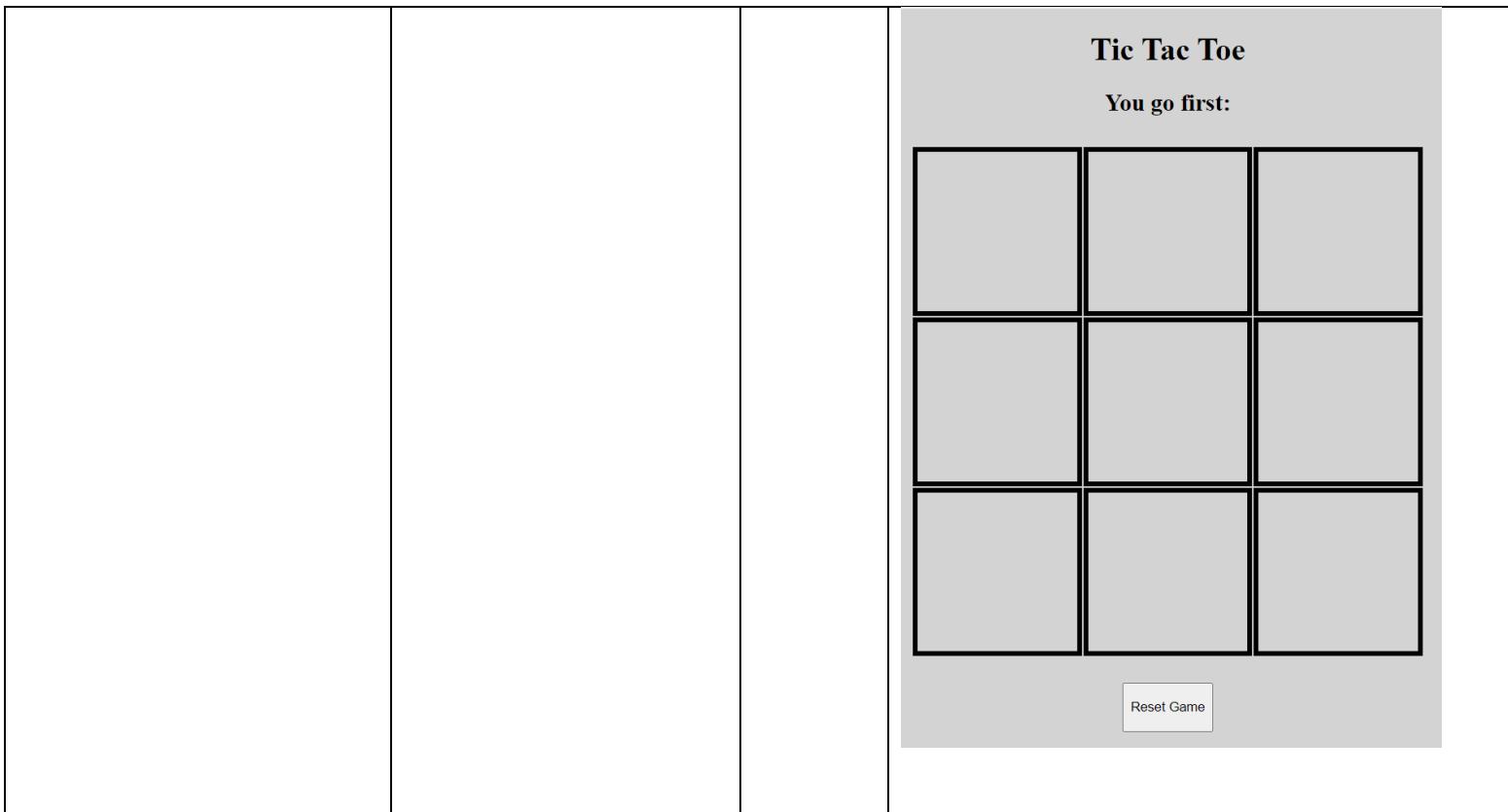
it didn't clear, the test failed. I reloaded the window and was able to recreate the issue

Case 2: The game is still going:

Before

X	X	O
	O	

After:

Changes to code made after testing:

So test number 8 or the reset button failed specifically when the game was over and it was a draw. I had recently added to that function and so immediately knew what the issue was. I found a logic error where a user makes a move and then immediately clicks reset, resulting in the server then responding with its move after the restart. I added an if statement to check if the program was waiting for the server's response and if so not to reset the board as highlighted here

```
function reset_game(){
    // reset globals
    // if the user has just in the last second moved and then clicked reset before the computers move
    // a logic error could occur where the game resets and then the server responds with its move
    // this is prevented here by ensuring that a server move isn't pending

    // LOGIC ERROR HERE, FAILS TO RESET WHEN DRAW
    if(next_to_go === "0"){
        return false
    }

    moves_left = 9;
    next_to_go = "X";
    board_positions = [
        ['', '', ''],
        ['', '', ''],
        ['', '', '']
    ];
    // update the html to reflect the board position matrix
    update_widget();
}
```

This caused the logic error. This is because there are 9 turns and the user makes all odd numbered turns, including the last turn which is 9. After the last turn in the case of a draw the game is over but the next to go has been changed to “O” after the move.

I initially thought I would correct this by adding an AND statement to the condition
If `next_to_go == "O"` AND `moves_left > 0`

However in the theoretical case where the user has won, the user would have made the winning play on their go and so next to go would be “O” and there would be moves left. Therefore I need to more generally check that next to go is “O” and that the game isn’t over. Here was my corrected code

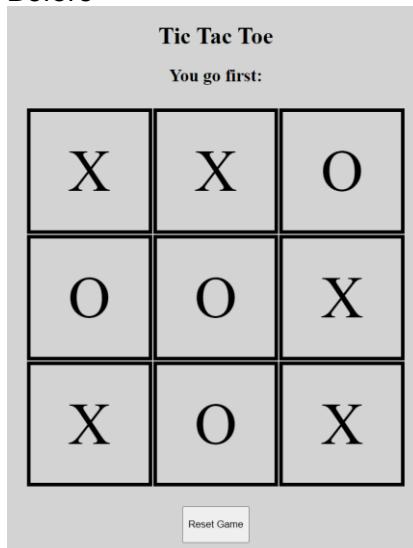
```
function reset_game(){
    // reset globals
    // if the user has just in the last second moved and then clicked reset before the computers move
    // a logic error could occur where the game resets and then the server responds with its move
    // this is prevented here by ensuring that a server move isn't pending

    // // // LOGIC ERROR HERE, FAILS TO RESET WHEN DRAW
    // // // if(next_to_go === "0"){
    // //     return false
    // // }

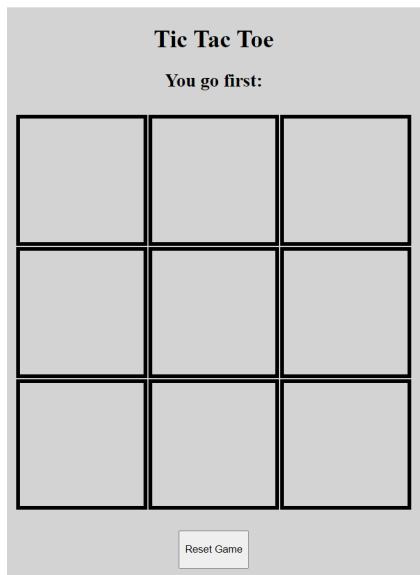
    // corrected code after reset button test
    if(next_to_go === "0" && !(is_game_over())){
        return false
    }

    moves_left = 9;
    next_to_go = "X";
    board_positions = [
        [ '', '', '' ],
        [ '', '', '' ],
        [ '', '', '' ]
    ];
    // update the html to reflect the board position matrix
    update_widget();
}
```

The test (test 8) now is successful. The reset button now works in all cases
Before



After:



Validation:

The validation is limited in this early prototype as there is a limited number of user inputs. Due to the lack of menus there are only really 2 types of input

- The user clicks the reset button
- The user clicks a square in the 3x3 grid

I have used validation to decide whether or not an action should be taken based on the current game state.

For the square click input:

The relevant action is to call the make_move function. The validation before-hand is that all the following conditions are met:

- The game isn't over
- It is the User's turn to make a move
- The move is a legal move (the target square is empty)

For the reset button click input:

The relevant action is to reset the global variable values and update the board in HTML to show this. The validation consists of a condition which if met causes the restart to be ignored. This check if both:

- It is the server's turn to move
- and the game isn't over

this prevents the user clicking to input as move and then immediately restarting. This would cause a logic error when the server's move is received and shown. I was not able to create this but this could happen if the server traffic or latency (network speed) meant that the server response time was slow. (This validation is the corrected version after a failed test).

Feedback from Stakeholder

I then received feedback from my stakeholders (namely my parents and friends). The feedback broadly focused on the user interface.

My parents told me midway through the process that the board should be larger so it fills the screen. I also increased the font size of the 'X' and 'O' characters in the squares so that they fill the boxes and are more obvious.

Then when the program was finished I got more feedback from friends:
Regarding the alerts

- They told me not to use alerts as they were too jarring. They interrupted the flow of the webpage and were not what they (the stakeholders) were used to.
- They also recognised the issue where sometimes the alert for the 'game is over' occurs before the last move is shown

Regarding the computer's move:

- They thought it would be a good idea to add a small delay before the computer moves so that the user has time to finish making their move, see the computers move, and react. They said that this would also make it seem more natural.
- They also said that it would be good if the text saying 'it's your turn' was dynamic and changed for when it was the computer's turn.

Other feedback included:

- Using a different background colour other than grey. This would help make the webpage more visually appealing and help differentiate the board in the foreground from the background
- There should be more of an effort to explain what the user should do when the page loads as it isn't necessarily intuitive.

The final piece of feedback was echoing my earlier concerns, as they said that it wasn't really a game as there was no way for them to win. In this way it wasn't fun.

Hopefully this issue will clear itself up when I move to a chess program. While it is an important technical achievement that my tic tac toe program is unbeatable, my chess program will be beatable. Not only will my algorithm not generate the theoretical best move but it will also be restricted in its thinking time as part of the difficulty settings.

It is worth also mentioning that I allowed my friends and parents to play with the program to help me create this feedback. In doing so it is important to recognise what they *didn't say* and what *didn't go wrong*.

- For one they at no point had any issue with the validation. Note this was before I accidentally created my reset button issue while trying last minute to tackle another logic error. This means that the validation was working (at least it was when they used it)
- They also didn't have any problem with the page loading or the server response times which means the WSGI HTTP and WebSocket servers were working.
- They also didn't have to deal with any crashes on the server side, resulting in the server failing to respond with a move, or any errors client side that prevented them from playing the game. This is good as it gives me confidence that the code is robust.

Planned changes after stakeholder feedback

There are a variety of changes that I plan to either implement in this prototype or (more likely) take into account when building prototype 2.

- I will not use alerts. In this case I could have used a hidden HTML element containing dynamically altered text to show that the game was over. This would also deal with the alert before last move problem.
- I will also try to have some consideration for colour. While it is a weak point of mine I can get feedback as I develop to make a less stark looking website.
- I will add a delay (if needed on top of thinking time) to the chess program so that it feels like the computer is thinking and doesn't break immersion

Evaluation

Overall I am happy with the prototype as I think that this iteration has been a useful stepping stone. Its primary goal was to get me started with the connection between the front end and backend (WebSocket) as well as the backend logic. This included the WSGI HTTP server and a basic version of the minimax function. It would have been exceedingly hard to tackle the problem of chess without tackling this simpler problem first. This prototype has been a success because it has made me feel more able (on a technical level) to tackle a simplified version of chess from my second prototype.

The feedback from stakeholders has also been clear and useful. My program has a rugged and industrial look as the user interface was not the primary goal. However despite this, the scrutiny of stakeholders as to how specifically it could be better has helped me to learn more about how to make my program more usable. I will add what I have learned to my intentions surrounding usability in my planning document. So the prototype met its primary goal of helping me to develop the backend but, to be clear, the front end user interface failed to be user friendly.

I accept that this prototype looks very basic and tic tac toe is far from chess. However another factor that has justified such 'small' steps to chess has been the complexity of learning full stack web development (front and back end integrated) and the heuristics used to tackle chess. This is evidenced in the informal and 'rough' research and planning into how the minimax algorithm and WebSocket worked (I have included it at the end).

One of my aims in the analysis and planning stage was to make my program suitable for a variety of users. I would do this by making it work for a variety of aspect ratios.

I used the chrome dev tools to change the aspect ratio between my laptop and an Ipad Air and decided that there were differences in the dimensions I preferred.

```
td{  
    border: 5px solid black;  
    overflow: hidden;  
  
    /* laptop */  
    width: 21vh;  
    height: 21vh;  
  
    /* ipad air */  
    /* width: 24vw;  
    height: 24vw; */  
    font-size: 80px;  
    text-align: center  
}
```

There are diminishing returns to refining a prototype when I could just bear the refinements in mind for the next prototype. One such refinement I would make if I had more time would be to add to the CSS so that it automatically used different styles for different aspect ratios.

Another change that I would have added had I had more time to refine is server side validation. Due to browsers having different versions of JS installed or users being able to circumvent client side validation, I would like to add server side validation. This could for example ensure that the client js wasn't miss reporting the current game state to be more favourable to the user. This could be done with sessions, ensuring that the client game state is a legal move away from what it was after the server last moved. This would be of particular importance if the incentive to cheat existed. This could occur if I added a trophy / rank system or user vs user matchups (both of which are low priority features I decided weren't suited to my target user in my analysis).

Iterative Development – Prototype 2

Aims for this iteration

The main aim for this iteration is to create a working chess engine. I may also attempt to start building the frontend interface but this goal is secondary. I then should be able to create a basic chess game that the user can play against the bot.

My chess engine should:

- be functional for basic chess but will not include special moves such as on passant, promotion and castling.
- at least have some user interface which is at minimum console based and allows the user to play a game of chess.
- have a minimax function that is efficient enough to run at depth 2 in a reasonable time.
- It should be rigorously tested to ensure that it is working and that there aren't any nasty surprises later down the line.

Functionality that the prototype will have

So to meet my main goal and successfully build a chess engine my end product must:

- Be able to correctly determine the legal moves available to a player, accounting for the different movements of pieces and check
- The engine should be able to identify check
- The engine should be able to give a static evaluation of each board state
- The engine should allow for a move to be executed: creating a new child board state that can be examined
- The engine should be able to identify then the game is over, who has won and the outcome type (stalemate, checkmate)

My minimax function should:

- Be able to beat a randomly moving opponent
- Have some consideration of efficiency and efficacy

My unit tests should:

- Be fully automated allowing them to be easily re-run to diagnose problems
- Be a mix of data driven and logic driven tests where appropriate
- My unit tests for minimax should be able to finish and so some efficiency is needed as it will need to undertake many games at depth 2 or 3

Annotated code screenshots with description

Quick note: I have annotated my code by adding comments explaining it in depth. Further comments are made with reference to specific function or classes in word.

I have not added comments explain my VUE GUI as it is now redundant and is no longer being developed (still useful as parts can be reused). I will also explain both the code and the tests in tandem as they were created in tandem (I test as I went along not at the end).

I first began by attempting to make my interface. This was not used in the final product but will be used in future iterations. I used VUE js to create the interface shown:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.1 vue page> cd chess_v2
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.1 vue page\chess_v2> npm run serve
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
```

```
> chess_v2@0.1.0 serve
> vue-cli-service serve
```

```
[]
```

DONE Compiled successfully in 7850ms

App running at:

- Local: <http://localhost:8080/>
- Network: <http://192.168.1.50:8080/>

Going to the local host, the webpage is running locally on my computer

The screenshot shows a chessboard with a light beige and dark brown square pattern. Eight white pawn icons are positioned on the second row from the bottom, and eight black pawn icons are on the second row from the top. Below the board is a row of eight squares, each containing a pawn icon, representing a pawn promotion row.

Turn 1: your turn

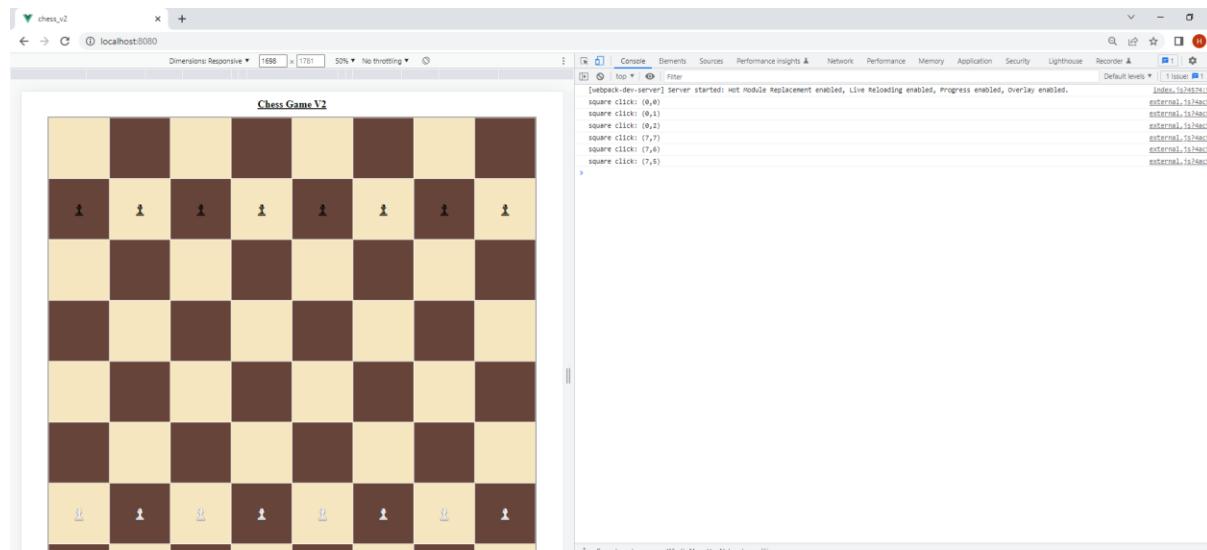
White Pieces Left: 2/6 Black Pieces Left: 3/6

White pieces left: Black pieces left:

Previous Moves:

White previous moves:	Black previous moves:
Move 1: B1 to B3	Move 2: A5 to A4
Move 3: B3 to A4	Move 4: E5 to E3

On clicking a square, a message is printed to the console to show that the correct square click is registered



The code for this user interface is below

```
<template>
    <h1>Chess Game V2</h1>
    <div :style="cssVars" class="chess_board">
        <table>
            <tr v-for="[row_num, row] in board.entries()" :key="row">
                <td v-for="[col_num, square] in row.entries()" :key="square" @click="square_click(row_num, col_num)">
                    <!-- {{row_num}}, {{col_num}} -->
                    <!-- {{square !== null? square: ""}} -->

                    <!-- playing around with white pieces -->
                    <!-- https://stackoverflow.com/questions/4772906/css-is-it-possible-to-add-a-black-outline-around-each-character-in-text -->
                    <span style="color:white; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-if="square == '♟'">♟</span>
                    <!-- <span style="color:black" v-if="square == '♟'">♟</span> -->
                    <!-- <span style="color:white" v-if="square == '♟'">♟</span> -->

                    <!-- <span style="color:black; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-else-if="square == '♙'">♙</span> -->
                    <span style="color:black" v-else-if="square == '♙'">♙</span>
                    <!-- <span v-else></span> -->
                </td>
            </tr>
        </table>
    </div>
```

```
<h2>Turn {{turn_num}}: {{next_to_go == 'user' ? "your turn": "computer's turn"}}</h2>

<div class="option_button">
    <button>Concede</button>
    <button>Reset</button>
</div>

<div class="pieces_left_table">
    <table>
        <tr>
            <th>White Pieces Left: {{pieces_left.black}}/6</th>
            <th>Black Pieces Left: {{pieces_left.white}}/6</th>
        </tr>
        <tr>
            <td>{{ '■'.repeat(pieces_left.white) }}</td>
            <td>{{ '■'.repeat(pieces_left.black) }}</td>
        </tr>
    </table>
</div>

<h2>Previous Moves:</h2>

<div class="previous_moves_table">
    <table>
        <tr>
            <th>White previous moves:</th>
            <th>Black previous moves:</th>
        </tr>
        <tr>
            <td>
                <div v-for="move in previous_moves" :key="move.num">
                    <span v-if="move.player=='white'">
                        Move {{move.num}}: ♕ {{ move.from }} to {{ move.to }}
                    </span>
                </div>
            </td>
            <td>
                <div v-for="move in previous_moves" :key="move.num">
                    <span v-if="move.player=='black'">
                        Move {{move.num}}: ♜ {{ move.from }} to {{ move.to }}
                    </span>
                </div>
            </td>
        </tr>
    </table>
</div>
```

```
        </tr>
    </table>
</div>

</template>

<script>
    import {handle_square_click} from '@/assets/scripts/external.js'
    // import * from '@/assets/scripts/external.js'
    const white_sq_color = '#f5e6bf';
    const black_sq_color = '#66443a';
    // white pawn: '♟'; black pawn: '♙'
    const pawn_white = '♙';
    const pawn_black = '♟';

    export default {
        name: "ChessGame",
        ready_for_next_move: false,
        // pawn characters
        // https://en.wikipedia.org/wiki/Chess_symbols_in_Unicode
        data(){return{
            board: [
                Array(8).fill(null),
                Array(8).fill(pawn_black),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(pawn_white),
                Array(8).fill(null)
            ],
            next_to_go: 'user',
            turn_num: 1,
            pieces_left: {
                black: 2,
                white: 3
            },
            previous_moves: [
                { num: 1, player: "white", from: "B1", to: "B3" },
                { num: 2, player: "black", from: "A5", to: "A4" },
                { num: 3, player: "white", from: "B3", to: "A4" },
                { num: 4, player: "black", from: "E5", to: "E3" },
            ],
        }},
        methods: {
            square_click(row, col) {
                // console.log(`square click: (${row},${col})`)
                handle_square_click(row, col)
            }
        }
    }
</script>
```

```
        }
    },
    // https://www.telerik.com/blogs/passing-variables-to-css-on-a-
vue-component
    computed: {
        cssVars(){return{
            '--black_sq_color': black_sq_color,
            '--white_sq_color': white_sq_color,
        };}
    }
}
</script>

<style scoped>
/* :root {
    https://www.vectorstock.com/royalty-free-vector/chess-field-in-
beige-and-brown-colors-vector-24923385
    https://imagecolorpicker.com/en
    --black_sq_color: #66443a;
    --white_sq_color: #f5e6bf;
} */


h1 {
    text-align: center;
    text-decoration: underline;
}
h2 {
    text-align: center;
    font-size: 32px;
}
table {
    margin: auto;
    table-layout: fixed;
    text-align: center;
}

.option_button {
    margin: auto;
    text-align: center;
    /* margin-left: 0.125vw;
    margin-right: 0.125vw; */
    margin-left: 20px;
    margin-right: 20px;
}

.chess_board td {
    height: 10.5vh;
    width: 10.5vh;
```

```

        /* dimension so it is all in view for ipad air portrait  (820 *
1180) */
        /* height: 12vw;
width: 12vw; */
text-align: center;
/* border: 3px black solid; */
font-size: 50px;
/* font-weight: 100; */

}

.chess_board table {
    border: 1px black solid;
}

/* these odd and even rules dictate the color of chess board squares */
.chess_board tr:nth-child(2n-1) > td:nth-child(2n-1) {
background-color: var(--white_sq_color);
/* color: black; */
}
.chess_board tr:nth-child(2n-1) > td:nth-child(2n) {
background-color: var(--black_sq_color);
/* color: white; */
}
.chess_board tr:nth-child(2n) > td:nth-child(2n-1) {
background-color: var(--black_sq_color);
/* color: white; */
}
.chess_board tr:nth-child(2n) > td:nth-child(2n) {
background-color: var(--white_sq_color);
/* color: black; */
}

.previous_moves_table th, td {
width: 35vw;
/* border: 3px solid black; */
font-size: 28px;
}
.pieces_left_table th, td {
width: 35vw;
/* border: 3px solid black; */
font-size: 28px;
}
</style>
```

It is not fully finished as the idea of developing a VUE js GUI was shelved.

On the one hand there are many benefits to a VUE GUI. Most importantly, the html content is dynamically updated from JavaScript variables. For example the board shown

automatically updates with the content of the 2d array board. The main downside however is that the single page nature of a VUE webpage clashed with what I wanted the flask server to do. This is where one single HTTP request is sent and instead multiple pages are handled by JavaScript loading different components. I used this source and decided that I wanted to implement my user interface like this:

<https://www.digitalocean.com/community/tutorials/how-to-add-authentication-to-your-app-with-flask-login>

I then moved on to implementing the backend chess engine in python. I knew that I intended to implement my chess engine using decision trees and specifically the minimax function. This function is suitable as chess is a complete information game as there is no information hidden such as a hand of cards. It is also a zero sum game as one player's good move worsens another player's chance of winning. It is also turn based.

To implement minimax I would need 2 functions. A utility function to determine a number for how favourable the current game state is to the maximiser (utility) and a child game state generating function that could provide all possible game states that could be achieved in one move from the current game state.

To implement the utility function I would need a robust game over function and a static evaluation function to provide an approximate guess as to the utility of a board state.

To implement the child game state generating function I will need to have a function to action a move and generate a child game state as well as a function to generate all legal moves of the current game state.

These function would need to be rigorously tested. If they contained logic errors, then the minimax function would fail without an easy way to determine why. However these functions themselves are hard to implement and so they would need to be broken down into simpler functions. I clearly would also need to employ some high quality testing in order to produce sufficiently reliable code.

I decided that at its most abstracted, I would be solving this problem with a series of objects built on vectors and rules. So I began by creating a vector class. It was sufficiently simple that it didn't need any major debugging. This would still provide me a good opportunity to get to grips with my unit-test process.

I decided to use a library called data driven tests (ddt) as it would let me write a single function to execute a single type of test, then run this test for every set of test data specified in a given file.

I thought that this would be better than generating the test data and expected outcome within the test function as this method is more transparent and reduces the risk of logic errors within the tests themselves.

Here was my vector class (**vector.py**)

```
# import dataclass to reduce boilerplate code
from dataclasses import dataclass

# frozen = true means that the objects will be immutable
@dataclass(frozen=True)
class Vector():
```

```
# 2d vector has properties i and j
i: int
j: int

# code to allow for + - and * operators to be used with vectors
def __add__(self, other):
    assert isinstance(other, Vector), "both objects must be instances
of the Vector class"
    return Vector(
        i=self.i + other.i,
        j=self.j + other.j
    )
def __sub__(self, other):
    assert isinstance(other, Vector), "both objects must be instances
of the Vector class"
    return Vector(
        i=self.i - other.i,
        j=self.j - other.j
    )
def __mul__(self, multiplier: int):
    return Vector(
        i=self.i * multiplier,
        j=self.j * multiplier
    )

# check if a vector is in board
def in_board(self):
    """Assumes that the current represented vector is a position
vector
    checks if it points to a square that isn't in the chess board"""
    return self.i in range(8) and self.j in range(8)

# alternative way to create instance, construct from chess square
@classmethod
def construct_from_square(cls, to_sqr):
    """Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1,
3)"""
    to_sqr = to_sqr.upper()
    letter, number = to_sqr

    # map letters and numbers to 0 to 7 and create new vector object
    return cls(
        i=ord(letter.upper()) - ord("A"),
        j=int(number)-1
    )
```

```

# this function is the reverse and converts a position vector to a
square
def to_square(self) -> str:
    letter = chr(self.i + ord("A"))
    number = self.j+1
    return f"{letter}{number}"

# this function checks if 2 vectors are equal
def __eq__(self, other) -> bool:
    try:
        # assert same subclass like rook
        assert isinstance(other, type(self))
        assert self.i == other.i
        assert self.j == other.j

    except AssertionError:
        return False
    else:
        return True

# used to put my objects in set
def __hash__(self):
    return hash((self.i, self.j))

```

and here were my tests
 starting with the python file containing the logic to run the unit tests (**test_vector.py**)

```

import ddt
import unittest

# from vector.vector import Vector
from vector import Vector

# function for path to vector related test data
def test_path(file_name):
    return f"./test_data/vector/{file_name}.yaml"

# augment my test case class with ddt decorator
@ddt.ddt
class Test_Case(unittest.TestCase):

    # performs many checks of construct form square
    @ddt.file_data(test_path("from_square"))
    def test_square_to_vector(self, square, expected_vector):
        self.assertEqual(
            Vector.construct_from_square(square),
            Vector(*expected_vector),

```

```

        msg=f"\nVector.construct_from_square('{square}') != Vector(i
={expected_vector[0]}, j={expected_vector[1]})"
    )

# performs many checks of adding vectors
@ddt.file_data(test_path("vector_add"))
def test_add_vectors(self, vector_1, vector_2, expected_vector):
    self.assertEqual(
        Vector(*vector_1) + Vector(*vector_2),
        Vector(*expected_vector),
        msg=f"\nVector(*{vector_1}) +
Vector(*{vector_2}) != Vector(*{expected_vector})"
    )

# performs many checks of multiplying vectors
@ddt.file_data(test_path("vector_multiply"))
def test_multiply_vectors(self, vector, multiplier, expected):
    self.assertEqual(
        Vector(*vector) * multiplier,
        Vector(*expected)
    )

# many tests of vector in board
@ddt.file_data(test_path("vector_in_board"))
def test_in_board(self, vector, expected):
    self.assertEqual(
        Vector(*vector).in_board(),
        expected,
        msg=f"\nVector(*{vector}).in_board() != {expected}"
    )

# many tests of vector out of board
@ddt.file_data(test_path('vector_to_square'))
def test_to_square(self, vector, expected):
    self.assertEqual(
        Vector(*vector).to_square(),
        expected
    )

if __name__ == '__main__':
    unittest.main()

```

Here are the test data files:

from_square.yaml

```
test1:
  square: 'G3'
```

```
expected_vector: [6, 2]

test2:
  square: 'A8'
  expected_vector: [0, 7]

test3:
  square: 'H1'
  expected_vector: [7, 0]

test4:
  square: 'C4'
  expected_vector: [2, 3]

test5:
  square: 'F6'
  expected_vector: [5, 5]

# # invalid
# test6:
#   square: 'A5'
#   expected_vector: [5, 5]
```

Vector_add.yaml

```
test1:
  vector_1: [1, 4]
  vector_2: [4, 6]
  expected_vector: [5, 10]

test2:
  vector_1: [1, 7]
  vector_2: [4, 6]
  expected_vector: [5, 13]

test3:
  vector_1: [1, 3]
  vector_2: [0, 6]
  expected_vector: [1, 9]

test4:
  vector_1: [-1, 32]
  vector_2: [3, 6]
  expected_vector: [2, 38]

test5:
  vector_1: [6, 0]
```

```
vector_2: [6, 7]
expected_vector: [12, 7]

# # invalid delete me
# test6:
#   vector_1: [6, 0]
#   vector_2: [6, 7]
#   expected_vector: [0, 7]
```

Vector_in_board.yaml

```
test1:
  vector: [0, 0]
  expected: True

test2:
  vector: [7, 0]
  expected: True
test3:
  vector: [0, 7]
  expected: True
test4:
  vector: [7, 7]
  expected: True

test5:
  vector: [4, 6]
  expected: True

test6:
  vector: [3, 2]
  expected: True

test7:
  vector: [-1, -1]
  expected: False

test8:
  vector: [-5, 4]
  expected: False

test9:
  vector: [8, 7]
  expected: False

test10:
```

```
vector: [-4, 4]
expected: False

test11:
vector: [10, 4]
expected: False

# adding comment inside causes logic error and false pass on test
# # invalid delete me
# test:
#   vector: [4, 6]
#   expected: True
```

Vector_multiply.yaml

```
test1:
vector: [1, 1]
multiplier: 1
expected: [1, 1]
test2:
vector: [1, 1]
multiplier: -1
expected: [-1, -1]
test3:
vector: [1, 1]
multiplier: 5
expected: [5, 5]
test4:
vector: [5, 0]
multiplier: 0
expected: [0, 0]
```

vector_to_square.yaml

```
test1:
vector: [0, 0]
expected: A1
test2:
vector: [5, 7]
expected: F8
test3:
vector: [6, 2]
expected: G3
test4:
vector: [0, 6]
expected: A7
test5:
```

```
vector: [7, 7]
expected: H8
```

as can be seen with the console:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -i test_vector.py
>>> list(filter(lambda name: name[:5] == "test_", dir(Test_Case)))
['test_add_vectors_1_test1', 'test_add_vectors_2_test2', 'test_add_vectors_3_test3', 'test_add_
vectors_4_test4', 'test_add_vectors_5_test5', 'test_in_board_01_test1', 'test_in_board_02_test2
', 'test_in_board_03_test3', 'test_in_board_04_test4', 'test_in_board_05_test5', 'test_in_board
_06_test6', 'test_in_board_07_test7', 'test_in_board_08_test8', 'test_in_board_09_test9', 'test
_in_board_10_test10', 'test_in_board_11_test11', 'test_multiply_vectors_1_test1', 'test_multipl
y_vectors_2_test2', 'test_multiply_vectors_3_test3', 'test_multiply_vectors_4_test4', 'test_squ
are_to_vector_1_test1', 'test_square_to_vector_2_test2', 'test_square_to_vector_3_test3', 'test
_square_to_vector_4_test4', 'test_square_to_vector_5_test5', 'test_to_square_1_test1', 'test_to
_square_2_test2', 'test_to_square_3_test3', 'test_to_square_4_test4', 'test_to_square_5_test5']

>>> ]
```

The decorators I added to my tests are unusual as they return many mutations of my original generic function that complete a specific test from my yaml file. This allows me to perform targeted testing

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest
-----
Ran 74 tests in 0.301s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_vector
-----
Ran 30 tests in 0.005s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_vector.Test_Case.test_to_square_4_test4
.
-----
Ran 1 test in 0.000s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> ]
```

As is shown above, I can run all test, just vector tests or a specific test form a yaml file. I know the tests are working as I can change some of the tests data to create a test that I expect to fail.

For example:

```
# adding comment inside causes logic error and false pass on test
# invalid delete me
test:
    vector: [4, 6]
    expected: False
```

I have uncommented out this invalid test in the **vector_in_board.yaml** file
When I run the tests:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_vector
....F.....
=====
FAIL: test_in_board_12_test (test_vector.Test_Case)
test_in_board_12_test
-----
Traceback (most recent call last):
  File "C:\Users\henry\AppData\Local\Programs\Python\Python310\lib\site-packages\ddt.py", line 220, in wrapper
    return func(self, *args, **kwargs)
  File "C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\test_vector.py", line 44, in test_in_board
    self.assertEqual(
AssertionError: True != False :
Vector(*[4, 6]).in_board() != False

-----
Ran 31 tests in 0.007s

FAILED (failures=1)
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> █
```

There is a summary of which of the tests fail and a traceback, including a message to help determine why one of the tests failed.

As part of testing my tests in this way I realised that my tests weren't working. I determined that this was because I had named them all test in my yaml file. To fix this I named them test1, test2, ect.

I think it is important to document tests and code at the same time as they were developed in tandem. I didn't move to code the next stage until all of my vector unit tests were working. This was valuable as it allowed me to use the vector module with the assumption that it was working perfectly in unit tests and code for later modules that built upon my vectors module. The next logic to code was the various rules for how pieces could move on a chess board. To do this I created a pieces module.

I wanted to make a class for each piece to describe its value (needed for the utility function) and the ways in which it can move within a chess board. As I knew that some logic would be repeated within the piece classes I decided to have a parent class piece and a series of child classes that inherit from it. Since I also knew that I would want to do general operations on a set of pieces later on (such as iterate through them for movement vectors or to sum up their value) I decided to use an abstract base class. This ensures that all of the piece classes have some key attributes and methods (they share the same interface). This means that I won't need to differentiate between pieces and have different logic for each one (with the exception of the king).

I found a series of standard values and matrices for determining addition value based on location within the board on the following website

https://www.chessprogramming.org/Simplified_Evaluation_Function

They were not in a useful form (e.g. csv download):

Bishops

```
// bishop
-20,-10,-10,-10,-10,-10,-10,-20,
-10, 0, 0, 0, 0, 0, -10,
-10, 0, 5, 10, 10, 5, 0, -10,
-10, 5, 5, 10, 10, 5, 5, -10,
-10, 0, 10, 10, 10, 10, 0, -10,
-10, 10, 10, 10, 10, 10, 10, -10,
-10, 5, 0, 0, 0, 0, 5, -10,
-20,-10,-10,-10,-10,-10,-10,-20,
```

We avoid corners and borders. Additionally we prefer lists.
To solve this I used a jupyter notebook to process them

```
def dec_gen_to_list(generator_func):
    def wrapper(*args, **kwargs):
        return list(
            generator_func(*args, **kwargs)
        )
    wrapper.__name__ = generator_func.__name__
    return wrapper
```

```
@dec_gen_to_list
def str_to_matrix(str_matrix):
    @dec_gen_to_list
    def str_to_row(row):
        for e in row.split(","):
            if not e:
                continue
            e = e.strip()
            yield int(e)
            # print(f'{e}')
    for row in str_matrix.split("\n"):
        if not row:
            continue

        # print(f'row: {row}')
        yield str_to_row(row)
```

```

pawn = str_to_matrix(""" 0,  0,  0,  0,  0,  0,  0,  0,
50, 50, 50, 50, 50, 50, 50, 50,
10, 10, 20, 30, 30, 20, 10, 10,
| 5,  5, 10, 25, 25, 10, 5,  5,
| 0,  0,  0, 20, 20, 0,  0,  0,
| 5, -5,-10, 0,  0,-10, -5,  5,
| 5, 10, 10,-20,-20, 10, 10,  5,
| 0,  0,  0,  0,  0,  0,  0,  0""")
pawn

```

Python

```

[[0, 0, 0, 0, 0, 0, 0, 0],
[50, 50, 50, 50, 50, 50, 50, 50],
[10, 10, 20, 30, 30, 20, 10, 10],
[5, 5, 10, 25, 25, 10, 5, 5],
[0, 0, 0, 20, 20, 0, 0, 0],
[5, -5, -10, 0, 0, -10, -5, 5],
[5, 10, 10, -20, -20, 10, 10, 5],
[0, 0, 0, 0, 0, 0, 0, 0]]

```

```

knight = str_to_matrix("""-50,-40,-30,-30,-30,-40,-50,
-40,-20, 0, 0, 0, -20,-40,
-30, 0, 10, 15, 15, 10, 0,-30,
-30, 5, 15, 20, 20, 15, 5,-30,
-30, 0, 15, 20, 20, 15, 0,-30,
-30, 5, 10, 15, 15, 10, 5,-30,
-40,-20, 0, 5, 5, 0,-20,-40,
-50,-40,-30,-30,-30,-40,-50""")
knight

```

Python

```

[[-50, -40, -30, -30, -30, -40, -50],
[-40, -20, 0, 0, 0, 0, -20, -40],
[-30, 0, 10, 15, 15, 10, 0, -30],
[-30, 5, 15, 20, 20, 15, 5, -30],
[-30, 0, 15, 20, 20, 15, 0, -30],
[-30, 5, 10, 15, 15, 10, 5, -30],
[-40, -20, 0, 5, 5, 0, -20, -40],
[-50, -40, -30, -30, -30, -40, -50]]

```

(repeat for all pieces)

I then wrote them to a json file for future use.



```

import json

with open("value_matrices.json", "w") as file:
    file.write(
        json.dumps(
            {
                'pawn': pawn,
                'knight': knight,
                'bishop': bishop,
                'castle': rook,
                'queen': queen,
                'king_start': king_start,
                'king_end': king_end
            },
        )
    )

```

Python

With this strategy for valuing a piece and my testes vector library I created my pieces library
pieces.py

```

# to do: make value and value matrix unchangeable of private
# https://stackoverflow.com/questions/31457855/cant-instantiate-abstract-
# class-with-abstract-methods

# import libraries and other local modules
import abc
from itertools import chain as iter_chain, product as iter_product
from typing import Callable

from vector import Vector
from assorted import ARBITRARILY_LARGE_VALUE

# Here is an abstract base class for piece,
# it dictates that all child object have the specified abstract attributes
# else and error will occur
# this ensures that all piece objects have the same interface

class Piece(abc.ABC):

    # required
    # color is public
    color: str | None
    # value and value matrix is protected
    # value is inherent value
    _value: int
    # value matrix is additional value based on location
    _value_matrix: tuple[tuple[float]]

    # must be given before init

```

```
@abc.abstractproperty
def _value_matrix(): pass

@abc.abstractproperty
def _value(): pass

@abc.abstractproperty
def color(): pass

@abc.abstractmethod
def symbol(self) -> str:
    """uses color to determine the appropriate symbol"""

# not needed as abstract method as some classes will now override
def __init__(self, color):
    self.color = color
    self.last_move = None

# this should use the position vector and value matrix to get the
value of the piece
def get_value(self, position_vector: Vector):
    # flip if black as matrices are all for white pieces
    if self.color == "W":
        row, column = 7-position_vector.j, position_vector.i
    else:
        row, column = position_vector.j, position_vector.i

    # return sum of inherent value + value relative to position on
board
    return self._value + self._value_matrix[row][column]

# this function should yield all the movement vectors that the piece can
move by
# this doesn't account for check and is based on rules specific to
each piece as well as checking if a vector is outside the board

@abc.abstractmethod
def generate_movement_vectors(self, pieces_matrix, position_vector):
    pass

# when str(piece) called give the symbol

def __str__(self):
    # return f"{self.color}{self.symbol}"
    return self.symbol()

# standard repr method
def __repr__(self):
```

```
        return f"{type(self).__name__}(color='{self.color}')"

# logic that would be otherwise repeated in many of the child classes
# determines the contents of a given square
def square_contains(self, square):
    """returns 'enemy' 'ally' or None for empty"""
    # check if empty
    if square is None:
        return "empty"
    # else the square must contain a piece, so examine its color
    if square.color == self.color:
        return "ally"
    else:
        return "enemy"

# again reduces repeated logic
# checks the result of a position vector
# if not illegal (out of board) the square contents is returned
def examine_position_vector(self, position_vector: Vector,
pieces_matrix):
    """returns 'enemy' 'ally' 'empty' or 'illegal' """
    # check if the vector is out of the board
    if not position_vector.in_board():
        return 'illegal'
    # for the rest of the code I can assume the vector is in board

    # else get the square at that vector
    row, column = 7-position_vector.j, position_vector.i
    square = pieces_matrix[row][column]

    # examine its contents
    return self.square_contains(square)

# equality operator
def __eq__(self, other):
    try:
        # assert same subclass like rook
        assert isinstance(other, type(self))
        assert self.color == other.color

        # i am not checking that pieces had the same last move as I
        # want to compare kings without for the check function
        # assert self.last_move == other.last_move

        # value and value_matrix should never be changes
    except AssertionError:
        return False
    else:
```

```
        return True

    # making pieces hashable allows for pieces matrices to be hashed and
    # allows for pieces and pieces matrices to be put in sets,
    # also essential for piping data between python interpreter instances
    # (different threads) for multitasking
    def __hash__(self):
        return hash((self.symbol(), self.color, self.last_move))

# this class inherits from Piece an so it inherits some logic and some
# requirements as to how its interface should be
# as many child classes are similar I will explain this one in depth and
# then only explain notable features of others
class Pawn(Piece):
    # defining abstract properties, needed before init
    _value = 100
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [50, 50, 50, 50, 50, 50, 50, 50],
        [10, 10, 20, 30, 30, 20, 10, 10],
        [5, 5, 10, 25, 25, 10, 5, 5],
        [0, 0, 0, 20, 20, 0, 0, 0],
        [5, -5, -10, 0, 0, -10, -5, 5],
        [5, 10, 10, -20, -20, 10, 10, 5],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]
    color = None

    # define symbol method (str method)
    def symbol(self): return f"{self.color}P"

    # override init constructor
    def __init__(self, color):
        # perform super's instructor
        super().__init__(color)

        # but in addition...
        # decide the vectors that the piece can move now as it is based on
color
        multiplier = 1 if color == "W" else -1

        # method defined here as it only used here
        # decided if the pawn is allowed to move foreward 2 based on
square contents and last move
        def can_move_foreword_2(square):
            return square is None and self.last_move is None

        # tuple contains pairs of vector and contition that must be met
```

```
# (in the form of a function that takes square and returns a
boolean)
    self.movement_vector_and_condition: tuple[Vector, Callable] = (
        # v(0, 1) for foreword
        (Vector(0, multiplier), lambda square:
self.square_contains(square) == "empty"),
        # v(0, 2) for foreword as first move
        (Vector(0, 2*multiplier), can_move_foreword_2),
        # v(-1, 1) and v(1, 1) for take
        (Vector(1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
        (Vector(-1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
    )

# generate movement vectors
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # iterate through movement vectors and conditions
    for movement_vector, condition in
self.movement_vector_and_condition:
        # get resultant vector
        resultant_vector = position_vector + movement_vector
        # if vector_out of range continue
        if not resultant_vector.in_board():
            continue

        # get the contents of the square corresponding to the
resultant
        row, column = 7-resultant_vector.j, resultant_vector.i
        piece = pieces_matrix[row][column]

        # if the condition is met, yield the vector
        if condition(piece):
            yield movement_vector


class Knight(Piece):
    _value = 320
    _value_matrix: tuple[tuple[float]] = [
        [-50, -40, -30, -30, -30, -40, -50],
        [-40, -20, 0, 0, 0, -20, -40],
        [-30, 0, 10, 15, 15, 10, 0, -30],
        [-30, 5, 15, 20, 20, 15, 5, -30],
        [-30, 0, 15, 20, 20, 15, 0, -30],
        [-30, 5, 10, 15, 15, 10, 5, -30],
        [-40, -20, 0, 5, 5, 0, -20, -40],
        [-50, -40, -30, -30, -30, -40, -50]
    ]
    color = None
```

```
# n for knight as king takes k
def symbol(self): return f"{self.color}N"

def generate_movement_vectors(self, pieces_matrix, position_vector):
    # this function yields all 8 possible vectors
    def possible_movement_vectors():
        vectors = (Vector(2, 1), Vector(1, 2))
        # for each x multiplier, y multiplier and vector combination
        for i_multiplier, j_multiplier, vector in iter_product((-1,
1), (-1, 1), vectors):
            # yield corresponding vector
            yield Vector(
                vector.i * i_multiplier,
                vector.j * j_multiplier
            )
    # iterate through movement vectors
    for movement_vector in possible_movement_vectors():
        # get resultant
        resultant_vector = position_vector + movement_vector
        # look at contents of square
        contents =
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix)
        # if square is empty yield vector
        if contents == "empty":
            yield movement_vector

class Bishop(Piece):
    _value = 330
    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -10, -10, -10, -10, -20],
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 10, 10, 5, 0, -10],
        [-10, 5, 5, 10, 10, 5, 5, -10],
        [-10, 0, 10, 10, 10, 10, 0, -10],
        [-10, 10, 10, 10, 10, 10, 10, -10],
        [-10, 5, 0, 0, 0, 0, 5, -10],
        [-20, -10, -10, -10, -10, -10, -10, -20]
    ]
    color = None

    def symbol(self): return f"{self.color}B"

    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # sourcery skip: use-itertools-product
        # repeat for all 4 vector directions
```

```
for i, j in iter_product((1, -1), (1, -1)):
    unit_vector = Vector(i, j)
    # iterate through length multipliers
    for multiplier in range(1, 8):
        # get movement and resultant vectors
        movement_vector = unit_vector * multiplier
        resultant_vector = position_vector + movement_vector

        # examine the contents of the square and use switch case
        to decide behaviour
        match
            self.examine_position_vector(position_vector=resultant_vector,
            pieces_matrix=pieces_matrix):
                case 'illegal':
                    # if vector extends out of the board stop
                    extending
                    break
                case 'ally':
                    # break of of for loop (not just match case)
                    # as cannot hop over piece so don't explore longer
                    vectors in same direction
                    break
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                    # break of of for loop (not just match case)
                    # as cannot hop over piece so don't explore longer
                    vectors in same direction
                    break
                case 'empty':
                    # is valid
                    yield movement_vector
                    # and keep exploring, don't break

class Rook(Piece):
    _value = 500
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [5, 10, 10, 10, 10, 10, 10, 5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [0, 0, 0, 5, 5, 0, 0, 0]
    ]
    color = None
```

```
# r for rook
def symbol(self): return f"{self.color}R"

# this code is very similar in structure to that of a bishop just with
different direction vectors
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # sourcery skip: use-itertools-product
    unit_vectors = (
        Vector(0, 1),
        Vector(0, -1),
        Vector(1, 0),
        Vector(-1, 0),
    )
    # for unit_vector, multiplier in iter_product(unit_vectors,
    range(1, 8)):
        for unit_vector in unit_vectors:
            for multiplier in range(1, 8):
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector

                # note cases that contain only break are not redundant,
                they break the outer for loop
                match
                    self.examine_position_vector(position_vector=resultant_vector,
                    pieces_matrix=pieces_matrix):
                        case 'illegal':
                            # if vector extends out of the board stop
                            extending
                                break
                        case 'ally':
                            # break of of for loop (not just match case)
                            # as cannot hop over piece so don't explore longer
                            vectors in same direction
                                break
                        case 'enemy':
                            # this is a valid move
                            yield movement_vector
                            # break of of for loop (not just match case)
                            # as cannot hop over piece so don't explore longer
                            vectors in same direction
                                break
                        case 'empty':
                            # is valid
                            yield movement_vector
                            # and keep exploring, don't break

class Queen(Piece):
    _value = 900
```

```

    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -5, -5, -10, -10, -20],
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 5, 5, 5, 0, -10],
        [-5, 0, 5, 5, 5, 5, 0, -5],
        [0, 0, 5, 5, 5, 5, 0, -5],
        [-10, 5, 5, 5, 5, 5, 0, -10],
        [-10, 0, 5, 0, 0, 0, 0, -10],
        [-20, -10, -10, -5, -5, -10, -10, -20]
    ]
    color = None

    def symbol(self): return f"{self.color}Q"

    # this code also uses a similar structure to the rook or bishop
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0,
        1), (-1, 0, 1)) if i != 0 and j != 0)
        unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1),
        (-1, 0, 1)) if i != 0 or j != 0)

        # for unit_vector, multiplier in iter_product(unit_vectors,
        range(1, 8)):
            for unit_vector in unit_vectors:
                for multiplier in range(1, 8):
                    movement_vector = unit_vector * multiplier
                    resultant_vector = position_vector + movement_vector
                    match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                        case 'illegal':
                            # if vector extends out of the board stop
extending
                            break
                        case 'ally':
                            # break of of for loop (not just match case)
                            # as cannot hop over piece so don't explore longer
vectors in same direction
                            break
                        case 'enemy':
                            # this is a valid move
                            yield movement_vector
                            # break of of for loop (not just match case)
                            # as cannot hop over piece so don't explore longer
vectors in same direction
                            break
                        case 'empty':
                            # is valid

```

```

        yield movement_vector
        # and keep exploring, don't break

class King(Piece):
    # not needed as static eval doesn't add up kings value
    _value = ARBITRARILY_LARGE_VALUE

    # there are 2 matrices to represent the early and late game for the
    king
    value_matrix_early: tuple[tuple[float]] = [
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-20, -30, -30, -40, -40, -30, -30, -20],
        [-10, -20, -20, -20, -20, -20, -20, -10],
        [20, 20, 0, 0, 0, 0, 20, 20],
        [20, 30, 10, 0, 0, 10, 30, 20]
    ]
    value_matrix_late = [
        [-50, -40, -30, -20, -20, -30, -40, -50],
        [-30, -20, -10, 0, 0, -10, -20, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -30, 0, 0, 0, 0, -30, -30],
        [-50, -30, -30, -30, -30, -30, -30, -50]
    ]
    color = None

    # initially value matrix is the early one
    _value_matrix: tuple[tuple[float]] = value_matrix_early

    # def __init__(self, *args, **kwargs):
    #     self._value_matrix = self.value_matrix_early
    #     super().__init__(self, *args, **kwargs)

    def symbol(self): return f"{self.color}K"

    # based on total pieces, changes the value matrix
    def update_value_matrix(self, pieces_matrix):
        # counts each empty square as 0 and each full one as 1 then sums
        them up to get total pieces
        # if total pieces less than or equal to 10: then late game
        if sum(int(isinstance(square, Piece)) for square in
iter_chain(pieces_matrix)) <= 10:
            self.value_matrix = self.value_matrix_late

```

```
# else early game
else:
    self.value_matrix = self.value_matrix_early

# this generates the movement vectors for the king
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # take the opportunity to update the value matrix
    self.update_value_matrix(pieces_matrix)

    # all 8 movement vectors
    unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1),
(-1, 0, 1)) if i != 0 or j != 0)

    # for each movement vector, get the resultant vector
    for movement_vector in unit_vectors:
        resultant_vector = position_vector + movement_vector

        # examine contents of square and use switch case to decide
        behaviour
        match
            self.examine_position_vector(position_vector=resultant_vector,
            pieces_matrix=pieces_matrix):
                case 'illegal':
                    continue
                case 'ally':
                    continue
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                case 'empty':
                    # is valid
                    yield movement_vector

# used by other modules to convert symbol to piece
PIECE_TYPES = {
    'P': Pawn,
    'N': Knight,
    'B': Bishop,
    'R': Rook,
    'K': King,
    'Q': Queen
}

# if __name__ == '__main__':
# ensures that all classes are valid (not missing any abstract properties)
# whenever the module is imported
Pawn('W')
Knight('W')
```

```
Bishop('W')
Rook('W')
Queen('W')
King('W')
```

I can remove one of the required abstract methods from the Rook class to show you the error this causes.

Change

```
# # r for rook
# def symbol(self): return f"{self.color}R"
```

Result

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python pieces.py
Traceback (most recent call last):
  File "C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\pieces.py", line 466, in <module>
    Rook('W')
TypeError: Can't instantiate abstract class Rook with abstract method symbol
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> █
```

This behaviour is valid as it means that, if I am sure that a variable contains an object of type Piece, not matter which piece, I can be sure that the interface will be the same (e.g. I can assume they all have a symbol method).

For the King class I gave it a value which is arbitrarily high. This comes from a file called **assorted.py**

```
# this file is just a file of short assorted constants and functions that
are general in use
# It only contains small functions as I have tried to group large, similar
functions logically in their own file

# this is used in the static evaluation and minimax process. It is used to
represent infinity in a way that still allows comparison
ARBITRARILY_LARGE_VALUE = 1_000_000

# this function is relatively redundant but allows for print statements in
debugging
# in later iteration this may be replaced with logging.
# it is useful as it allows for DEBUG print statements without needing to
remove them when finished
DEBUG = True
def dev_print(*args, **kwargs):
    if DEBUG:
```

```

print(*args, **kwargs)

# this is an exception that allows for the game data to be bound to it
# this allows for the relevant chess game that caused the error to be
examined afterwards
# it is a normal exception except the constructor has been modified to
save the game data as a property
class __ChessExceptionTemplate__(Exception):
    def __init__(self, *args, **kwargs) -> None:
        # none if key not present
        self.game = kwargs.pop("game", None)

    super().__init__(*args, **kwargs)

# these are custom exceptions.
# they contain no logic but have distinct types allowing for targeted
error handling
class InvalidMove(__ChessExceptionTemplate__):
    pass
class NotUserTurn(__ChessExceptionTemplate__):
    pass

```

The kings inherent value isn't relevant as both players will always have a king and so the value will always cancel out. I also used 2 value matrices for the king. I determined that the late game is when there are 10 or less total pieces though this is arbitrary.

The testing for the pieces module was as follows:

Test_pieces.py

```

import unittest
import ddt

import pieces
# as vector is already tested we can use it here and assume it won't cause
any logic errors
from vector import Vector

def test_dir(file_name): return f"test_data/pieces/{file_name}.yaml"

EMPTY_PIECES_MATRIX = ((None,) * 8,) * 8

@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test is based on testing the movement vectors of a piece
    places at some position in an empty board are as expected
    @ddt.file_data(test_dir('test_empty_board'))

```

```

def test_empty_board(self, piece_type, square, expected_move_squares):
    # deserialize
    position_vector = Vector.construct_from_square(square)
    piece: pieces.Piece = pieces.PIECE_TYPES[piece_type]('W')
    movement_vectors =
        piece.generate_movement_vectors(pieces_matrix=EMPTY_PIECES_MATRIX,
                                         position_vector=position_vector)
    resultant_squares = list(
        map(
            lambda movement_vector:
                (movement_vector+position_vector).to_square(),
            movement_vectors
        )
    )
    # assert as expected
    # can use sets to prevent order being an issue as vectors are
    # hashable
    self.assertEqual(
        set(resultant_squares),
        set(expected_move_squares),
        msg=f"\n\nactual movement squares
{sorted(resultant_squares)} != expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
    )

    # this test assert that that a pieces movement vectors are as expected
    # when the piece is surrounded by other pieces
    @ddt.file_data(test_dir('test_board_populated'))
    def test_board_populated(self, pieces_matrix, square,
                           expected_piece_symbol, expected_move_squares):
        # deserialize
        def list_map(function, iterable): return list(map(function,
        iterable))

        def descriptor_to_piece(descriptor) -> pieces.Piece:
            # converts WN to knight object with a color attribute of white
            if descriptor is None:
                return None

            color, symbol = descriptor
            piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
            return piece_type(color=color)

        def row_of_symbols_to_pieces(row): return
        list_map(descriptor_to_piece, row)

        # update pieces_matrix replacing piece descriptors to piece
        # objects

```

```

pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)

position_vector = Vector.construct_from_square(square)
row, column = 7-position_vector.j, position_vector.i

# assert piece as expected
piece: pieces.Piece = pieces_matrix[row][column]
self.assertEqual(
    piece.symbol(),
    expected_piece_symbol,
    msg=f"\nPiece at square {square} was not the expected piece"
)

# assert movement vectors as expected
movement_vectors =
piece.generate_movement_vectors(pieces_matrix=pieces_matrix,
position_vector=position_vector)
resultant_squares = list(
    map(
        lambda movement_vector:
(movement_vector+position_vector).to_square(),
        movement_vectors
    )
)
self.assertEqual(
    set(resultant_squares),
    set(expected_move_squares),
    msg=f"\n\nactual movement squares
{sorted(resultant_squares)} != expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
)

if __name__ == '__main__':
    unittest.main()

```

test_board_populated.yaml

```

test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]

```

```
square: B1
expected_piece_symbol: WP
expected_move_squares: [
    A2, C2, B2, B3
]
test2:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BQ, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D4
expected_piece_symbol: BQ
expected_move_squares: [
    D5,
    E4, F4, G4,
    E3, F2, G1,
    D3, D2, D1,
    C4,
    C5, B6, A7
]
test3:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D4
expected_piece_symbol: BR
expected_move_squares: [
    D5,
    E4, F4, G4,
    D3, D2, D1,
    C4,
]
test4:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
```

```
[null, null, null, BP, null, null, null, null],
[null, null, WP, null, WP, null, null, null],
[null, null, null, null, null, null, null, null]
]
square: D7
expected_piece_symbol: BP
expected_move_squares: [
    D6, D5,
    C6, E6
]
test5:
pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D8
expected_piece_symbol: BK
expected_move_squares: [
    C7, D7, E7,
    C8, E8
]
test6:
pieces_matrix: [
    [BR, BK, BB, BK, BQ, BB, BK, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WN, WB, WK, WQ, WB, WN, WR ]
]
square: B1
expected_piece_symbol: WN
expected_move_squares: [A3, C3]
test7:
pieces_matrix: [
    [BR, BN, BB, BK, BQ, BB, BN, BR ],
```

```
[BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP  ],
[null, null, null, null, null, null, null, null],
[WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP  ],
[WR,    WN,    WB,    WK,    WQ,    WB,    WN,    WR  ]
]
square: G1
expected_piece_symbol: WN
expected_move_squares: [F3, H3]
```

test_empty_board.yaml

```
test1:
  piece_type: 'N'
  square: E4
  expected_move_squares: [
    D2, F2, D6, F6, C5, C3, G5, G3
  ]
test2:
  piece_type: Q
  square: C3
  expected_move_squares: [
    A1, B2, D4, E5, F6, G7, H8,
    C1, C2, C4, C5, C6, C7, C8,
    A3, B3, D3, E3, F3, G3, H3,
    A5, B4, D2, E1
  ]
test3:
  piece_type: K
  square: B2
  expected_move_squares: [
    C1, C2, C3,
    B1, B3,
    A1, A2, A3,
  ]
test4:
  piece_type: P
  square: E3
  expected_move_squares: [
    E4, E5
  ]
test5:
  piece_type: R
  square: F6
  expected_move_squares: [
    F1, F2, F3, F4, F5, F7, F8,
```

```

    A6, B6, C6, D6, E6, G6, H6
]
test6:
piece_type: B
square: D7
expected_move_squares: [
    C8, E6, F5, G4, H3,
    E8, C6, B5, A4,
]
test7:
piece_type: K
square: D8
expected_move_squares: [
    C7, D7, E7,
    C8, E8
]

```

These tests are explained by the comments. They focus on testing the movement of the pieces (needed for the legal moves function).

This sub problem of deciding where a piece could move was made easier by abstracting away complications such as how check affects the legal moves. This additional logic will be added on by later modules that will rely on the pieces module. I wrote the tests as I coded the pieces module and so I was able to detect and fix issues as I developed the module until and the tests needed had been written and were working.

Evidence:

```

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_pieces
.....
Ran 14 tests in 0.004s
OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>

```

There was, in hind sight a flaw with my testing of this module which I will get to later (mirroring where I was in my development journey when I discovered it).

Next I created a board state class. This class was to be responsible for keeping track of a single board state: a snapshot in the whole game. It would include a utility function, a move executer function and a generate legal moves function. It would be immutable. This was because I knew that the minimax function would want to make many different combinations of moves on a given board state without irreversibly mutating the object or needing to make a deep copy.

Here is the class I came up with:

Board_state.py

```

from copy import deepcopy
from dataclasses import dataclass
from itertools import product as iter_product

```

```

import pieces as pieces_mod
from assorted import ARBITRARILY_LARGE_VALUE
from vector import Vector

# this is a pieces matrix for the starting position in chess
# white is at the bottom as it is from the user's perspective and I am
# currently assuming the user is white.
# I can change this in the frontend later
STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
    (
        pieces_mod.Rook(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Queen(color="B"),
        pieces_mod.King(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Rook(color="B")
    ),
    (pieces_mod.Pawn(color="B"),)*8,
    (None,)*8,
    (None,)*8,
    (None,)*8,
    (None,)*8,
    (pieces_mod.Pawn(color="W"),)*8,
    (
        pieces_mod.Rook(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Queen(color="W"),
        pieces_mod.King(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Rook(color="W")
    )
)
)

# frozen makes each instance of the board_state class immutable
@dataclass(frozen=True)
class Board_State():
    next_to_go: str = "W"
    # the pieces matrix keeps track of the board positions and the pieces
    pieces_matrix: tuple[tuple[pieces_mod.Piece]] = STARTING_POSITIONS

    # this function outputs the board in a user friendly way
    # 8| BR  BN  BB  BQ  BK  BB  BN  BR
    # 7|   .   BP  BP  BP  BP  BP  BP
    # 6|   .   .   .   .   .   .   .

```

```

# 5| BP  BP  .  .  .  .  .  .
# 4| .  .  .  WP  .  WP  .  .
# 3| .  .  .  .  .  .  .  .
# 2| WP  WP  WP  .  WP  .  WP  WP
# 1| WR  WN  WB  WQ  WK  WB  WN  WR
#   (A  B  C  D  E  F  G  H )

def print_board(self):
    # convert each piece to a symbol BP and replace none with dots
    # add numbers to left and add letters at the bottom
    numbers = range(8, 0, -1)
    letters = [chr(i) for i in range(ord("A"), ord("H")+1)]
    for row, number in zip(self.pieces_matrix, numbers):
        # clean up row of pieces
        symbols_row = map(lambda piece: piece.symbol() if piece else
". ", row)
        pretty_row = f"{number}| {' '.join(symbols_row)}"
        print(pretty_row)
    print(f" ({' '.join(letters)})")

def get_piece_at_vector(self, vector: Vector):
    # this function exists as it is a really common operation.
    # due to vector 0, 0 pointing the the bottom left not top left of
    # the 2d array,
    # some correction is needed
    column, row = vector.i, 7-vector.j
    return self.pieces_matrix[row][column]

# this function should yield all the pieces on the board
def generate_all_pieces(self):
    # nested loop for i and j to iterate through all possible vectors
    for i, j in iter_product(range(8), range(8)):
        position_vector = Vector(i,j)
        # get the contents of the corresponding square
        piece = self.get_piece_at_vector(position_vector)

        # skip if none: skip if square empty
        if piece:
            yield piece, position_vector

# this yields all pieces of a given color:
# used when examining legal moves of a given player
def generate_pieces_of_color(self, color=None):
    # be default give pieces of player next to go
    if color is None:
        color = self.next_to_go

    # return all piece and position vector pairs
    # filtered by piece must match in color

```

```

yield from filter(
    # lambda piece, _: piece.color == color,
    lambda piece_and_vector: piece_and_vector[0].color == color,
    self.generate_all_pieces()
)

# determines if a given player is in check based on the player's color
def color_in_check(self, color=None):
    # default is to check if the player next to go is in check
    if color is None:
        color = self.next_to_go

    # let us now be A's turn
    # I use this player a and b model to keep track of the logic here
    color_a = color
    color_b = "W" if color == "B" else "B"

    # we will examine all the movement vectors of B's pieces
    # if any of them could take the A's King then currently A is in
    # check as their king is threatened by 1 or more pieces (which could take it
    # next turn)
    # for each of b's pieces
    for piece, position_v in
self.generate_pieces_of_color(color=color_b):
        movement_vs = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=position_v
        )
        # for each movement vector that the piece could make
        for movement_v in movement_vs:
            resultant = position_v + movement_v
            # check the contents of the square
            to_square = self.get_piece_at_vector(resultant)
            # As_move_threatens_king_A = isinstance(to_square,
            pieces_mod.King) and to_square.color == color_a
                # if the contents is A's king then the a is in check.
                As_move_threatens_king_A = (to_square ==
            pieces_mod.King(color=color_a))
                    # if As_move_threatens_king_A then return true
                    if As_move_threatens_king_A:
                        return True
            # if none of B's pieces were threatening to take A's king then A
            isn't in check
            return False

    # this function is a generator to be iterated through.
    # it is responsible for yielding every possible move that a given
    player can make

```

```

# yields this as a position and a movement vector
def generate_legal_moves(self):
    # iterate through all pieces belonging to player next to go
    for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
        movement_vectors = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=piece_position_vector
        )
        # iterate through movement vectors of this piece
        for movement_vector in movement_vectors:
            # examine resulting child game state
            child_game_state =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
                # determine if current player next to go (different to
next to go of child game state) is in check
                is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
                    # only yield the move if it doesn't result in check
                    if not is_check_after_move:
                        yield piece_position_vector, movement_vector

# determines if the game is over
# returns over, winner
def is_game_over_for_next_to_go(self):
    # sourcery skip: remove-unnecessary-else, swap-if-else-branches

        # in all cases, the game is over if a player has no legal moves
left
    if not list(self.generate_legal_moves()):
        # if b in check
        if self.color_in_check():
            # checkmate for b, a wins
            winner = "W" if self.next_to_go == "B" else "B"
            return True, winner
        else:
            # stalemate
            return True, None
    # game not over
    return False, None

# this function is responsible for generating a static evaluation for
a given board-stat
# it should be used by a maximiser or minimiser
# starting position should have an evaluation of 0
def static_evaluation(self):
    # if over give an appropriate score for win loss or draw

```

```
over, winner = self.is_game_over_for_next_to_go()
if over:
    match winner:
        case None: multiplier = 0
        case "W": multiplier = 1
        case "B": multiplier = -1
    # return winner * ARBITRARILY_LARGE_VALUE
    return multiplier * ARBITRARILY_LARGE_VALUE

    # this function takes a piece as an argument and uses its color to
    decide if its value should be positive or negative
    def get_piece_value(piece: pieces_mod.Piece, position_vector:
Vector):
        # this function assumes white is maximizer and so white pieces
        have a positive score and black negative
        match piece.color:
            case "W": multiplier = 1
            case "B": multiplier = (-1)
        value = multiplier * piece.get_value(position_vector)
        # print(f"{piece.symbol()} at {position_vector.to_square} has
        value {value}")
        return value

        # for each piece, get the value (+/-)
values = map(
    lambda x: get_piece_value(*x),
    self.generate_all_pieces()
)
# sum values for static eval
return sum(values)

def make_move(self, from_position_vector: Vector, movement_vector:
Vector):
    resultant_vector = from_position_vector + movement_vector

    # make a copy of the position vector, deep copy is used to
    ensure no parts are shared by reference
    new_pieces_matrix = deepcopy(self.pieces_matrix)

    # convert to list
    new_pieces_matrix = list(map(list, new_pieces_matrix))

    # look at square with position vector
    row, col = 7-from_position_vector.j, from_position_vector.i
    # get piece that's moving
    piece: pieces_mod.Piece = new_pieces_matrix[row][col]
    # set from square to blank
    new_pieces_matrix[row][col] = None
```

```

# update piece to keep track of its last move
piece.last_move = movement_vector

# set to square to this piece
row, col = 7-resultant_vector.j, resultant_vector.i
new_pieces_matrix[row][col] = piece

# convert back to tuple
new_pieces_matrix = tuple(map(tuple, new_pieces_matrix))

# update next to go
new_next_to_go = "W" if self.next_to_go == "B" else "B"

# return new board state instance
return Board_State(
    next_to_go=new_next_to_go,
    pieces_matrix=new_pieces_matrix
)

```

I wrote the functions and tested them in the order of dependency.

Starting with:

- Get_piece_at_vecotor as it had not dependency to other functions

Then in order

- Generate_all_pieces
- Generate_pieces_of_color
- Color_in_check
- Generate_legal_moves
- Game_over
- Static_Evaluation

I had a significant issue in testing this module, specifically the generate legal moves function and the static evaluation function. I will show all the testing code and data and then explain the issue.

Test_board_state.py

```

import unittest
import ddt

from board_state import Board_State
# tested so assumed correct
import pieces
from vector import Vector

```

```

def test_dir(file_name): return f"test_data/board_state/{file_name}.yaml"

# code used to deserialize
# code repeated from test_pieces, opportunity to reduce redundancy
def list_map(function, iterable): return list(map(function, iterable))
def tuple_map(function, iterable): return tuple(map(function, iterable))

def descriptor_to_piece(descriptor) -> pieces.Piece:
    # converts WN to knight object with a color attribute of white
    if descriptor is None:
        return None

    color, symbol = descriptor
    piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
    return piece_type(color=color)

def deserialize_pieces_matrix(pieces_matrix, next_to_go="W") ->
Board_State:
    def row_of_symbols_to_pieces(row):
        return list_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)
    board_state: Board_State = Board_State(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)

    return board_state

@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test isn't data driven,
    # it tests that the static evaluation is 0 for starting positions
    def test_static_eval_starting_positions(self):
        self.assertEqual(
            Board_State().static_evaluation(),
            0
        )

        # this test is testing that a list of all pieces and there position
        # vectors can be generated
        @ddt.file_data(test_dir('generate_all_pieces'))
        def test_generate_all_pieces(self, pieces_matrix, pieces_and_squares):
            pieces_and_squares = tuple_map(tuple, pieces_and_squares)

            board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix)

```

```
# use set as order irrelevant
all_pieces: set[pieces.Piece, Vector] =
set(board_state.generate_all_pieces())

# convert square to vector, allowed as this is tested
def deserialize(data_unit):
    # unpack test data unit
    descriptor, square = data_unit
    # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
    return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

def serialize(data_unit):
    piece, vector = data_unit
    # return [piece.symbol(), vector.to_square()]
    return (piece.symbol(), vector.to_square())

all_pieces_expected: set[pieces.Piece, Vector] =
set(map(deserialize, pieces_and_squares))
# legal_moves_expected: list[pieces.Piece, Vector] = sorted(
#     map(deserialize, pieces_and_squares),
#     key=repr
# )

self.assertEqual(
    all_pieces,
    all_pieces_expected,
    msg=f"\nactual {list_map(serialize, all_pieces)} != expected
{list_map(serialize, all_pieces_expected)}"
)

# this test is to test the function responsible for getting the piece
at a given position vector
@ddt.file_data(test_dir('piece_at_vector'))
def test_piece_at_vector(self, pieces_matrix,
vectors_and_expected_piece):
    board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix)

    # could use all method and one assert but this would have been
less readable, also hard to make useful message,
    # wanting different messages implies multiple asserts should be
completed
    for vector, expected_piece in vectors_and_expected_piece:
        # deserialize vector / cast to Vector
        vector: Vector = Vector(*vector)
```

```
# repeat for piece
    expected_piece: pieces.Piece =
descriptor_to_piece(expected_piece)

    # actual
    piece: piece.Piece = board_state.get_piece_at_vector(vector)
    msg = f"Piece at vector {repr(vector)} is {repr(piece)} not
expected piece {repr(expected_piece)}"

    # this test ensures that all the pieces belonging to a specific color
and there position vectors can be identified
    @ddt.file_data(test_dir('generate_pieces_of_color'))
    def test_generate_pieces_of_color(self, pieces_matrix, color,
pieces_and_squares):
        pieces_and_squares = tuple_map(tuple, pieces_and_squares)

        board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix)
        # use set as order irrelevant
        legal_moves_actual: set[pieces.Piece, Vector] =
set(board_state.generate_pieces_of_color(color))

        # convert square to vector, allowed as this is tested
        def deserialize(data_unit):
            # unpack test data unit
            descriptor, square = data_unit
            # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
            return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

        def serialize(data_unit):
            piece, vector = data_unit
            # return [piece.symbol(), vector.to_square()]
            return (piece.symbol(), vector.to_square())

        legal_moves_expected: set[pieces.Piece, Vector] =
set(map(deserialize, pieces_and_squares))

        self.assertEqual(
            legal_moves_actual,
            legal_moves_expected,
            msg=f"\nactual {list_map(serialize,
legal_moves_actual)} != expected {list_map(serialize,
legal_moves_expected)}"
        )
```

```
# this test ensures that the chess engine can determine if a specified
player is currently in check
@ddt.file_data(test_dir('color_in_check'))
def test_color_in_check(self, pieces_matrix, white_in_check,
black_in_check):
    board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix)

    self.assertEqual(
        board_state.color_in_check("W"),
        white_in_check,
        msg=f"white {'should' if white_in_check else 'should not'} be
in check but {'is' if board_state.color_in_check('W') else 'is not'}"
    )
    self.assertEqual(
        board_state.color_in_check("B"),
        black_in_check,
        msg=f"black {'should' if black_in_check else 'should not'} be
in check but {'is' if board_state.color_in_check('B') else 'is not'}"
    )

# this test ensures that a game over situation can be identified and
its nature discerned
@ddt.file_data(test_dir("game_over"))
def test_game_over(self, pieces_matrix, expected_over,
expected_outcome, next_to_go):
    board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix, next_to_go=next_to_go)

    self.assertEqual(
        board_state.is_game_over_for_next_to_go(),
        (expected_over, expected_outcome),
        msg=f"\nactual {board_state.is_game_over_for_next_to_go()} !="
expected {(expected_over, expected_outcome)}"
    )

# this is one of the most important functions to test
# this function determines all the possible legal moves a player can
make with their pieces, accounting for check
# this test checks this for many inputs
@ddt.file_data(test_dir("generate_legal_moves"))
def test_generate_legal_moves(self, pieces_matrix, next_to_go,
expected_legal_moves):
    board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)
    actual_legal_moves = set(board_state.generate_legal_moves())
```

```

# convert square to vector, allowed as this is tested
def deserialize_expected_legal_moves():
    # unpack test data unit
    for test_datum in expected_legal_moves:
        from_square, all_to_squares = test_datum
        for to_square in all_to_squares:
            position_vector: Vector =
                Vector.construct_from_square(from_square)
            movement_vector: Vector =
                Vector.construct_from_square(to_square) - position_vector

            yield (position_vector, movement_vector)

expected_legal_moves = set(deserialize_expected_legal_moves())
self.assertEqual(
    actual_legal_moves,
    expected_legal_moves,
)

```

```

if __name__ == '__main__':
    unittest.main()

```

Color_in_check.yaml

```

test1:
  pieces_matrix: [
    [BR,    BK,    BB,    BK,    BQ,    BB,    BK,    BR  ],
    [BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP  ],
    [null, null, null, null, null, null, null, null],
    [WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP  ],
    [WR,    WK,    WB,    WK,    WQ,    WB,    WK,    WR  ]
  ]
  white_in_check: false
  black_in_check: false
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, WK, WQ, BK],
    [null, null, null, null, null, null, null, null],
  ]

```

```
[null, null, null, null, null, null, null, null],  
]  
white_in_check: false  
black_in_check: true  
  
test3:  
pieces_matrix: [  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null, BQ],  
    [null, null, null, null, null, null, null, null, null],  
    [null, null, null, WP, WP, null, null, null],  
    [null, null, null, WB, WK, WB, null, null],  
]  
white_in_check: true  
black_in_check: false  
  
test4:  
pieces_matrix: [  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, WK, null, BK],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, WR]  
]  
white_in_check: false  
black_in_check: true  
test5:  
pieces_matrix: [  
    [null, null, null, BK, null, null, null, null],  
    [null, null, null, WP, null, null, null, null],  
    [null, null, null, WK, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
]  
white_in_check: false  
black_in_check: false  
test6:  
pieces_matrix: [
```

```
[null, null, BK, null, null, null, null, null],
[null, null, null, WP, null, null, null, null],
[null, null, null, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true
test7:
pieces_matrix: [
[null, null, null, null, BK, null, null, null],
[null, null, null, WP, null, null, null, null],
[null, null, null, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true
test8:
pieces_matrix: [
[null, null, null, null, null, null, null, null],
[null, null, null, WP, BK, null, null, null],
[null, null, null, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
white_in_check: true
black_in_check: true
test9:
pieces_matrix: [
[null, null, null, null, null, null, null, null],
[null, null, BK, WP, null, null, null, null],
[null, null, null, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
```

```
white_in_check: true
black_in_check: true
```

Game_over.yaml

```
test1:
  next_to_go: W
  pieces_matrix: [
    [BR,    BK,    BB,    BK,    BQ,    BB,    BK,    BR  ],
    [BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP  ],
    [null, null, null, null, null, null, null, null],
    [WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP  ],
    [WR,    WK,    WB,    WK,    WQ,    WB,    WK,    WR  ]
  ]
  expected_over: false
  expected_outcome: null
test2:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK,    null, null, null, null],
    [null, null, null, WP,    null, null, null, null],
    [null, null, null, WK,    null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_over: true
  expected_outcome: null
test3:
  next_to_go: B
  pieces_matrix: [
    [WR,    null, null, null, null, null, BK,    null],
    [null, WR,    null, null, null, null, null, null],
    [null, null, null, null, null, null, null, WK  ],
  ]
```

```
]
expected_over: true
expected_outcome: W
test4:
next_to_go: B
pieces_matrix: [
[null, null, null, BK, null, null, null, null],
[null, null, null, WP, null, null, null, null],
[null, null, WP, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
expected_over: true
expected_outcome: null
test5:
next_to_go: B
pieces_matrix: [
[null, null, null, BK, null, null, null, null],
[null, null, WP, WP, null, null, null, null],
[null, null, null, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
expected_over: true
expected_outcome: W
test6:
next_to_go: W
pieces_matrix: [
[BR, null, null, null, null, BR, BK, null],
[BP, null, null, null, null, BP, BP, null],
[null, BP, null, null, null, null, null, null],
[null, null, BQ, null, null, null, null, null],
[null, null, null, null, null, null, null, WQ ],
[null, null, null, null, null, null, null, null],
[WP, null, null, null, null, WP, WB, null],
[WR, null, null, null, null, null, null, WR ],
]
expected_over: false
expected_outcome: null
test7:
next_to_go: B
pieces_matrix: [
```

```
[BR, null, null, null, null, BR, BK, WQ ],
[BP, null, null, null, null, BP, BP, null],
[null, BP, null, null, null, null, null, null],
[null, null, BQ, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, WP, null],
[WP, null, null, null, null, WP, WB, null],
[WR, null, null, null, null, null, null, WR ],
]
expected_over: true
expected_outcome: W
```

generate_all_pieces.yaml

```
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]
  pieces_and_squares: [
    [WP, B1],
    [BP, A2],
    [BP, C2]
  ]
]

test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  pieces_and_squares: [
    [WR, B2],
    [BR, B4],
    [BP, C3],
    [BR, D4],
    [BP, D6],
  ]
```

```
[BP, G3],
[WB, G4],
[BR, E5]
]
```

Generate_legal_moves.yaml

```
test1:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_legal_moves: []
test2:
  next_to_go: W
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_legal_moves: [
    [B1, [B2, B3, A2, C2]]
  ]
test3:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
```

```
expected_legal_moves: [
    [A2, [A1, B1]],
    [C2, [C1, B1]],
]
test4:
next_to_go: W
pieces_matrix: [
    [BR, BN, BB, BK, BQ, BB, BN, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WN, WB, WK, WQ, WB, WN, WR ]
]
expected_legal_moves: [
    [A2, [A3, A4]],
    [B2, [B3, B4]],
    [C2, [C3, C4]],
    [D2, [D3, D4]],
    [E2, [E3, E4]],
    [F2, [F3, F4]],
    [G2, [G3, G4]],
    [H2, [H3, H4]],
    [B1, [A3, C3]],
    [G1, [F3, H3]]
]
test5:
next_to_go: W
pieces_matrix: [
    [BK, BP, null, null, null, null, WP, WK],
    [BP, BP, null, null, null, null, WP, WP],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WN, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
expected_legal_moves: [
    [B2, [A4, C4, D1, D3]],
    [G4, [
        H3, F5, E6, D7, C8,
        H5, F3, E2, D1
    ]],
]
```

Generate_pieces_of_color.yaml

```
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]
  color: W
  pieces_and_squares: [
    [WP, B1]
  ]
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]
  color: B
  pieces_and_squares: [
    [BP, A2],
    [BP, C2]
  ]
test3:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  color: W
  pieces_and_squares: [
```

```

        [WR, B2],
        [WB, G4]
    ]
test4:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
color: B
pieces_and_squares: [
    [BR, B4],
    [BP, C3],
    [BR, D4],
    [BP, D6],
    [BP, G3],
    [BR, E5]
]

```

Piece_at_vector.yaml

```

test1:
pieces_matrix: [
    [BR, BN, BB, BK, BQ, BB, BN, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WN, WB, WK, WQ, WB, WN, WR ]
]
vectors_and_expected_piece: [
    [[0, 0], WR],
    [[1, 1], WP],
    [[2, 4], null],
    [[7, 3], null],
    [[5, 7], BB],
    [[2, 6], BP]
]

test2:

```

```
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
vectors_and_expected_piece: [
    [[0, 0], null],
    [[1, 1], WR],
    [[1, 3], BR],
    [[0, 7], null],
    [[7, 0], null],
    [[7, 7], null],
    [[3, 3], BR],
    [[3, 5], BP],
    [[6, 3], WB],
    [[6, 1], null]
]
```

The tests now work fine:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_board_state
-----
Ran 30 tests in 0.277s
OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> █
```

But I was extremely stumped as I was testing my legal moves generator function when I got this issue:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_board_state
F....
=====
FAIL: test_generate_legal_moves_1_test1 (test_board_state.Test_Case)
test_generate_legal_moves_1_test1
-----
Traceback (most recent call last):
  File "C:\Users\henry\AppData\Local\Programs\Python\Python310\lib\site-packages\ddt.py", line 220, in wrapper
    return func(self, *args, **kwargs)
  File "C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\test_board_state.py", line 177, in test_generate_legal_moves
    self.assertEqual(
AssertionError: Items in the first set but not the second:
(Vector(i=3, j=7), Vector(i=-1, j=0))
(Vector(i=3, j=7), Vector(i=1, j=-1))
(Vector(i=3, j=7), Vector(i=0, j=-1))
(Vector(i=3, j=7), Vector(i=-1, j=-1))
(Vector(i=3, j=7), Vector(i=-1, j=0))

-----
Ran 5 tests in 0.048s
FAILED (failures=1)
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

(other tests disabled to show only this issue)

The test that had failed was (**generate_legal_moves.yaml**)

```
test1:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  expected_legal_moves: []
```

There are supposed to be no legal moves as black in in checkmate. Instead the restriction of check was ignored and my function said that the black king could move to all 5 adjacent squares. I added this test to the check tests:

(color_in_check.yaml)

```
test5:
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null]
```

```
[null, null, null, null, null, null, null, null, null],  
]  
white_in_check: false  
black_in_check: false  
test6:  
pieces_matrix: [  
[null, null, BK, null, null, null, null, null, null],  
[null, null, null, WP, null, null, null, null, null],  
[null, null, null, WK, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
]  
white_in_check: false  
black_in_check: true  
test7:  
pieces_matrix: [  
[null, null, null, null, BK, null, null, null, null],  
[null, null, null, WP, null, null, null, null, null],  
[null, null, null, WK, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
]  
white_in_check: false  
black_in_check: true  
test8:  
pieces_matrix: [  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, WP, BK, null, null, null, null],  
[null, null, null, WK, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
[null, null, null, null, null, null, null, null, null],  
]  
white_in_check: true  
black_in_check: true  
test9:  
pieces_matrix: [  
[null, null, null, null, null, null, null, null, null],  
[null, null, BK, WP, null, null, null, null, null],  
[null, null, null, WK, null, null, null, null, null],
```

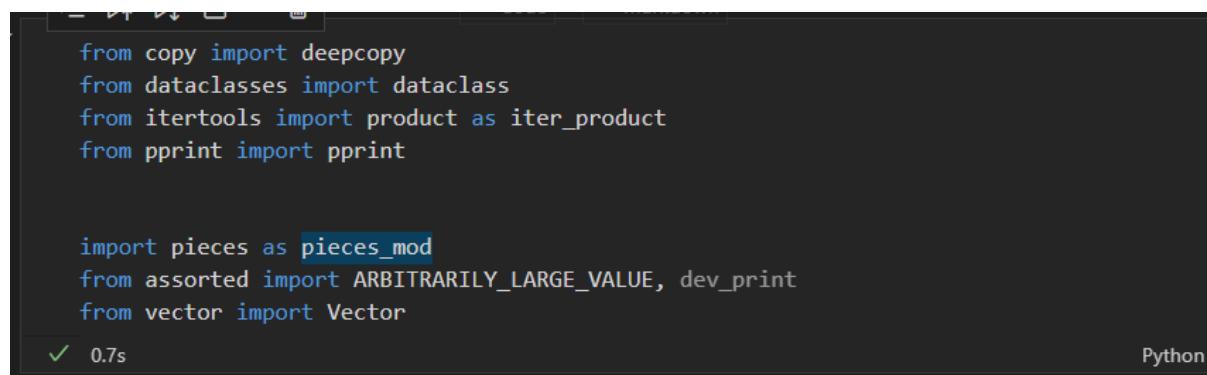
```
[null, null, null, null, null, null, null, null],
]
white_in_check: true
black_in_check: true
```

All these tests passed which really confused me. The check function correctly determined that the child game states resulting from each of these moves were check for black.

To further analyse this I created a jupyter notebook. I tried to replicate the exact function call being made by the test with a version of the legal moves generator that included a lot of print statements to reveal the inner workings.

I was able to recreate the contradictory outcomes

Cell 1:



```
from copy import deepcopy
from dataclasses import dataclass
from itertools import product as iter_product
from pprint import pprint

import pieces as pieces_mod
from assorted import ARBITRARILY_LARGE_VALUE, dev_print
from vector import Vector
```

✓ 0.7s Python

Cell 2 (copied as too big to screenshot)

```
STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
(
    pieces_mod.Rook(color="B"),
    pieces_mod.Knight(color="B"),
    pieces_mod.Bishop(color="B"),
    pieces_mod.Queen(color="B"),
    pieces_mod.King(color="B"),
    pieces_mod.Bishop(color="B"),
    pieces_mod.Knight(color="B"),
    pieces_mod.Rook(color="B")
),
(pieces_mod.Pawn(color="B"),)*8,
(None,)*8,
(None,)*8,
(None,)*8,
(None,)*8,
```

```
(pieces_mod.Pawn(color="W")),)*8,
(
    pieces_mod.Rook(color="W"),
    pieces_mod.Knight(color="W"),
    pieces_mod.Bishop(color="W"),
    pieces_mod.Queen(color="W"),
    pieces_mod.King(color="W"),
    pieces_mod.Bishop(color="W"),
    pieces_mod.Knight(color="W"),
    pieces_mod.Rook(color="W")
)
)

@dataclass(frozen=True)
class Board_State():
    pieces_matrix: tuple[tuple[pieces_mod.Piece]]
    next_to_go: int = "W"
    # pieces_matrix: tuple[tuple[pieces_mod.Piece]] = STARTING_POSITIONS

    def print_board(self, print_function=pprint):
        print_function(
            list(map(
                lambda row: list(map(
                    lambda piece: None if piece is None else
piece.symbol(),
                    row
                )),
                self.pieces_matrix
            )))
        )

    def get_piece_at_vector(self, vector: Vector):
        column, row = vector.i, 7-vector.j
        # column, row = vector.i, vector.j
        return self.pieces_matrix[row][column]

    def generate_all_pieces(self):
        # for i, j in zip(range(8), range(8)):
        # should be product
        for i, j in iter_product(range(8), range(8)):
            position_vector = Vector(i,j)
            piece = self.get_piece_at_vector(position_vector)
            # skip if none
            if piece:
                yield piece, position_vector
            # if piece:
            #     dev_print(f"not skipping {position_vector}")
            #     yield piece, position_vector
```

```
# else:
#     dev_print(f"skipping {position_vector}")

def generate_pieces_of_color(self, color=None):
    if color is None:
        color = self.next_to_go

    def same_color(data_item):
        piece, _ = data_item
        return piece.color == color

    yield from filter(
        same_color,
        self.generate_all_pieces()
    )

def color_in_check(self, color=None):
    if color is None:
        color = self.next_to_go

    # it is now A's turn
    color_a = color
    color_b = "W" if color == "B" else "B"

    # we will examine all the movement vectors of B's pieces
    # if any of them could take the A's King then currently A is in
    check as their king is threatened by 1 or more pieces (which could take it
    next turn)
    for piece, position_v in
self.generate_pieces_of_color(color=color_b):
        movement_vs = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=position_v
        )
        for movement_v in movement_vs:
            resultant = position_v + movement_v
            to_square = self.get_piece_at_vector(resultant)
            # print(f"to_square --> {to_square!r}")
            # dev_print(f"piece at square {resultant.to_square()} is
{to_square.symbol() if to_square else '<empty>'}")
            # dev_print(f"piece at square {resultant.to_square()} is
{repr(to_square) if to_square else '<empty>'}")
            # As_move_threatens_king_A = isinstance(to_square,
pieces_mod.King) and to_square.color == color_a
            As_move_threatens_king_A = (to_square ==
pieces_mod.King(color=color_a))
```

```

        # print(f"to_square == pieces_mod.King(color='B') --> {to_square!r} == {pieces_mod.King(color='B')!r} --> {to_square == pieces_mod.King(color='B')}")
        # dev_print(f"therefore king {'IS' if As_move_threatens_king_A else 'NOT'} threatened as square {'DOES' if As_move_threatens_king_A else 'DOES NOT'} contain {pieces_mod.King(color=color_a)!r} instead containing {repr(to_square) if to_square else '<empty>'}")
        # if As_move_threatens_king_A break out of all 3 loops
        if As_move_threatens_king_A:
            # dev_print(f"color_in_check(color='{color}')")
        returning True")
            # dev_print(f"piece {piece.symbol()} at {position_v.to_square()} moving to {resultant.to_square()} IS threatening king")
            return True
        # else:
            # dev_print(f"piece {piece.symbol()} at {position_v.to_square()} moving to {resultant.to_square()} NOT threatening king")
        # dev_print(f"color_in_check(color='{color}') returning False")
        return False

def generate_legal_moves(self):
    # dev_print(f"analysing legal moves for next to go {self.next_to_go}")
    for piece, piece_position_vector in self.generate_pieces_of_color(color=self.next_to_go):
        # dev_print(f"analysing moves for piece: {piece.symbol()}")
        movement_vectors = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=piece_position_vector
        )
        for movement_vector in movement_vectors:
            # dev_print(f"\tpiece {piece.symbol()}:_analysing movement vector {repr(movement_vector)}")
            child_game_state: Board_State =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
            # dev_print(f"\t\tmovement vector {repr(movement_vector)}: results in child game state")
            # child_game_state.print_board(
            #     print_function=lambda rows: list(map(
            #         lambda item: print(f"\t\t{item}"),
            #         rows
            #     )))
            # )

```

```

        # dev_print(repr(child_game_state))
        is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
        dev_print(f"\t\tThis game state in check? for
{self.next_to_go}: {is_check_after_move}")

        if not is_check_after_move:
            yield piece_position_vector, movement_vector

# def generate_legal_moves(self):
#     for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
    movement_vectors = piece.generate_movement_vectors(
#         pieces_matrix=self.pieces_matrix,
#         position_vector=piece_position_vector
#     )
    for movement_vector in movement_vectors:
#         child_game_state =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
        is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
        if not is_check_after_move:
#             yield piece_position_vector, movement_vector

def is_game_over_for_next_to_go(self):
    # check if in checkmate
    # for player a
    # if b has no moves
    if not list(self.generate_legal_moves()):
        # if b in check
        if self.color_in_check():
            # checkmate for b, a wins
            return True, self.next_to_go
        else:
            # stalemate
            return True, None

    return False, None

def static_evaluation(self):
    """Give positive static evaluation if white is winning"""
    def generate_all_pieces():
        for i, j in iter_product(range(8), range(8)):
            piece_position_vector = Vector(i, j)
            piece: pieces_mod.Piece =
self.get_piece_at_vector(piece_position_vector)
            if not piece:

```

```
        continue

        yield piece, piece_position_vector

over, winner = self.is_game_over_for_next_to_go()
if over:
    match winner:
        case None: multiplier = 0
        case "W": multiplier = 1
        case "B": multiplier = -1
    return winner * ARBITRARILY_LARGE_VALUE

else:
    return sum(piece.get_value(position_vector) * multiplier for
piece, position_vector in generate_all_pieces())

def make_move(self, from_position_vector: Vector, movement_vector:
Vector):
    to_position_vector = from_position_vector + movement_vector
    # poor code, this below line can cause infinite recursion when
legal_moves generator called post check changes
    # assert (from_position_vector, movement_vector) in
self.generate_legal_moves()

    new_pieces_matrix = deepcopy(self.pieces_matrix)
    # convert to list
    new_pieces_matrix = list(map(list, new_pieces_matrix))

    # set from to blank
    row, col = 7-from_position_vector.j, from_position_vector.i
    piece: pieces_mod.Piece = new_pieces_matrix[row][col]

    piece.last_move = movement_vector

    new_pieces_matrix[row][col] = None

    # set to square to this piece
    row, col = 7-to_position_vector.j, to_position_vector.i
    new_pieces_matrix[row][col] = piece

    # convert back to tuple
    new_pieces_matrix = tuple(map(tuple, new_pieces_matrix))
    new_next_to_go = "W" if self.next_to_go == "B" else "B"

    return Board_State(
        next_to_go=new_next_to_go,
        pieces_matrix=new_pieces_matrix
    )
```

Cell 3:

```
# tested so assumed correct
import pieces
from vector import Vector

def test_dir(file_name): return f"test_data/board_state/{file_name}.yaml"

# code repeated from test_pieces, opportunity to reduce redundancy

def list_map(function, iterable): return list(map(function, iterable))
def tuple_map(function, iterable): return tuple(map(function, iterable))

def descriptor_to_piece(descriptor) -> pieces.Piece:
    # converts WN to knight object with a color attribute of white
    if descriptor is None:
        return None

    color, symbol = descriptor
    piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
    return piece_type(color=color)

def deserialize_pieces_matrix(pieces_matrix, next_to_go="W") ->
Board_State:
    def row_of_symbols_to_pieces(row):
        return list_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)
    board_state: Board_State = Board_State(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)

    return board_state
```

cell 4 though 6

```
next_to_go = "B"
pieces_matrix = [
    [None, None, None, "BK", None, None, None, None],
    [None, None, None, "WP", None, None, None, None],
    [None, None, None, "WK", None, None, None, None],
    [None, None, None, None, None, None, None, None]
]
expected_legal_moves = []

[4] ✓ 0.6s Python

board_state: Board_State = deserialize_pieces_matrix(pieces_matrix=pieces_matrix, next_to_go=next_to_go)

[5] ✓ 0.4s Python

board_state.print_board()

[6] ✓ 0.6s Python
...
[[None, None, None, 'BK', None, None, None, None],
 [None, None, None, 'WP', None, None, None, None],
 [None, None, None, 'WK', None, None, None, None],
 [None, None, None, None, None, None, None, None]]
```

cell 7-8

```
    ▶ v set(board_state.generate_legal_moves())
[7]   ✓ 0.6s
...
...      This game state in check? for B: False
      This game state in check? for B: False

{(Vector(i=3, j=7), Vector(i=-1, j=-1)),
 (Vector(i=3, j=7), Vector(i=-1, j=0)),
 (Vector(i=3, j=7), Vector(i=0, j=-1)),
 (Vector(i=3, j=7), Vector(i=1, j=-1)),
 (Vector(i=3, j=7), Vector(i=1, j=0))}

board_state = Board_State(
    next_to_go="W",
    pieces_matrix= [
        [None, None, pieces.King(color="B"), None, None, None, None, None],
        [None, None, None, pieces.Pawn(color="W"), None, None, None, None],
        [None, None, None, pieces.King(color="W"), None, None, None, None],
        [None, None, None, None, None, None, None, None]
    ]
)
[8]   ✓ 0.7s
```

Cell 9-11

```
    ▶ v board_state.color_in_check("W")
[9]   ✓ 0.5s
...
...  False

    board_state.color_in_check("B")
[10]  ✓ 0.6s
...
...  True

    pieces.King("B") == pieces.King("B")
[11]  ✓ 0.5s
...
...  True
```

This highlights the issue and apparent contradiction in logic. My chess program is checking each child game state and reporting that black isn't in check, yielding all 5 moves. Then for one of the child game states the color in check functions seen to directly contradict this by correctly identifying that black is in check.

I later realised the issue and why all my unit tests for the check function had worked and yet a practical test via the legal moves generator function failed.

The line that was the issue what part of the Piece.__eq__ method

```

105 # equality operator
106 def __eq__(self, other):
107     try:
108         # assert same subclass like rook
109         assert isinstance(other, type(self))
110         assert self.color == other.color
111
112         # i am not checking that pieces had the same last move as I want to compare kings without for the check function
113         # assert self.last_move == other.last_moves
114
115         # value and value_matrix should never be changes
116     except AssertionError:
117         return False
118     else:
119         return True
120

```

Line 113 was the issue. Shown commented out.

This line meant that a king that had been moved was not equal to a king that had just been initialised for the purpose of a test. This is because they would have different last moves attributes. The comparison in the check function is to a king of the same color but with last_moves set to None. This worked for the unit tests as the kings in the tests also had not moved. Once this line was removed, all the tests passed.

This was an issue with the pieces testing ad my assumption that the module was robust, but it was fixed.

Next I moved onto a module that would contain a game class that was able to keep track of the whole game.

It should enable user moves to be implemented and computer moves to be generated and implemented (I will get to minimax).

Here was my code:

Game.py

```

# import other modules
from board_state import Board_State
from minimax import minimax
from vector import Vector
from assorted import ARBITRARILY_LARGE_VALUE, dev_print, NotUserTurn,
InvalidMove

# the game class is used to keep track of a chess game between a user and
the computer
class Game(object):
    # it keeps track of:
    # the player's color's

```

```

player_color_key: dict
# the difficulty or depth of the game
depth: int
# the current board state
board_state: Board_State
# the number of moves so far
move_counter: int
# a table of game state hashes and there frequency
piece_matrix_occurrence_hash_table: list

# this adds a new game state to the frequency table
def add_new_piece_matrix_to_hash_table(self, piece_matrix):
    # the pieces matrix is hashed
    matrix_hash = hash(piece_matrix)

        # if this pieces matrix has been encountered before, add 1 to the
frequency
        if matrix_hash in self.piece_matrix_occurrence_hash_table.keys():
            self.piece_matrix_occurrence_hash_table[matrix_hash] += 1
        # else set frequency to 1
        else:
            self.piece_matrix_occurrence_hash_table[matrix_hash] = 1

# determines if there is a 3 repeat stalemate
def is_3_board_repeats_in_game_history(self):
    # if any of the board states have been repeated 3 or more times:
stalemate
    return any(value >= 3 for value in
self.piece_matrix_occurrence_hash_table.values())

# constructor for game object
def __init__(self, depth=2, user_color="W") -> None:
    # based on user's color, determine color key
    self.player_color_key = {
        "W": 1 if user_color=="W" else -1,
        "B": -1 if user_color=="W" else 1
    }
    # set depth property from parameters
    self.depth = depth

    # set attributes for game at start
    self.board_state = Board_State()
    self.move_counter = 0
    self.piece_matrix_occurrence_hash_table = {}

# this function validates if the user's move is allowed and if so,
makes it
def implement_user_move(self, from_square, to_square) -> None:

```

```
# check that the user is allowed to move
which_player_next_to_go = self.player_color_key.get(
    self.board_state.next_to_go
)
if which_player_next_to_go != 1:
    raise NotUserTurn(game=self)

# unpack move into vector form
# invalid square syntaxes will cause a value error here
try:
    position_vector = Vector.construct_from_square(from_square)
    movement_vector = Vector.construct_from_square(to_square) -
position_vector
except Exception:
    raise ValueError("Square's not in valid format")

# if the move is not in the set of legal moves, raise and
appropriate exception
if (position_vector, movement_vector) not in
self.board_state.generate_legal_moves():
    raise InvalidMove(game=self)

# implement move
self.board_state =
self.board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)
# adjust other properties that keep track of the game
self.move_counter += 1
self.add_new_piece_matrix_to_hash_table(self.board_state.pieces_ma
trix)

return (position_vector, movement_vector),
self.board_state.static_evaluation()

# this function determines if the game is over and if so, what is the
nature of the outcome
def check_game_over(self):
    # returns: over: bool, winning_player: (1/-1), classification: str
    # check for 3 repeat stalemate
    if self.is_3_board_repeats_in_game_history():
        return True, None, "Stalemate"

    # determine if board state is over for next player
    over, winner = self.board_state.is_game_over_for_next_to_go()
    # switch case statement to determine the appropriate values to be
returned in each case
    match (over, winner):
        case False, _:
```

```

        victory_classification = None
        winning_player = None
    case True, None:
        victory_classification = "Stalemate"
        winning_player = None
    case True, winner:
        victory_classification = "Checkmate"
        winning_player = self.player_color_key[winner]
# return appropriate values
return over, winning_player, victory_classification

# this function determines and implements the computer move
def implement_computer_move(self, best_move_function=None):
    # for use with testing bots, a best move function can be provided
for use, but minimax if the default

    # get next to go player (1/-1)
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    # check that is it the computer's turn
    if which_player_next_to_go != -1:
        raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

    # if no function provided, default to minimax
    if best_move_function is None:
        score, best_child, best_move = minimax(
            board_state = self.board_state,
            # assume white / user always maximizer
            # is_maximizer = (self.board_state.next_to_go == "W"),
            is_maximizer = False,
            # depth is based of difficulty of game based on depth
parameter
            depth = self.depth,
            # default values for alpha and beta
            alpha = (-1)*ARBITRARILY_LARGE_VALUE,
            beta = ARBITRARILY_LARGE_VALUE
        )
    else:
        # otherwise use provided function,
        # the provided function should take game as an argument and
then return data in the same format as the minimax function
        score, best_child, best_move = best_move_function(self)

    # adjust properties that keep track of the game state
    self.board_state = best_child
    self.move_counter += 1

```

```
        self.add_new_piece_matrix_to_hash_table(self.board_state.pieces_matrix)

        # incase is it wanted for a print out ect, return move and score
        return best_move, score
```

in tandem with this I created a minimax function which I will go into more detail about shortly.

Rather than make a test for this I created a rudementry console chess game to play chess using this library.

```
from game import Game
from assorted import InvalidMove


# create new chess game
# difficulty set to depth 2
game = Game(depth=2)

# this function informs the user of the details when the game is over
def handle_game_over(winner, classification):
    print(f"The game is over, the {'user' if winner==1 else 'computer'} has won in a {classification}")

# print out the starting board
game.board_state.print_board()
print()

# keep game going until loop broken
while True:
    # user goes first
    print("Your go USER:")

    # while loop and error checking used to ensure move input
    while True:
        try:
            print("Please enter move in 2 parts")
            from_square = input("From square: ")
            to_square = input("To square: ")
            # check
            game.implement_user_move(from_square, to_square)
        except InvalidMove:
            print("This isn't a legal move, try again")
        except ValueError:
            print("This isn't valid input, try again")
        else:
            # if it worked break out of the loop
            break
```

```
# if move results in check then output this
if game.board_state.color_in_check():
    print("CHECK!")
# print out the current board_state
game.board_state.print_board()
print()

# check if game over after user's move
over, winner, classification = game.check_game_over()
# if over, handle it.
if over:
    handle_game_over(winner=winner, classification=classification)
    break

# alternate, it is now the computers go
print("Computer's go: ")

# get the computers move
move, _ = game.implement_computer_move()

# print out the board again
game.board_state.print_board()

# print out the computer's move in terms of squares
position_vector, movement_vector = move
resultant_vector = position_vector + movement_vector
piece_symbol =
game.board_state.get_piece_at_vector(resultant_vector).symbol()
print(f"Computer Moved {piece_symbol}: {position_vector.to_square()}"
      to {resultant_vector.to_square()}")

# print check if applicable
if game.board_state.color_in_check():
    print("CHECK!")

# check if the game is over, if so handle it
over, winner, classification = game.check_game_over()
if over:
    handle_game_over(winner=winner, classification=classification)
    break
# create a new line to separate for the user's next move
print()
```

This allowed the user to play check against the computer. The game wasn't configurable as the user was white against depth 2 minimax but it was playable.

Here is an example of it running:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python
console_chess.py
```

```
8| BR BN BB BQ BK BB BN BR
7| BP BP BP BP BP BP BP BP
6| . . . . . . . .
5| . . . . . . . .
4| . . . . . . . .
3| . . . . . . . .
2| WP WP WP WP WP WP WP WP
1| WR WN WB WQ WK WB WN WR
(A B C D E F G H )
```

Your go USER:

Please enter move in 2 parts

From square: D2

To square: D4

```
8| BR BN BB BQ BK BB BN BR
7| BP BP BP BP BP BP BP BP
6| . . . . . . . .
5| . . . . . . . .
4| . . . WP . . . .
3| . . . . . . . .
2| WP WP WP . WP WP WP WP
1| WR WN WB WQ WK WB WN WR
(A B C D E F G H )
```

Computer's go:

```
8| BR . BB BQ BK BB BN BR
7| BP BP BP BP BP BP BP BP
6| . . BN . . . . .
5| . . . . . . . .
4| . . . WP . . . .
3| . . . . . . . .
2| WP WP WP . WP WP WP WP
1| WR WN WB WQ WK WB WN WR
(A B C D E F G H )
```

Computer Moved BN: B8 to C6

Your go USER:

Please enter move in 2 parts

From square: rubbish that is invalid input

To square: more rubbish

This isn't valid input, try again

Please enter move in 2 parts

From square: a0

To square: i9

This isn't a legal move, try again

Please enter move in 2 parts

From square: e1
To square: e2
This isn't a legal move, try again
Please enter move in 2 parts

From square: E2
To square: E3
8| BR . BB BQ BK BB BN BR
7| BP BP BP BP BP BP BP BP
6| . . BN
5|
4| . . . WP
3| WP
2| WP WP WP . . WP WP WP
1| WR WN WB WQ WK WB WN WR
(A B C D E F G H)

Computer's go:

8| BR . BB BQ BK BB . BR
7| BP BP BP BP BP BP BP BP
6| . . BN . . BN . .
5|
4| . . . WP
3| WP
2| WP WP WP . . WP WP WP
1| WR WN WB WQ WK WB WN WR
(A B C D E F G H)

Computer Moved BN: G8 to F6

Your go USER:

Please enter move in 2 parts

From square: F2

To square: F3
8| BR . BB BQ BK BB . BR
7| BP BP BP BP BP BP BP BP
6| . . BN . . BN . .
5|
4| . . . WP
3| WP WP . .
2| WP WP WP . . WP WP
1| WR WN WB WQ WK WB WN WR
(A B C D E F G H)

Computer's go:

8| BR . BB BQ BK BB . BR
7| BP BP BP . BP BP BP BP
6| . . BN . . BN . .
5| . . . BP
4| . . . WP

3	WP	WP	.	.
2	WP	WP	WP	.	.	.	WP	WP
1	WR	WN	WB	WQ	WK	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer Moved BP: D7 to D5

Your go USER:

Please enter move in 2 parts

From square: C2

To square: C3

8	BR	.	BB	BQ	BK	BB	.	BR
7	BP	BP	BP	.	BP	BP	BP	BP
6	.	.	BN	.	.	BN	.	.
5	.	.	.	BP
4	.	.	.	WP
3	.	.	WP	.	WP	WP	.	.
2	WP	WP	WP	WP
1	WR	WN	WB	WQ	WK	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer's go:

8	BR	.	.	BQ	BK	BB	.	BR
7	BP	BP	BP	.	BP	BP	BP	BP
6	.	.	BN	.	BB	BN	.	.
5	.	.	.	BP
4	.	.	.	WP
3	.	.	WP	.	WP	WP	.	.
2	WP	WP	WP	WP
1	WR	WN	WB	WQ	WK	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer Moved BB: C8 to E6

Your go USER:

Please enter move in 2 parts

From square: D1

To square: D4

This isn't a legal move, try again

Please enter move in 2 parts

From square: D1

To square: A4

8	BR	.	.	BQ	BK	BB	.	BR
7	BP	BP	BP	.	BP	BP	BP	BP
6	.	.	BN	.	BB	BN	.	.
5	.	.	.	BP
4	WQ	.	.	WP
3	.	.	WP	.	WP	WP	.	.
2	WP	WP	WP	WP
1	WR	WN	WB	.	WK	WB	WN	WR

```
(A   B   C   D   E   F   G   H )
Computer's go:
8| BR   .   .   BQ   BK   BB   .   BR
7| BP   BP   BP   .   .   BP   BP   BP
6| .   .   BN   .   BB   BN   .   .
5| .   .   .   BP   BP   .   .   .
4| WQ   .   .   WP   .   .   .   .
3| .   .   WP   .   WP   WP   .   .
2| WP   WP   .   .   .   .   WP   WP
1| WR   WN   WB   .   WK   WB   WN   WR
(A   B   C   D   E   F   G   H )
```

Computer Moved BP: E7 to E5

Your go USER:

Please enter move in 2 parts

From square: A4

To square: B4

```
8| BR   .   .   BQ   BK   BB   .   BR
7| BP   BP   BP   .   .   BP   BP   BP
6| .   .   BN   .   BB   BN   .   .
5| .   .   .   BP   BP   .   .   .
4| .   WQ   .   WP   .   .   .   .
3| .   .   WP   .   WP   WP   .   .
2| WP   WP   .   .   .   .   WP   WP
1| WR   WN   WB   .   WK   WB   WN   WR
(A   B   C   D   E   F   G   H )
```

Computer's go:

```
8| BR   .   .   BQ   BK   .   .   BR
7| BP   BP   BP   .   .   BP   BP   BP
6| .   .   BN   .   BB   BN   .   .
5| .   .   .   BP   BP   .   .   .
4| .   BB   .   WP   .   .   .   .
3| .   .   WP   .   WP   WP   .   .
2| WP   WP   .   .   .   .   WP   WP
1| WR   WN   WB   .   WK   WB   WN   WR
(A   B   C   D   E   F   G   H )
```

Computer Moved BB: F8 to B4

Your go USER:

Please enter move in 2 parts

From square: C3

To square: B4

```
8| BR   .   .   BQ   BK   .   .   BR
7| BP   BP   BP   .   .   BP   BP   BP
6| .   .   BN   .   BB   BN   .   .
5| .   .   .   BP   BP   .   .   .
```

4	.	WP	.	WP
3	WP	WP	.	.
2	WP	WP	WP	WP
1	WR	WN	WB	.	WK	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer's go:

8	BR	.	.	BQ	BK	.	.	BR
7	BP	BP	BP	.	.	BP	BP	BP
6	.	.	BN	.	BB	BN	.	.
5	.	.	.	BP
4	.	WP	.	BP
3	WP	WP	.	.
2	WP	WP	WP	WP
1	WR	WN	WB	.	WK	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer Moved BP: E5 to D4

Your go USER:

Please enter move in 2 parts

From square: E1

To square: D2

8	BR	.	.	BQ	BK	.	.	BR
7	BP	BP	BP	.	.	BP	BP	BP
6	.	.	BN	.	BB	BN	.	.
5	.	.	.	BP
4	.	WP	.	BP
3	WP	WP	.	.
2	WP	WP	.	WK	.	.	WP	WP
1	WR	WN	WB	.	.	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer's go:

8	BR	.	.	BQ	BK	.	.	BR
7	BP	BP	BP	.	.	BP	BP	BP
6	.	.	BN	.	BB	BN	.	.
5	.	.	.	BP
4	.	WP
3	BP	WP	.	.
2	WP	WP	.	WK	.	.	WP	WP
1	WR	WN	WB	.	.	WB	WN	WR
	(A	B	C	D	E	F	G	H)

Computer Moved BP: D4 to E3

CHECK!

Your go USER:

Please enter move in 2 parts

From square: D2

```
To square: E3
8| BR  .  .  BQ  BK  .  .  BR
7| BP  BP  BP  .  .  BP  BP  BP
6| .  .  BN  .  BB  BN  .  .
5| .  .  .  BP  .  .  .  .
4| .  WP  .  .  .  .  .  .
3| .  .  .  .  WK  WP  .  .
2| WP  WP  .  .  .  .  WP  WP
1| WR  WN  WB  .  .  WB  WN  WR
(A  B  C  D  E  F  G  H )
```

Computer's go:

```
8| BR  .  .  BQ  BK  .  .  BR
7| BP  BP  BP  .  .  BP  BP  BP
6| .  .  BN  .  BB  BN  .  .
5| .  .  .  .  .  .  .  .
4| .  WP  .  BP  .  .  .  .
3| .  .  .  .  WK  WP  .  .
2| WP  WP  .  .  .  .  WP  WP
1| WR  WN  WB  .  .  WB  WN  WR
(A  B  C  D  E  F  G  H )
```

Computer Moved BP: D5 to D4

CHECK!

Your go USER:

```
Please enter move in 2 parts
From square: E3
To square: D4
This isn't a legal move, try again
```

You can see 1 problem and some sophisticated behaviours.

- It prints out the board after each move, makes it clear who's turn it is and what move the computer has made.
- We can see that the validation works as I try in my second move to move in invalid ways or to input jargon and the program asks me again.
- We can also see that, when given the opportunity the computer sacrificed a rook to take a queen.
- We see that I cannot move in a way that causes check. This is after I moved out my king to present an easy check. We also see that it printed check which is useful

The issue we see is when the computer moves a pawn from E7 to E5 over the top of a bishop. This isn't a legal move but I forgot to check that the middle square and more to squares were empty when coding in the pawns movement. **This will be corrected for future.**

Testing

I created the following automated test for the minimax function

```
# import local modules
# cannot import game as causes circular import, if necessary put in same
file
from board_state import Board_State
from assorted import ARBITRARILY_LARGE_VALUE
from vector import Vector

# my minimax function takes as arguments:
# Board_State, is_maximiser, alpha and beta (used for pruning) and
check_extra_depth (produces better outcome but slower)
# it returns
# score, child, move

def minimax(board_state: Board_State, is_maximizer: bool, depth, alpha,
beta, check_extra_depth=True):
    # sourcery skip: low-code-quality, remove-unnecessary-else, swap-if-
else-branches
    # assume white is maximizer
    # when calling, if give appropriate max min arg

    # base case
    # if over or depth==0 return static evaluation
    over, _ = board_state.is_game_over_for_next_to_go()
    if depth == 0 or over:
        # special recursive case 1
        # examine terminal nodes that are check to depth 2 (variable
depth)

            # to avoid goose chaises, extra resources are allowed if check not
already explored
            if board_state.color_in_check() and check_extra_depth and not
over:
                # print(f"checking board state {hash(board_state)} at
additional depth due to check")
                return minimax(
                    board_state=board_state,
                    is_maximizer=is_maximizer,
                    depth=2,
                    alpha=alpha,
                    beta=beta,
                    check_extra_depth=False
                )
            else:
                # static eval works for game over to
```

```
        return board_state.static_evaluation(), None, None

    # define variables used to return more than just score (move and
    child)
    best_child_game_state: Board_State | None = None
    best_move_vector: Vector | None = None

    # function yields move ordered by how favorable they are (low depth
    minimax approximation)
    def gen_ordered_child_game_states():
        # this function does a low depth minimax recursive call (special
        recursive case 2) to give a move a score
        def approx_score_move(move):
            child_game_state = board_state.make_move(*move)

            return minimax(
                board_state=child_game_state,
                depth=depth-2,
                is_maximizer=not is_maximizer,
                alpha=alpha,
                beta=beta,
                check_extra_depth=False
            )[0]
            # print(f"approx_score_move(move={move!r}) -> {result!r}")

        # if depth is 1 or less just yield moves from legal moves
        if depth <= 1:
            yield from board_state.generate_legal_moves()
        # else sort them
        else:
            # sort best to worse
            # sort ascending order if minimizer, descending if maximizer
            yield from sorted(
                board_state.generate_legal_moves(),
                key=approx_score_move,
                reverse=is_maximizer
            )

    if is_maximizer:
        # set max to -infinity
        maximum_evaluation = (-1)*ARBITRARILY_LARGE_VALUE

        # iterate through moves and resulting game states
        for position_vector, movement_vector in
gen_ordered_child_game_states():
```

```
        child_game_state =
board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)

        # evaluate each one
        # general recursive case 1
        evaluation, _, _ = minimax(
            board_state=child_game_state,
            is_maximizer=not is_maximizer,
            depth=depth-1,
            alpha=alpha,
            beta=beta,
            check_extra_depth=check_extra_depth
        )

        # update alpha and max evaluation
        if evaluation > maximum_evaluation:
            maximum_evaluation = evaluation
            best_child_game_state = child_game_state
            best_move_vector = (position_vector, movement_vector)
            alpha = max(alpha, evaluation)

        # where possible, prune
        if beta <= alpha:
            # print("Pruning!")
            break
        # once out of loop, return result
        return maximum_evaluation, best_child_game_state, best_move_vector

else:
    minimum_evaluation = ARBITRARILY_LARGE_VALUE

    for position_vector, movement_vector in
gen_ordered_child_game_states():
        child_game_state =
board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)
        evaluation, _, _ = minimax(
            board_state=child_game_state,
            is_maximizer=not is_maximizer,
            depth=depth-1,
            alpha=alpha,
            beta=beta,
            check_extra_depth=check_extra_depth
        )

        if evaluation < minimum_evaluation:
            minimum_evaluation = evaluation
```

```
        best_child_game_state = child_game_state
        best_move_vector = (position_vector, movement_vector)
        beta = min(beta, evaluation)

    if beta <= alpha:
        # print("Pruning!")
        break
    return minimum_evaluation, best_child_game_state, best_move_vector
```

it features some efficiency upgrades:

- Alpha beta pruning
- Child nodes examined best first to enhance pruning
- Variable depth in check to close out games

I then created a test for the minimax function

```
# this test is responsible for testing various mutations of the minimax
function and how they play, it is not a data driven test

# imports
from random import choice as random_choice
import unittest
# from functools import wraps
import multiprocessing
import os.path
import csv

from game import Game
from minimax import minimax
from assorted import ARBITRARILY_LARGE_VALUE
# from board_state import Board_State
# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given
# function a given number of times
# def repeat_decorator_factory(times):
#     def decorator(func):
#         @wraps(func)
#         def wrapper(*args, **kwargs):
#             for _ in range(times):
#                 func(*args, **kwargs)
#         return wrapper
#     return decorator
```

```
# this is a utility function that maps a function across an iterable but
also converts the result to a list data structure
def list_map(func, iter):
    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a
message
# it converts pieces to symbols
def map_pieces_matrix_to_symbols(pieces_matrix):
    return list_map(
        lambda row: list_map(
            lambda square: square.symbol() if square else None,
            row
        ),
        pieces_matrix
    )

# this functions updates a CSV file with the moves and scores of a chess
game for graphical analysis in excel
def csv_write_move_score(file_path, move, score):
    # convert move to a pair of squares
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector

    from_square = position_vector.to_square()
    to_square = resultant_vector.to_square()

    # if file doesn't exist, create is and add the headers
    if not os.path.exists(file_path):
        with open(file_path, "w", newline="") as file:
            writer = csv.writer(file, delimiter=",")
            writer.writerow(("from_square", "to_square", "score"))

    # add data as a new row
    with open(file_path, "a", newline="") as file:
        writer = csv.writer(file, delimiter=",")
        writer.writerow((from_square, to_square, score))

# this contains the majority of the logic to do a bot vs bot test with the
game class
# it is a component as it isn't the whole test
def minimax_test_component(description, good_bot, bad_bot,
success_criteria, write_to_csv=False):
    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,
    # the test is designed to assert that good bot wins (and or draws in
some cases)
```

```
# generate csv path
if write_to_csv:
    # not sure why but the description sometimes contains an erroneous
    colon, this is caught and removed
    # was able to locate bug to here, as it is a test I added a quick
    fix
    # bug located, some description strings included them
    csv_path = f"test_reports/{description}.csv".replace(" ",
"").replace(":", "")

# start a new blank game
# depth irrelevant as computer move function passed as parameter
game: Game = Game()

# keep them making moves until return statement breaks loop
while True:
    # get move choice from bad bot
    _, _, move_choice = bad_bot(game)

    # serialised to is can be passed as a user move (reusing game
    class)
    position_vector, movement_vector = move_choice
    resultant_vector = position_vector + movement_vector
    from_square, to_square = position_vector.to_square(),
    resultant_vector.to_square()

    # implement bad bot move and update csv
    move, score = game.implement_user_move(from_square=from_square,
    to_square=to_square)
    if write_to_csv:
        csv_write_move_score(
            file_path=csv_path,
            move=move,
            score=score
        )

    # see if this move causes the test to succeed or fail or keep
    going
    success, msg, board_state = success_criteria(game,
    description=description)
    if success is not None:
        return success, msg, board_state

    # providing good bot function, implement good bot move and update
    csv
    move, score =
game.implement_computer_move(best_move_function=good_bot)
```

```

    if write_to_csv:
        csv_write_move_score(
            file_path=csv_path,
            move=move,
            score=score
        )

        # again check if this affects the test
        success, msg, board_state = success_criteria(game,
description=description)
        if success is not None:
            return success, msg, board_state

        # # if needed provide console output to clarify that slow bot
hasn't crashed
        # if game.move_counter % 10 == 0 or depth >= 3:
        # print(f"Moves {game.move_counter}: static evaluation ->
{game.board_state.static_evaluation()}, Minimax evaluation -> {score} by
turn {description}")

# below are some function that have been programmed as classes with a
__call__ method.
# these are basically fancy functions that CAN BE HASHED.
# I had to manually do this under the hood hashing as it is needed to
allow communication between the threads
# a job must be hashable to be piped to a thread (separate python
instance)

class Random_Bot():
    # picks a random move
    def __call__(self, game):
        # determine move at random
        legal_moves = list(game.board_state.generate_legal_moves())
        assert len(legal_moves) != 0
        # match minimax output structure
        # score, best_child, best_move
        return None, None, random_choice(legal_moves)
    def __hash__(self) -> int:
        return hash("I am random bot, I am a unique singleton so each
instance can share a hash")

# picks a good move
# has constructor to allow for configuration
class Good_Bot():

```

```

# configure for depth and allow variable depth
def __init__(self, depth, check_extra_depth):
    self.depth = depth
    self.check_extra_depth = check_extra_depth

# make minimax function call given config
def __call__(self, game):
    return minimax(
        board_state=game.board_state,
        is_maximizer=(game.board_state.next_to_go == "W"),
        depth=self.depth,
        alpha=(-1)*ARBITRARILY_LARGE_VALUE,
        beta=ARBITRARILY_LARGE_VALUE,
        check_extra_depth=self.check_extra_depth
    )
def __hash__(self) -> int:
    return hash(f"Good_Bot(depth={self.depth},"
check_extra_depth={self.check_extra_depth})")

# used to look at a game and decide if the test should finish
class Success_Criteria():
    # constructor allow config for stalemates to sill allow test to pass
    def __init__(self, allow_stalemate_3_states_repeated: bool):
        self.allow_stalemate_3_states_repeated =
allow_stalemate_3_states_repeated
    def __call__(self, game: Game, description):
        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do
        match game.check_game_over():

            # if 3 repeat stalemate, check with config wether is is allows
            case True, None, "Stalemate":
                # game.board_state.print_board()
                if game.is_3_board_repeats_in_game_history and
self.allow_stalemate_3_states_repeated:
                    # game.board_state.print_board()
                    # print(f"Success: Stalemate at {game.moves} moves in
test {description}: 3 repeat board states, outcome specify included in
allowed outcomes")
                    return True, f"Success: Stalemate at
{game.move_counter} moves in test {description}: 3 repeat board states,
outcome specify included in allowed outcomes",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
            else:
                # game.board_state.print_board()
                # print(f"FAILURE: ({description}) stalemate caused (3
repeats? -> {game.is_3_board_repeats_in_game_history()})")

```

```

        return False, f"FAILURE: ({description}) stalemate
caused (3 repeats? -> {game.is_3_board_repeats_in_game_history()})", 
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
    # good bot loss causes test to fail
    case True, 1, "Checkmate":
        # game.board_state.print_board()
        # print(f"Failure: ({description}) computer lost")
        return False, f"Failure: ({description}) computer lost",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
    # good bot win causes test to pass
    case True, -1, "Checkmate":
        # game.board_state.print_board()
        # print(f"SUCCESS: ({description}) Game has finished and
been won in {game.move_counter} moves")
        return True, f"SUCCESS: ({description}) Game has finished
and been won in {game.move_counter} moves",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
    # if the game isn't over, return success as none and test will
continue
    case False, _, _:
        return None, None, None

def hash(self):
    return
hash(f"Success_Criteria(allow_stalemate_3_states_repeated={self.allow_stal
emate_3_states_repeated})")

# given a test package (config for one test), carry it out
def execute_test_job(test_data_package):
    # deal with unexplained bug where argument is tuple / list length 1
    # containing relevant dict (quick fix as only a test)
    # I was able to identify that this is where it occurs and add a
    # correction but I am not sure what the cause of the bug is
    if any(isinstance(test_data_package, some_type) for some_type in
(tuple, list)):
        if len(test_data_package) == 1:
            test_data_package = test_data_package[0]

    # print(f"test_data_package --> {test_data_package}")

    # really simple, call minimax test component providing all keys in
    # package as keyword arguments
    return minimax_test_component(**test_data_package)

# this function takes an iterable of hashable test_packages

```

```
# it all 8 logical cores on my computer to multitask to finish the test
sooner
def pool_jobs(test_data):
    # counts logical cores
    # my CPU is a 10th gen i7
    # it has 4 cores and 8 logical cores due to hyper threading
    # with 4-8 workers I can use 100% of my CPU
    cores = multiprocessing.cpu_count()

    # create a pool
    with multiprocessing.Pool(cores) as pool:
        # map the execute_test_job function across the set of test
        packages using multitasking
        # return the result
        return pool.map(
            func = execute_test_job,
            iterable = test_data
        )

# test case contains unit tests
# multitasking only occurs within a test, tests are themselves executed
sequentially
# I could pool all tests into one test function but this way multiple
failures can occur in different tests
# (one single test would stop at first failure)

class Test_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and
    checks the results with a unittest
    # a failure is correctly identified to correspond to the function that
    called this function
    # reduces repeated logic
    def check_test_results(self, test_results):
        for success, msg, final_pieces_matrix in test_results:
            # print()
            # for row in final_pieces_matrix:
            #     row = " ".join(map(
            #         lambda square: str(square).replace("None", ". "),
            #         row
            #     ))
            #     print(row)
            # print(msg)

            # i choose to iterate rather than assert all as this allows me
            to have the appropriate message on failure
            self.assertTrue(
                success,
```

```
        msg=msg
    )

# tests basic minimax vs random moves
def test_vanilla_depth_1_vs_randotron(self):
    # 10 trials as outcome is linked to a random behaviour
    trials = 10

        # test package generated to include relevant data and logic (bots
and success criteria)
    test_data = {
        "description": "test: depth 1 vanilla vs randotron",
        "good_bot": Good_Bot(depth=1, check_extra_depth=False),
        "bad_bot": Random_Bot(),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
        "write_to_csv": False
    }

        # only trial will write to a csv
    test_data_but_to_csv = test_data
    test_data_but_to_csv["write_to_csv"] = True

    self.check_test_results(
        pool_jobs(
            (trials-1) * [test_data] + [test_data_but_to_csv]
        )
    )

def test_advanced_depth_1_vs_randotron(self):
    trials = 10

    test_data = {
        "description": "test: depth 1 advanced vs randotron",
        "good_bot": Good_Bot(depth=1, check_extra_depth=True),
        "bad_bot": Random_Bot(),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
        "write_to_csv": False
    }

    test_data_but_to_csv = test_data
    test_data_but_to_csv["write_to_csv"] = True

    self.check_test_results(
        pool_jobs(
            (trials-1) * [test_data] + [test_data_but_to_csv]
        )
    )
```

```
)  
  
def test_depth_2_vs_randotron(self):  
    trials = 10  
  
    test_data = {  
        "description": "test: depth 2 advanced vs randotron",  
        "good_bot": Good_Bot(depth=2, check_extra_depth=True),  
        "bad_bot": Random_Bot(),  
        "success_criteria":  
            Success_Criteria(allow_stalemate_3_states_repeated=False),  
            "write_to_csv": False  
    }  
  
    test_data_but_to_csv = test_data  
    test_data_but_to_csv["write_to_csv"] = True  
  
    self.check_test_results(  
        pool_jobs(  
            (trials-1) * [test_data] + [test_data_but_to_csv]  
        )  
    )  
  
def test_depth_1_advanced_vs_depth_1_vanilla(self):  
    # only one trial needed as outcome is deterministic  
    trials = 1  
  
    test_data = {  
        "description": "test: depth 1 vanilla vs depth 1 variable  
check",  
        "good_bot": Good_Bot(depth=1, check_extra_depth=True),  
        "bad_bot": Good_Bot(depth=1, check_extra_depth=False),  
        # they aren't different enough in efficacy to guarantee no  
draws  
        "success_criteria":  
            Success_Criteria(allow_stalemate_3_states_repeated=True),  
            "write_to_csv": True  
    },  
  
    self.check_test_results(  
        pool_jobs(trials*[test_data])  
    )  
  
def test_depth_2_vs_depth_1(self):  
    trials = 1  
  
    test_data = {
```

```
        "description": "test: depth 2 vs depth 1",
        "good_bot": Good_Bot(depth=2, check_extra_depth=True),
        "bad_bot": Good_Bot(depth=1, check_extra_depth=True),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": True
        }

        self.check_test_results(
            pool_jobs(trials*[test_data])
        )

def test_depth_3_vs_depth_2(self):
    trials = 1

    test_data = {
        "description": "test: depth 3 vs depth 2",
        "good_bot": Good_Bot(depth=3, check_extra_depth=True),
        "bad_bot": Good_Bot(depth=2, check_extra_depth=True),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": True
        }

        self.check_test_results(
            pool_jobs(trials*[test_data])
        )

def test_depth_3_vs_depth_1(self):
    trials = 1

    test_data = {
        "description": "test: depth 3 vs depth 1",
        "good_bot": Good_Bot(depth=3, check_extra_depth=True),
        "bad_bot": Good_Bot(depth=1, check_extra_depth=True),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": True
        }

        self.check_test_results(
            pool_jobs(trials*[test_data])
        )

def test_depth_3_vs_randomtron(self):
    trials = 4

    test_data = {
```

```

    "description": "test: depth 3 vs randotron",
    "good_bot": Good_Bot(depth=3, check_extra_depth=True),
    "bad_bot": Random_Bot(),
    "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
        "write_to_csv": False
    }

    test_data_but_to_csv = test_data
    test_data_but_to_csv[ "write_to_csv" ] = True

    self.check_test_results(
        pool_jobs(
            (trials-1) * [test_data] + [test_data_but_to_csv]
        )
    )

if __name__ == '__main__':
    unittest.main()

```

I went to great effort to improve the tests efficiency and this wasn't wasted. It allowed me to perform tests with many trials simultaneously in order to fully utilise my CPU.

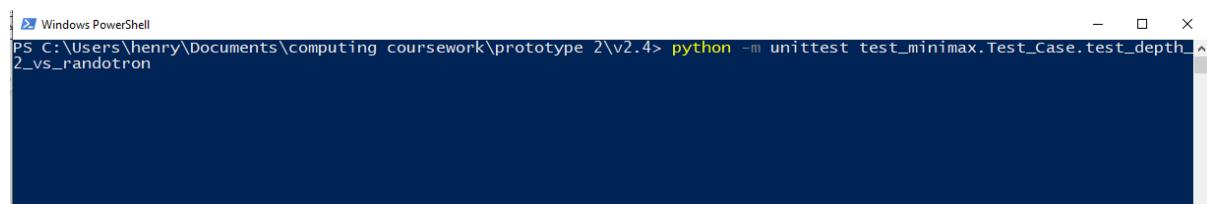
Currently all the test I have run today were successful **but** even given a whole afternoon I haven't been able to run all the tests. This is due to the depth 3 tests. Due to variable depth, the number of static evaluations can be (branching factor)⁵ (really slow). This combinatorial explosion means that it takes more than 4 hours to run the tests.

The tests I have run successfully are:

- test_depth_1_vanilla_vs_depth_1_variable_check
- test_depth_2_vs_depth_1
- test_depth_2_advanced_vs_randotron
- test_depth_3_vs_depth_1
- test_depth_1_advanced_vs_randotron

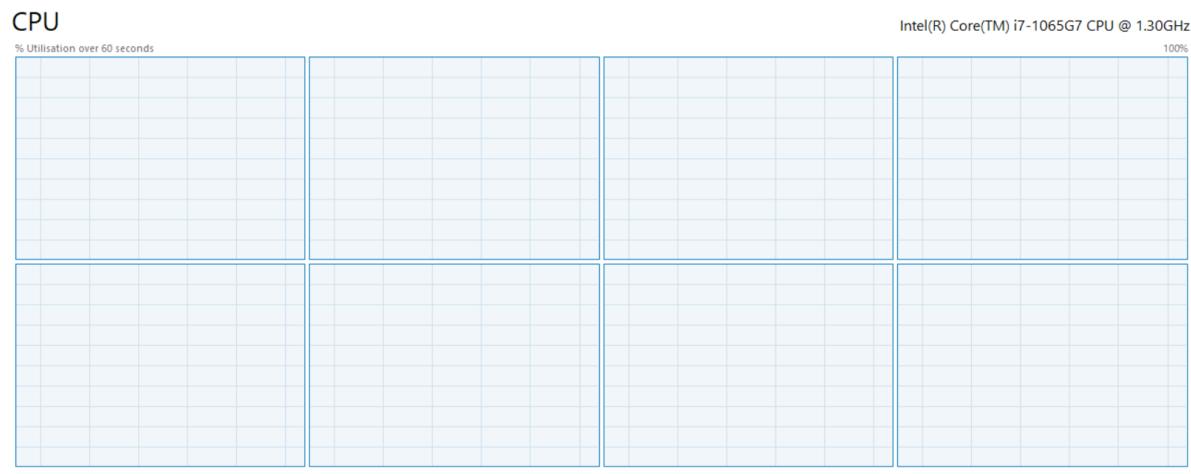
Here is me showing how multitasking has allowed me to fully utilise my CPU

Let me run depth 2 vs randotron with 10 trials to demonstrate:

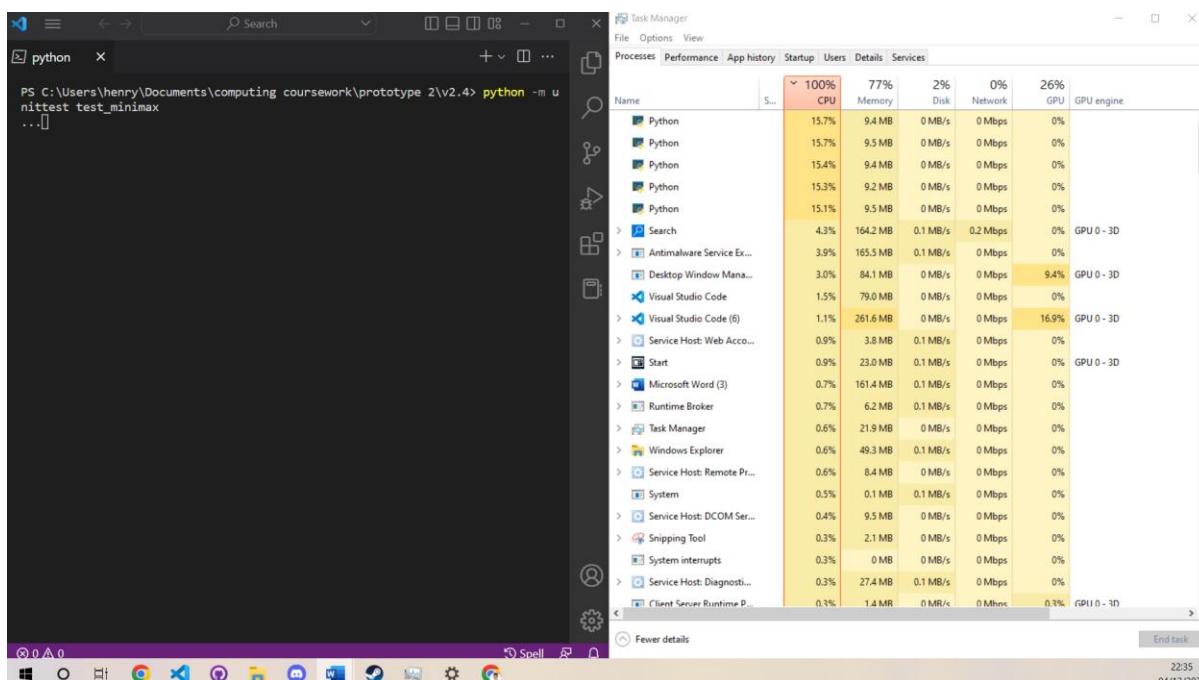
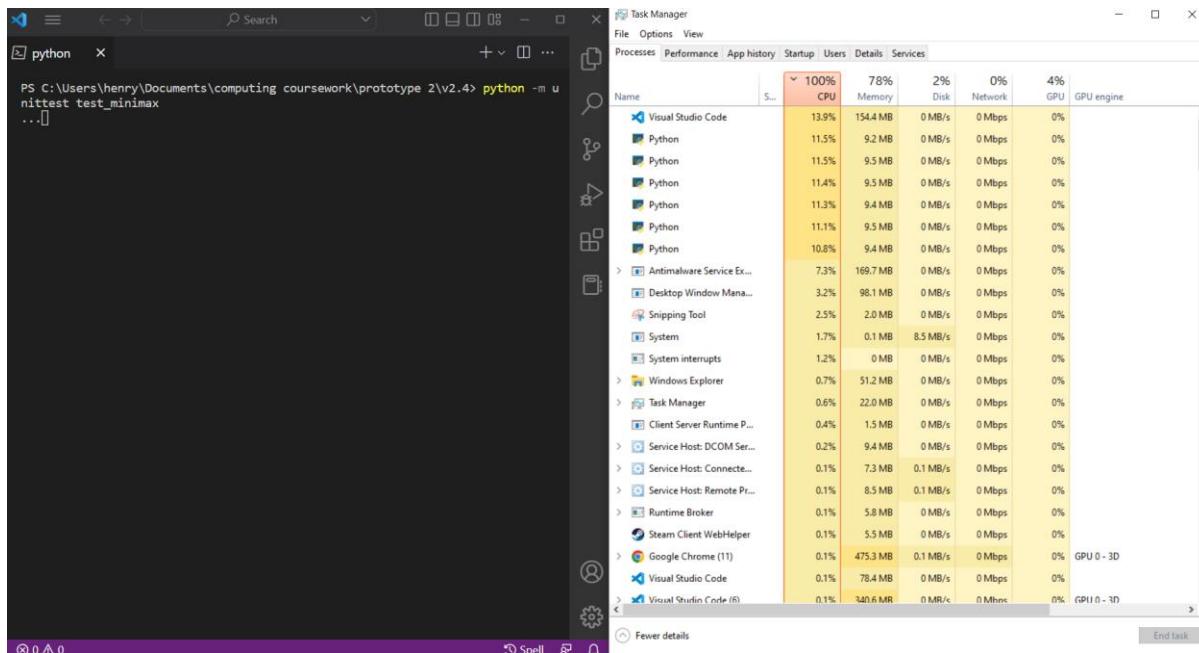


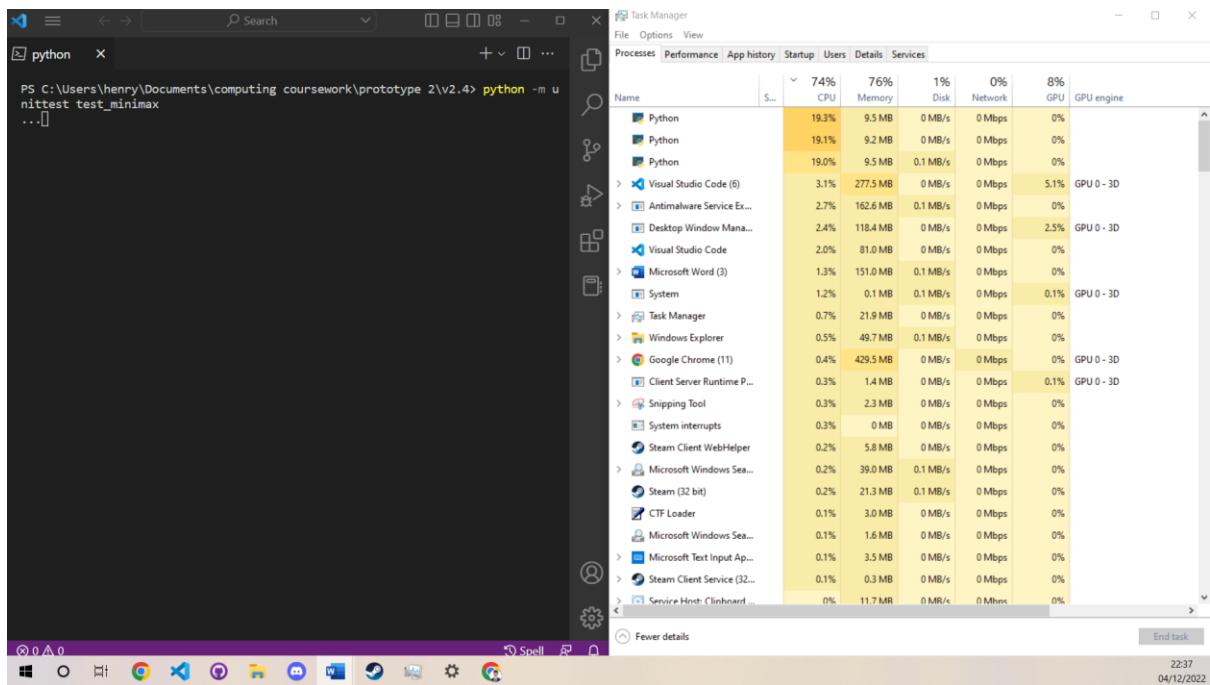
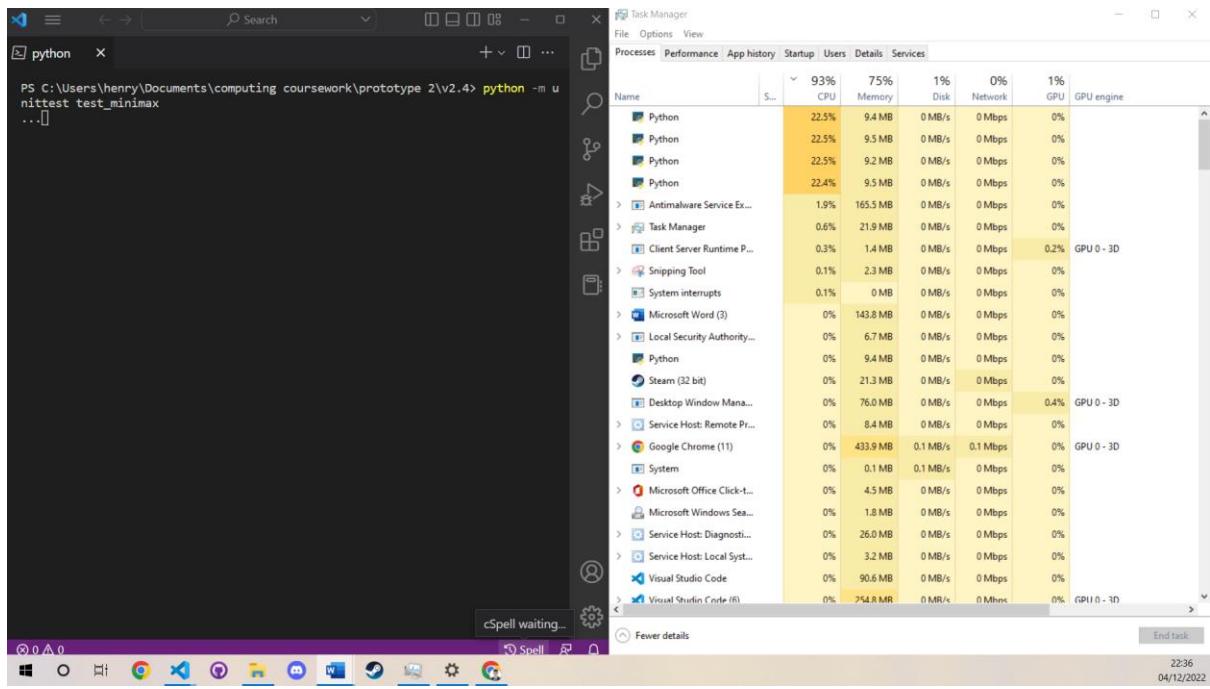
A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell". The command entered is "PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_minimax.Test_Case.test_depth_2_vs_randotron". The window is dark-themed.

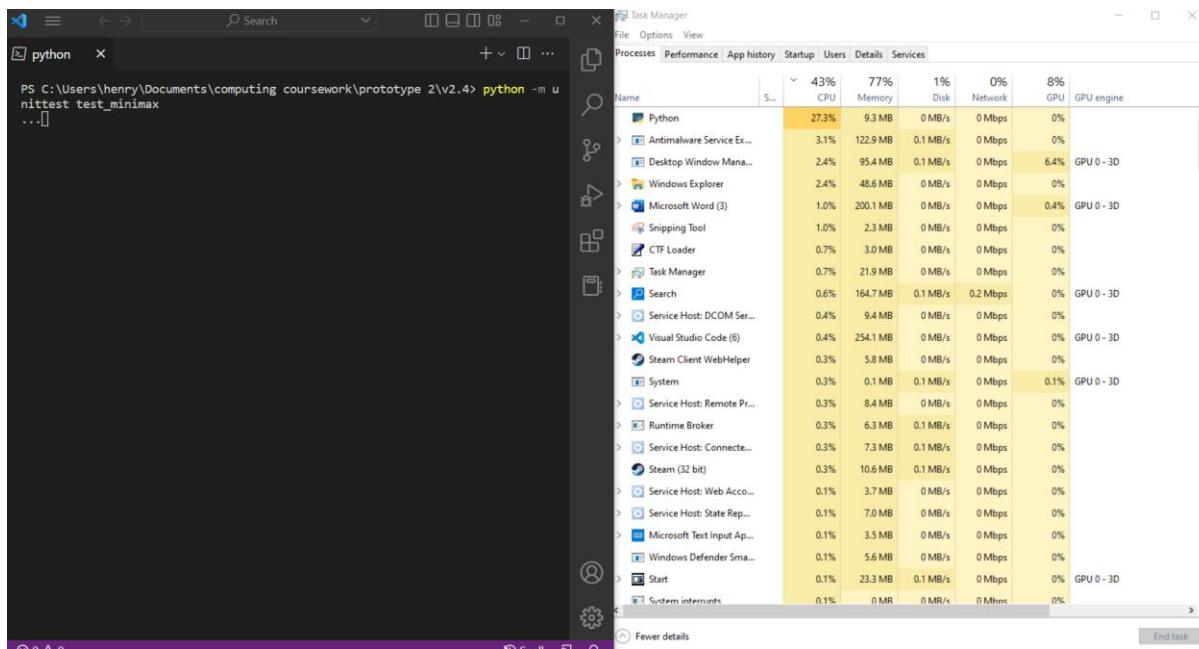
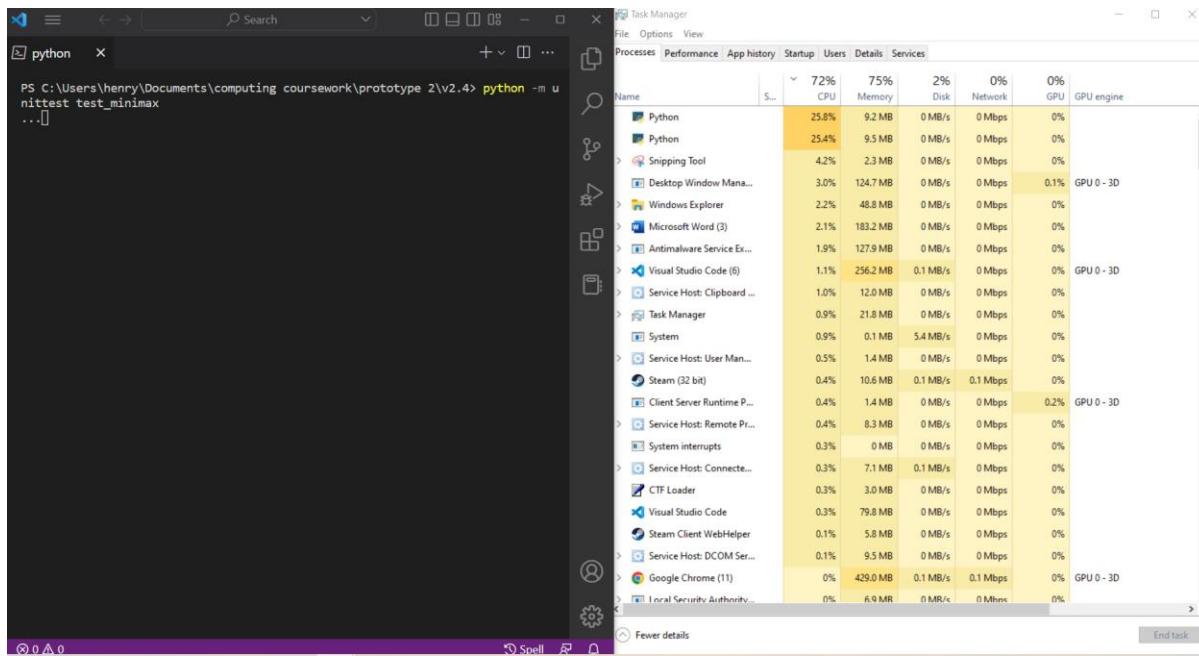
Name	S...	100% CPU	79% Memory	1% Disk	0% Network	0% GPU	GPU engine	Power usage	Power usage t...
Windows PowerShell (11)		90.1%	118.4 MB	0 MB/s	0 Mbps	0%		Very high	Very high
Python		11.9%	9.3 MB	0 MB/s	0 Mbps	0%		Very high	Low
Python		11.7%	9.4 MB	0 MB/s	0 Mbps	0%		Very high	Low
Python		11.4%	9.4 MB	0 MB/s	0 Mbps	0%		Very high	Low
Python		11.4%	9.4 MB	0 MB/s	0 Mbps	0%		Very high	Low
Python		11.1%	9.3 MB	0 MB/s	0 Mbps	0%		High	Low
Python		11.0%	9.4 MB	0 MB/s	0 Mbps	0%		High	Low
Python		11.0%	9.3 MB	0 MB/s	0 Mbps	0%		High	Low
Python		10.7%	9.3 MB	0 MB/s	0 Mbps	0%		High	Low
Windows PowerShell		0%	31.4 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Console Window Host		0%	3.0 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Python		0%	9.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Antimalware Service Ex...		1.9%	153.6 MB	0.1 MB/s	0 Mbps	0%		Low	Very low
Microsoft Windows Sea...		1.2%	52.7 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Search		1.0%	155.9 MB	0.1 MB/s	0 Mbps	0%	GPU 0 - 3D	Very low	Very low
Task Manager		0.9%	24.1 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Windows Explorer		0.7%	116.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Service Host: Web Acco...		0.7%	3.9 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Desktop Window Mana...		0.6%	79.6 MB	0 MB/s	0 Mbps	0.3%	GPU 0 - 3D	Very low	Very low
Service Host: Remote Pr...		0.5%	8.4 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Runtime Broker		0.3%	11.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Service Host: DCOM Ser...		0.3%	9.4 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Service Host: User Man...		0.3%	1.5 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Workplace or school ac...		0.3%	4.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code (18)		0.2%	859.3 MB	0 MB/s	0 Mbps	0%	GPU 0 - 3D	Very low	Very low
		0.1%	2.1 MB	0 MB/s	0 Mbps	0%		Very low	Very low



Before doing this my CPU was only ever able to devote 25% power to my python program.







As tasks finish the number of workers decreases. As my computer has 4 physical cores and 8 logical cores, near 100% cpu usage can continue until there are less than 4 threads. At this point each thread can only use 25% of the available CPU power. This means that the test doesn't use the CPU fully at the end. This is only problematic is one of the games is particularly long and tends to stalemate as the next test cannot begin until this one has finished.

The first time I created the randobot and ran the test for depth 1 I found the issue where the game was resulting in a stalemate where it was endlessly repeating. I looked it up and in chess if a game state is repeated 3 times a stalemate occurs. I implemented this using the hash and frequency table in the Game class. This does end the game when this occurs. However it does mean that that the minimax function is unaware that a stalemate can occur this way as the logic isn't happening in the board state class. This shouldn't be an issue

however as, when an infinite loop occurs the minimax will anticipate that its mean evaluation won't change and so the stalemate wouldn't affect its behaviour greatly anyway.

I have added this to my unit test as bots of similar skill level may sometimes draw. This can be specified in the test criteria.

I also recorded the utility values of the various tests in CSV files so that it could be graphed. I had a problem with this where the workers from each thread were all adding to the file, making it a useless mess. The fix was to add a .copy()

```
# tests basic minimax vs random moves
def test_vanilla_depth_1_vs_randotron(self):
    # 10 trials as outcome is linked to a random behaviour
    # trials = 10
    trials = 5

        # test package generated to include relevant data and logic (bots
and success criteria)
    test_data = {
        "description": "test: depth 1 vanilla vs randotron",
        "good_bot": Good_Bot(depth=1, check_extra_depth=False),
        "bad_bot": Random_Bot(),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
        "write_to_csv": False
    }

    # only trial will write to a csv
    test_data_but_to_csv = test_data.copy()
    test_data_but_to_csv["write_to_csv"] = True

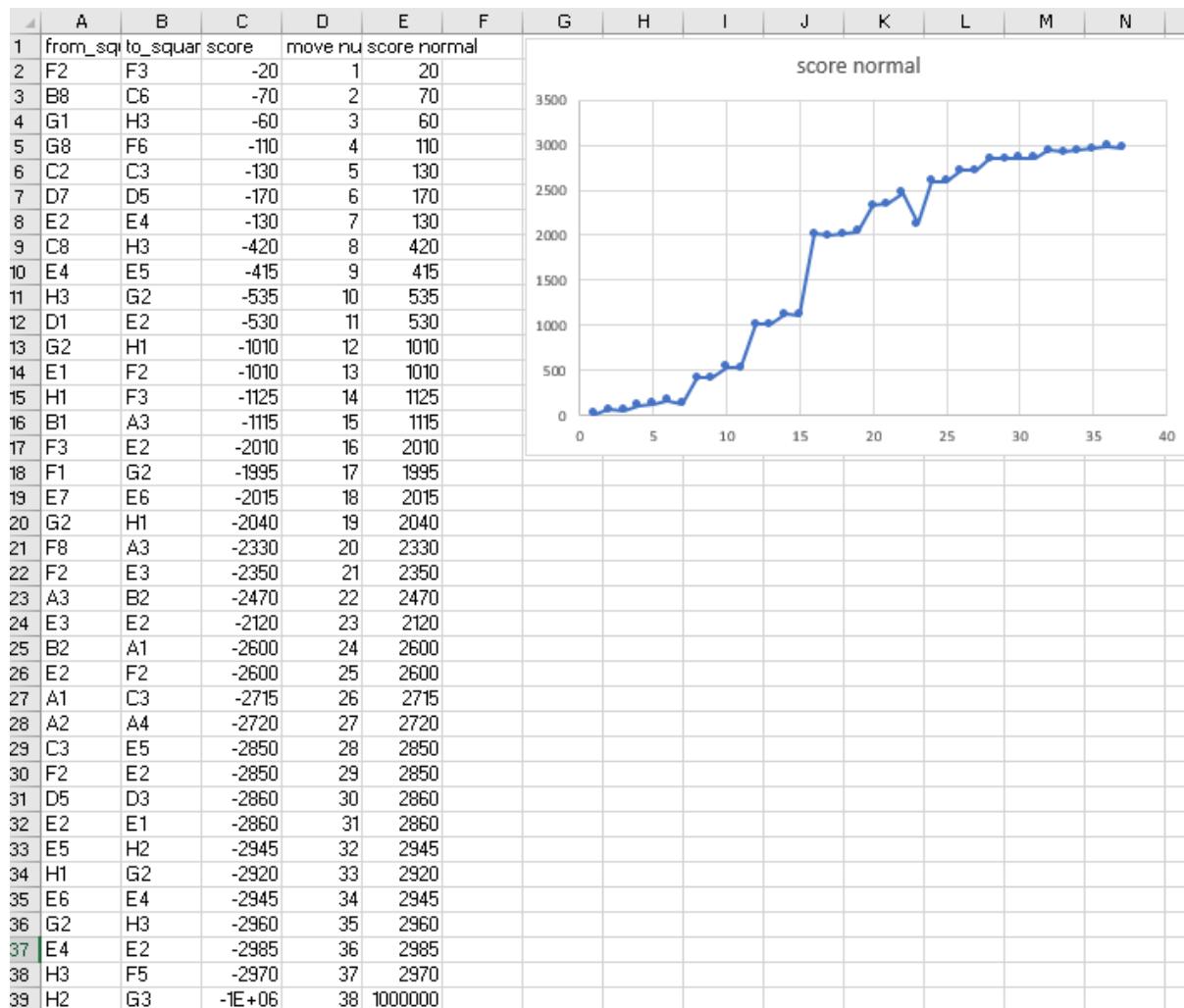
    self.check_test_results(
        pool_jobs(
            (trials-1) * [test_data] + [test_data_but_to_csv]
        )
    )
}
```

The issue was that I was passing the dictionary by reference to a new variable and so I wasn't mutating one test to use the CSV file but all of them.

This issue was particularly frustrating as I was waiting hours for a set of corrupted data.

I have left my unit test running overnight and in 9 hours it still hasn't finished. The depth 3 vs depth 2 and depth 3 vs randotron tests haven't finished. This highlights an issue with the minimax testing as the depth 3 tests far too long. I should still be able to analyse some of the other data:

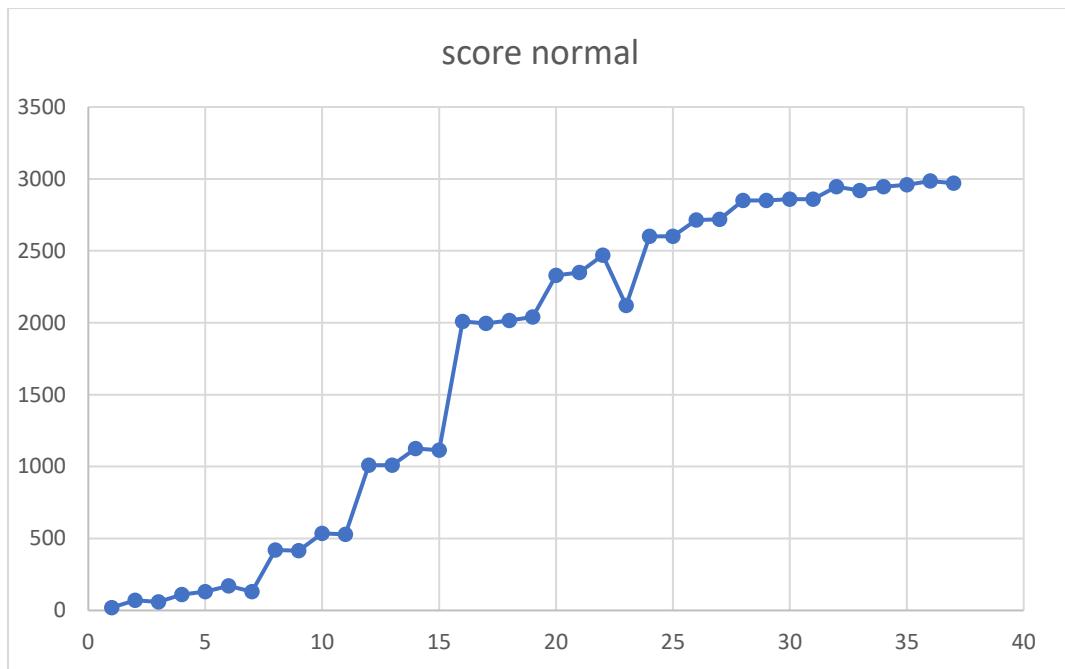
Here is the data to illustrate, I added columns D and E and then created a graph



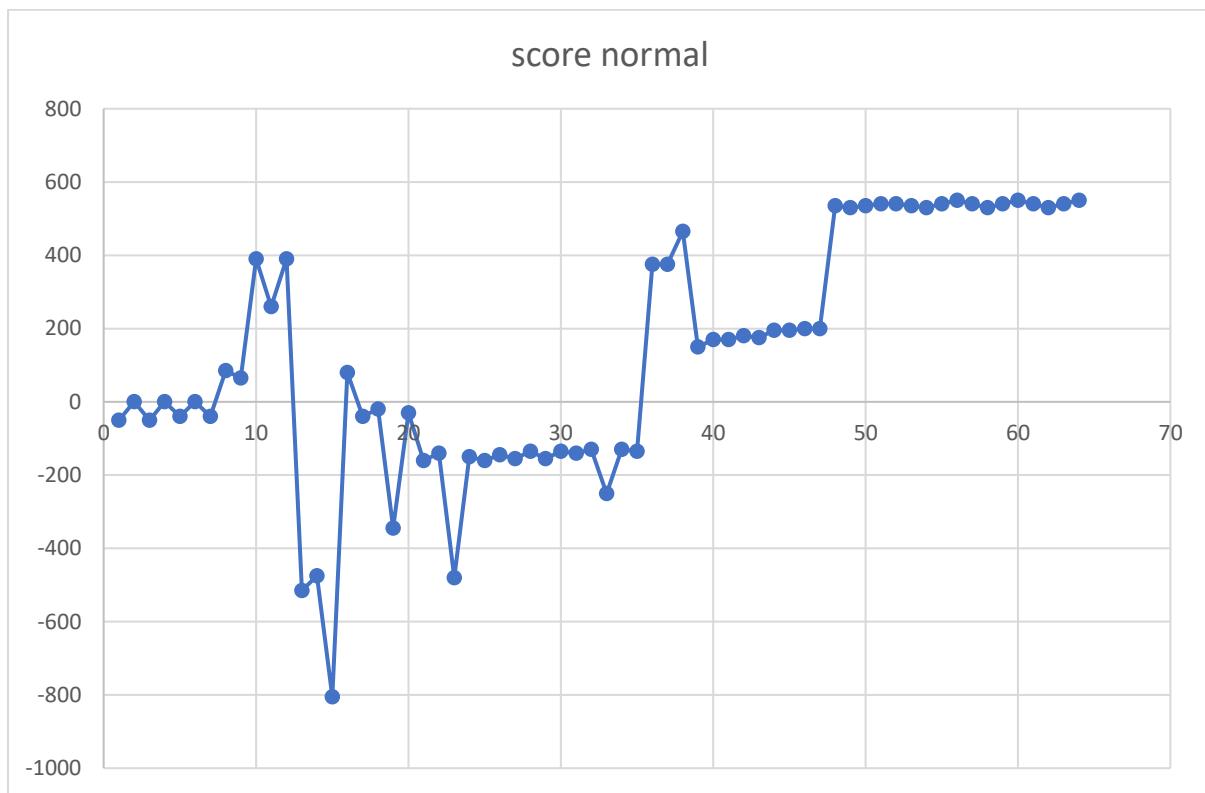
It shows the static evaluations and how it improves for the computer until a win

I will now show some of the graphs:

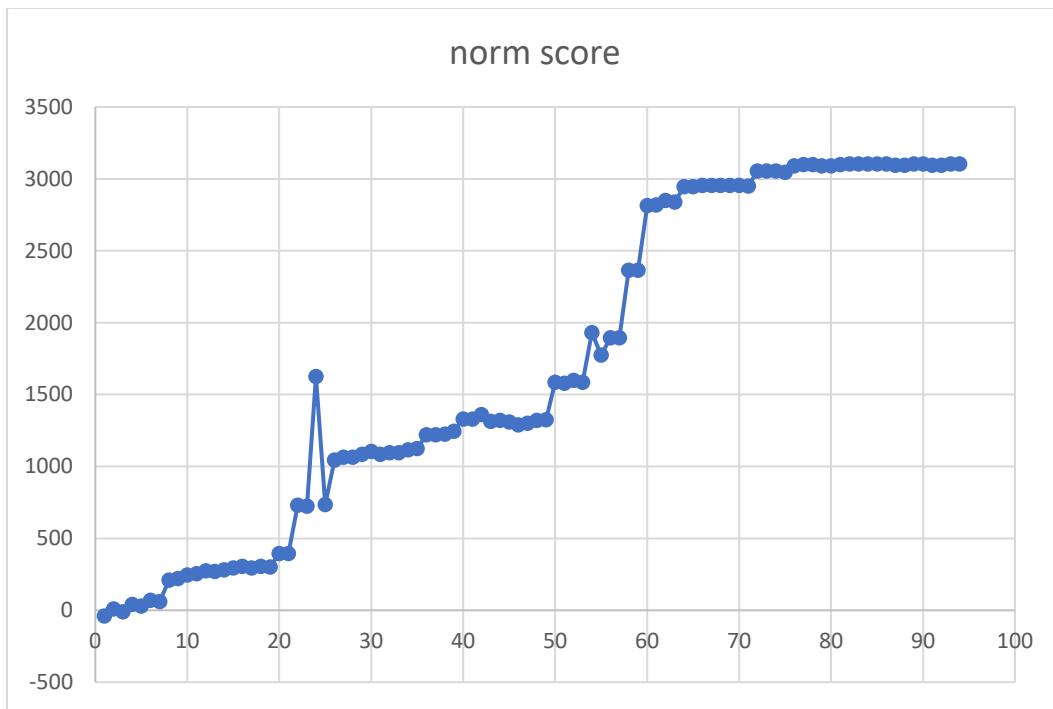
Depth 1 advanced vs Ranotron (ends in a win)



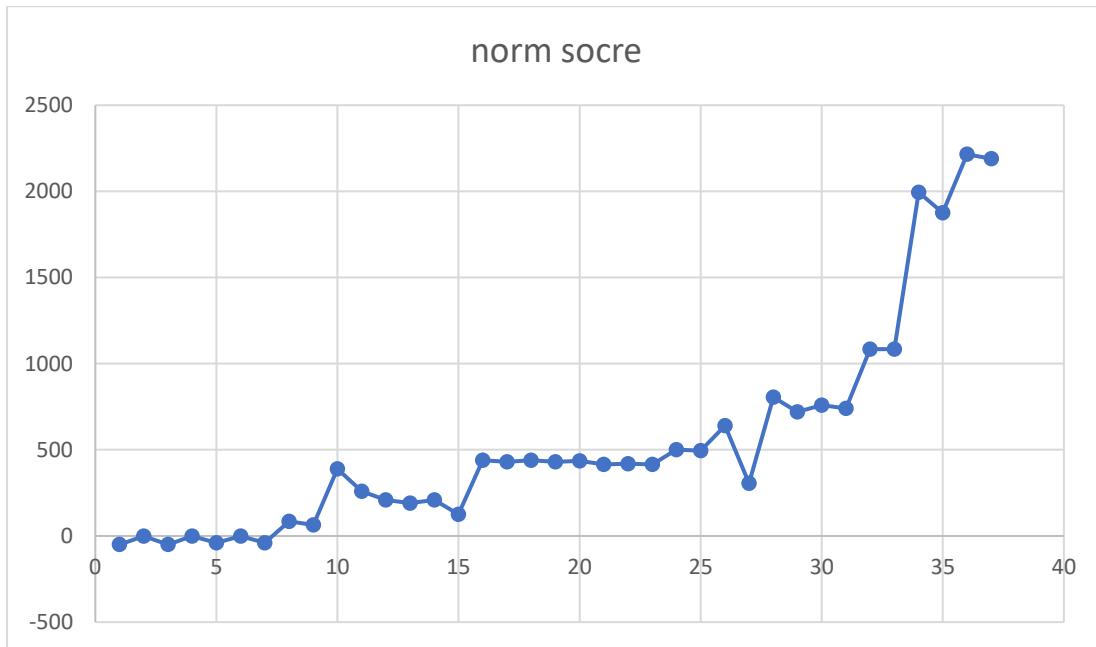
Depth 1 advanced vs depth 1 vanilla (ends in stalemate 3 repeat)



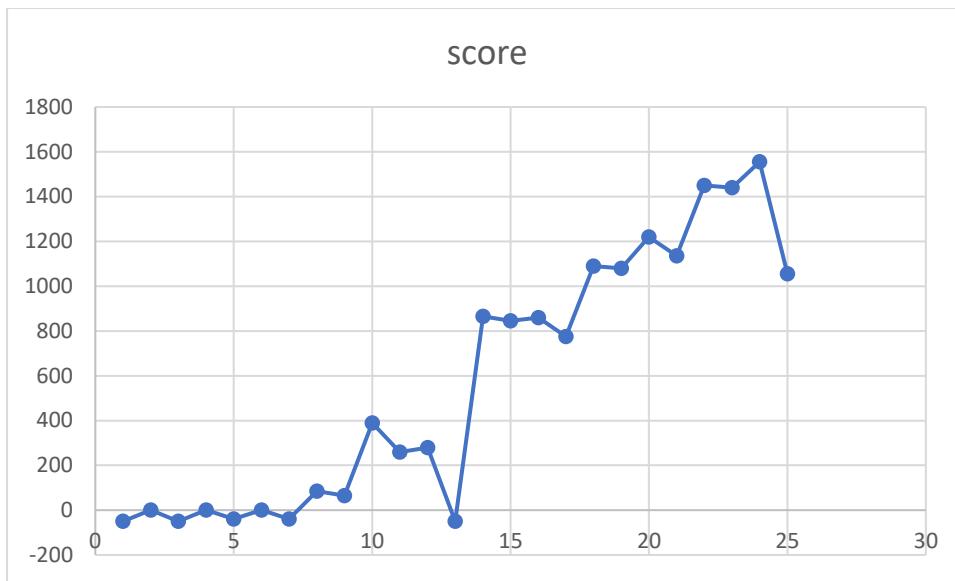
Depth 2 advanced vs Ranotron (ends in win):



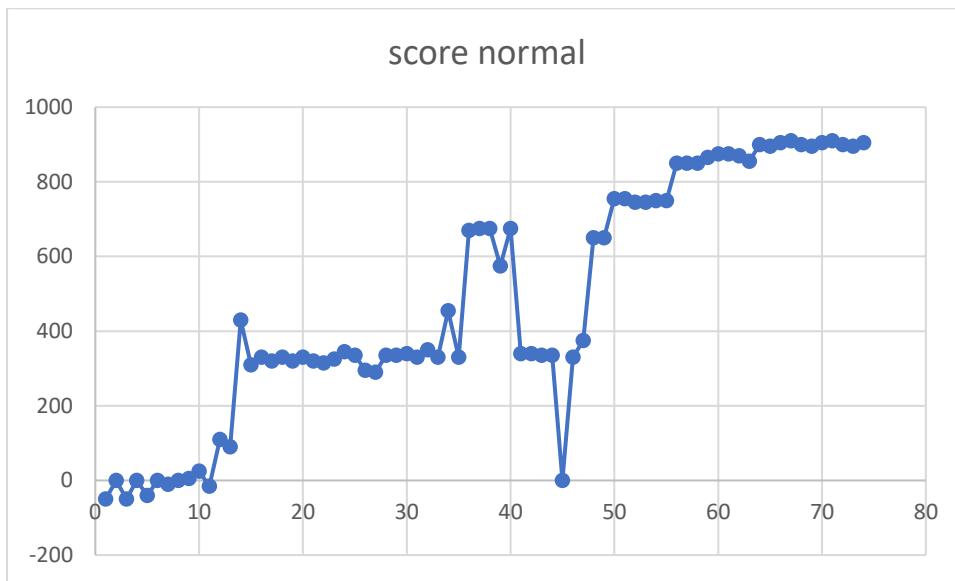
Depth 2 advanced vs depth 1 advanced (deterministic: win every time):



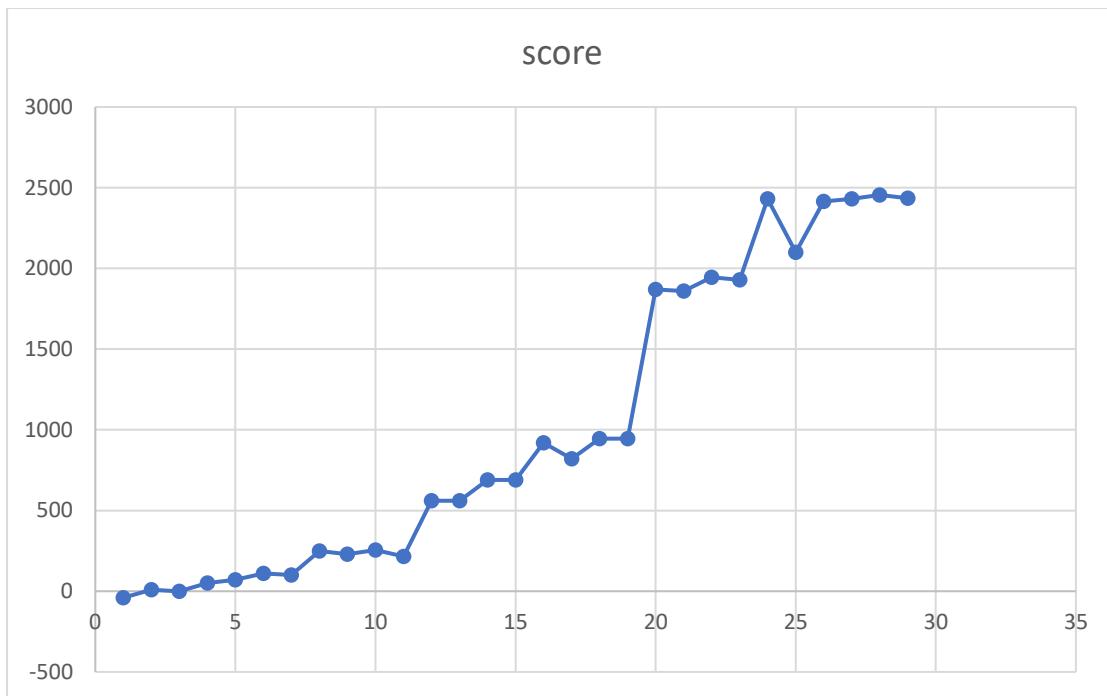
Depth 3 advanced vs depth 1 advanced (deterministic: win every time)



Depth 3 vs depth 2:



Depth 1 vanilla vs Ranotron:



I found analysing these graphs to be very interesting. It seems that both depth 1 and depth 2 can occasionally stalemate with Ranotron. It also seems that depth 1 vanilla vs advanced can stalemate. I believe that this is because they are quite well matched algorithms therefore the better one can only guarantee win or draw.

I thought it was also nice to see how depth 3 vs depth 1 got to checkmate quicker (25 moves) than depth 2 (37 moves). This shows that these algorithms are at their best when playing a rational opponent they can predict.

The antithesis of this is that, while depth 2 wins against depth 1, depth 1 wins quicker against Ranotron. This is because the minimax function assumes the opponent will play optimally. If the opponent isn't playing rationally then carefully selected move will result in the game lasting longer than necessary.

We also see that depth 3 vs depth 2 resulted in a stalemate however depth 3 was able to maximise its score. This shows how the algorithms are relatively similar and are a match for each other (able to draw)

I hope to get the graphs for the other tests. To do this I will run the tests that haven't finished again another time. However not finishing is still an outcome. It shows that it would not currently be possible to play a chess game vs depth 3 as it takes too long. This is due to the combinatorial explosion at a higher depth and the exponential time complexity of the algorithm.

We can also see that depth 1 vanilla beat Ranotron faster (29 moves vs 37 moves) than vanilla depth 1. We also saw that even though these 2 algorithms drew in a 3 repeat stalemate, the advanced version was able to have a higher static evaluation.

All of these results show that:

- my minimax algorithm works

- a higher depth beat lower depth in effectiveness
- variable depth checking beats without in effectiveness

I also saw from manually running the algorithm with and without, that alpha beta pruning was improving performance. It is unclear whether pre-sorting child nodes has improved performance as this performance increase would only kick in at higher depths which I was unable to run before.

Validation:

With regards to validation, some was added in the game class and console chess program. This included:

- invalid input for users move (not chess squares) → value error
- illegal move → illegal move error
- not the users turn to go → not user turn error

This validation worked well enough to allow for the console chess game to be robust. However, the intended use for this engine is in a webserver as part of a full stack website. This means that I will need both client side and server side validation. The server side validation will be added to the flask server.

I intend to add more robust validation when I write the flask server, the chess engine will assume that the inputs it gets are valid.

Feedback from Stakeholders

I received feedback from a stakeholder who played the chess game:

They were very impressed that it could play chess and, in their game, the pawn forward 2 bug didn't arise.

Regarding the user interface they liked:

- the fact you could see the letters and numbers at the side of the board
- the fact that it printed out the computer's move
- the fact that it printed out check
- the fact that it correctly prevented illegal moves when in check
- the fact that it took pieces of high value when given the opportunity

They suggested:

- that I find a way of clearing the console / text output after the computer's move so that the user only sees the one relevant board and not many older ones if they scroll up.

What I took from this feedback:

I will keep this suggestion in mind if I am unable to get the GUI to work in the next prototype. I will also consider adding letters and squares to the edges of the chess board in my GUI. While the user will input their moves by clicking, this will help contextualise the move history.

I did have an issue with the minimax function where the static evaluations were not what they should have been. I realised that the static evaluation for the starting positions was not 0. To correct this I made a test and then reviewed my code. The corrections was to flip the value matrices for black pieces as the matrices were not symmetrical and were from white's perspective. This corrections was easy as it could be made just in the pieces class.

```
# this should use the position vector and value matrix to get the
value of the piece
```

```

def get_value(self, position_vector: Vector):
    # flip if black as matrices are all for white pieces
    if self.color == "W":
        row, column = 7-position_vector.j, position_vector.i
    else:
        row, column = position_vector.j, position_vector.i

    # return sum of inherent value + value relative to positon on
board
    return self._value + self._value_matrix[row][column]

```

I also attempted to refactor the King.movement_vecotor method, accidentally breaking it. This was an easy fix as I ran the unit tests which highlighted the issue.

Changes/Fixes that I now plan to make to the design or code as a result of testing and feedback

- I intend to make changes to the command line interface (CLI) to make it clearer by removing old boards
- I intend to make the minimax function more efficient so that I can run at depth 3 in a reasonable time frame.
- I intend to fix the pawn forward 2 issue and add a test to ensure that it is fixed.
-

Evaluation

Overall I am extremely happy with this iteration.

I feel that in this iteration I accomplished my ambitions goals and created a robust chess engine complete with a chess bot that can beat me (I believe minimax depth 3 with variable depth surpasses my average ability). With odd exceptions such as the pawn move forward 2 bug, everything is fully functional and tested. This is a lesson in how testing isn't always perfect however, without the automated testing, I don't think I would have succeeded.

My stakeholders seem very impressed that my program can actually play chess in an intelligent manner. I am also pleasantly surprised by how well the final product has turned out.

While my testing was not a complete success (as expected) it was invaluable. My pieces module ended up having the most undiagnosed issues so in hindsight I should have tested it more thoroughly. I realised that most of the time testing was not writing tests but debugging functions. Because tests were quick to write I approached the problem like an engineer and wrote many. I was extremely happy with the results. They allowed me to debug essential function to allow my minimax function to work.

Evidence of testing (excluding minimax tests)

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest
-----
Ran 74 tests in 0.288s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> █
```

The interface was not a focus of the project and so it received limited time investment. However, I was able to make the interface clear to my stakeholders by using standard chess symbols for pieces and standard coordinates. Because of this I think that the interface was also, considering requirements and expectations at this state, a success.

This prototype was a success as it was able to achieve the specified aims beyond the bare minimum. This will make producing the next prototype much easier as I have already created a robust chess engine.

Iterative Development – Prototype 3:

The aim for this prototype is to bring together all of the elements created in the previous prototypes and improve them to create a final program. For example, I will use my exploration into sockets from prototype 1 and the chess computer that I created in prototype 2.

This final prototype should include a graphical user interface in the form of a webpage. The interface should be interactive and should be connected to the server, allowing for a full chess game where the computer make moves to be facilitated.

Functionality that the prototype should have:

The prototype should aim to meet all desirable, high priority and essential components of the success criteria. To do this it must meet all of the 31 points laid out in the design specification within the design section.

To do this the move engine must be optimised. Then the code for the client and server API handlers must be created. Then client side code including the `Chess_Board` class and DOM methods must be created.

Annotated code screenshots with description:

The first part of this project was to improve the structure of the python project. The previous structure uses many python files in the same folder for easy import statements. The new directory structure has divided the code up logically into different modules. Here is the directory tree.

```
PS C:\Users\henry\Documents\computing coursework\prototype 6> tree  
/f
```

```
Folder PATH listing for volume OS  
Volume serial number is 4ED8-B070
```

```
C:  
|   app.py  
|   run_tests_fast.ps1  
|   run_tests_slow.ps1  
|   schema_playground.ipynb  
|   __init__.py
```

```
   |--.vscode  
       settings.json
```

```
   |--assorted  
       |   chess_exceptions.py  
       |   general.py  
       |   safe_hash.py  
       |   __init__.py
```

```
|   └── pycache_
|       assorted.cpython-310.pyc
|       chess_exceptions.cpython-310.pyc
|       general.cpython-310.pyc
|       random_board_state.cpython-310.pyc
|       safe_hash.cpython-310.pyc
|       __init__.cpython-310.pyc
|
└── chess_functions
    board_state.py
    pieces.py
    test_fast_board_state.py
    test_fast_pieces.py
    test_fast_vector.py
    vector.py
    __init__.py
|
└── test_data
    ├── board_state
    |   color_in_check.yaml
    |   game_over.yaml
    |   generate_all_pieces.yaml
    |   generate_legal_moves.yaml
    |   generate_pieces_of_color.yaml
    |   piece_at_vector.yaml
    |
    ├── pieces
    |   test_board_populated.yaml
    |   test_empty_board.yaml
    |
    └── vector
        from_square.yaml
        vector_add.yaml
        vector_in_board.yaml
        vector_multiply.yaml
        vector_to_square.yaml
|
└── pycache_
    assorted.cpython-310.pyc
    board_state.cpython-310.pyc
    pieces.cpython-310.pyc
    test_board_state.cpython-310.pyc
    test_fast_board_state.cpython-310.pyc
    test_fast_pieces.cpython-310.pyc
    test_fast_vector.cpython-310.pyc
    test_pieces.cpython-310.pyc
    test_vector.cpython-310.pyc
    vector.cpython-310.pyc
```

```
        __init__.cpython-310.pyc

chess_game
    console_chess.py
    game.py
    game_web.py
    test_fast_game.py
    __init__.py

test_data
    └── test_save_and_restore
        saved_game.game

__pycache__
    console_chess.cpython-310.pyc
    game.cpython-310.pyc
    game_web.cpython-310.pyc
    game_with_difficulty.cpython-310.pyc
    test_fast_game.cpython-310.pyc
    __init__.cpython-310.pyc

database
    create_database.py
    database.db
    handle_games.py
    models.py
    __init__.py

__pycache__
    create_database.cpython-310.pyc
    handle_games.cpython-310.pyc
    models.cpython-310.pyc
    move_model.cpython-310.pyc
    move_model_schema.cpython-310.pyc
    __init__.cpython-310.pyc

move_engine
    broken_minimax_parallel.py
    cache_managers.py
    minimax.py
    minimax_parallel.py
    parallel_minimax_testing.py
    test_slow_timed_minimax_engine.py
    __init__.py

    test_reports
        └── test_depth_2_vs_depth_1
            test_1.csv
```

```
test_1.games
└── test_depth_3_vs_depth_2
    ├── test_1.csv
    └── test_1.games

__pycache__
    cache_managers.cpython-310.pyc
    minimax.cpython-310.pyc
    minimax_parallel.cpython-310.pyc
    test_minimax.cpython-310.pyc
    test_slow_minimax.cpython-310.pyc
    test_slow_timed_minimax_engine.cpython-310.pyc
    test_slow_time_minimax.cpython-310.pyc
    test_time_minimax.cpython-310.pyc
    __init__.cpython-310.pyc

schemas
└── cache_item_schema.py
    socket_schemas.py
    __init__.py

__pycache__
    cache_item_schema.cpython-310.pyc
    move_schema.cpython-310.pyc
    socket_schemas.cpython-310.pyc
    __init__.cpython-310.pyc

test_reports
└── test_bots_by_depth
    ├── test_depth_2_vs_depth_1.csv
    └── test_depth_3_vs_depth_2.csv

    test_bots_by_time
        test_10s_timed_bot_vs_2s_timed_bot.csv
        test_10s_timed_bot_vs_5s_timed_bot.csv
        test_12s_timed_bot_vs_2s_timed_bot.csv
        test_13s_timed_bot_vs_5s_timed_bot.csv
        test_14.0s_timed_bot_vs_10s_timed_bot.csv
        test_15s_timed_bot_vs_5s_timed_bot.csv
        test_17s_timed_bot_vs_2s_timed_bot.csv
        test_20s_timed_bot_vs_10s_timed_bot.csv
        test_20s_timed_bot_vs_5s_timed_bot.csv
        test_21.0s_timed_bot_vs_15s_timed_bot.csv
        test_28.0s_timed_bot_vs_20s_timed_bot.csv
        test_30s_timed_bot_vs_15s_timed_bot.csv
        test_40s_timed_bot_vs_10s_timed_bot.csv
```

```
test_40s_timed_bot_vs_20s_timed_bot.csv
test_60s_timed_bot_vs_30s_timed_bot.csv
test_6s_timed_bot_vs_2s_timed_bot.csv
test_7s_timed_bot_vs_2s_timed_bot.csv
test_9s_timed_bot_vs_5s_timed_bot.csv

test_depth_vs_randomtron
    test_depth_1_num_1.csv
    test_depth_1_num_2.csv
    test_depth_1_num_3.csv
    test_depth_2_num_1.csv
    test_depth_2_num_2.csv
    test_depth_2_num_3.csv

website
    flask_server.py
    secret_key.key
    __init__.py

static
    chess_class.js
    favicon.ico
    initial_game_data.js
    initial_game_data.json
    main.js
    style.css
    vector.js

templates
    chess_game.html

__pycache__
    flask_server.cpython-310.pyc
    __init__.cpython-310.pyc
```

This has allowed each python module to be self-contained like a class. All objects from all files within the folder can be accessed internally. The `__init__.py` files dictate which objects will be exported as part of the module.

I will now explain each module in turn and any changes that were made.

Assorted Module:

This module contained general functions, variables and objects that were small and used throughout the whole program.

Here is the `__init__.py` file for the module which decides which objects are exported by the module:

```
# this file decides that components from within this folder are exported
# as part of the assorted module
from .general import ARBITRARILY_LARGE_VALUE, cache_decorator, dev_print
from .safe_hash import safe_hash
from .chess_exceptions import TimeOutError, InvalidMove, NotUserTurn,
NotComputerTurn, UnexplainedErroneousMinimaxResultError
```

And is the code that for these objects:

General.py

```
# this file is just a file of short assorted constants and functions that
# are general in use
# It only contains small functions as I have tried to group large, similar
# functions logically in there own file

# from functools import lru_cache
# cache_decorator = lru_cache(maxsize=10000)
def cache_decorator(func): return func

# this is used in the static evaluation and minimax process. It is used to
# represent infinity in a way that still allows comparrison
ARBITRARILY_LARGE_VALUE = 1_000_000

# this function is relatively redundant but allows for print statements in
# debugging
# in later iteration this may be replaced with logging.
# it is useful as it allows for DEBUG print statements without needing to
# remove them when finished
DEBUG = False
def dev_print(*args, **kwargs):
    if DEBUG:
        print(*args, **kwargs)
```

In the above file:

- The dev print method was rarely used as I found it easier to comment out print statements that were used in development
- The ARBITRARILY_LARGE_VALUE variable is used to represent infinity. This is needed to provide static evaluations of board states that are checkmate or to set the initial values for alpha and beta in the move engine.
- The cache decorator was applied to various function within the Chess Engine module. You can see that ultimately I set the decorator to do nothing as the use of the decorator made to program noticeable slower. This was not at all the expected outcome.

chess_exceptions.py

```
# this error is used to cause the timed minimax function
# raising and then catching this error allows for the algorithm to be
# self-interrupting
class TimeOutError(Exception):
    pass

# this is an exception that allows for the game data to be bound to it
# this allows for the relevant chess game that caused the error to be
# examined afterwards
# it is a normal exception except the constructor has been modified to
# save the game data as a property
class __ChessExceptionTemplate__(Exception):
    def __init__(self, *args, **kwargs) -> None:
        # none if key not present
        self.game = kwargs.pop("game", None)

    super().__init__(*args, **kwargs)

# these are custom exceptions.
# these are used primarily by the game class
# they contain no logic but have distinct types allowing for targeted
error handling
class InvalidMove(__ChessExceptionTemplate__):
    pass

class NotUserTurn(__ChessExceptionTemplate__):
    pass

class NotComputerTurn(__ChessExceptionTemplate__):
    pass

class UnexplainedErroneousMinimaxResultError(__ChessExceptionTemplate__):
    pass
```

The above exceptions could hold data about a chess game but generally their functionality and logic wasn't used. They were used as specific names expressions that could be raised and caught using error handling in the Game Manager and Move Engine Modules.

```
from hashlib import sha256
```

```
# this hash function produces an integer hash of a python object
# it is necessary as the in built hash function produces different hashes
# for the same object,
# this function always produces the same hash for a given object, this
# allows the hash to be stored in a database for later use
def safe_hash(item):
    # convert item (usually tuple) to string
    encoded_string = repr(item).encode("utf-8")
    # produce a hexadecimal string hash of the object
    hex_hash = sha256(encoded_string).hexdigest()
    # convert this to an integer
    # int_hash = int(hex_hash, 16)
    # return int_hash
    return hex_hash
```

This hash function that uses sha256 was needed as the python hash function produces a smaller hash. In addition the value that the hash function produces for the same input changes as the program is stopped and rerun. This means that if I want to store the hash of a python object in a persistent database, I will need to use the above hash function.

Chess Functions Module (Chess Engine):

Within this module there have been minimal changes. This module was completed and tested in prototype 2.

The main changes were:

- A hash function was used to hash the board state for use in a minimax cache database entry
- The create random board state function was moved to the board state file (to avoid it being repeated elsewhere)

```
# this function produces a random board state
# this is done by selecting and then implementing a legal move at random
# up to so many moves
# it is used in unittest
def random_board_state(moves: int):
    board_state = Board_State()
    # keep trying to get a non over board state
    while True:
        try:
            # iteratively pick a move at random and implement it until the
            # right number of move is reached
            for _ in range(moves):
                legal_moves = list(board_state.generate_legal_moves())
                assert len(legal_moves) > 0
```

```

        random_move = random_choice(legal_moves)
        board_state = board_state.make_move(*random_move)

    except AssertionError:
        # if no legal moves (over) then try again
        continue
    else:
        return board_state

```

The board state module had the following `__init__.py` file that decided what functions it exports:

```

# this file decided which function and classes from within this folder
should be exported as part of the chess functions module.

from .board_state import Board_State, random_board_state
from .vector import Vector
from .pieces import PIECE_TYPES, Piece, Pawn, Knight, Bishop, Rook, King,
Queen

```

Other changes that were made were due to failed tests. These will be explained later.

Move Engine:

This module is responsible for executing an efficient minimax search to determine the computer's move. I have heavily modified the code to add various optimisations.

Here is the file `minimax.py`:

```

# import local modules
# cannot import game as causes circular import, if necessary put in same
file
from board_state import Board_State
from assorted import ARBITRARILY_LARGE_VALUE
from vector import Vector

# my minimax function takes as arguments:
# Board_State, is_maximiser, alpha and beta (used for pruning) and
check_extra_depth (produces better outcome but slower)
# it returns
# score, child, move

def minimax(board_state: Board_State, is_maximizer: bool, depth, alpha,
beta, check_extra_depth=True):
    # sourcery skip: low-code-quality, remove-unnecessary-else, swap-if-
else-branches
    # assume white is maximizer

```

```

# when calling, if give appropriate max min arg

# base case
# if over or depth==0 return static evaluation
over, _ = board_state.is_game_over_for_next_to_go()
if depth == 0 or over:
    # special recursive case 1
    # examine terminal nodes that are check to depth 2 (variable
depth)

        # to avoid goose chaises, extra resources are allowed if check not
already explored
        if board_state.color_in_check() and check_extra_depth and not
over:
            # print(f"checking board state {hash(board_state)} at
additional depth due to check")
            return minimax(
                board_state=board_state,
                is_maximizer=is_maximizer,
                depth=2,
                alpha=alpha,
                beta=beta,
                check_extra_depth=False
            )
        else:
            # static eval works for game over to
            return board_state.static_evaluation(), None, None

    # define variables used to return more than just score (move and
child)
    best_child_game_state: Board_State | None = None
    best_move_vector: Vector | None = None

    # function yields move ordered by how favorable they are (low depth
minimax approximation)
    def gen_ordered_child_game_states():
        # this function does a low depth minimax recursive call (special
recursive case 2) to give a move a score
        def approx_score_move(move):
            child_game_state = board_state.make_move(*move)

            return minimax(
                board_state=child_game_state,
                depth=depth-2,
                is_maximizer=not is_maximizer,
                alpha=alpha,
                beta=beta,
                check_extra_depth=False
            )

        moves = [approx_score_move(m) for m in moves]
        moves.sort(key=lambda m: m[0], reverse=True)
        return moves

```

```
)[0]
# print(f"approx_score_move(move={move!r})  -> {result!r}")

# if depth is 1 or less just yield moves from legal moves
if depth <= 1 :
    yield from board_state.generate_legal_moves()
# else sort them
else:
    # sort best to worse
    # sort ascending order if minimizer, descending if maximizer
    yield from sorted(
        board_state.generate_legal_moves(),
        key=approx_score_move,
        reverse=is_maximizer
    )

if is_maximizer:
    # set max to -infinity
    maximum_evaluation = (-1)*ARBITRARILY_LARGE_VALUE

    # iterate through moves and resulting game states
    for position_vector, movement_vector in
gen_ordered_child_game_states():
        child_game_state =
board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)

        # evaluate each one
        # general recursive case 1
        evaluation, _, _ = minimax(
            board_state=child_game_state,
            is_maximizer=not is_maximizer,
            depth=depth-1,
            alpha=alpha,
            beta=beta,
            check_extra_depth=check_extra_depth
        )

        # update alpha and max evaluation
        if evaluation > maximum_evaluation:
            maximum_evaluation = evaluation
            best_child_game_state = child_game_state
            best_move_vector = (position_vector, movement_vector)
            alpha = max(alpha, evaluation)
```

```

# where possible, prune
if beta <= alpha:
    # print("Pruning!")
    break
# once out of loop, return result
return maximum_evaluation, best_child_game_state, best_move_vector

else:
    minimum_evaluation = ARBITRARILY_LARGE_VALUE

    for position_vector, movement_vector in
gen_ordered_child_game_states():
        child_game_state =
board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)
        evaluation, _, _ = minimax(
            board_state=child_game_state,
            is_maximizer=not is_maximizer,
            depth=depth-1,
            alpha=alpha,
            beta=beta,
            check_extra_depth=check_extra_depth
        )

        if evaluation < minimum_evaluation:
            minimum_evaluation = evaluation
            best_child_game_state = child_game_state
            best_move_vector = (position_vector, movement_vector)
            beta = min(beta, evaluation)

        if beta <= alpha:
            # print("Pruning!")
            break
    return minimum_evaluation, best_child_game_state, best_move_vector

```

I have implemented decomposed the large minimax function into sub function within a class. Some parameters are provided directly to the minimax method within the class while others that configure the move engine (e.g. is cache allowed) are instead parameters of the objects constructor. This has helped reduce the parameters to the minimax method which helps avoid unnecessary complexity.

To use the minimax algorithm, the object is initialised and then called like a function. Calling the object in this way runs the `__call__` method which runs the `minimax_first_call` method. This method is run only the first time the minimax function is called (not on any recursive calls). The first call handler function is responsible for starting up the cache manager object and using the validator function to check the minimax output if enables. The cache handler objects allow for minimax data to be cached to improve the functions efficiency. Some of the objects has a start up and cool down sequence (e.g. opening and then disposing of a database session). The use of a context manager vie

the “with” command calls the `__enter__()` method on the cache manager object, then completed the minimax search and then calls the `__exit__()` method on the object.

The method names minimax if responsible for the bulk of the logic behind the minimax algorithm in this implementation although some large blocks of logic have been extracted into separate private methods.

The core principles behind the algorithm remain the same. It has a base case to stop recurring and evaluate a terminal node. A minimax search of a given `Board_State` object at a depth of N then uses the recursive case. It iterates through the legal moves of the board and then recursively evaluates the child nodes with a depth of $n-1$. It then updates its pruning variables and its best variables. Once it stops iterating it returns the best move and score.

In the above algorithm, the base case has been abstracted out into another function called `pseudo_base_case()`. This method will normally return a static evaluation of the board (base case) but if check has been encountered, it can search to an additional depth using a hidden recursive case. This optimisation should improve the quality of the result.

Minimax method also extracts out the logic of getting the legal moves and child board states to another method called `generate_move_child`. This method uses depth $N-2$ searches (another hidden recursive case) to estimate the score of each of the child nodes before they are searched. This allows it to sort the child nodes in order from best to worst. This should improve the frequency by which parts of the tree can be pruned. In addition, if a depth $N-1$ search has already been completed and cached, then this sorting will not cost any additional computations.

The only other change is that the minimax method checks the cache first to see if it can retrieve the result and skip the rest of the function. Then if this wasn’t possible, at the end of the function, the search result is added to the cache. Recursive sub-searches should also be able to retrieve and save results to and from the cache. This means that even if the cache doesn’t already contain the whole search, it may still contain part of it. This will save computations and improve average time complexity at the cost of a worse space complexity.

I then created a series objects that were designed to allow me to easily cache the minimax search results. Each object has the same interface meaning that they can be used in the same way. All logic around how it stores the results is self-contained and abstracted.

Cache_managers.py

```
# import from external modules
import os
import json
from time import perf_counter

# import local modules
from assorted import safe_hash
from chess_functions import Board_State, Vector
from database import create_session, end_session, persistent_DB_engine,
volatile_RAM_engine, Minimax_Cache_Item
```

```
from schemas import minimax_cache_item_schema

# create an blank object with no logic that can be used with a context
manager
class Blank_Context():
    def __enter__(self, *args, **kwargs):
        # def __exit__(self, exception_type, exception_value, traceback):
        pass

    def __exit__(self, *args, **kwargs):
        pass

# ram cache doesn't need access to a context manager so it uses the blank
one
class RAM_cache(Blank_Context):
    # cache is stored in a local dictionary property in the ram cache
object
    def __init__(self) -> None:
        self.memoization_cache = dict()

    def cache_size(self):
        return len(self.memoization_cache)

    # the result of a minimax call is added to the cache
    def add_to_cache(self, board_state: Board_State, depth, score, move):
        # moves are stored as tuples of integers
        def serialize_move(move):
            if move is None:
                return None

            return (
                (move[0].i, move[0].j),
                (move[1].i, move[1].j)
            )

        # ensure that the cache exists in the memoization_cache property
        assert self.memoization_cache is not None

        # can use a board_state as a key as it is hashable
        # the board state hash is used as the key within the dictionary
        # which acts as a hash table)
        board_state_hash = board_state.database_hash()

        # print(f"CALL: add_to_cache(self, board_state={hash(board_state)}"
        depth={depth}, score={score}, move={move})")
        # print(f"self.memoization_cache.get(board_state_hash) is"
None      -->     {self.memoization_cache.get(board_state_hash) is None}")
```

```
# decide if the cache should be updated based on whether the
record already existed and if so, its depth
    if self.memoization_cache.get(board_state_hash) is None:
        needs_update = True
    else:
        best_depth_in_cache =
self.memoization_cache[board_state_hash]["depth"]
            # print(f"best_depth_in_cache < depth    --
> {best_depth_in_cache} < {depth}    ---> {best_depth_in_cache <
depth}")
        needs_update = best_depth_in_cache < depth

    # print(f"Cache: {self.memoization_cache}")

    # if need an update then add the record (lesser depth cache
overwritten)
    if needs_update:
        new_data_item = {
            "depth": depth,
            "score": score,
            "move": serialize_move(move)
        }

        self.memoization_cache[board_state_hash] = new_data_item
        # if depth >= 1:
        # print(f"New data {board_state_hash} added to
cache: {new_data_item}")
            # print(f"new_cache_size: {self.cache_size()}")
        # print(f"Cache needed updating: {needs_update}")

# this function retrieves an item from cache
def search_cache(self, board_state, depth):
    # move is converted from tuple of integers back to vectors when
deserialized
    def deserialize_move(serialised_move):
        if serialised_move is None:
            return None

        return (
            Vector(*serialised_move[0]),
            Vector(*serialised_move[1])
        )

    assert self.memoization_cache is not None

    # get the hash of the board state
    # board_state_hash = hash(board_state)
    board_state_hash = board_state.database_hash()
```

```
# if not in cache, return none
    if self.memoization_cache.get(board_state_hash) is None:
        # if depth >= 2:
            # print(f"item {board_state_hash} not in cache
(depth={depth})")
        return None
    # else check the depth is adequate and return if appropriate
    else:
        best_depth_in_cache =
self.memoization_cache[board_state_hash]["depth"]
        record_useful = best_depth_in_cache >= depth
        if record_useful:
            # if depth >= 2:
                # print(f"RAM_Cache used for
{board_state_hash}: {self.memoization_cache[board_state_hash]}")

            data_item =
self.memoization_cache[board_state_hash].copy()
            data_item["move"] = deserialize_move(data_item["move"])

        return data_item
    else:
        # if depth >= 2:
            # print(f"cache not useful for item
{board_state_hash}: {self.memoization_cache[board_state_hash]}, required
depth is {depth}")
            # print(f"cache searched but not used")
        return None

# this function creates a persistent cache in a json file
class JSON_Cache(RAM_cache):
    # when the object is created, load the contents of the json file into
the memoization cache variable
    def __init__(self, file_path=None) -> None:
        # print("Initializing json cache")
        if file_path is not None:
            self.file_path = file_path
        else:
            self.file_path = r"./database/minimax_cache.json"

        # default
        self.memoization_cache = {}

    # try get data from file
    if os.path.exists(self.file_path):
        with open(self.file_path, "r") as file:
            content = file.read()
```

```
try:
    assert content != "", "Json file was blank / empty"
    cache_data = json.loads(content)

    # fix that in json, board state hash key is sting
    new_items = map(
        lambda item: (int(item[0]), item[1]),
        cache_data.items()
    )
    cache_data = dict(tuple(new_items))

    self.memoization_cache = cache_data
except Exception as e:
    # print(e)
    # print("Failed to load json cache data so using blank
cache")
    pass

# inherited ram cache method add to and retrieve cache from the
memoization cache variable

# write the memoization cache dictionary to a json file
def save(self):
    print(f"Saving cache (size={self.cache_size()}) to json file")
    with open(self.file_path, "w") as file:
        file.write(
            json.dumps(
                self.memoization_cache
            )
        )

# do this when the cache manager is closed by the context manager
def __exit__(self, *args, **kwargs):
    self.save()

# this json cache doesn't save automatically
class JSON_Cache_Manual_Save(JSON_Cache):
    def __exit__(self, *args, **kwargs):
        pass

# database cache connects to the database to store minimax cached results
# it uses a database in RAM for small cache and a persistent database for
searches at a greater depth
class DB_Cache():
    # by default, depth 0 uses ram cache and greater depth uses persistent
cache
    def __init__(self, min_DB_depth=1):
```

```
# set the min depth parameter as a property and then define other
properties with starting values
    self.min_DB_depth = min_DB_depth

    self._RAM_cache = None
    self._DB_session = None
    self.engaged = False
# a cached depth 3 call can be used if a depth 2 call is needed
    self.allow_greater_depth = True

    self.time_delta_DB = 0
    self.time_delta_schemas = 0
    self.checks_to_cache = 0
    self.retrieved_item = 0
    self.items_added = 0

def __enter__(self, *args, **kwargs):
    # when the context manager is used and the boot up sequence runs
(this method)
        # a database session is created (using the database module)

        # assert all(e is None for e in (self._RAM_cache,
self._DB_session))
        assert not self.engaged, "Must not be engaged (entered) already to
enter"
        self.engaged = True

        # these variables keep track of various data points while the
cache is used within a context manager
        self.time_delta_DB = 0
        self.time_delta_schemas = 0
        self.checks_to_cache = 0
        self.retrieved_item = 0
        self.items_added = 0

        # this allows for the cache manager to be entered multiple times
on accident without making multiple sessions
        # self._RAM_cache = RAM_cache()
        # self._persistent_DB_session =
scoped_session(sessionmaker(bind=persistent_DB_engine))
        # self._volatile_RAM_session =
scoped_session(sessionmaker(bind=volatile_RAM_engine))
        self._persistent_DB_session = create_session(persistent_DB_engine)
        self._volatile_RAM_session = create_session(volatile_RAM_engine)

def __exit__(self, *args, **kwargs):
    # when the context manager runs the close function of the cache
manager (this method)
```

```
# the session objects are closed and discarded and tracker
variables are set back to 0

    assert self.engaged, "Must be engaged (entered) to exit"
    self.engaged = False

    end_session(self._persistent_DB_session)
    self._persistent_DB_session = None

    end_session(self._volatile_RAM_session)
    self._volatile_RAM_session = None

    # print(f"Cache session lasted {round(self.time_delta_DB,
2)}+{round(self.time_delta_schemas, 2)} sec (DB + schema time):
{self.checks_to_cache} checks made, {self.retrieved_item} items retrieved,
{self.items_added} items added")

    self.time_delta_DB = 0
    self.time_delta_schemas = 0
    self.checks_to_cache = 0
    self.retrieved_item = 0
    self.items_added = 0

def _search_DB_cache(self, session, board_state, depth):
    # this function hashes the board state and searches the database
for cache corresponding to that hash

    # board_state_hash = str(hash(board_state))
    board_state_hash = board_state.database_hash()

    start = perf_counter()

    # cache must have a matching hash and a sufficient depth
    # ORM allows for queries to be written in a pythonic way
    if self.allow_greater_depth:
        result = session.query(Minimax_Cache_Item)\.
            .where(
                Minimax_Cache_Item.board_state_hash ==
board_state_hash
            )\.
            .where(
                Minimax_Cache_Item.depth >= depth
            )\.
            .first()
    else:
        result = session.query(Minimax_Cache_Item)\.
            .where(
```

```
        Minimax_Cache_Item.board_state_hash ==  
board_state_hash  
    )\\  
    .where(  
        Minimax_Cache_Item.depth == depth  
    )\\  
    .first()  
  
    stop = perf_counter()  
    self.time_delta_schemas += stop - start  
  
    # if not item found then return none  
    if result is None:  
        return None  
  
    start = perf_counter()  
  
    # use the schema object to deserialize the cached item before  
returning it.  
    result = minimax_cache_item_schema.dump(result)  
  
    stop = perf_counter()  
    self.time_delta_schemas += stop - start  
  
    return result  
  
def _add_to_DB_cache(self, session, board_state: Board_State, score,  
depth, move):  
    # this function adds an item to the cache  
  
    start = perf_counter()  
  
    # board state is hashed  
    # print("adding item to database cache")  
    # board_state_hash = str(hash(board_state))  
    board_state_hash = board_state.database_hash()  
  
    # if depth >= 2:  
    #     print(f"Adding item {board_state_hash} at depth {depth} to  
cache")  
  
    # new database entry item created with schema  
    new_item = minimax_cache_item_schema.load(  
        dict(  
            board_state_hash=board_state_hash,  
            score=score,  
            depth=depth,  
            move=move
```

```
        )
    )

    stop = perf_counter()
    self.time_delta_schemas += stop - start

    start = perf_counter()

    # other searches at a lesses depth are deleted
    # (assumes cache already checked and the other cache is worse)
    session.query(Minimax_Cache_Item) \
        .where(
            Minimax_Cache_Item.board_state_hash == board_state_hash
    ) \
        .delete()

    # print(f"move    -->  {move!r}")

    # print(f"New item:  {new_item!r}")

    # the item is added to the database and committed
    session.add(new_item)
    session.commit()

    stop = perf_counter()
    self.time_delta_DB += stop - start

def search_cache(self, board_state: Board_State, depth):
    # this function searches the cache
    assert self.engaged, "Context manager must be used"

    self.checks_to_cache += 1

    # not sure why depth sometimes is a single length tuple containing
    an int
    # here is a quick fix
    if isinstance(depth, tuple):
        if len(depth) == 1:
            depth = depth[0]

    # print(f"(depth, self.min_DB_depth)      -->      {(depth,
    self.min_DB_depth)}")

    # decide which session to use based on depth
    if depth < self.min_DB_depth:
        session = self._volatile_RAM_session
```

```
        # return self._RAM_cache.search_cache(board_state=board_state,
depth=depth)
    else:
        session = self._persistent_DB_session
        # return self._search_DB_cache(board_state=board_state,
depth=depth)

        # search the cache with this session and then return the result
        result = self._search_DB_cache(session=session,
board_state=board_state, depth=depth)

        if result is not None:
            self.retrieved_item += 1

    return result


def add_to_cache(self, board_state: Board_State, score, depth, move):
    # this function adds to the cache
    # this function assumes that no move valuable cache already exists
    # (it overwrites all other cache)
    assert self.engaged, "Context manager must be used"

    # this is used to tackle any bugs elsewhere in the program
    # it ensures that no invalid cache items are added to the database
    def absolute(x): return x if x >= 0 else 0-x
    if absolute(score) > 1_000_000 or (move is None and depth > 0):
        # don't add erroneous data to cache
        return None


    # decides that session to use based on depth
    if depth < self.min_DB_depth:
        session = self._volatile_RAM_session
        # self._RAM_cache.add_to_cache(board_state=board_state,
score=score, depth=depth, move=move)
    else:
        session = self._persistent_DB_session
        # self._add_to_DB_cache(board_state=board_state, score=score,
depth=depth, move=move)

        # the caches item is added to the database
        self._add_to_DB_cache(session=session, board_state=board_state,
score=score, depth=depth, move=move)

    self.items_added += 1
```

```

# the hash function describes only the data that makes this cache
object unique
def __hash__(self) -> int:
    hash(safe_hash(
        (
            "DB_Cache",
            # self._RAM_cache,
            # self._DB_session,
            self.min_DB_depth,
            self.engaged,
            self.allow_greater_depth,
            # self.time_delta_DB
            # self.time_delta_schemas
            # self.checks_to_cache
            # self.retrieved_item
            # self.items_added
        )
    )))

```

Here we can see that I created a series of increasingly complex cache managers. I wanted to have a form of persistent cache as this would allow the program to learn. I decided to use a database rather than a json file for this as a database is more scalable and has better look up times. In addition, a database can be accessed simultaneously by many concurrent processes whereas a json file cannot.

I then created some more efficient versions of the matrix algorithm that inherited from the Move Engine Class.

Minimax_parallel.py

```

# import external and local modules

# import sys
import multiprocessing
from time import perf_counter

from .minimax import Move_Engine, is_time_expired

from .cache_managers import RAM_cache, DB_Cache

from chess_functions import Board_State
from assorted import ARBITRARILY_LARGE_VALUE, TimeOutError,
UnexplainedErroneousMinimaxResultError

# import time
def get_cores():

```

```
return multiprocessing.cpu_count()
# return multiprocessing.cpu_count()//2

# Classes with the name of JOB are essentially functions that represent a
# general task that can be completed concurrently
# they are classes as functions cannot be easily piped between threads

# this object represents a job that a concurrent worker should complete,
# I used an object as a function cannot be sent between workers
class Minimax_Sub_Job:
    # contractor: configure bot by setting these parameters as properties
    def __init__(self, board_state, depth, args, kwargs, additional_depth,
cache_allowed, max_time=None) -> None:

        # create a cache manager as necessary
        self.cache_allowed = cache_allowed
        self.cache_manager = DB_Cache() if cache_allowed else None

        # create a move engine as necessary
        self.move_engine: Move_Engine = Move_Engine(
            cache_manager=self.cache_manager,
            cache_allowed=cache_allowed,
            additional_depth=additional_depth,
        )
        # assert isinstance(self.move_engine, Move_Engine),
f"Minimax_Sub_Job.__init__      self.move_engine of unexpected type
{type(self.move_engine)}\n{self.move_engine!r}"

        # assign parameters as properties
        self.board_state: Board_State = board_state

        self.depth = depth
        self.max_time = max_time

        # these are other erroneous argument that could be passed to the
minimax call
        self.args = args
        self.kwargs = kwargs

    def __call__(self, legal_moves_sub_array):
        # perform part of a minimax search using a sub set of legal moves

        # print(f"running minimax job with legal moves sub array:
(hash={hash(str(legal_moves_sub_array))})")
```

```

        # print(f"STARTING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
        # print(legal_moves_sub_array)d_state,
        # result = self.move_engine.minimax(
        # assert isinstance(self.move_engine, Move_Engine),
f"Minimax_Sub_Job.__call__      self.move_engine of unexpected type
{type(self.move_engine)}\n{self.move_engine!r}"

        if self.cache_allowed:
            with self.cache_manager:
                result = self.move_engine.minimax(
                    board_state=self.board_state,
                    depth=self.depth,
                    legal_moves_to_examine=legal_moves_sub_array,
                    # is_time_expired = self.is_time_expired,
                    max_time=self.max_time,
                    *self.args,
                    **self.kwargs
                )
        else:
            result = self.move_engine.minimax(
                board_state=self.board_state,
                depth=self.depth,
                legal_moves_to_examine=legal_moves_sub_array,
                # is_time_expired = self.is_time_expired,
                max_time=self.max_time,
                *self.args,
                **self.kwargs
            )

        # print(f"FINISHING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
        # print(f"Minimax sub job finished
(hash={hash(str(legal_moves_sub_array))}))")
        return result

# this bot is responsible for performing a low depth search of a move to
be used to presort the moves
class Presort_Moves_Sub_Job:
    # construct object
    def __init__(self, depth, board_state: Board_State, cache_allowed,
max_time=None) -> None:
        # def __init__(self, depth, board_state: Board_State, move_engine:
Move_Engine, cache_allowed, cache_manager, max_time=None) -> None:

```

```
# cache manager parameter may not be needed if the cache manager
object is also bound to the move engine

    # assign parameters as properties
    self.depth = depth
    self.board_state = board_state

    # create a cache manager as necessary
    self.cache_allowed = cache_allowed
    self.cache_manager = DB_Cache() if cache_allowed else None

    # create a move engine object
    self.move_engine = Move_Engine(
        presort_moves=True,
        additional_depth=0,
        cache_allowed=self.cache_allowed,
        cache_manager=self.cache_manager,
        # cache_allowed=self.cache_allowed,
        # cache_manager=self.cache_manager,
    )

    # self.cache_allowed = cache_allowed
    # self.cache_manager = cache_manager

    self.is_time_expired = is_time_expired
    self.max_time = max_time

def __call__(self, move):
    # cache manager already built into the minimax move engine first
    call method

        # print(f"Presort job using cache
manager: {self.cache_manager!r}")
        # with self.cache_manager:
        #     child = self.board_state.make_move(*move)
        #     child_score, _ = self.move_engine.minimax_first_call(
        #         board_state = child,
        #         depth = self.depth
        #     )
        #     # print("Closing cache")
        #     result = (child_score, move)
    # return result

    # self interrupting
    if is_time_expired(self.max_time):
        raise TimeOutError()
```

```

        # score a child, use minimax first call to add cache manager being
        opened twice
        # no need to say no parallel as a basic move engine is used
        child = self.board_state.make_move(*move)
        if self.cache_allowed:
            with self.cache_manager:
                # child_score, _ = self.move_engine.minimax_first_call(
                child_score, _ = self.move_engine.minimax(
                    board_state=child,
                    depth=self.depth,
                    # is_time_expired = self.is_time_expired,
                    max_time=self.max_time,
                )
        else:
            # child_score, _ = self.move_engine.minimax_first_call(
            child_score, _ = self.move_engine.minimax(
                board_state=child,
                depth=self.depth,
                # is_time_expired = self.is_time_expired,
                max_time=self.max_time,
            )

    return (child_score, move)

# this is the main class to complete a minimax search in parallel.
# some methods inherited from parent class while some are overwritten
class Parallel_Move_Engine(Move_Engine):
    # construct the engine using parameters
    def __init__(self, cache_manager: RAM_cache | None = DB_Cache(),
    cache_allowed: bool = True, parallel=True, min_parallel_depth=2,
    workers=get_cores(), **kwargs) -> None:
        self.parallel = parallel
        if parallel and cache_allowed:
            assert isinstance(cache_manager, DB_Cache), "Only DB cache can
            be used in parallel"
            self.min_parallel_depth = min_parallel_depth
            self.workers = workers

        # use parent class construction
        super().__init__(cache_manager=cache_manager,
        cache_allowed=cache_allowed, **kwargs)

    def should_use_parallel(self, board_state: Board_State, depth):
        return depth >= self.min_parallel_depth and self.parallel

```

```
# this function breaks up the legal moves array into many sub arrays
# that have a similar distribution of good and bad moves
def break_up_legal_moves_to_segments(self, legal_moves: list,
number_sub_arrays):
    # print("Legal moves:")
    # print(legal_moves)

    legal_move_sub_arrays = [[] for _ in range(number_sub_arrays)]

    # repeatedly add next best move to the end of the sub array
    # iterating through the sub arrays to give each one a similar
distribution of move quality
    i = 0
    while legal_moves:
        legal_move_sub_arrays[i].append(
            legal_moves.pop(0)
        )
        i = (i+1) % number_sub_arrays

    # reverse sub arrays so in order of best to worst
    legal_move_sub_arrays = list(map(
        lambda e: list(reversed(e)),
        legal_move_sub_arrays
    ))

    # print("Broken down")
    # for sub_array in legal_move_sub_arrays:
    #     print(sub_array)

    return legal_move_sub_arrays

# this function runs the presort moves process in parallel

def generate_move_child_in_parallel(self, board_state: Board_State,
depth: int, is_maximizer, give_child=True, max_time=None):
    # print("generate_move_child_in_parallel called")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # explore_depth = max(depth-3, 0)
    explore_depth = max(depth-2, 0)

    if (not self.parallel):
        yield from super().generate_move_child(
            board_state=board_state,
```

```
        depth=depth,
        is_maximizer=is_maximizer,
        give_child=give_child,
        # is_time_expired=is_time_expired
        max_time=max_time
    )
else:
    # if can do presort concurrently
    # # print("getting legal moves")
    legal_moves = list(board_state.generate_legal_moves())

    # print("setting up job")
    # create job
    job = Presort_Moves_Sub_Job(
        depth=explode_depth,
        board_state=board_state,
        cache_allowed = self.cache_allowed,
        max_time=max_time,
    )

    # concurrently map the job onto the iterable of legal moves
    # print("generate_move_child_in_parallel: Using
multiprocessing pool to presort in parallel")
    with multiprocessing.Pool(self.workers) as pool:
        moves_and_scores = pool.map(
            func=job,
            iterable=legal_moves
        )

    # print("generate_move_child_in_parallel: finished with
multiprocessing pool")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # sort by score
    moves_and_scores = sorted(
        moves_and_scores,
        key=lambda triplet: triplet[0],
        reverse=is_maximizer
    )
    # use map to get rid of score

def move_and_blank_child(score_and_move):
    _, move = score_and_move
    return (move, None)
```

```
        moves_scores_children = map(move_and_blank_child,
moves_and_scores)

        # print("children sorted by score")

        yield from moves_scores_children

# this function performs a minimax search in parallel

def parallel_minimax(self, board_state, depth, max_time=None, *args,
**kwargs):
    # print("Call parallel_minimax")
    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # print("parallel_minimax, checking cache")
    # if cache can be used then check if this search has already been
complete
    if self.cache_allowed:
        result = self.cache_manager.search_cache(
            board_state=board_state,
            depth=depth
        )
        if result is not None:
            # print("no call needed as cache is sufficient")
            return result["score"], result["move"]

    # print("Parallel minimax called")

    # assert isinstance(depth, int)

    # assert depth >= 2, "depth must be greater or equal to 2 to do
parallelization"

    is_maximizer =
self.color_maximizer_key.get(board_state.next_to_go)

    # print("started getting moves_and_child_sorted")
    # print("getting legal moves, presorted")

    # generate presorted moves iterable
    # print("parallel_minimax: calling
generate_move_child_in_parallel")
    moves_sorted = list(self.generate_move_child_in_parallel(
        board_state=board_state,
        depth=depth,
```

```
        is_maximizer=is_maximizer,
        give_child=False,
        # is_time_expired=is_time_expired,
        max_time=max_time,
    ))

    # print("parallel_minimax: finished
generate_move_child_in_parallel")

    # print("finished getting moves_and_child_sorted")
    # print("moves_sorted: ")
    # print(moves_sorted)

    # break up these pre sorted legal moves into segments for each sub
job to work on
    # print("parallel_minimax, breaking up moves into sub arrays")
    legal_move_sub_arrays = self.break_up_legal_moves_to_segments(
        legal_moves=moves_sorted,
        number_sub_arrays=self.workers
    )

    # print("parallel_minimax: finished
break_up_legal_moves_to_segments")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # print("legal moves sub arrays generated")

    # print("Printing legal moves sub arrays")
    # print(legal_move_sub_arrays)

    # print("Defining job for workers")

    # print("parallel_minimax, defining sub job")
    # use these legal moves to do many simultaneous minimax operations

    # I updated the sub job classes so that I don't need to pass a
cache manager object to them
    # this was causing issues as the database connection that the
database cache manager contained couldn't be piped between workers

    # # create a job function
    # job = Minimax_Sub_Job(
    #     # move_engine=self,
    #     move_engine=Move_Engine(
```

```
#           cache_allowed=False,
#           cache_manager=None,
#           # cache_manager=cache_manager,
#           # cache_allowed=cache_allowed,
#
#           # is this the miracle fix?
#           additional_depth=self.additional_depth,
#
#           use_validator=self.use_validator,
#
#       ),
#       board_state=board_state,
#       depth=depth,
#
#       # is_time_expired = is_time_expired,
#       max_time=max_time,
#
#       args=args,
#       kwargs=kwargs,
#   )
#
# create a job function
#
job = Minimax_Sub_Job(
    board_state=board_state,
    depth=depth,
    max_time=max_time,
    additional_depth=self.additional_depth,
    cache_allowed=self.cache_allowed,
    args=args,
    kwargs=kwargs,
)
#
# print(f"Using pool to complete jobs in parallel (pool size = {cores})")
#
# map this job function onto the legal moves sub array in parallel
# print("parallel_minimax: Using multiprocessing pool")
with multiprocessing.Pool(self.workers) as pool:
    minimax_sub_job_results = pool.map(
        func=job,
        iterable=legal_move_sub_arrays
    )
#
# self interrupting
if is_time_expired(max_time):
    raise TimeOutError()
```

```
# print("parallel_minimax: multiprocessing finished")

# set a best move and score variable with starting values
best_move = None
if is_maximizer:
    best_score = 0-(ARBITRARILY_LARGE_VALUE + 1)
else:
    best_score = ARBITRARILY_LARGE_VALUE + 1

# select best move
for result in minimax_sub_job_results:
    score, move = result

    # if this move is better then update the best move and score
variables
    if (is_maximizer and score > best_score) or (not is_maximizer
and score < best_score):
        best_score = score
        best_move = move

# add the result to the cache
if self.cache_allowed:
    self.cache_manager.add_to_cache(
        board_state=board_state,
        depth=depth,
        move=best_move,
        score=best_score
    )

# print("finished getting best outcome")

return best_score, best_move

# this function is able to perform handle the initial non recursive
call for a parallel minimax search

def minimax_first_call_parallel(self, board_state: Board_State, depth,
max_time=None, *args, **kwargs):
    # print(f"CALL  minimax_first_call_parallel")
    # print(f"CALL minimax_first_call_parallel(self,
board_state={hash(board_state)}, depth={depth}, *args, **kwargs)")
    # assert isinstance(depth, int)

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()
```

```
    should_use_parallel =
self.should_use_parallel(board_state=board_state, depth=depth)
    # print(f"minimax_first_call_parallel:  should_use_parallel  --> {should_use_parallel}")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # I find the below code clearer, but unfortunately it won't work
    # as the functions cannot be pickled and sent/piped between the workers

    # if should_use_parallel:
    #     def chosen_minimax_function():
    #         return self.parallel_minimax(
    #             board_state=board_state,
    #             depth=depth,
    #             cache_allowed=self.cache_allowed,
    #             cache_manager=self.cache_manager,
    #             # is_time_expired = is_time_expired,
    #             max_time=max_time,
    #             *args,
    #             **kwargs
    #         )
    # else:
    #     def chosen_minimax_function():
    #         return self.minimax(
    #             board_state=board_state,
    #             depth=depth,
    #             max_time=max_time,
    #             *args, **kwargs
    #         )

    # if self.cache_allowed:
    #     with self.cache_manager:
    #         result = chosen_minimax_function()

    # else:
    #     result = chosen_minimax_function()

    # the below code simply make 2 decisions,
    # should parallel minimax be used or vanilla minimax?
    # should cache be used?
    # it is unfortunate that the above function based solution didn't
work as be bellow code features lots of repetition
if should_use_parallel:
    if self.cache_allowed:
        with self.cache_manager:
```

```
        result = self.parallel_minimax(
            board_state=board_state,
            depth=depth,
            max_time=max_time,
            *args,
            **kwargs
        )
    else:
        result = self.parallel_minimax(
            board_state=board_state,
            depth=depth,
            max_time=max_time,
            *args,
            **kwargs
        )

else:
    if self.cache_allowed:
        with self.cache_manager:
            result = self.minimax(
                board_state=board_state,
                depth=depth,
                max_time=max_time,
                *args, **kwargs
            )
    else:
        result = self.minimax(
            board_state=board_state,
            depth=depth,
            max_time=max_time,
            *args, **kwargs
        )

if self.use_validator:
    self.validator(result=result, board_state=board_state)

return result

# when the object is called, use minimax_first_call_parallel to handle
the call
def __call__(self, *args, **kwargs):
    return self.minimax_first_call_parallel(*args, **kwargs)

# this move engine class represents be best configuration of the move
engine
# it also updates some functions form parallel minimax (improves upon it)
class Move_Engine_Prime(Parallel_Move_Engine):
    def __init__(self) -> None:
```

```
super().__init__(
    parallel=True,
    cache_allowed=True,
    cache_manager=DB_Cache(min_DB_depth=1),
    additional_depth=1,
    # additional_depth=0,
    presort_moves=True,
    color_maximizer_key={"W": True, "B": False},
    use_validator=False,
    # workers = get_cores()
    workers=5
)
self.depth = 2

# when called, use standard depth if no depth parameter provided
def __call__(self, board_state: Board_State, depth=None, *args,
**kwargs):
    if depth is None:
        return super().__call__(board_state=board_state,
depth=self.depth, *args, **kwargs)
    else:
        return super().__call__(board_state=board_state, depth=depth,
*args, **kwargs)

    # this function returns a boolean to decide if parallel processing is
needed.
    # it takes the board state as a parameter as the evaluation could
depend on the legal moves to analyze
def should_use_parallel(self, board_state: Board_State, depth):
    # if not allowed to use parallel or hte game is over then return
false
    if not self.parallel:
        return False

    over, _ = board_state.is_game_over_for_next_to_go()
    if over:
        return False

    # otherwise begin by estimating the depth that the decision tree
will be searched to
    if board_state.check_encountered:
        likely_depth = depth + 0.5 * self.additional_depth
    else:
        likely_depth = depth

    # then estimate the branching factor by looking at the average
legal moves available to both players
    board_state_a = board_state
```

```
legal_moves_a = list(board_state.generate_legal_moves())
some_legal_move_a = legal_moves_a[0]

board_state_b = board_state_a.make_move(*some_legal_move_a)

branching_factor_a = len(legal_moves_a)
branching_factor_b =
len(list(board_state_b.generate_legal_moves()))

likely_branching_factor = 1/2 * (branching_factor_a +
branching_factor_b)

# use an exponential formula to create and estimate for static
evaluations
estimated_static_eval = likely_branching_factor ** likely_depth
# print({"estimated_static_eval": estimated_static_eval})

# if the number of evaluations exceeds that of depth 2 on the
starting positions, then use parallel
return (estimated_static_eval >= 400)

def break_up_legal_moves_to_segments(self, legal_moves: list,
number_sub_arrays):
    # use the original break up legal moves function as it is better
    return super().break_up_legal_moves_to_segments(legal_moves,
number_sub_arrays)

    # the bellow code was experimental and turned out to not improve
performance,
    # it divided the scored legal move up differently into sub arrays
    # it put the best few in the first sub array and then the next
best few in the next and so on
    # this was worse than the previous function. this is because
creating a distribution or good and bad from best to worst improved the
pruning

# legal_move_sub_arrays = [[] for _ in range(number_sub_arrays)]

# number_legal_moves = len(legal_moves)
# number_workers = self.workers

# moves_per_worker = number_legal_moves // number_workers

# workers_with_extra = number_legal_moves % number_workers
# workers_without_extra = number_workers - workers_with_extra
```

```

        # moves_per_sub_array = [moves_per_worker] * workers_without_extra
+ [moves_per_worker+1] * workers_with_extra
        # assert sum(moves_per_sub_array) == number_legal_moves

        # # print(moves_per_sub_array)

        # # >>> x = [1,2,3,4,5]
        # # >>> x[:2]
        # # [1, 2]
        # # >>> x[2:]
        # # [3, 4, 5]

        # for worker_index in range(number_workers):
        #     moves_in_array = moves_per_sub_array[worker_index]
        #     legal_move_sub_arrays[worker_index] =
legal_moves[:moves_in_array]
        #     legal_moves = legal_moves[moves_in_array:]

        # # print("Broken down")
        # # for sub_array in legal_move_sub_arrays:
        # #     print(sub_array)

        # yield from legal_move_sub_arrays

# this minimax algorithm explores the decision tree to a given depth
before stopping
class Move_Engine_Timed(Move_Engine_Prime):
    # explore for a certain amount of time
    def timed_call(self, board_state: Board_State, time):
        # so that internal jobs can access the time used up indicator
        # self.max_time = perf_counter() + time
        # take 3 seconds to close threads

        time_delta_allowed = time
        # time_delta_allowed = max(time - 3, 0)
        # time_delta_allowed = max(time - 10, 0)
        # time_delta_allowed = max(time - 4, 0.5)
        # print({"time_delta_allowed": time_delta_allowed})

        # stop at max time
        self.max_time = perf_counter() + time_delta_allowed

    def time_used_up():
        return perf_counter() >= self.max_time

    # need minimum of depth 1 as depth 0 doesn't give a move
    # ensure that even if time = 0, at least a depth 1 search is
completed

```

```
# deepest_result =
self.minimax_first_call(board_state=board_state, depth=1)
    deepest_result = self.minimax_first_call(board_state=board_state,
depth=1, variable_depth=0)
    depth = 2

    def absolute(x): return x if x >= 0 else 0-x
    # while True:

        # while there is time left, keep searching to a greater depth
        while not time_used_up():
            # print(f"Iterating again: time_used_up() --> {time_used_up()}")
            try:
                # deepest_result =
                self.minimax_first_call_parallel(board_state=board_state, depth=depth,
max_time=self.max_time, variable_depth=0)
                    score, move = result

                    # if the move or score are invalid then raise the
appropriate error
                    if absolute(score) == 1_000_001 or move is None:
                        raise UnexplainedErroneousMinimaxResultError()

                except UnexplainedErroneousMinimaxResultError:
                    # I don't know the cause of the error but I can catch it
here and prevent it causing issues and producing unexpected behaviour
                    break
                except TimeOutError:
                    # print(f"Breaking: time_used_up() --> {time_used_up()}")
                    break
            else:
                # print(f"Completes depth={depth} so incrementing depth")
                # print(f"Result at depth {depth}: {deepest_result}")

                    # if neither the time is used up or an erroneous result
was produced, keep searching at a greater depth
                    deepest_result = result
                    depth += 1

    # return the best move
move, _ = deepest_result
```

```
assert move is not None

# print(f"Returning result at greatest depth={depth}")
return deepest_result, depth

def __call__(self, board_state: Board_State, time, *args, **kwargs):
    # use the timed_call handler function

    assert not board_state.is_game_over_for_next_to_go()[0]

    # start = perf_counter()
    deepest_result, _ = self.timed_call(board_state=board_state,
time=time)
    # end = perf_counter()

    # time_delta = end - start
    # print(f"timed minimax set to {time} seconds actually took
{round(time_delta, 3)} seconds")
    return deepest_result

# the below function was used to take benchmarks of the timed minimax
algorithm in order to gauge how accurately it stuck to its time limit
def check_timed():
    board_state = Board_State()
    move_engine = Move_Engine_Timed()
    time = 0

    # for _ in range(6):
    #     start = perf_counter()
    #     result, depth = move_engine(board_state, 1)
    #     time += perf_counter() - start
    #     print(f"At total time {time} sec:
depth={depth};  result={result}")

    # for _ in range(6):
    #     start = perf_counter()
    #     result, depth = move_engine(board_state, 2)
    #     time += perf_counter() - start
    #     print(f"At total time {time} sec:
depth={depth};  result={result}")

    # for _ in range(6):
    #     start = perf_counter()
    #     result, depth = move_engine(board_state, 5)
    #     time += perf_counter() - start
    #     print(f"At total time {time} sec:
depth={depth};  result={result}")
```

```

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 10)
#     time += perf_counter() - start
#     print(f"At total time {time} sec:
depth={depth};  result={result}")

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 15)
#     time += perf_counter() - start
#     print(f"At total time {time} sec:
depth={depth};  result={result}")

while True:
    start = perf_counter()
    result, depth = move_engine(board_state, 20)
    time += perf_counter() - start
    print(f"At total time {time} sec:
depth={depth};  result={result}")

if __name__ == "__main__":
    check_timed()

```

The above code contains 3 main classes and 3 sub classes.

Parallel_Move_Engine inherits from the vanilla move engine and adds concurrency

Move_Engine_Prime inherits from Parallel_Move_Engine and adds the optimal configurations by default. It also improves on some of the methods.

Move_Engine_Timed inherits from Move_Engine_Prime and includes the ability to search the minimax tree for a given amount of time before stopping and using an insurance search.

Parallel_Move_Engine uses the multiprocessing library to make use of all the cores on my computer when performing a computationally intensive minimax search.

It parallelises 2 parts of the vanilla minimax algorithm. It firstly evaluates the child nodes for pre-sorting in parallel. Then it breaks up the array of child nodes into sub arrays (breaks the tree into sub trees). It then searches each of these sub arrays of child nodes (sub trees) in parallel. This greatly improved time efficiency as the minimax function was limited by my processor speed.

To achieve this, the main obstacle was ensuring that all my objects were pickleable. This allowed them to be sent to each individual worker. Because of this, I had to use callable objects rather than functions. This is why I created Presort_Moves_Sub_Job and Minimax_Sub_Job. Each of these represents a function that will be executed many times on many different parameters concurrently by different workers running on different cores of my CPU.

Minimax prime improved the method that decided whether or not to use the parallel minimax function. I decide to use 400+ static evaluations as the cut off point as I found that evaluating the starting positions at depth 2 took a similar amount of time in parallel as using the vanilla minimax algorithm (10-12 seconds).

I decided to use 5 workers as my laptops CPU has 4 physical cores and 8 logical cores. I found that this meant that between 4 and 8 task could be completed concurrently. I realised that when I used 4 workers, not all the CPU's power was used. As a result I decided to use 5 workers. I wanted to keep the number of workers as low as possible while fully using the whole CPU. This was in an effort to maximise the size of the legal move sub arrays that each worker was processing. This allowed for increased opportunities for pruning.

Task Manager									
Processes		File	Options	View					
Name	S...	100% CPU	77% Memory	3% Disk	0% Network	16% GPU	GPU engine	Power usage	Power usage t...
Python		15.9%	1.1%	0 MB/s	0 Mbps	0%		Very high	Low
Python		15.7%	1.1%	0 MB/s	0 Mbps	0%		Very high	Low
Python		15.6%	1.1%	0 MB/s	0 Mbps	0%		Very high	Low
Python		15.6%	1.1%	0 MB/s	0 Mbps	0%		Very high	Low
Python		15.3%	1.1%	0 MB/s	0 Mbps	0%		Very high	Low
> Search		6.2%	3.6%	0.1 MB/s	0.2 Mbps	1.7%	GPU 0 - 3D	Moderate	Very low
> Antimalware Service Ex...		5.0%	5.4%	0.1 MB/s	0 Mbps	0%		Moderate	Very low
Desktop Window Mana...		2.2%	2.6%	0.1 MB/s	0 Mbps	13.0%	GPU 0 - 3D	Low	Very low
> Microsoft Windows Sea...		1.5%	1.4%	0.1 MB/s	0 Mbps	0%		Low	Very low
System		1.1%	0.1%	0.1 MB/s	0 Mbps	0.1%	GPU 0 - 3D	Very low	Very low
> Start		0.9%	0.7%	0.1 MB/s	0 Mbps	0%	GPU 0 - 3D	Very low	Very low
Windows Explorer		0.9%	1.4%	0.1 MB/s	0 Mbps	0%		Very low	Very low

Here you can see that a parallel minimax search of depth 2 of the starting positions has created 5 independent python instances in the task manager. These instances are using almost all the CPU's power and are each performing a Minimax_Sub_Job function on a different segment of the legal moves.

The Move_Engine_Timed was my final iteration of the minimax algorithm. It features all the optimisations that I planned in my design section. This means that it is efficient and controllable as I can now more easily limit how much it can explore the decision tree. This allow me to make more difficulty setting and to make the algorithm more consistent and predictable in how long it takes.

Game Manager

I added to the main Game class from prototype 2. The main edition was code to more easily represent a console game of chess. This was used in the console chess program as well as in unit tests.

```
# import other modules
from chess_functions import Board_State, Vector
from move_engine import Move_Engine_Timed
```

```
from assorted import NotUserTurn, NotComputerTurn, InvalidMove, safe_hash

from time import perf_counter
import os

# this function wipes the console to allow the updated board to be
displayed
def clear_console():
    # https://stackoverflow.com/questions/517970/how-to-clear-the-
interpreter-console
    os.system('cls')

# this function allows a function's execution time to be measured
def time_function(function):
    start = perf_counter()
    result = function()
    end = perf_counter()
    time_delta = end - start
    return result, time_delta

# the game class is used to keep track of a chess game between a user and
the computer
class Game(object):
    # constructor for game object
    def __init__(self, time=10, user_color="W", echo=False) -> None:
        # based on user's color, determine color key
        self.player_color_key = {
            "W": 1 if user_color=="W" else -1,
            "B": -1 if user_color=="W" else 1
        }
        # set depth property from parameters
        self.time = time
        self.echo = echo

        # set attributes for game at start
        self.board_state = Board_State()
        self.move_counter = 0
        self.move_engine = Move_Engine_Timed()
        self.game_history_output = ()

    # hash function describes all unique properties of the game as a
unique number
    def __hash__(self):
        return hash(safe_hash((
```

```
        "Game",
        self.player_color_key,
        self.time,
        self.echo,
        self.board_state,
        self.move_counter,
        self.move_engine,
        self.game_history_output,
    )))

# this function determines if 2 game objects are equal
def __eq__(self, other: object) -> bool:
    if not isinstance(other, Game):
        return False

    return hash(other) == hash(self)

@staticmethod
def create_row(moving_player, time_delta, move_count, new_utility,
future_utility, move_description, white_pieces_taken, black_pieces_taken,
number_legal_moves) -> str:
    # print("(moving_player, time_delta, move_count, new_utility,
future_utility, move_description, white_pieces_taken, black_pieces_taken,
number_legal_moves)")
    # print((moving_player, time_delta, move_count, new_utility,
future_utility, move_description, white_pieces_taken, black_pieces_taken,
number_legal_moves))

    # this function creates formatted string that represents a row in
    # a table and contains all the data provided as parameters
    if time_delta is None:
        time_delta = ""
    if future_utility is None:
        future_utility = ""
    return f"| {move_count:^14} | {moving_player:^15} | "
    # {new_utility:^15} | {future_utility:^15} | {time_delta:^15} |
    # {number_legal_moves:^15} | {move_description:<40} |
    # {white_pieces_taken:<35} | {black_pieces_taken:<35} |"

def print_game_history(self):
    # this function clears the console and then print out a table that
    # contains the history of the game

    clear_console()
    # https://www.geeksforgeeks.org/string-alignment-in-python-f-string/
    print("Moves History:")
```

```
print()
# print table header
# print(self.create_row('Moving Color', 'Move Count', 'Time Taken
(sec)', 'New Utility', 'Move', 'White Pieces Taken', 'Black Pieces
Taken'))
print(
    self.create_row(
        moving_player="Moving Color",
        time_delta="Time Taken",
        new_utility="New Utility",
        future_utility="Future Utility",
        move_description="Move Description",
        white_pieces_taken="White Pieces Taken",
        black_pieces_taken="Black Pieces Taken",
        move_count="Move Number",
        number_legal_moves="NO legal moves"
    )
)
# print each row
for item in self.game_history_output:
    print(item)
print()
# print the board
self.board_state.print_board()

def make_move(self, move, time_delta, future_utility):
    # if echo, gather some data before the move
    if self.echo:
        color_moving = "White (+)" if self.board_state.next_to_go ==
"W" else "Black (-)"
        # print({"move": move})
        position_vector: Vector = move[0]
        resultant_vector: Vector = move[0] + move[1]
        moving_piece, taken_piece =
self.board_state.get_piece_at_vector(position_vector),
self.board_state.get_piece_at_vector(resultant_vector)
        from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()
        no_legal_moves = self.board_state.number_legal_moves

        # adjust properties that keep track of the game state
        # implement the move on the board state
        self.board_state = self.board_state.make_move(*move)
        # print(f"self.board_state.next_to_go --> {self.board_state.next_to_go}")
        self.move_counter += 1
```

```
# if echo, gather move data, format and create a row and then
print out the move history
    if self.echo:
        # find various data points
        utility_score = self.board_state.static_evaluation()
        move_number = self.move_counter

        if taken_piece is None:
            move_description = f"{moving_piece} moved from
{from_square} to {to_square}"
        else:
            move_description = f"{moving_piece} moved from
{from_square} to {to_square}, taking piece {taken_piece}"

    def pieces_missing_characters(color):
        return list(map(
            str,
            self.board_state.generate_pieces_taken_by_color(color)
        ))

        white_pieces_taken = " ".join(pieces_missing_characters("W"))
        # white_pieces_taken = "(No pieces taken)" if
white_pieces_taken == "" else white_pieces_taken
        black_pieces_taken = " ".join(pieces_missing_characters("B"))
        # black_pieces_taken = "(No pieces taken)" if
black_pieces_taken == "" else black_pieces_taken

        if time_delta is not None:
            time_delta = f"{{round(time_delta, 2)}} sec"
        # else:
        #     time_delta = ""

        # create a row
        new_row = self.create_row(
            moving_player=color_moving,
            time_delta=time_delta,
            move_count=move_number,
            new_utility=utility_score,
            move_description=move_description,
            white_pieces_taken=white_pieces_taken,
            black_pieces_taken=black_pieces_taken,
            number_legal_moves=no_legal_moves,
            future_utility=future_utility,
        )

        # add the row to the history
        self.game_history_output =
tuple(list(self.game_history_output) + [new_row])
```

```
# create a new row to show check if appropriate
if self.board_state.color_in_check():
    check_msg = f"CHECK: {self.board_state.next_to_go} in
check"
    new_row = self.create_row(
        moving_player="-",
        time_delta="-",
        move_count="-",
        new_utility="-",
        move_description=check_msg,
        white_pieces_taken="-",
        black_pieces_taken="-",
        number_legal_moves="-",
        future_utility="-",
    )
    self.game_history_output =
tuple(list(self.game_history_output) + [new_row])

# create a new row to show game over if appropriate
over, winner = self.board_state.is_game_over_for_next_to_go()
if over:
    how = "Stalemate" if winner is None else "Checkmate"
    winning_color = "White" if winner == "W" else "B"

    if how == "Stalemate":
        if
self.board_state.is_3_board_repeats_in_game_history():
            game_over_msg = "Stalemate: both players draw (3
repeats of the board state)"
        else:
            game_over_msg = "Stalemate: both players draw (no
legal moves and not in check)"
    else:
        game_over_msg = f"Checkmate: {winning_color} wins"

    new_row = self.create_row(
        moving_player="-",
        time_delta="-",
        move_count="-",
        new_utility="-",
        move_description=game_over_msg,
        white_pieces_taken="-",
        black_pieces_taken="-",
        number_legal_moves="-",
        future_utility="-",
    )
```

```
        self.game_history_output =
tuple(list(self.game_history_output) + [new_row])

        # print the game history
        self.print_game_history()

# this function validates if the user's move is allowed and if so,
makes it
def implement_user_move(self, from_square, to_square, time_delta=None,
estimated_utility=None) -> None:
    # check that the user is allowed to move
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    if which_player_next_to_go != 1:
        raise NotUserTurn(game=self)

    # unpack move into vector form
    # invalid square syntaxes will cause a value error here
    try:
        position_vector = Vector.construct_from_square(from_square)
        movement_vector = Vector.construct_from_square(to_square) -
position_vector
    except Exception:
        raise ValueError("Square's not in valid format")

    move = (position_vector, movement_vector)

    # if the move is not in the set of legal moves, raise and
appropriate exception
    if move not in self.board_state.generate_legal_moves():
        raise InvalidMove(game=self)

    self.make_move(move, time_delta, estimated_utility)

    return move, self.board_state.static_evaluation()

# this function determines if the game is over and if so, what is the
nature of the outcome
def check_game_over(self) -> list[bool, str, str]:
    # returns: over: bool, winning_player: (1/-1), classification: str

    # determine if board state is over for next player
    over, winner = self.board_state.is_game_over_for_next_to_go()
    # switch case statement to determine the appropriate values to be
    returned in each case
    match (over, winner):
        case False, _:
```

```

        victory_classification = None
        winning_player = None
    case True, None:
        if self.board_state.is_3_board_repeats_in_game_history():
            victory_classification = "stalemate board repeat"
        else:
            victory_classification = "stalemate no legal moves"
            winning_player = None
    case True, winner:
        victory_classification = "checkmate"
        winning_player = self.player_color_key[winner]
# return appropriate values
return over, winning_player, victory_classification

# this function determines and implements the computer move
def implement_computer_move(self, best_move_function=None):
    # for use with testing bots, a best move function can be provided
for use, but minimax if the default

    # get next to go player (1/-1)
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    # check that is it the computer's turn
    if which_player_next_to_go != -1:
        raise NotComputerTurn()
        # raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

    # if no function provided, default to minimax
    if best_move_function is None:
        def best_move_function(self):
            return self.move_engine(
                board_state = self.board_state,
                # depth is based of difficulty of game based on depth
parameter
                time = self.time,
            )

        # otherwise use provided function,
        # the provided function should take game as an argument and then
return data in the same format as the minimax function
        result, time_delta = time_function(
            lambda: best_move_function(self)
        )
        # print({"result": result})
        score, best_move = result
        # best_move, score = result

```

```

# print({"best_move": best_move})
# print({"score": score})

assert best_move is not None

self.make_move(best_move, time_delta, score)

# incase is it wanted for a print out ect, return move and score
return best_move, score

```

Below is an example of the additional functionality being used to show what is happening in a unit test of depth 3 bot playing a depth 2 bot

Moves History:									
Move Number	Moving Color	New Utility	Future Utility	Time Taken	No legal moves	Move Description	White Pieces Taken	Black Pieces Taken	
1	White (+)	50	0	17.64 sec	20	WN moved from B1 to C3			
2	Black (-)	0	0	108.59 sec	20	BN moved from B8 to C6			
3	White (+)	50	0	24.89 sec	22	WN moved from G1 to F3			
4	Black (-)	0	-20	195.9 sec	22	BN moved from G8 to F6			
5	White (+)	20	-20	26.37 sec	24	WP moved from D2 to D3			
8 BR · BB BQ BK BB · BR 7 BP BP BP BP BP BP 6 · BN · BN · BN 5 · · · · · · 4 · · · · · · 3 · · WN WP · MN · 2 WP WP WP · WP WP WP 1 WR · WB WQ WK WB · WR (A B C D E F G H)									

Moves History:									
Move Number	Moving Color	New Utility	Future Utility	Time Taken	No legal moves	Move Description	White Pieces Taken	Black Pieces Taken	
1	Black (-)	0	0	17.64 sec	20	WN moved from B1 to C3			
2	Black (-)	0	0	108.59 sec	20	BN moved from B8 to C6			
3	White (+)	50	0	24.89 sec	22	WN moved from G1 to F3			
4	Black (-)	0	-20	195.9 sec	22	BN moved from G8 to F6			
5	White (+)	20	-20	26.37 sec	24	WP moved from D2 to D3			
6	Black (-)	-20	-40	318.03 sec	24	BP moved from E7 to E5			
7	White (+)	0	-40	51.39 sec	31	WP moved from E2 to E3			
8	Black (-)	5	-80	169.44 sec	30	BN moved from F6 to G4			
9	White (+)	25	-80	50.4 sec	29	WP moved from F2 to F4			
10	Black (-)	15	-90	1169.82 sec	34	BQ moved from D8 to F6			
11	White (+)	15	-90	61.40 sec	31	WK moved from E1 to E2			
12	Black (-)	5	-325	1471.92 sec	40	BN moved from C6 to D4			
-	-	-	-	-	-	CHECK: W in check	-	-	
13	White (+)	5	-325	9.76 sec	3	WK moved from E2 to D2			
8 BR · BB BQ BK BB · BR 7 BP BP BP BP BP BP 6 · BN · BN · BN 5 · · · · · · 4 · · · · · · 3 · · WN WP · MN · 2 WP WP WP · WP WP WP 1 WR · WB WQ WK WB · WR (A B C D E F G H)									

Moves History:									
Move Number	Moving Color	New Utility	Future Utility	Time Taken	No legal moves	Move Description	White Pieces Taken	Black Pieces Taken	
1	White (-)	50	0	17.64 sec	20	WN moved from B1 to C3			
2	Black (-)	0	0	108.59 sec	20	BN moved from B8 to C6			
3	White (+)	50	0	24.89 sec	22	WN moved from G1 to F3			
4	Black (-)	0	-20	195.9 sec	22	BN moved from G8 to F6			
5	White (+)	20	-20	26.37 sec	24	WP moved from D2 to D3			
6	Black (-)	-20	-40	318.03 sec	24	BP moved from E7 to E5			
7	White (+)	0	-40	51.39 sec	31	WP moved from E2 to E3			
8	Black (-)	5	-80	169.44 sec	30	BN moved from F6 to G4			
9	White (+)	25	-80	50.4 sec	29	WP moved from F2 to F4			
10	Black (-)	15	-90	1169.82 sec	34	BQ moved from D8 to F6			
11	White (+)	15	-90	61.40 sec	31	WK moved from E1 to E2			
12	Black (-)	5	-325	1471.92 sec	40	BN moved from C6 to D4			
-	-	-	-	-	-	CHECK: W in check	-	-	
13	White (+)	5	-325	9.76 sec	3	WK moved from E2 to D2			
8 BR · BB BQ BK BB · BR 7 BP BP BP BP BP BP 6 · BN · BN · BN 5 · · · · · · 4 · · · · · · 3 · · WN WP · MN · 2 WP WP WP · WP WP WP 1 WR · WB WQ WK WB · WR (A B C D E F G H)									

Moves History:									
Move Number	Moving Color	New Utility	Future Utility	Time Taken	No legal moves	Move Description	White Pieces Taken	Black Pieces Taken	
1	White (+)	50	0	17.64 sec	20	WN moved from B1 to G5			
2	Black (-)	0	0	100.59 sec	20	BN moved from B6 to C6			
3	White (+)	50	0	24.14 sec	22	WN moved from E1 to F3			
4	Black (-)	0	-20	195.9 sec	22	BN moved from G8 to F6			
5	White (+)	20	-20	26.37 sec	24	MP moved from D2 to D3			
6	Black (-)	-20	-40	318.03 sec	24	BP moved from E7 to E5			
7	White (+)	0	-40	51.39 sec	31	MP moved from E2 to E3			
8	Black (-)	5	-80	620.44 sec	30	BN moved from F6 to G4			
9	White (+)	25	-80	-	29	MP moved from E3 to E2			
10	Black (-)	15	-98	1160.82 sec	34	MP moved from B8 to F6			
11	White (+)	15	-98	61.49 sec	31	WK moved from E1 to E2			
12	Black (-)	5	-325	1471.92 sec	40	BN moved from C6 to D4			
-	-	-	-	-	-	CHECK: W in check	-	-	
13	White (+)	5	-325	9.76 sec	3	MK moved from E2 to D2			
14	Black (-)	-100	-550	524.41 sec	47	BN moved from G4 to F2, taking piece MP	MP		
15	White (+)	-100	-550	66.57 sec	26	WQ moved from B8 to E1	MP		
16	Black (-)	-420	-560	739.1 sec	49	BN moved from D4 to F3, taking piece WN	WP WN		
17	White (+)	-110	-560	15.39 sec	3	CHECK: W in check	-	-	
						MP moved from G2 to F3, taking piece BN	WP WN		
8 BR . BB . BK BB . BR									
7 BP BP BP BP . BP BP									
6 BQ .									
5 MP . .									
4 MP . .									
3 WN WP . MP .									
2 WP WP WP WP . BN . WP									
1 WR . . WB . WQ WB . WR									
(A B C D E F G H)									

I then created a new class called Game_Website. It is responsible to keep track of a chess game that is going in on the client side on my webpage user interface. It originally inherited from the Game class but then I began overwriting so many methods that it was complicating the class. As a result the Game_Website class is similar but separate from the game class.

Game_web.py

```
# this class is a variant of the game class that is to be used in a web
game of chess
# it originally inherited from the game class but then it became too
different so I no longer made it inherit from the game class

# from .game import Game

from assorted import safe_hash
from chess_functions import Board_State, Vector
from move_engine import Move_Engine_Timed
from schemas import serialize_piece, serialize_move

# import other modules
from chess_functions import Board_State, Vector
from move_engine import Move_Engine_Timed
from assorted import NotUserTurn, NotComputerTurn, InvalidMove, safe_hash

from time import perf_counter
import os

# # this function clears the console
# def clear_console():
#     # https://stackoverflow.com/questions/517970/how-to-clear-the-
#     interpreter-console
#     os.system('cls')
```

```
# # this function allows for a functions performance to be measured
# def time_function(function):
#     start = perf_counter()
#     result = function()
#     end = perf_counter()
#     time_delta = end - start
#     return result, time_delta

# the game class is used to keep track of a chess game between a user and
the computer
class Game_Website(object):
    # # it keeps track of:
    # # the player's color's
    # player_color_key: dict
    # # the difficulty or depth of the game
    # depth: int
    # # the current board state
    # board_state: Board_State
    # # the number of moves so far
    # move_counter: int

    # move_engine: Move_Engine

    # game_history_output: tuple[str]

    # constructor for game object
    # def __init__(self, time, user_color="W", echo=True) -> None:
def __init__(self, difficulty="medium", user_color="W") -> None:
    self.difficulty = difficulty.strip().lower()

    # based on user's color, determine color key
    self.player_color_key = {
        "W": 1 if user_color == "W" else -1,
        "B": -1 if user_color == "W" else 1
    }
    # set depth property from parameters

    # set attributes for game at start
    # start with a blank board
    self.board_state = Board_State()
    self.move_counter = 0
    # use the timed move engine to find moves
    self.move_engine = Move_Engine_Timed()
    self.move_history_output = []

    # the time function is treated as a property to emulate how it was
before when it was a property (before difficulty)
    @property
```

```
def time(self):
    # return 0.5
    match self.difficulty:
        case "really_easy": return 1
        case "easy": return 2
        case "medium": return 5
        case "hard": return 10
        case "really_hard": return 15
        case "extreme": return 30
        case "legendary": return 60

    raise ValueError(f"Difficulty {self.difficulty} not recognised")

# this functions adds a move to the move history,
# it doesn't make the move
def add_move_to_history(self, move):
    # should be called pre move
    # color_moving = "White" if self.board_state.next_to_go == "W"
    else "Black"
    # print({"move": move})

    # unpack move and determine squares and relevant pieces
    position_vector: Vector = move[0]
    resultant_vector: Vector = move[0] + move[1]
    moving_piece, taken_piece =
    self.board_state.get_piece_at_vector(position_vector),
    self.board_state.get_piece_at_vector(resultant_vector)
        from_square, to_square = position_vector.to_square(),
    resultant_vector.to_square()

    # increment move count
    self.move_counter += 1

    # generate string move description using this data
    if taken_piece is None:
        # move_description = f"Move {self.move_counter}:
    {color_moving} moved {moving_piece} from {from_square} to {to_square}"
            # move_description = f"{self.move_counter}: {color_moving}
    moved {moving_piece} from {from_square} to {to_square}"
        move_description = f"{self.move_counter}: {moving_piece} from
    {from_square} to {to_square}"
    else:
        # move_description = f"Move {self.move_counter}:
    {color_moving} moved {moving_piece} from {from_square} to {to_square},
    taking piece {taken_piece}"
            # move_description = f"{self.move_counter}: {color_moving}
    moved {moving_piece} from {from_square} to {to_square}, taking piece
    {taken_piece}"
```

```
        move_description = f"{self.move_counter}: {moving_piece} from {from_square} to {to_square}, taking piece {taken_piece}"  
  
        # append this string to the move history list  
        self.move_history_output.append(move_description)  
  
    # this function makes a move on the board state and logs this in the game history  
    def make_move(self, move):  
        assert not self.board_state.is_game_over_for_next_to_go()[0],  
"Cannot make move if game is over"  
        assert move is not None, "Cannot move as move parameter not valid"  
  
        self.add_move_to_history(move)  
  
        # adjust properties that keep track of the game state  
        self.board_state = self.board_state.make_move(*move)  
  
  
    # this function validates if the user's move is allowed and if so, makes it  
    # input form javascript is a move in terms of vectors  
    def implement_user_move(self, move: list[Vector]) -> None:  
        assert not self.board_state.is_game_over_for_next_to_go()[0],  
"Cannot make move if game is over"  
  
        # check that the user is allowed to move  
        which_player_next_to_go = self.player_color_key.get(  
            self.board_state.next_to_go  
        )  
        # check it is the user's go  
        if which_player_next_to_go != 1:  
            raise NotUserTurn(game=self)  
  
        # if the move is not in the set of legal moves, raise and appropriate exception  
        if move not in self.board_state.generate_legal_moves():  
            self.board_state.print_board()  
            print(f"Move {move} not in legal moves")  
            raise InvalidMove(game=self)  
  
        # make the move  
        self.make_move(move)  
  
        # no return data needed  
        # return move, self.board_state.static_evaluation()
```

```
# this function determines if the game is over and if so, what is the
nature of the outcome
def check_game_over(self):
    # returns: over: bool, winning_player: (W / B), classification:
str

    # determine if board state is over for next player
    over, winner = self.board_state.is_game_over_for_next_to_go()
    # switch case statement to determine the appropriate values to be
returned in each case
    match (over, winner):
        case False, _:
            victory_classification = None
            winning_player = None
        case True, None:
            if self.board_state.is_3_board_repeats_in_game_history():
                victory_classification = "stalemate board repeat"
            else:
                victory_classification = "stalemate no legal moves"
                winning_player = None
        case True, winner:
            victory_classification = "checkmate"
            winning_player = self.player_color_key[winner]

    # return appropriate values
    return over, winning_player, victory_classification

# this function is used to determine the AI's move
# the time delta can be explicitly given or it can use the time
provided manually as a parameter
def best_move_function(self, time):
    if time is None:
        time = self.time

    # returns just the move
    return self.move_engine(
        board_state=self.board_state,
        # depth is based of difficulty of game based on depth
parameter
        time=time,
    )[1]

    # this function is used to generate a description of the move that the
computer has completed for the client side GUI
    # it is needed to provide highlighting ect.
    # it doesn't make the computer move
```

```
def generate_computer_move_description(self, move,
previous_board_state):

    # determine various qualities about the move
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector

    moved_piece =
previous_board_state.get_piece_at_vector(position_vector)
    taken_piece =
previous_board_state.get_piece_at_vector(resultant_vector)

    from_square = position_vector.to_square()
    to_square = resultant_vector.to_square()

    # return dictionary of this data ready for json serialization
    # to do this, ensure all objects such as pieces and vectors are
    serialized.

    move_description = {
        "moved_piece": serialize_piece(moved_piece),
        "taken_piece": serialize_piece(taken_piece),
        "from_square": from_square,
        "to_square": to_square,
        "move": serialize_move(move)
    }

    return move_description

# this function completed the computer's move in a given time delta
and returns a description
def implement_computer_move_and_report(self, time=None):
    previous_board_state = self.board_state

    # get next to go player (1/-1)
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    # check that is it the computer's turn
    if which_player_next_to_go != -1:
        raise NotComputerTurn()
        # raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

    # get the computer's move
    best_move = self.best_move_function(time=time)
```

```

        # defensive design to detect issues, ensure a move has been
calculated
        assert best_move is not None

        # implement the move and return a description
        self.make_move(best_move)

        return self.generate_computer_move_description(best_move,
previous_board_state)

def __eq__(self, other: object) -> bool:
    # this method used the hash function to determine if 2 of these
objects are the same
    if not isinstance(other, Game_Website):
        return False

    return hash(self) == hash(other)

def __hash__(self):
    # this function encodes all the data that makes a game unique in a
hash to be stored in a database
    return hash(safe_hash((
        "Game_Website",
        self.player_color_key,
        self.time,
        self.board_state,
        self.move_counter,
        # self.move_engine,
        self.move_history_output,
        self.difficulty
    )))

```

The main changes are:

- It has no functionality to log the game to the console.
- It also produces a different game history array designed for the client side JavaScript to use
- It produces a description of the computer's move to allow the client side JavaScript to display the move properly with highlighting. This description is in the form of a dictionary that includes pieces and vectors involved in the move.
- It uses the timed minimax algorithm to produce the computer move
- It also features different difficulty setting which correspond to a different exploration time by the timed minimax algorithm.
- The resulting object can also be hashed and pickled to binary form in order to be stored in a database.

The chess game (Game Manager) module export the following objects through its `__init__.py` file:

```
from .game import Game
# from .game_with_difficulty import Game_With_Difficulty
from .game_web import Game_Website
```

It exports the Game class and the Game_Website class.

Database module:

The database module is responsible for setting up and managing the database that the save game feature and minimax cache use.

The `__init__.py` file for the database module is unusual as it includes some logic:

```
# this file does 2 things:
# it decides which objects should be exported as part of the database
# module
# it initialise the database (creates a connection and creates all tables)

# objects to export
from .create_database import create_session, end_session, create_engines,
end_engines
from .models import Minimax_Cache_Item, create_tables
from .handle_games import get_saved_game, save_game

# initialise database
engines_dict = create_engines(echo=False)

create_tables(engines_dict)

persistent_DB_engine = engines_dict["persistent_DB_engine"]
volatile_RAM_engine = engines_dict["volatile_RAM_engine"]
```

Its main responsibilities are to:

- Export the relevant functions and classes to allow the rest of the program to access the database
- Create the database connection and all the tables that should exist (if they don't already).

Create_database.py

```
# external modules used
import os
import sqlalchemy as sqla
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.pool import QueuePool
```

```
# path to database file
DB_path = os.path.join(os.getcwd(), 'database', 'database.db')

# this function creates 2 database engines, one in RAM and one is
# secondary storage
def create_engines(echo):
    # echo means, should all the queries be printed out

    # caution with below code, could cause unexpected behavior if not
    # thread safe, further research needed
    # I don't know much about threading in python, I find it confusing due
    # to complexities around the global interpreter lock
    # as a result I had to just try various arguments to create these
    # engines in a way that allowed me to use them with my flask server
    persistent_DB_engine = sqla.create_engine(
        'sqlite:///+' + DB_path,
        echo=echo,
        poolclass=QueuePool,
        connect_args={'check_same_thread': False}
    )

    volatile_RAM_engine = sqla.create_engine(
        "sqlite:///memory:",
        echo=echo,
        poolclass=QueuePool,
        connect_args={'check_same_thread': False}
    )
    return {
        "persistent_DB_engine": persistent_DB_engine,
        "volatile_RAM_engine": volatile_RAM_engine
    }

# this function safely terminates the database connection
def end_engines(engines):
    for engine in engines.values():
        engine.dispose()

# this function creates a session to access the database
def create_session(engine):
    return scoped_session(sessionmaker(bind=engine))

# this function safely ends a session
def end_session(session):
    session.commit()
    session.close()
```

this file contains the relevant functions to create and safely destroy both engine objects (the master connection to the data base) and session objects (a temporary connection used when actively querying that database).

It creates an engine corresponding to both a persistent database in a “database.db” file and a volatile database in RAM.

Models.py

```
# import external modules
import sqlalchemy as sqla
from sqlalchemy.ext.declarative import declarative_base
import pickle

# cannot be used for type hings as it has a high risk of causing a
circular import
# from chess_game import Game_Website

# create a base object with which I will make my database tables (part of
the ORM)
Base = declarative_base()

# create an object and a table (ORM used) to represent a cached minimax
call
class Minimax_Cache_Item(Base):

    # here is the metadata and the columns of the database
    __tablename__ = "Minimax_Cache"

    primary_key = sqla.Column(sqla.Integer, primary_key=True)

    board_state_hash = sqla.Column(sqla.String())
    depth = sqla.Column(sqla.INT())
    score = sqla.Column(sqla.INT())
    # 4 character move string encoded by the 4 digits of the from and
movement vectors
    move = sqla.Column(sqla.String())

    # a single entry can be created by initializing an object with this
class
    def __init__(self, board_state_hash: str, depth, move: str, score:
int):
        # hash board state
        self.board_state_hash = str(board_state_hash)
        self.depth = depth
```

```
        self.move = move
        self.score = score

        # string description of object
        def __repr__(self) -> str:
            return
f"Minimax_Cache_Item(board_state_hash='{self.board_state_hash}', depth={self.depth}, move='{self.move}', score={self.score})"

# this object represents an entry in a database table that saves games for
# later reloading
class Saved_Game(Base):
    # metadata as well as columns defined here
    __tablename__ = "Saved_Games"

    primary_key = sqla.Column(sqla.Integer, primary_key=True)

    # game_hash = sqla.Column(sqla.String())
    cookie_key = sqla.Column(sqla.String())
    raw_game_data = sqla.Column(sqla.BINARY())

    # initializing an object from this class allow a new entry in the
    # database to be made
    def __init__(self, game, cookie_key):
        self.cookie_key = str(cookie_key)
        self.raw_game_data = bytes(
            pickle.dumps(
                game
            )
        )

    # string description of object
    def __repr__(self) -> str:
        return f"Saved_Game(cookie_key='{self.cookie_key}')"

# create indexes on cookie key and board state hash as these are the
# columns that I will use to search for a specific entry
sqla.schema.Index("board_state_hash", Minimax_Cache_Item.board_state_hash)
sqla.schema.Index("cookie_key", Saved_Game.cookie_key)

# for each engine, build these tables
def create_tables(engines_dict):
    for engine in engines_dict.values():
        Base.metadata.create_all(engine, checkfirst=True)
```

This file contains a pair of classes that represent a single database entry within each of my 2 database table respectively. As the classes have been created using the SQL-Alchemy ORM, I can directly interact with the database by interacting with these objects.

After the tables are defined using the schema classes I then create and index for each table to set one of the columns to be a secondary key. I cannot create the tables without a standard incrementing primary key but I intend to use these secondary keys to query the tables.

The Saved_Game table does store binary data as one of the columns. While in principle this can be inefficient, the binary data stored is very small (2-4 kb). This means it can be stored directly in the database without significant scalability issues.

The create_tables function will create the relevant tables in each of the engines, once the engines are created.

handle_games.py

```
# import external modules and local models
import pickle

from .models import Saved_Game

# cannot import for type hint due to high risk of circular imports
# from chess_game import Game_Website

# this method queries the database for a game by cookie id
# and then deserializes the binary data to restore the game
def get_saved_game(cookie_key, session):

    # query database for game by cookie key
    result: Saved_Game = session.query(Saved_Game) \
        .where(
            # Saved_Game.game_hash == str(hash(game_hash))
            Saved_Game.cookie_key == str(cookie_key)
        ) \
        .first()

    # if the result is none (possible if browser has cookie that is
    # invalid) then return none
    if result is None:
        # print(f"get_saved_game game not in database so returning none")
        return None

    # now deserialize the game from the binary data
    game = pickle.loads(
        result.raw_game_data
    )
```

```

# print(f"this game recovered from database under
cookie_key: {cookie_key}")
# game.board_state.print_board()

# print(f"game recovered from database: {hash(game)}")
# return the game
return game

# this function saves a game to the database under a certain cookie ID.
def save_game(game, cookie_key, session):
    # print("saving this game into database")
    # game.board_state.print_board()
    # print(f"saving game in database game with hash {hash(game)} under
cookie_key: {cookie_key}")

    # delete any old games with the same cookie id
    session.query(Saved_Game) \
        .where(
            # Saved_Game.game_hash == str(hash(game_hash))
            Saved_Game.cookie_key == str(cookie_key)
        ) \
        .delete()

    # create a new entry in the database and commit
    session.add(
        Saved_Game(game, cookie_key)
    )
    session.commit()

```

It contains a pair of function that are able to save a Game_Website object to the database as binary and the query and deserialize it to restore the Game_Website object.

I have used the pickle library to convert my python object to and from binary.

We can also see that I can execute SQL queries in a pythonic way thanks to the use of an ORM framework. This adds validation to prevent SQL injection attacks. This is because, whatever the malicious code stored in the cookie, it will not be executed and will be treated as a sting due to the type definition in the Saved_Game class.

Schema Module:

The schema module is responsible for serialisation and deserialisation processes. This is used to serialise python objects' data into a form that can be stored in a database or sent to a client's browser. The python objects can also be restored using the corresponding deserialization functions if needed.

The schema module's `__init__.py` file exports for following functions:

```
from .cache_item_schema import minimax_cache_item_schema
from .socket_schemas import serialize_legal_moves,
serialize_pieces_matrix, deserialize_move, serialize_piece,
serialize_move, deserialize_pieces_matrix
```

Here is `cache_item_schema.py`:

```
# import external and external modules
from marshmallow import Schema, fields, pre_load, pre_dump, post_load,
post_dump

from chess_functions import Vector
from database import Minimax_Cache_Item

# the vector schema describes how to serialize and deserialize a vector
object
# it converts between Vector(i=a, j=b) and [a, b]
class Vector_Schema(Schema):
    i = fields.Integer(required=True)
    j = fields.Integer(required=True)

    @pre_load
    def pre_load(self, vector, **kwargs):
        data = {}
        data["i"] = vector.i
        data["j"] = vector.j
        return data

    @post_dump
    def post_dump(self, data, **kwargs):
        return Vector(
            i=data["i"],
            j=data["j"]
        )

# this move schema is for database cache
# it converts between [Vector(a, b), Vector(c, d)] and [[a,b], [c,d]]
# is used the nested Vector Schema to do this
class Move_Schema(Schema):
    from_vector = fields.Nested(Vector_Schema, required=True)
    movement_vector = fields.Nested(Vector_Schema, required=True)

    @pre_load
```

```
def pre_load(self, vector_tuple, **kwargs):
    # print({"vector_tuple": vector_tuple})
    from_vector, movement_vector = vector_tuple
    data = {}
    data["from_vector"] = from_vector
    data["movement_vector"] = movement_vector
    return data

@post_dump
def post_dump(self, data, **kwargs):
    return (
        data["from_vector"],
        data["movement_vector"]
    )

# this method deserializes a move string into a pair of Vectors stored in
# a dictionary
def get_deserialized_move(move):
    # assert len(move) == 4
    # if move == "None":
    if move == None:
        return None

    move = move[1:-1]
    move = move.split(", ")
    from_vector = Vector(
        i=move[0],
        j=move[1]
    )
    movement_vector = Vector(
        i=move[2],
        j=move[3]
    )
    return dict(
        from_vector=from_vector,
        movement_vector=movement_vector
    )

# this method converts a pair of vectors stored in a dictionary to a
# string representation of the move
def get_serialised_move(move):
    if move is None:
        # return "None"
        return None

    return str((
        move["from_vector"]["i"],
```

```
        move["from_vector"]["j"],
        move["movement_vector"]["i"],
        move["movement_vector"]["j"],
    )))
}

# this class uses the move schema and the above methods to serialize and
deserialize a move so that it can be stored in a database
class Minimax_Cache_Item_Schema(Schema):
    board_state_hash = fields.String(required=True, load_only=True)
    depth = fields.Integer(required=True)
    score = fields.Integer(required=True)
    move = fields.Nested(Move_Schema, required=True, allow_none=True)

    @post_load
    def post_load(self, data, **kwargs):
        # assert isinstance(data["depth"], int)
        # assert isinstance(data["score"], int)
        # print(f"post_load: serializing move from {data['move']} to
{get_serialised_move(data['move'])}")

        return Minimax_Cache_Item(
            board_state_hash=data["board_state_hash"],
            depth=data["depth"],
            score=data["score"],
            move=get_serialised_move(
                data["move"]
            )
        )

    @pre_dump
    def pre_dump(self, minimax_cache_item: Minimax_Cache_Item, **kwargs):
        # print(f"pre_dump: deserializing move from
{minimax_cache_item.move} to
{get_deserialized_move(minimax_cache_item.move)}")
        return dict(
            depth=minimax_cache_item.depth,
            move=get_deserialized_move(
                minimax_cache_item.move
            ),
            score=minimax_cache_item.score
        )

# this object can be used to serialize and deserialize moves to be stored
in the database
# it is the final product of this file and the object that will be
exported
minimax_cache_item_schema = Minimax_Cache_Item_Schema()
```

This code will allow me to convert a dictionary description of all the properties of a minimax search to a Minimax_Cache_Item object that can be used as an entry in a database.

I also use schemas to serialise data for sending to the client:

Socket schemas.py:

```
from marshmallow import Schema, fields, pre_load, pre_dump, post_load,
post_dump
from chess_functions import Vector, Piece, PIECE_TYPES

# different schema to serialize and deserialize a vector
class Vector_Schema(Schema):
    i = fields.Integer(required=True)
    j = fields.Integer(required=True)

    @post_load
    def get_vector_from_internal_data(self, internal_data, **kwargs):
        # print({"internal_data": internal_data})
        return Vector(**internal_data)

    @pre_load
    def get_internal_data_from_list(self, list_data, **kwargs):
        return {"i": list_data[0], "j": list_data[1]}

    @post_dump
    def make_list_from_internal_data(self, internal_data, **kwargs):
        return [internal_data['i'], internal_data['j']]

letter_symbol_pairs = (
    ("P", "♙"),
    ("K", "♔"),
    ("Q", "♕"),
    ("R", "♖"),
    ("B", "♗"),
    ("N", "♘"),
)
# these functions get the symbol or letter associated with each piece by
# searching the above tuple
def get_symbol_from_letter(target_letter):
    for letter, symbol in letter_symbol_pairs:
        if letter == target_letter.strip().upper():
            return symbol
    raise ValueError(f"letter '{target_letter}' not found")
```

```
def get_letter_from_symbol(target_symbol):
    for letter, symbol in letter_symbol_pairs:
        if symbol == target_symbol.strip():
            return letter
    raise ValueError(f"symbol '{symbol}' not found")

# this schema serialised and deserializes between the Pieces object and
# format use in JSON sent to the client
class Piece_Schema(Schema):
    color = fields.String(required=True)
    letter = fields.String(required=True)

    @pre_load
    def pre_load(self, data, **kwargs):
        color, symbol = data
        letter = get_letter_from_symbol(symbol)
        return {
            "color": color,
            "letter": letter,
        }

    @post_load
    def post_load(self, data, **kwargs):
        color, letter = data["color"], data["letter"]
        Piece_Type = PIECE_TYPES[letter]
        return Piece_Type(color)

    @pre_dump
    def pre_dump(self, piece: Piece, **kwargs):
        color, letter = piece.symbol()
        return {
            "color": color,
            "letter": letter,
        }

    @post_dump
    def post_dump(self, data, **kwargs):
        color, letter = data["color"], data["letter"]
        symbol = get_symbol_from_letter(letter)
        return [color, symbol]

# these function are utility function that can map a function across a 1d
# or 2d array, transforming all the elements
def list_map(some_function, array):
    return list(map(some_function, array))
```

```
def two_d_list_map(some_function, two_d_array):
    return list_map(
        lambda one_d_array: list_map(
            some_function,
            one_d_array
        ),
        two_d_array
    )

# create objects from the schemas with which to serialize and deserialize
vector_schema = Vector_Schema()
piece_schema = Piece_Schema()

# here are functions that use the schemas to serialize and deserialize
key data points to and from a dictionary (JSON like) format
def serialize_legal_moves(legal_moves):
    return two_d_list_map(vector_schema.dump, legal_moves)

def deserialize_legal_moves(legal_moves):
    return two_d_list_map(vector_schema.load, legal_moves)

def serialize_move(move):
    return list_map(vector_schema.dump, move)

def deserialize_move(move):
    return list_map(vector_schema.load, move)

def serialize_piece(piece):
    if piece is None:
        return [None, None]
    else:
        return piece_schema.dump(piece)

def deserialize_piece(piece):
    # print(piece, end=";   ")
    if piece == [None, None]:
        return None
    else:
        return piece_schema.load(piece)

def serialize_pieces_matrix(pieces_matrix):
    return two_d_list_map(serialize_piece, pieces_matrix)

def deserialize_pieces_matrix(pieces_matrix):
    # print("deserialize_pieces_matrix  called")
    return two_d_list_map(deserialize_piece, pieces_matrix)
```

Here is a jupyter notebook where I show what each of these schemas does:

```
[1]     from schemas import *
         from chess_functions import Board_State, Pawn
[1]     Python
D ▾  serialize_legal_moves(list(Board_State().generate_legal_moves()))
[2]     Python
...
[[[0, 1], [0, 1]],
 [[0, 1], [0, 2]],
 [[1, 0], [-1, 2]],
 [[1, 0], [1, 2]],
 [[1, 1], [0, 1]],
 [[1, 1], [0, 2]],
 [[2, 1], [0, 1]],
 [[2, 1], [0, 2]],
 [[3, 1], [0, 1]],
 [[3, 1], [0, 2]],
 [[4, 1], [0, 1]],
 [[4, 1], [0, 2]],
 [[5, 1], [0, 1]],
 [[5, 1], [0, 2]],
 [[6, 0], [-1, 2]],
 [[6, 0], [1, 2]],
 [[6, 1], [0, 1]],
 [[6, 1], [0, 2]],
 [[7, 1], [0, 1]],
 [[7, 1], [0, 2]]]

[3]     piece = Pawn("W")
         piece_schema.dump(piece)
[3]     Python
...
['W', '♟']
```

```
▶ 
    pieces_matrix = Board_State().pieces_matrix
    serialize_pieces_matrix(pieces_matrix)
[4] 
... Output exceeds the size limit. Open the full output data in a text editor
[[['B', 'X'],
  ['B', '▲'],
  ['B', '▲'],
  ['B', '■'],
  ['B', '■'],
  ['B', '▲'],
  ['B', '▲'],
  ['B', 'X']],
 [[['B', '▲'],
  ['B', '▲']],
  [[None, None],
   [None, None]],
  [[None, None],
   ...
   ['W', '■'],
   ['W', '▲'],
   ['W', '▲'],
   ['W', '▲'],
   ['W', 'X']]]
```



```
[5] 
    deserialize_piece(['W', '▲'])
... Pawn(color='W')
```

```
▶ [6]     serialize_piece(None)
...     [None, None]
Python

[8]     print(deserialize_piece([None, None]))
...     None
Python
```

These functions will be used as shown to format data from python objects to data structures like dictionaries that can be easily converted to JSON.

Website Module:

This module is responsible for creating a webserver to host the webpage and for handling all the client page's requests.

Here is the `__init__.py` file:

```
from .flask_server import app
```

As we can see, the only component that is exported is called `app`. This is a flask object that has been given all the functionality we want in our webserver / socket server.

The website is run by a file in the root directory called `app.py`

`app.py`

```
from website import app
from database import persistent_DB_engine, volatile_RAM_engine,
end_engines
# from geventwebsocket.server import WSGIServer
# import socketio
# import flask

host = '127.0.0.1'
port_num = 5000
url = f"http://{{host}}:{{port_num}}"

def run_app():
    print(url)
    try:
```

```
app.run(
    host=host,
    port=port_num,
    debug=True
)
except KeyboardInterrupt:
    end_engines([persistent_DB_engine, volatile_RAM_engine])

# def run_app():
#     print(url)
#     try:
#         socketio.run(app)
#     except KeyboardInterrupt:
#         pass

# def run_app():
#     http_server = WSGIServer(('0.0.0.0', 5000), app)
#     http_server.serve_forever()

# def run_app():
#     # socketio.run(app, host='0.0.0.0', port=5000, debug=True,
use_reloader=False)
#     # flask.run(app, host='0.0.0.0', port=5000, debug=True,
use_reloader=False)
#     app.run(host=host, port=port_num, debug=True, use_reloader=False)

# use python -m flask run to stop it raising warnings about how it should
be deployed
if __name__ == "__main__":
    run_app()
```

```
from website import app
from database import persistent_DB_engine, volatile_RAM_engine,
end_engines

# variables to decide the URL of the website
host = '127.0.0.1'
port_num = 5000
url = f"http://{host}:{port_num}"

# this function runs the flask server (hosting the website)
# to close the server, use ctrl + c to create a keyboard interrupt,
# this will safely destroy all engine connections to the database
def run_app():
    print(url)
    try:
```

```
        app.run(
            host=host,
            port=port_num,
            debug=True
        )
    except KeyboardInterrupt:
        end_engines([persistent_DB_engine, volatile_RAM_engine])

# use python -m flask run to stop it raising warnings about how it should
# be deployed
if __name__ == "__main__":
    run_app()
```

the website is run with the following command:

```
PS C:\Users\henry\Documents\computing coursework\prototype 6> python -m flask run
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

The python file flask_server.py contains all the server side logic of the webserver and socket server:

```
# import external modules
import flask
import os
from flask_socketio import SocketIO
import datetime
import pickle

# import local modules
from assorted import safe_hash
from database import save_game, get_saved_game, persistent_DB_engine,
create_session, end_session
from chess_game import Game_Website
from schemas import serialize_legal_moves, deserialize_move,
serialize_pieces_matrix, serialize_piece
```

```
# fix tabs don't exist on reload bug:  
https://stackoverflow.com/questions/44531360/flask-blogging-error-table-doesnt-exist-tables-not-being-created  
  
# establish various important directories in variables  
basedir = os.getcwd()  
  
static_folder_path = os.path.join(basedir, 'website', 'static')  
template_folder_path = os.path.join(basedir, 'website', 'templates')  
key_file_path = os.path.join(basedir, 'website', 'secret_key.key')  
  
# retrieve the secret key  
with open(key_file_path, "r") as file:  
    secret_key = file.read()  
  
# sanitize secret key to remove white space  
secret_key = secret_key\  
    .replace(" ", "")\  
    .replace("\n", "")\  
    .replace(chr(13), "")  
  
# create an app object and configure it  
app = flask.Flask(  
    __name__,  
    static_folder=static_folder_path,  
    template_folder=template_folder_path,  
)  
# app.config['SEND_FILE_MAX_AGE_DEFAULT'] = 0  
app.secret_key = secret_key  
  
# create the socketio object (bound to the app object)  
# socketio = SocketIO(app, async_mode=None)  
# socketio = SocketIO(app, async_mode="gevent")  
socketio = SocketIO(app, async_mode=None, threaded=True, echo=False)  
  
@app.route("/")  
def index():  
    # this is the only http endpoint,  
    # it renders the appropriate html file  
  
    response = flask.make_response(  
        flask.render_template("chess_game.html"))
```

```
)  
  
    # ot then adds a cookie to the response that contains a cookie key  
    # corresponding to the game in the session  
    expires = datetime.datetime.now() + datetime.timedelta(hours=48)  
  
    cookie_key = flask.session["cookie_key"]  
    cookie_formatted = str(cookie_key).encode("utf-8")  
  
    response.set_cookie(  
        "chess_game_cookie_key",  
        cookie_formatted,  
        expires=expires  
    )  
  
    # print(f"index: cookie created: {cookie_key}")  
  
    # the response is returned, creating the cookie and providing the html  
    file for the webpage  
    return response  
  
# the below functions both get and save a Game_Website object to the flask  
session as a binary blob  
# this prevent issues that can occur when python objects are saved  
directly in the flask session  
def get_game_in_session() -> Game_Website:  
    # print(repr(flask.session["game"]))  
    # print(repr(pickle.loads(flask.session["game"])))  
    return pickle.loads(  
        flask.session["game"]  
    )  
  
def set_game_in_session(game: Game_Website):  
    flask.session["game"] = pickle.dumps(  
        game  
    )  
  
# def get_game_in_session() -> Game_Website:  
#     return flask.session["game"]  
  
# def set_game_in_session(game: Game_Website):  
#     flask.session["game"] = game  
  
# this handler function is responsible for initializing a flask session  
def create_flask_session():
```

```
# if flask.session.get("session_setup_complete"):
#     return None
# else:
#     flask.session["session_setup_complete"] = True

# it first checks the cookies

# print("Creating session")
# check cookies
# game_hash_cookie = flask.request.cookies.get("chess_game_hash")
game_cookie = flask.request.cookies.get("chess_game_cookie_key")

# print(f"In creating session, cookie chess_game_cookie_key fetched,
it contains: {game_cookie}")

# if the appropriate cookie doesn't exist, a random number is used as
the cookie key,
# and a fresh game is created
if game_cookie is None:
    print("Cookie was none / doesn't exist")
    game = Game_Website()
    cookie_key = safe_hash(os.urandom(32))

# if a cookie id was recovered
else:

    # the id is converted into a python sting that contains
    # hexadecimal characters
    hex_string = game_cookie.encode("utf-8").hex()
    cookie_key = bytes.fromhex(hex_string).decode('utf-8')
    print(f"flask cookie contained hash: {cookie_key}")

    # a database session is created
    session_DB = create_session(persistent_DB_engine)

    # the game object is retrieved from the database
    # database_lookup_result: Game_Website =
get_saved_game(game_hash=game_hash, session=session)
    database_lookup_result: Game_Website =
get_saved_game(cookie_key=cookie_key, session=session_DB)
    # print("database_lookup_result")
    # print(database_lookup_result)

    # the session is disposed of
end_session(session_DB)
    session_DB = None
```

```
# if the game was found in the database, use it, else create a new
game
    if database_lookup_result is None:
        print(f"game: NOT successfully read from database (returned
None)")
        game = Game_Website()
    else:
        print(f"game successfully read from database")
        game = database_lookup_result
        print(f"game with hash {hash(game)} successfully read from
database")

# now that a game object and a cookie_key have been determined, store
these in the session object
    # the index http route will create the appropriate cookie using this
id

    # print(repr(game))
    set_game_in_session(game)
    # print(f"At the end of create_flask_session, setting cookie_key to
: {cookie_key}")
    flask.session["cookie_key"] = cookie_key

    # print(f"At the end of create_flask_session, game used has
ID: {hash(game)})"

# this handler function is responsible for clearing up a flask session
# this involves saving the game to the database
def close_flask_session():
    # if flask.session.get("session_setup_complete"):
    #     flask.session["session_setup_complete"] = False
    # else:
    #     return None

    # print("Closing session")
    # game: Game_Website = get_game_in_session()

    # get the game from the database
    game: Game_Website = pickle.loads(
        flask.session["game"]
    )

    # if the window was closed part way though the computer move
    # quickly decide the computers move (time=0) so that game can be
resorted on the user's turn
```



```
close_flask_session()

@app.route('/stop_and_save_game')
def handle_close_session(*args, **kwargs):
    # this function sets the flag to indicate that the session needs to be
    reinitialized
    flask.session["session_created"] = False
    return "session closed"

# here I have created a socket handler function
# it is a decorator as it takes a function as a parameter and returns a
modified function with additional functionality
def bind_socket_handler(event_name, respond=True):
    def decorator(function):
        request_event = f"{event_name}_request"
        response_event = f"{event_name}_response"

        # new function to be returned takes incoming payload as an
        argument
        def wrapper(incoming_payload: dict | None = None):
            # print(f"Handling event: {request_event}")

            # if there was an incoming payload, this is given to the
            original function as an argument after being deserialized
            if incoming_payload is None:
                result = function()
            else:
                # print({"incoming_payload": incoming_payload})
                if not isinstance(incoming_payload, dict):
                    raise TypeError(f"incoming payload of unexpected
type: {type(incoming_payload)}")

                result = function(incoming_payload)

            # if the respond flag is true, the return value of the
            function is send back to the client
            if respond:
                outgoing_payload: dict = result
                # print(f"Sending response payload by event
{response_event}")
                # print({"outgoing_payload": outgoing_payload})

                if not isinstance(outgoing_payload, dict):
                    raise TypeError(f"outgoing payload of unexpected
type: {type(outgoing_payload)}")
```

```
        socketio.emit(response_event, outgoing_payload)

    # the wrapper function name is set to the name of the original
function
    wrapper.__name__ = function.__name__

    # bind wrapper function to happen when request corresponding to
the event is received
    # use decorator in manual way to not overwrite wrapper
    socketio.on(request_event)(wrapper)

    # return wrapper for any further use
    return wrapper

return decorator

# the below function accesses the game object in the session
# it returns a dictionary of serialized data about this game object
def generate_game_update_data():
    game: Game_Website = get_game_in_session()

    # collect various data point from the game object and serialize as
necessary
    next_to_go = game.board_state.next_to_go
    difficulty = game.difficulty

    legal_moves = list(game.board_state.generate_legal_moves())
    legal_moves_serialised = serialize_legal_moves(legal_moves)

    pieces_matrix = game.board_state.pieces_matrix
    pieces_matrix_serialised = serialize_pieces_matrix(pieces_matrix)

    next_to_go_in_check = game.board_state.color_in_check()

    over, winning_player, victory_classification = game.check_game_over()

    game_over_data = {
        "over": over,
        "winning_player": winning_player,
        "victory_classification": victory_classification,
    }

    pieces_missing = {}
    for color in ("B", "W"):
        pieces_missing[color] = list(map(
            serialize_piece,
```

```
        game.board_state.generate_pieces_taken_by_color(color)
    ))
```

```
move_history = game.move_history_output
```

```
# this is the format of the outgoing payload,
# flask will convert it to JSON automatically
# this payload contains all the data that the client needs to be
updated about the game state
payload = {
    "difficulty": difficulty,
    "next_to_go": next_to_go,
    "game_over_data": game_over_data,
    "legal_moves": legal_moves_serialised,
    "pieces_matrix": pieces_matrix_serialised,
    "check": next_to_go_in_check,
    "pieces_taken": pieces_missing,
    "move_history": move_history,
}
return payload
```



```
# using my custom socket decorator greatly simplifies the process of using
sockets
```

```
# on request get update from the client, return the generic payload of
game data
@bind_socket_handler("get_update")
def get_update():
    """Sends all data from game object to client to update chess game
    This data is also updated after the user move and after the computer
move"""
    result = generate_game_update_data()
    # print(f"Update payload for game with
hash {hash(get_game_in_session())}")
    # print(result)
    return result
```



```
# on request to implement the computer move
@bind_socket_handler("implement_computer_move")
def implement_computer_move():

    # get the game and validate that the computer can go
    game = get_game_in_session()
    assert game.board_state.next_to_go == "B"
    assert not game.board_state.is_game_over_for_next_to_go()[0]
```

```
# generate the move and a dictionary of data about it for the client
# print("generating move")
move_description = game.implement_computer_move_and_report()

# update the session with the new mutated game object
set_game_in_session(game)

# print("after computer move: game in session is:")
# get_game_in_session().board_state.print_board()

# generate a generic update payload and add teh move description
before returning
outgoing_payload = generate_game_update_data()
outgoing_payload["computer_move_description"] = move_description

# print(outgoing_payload)

return outgoing_payload

# when a request comes to reset the game
# (no response needed)
@bind_socket_handler("reset_game", respond=False)
def reset_game():
    # preserve the difficulty setting
    old_game = get_game_in_session()
    # difficulty preserved
    difficulty = old_game.difficulty

    # create a new game object with the old difficulty and update the
    # session
    new_game = Game_Website(difficulty=difficulty)

    set_game_in_session(new_game)

    # return generate_game_update_data()

# takes input from client about the user's move
@bind_socket_handler("implement_user_move")
def implement_user_move(incoming_payload):
    game = get_game_in_session()

    # validate that the user can move
    assert game.board_state.next_to_go == "W"
    assert not game.board_state.is_game_over_for_next_to_go()[0]

    # print("python function called: implement_user_move")
```

```
# deserialize the user's move into vectors
user_move = tuple(
    deserialize_move(
        incoming_payload["user_move"]
    )
)
# implement the move on the game object
game.implement_user_move(user_move)

# print("python function finished: implement_user_move")
# update the session with the new mutated game object
set_game_in_session(game)

# print("after user move: game in session is:")
# get_game_in_session().board_state.print_board()

# generate and return an update payload
return generate_game_update_data()

# on request to change difficulty
@bind_socket_handler("change_difficulty", respond=False)
def change_difficulty(incoming_payload):
    game = get_game_in_session()

    # get the new difficulty from the payload
    new_difficulty = incoming_payload["new_difficulty"]
    # print(f"Changing difficulty to {new_difficulty}")

    # mutate the game object and save it to the session
    game.difficulty = new_difficulty

    set_game_in_session(game)

    # doesn't return an update
    # return generate_game_update_data()

# this code below was used to generate the initial game data json file
# that I no longer use
# before the reload game feature, this prevented the client needing to
# request the initial game data immediately on loading

# app = None
# if __name__ == "__main__":
#     import json
#     game: Game_Website = Game_Website()
```

```
#     next_to_go = game.board_state.next_to_go
#     difficulty = game.difficulty

#     legal_moves = list(game.board_state.generate_legal_moves())
#     legal_moves_serialised = serialize_legal_moves(legal_moves)

#     pieces_matrix = game.board_state.pieces_matrix
#     pieces_matrix_serialised = serialize_pieces_matrix(pieces_matrix)

#     next_to_go_in_check = game.board_state.color_in_check()

#     over, winning_player, victory_classification =
game.check_game_over()

#     game_over_data = {
#         "over": over,
#         "winning_player": winning_player,
#         "victory_classification": victory_classification,
#     }

#     pieces_missing = {}
#     for color in ("B", "W"):
#         pieces_missing[color] = list(map(
#             serialize_piece,
#             game.board_state.generate_pieces_taken_by_color(color)
#         ))

#     move_history = game.move_history_output

#     payload = {
#         "difficulty": difficulty,
#         "next_to_go": next_to_go,
#         "game_over_data": game_over_data,
#         "legal_moves": legal_moves_serialised,
#         "pieces_matrix": pieces_matrix_serialised,
#         "check": next_to_go_in_check,
#         "pieces_taken": pieces_missing,
#         "move_history": move_history,
#     }

#     with open("./website/static/initial_game_data.json", "w") as file:
#         file.write(
#             json.dumps(payload)
#         )
```

The above code contains:

- A http route to set a cookie and give the webpage html
- Many socket event handlers to allow the client and the server to communicate
- Functions for saving the game in the session to the database and reloading it into the session from the id in the client's cookie.

Due to the fact that I couldn't set cookies to the client browser in any method other than the index http handler (only http request direct from browser) I needed to decide the cookie key that I would use before the final game was saved. This means that I had to use a large random number and couldn't use the hash of the game state. This is because the handle session close method couldn't set cookies.

I then used client side HTML, CSS and JavaScript for the front end component of the website:

Chess game.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chess Game</title>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js"
        integrity="sha512-
9mpsATI0KClwt+xVZfbcf2lJ8IFBAwsubJ6mI3rtULwyM3fBmQFzj0It4tGqxLOGQwGfJdk/G+
fANnxfq9/cew=="
        crossorigin="anonymous" referrerpolicy="no-referrer"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-
js/4.1.1/crypto-js.min.js"
        integrity="sha512-
E8QSvWZ0eCLGk4km3hxSsNmGwbLtSCSUcewDQPQWZF6pEU8GLT8a5fF32w0l1i8ftdMhssTrF/
OhyGwlwonTcXA=="'
        crossorigin="anonymous" referrerpolicy="no-referrer"></script>

    <!-- <script type="application/json" src="{{ url_for('static',
filename='initial_game_data.json') }}></script> -->
    <script type="text/javascript" src="{{ url_for('static',
filename='initial_game_data.js') }}></script>

    <script type="text/javascript" src="{{ url_for('static',
filename='vector.js') }}></script>
```

```
<script type="text/javascript" src="{{ url_for('static',
filename='chess_class.js') }}></script>

<link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
<link rel="shortcut icon" href="{{ url_for('static',
filename='favicon.ico') }}" type="image/x-icon">

</head>
<body>
<div class="centered">

    <h1 id="top_title"></h1>
    <!-- <span class="spacer" style="height: 1vmin;"></span> -->
    <span class="spacer"></span>
    <div id="board"></div>

    <!-- <span class="spacer" style="height: 1vmin;"></span> -->
    <span class="spacer"></span>
    <div class="option_button_container">
        <!-- <button class="option_button" id="concede_button">Concede
Game</button> -->
        <button class="option_button"
id="concede_button">Concede</button>
        <!-- <button class="option_button" id="restart_button">Restart
Game</button> -->
        <button class="option_button"
id="restart_button">Restart</button>
    </div>

    <!-- <div class="spacer"></div> -->
    <span class="spacer"></span>
    <span class="spacer"></span>

    <div class="difficulty_form">
        <h1>Change Difficulty: (AI thinking time)</h1>
        <!-- <p>Change Difficulty:</p> -->
        <form id="difficulty_form">
            <label>
                <input type="radio" name="difficulty"
value="really_easy" onclick="handle_radio_button_click(this)">
                    Trivial (1 sec)
            </label>
            <label>
                <input type="radio" name="difficulty" value="easy"
onclick="handle_radio_button_click(this)">
                    Easy (2 sec)
            </label>
        </form>
    </div>
</div>
```

```
        </label>
        <label>
            <input type="radio" name="difficulty" value="medium"
onclick="handle_radio_button_click(this)">
                Medium (5 sec)
        </label>

        <br>

        <label>
            <input type="radio" name="difficulty" value="hard"
onclick="handle_radio_button_click(this)">
                Hard (10 sec)
        </label>
        <label>
            <input type="radio" name="difficulty"
value="really_hard" onclick="handle_radio_button_click(this)">
                Challenge (15 sec)
        </label>
        <label>
            <input type="radio" name="difficulty" value="extreme"
onclick="handle_radio_button_click(this)">
                Extreme (30 sec)
        </label>

        <br>

        <label>
            <input type="radio" name="difficulty"
value="legendary" onclick="handle_radio_button_click(this)">
                Legendary (60 sec)
        </label>
    </form>
</div>

<!-- &lt;div class="spacer"&gt;&lt;/div&gt; --&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;

&lt;div class="pieces_taken_table"&gt;
    &lt;h1&gt;Pieces Taken:&lt;/h1&gt;
    &lt;table&gt;
        &lt;tr&gt;
            &lt;td&gt;White Pieces Taken&lt;/td&gt;
            &lt;td&gt;Black Pieces Taken&lt;/td&gt;
        &lt;/tr&gt;
        &lt;tr&gt;
            &lt;th id="num_pieces_taken_white"&gt;&lt;/th&gt;</pre>
```

```
        <th id="num_pieces_taken_black"></th>
    </tr>
    <tr>
        <th id="which_pieces_taken_white">-</th>
        <th id="which_pieces_taken_black">-</th>
    </tr>
</table>
</div>

<!-- &lt;div class="spacer"&gt;&lt;/div&gt; --&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;

&lt;div class="previous_moves_table"&gt;
    &lt;h1&gt;Moves History:&lt;/h1&gt;
    &lt;table id="pieces_taken_table_tag"&gt;

        &lt;tr&gt;
            &lt;td&gt;
                White previous moves:
            &lt;/td&gt;
            &lt;td&gt;
                Black previous moves:
            &lt;/td&gt;
        &lt;/tr&gt;

        &lt!-- &lt;tr&gt;
            &lt;td&gt;
                white move
            &lt;/td&gt;
            &lt;td&gt;
                black move
            &lt;/td&gt;
        &lt;/tr&gt; --&gt;

    &lt;/table&gt;
&lt;/div&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;

&lt;/div&gt;
&lt;script type="text/javascript" src="{{ url_for('static',
filename='main.js') }}"&gt;&lt;/script&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>
```

The above html contains some bare bones tags with ID's that allow them to be accessed by JS DOM (document object model) manipulator methods.

The chess board is drawn by JavaScript.

```
src="{{ url_for('static', filename='main.js') }}"
```

The strange source and href attributes of script and style tags that link to external files will be overwritten with the dynamic URL of the file by the flask server when the HTML is rendered.

Main.js

```
// these constants define the colors of text, squares and text shadow
// throughout the program
const white_sq_bg_color = '#f5e6bf';
// const white_sq_bg_color = '#d9cba7';
const black_sq_bg_color = '#66443a';
const white_piece_color = '#ffffff';
const black_piece_color = '#000000';
// const white_shadow = '-1px -1px 0 #000, 1px -1px 0 #000, -1px 1px 0
#000, 1px 1px 0 #000'
const black_shadow = '-1px -1px 0 #fff, 1px -1px 0 #fff, -1px 1px 0 #fff,
1px 1px 0 #fff'
const white_shadow = '-2px -2px 0 #000, 2px -2px 0 #000, -2px 2px 0 #000,
2px 2px 0 #000'
// const black_shadow = '-2px -2px 0 #fff, 2px -2px 0 #fff, -2px 2px 0
#fff, 2px 2px 0 #fff'

// this function creates a series of div tags that represent the
individual squares in the chess board
// it decides their color, position vector and hence their ID and click
event functions
function create_board_widget() {
    let board = document.getElementById("board");
    for (let row = 0; row <= 7; row++) {
        for (let col = 0; col <= 7; col++) {
            let i = col;
            let j = 7-row;

            // if row and column are both odd or both even then white, so
            add together and check if even
            let sum_is_even = ((row + col) % 2 == 0);
            // console.log(`square (${i},${j}), sum is even --> ${sum_is_even}`)
            let square_bg_color = sum_is_even ? white_sq_bg_color :
black_sq_bg_color;
            let square = document.createElement("div");
            // square.textContent = `(${i},${j})`;
            square.id = `square_${i}${j}`;
```

```
        square.classList.add("square");
        square.style.backgroundColor = square_bg_color;
        square.addEventListener("click",
function(){handle_square_click(i, j)})
            board.append(square)
        }
    }
}

// this function returns the html element of a square given a position
vector
function get_square_at_vector(v) {
    let [i, j] = v;
    let id = `square_${i}${j}`;
    let square = document.getElementById(id);
    return square;
}

// this function uses the pieces matrix from the board object to populate
the board and its squares with pieces
// it must decide their color and text shadow as well
function add_pieces(board) {
    let pieces_matrix = board.pieces_matrix
    // iterate through rows and columns
    for (let row = 0; row <= 7; row++) {
        for (let col = 0; col <= 7; col++) {
            // get the vector and decide the color
            let [i, j] = [col, 7-row]
            let [color_char, symbol] = pieces_matrix[row][col];
            let color = (color_char == "W") ? white_piece_color :
black_piece_color;

                // console.log(`get_square_at_vector([i, j]) ---> ${get_square_at_vector([i, j])}`)
            square = get_square_at_vector([i, j]);

            // square.innerText = `(${i}, ${j})`;
            // square.style.fontSize = "20px"
            square.innerText = symbol;
            square.style.color = color;
            square.style.textShadow = (color_char == "W") ? white_shadow :
black_shadow;
            square.style.fontSize = "9vmin"

        }
    }
}
```

```
}
```

```
// this function uses the highlighted_squares method of the board object
// to add highlighting to the chess board
// this highlights clicked pieces red and squares they can move to green
function add_highlighting(board) {
    let new_highlighting = board.get_highlighted_squares()
    // console.log(`add_highlighting function called with
${new_highlighting}`)

    // iterate through the position vectors and colors form the
get_highlighted_squares method
    for (let i in new_highlighting) {
        let [vector, color] = new_highlighting[i];

        // console.log(`Adding ${color} to square at vector ${vector}`);
        square = get_square_at_vector(vector);
        // console.log({ vector })
        // console.log({ square })
        // if (square.innerText == null) {

            // if a square should be highlighted but is empty: add a center
dot
            if (square.innerText.trim() == "") {
                square.innerText = ".";
                // square.style.fontSize = "20vmin";
                square.style.fontSize = "10vmin";
            }
            square.style.color = color;
            // square.style.backgroundColor = color;
        }
    }
}

// this function was not needed in the end
// if wipes the board of all pieces and highlighting
function clear_board() {
    for (let i = 0; i <= 7; i++) {
        for (let j = 0; j <= 7; j++) {
            square = get_square_at_vector([i, j]);
            // console.log(`Square at vector ${[i, j]} is ${square}`);
            square.textContent = "";
            // square.innerText = "";
            square.style.color = "black";
            square.style.textShadow = "";
            square.style.fontSize = "9vmin"
        }
    }
}
```

```

}

// this function updates the main title widget with a new title based on
the board object
function update_main_title(board, manual_new_title=null) {

    if (manual_new_title !== null) {
        document.getElementById("top_title").innerText = manual_new_title
        return null;
    }

    let title_msg

    if (board.just_conceded) {
        // console.log({board})
        // console.log(JSON.stringify(board))
        // console.log(board.just_conceded)
        title_msg = "Game Over: You Conceded"
    }
    else {
        title_msg = `${(board.next_to_go === "W") ? "Your Go:" :
"Computer's Go... (please wait)"}${(board.check) ? " (CHECK)" : ""}`
    }

    document.getElementById("top_title").innerText = title_msg
}

// this function adds the pieces taken data (number and specific pieces)
to the table
function update_pieces_taken(board) {

    let num_B_taken = board.pieces_taken.B.length
    let num_W_taken = board.pieces_taken.W.length
    let B_pieces_conjugation = (num_B_taken == 1) ? "Piece": "Pieces"
    let W_pieces_conjugation = (num_W_taken == 1) ? "Piece": "Pieces"

    document.getElementById("num_pieces_taken_black").innerText =
(num_B_taken > 0) ? `${num_B_taken} ${B_pieces_conjugation} Taken` : "No
Pieces Taken";
    document.getElementById("num_pieces_taken_white").innerText =
(num_W_taken > 0) ? `${num_W_taken} ${W_pieces_conjugation} Taken` : "No
Pieces Taken";

    // document.getElementById("which_pieces_taken_black").innerText =
(board.pieces_taken.B.length > 0) ? `Pieces Lost:
${board.pieces_taken.B.map(x => x[1]).join(", ")}` : "-";
}

```

```
document.getElementById("which_pieces_taken_black").innerText =
(board.pieces_taken.B.length > 0) ? `${board.pieces_taken.B.map(x =>
x[1]).join(", ")}: "-";
// document.getElementById("which_pieces_taken_white").innerText =
(board.pieces_taken.W.length > 0) ? `Pieces Lost:
${board.pieces_taken.W.map(x => x[1]).join(", ")}: "-`;
document.getElementById("which_pieces_taken_white").innerText =
(board.pieces_taken.W.length > 0) ? `${board.pieces_taken.W.map(x =>
x[1]).join(", ")}: "-`;
}

// this function auto selects one of the difficulty radio buttons
function set_selected_difficulty(board) {
    let difficulty = board.difficulty
    // console.log({difficulty})

    document.querySelector(`input[value=${difficulty}]`).checked = true;
    // document.querySelector(`input[value="${difficulty}"]`).checked =
true;
    // document.querySelector(`input[value=${difficulty}]`).checked =
true;
}

// this function adds moves to the move history table
function set_widget_move_history(board) {
    let moves = board.move_history.length;

    let rows
    let move_history_output = Array.from(board.move_history)
    // depending on the number of moves made in the game, extra cells may
need to be added to make the table look right
    if (moves == 0) {
        rows = 1;
        move_history_output.push("-")
        move_history_output.push("-")
    }
    else if (moves % 2 == 0) {
        rows = moves / 2
    }
    else {
        rows = (moves + 1) / 2
        move_history_output.push("-")
    }
    // console.log({half_moves_rounded_up});
    // have a row for each number in half_moves_rounded_up
```

```
// delete old elements from the table
let old_rows =
Array.from(document.querySelectorAll('.temporary_previous_moves_table_item'));

if (old_rows.length > 0) {
    for (let i in old_rows) {
        old_rows[i].removeChild(old_rows[i].firstChild);
        old_rows[i].removeChild(old_rows[i].firstChild);
        old_rows[i].parentNode.removeChild(old_rows[i]);
    }
}

// iterate through the rows and add new TD tags to add the new row to
the table
table = document.getElementById("pieces_taken_table_tag")
// console.log({table})

let white_move
let black_move
let new_row
let cells
for (let r=0; r<rows; r++) {
    white_move = move_history_output[2*r];
    black_move = move_history_output[1+ 2*r];

    new_row = document.createElement("tr")
    cells = [
        document.createElement("td"),
        document.createElement("td")
    ]

    cells[0].innerText = white_move;
    cells[1].innerText = black_move;

    new_row.classList.add("temporary_previous_moves_table_item")
    cells[0].classList.add("small_text")
    cells[1].classList.add("small_text")

    new_row.appendChild(cells[0]);
    new_row.appendChild(cells[1]);

    table.appendChild(new_row)
}
```

```
// gets the hash of a given piece of data
// https://stackoverflow.com/questions/54701686/matching-cryptojs-sha256-with-hashlib-sha256-for-a-json
function get_hash_of_data(data, stringify = true) {
    let data_string
    if (stringify) {
        data_string = JSON.stringify(data)
    }
    else {
        data_string = data
    }

    return CryptoJS.SHA256(data_string).toString(CryptoJS.enc.Base64);
}

// holds a table that keep track of the hashes of various items of data
// includes data item name as a key and current hash + function to find
hash as values
previous_data_hashes = {
    pieces_taken: [null, (board) => get_hash_of_data(board.pieces_taken)],
    move_history: [null, (board) => get_hash_of_data(board.move_history)],
    piece_layout: [null, (board) => get_hash_of_data([board.pieces_matrix,
board.possible_to_vectors, board.selected_from_vector])],
    highlighting: [null, (board) =>
get_hash_of_data([board.possible_to_vectors,
board.selected_from_vector])],
    difficulty: [null, (board) => get_hash_of_data(board.difficulty,
stringify=false)]
};

// this function only runs the DOM update method given if the data to
display changed
function update_as_necessary(board, update_function, hashes_table_key) {
    // create a new hash
    // only if it is different from the old hash, then update the board
and the hash in the data previous_data_hashes object

    // console.log(previous_data_hashes[hashes_table_key])
    let [old_hash, compute_hash] = previous_data_hashes[hashes_table_key];
    let new_hash = compute_hash(board);

    if (new_hash !== old_hash) {
        // console.log(`hashes are different, updating
${hashes_table_key}`)
        previous_data_hashes[hashes_table_key][0] = new_hash
        update_function(board)
    }
    // else {
}
```

```
//      console.log(`hashes the same, NOT updating
${hashes_table_key}` )
// }
}

// update all dom widgets as necessary, always update title
function update_board_widget(board) {
    // clear_board();
    // console.log("board.pieces_matrix")
    // console.log(board.pieces_matrix)

    // add_pieces(board);
    // add_highlighting(board);
    // update_main_title(board);
    // set_selected_difficulty(board);
    // update_pieces_taken(board);
    // set_widget_move_history(board);

    update_main_title(board);
    update_as_necessary(board, add_pieces, "piece_layout")
    update_as_necessary(board, add_highlighting, "highlighting")
    update_as_necessary(board, set_selected_difficulty, "difficulty")
    update_as_necessary(board, update_pieces_taken, "pieces_taken")
    update_as_necessary(board, set_widget_move_history, "move_history")
}

// calls teh appropriate square click function on the board and then
updates the DOM
function handle_square_click(i, j) {
    // board is a global variable
    // console.log(`Square at (${i}, ${j}) has been clicked`);
    board.handle_square_click([i, j]);
    // console.log(board.pieces_matrix);
    update_board_widget(board);
    // alert("about to clear board");
    // clear_board();
}

// whole file loads after html
// create chess board html elemtent
create_board_widget();
```

```
// create board class
let board = new Chess_Board(
    update_board_widget,
    update_main_title
);

// bind buttons to methods of the board class
document.getElementById("restart_button").addEventListener("click",
function () {
    board.reset_game()
})
document.getElementById("concede_button").addEventListener("click",
function () {
    // update_main_title(null, "You Conceded The Game")
    board.concede_game()
})

// bind a change to the radio buttons to a handler method in the board
// class
function handle_radio_button_click(radio_button) {
    let new_difficulty = radio_button.value;
    if (new_difficulty !== board.difficulty) {
        // console.log(`Setting new difficulty to ${new_difficulty}`)
        board.change_difficulty(new_difficulty);
    }
}

// before letting the window close, send the server a warning so it can
// save the game
// also produces a pop up box to prevent accidental closing of the game
window.onbeforeunload = () => fetch('/stop_and_save_game');
```

The main function of this file is to:

- Define methods that can affect the DOM to display outputs (changes in the `Chess_Board` object)
- Create the chess board widget.
- Create an instance of the `Chess_Board` class that is held in global variable called `board`.
- Tie together button click events to the board objects handlers and also provide the board with functions needed to update the html widgets.

The other main JavaScript file is the `chess_class.js` file:

```
// utility functions for use in the program

function arrays_are_equal(a1, a2) {
    return JSON.stringify(a1) == JSON.stringify(a2)
```

```
}
```

```
function two_d_array_contains_sub_array(two_d_array, sub_array) {
    for (let i in two_d_array) {
        if (arrays_are_equal(two_d_array[i], sub_array)) {
            return true;
        }
    }
    return false;
}
```

```
function assert(condition, message) {
    if (!condition) {
        message = (message === null? "Assertion error": message);
        throw new Error(message);
    }
}
```

```
// socket object to manage socket connection
let socket = io();
```

```
// this is a factory function that takes an event and returns an
asynchronous function
// the function will send a request to the server and then await and
return the server's resonance
function async_function_factory_send_and_receive(event_name) {
    // returns an async function
    async function external_get_response(outgoing_payload = null) {

        // this promise resolves then a response from the server is
        received, the response is given
        const server_response_promise = new Promise(function (resolve) {
            socket.on(` ${event_name}_response`, function (data) {
                resolve(data);
            })
        });

        // the request is sent to the server, along with an outgoing
        payload if appropriate
        if (outgoing_payload === null) {
            socket.emit(` ${event_name}_request`);
        }
        else {
            socket.emit(
                ` ${event_name}_request`,
                outgoing_payload
            )
        }
    }
}
```

```
        );
    }

    // the server's response is then awaited
    let response_payload = await server_response.promise
    // console.log({ response_payload })

    // if the response contains any actual data then it is returned
    var is_null = (response_payload === null)
    var is_empty_sting = (response_payload === "")
    var is_empty_dict = (Object.keys(response_payload).length === 0)

    if (is_null || is_empty_sting || is_empty_dict) {
        return null;
    }
    else {
        return response_payload
    }

    // the async function with this behaviour is returned to be reused
}
return external_get_response
}

// define async functions to make requests to server

// simpler functions that don't need to send and data to the server
external_get_update =
async_function_factory_send_and_receive("get_update");
external_implement_computer_move_and_update =
async_function_factory_send_and_receive("implement_computer_move")
external_reset_game =
async_function_factory_send_and_receive("reset_game")

// more complex functions that must crate an outgoing payload and send it
// to the server, then return the response
async function external_implement_user_move_and_update(from_vector,
to_vector) {
    movement_vector = subtract_vectors(to_vector, from_vector);
    outgoing_payload = {
        "user_move": [from_vector, movement_vector],
    };
    data_exchange_handler_function =
    async_function_factory_send_and_receive("implement_user_move")
    server_data = await data_exchange_handler_function(outgoing_payload);

    return server_data;
}
```

```
}
```

```
async function external_change_difficulty(new_difficulty) {
    // console.log(`sending server new difficulty of ${new_difficulty}`)
    outgoing_payload = { "new_difficulty": new_difficulty }
    external_reset_game =
        async_function_factory_send_and_receive("reset_game")
        data_exchange_handler_function =
            async_function_factory_send_and_receive("change_difficulty")
            await data_exchange_handler_function(outgoing_payload);
}
```

```
// this class contains all the data and behaviour of the client side chess
game
class Chess_Board {
    // before reload feature, constructor just used initial game data
    // constructor(update_board_widget_function) {
    //     this.update_board_widget = function () {
    //         update_board_widget_function(this)
    //     }

    //     this.update_from_server_data(INITIAL_GAME_DATA)
    //     this.just_reset = true
    //     this.just_conceded = false
    // };

    // new constructor sets dom manipulation as properties
    // then makes a request to the server to determine it properties
    constructor(update_board_widget_function, update_main_title_widget) {
        this.update_board_widget = function () {
            update_board_widget_function(this)
        }
        this.update_main_title_widget = function (msg) {
            update_main_title_widget(this, msg)
        }

        // this.update_from_server_data(INITIAL_GAME_DATA)
        this.direct_server_grand_update()
    };

    // this updates the board with the server's data but also sets reset
    flags to false
    async direct_server_grand_update() {
        let server_data = await external_get_update();
        this.update_from_server_data(server_data);
        this.just_reset = false
        this.just_conceded = false
    }
}
```

```
        assert(this.next_to_go !== "B")

    }

    // this function takes server data and unpacks it,
    // the properties of the board object are then updated with this new
    server data
    // the board widget is then redrawn
    update_from_server_data(server_data, user_input_disabled = false) {
        this.possible_to_vectors = [];
        this.selected_from_vector = null;
        this.user_input_disabled = user_input_disabled

        this.pieces_matrix = server_data.pieces_matrix;
        this.legal_moves = server_data.legal_moves;
        this.next_to_go = server_data.next_to_go;
        this.check = server_data.check;
        this.just_reset = false

        // console.log("server_data")
        // console.log(server_data)

        // console.log("server_data.game_over_data")
        // console.log(server_data.game_over_data)

        this.game_over = server_data.game_over_data.over;
        delete server_data.game_over_data.over;
        this.over_data = server_data.game_over_data;

        this.pieces_taken = server_data.pieces_taken;
        this.move_history = server_data.move_history;
        this.difficulty = server_data.difficulty;

        this.update_board_widget();

    }

    // these 3 functions are run when certain buttons are clicked

    // changes difficulty property and conveys the change to the server
    change_difficulty(new_difficulty) {
        this.difficulty = new_difficulty
        external_change_difficulty(new_difficulty);
    }

    // simply disables the game, the only option now it to restart
    concede_game() {
```

```
this.just_conceded = true
this.user_input_disabled = true;

this.over = true;
this.highlighted_squares = []
this.selected_from_vector = null
this.update_board_widget()
}

// this function resets the board properties to the initial game data
and then informs the server of this change
reset_game() {
    let temp_difficulty = this.difficulty;
    this.update_from_server_data(INITIAL_GAME_DATA);
    this.difficulty = temp_difficulty;
    this.just_reset = true
    this.just_conceded = false
    this.update_board_widget()
    external_reset_game();
}

// this function handles implementing a user move
// it validates that the user can go
// it then updates board's properties with the new board data from the
server
async make_user_move(from_vector, to_vector) {
    assert(
        this.next_to_go == "W",
        "User must be next to go in order to implement a user move"
    )
    // console.log("make_user_move called")
    let server_data = await
external_implement_user_move_and_update(from_vector, to_vector);
    // console.log("updating the board with this server data")
    // console.log({server_data})
    this.update_from_server_data(server_data, true)

}

// this function implements a computer move
async make_computer_move() {
    console.log("make_computer_move called")

    // validates that it is the computer's turn
    assert(
        this.next_to_go == "B",
        "User must be next to go in order to implement a user move"
    )
}
```

```
// created a promise to wait some amount of time (1 sec)
const wait_before_displaying.promise = new Promise((resolve) =>
setTimeout(resolve, 1000))

// const wait_a_second.promise = new Promise((resolve) =>
setTimeout(resolve, 0))

// gets the new server data after computer move
let server_data = await
external_implement_computer_move_and_update();
// console.log({server_data})

// get and remove move special data
let move_description = server_data["computer_move_description"];
delete server_data["computer_move_description"];
// console.log({ move_description })

// move description unpacked
let [from_vector, movement_vector] = move_description.move
let resultant_vector = add_vectors(from_vector, movement_vector)

// no change made for a second to the user can see the board after
their move
await wait_before_displaying.promise

// if the game is reset or conceded in that time, abort the
function
if (this.just_reset || this.just_conceded) {
    return null
}

// the relevant highlighting is applied to show the computer's
move
this.selected_from_vector = from_vector
this.possible_to_vectors = [resultant_vector,]
this.update_board_widget()

// the move is highlighted for 0.8 seconds before the new board
state is shown
const wait_to_show_move_highlights = new Promise((resolve) =>
setTimeout(resolve, 800));
await wait_to_show_move_highlights

// again, abort function if game reset / conceded in that time
if (this.just_reset || this.just_conceded) {
    return null
}
```

```
// console.log(`Implementing computer move:`)
// console.log(JSON.stringify(move_description))
// console.log(`Updating board to:`)
// console.log(JSON.stringify(server_data.pieces_matrix))

// show the new board positions after the computer's move
this.update_from_server_data(server_data, true);

// console.log("move_description.taken_piece")
// console.log(move_description.taken_piece)

// if (arrays_are_equal(move_description.taken_piece, [null,
null])) {
//     alert(`Computer moved ${move_description.moved_piece[1]}
from ${move_description.from_square} to ${move_description.to_square}`)
// }
// else {
//     alert(`Computer moved ${move_description.moved_piece[1]}
from ${move_description.from_square} to ${move_description.to_square}
taking ${move_description.taken_piece}`)
// }

}

// this function choreographs implementing the user move and then the
computer move
// if checks if the game is over after each one
async user_move_and_computer_move_cycle(from_vector, to_vector) {
    await this.make_user_move(from_vector, to_vector)

    if (this.game_over) {
        this.handle_game_over()
        return null
    }

    // if (this.check) {
    //     alert("CHECK");
    // }

    await this.make_computer_move()

    if (this.game_over) {
        this.handle_game_over()
        return null
    }
}
```

```
// if (this.check) {
//     alert("CHECK");
// }

// the user can now access the board to input another move
this.user_input_disabled = false

}

// this function displays the appropriate output when the game is over
handle_game_over() {
    // make sure the game is over
    assert(this.game_over)
    let winning_player = this.over_data.winning_player;
    let victory_classification =
this.over_data.victory_classification;

    // manually set the title message to show this
    let msg;
    switch (victory_classification) {
        case "checkmate":
            msg = (winning_player == 1) ? "Checkmate: congratulations, you won!" : "Checkmate: you lost, better luck next time"
            break
        case "stalemate board repeat":
            msg = "Stalemate: Threefold Repetition (Draw)"
            break
        case "stalemate no legal moves":
            msg = "Stalemate (Draw)"
            break
        default:
            throw new error("Invalid victory classification")
    }

    // update the title message
    this.update_main_title_widget(msg);

    // the board remains disabled

    // alert(msg)

    // this.reset_game();
}

// this handler function responds to the user clicking a certain square
```

```
handle_square_click(vector) {

    // if user input is disabled, ignore the click
    if (this.user_input_disabled) {
        // alert("board disabled")
        // console.log("board disabled")
        return null;
    }

    // else disable the board and decide what to do
    this.user_input_disabled = true;

    // check if they have clicked a green highlighted square they
    could move to,
    // let from_square_already_selected = this.selected_from_vector
    !== null;
    let is_valid_move =
two_d_array_contains_sub_array(this.possible_to_vectors, vector);
    // console.log(`Checking
if [${vector}] in ${JSON.stringify(this.possible_to_vectors)} --> result was ${is_valid_move}`);

    // if they have, cause the appropriate user then computer move
    if (is_valid_move) {
        // async function call
        // console.log("about to call make_user_move")
        // this.make_user_move(this.selected_from_vector, vector);
        this.user_move_and_computer_move_cycle(this.selected_from_vector, vector);
    }

    // else if not valid or no piece already selected then reselect
    else {
        // get piece at vector clicked
        let [i, j] = vector;
        let [row, col] = [7 - j, i];
        let [color, _] = this.pieces_matrix[row][col];

        // if the piece is one of the users then highlight red
        if (color == this.next_to_go) {
            this.selected_from_vector = vector;
        }
        else {
            this.selected_from_vector = null;
        }

        // then iterate through the legal moves and identify any
        squares that the user could move the selected piece to
    }
}
```

```
// if there are any, highlight green
// if (color == this.next_to_go) {
if (color == "W") {
    let position_vector; let movement_vector; let
resultant_vector;
    this.possible_to_vectors = [];
    for (let i in this.legal_moves) {
        [position_vector, movement_vector] =
this.legal_moves[i];
        if (arrays_are_equal(position_vector, vector)) {
            resultant_vector = add_vectors(position_vector,
movement_vector);
            this.possible_to_vectors.push(resultant_vector);
        }
    }
}
// update the board to show any highlighting changes
this.update_board_widget()
}
this.user_input_disabled = false
}

// this method returns an array of position vectors of squares and
there color
// it highlights the selected from vector red and all selected to
vectors green
get_highlighted_squares() {
    // returns a 2d array of vector and then color
    let highlighted_squares = [];
    if (this.selected_from_vector !== null) {
        highlighted_squares.push([
            this.selected_from_vector,
            "red"
        ]);
        for (let i in this.possible_to_vectors) {
            highlighted_squares.push([
                this.possible_to_vectors[i],
                "green"
            ]);
        }
    }
    // console.log({highlighted_squares})
    return highlighted_squares;
}
}
```

The main functions of this file are:

- Define 5 asynchronous functions that can be used to communicate with the server. To do this I created a general promised factory to ensure that I kept repeat logic to a minimum
- Define the Chess_Board class.
 - This will contain data about the current chess board state in public properties. These properties will be accessed by the DOM manipulation methods.
 - It will contain handler functions for the 4 main inputs, square click, reset or concede button click and difficulty radio button change
 - It uses internal variables to keep track of the 2 square clicks needed to input a move and to produce the relevant highlighting in the process
 - It will be able to orchestrate the execution of a user's move and then the computer move. This is in addition to checking if the game is over, and if so producing the correct output, after each move

I also used a small file called vector.js that contained some logic around adding vectors:

```
// the following functions execute add or subtract operations on a pair of
// vectors
function add_vectors(v1, v2) {
    // v1 + v2
    return [
        v1[0] + v2[0],
        v1[1] + v2[1],
    ]
}

function subtract_vectors(v1, v2) {
    // v1 - v2
    return [
        v1[0] - v2[0],
        v1[1] - v2[1],
    ]
}

// use arrays are equal method instead
// function vectors_are_equal(v1, v2) {
//     return JSON.stringify(v1.map(Number)) ==
JSON.stringify(v2.map(Number))
// }
```

The only remaining component of the frontend was the CSS:

```
#board {
    width: 80vmin;
    height: 80vmin;

    /* width: 800px;
```

```
height: 800px; /*  
/* margin-top: 10vmin;  
margin-bottom: 5vmin; */  
  
/* margin-top: 0;  
margin-bottom: 0; */  
  
/* margin-top: 2vmin;  
margin-bottom: 2vmin; */  
  
margin-left: auto;  
margin-right: auto;  
  
  
display: flex;  
flex-wrap: wrap;  
  
border: 2px solid black;  
border-collapse: collapse;  
  
}  
  
.square {  
width: 10vmin;  
height: 10vmin;  
/* width: 100px;  
height: 100px; */  
display: flex;  
align-items: center;  
justify-content: center;  
font-size: 9vmin;  
/* text-shadow: -1px -1px 0 #000, 1px -1px 0 #000, -1px 1px 0 #000,  
1px 1px 0 #000; */  
}  
  
  
#top_title {  
/* height: 8vh; */  
height: 6vmin;  
width: 80vw;  
  
font-size: 6vmin;  
  
margin: 0 auto;  
  
text-align: center;
```

```
/* border: 2px solid red; */

padding-top: 0;
margin-top: 0;
border-top: 0;
/* clip-path: polygon(0 0, 100% 0, 100% 30%, 0 100%); */

}

body {
    width: 100%;
    height: 100%;
    margin: 0 auto;
    text-align: center;
}

/* .button_wrapper { */
.centered {
    display: flex;
    align-items: center;
    justify-content: center;
    flex-direction: column;
}

.option_button_container {
    display: flex;
    justify-content: space-between;
    /* width: 100vmin; */
    width: 60vmin;
    /* margin-top: 2vh; */
}

.option_button {
    /* height: 6vmin;
    width: 8vmin;
    min-height: 4.5ch;
    min-width: 8ch;
    font-size: 5vmin; */

    /* height: 6vmin;
    width: 8vmin;
    min-width: 14ch;
    font-size: 5vmin; */

    height: 6vmin;
    width: 8vmin;
    min-width: 8ch;
```

```
font-size: 4vmin;

text-align: center;
/* font-size: 5vmin; */
/* margin-top: 2vh; */
/* border: 2px solid green; */

/* margin-left: 5vw;
margin-right: 5vw; */
}

.option_button:first-child {
    margin-left: 0;
    /* margin-right: 2ch; */
}
}

.option_button:last-child {
    /* margin-left: 2ch; */
    margin-right: 0;
}
}

.pieces_taken_table th, td {
    width: 35vw;
    border: 2px solid black;
    font-size: 28px;
}

.spacer {
    display: block;
    /* height: 5vh; */
    /* height: 3vmin; */
    height: 2.5vmin;
    width: 100%;
    margin: 0;
    padding: 0;
    border: none;
    overflow: hidden;
    font-size: 0;
    line-height: 0;
}
/*
.small_text {
    font-size: 4vmin;
} */
```

```
#difficulty_form form {
    display: inline-flex;
    flex-direction: row;
}

.difficulty_form div{
    /* width: 80vh; */

    width: 80vmin;
    /* width: 100vmin; */

    /* background-color: orange; */
    /* border: 2px solid orange; */
}

.difficulty_form label {
    margin-right: 6vw;
    width: 10vh;
    height: 5vh;
    /* font-size: 4vh; */
    font-size: 4vmin;
    /* margin: 5vh, auto; */
}

h1 {
    font-size: 4vmin;
    padding-top: 0;
    padding-bottom: 0;
    margin-top: 0;
    margin-bottom: 0;
    border-top: 0;
    border-bottom: 0;
}

#which_pieces_taken_white {
    background-color: #66443a;
    font-size: 4vmin;
    color: white;
    text-shadow: -2px -2px 0 #000,
        2px -2px 0 #000,
        -2px 2px 0 #000,
        2px 2px 0 #000;
}

#which_pieces_taken_black {
    background-color: #fae09f;
    font-size: 4vmin;
```

```
color: black;  
/* text-shadow: -1px -1px 0 #ffff,  
   1px -1px 0 #ffff,  
   -1px 1px 0 #ffff,  
   1px 1px 0 #ffff; */  
  
/* body {background-color: red} */
```

It took a lot of experimentation but the resulting CSS is able to dictate the layout of each widget on the board, the widgets relative dimensions and its aesthetics such as color and text size.

The other design objective that the CSS achieves is ensuring that the webpage works on a range of aspect ratios.

This was done through the use of CSS properties like:

- Vmin: percentage of height or width, whichever is smaller
- Vmax: percentage of height or width, whichever is larger
- Vh: percentage of view height
- Vw: percentage of view width.

Demonstrating the end result

Here is the final result:



Scrolling down...

The screenshot shows the same chessboard and interface as the previous image, but with additional information at the bottom. It includes a "Change Difficulty: (AI thinking time)" section with radio buttons for Trivial (1 sec), Hard (10 sec), Easy (2 sec), Challenge (15 sec), Medium (5 sec), Legendary (60 sec), and Extreme (30 sec). The "Medium (5 sec)" option is selected. Below this is a "Pieces Taken:" table:

White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

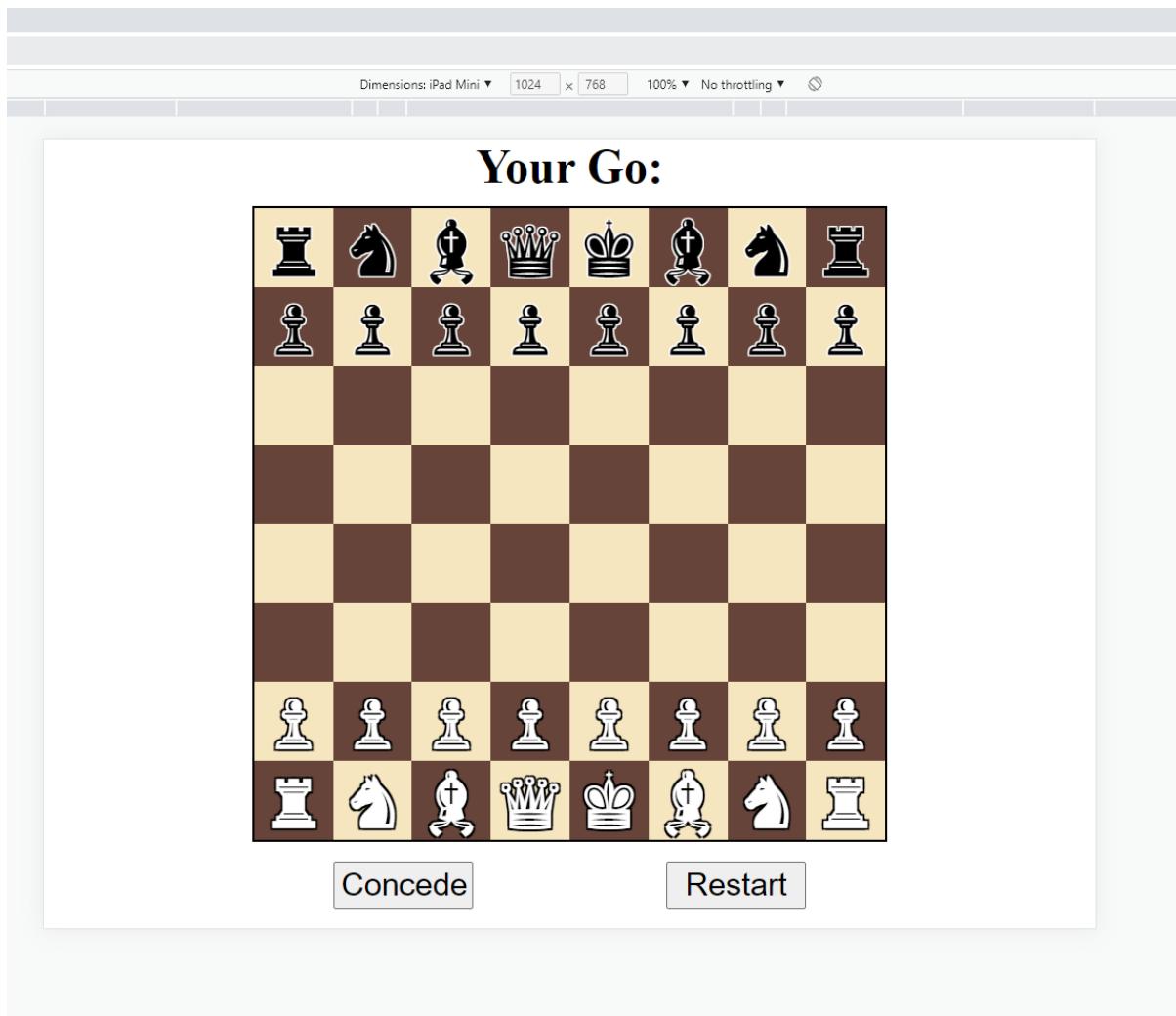
At the bottom is a "Moves History:" table:

White previous moves:	Black previous moves:
-	-

The application window has a title bar "Chess Game" and a status bar at the bottom showing "127.0.0.1:5000". The taskbar at the bottom of the screen shows various icons for other applications like Google Chrome, Microsoft Word, and File Explorer.

The above 2 screenshots show how the page looks from a landscape orientation with a laptop.

I will now show some other aspect ratios using chrome dev tools:



The screenshot shows a chessboard with white pieces on light squares and black pieces on dark squares. The board is set up with standard chess pieces: two rooks, two knights, two bishops, a queen, a king, and eight pawns. Below the board are two buttons: "Concede" on the left and "Restart" on the right. Above the board, the browser's dimensions are displayed as 1024 x 768 at 100% scale with no throttling.

Change Difficulty: (AI thinking time)

- Trivial (1 sec) Easy (2 sec) Medium (5 sec)
- Hard (10 sec) Challenge (15 sec) Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

Moves History:

White previous moves:	Black previous moves:
-	-

Above is the webpage viewed in a landscape orientation with the iPad Air.

Your Go:

Concede Restart

Change Difficulty: (AI thinking time)

Trivial (1 sec) Easy (2 sec) Medium (5 sec)
 Hard (10 sec) Challenge (15 sec) Extreme (30 sec)
 Legendary (60 sec)

Pieces Taken:

White Pieces Taken	Black Pieces Taken
No Pieces	No Pieces

Dimensions: iPhone 6/7/8 Plus ▾ 414 × 736 100% ▾ No throttling ▾

Concede Restart

Change Difficulty: (AI thinking time)

Trivial (1 sec) Easy (2 sec) Medium (5 sec)
 Hard (10 sec) Challenge (15 sec) Extreme (30 sec)
 Legendary (60 sec)

Pieces Taken:

White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
=	-

Moves History:

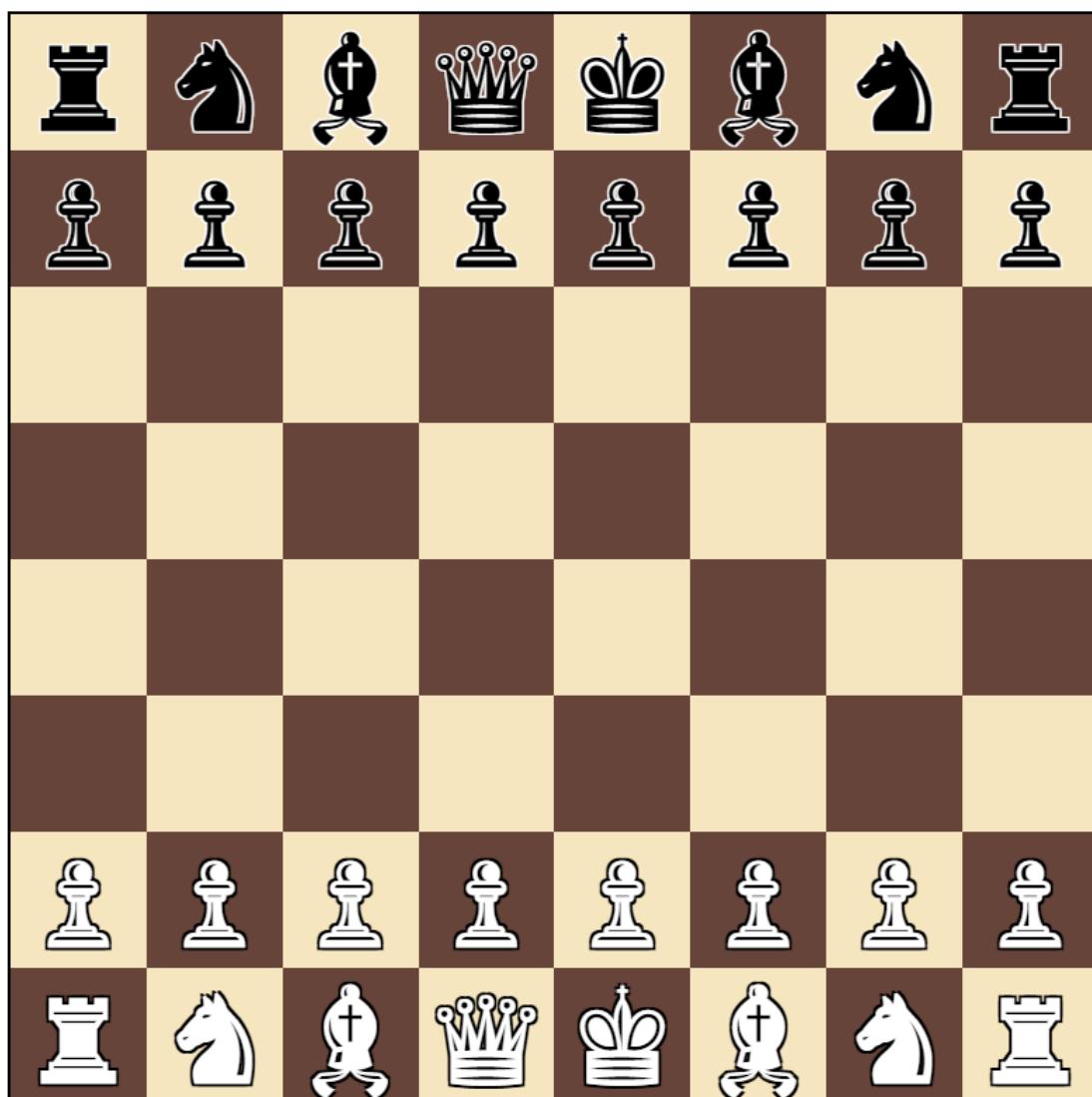
White previous moves:	Black previous moves:
-	-

Above you can see the program running in a portrait iPhone 8. As can be seen, some of the elements are overly large. For instance the font size of the pieces taken and moves history tables are very large. However, all the elements are visible and no are too large or small to see or interact with. This means that even though the styles do not perfectly translate to every device, the user interface is still perfectly functional and just as usable on a wide range of devices and aspect ratios.

Example Play

Here are some screenshot of the game as I play the computer:

I have opened a new game on medium difficulty:

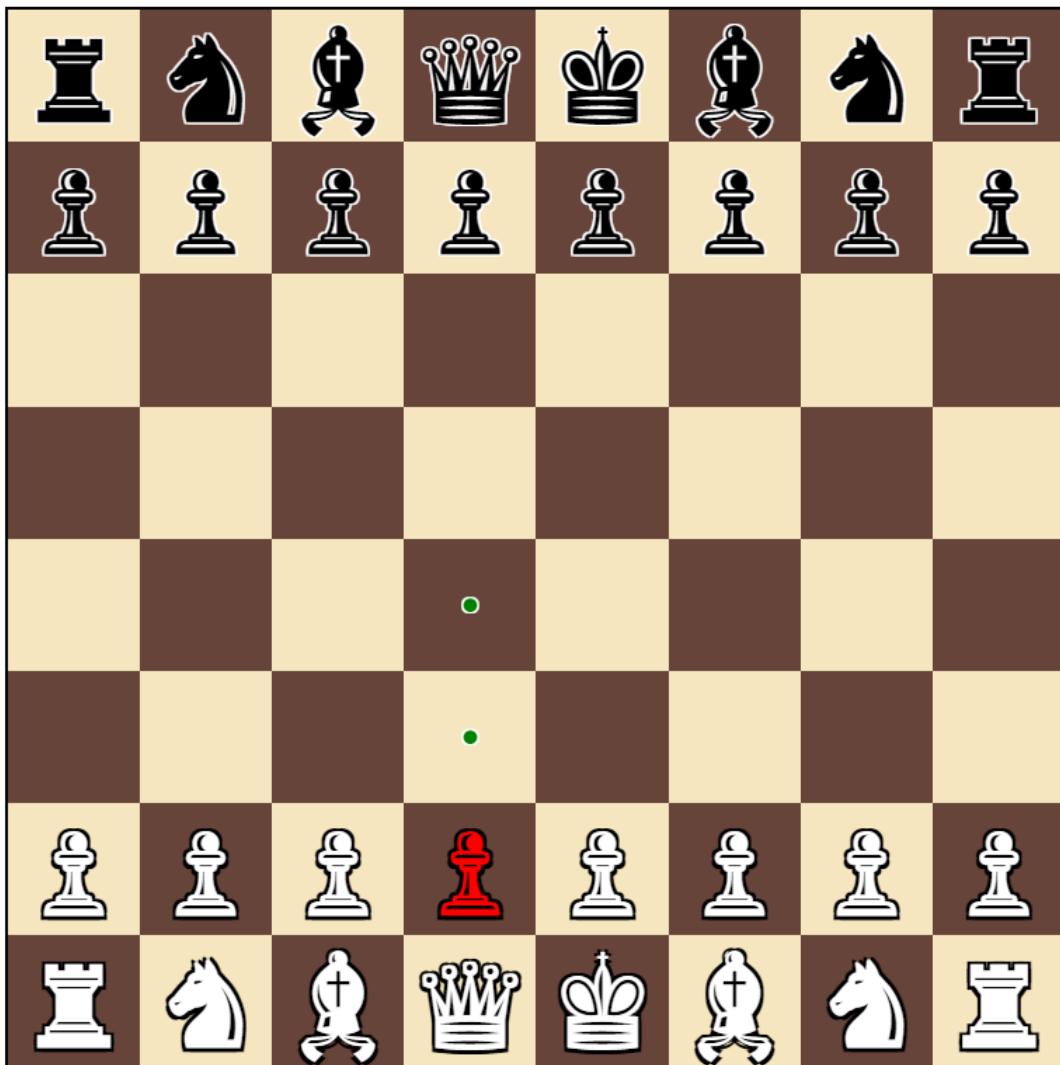


Concede

Restart

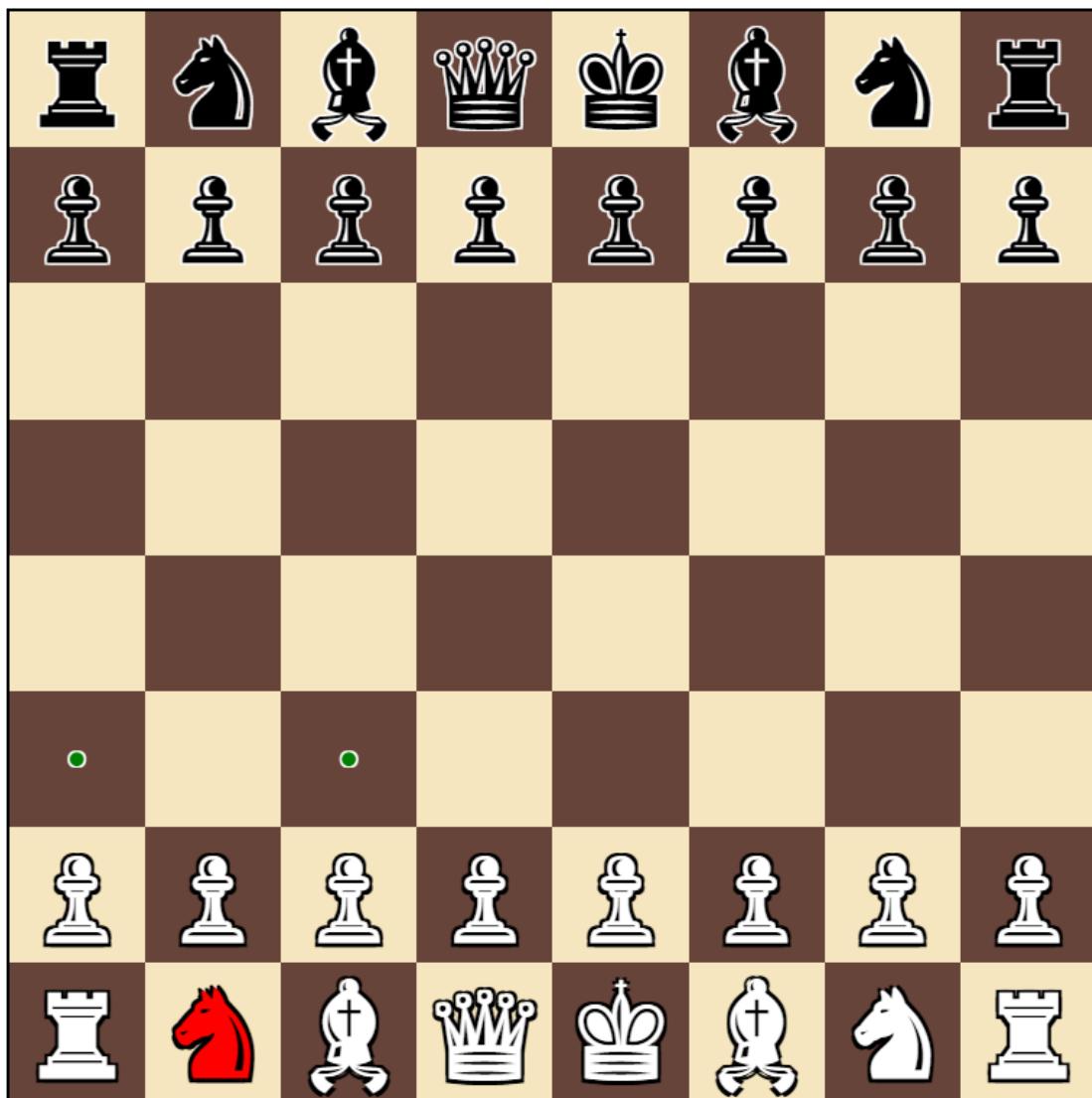
We can see that the computer has prompted me that it is my go. I will not click the white pawn in front of the queen:

Your Go:

[Concede](#)[Restart](#)

If I click off this square onto another square I get back to the blank board I started with. If I click some of my other pieces they are also highlighted red. If they can make any moves, these are highlighted in green.

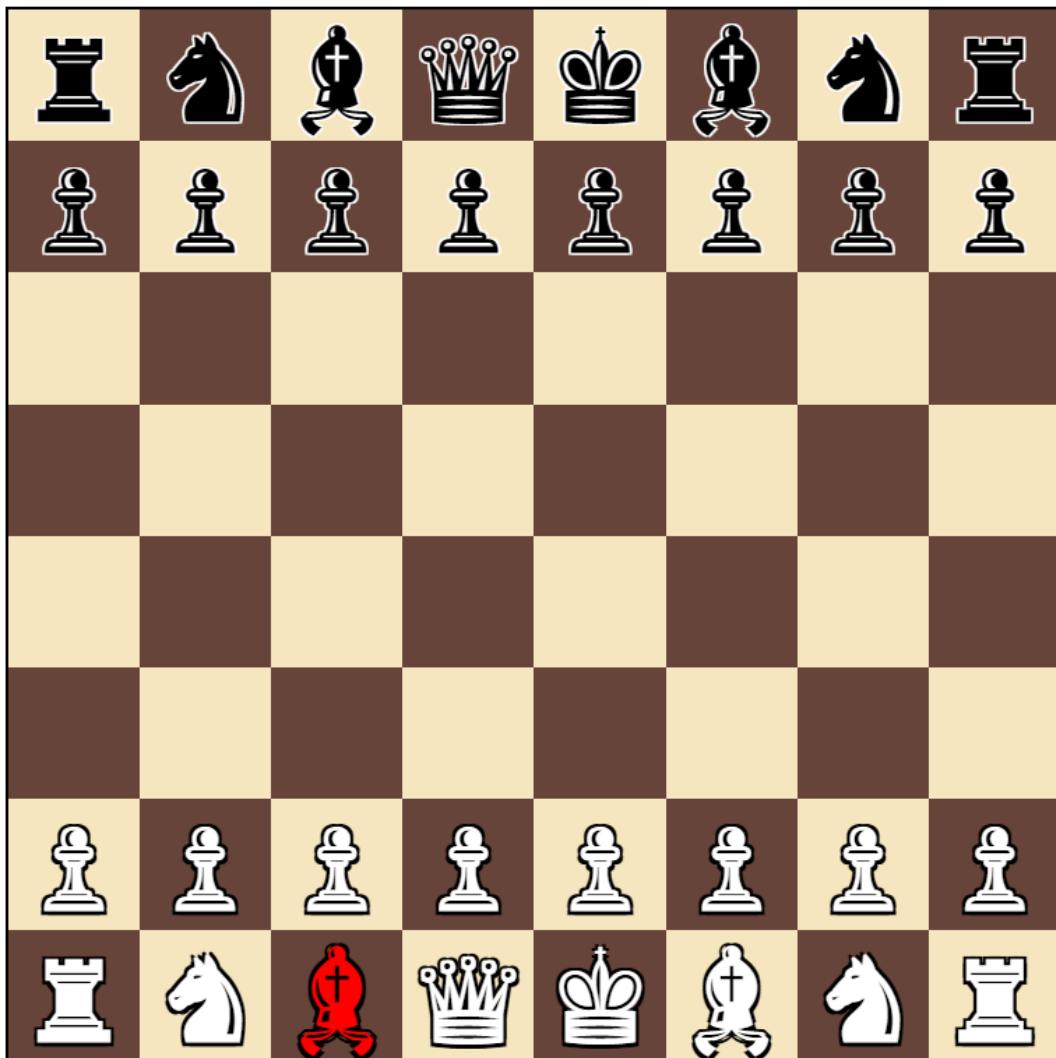
Your Go:



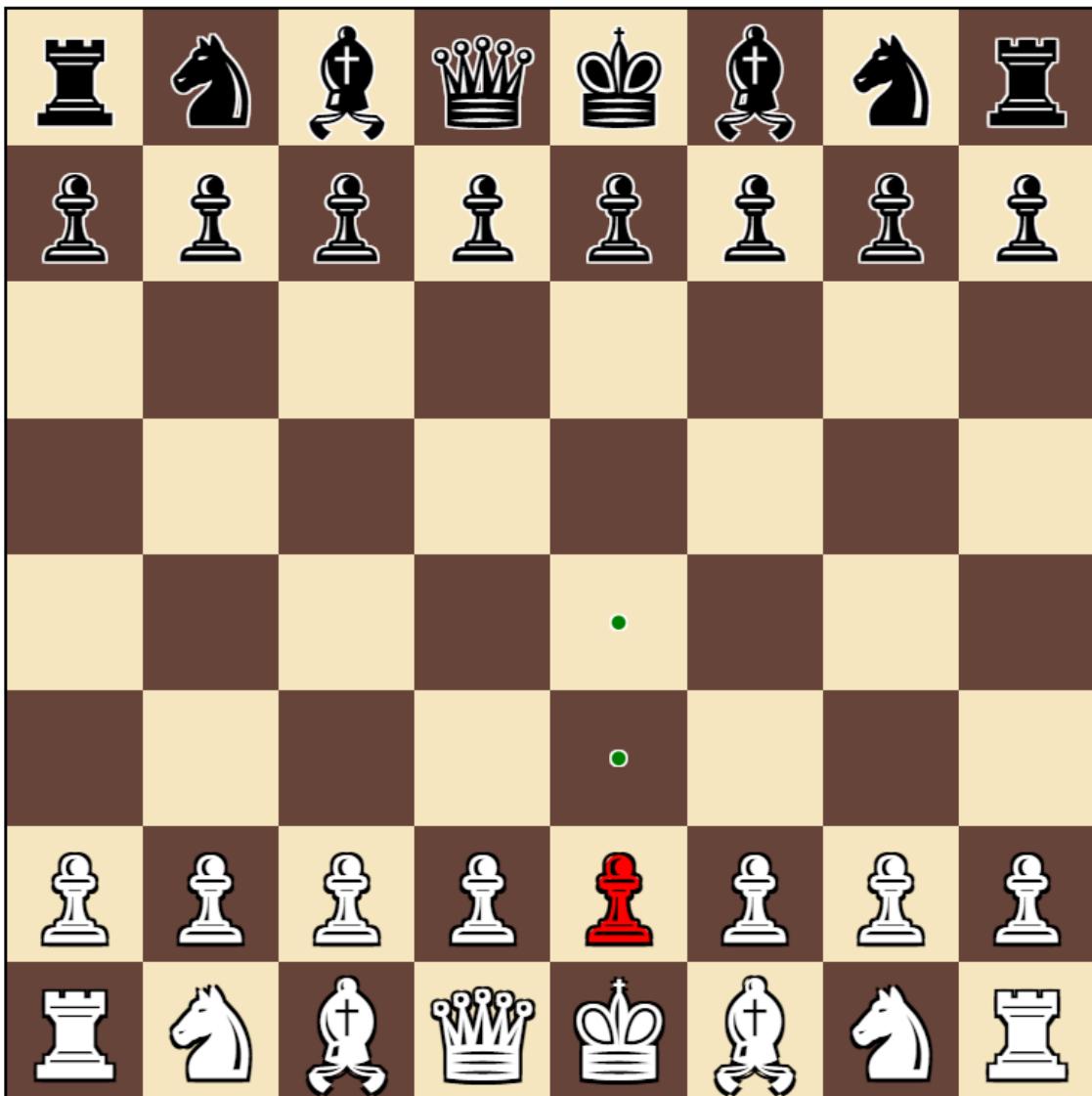
Concede

Restart

Your Go:

[Concede](#)[Restart](#)

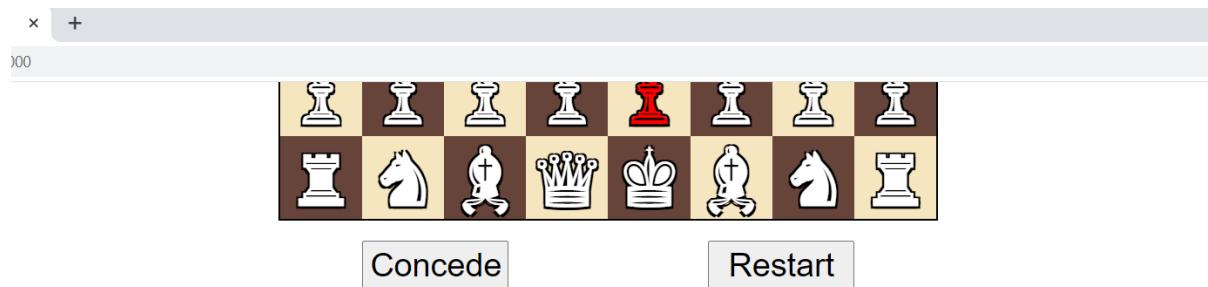
Your Go:

ConcedeRestart

Clicking other squares including the computers squares remove the highlighting but doesn't cause any unexpected behaviours (such as moving for the computer or moving my piece to that square).

I will click the front most green square. I have set the difficulty to be higher to slow the program down and allow me to easily capture what happens when the computer moves. Changing the difficulty has not affected the highlighting.

And here is a screenshot of the below widgets before I move:



Change Difficulty: (AI thinking time)

- Trivial (1 sec)
- Easy (2 sec)
- Medium (5 sec)
- Hard (10 sec)
- Challenge (15 sec)
- Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

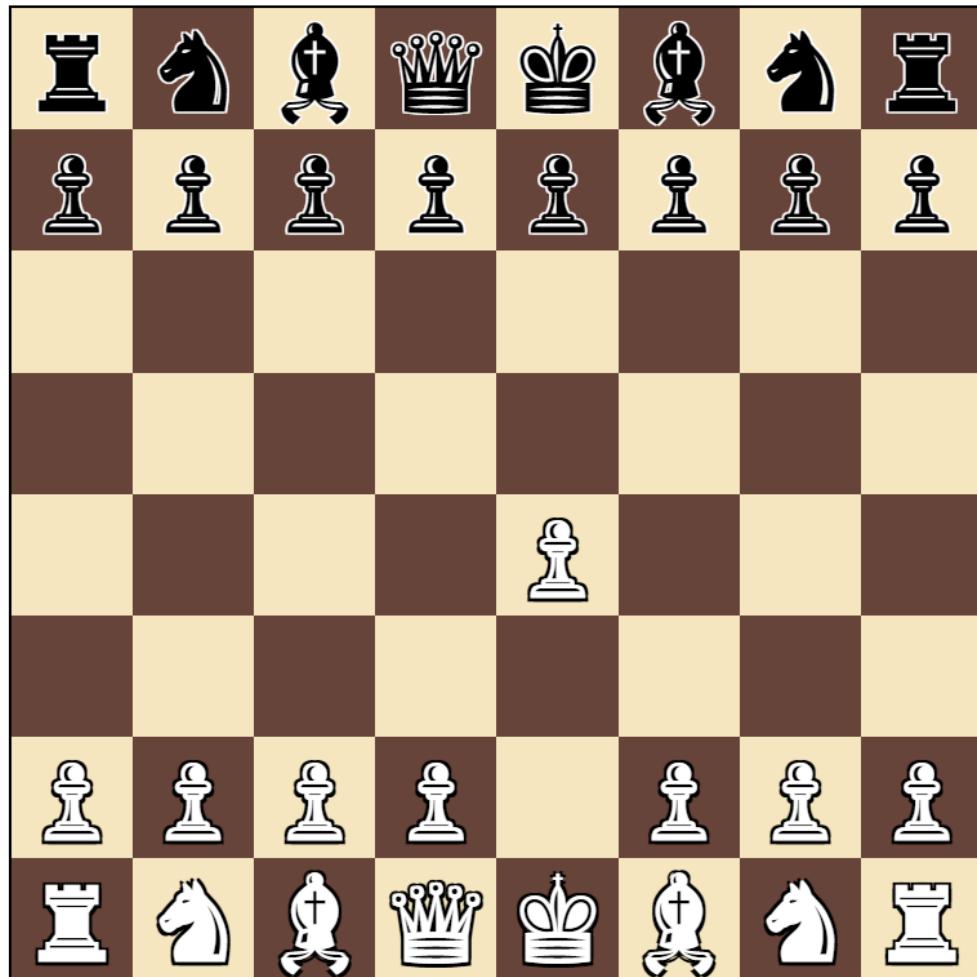
Moves History:

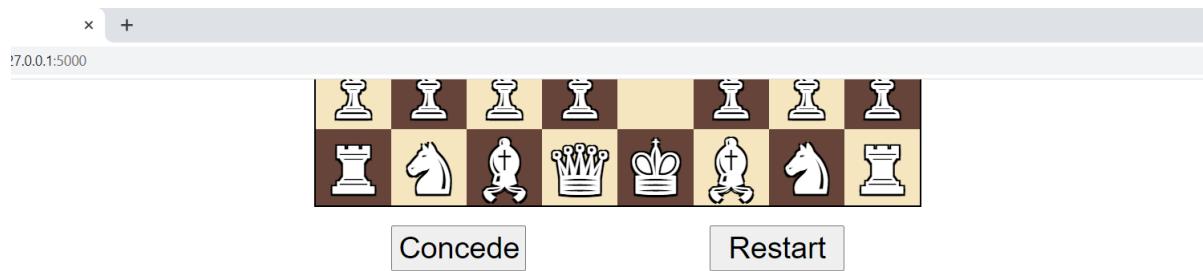
White previous moves:	Black previous moves:
-	-

Then I clicked the front most green square and saw:

That my pawn has moved forwards, the title element had changed and the move history had changed.

Computer's Go... (please wait)

[Concede](#)[Restart](#)

**Change Difficulty: (AI thinking time)**

- Trivial (1 sec)
- Easy (2 sec)
- Medium (5 sec)
- Hard (10 sec)
- Challenge (15 sec)
- Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

Moves History:

White previous moves:	Black previous moves:
1: WP from E2 to E4	-

I then saw that after my move, the computer made its move. This was shown by the following highlighting for 0.8 seconds

Computer's Go... (please wait)



Concede

Restart

Then the board updated again to show the computer's move and the main title told me it was my turn:

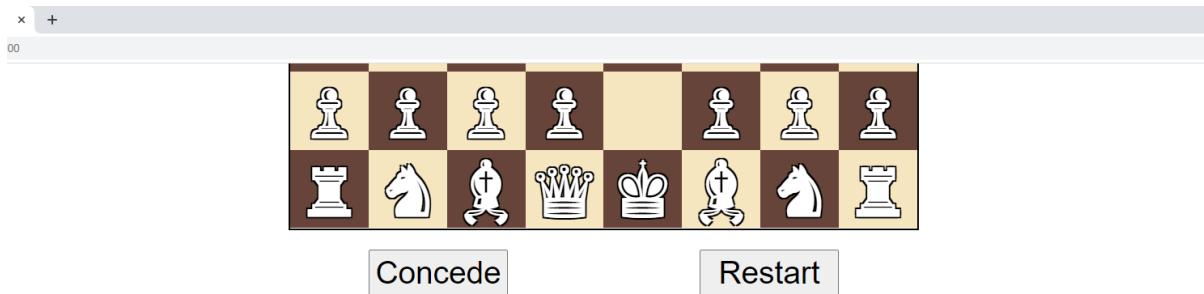
Your Go:



Concede

Restart

The move history widget had updated:

**Change Difficulty: (AI thinking time)**

- Trivial (1 sec)
- Easy (2 sec)
- Medium (5 sec)
- Hard (10 sec)
- Challenge (15 sec)
- Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

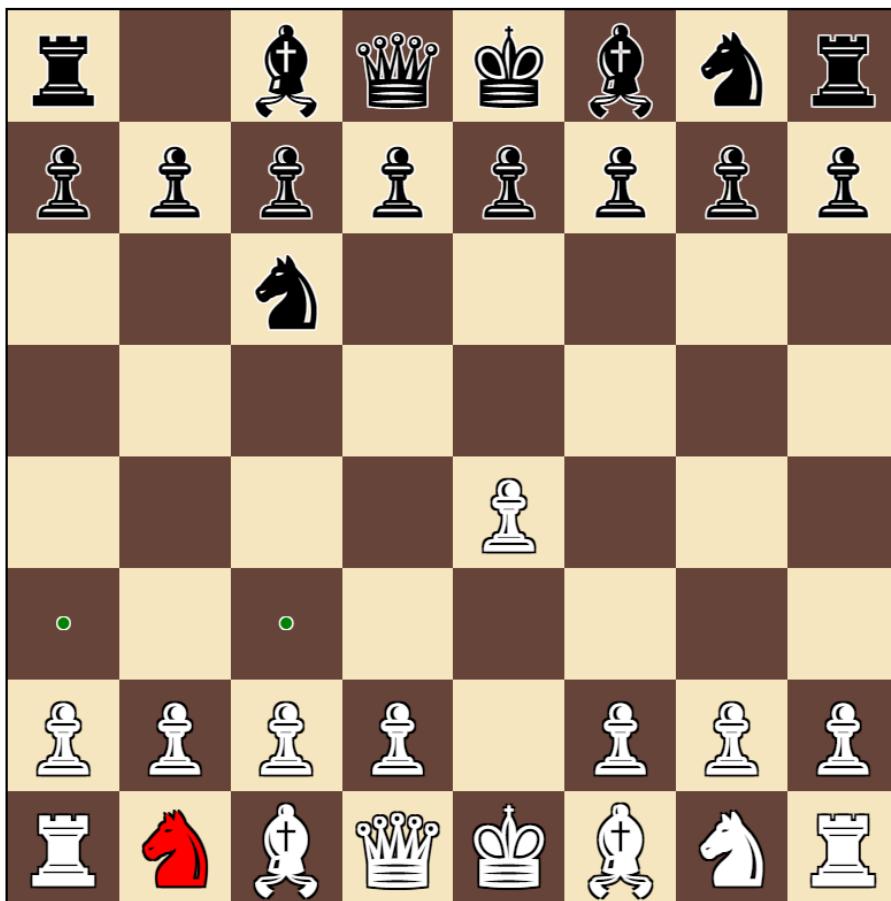
White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

Moves History:

White previous moves:	Black previous moves:
1: WP from E2 to E4	2: BN from B8 to C6

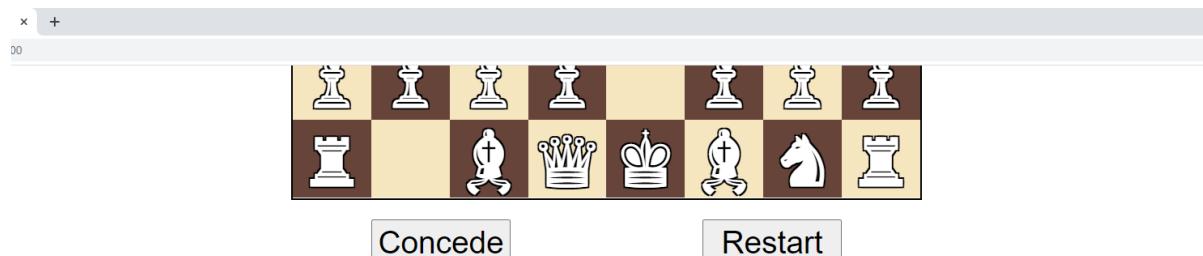
I was able to click to move another piece, one of my knights forward:

Your Go:

[Concede](#)[Restart](#)

Computer's Go... (please wait)

[Concede](#)[Restart](#)

**Change Difficulty: (AI thinking time)**

- Trivial (1 sec)
- Easy (2 sec)
- Medium (5 sec)
- Hard (10 sec)
- Challenge (15 sec)
- Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

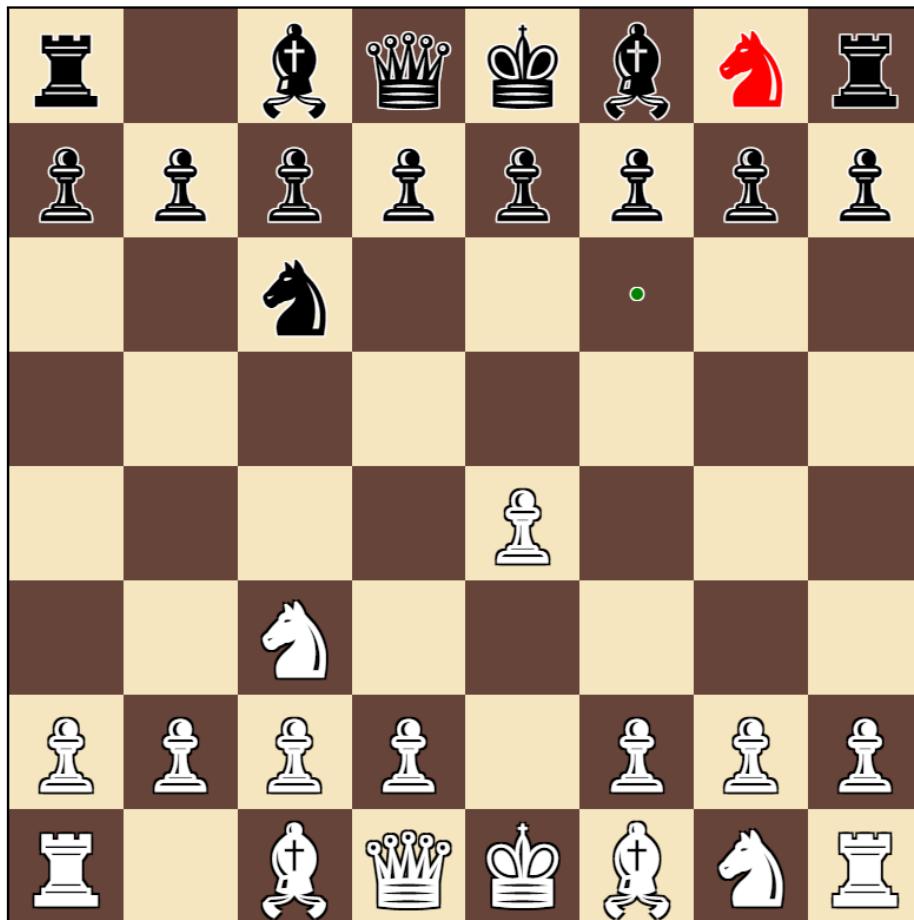
White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

Moves History:

White previous moves:	Black previous moves:
1: WP from E2 to E4	2: BN from B8 to C6
3: WN from B1 to C3	-

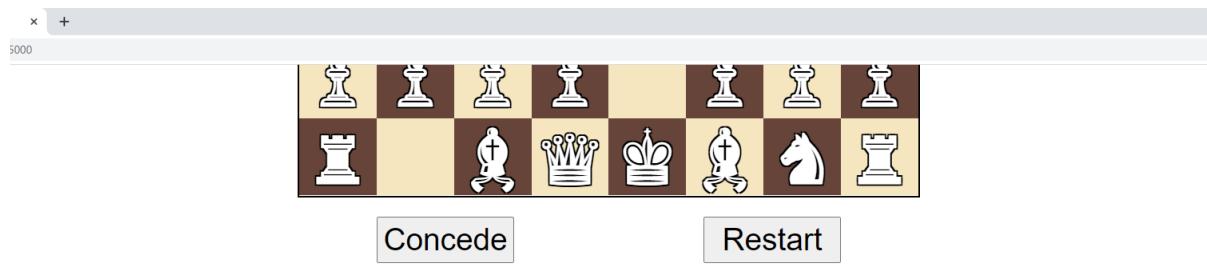
Then the computer moved again:

Computer's Go... (please wait)

[Concede](#)[Restart](#)

Your Go:

[Concede](#)[Restart](#)

**Change Difficulty: (AI thinking time)**

- Trivial (1 sec) Easy (2 sec) Medium (5 sec)
- Hard (10 sec) Challenge (15 sec) Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

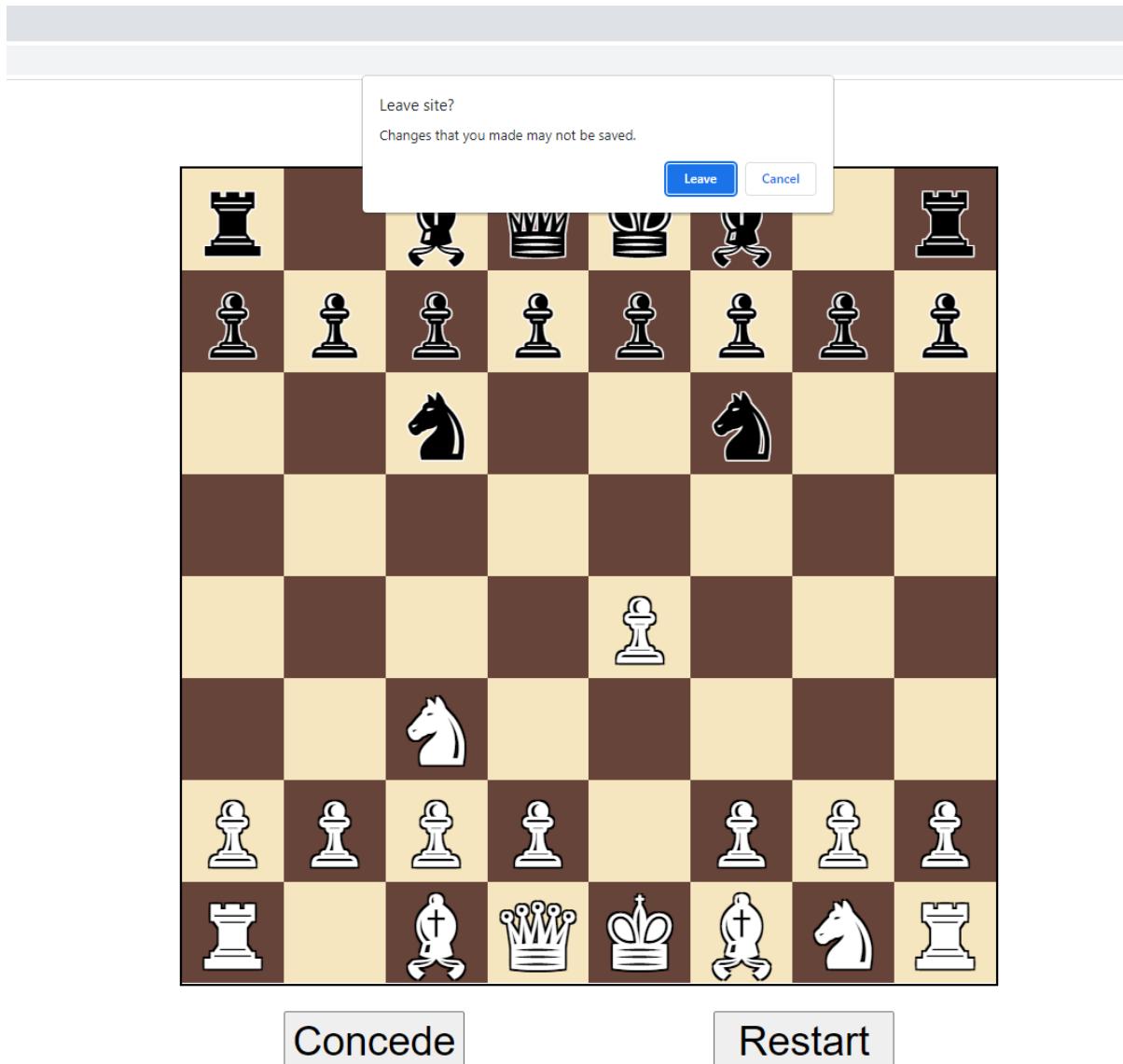
White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

Moves History:

White previous moves:	Black previous moves:
1: WP from E2 to E4	2: BN from B8 to C6
3: WN from B1 to C3	4: BN from G8 to F6

I was then able to close the tab and reopen it to the exact same game.

I saw this notice:



Then the tab closed and I reopened it on the same chess game. It realises the message may be a little misleading but I am not sure how to change it.

I was then able to click the concede button.

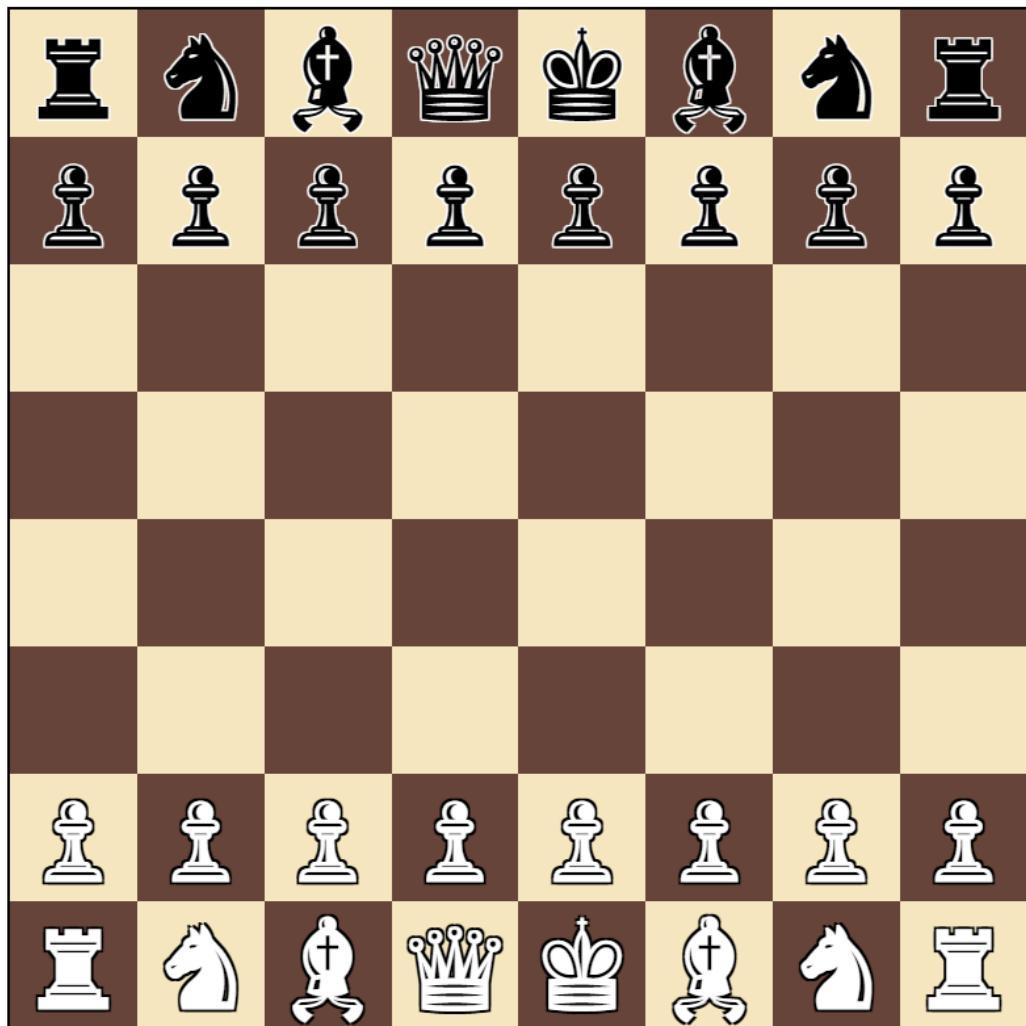
Game Over: You Conceded

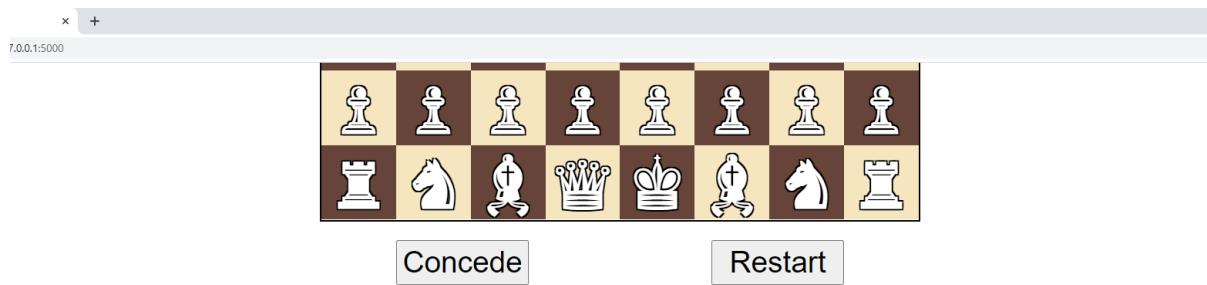
[Concede](#)[Restart](#)

This caused the main message at the top to update and the board to become non-interactable.

I was able to change the game's difficulty and then press restart to revert the game to the starting board positions and a blank move history:

Your Go:

[Concede](#)[Restart](#)



Change Difficulty: (AI thinking time)

- Trivial (1 sec)
- Easy (2 sec)
- Medium (5 sec)
- Hard (10 sec)
- Challenge (15 sec)
- Extreme (30 sec)
- Legendary (60 sec)

Pieces Taken:

White Pieces Taken	Black Pieces Taken
No Pieces Taken	No Pieces Taken
-	-

Moves History:

White previous moves:	Black previous moves:
-	-

The new difficulty I set before restarting the game was preserved

Testing:

To test the program I first added to my automated tests to add functionality to test some of the additional modules and features created. I then tested the program by running the website and playing chess against it. By testing the program as if I were a user, using the program I was able to identify various issues. I then created unit tests to identify some of the issues and then fixed my code to prevent the issue. I was then able to run the unit test again to verify that the issue had been solved.

Automated unit tests:

Firstly I created additional minimax tests and combined together all minimax tests into one test file:

```
# this test is responsible for testing various mutations of the minimax
function and how they play, it is not a data driven test

# imports of external modules.
from random import choice as random_choice, randint
# from itertools import product as iter_product
import pickle
import unittest
# from functools import wraps
import multiprocessing
import os
import csv
from time import perf_counter
```

```
# imports of local modules
from chess_game import Game
# from chess_functions import Board_State
# from minimax import Move_Engine
from move_engine import Move_Engine_Timed, Move_Engine_Prime
from chess_functions import random_board_state

# from assorted import ARBITRARILY_LARGE_VALUE
# from board_state import Board_State
# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given
function a given number of times
# def repeat_decorator_factory(times):
#     def decorator(func):
#         @wraps(func)
#         def wrapper(*args, **kwargs):
#             for _ in range(times):
#                 func(*args, **kwargs)
#         return wrapper
#     return decorator

# this is a utility function that maps a function across an iterable but
also converts the result to a list data structure
def list_map(func, iter):
    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a
message
# it converts pieces to symbols if they are not none
def map_pieces_matrix_to_symbols(pieces_matrix):
    return list_map(
        lambda row: list_map(
            lambda square: square.symbol() if square else None,
            row
        ),
        pieces_matrix
    )

# this functions updates a CSV file with the moves and scores of a chess
game for graphical analysis in excel
# each move in the game causes a new row to be added
def csv_write_move_score(file_path, move_counter, move, score):
    # convert move to a pair of squares
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector
```

```
from_square = position_vector.to_square()
to_square = resultant_vector.to_square()

# # if file doesn't exist, create is and add the headers
# if not os.path.exists(file_path):
#     with open(file_path, "w", newline="") as file:
#         writer = csv.writer(file, delimiter=",")
#         writer.writerow(("from_square", "to_square", "score"))

# add data as a new row
with open(file_path, "a", newline="") as file:
    writer = csv.writer(file, delimiter=",")
    # writer.writerow(("Move_counter", "B_score", "from_square",
# "to_square", "W_score"))
    writer.writerow((move_counter, -score, from_square, to_square,
score))

# this function is used to time another function to allow for performance
testing
def time_function(function):
    start = perf_counter()
    result = function()
    end = perf_counter()
    time_delta = end - start
    return result, time_delta

# this function aims to save games to a file as part of an unfinished
feature
def save_games(games: tuple, file_path):
    with open(file_path, "wb") as file:
        file.write(
            pickle.dumps(
                games
            )
    )

# this method aims to load a game from a file as part of another
unfinished feature
def load_games(file_path):
    if not os.path.exists(file_path):
        return dict()
    with open(file_path, "rb") as file:
        return pickle.loads(
            file.read()
    )
```

```
# the below function contains the logic to perform a test between 2
minimax bots to assert that the good bot is better

# this contains the majority of the logic to do a bot vs bot test with the
game class
# it is a component as it isn't the whole test
def minimax_test_component(description, good_bot, bad_bot,
success_criteria, write_to_csv, csv_folder, csv_file_name,
load_game=False, save_game=False):
    # print(f"CALL minimax_test_component(description={description},
write_to_csv={write_to_csv})")
    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,
    # the test is designed to assert that good bot wins (and or draws in
some cases)

    # if the write to csv file is enabled
    # generate csv path
    if write_to_csv:
        # not sure why but the description sometimes contains an erroneous
colon, this is caught and removed
        # was able to locate bug to here, as it is a test I added a quick
fix
        # bug located, some description strings included them

        # create folders
        if not os.path.exists(csv_folder):
            os.makedirs(csv_folder)

        # create file with headers if it doesn't exist
        csv_path = f"{csv_folder}/{csv_file_name}.csv"

        # overwrite so it is blank
        with open(csv_path, "w", newline="") as file:
            writer = csv.writer(file, delimiter=",")
            writer.writerow(("Move_counter", "B_score", "from_square",
"to_square", "W_score"))

    # pickle_file_path = f"{csv_folder}/{csv_file_name}.games"
    # if load_game and os.path.exists(pickle_file_path):
    #     print("Attempting to load game")
    #     indexes = (-1, -2)
    #     games = load_games(pickle_file_path)
    #     for game in games:
    #         print(hash(game))
    #         print(game.board_state.next_to_go)
    #     for index in indexes:
```

```
#         try:
#             game = games[index]
#             assert game.board_state.next_to_go == "W", "next to go
wasn't user so use other game"
#             except Exception as e:
#                 print(e)
#                 continue
#             else:
#                 print(f"Using game {hash(game)} at index {index}")
#                 games = games[:index]
#         else:

#         create a new game
print("Creating new game")
game: Game = Game(echo=True)
# games = tuple()

# start a new blank game
# depth irrelevant as computer move function passed as parameter
print()
print(f"Beginning test game: {description}")

# def record_new_game(games, new_game):
#     games = tuple(list(games) + [new_game])
#     save_games(games, pickle_file_path)
#     return games

# if save_game:
#     games = record_new_game(games, game)

# keep them making moves until return statement breaks loop
while True:
    # print()
    # game.board_state.print_board()
    # print()
    # get move choice from bad bot

        # the bad bots move is implemented as the user as the user plays
as white (advantage)
        result, time_delta = time_function(
            lambda: bad_bot(game)
        )
        score, move_choice = result
        # print({"move_choice": move_choice})

        # serialised to is can be passed as a user move (reusing game
class)
        position_vector, movement_vector = move_choice
```

```
        resultant_vector = position_vector + movement_vector
        from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()

        # implement bad bot move and update csv
        # move, score = game.implement_user_move(from_square=from_square,
to_square=to_square, time_delta=time_delta)
        move, _ = game.implement_user_move(from_square=from_square,
to_square=to_square, time_delta=time_delta, estimated_utility=score)

        # games = record_new_game(games, game)

        if write_to_csv:
            csv_write_move_score(
                file_path=csv_path,
                move=move,
                score=game.board_state.static_evaluation(),
                move_counter=game.move_counter
            )

            # piece_moved =
game.board_state.get_piece_at_vector(resultant_vector)
            # print(f"Move {game.move_counter}: bad bot moved {piece_moved}
from {from_square} to {to_square} with a score perceived of {score}")

            # if game.board_state.color_in_check():
            #     print(f"CHECK: {game.board_state.next_to_go} in check")

            # see if this move causes the test to succeed or fail or keep
going

            # use the provided success criteria to determine if the game
should be over.
            success, msg, board_state = success_criteria(game,
description=description)
            # result = success_criteria(game, description=description)
            # print({"result": result})
            # success, msg, board_state = result

            if success is not None:
                # game.board_state.print_board()
                return success, msg, board_state

        # repeat for the good bot
        # providing good bot function, implement good bot move and update
csv
        move, score =
game.implement_computer_move(best_move_function=good_bot)
```

```
# games = record_new_game(games, game)

# update CSV
if write_to_csv:
    csv_write_move_score(
        file_path=csv_path,
        move=move,
        score=game.board_state.static_evaluation(),
        move_counter=game.move_counter
    )
# unpack move in terms of to and from squares
position_vector, movement_vector = move
resultant_vector = position_vector + movement_vector
# piece_moved =
game.board_state.get_piece_at_vector(resultant_vector)
from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()
    # print(f"Move {game.move_counter}: good bot moved {piece_moved}
from {from_square} to {to_square} with a score perceived of {score}")

    # if game.board_state.color_in_check():
    #     print(f"CHECK: {game.board_state.next_to_go} in check")

    # again check if this affects the test
success, msg, board_state = success_criteria(game,
description=description)
if success is not None:
    # game.board_state.print_board()
    return success, msg, board_state

    # # if needed provide console output to clarify that slow bot
hasn't crashed
    # if game.move_counter % 10 == 0 or depth >= 3:
    # print(f"Moves {game.move_counter}: static evaluation ->
{game.board_state.static_evaluation()}, Minimax evaluation -> {score} by
turn {description}")

# below are some function that have been programmed as classes with a
__call__ method.
# these are basically fancy functions that CAN BE HASHED.
# I had to manually do this under the hood hashing as it is needed to
allow communication between the threads
# a job must be hashable to be piped to a thread (separate python
instance)

# this is a pipe-able object that makes a random move
```

```
class Random_Bot():
    # picks a random move
    def __call__(self, game):
        # determine move at random
        legal_moves = list(game.board_state.generate_legal_moves())
        assert len(legal_moves) != 0
        # match minimax output structure
        # score, best_move
        return None, random_choice(legal_moves)

    def __hash__(self) -> int:
        return hash("I am random bot, I am a unique singleton so each
instance can share a hash")

# this bot picks a good move
# it exploration is limited by time
# has constructor to allow for configuration
class Bot_By_Time():
    # configure for depth and allow variable depth
    def __init__(self, time, cache_allowed=False):
        self.time = time
        self.move_engine = Move_Engine_Timed()
        self.move_engine.cache_allowed = cache_allowed

    # make minimax function call given config
    def __call__(self, game):
        return self.move_engine(
            board_state=game.board_state,
            time=self.time
        )
        # result = self.move_engine(
        #     board_state=game.board_state,
        #     time=self.time
        # )
        # print(f"Bot_By_Time:    result={result}")
        # return result

    def __hash__(self) -> int:
        return hash(f"Bot_By_Time(time={self.time})")

# this is a bot that has its exploration limited by depth
class Bot_By_Depth():
    # configure for depth and allow variable depth
    def __init__(self, depth, cache_allowed=False):
        self.depth = depth
        self.move_engine = Move_Engine_Prime()
        self.move_engine.cache_allowed = cache_allowed
```

```
# make minimax function call given config
def __call__(self, game):
    return self.move_engine(
        board_state=game.board_state,
        depth=self.depth
    )
# result = self.move_engine(
#     board_state=game.board_state,
#     depth=self.depth
# )
# print(f"Bot_By_Depth:    result={result}")
# return result

def __hash__(self) -> int:
    return hash(f"Bot_By_Depth(depth={self.depth})")

# used to look at a game and decide if the test should finish
class Success_Criteria():
    # constructor allow config for stalemates to sill allow test to pass
    def __init__(self, allow_stalemate_3_states_repeated: bool):
        self.allow_stalemate_3_states_repeated =
allow_stalemate_3_states_repeated

    def __call__(self, game: Game, description):
        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do
        match game.check_game_over():

            # if 3 repeat stalemate, check with config wether is is allows
            # case True, None, "Stalemate":
            case True, None, _:
                # game.board_state.print_board()
                if game.board_state.is_3_board_repeats_in_game_history()
and self.allow_stalemate_3_states_repeated:
                    # game.board_state.print_board()
                    # print(f"Success: Stalemate at {game.move_counter}
moves in test {description}: 3 repeat board states, outcome specify
included in allowed outcomes")
                    # return True, f"Success: Stalemate at
{game.move_counter} moves in test {description}: 3 repeat board states,
outcome specify included in allowed outcomes",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
                    return (
                        True,
                        f"Success: Stalemate at in test {description}: 3
repeat board states, outcome specify included in allowed outcomes",
```

```
                map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
            )
        else:
            # game.board_state.print_board()
            # print(f"FAILURE: ({description}) stalemate caused (3
repeats? -> {game.board_state.is_3_board_repeats_in_game_history()}")
            return (
                False,
                f"FAILURE: ({description}) stalemate caused (3
repeats? -> {game.board_state.is_3_board_repeats_in_game_history()}",
                map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
            )
        # good bot loss causes test to fail
        # case True, 1, "Checkmate":
        case True, 1, _:
            # game.board_state.print_board()
            # print(f"Failure: ({description}) computer lost")
            return (
                False,
                f"Failure: ({description}) computer lost",
                map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
            )
        # good bot win causes test to pass
        # case True, -1, "Checkmate":
        case True, -1, _:
            # game.board_state.print_board()
            # print(f"SUCCESS: ({description}) Game has finished and
been won in {game.move_counter} moves")
            return (
                True,
                f"SUCCESS: ({description}) Game has finished and been
won in {game.move_counter} moves",
                map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
            )
        # if the game isn't over, return success as none and test will
continue
        case False, _, _:
            return (
                None,
                None,
                None,
            )
    def hash(self):
```

```
        return
hash(f"Success_Criteria(allow_stalemate_3_states_repeated={self.allow_stalemate_3_states_repeated})")

# given a test package (config for one test), carry it out
def execute_test_job(test_data_package):
    # deal with unexplained bug where argument is tuple / list length 1
    # containing relevant dict (quick fix as only a test)
    # I was able to identify that this is where it occurs and add a
    # correction but I am not sure what the cause of the bug is
    if any(isinstance(test_data_package, some_type) for some_type in
(tuple, list)):
        if len(test_data_package) == 1:
            test_data_package = test_data_package[0]

    # print(f"test_data_package    -->    {test_data_package}")

    # really simple, call minimax test component providing all keys in
    # package as keyword arguments
    return minimax_test_component(**test_data_package)

# this pool jobs function for completing tests in parallel is not needed
# when the move engine is parallelized

# this function takes an iterable of hashable test_packages
# it all 8 logical cores on my computer to multitask to finish the test
# sooner
def pool_jobs(test_data):
    # counts logical cores
    # my CPU is a 10th gen i7
    # it has 4 cores and 8 logical cores due to hyper threading
    # with 4-8 workers I can use 100% of my CPU
    cores = multiprocessing.cpu_count()

    # create a pool
    with multiprocessing.Pool(cores) as pool:
        # map the execute_test_job function across the set of test
        # packages using multitasking
        # return the result
        # return pool.map(
        #     func = execute_test_job,
        #     iterable = test_data
        # )
        yield from pool.map(
            func=execute_test_job,
            iterable=test_data
        )
```

```
# test case contains unit tests
# multitasking only occurs within a test, tests are themselves executed
sequentially
# I could pool all tests into one test function but this way multiple
failures can occur in different tests
# (one single test would stop at first failure)

overnight = False
RANDOM_TRIAL_NUM = 3

# this function asserts that all elements in a list are the same
def test_same(some_list):
    assert len(some_list) > 0
    return all(some_list[0] == element for element in some_list)

# @unittest.skip("Takes too long")
class Test_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and
    checks the results with a unittest
    # a failure is correctly identified to correspond to the function that
    called this function
    # reduces repeated logic
    def check_test_results(self, test_results):
        # for success, msg, final_pieces_matrix in test_results:
        for success, msg, _ in test_results:
            # print()
            # for row in final_pieces_matrix:
            #     row = " ".join(map(
            #         lambda square: str(square).replace("None", ". "),
            #         row
            #     ))
            #     print(row)
            # print(msg)

            # i choose to iterate rather than assert all as this allows me
            to have the appropriate message on failure
            self.assertTrue(
                success,
                msg=msg
            )

    # this function tests that a cached call always produces the same
    result as one without caching
    # it also ensures that the second cache call for the same search is
    much faster
```

```
# this test is completed on many random board states
def test_cache_is_faster(self):
    trials = 40
    # trials = 5

    # create and configure move engines
    move_engine_with_cache = Move_Engine_Prime()
    move_engine_without_cache = Move_Engine_Prime()

    move_engine_without_cache.cache_allowed = False

    for move_engine in (move_engine_with_cache,
    move_engine_without_cache):
        move_engine.depth = 2
        move_engine.cache_manager.allow_greater_depth = False

    # repeat test for many trials
    for _ in range(trials):
        # note random_board_state as a function is inefficient as it
is used for testing only, takes a while
        # generate a random board state
        board_state = random_board_state(60)
        # turn of 3 repeat stalemates so cache and without behave the
same
        board_state.three_repeat_stalemates_enabled = False

        # board_state.print_board()

        # check result with out cache
        true_result = move_engine_without_cache(board_state)
        true_result = list(true_result)
        true_result[1] = list(true_result[1])

        # attempt 1 with cache
        first_cache_result, first_cache_time_delta = time_function(
            lambda: move_engine_with_cache(board_state)
        )
        first_cache_result = list(first_cache_result)
        first_cache_result[1] = list(first_cache_result[1])

        # check that the results were the same
        self.assertEqual(
            first_cache_result,
            true_result,
            "The first cache call should give the same result as
without cache"
        )
```

```
# attempt 2 with cache
second_cache_result, second_cache_time_delta = time_function(
    lambda: move_engine_with_cache(board_state)
)

second_cache_result = list(second_cache_result)
second_cache_result[1] = list(second_cache_result[1])

# check that the results were again the same
self.assertEqual(
    first_cache_result,
    true_result,
    "The second cache call should give the same result as
without cache"
)

# check that with cache was much faster
self.assertLessEqual(
    second_cache_time_delta,
    0.2 * first_cache_time_delta,
    "When using database cache, the lookup should be at least
5 times quicker than calculation"
)

# tests basic minimax vs random moves
def test_timed_vs_randomron(self):
    trials = RANDOM_TRIAL_NUM

        # test package generated to include relevant data and logic (bots
and success criteria)
    def generate_jobs():
        for num in range(1, trials+1):
            for time in [2, 5, 10, 20]:
                yield {
                    "description": f"test: {time}s timed move engine
vs randomron",
                    "good_bot": Bot_By_Time(time=time),
                    "bad_bot": Random_Bot(),
                    "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
                    "write_to_csv": True,
                    "csv_folder":
"./test_reports/test_timed_vs_randomron",
                    "csv_file_name": f"test_timed_{time}s_num_{num}"
                }

    def generate_test_results():
        for test_specs in generate_jobs():
```

```
yield minimax_test_component(**test_specs)

self.check_test_results(
    generate_test_results()
)

# tests basic minimax vs random moves
@unittest.skip("takes too long")
def test_depth_vs_randomtron(self):
    trials = RANDOM_TRIAL_NUM

        # test package generated to include relevant data and logic
(bots and success criteria)
    def generate_jobs():
        for num in range(1, trials+1):
            # for depth in range(4):
            for depth in [1, 2]:
                yield {
                    "description": f"test: depth {depth} move
engine vs randomtron",
                    "good_bot": Bot_By_Depth(depth=depth),
                    "bad_bot": Random_Bot(),
                    "success_criteria": Success_Criteria(
                        allow_stalemate_3_states_repeated=(depth<=
1)
                ),
                "write_to_csv": True,
                "csv_folder":
"../test_reports/test_depth_vs_randomtron",
                "csv_file_name":
f"test_depth_{depth}_num_{num}"
            }

    def generate_test_results():
        for test_specs in generate_jobs():
            yield minimax_test_component(**test_specs)

    self.check_test_results(
        generate_test_results()
    )

# test that bots that can explore more are better
# @unittest.skip("takes too long")
def test_bots_by_depth(self):
    # test package generated to include relevant data and logic (bots
and success criteria)
    def generate_jobs():
        # don't do depth 0s as they cannot decide a move
```

```

        for depth_greater in (3,):
            # for depth_greater in (2, 3):
            #     # ensure depth_a >= depth_b
            #     # for depth_lesser in range(1, depth_greater):
            #     # for depth_lesser in range(1, depth_greater+1):
            #     # allow_draw = (depth_lesser == depth_greater)
            depth_lesser = depth_greater - 1
            yield {
                "description": f"test: depth {depth_greater} bot vs
depth {depth_lesser} bot",
                "good_bot": Bot_By_Depth(depth=depth_greater),
                "bad_bot": Bot_By_Depth(depth=depth_lesser),
                "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
                # "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=allow_draw),
                "write_to_csv": True,
                "csv_folder": "./test_reports/test_bots_by_depth",
                "csv_file_name":
f"test_depth_{depth_greater}_vs_depth_{depth_lesser}"
            }

        def generate_test_results():
            for test_specs in generate_jobs():
                yield minimax_test_component(**test_specs)

            self.check_test_results(
                generate_test_results()
            )

# test that bots that can explore more are better
def test_bots_by_time(self):
    # test package generated to include relevant data and logic (bots
and success criteria)
    def generate_jobs():
        time_deltas = [2, 5, 10, 15, 20]
        for time_delta_lesser in time_deltas:
            time_delta_greater = max(1.4*time_delta_lesser,
4+time_delta_lesser)
            # time_delta_greater = max(2*time_delta_lesser,
10+time_delta_lesser)
            # time_delta_greater = max(4*time_delta_lesser,
15+time_delta_lesser)
            yield {
                "description": f"test: {time_delta_greater}s timed bot
vs {time_delta_lesser}s timed bot",
                "good_bot": Bot_By_Time(time=time_delta_greater),
                "bad_bot": Bot_By_Time(time=time_delta_lesser),
            }

```

```
# "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
    "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=True),  
    "write_to_csv": True,  
    "csv_folder": "./test_reports/test_bots_by_time",  
    "csv_file_name":  
f"test_{time_delta_greater}s_timed_bot_vs_{time_delta_lesser}s_timed_bot"  
}  
  
def generate_test_results():  
    for test_specs in generate_jobs():  
        yield minimax_test_component(**test_specs)  
  
    self.check_test_results(  
        generate_test_results()  
)  
  
# test what the parallel move engine doesn't produce a different  
output  
def test_deterministic_outcomes_parallel(self):  
    # create move engines and configures them  
    with_parallel_engine = Move_Engine_Prime()  
  
    without_parallel_engine = Move_Engine_Prime()  
  
    for engine in (with_parallel_engine, without_parallel_engine):  
        engine.cache_allowed = False  
        engine.additional_depth = 0  
  
        with_parallel_engine.parallel = True  
        without_parallel_engine.parallel = False  
  
        def always(*args, **kwargs): return True  
        def never(*args, **kwargs): return False  
  
        with_parallel_engine.should_use_parallel = always  
        without_parallel_engine.should_use_parallel = never  
  
    # generate a large number of random board states  
    def gen_random_board_states():  
        for _ in range(40):  
            yield random_board_state(moves=randint(0, 10))  
        for _ in range(20):  
            yield random_board_state(moves=randint(0, 20))  
        for _ in range(10):  
            yield random_board_state(moves=randint(0, 40))
```

```
# test that for each board state, the move engines produce the
# same deterministic outcome
    for board_state in gen_random_board_states():
        score_parallel, move_parallel =
with_parallel_engine(board_state)
        score_non_parallel, move_non_parallel =
without_parallel_engine(board_state)

        fail_msg = f"Test hash={hash(board_state)}: score parallel !="
without --> {score_parallel} != {score_non_parallel}"
        # assert score_parallel == score_non_parallel, fail_msg
        self.assertEqual(
            score_parallel, score_non_parallel, fail_msg
        )

        fail_msg = f"Test hash={hash(board_state)}: move parallel !="
without --> {move_parallel} != {move_non_parallel}"
        # assert move_parallel == move_non_parallel, fail_msg
        self.assertEqual(
            move_parallel, move_non_parallel, fail_msg
        )

# for trial in range(1, 101):
#     try:
#         moves = random_choice(range(10, 50))
#         board_state = random_board_state(moves)

#         score_parallel, _ = with_parallel_engine(board_state)
#         score_non_parallel, _ =
without_parallel_engine(board_state)

#         fail_msg = f"Test moves={moves}"
hash={hash(board_state)}: score parallel != without --> {score_parallel}
!= {score_non_parallel}"
#         assert score_parallel == score_non_parallel, fail_msg
#         except AssertionError as e:
#             print(f"TEST FAILURE: (trial={trial})")
#             board_state.print_board()
#             print(repr(board_state))
#             self.fail(str(e))
#         except Exception as e:
#             self.fail(f"UNEXPECTED ERROR (trial =
{trial}): {str(e)}")

if __name__ == '__main__':
    unittest.main()
```

I added tests to verify that the minimax algorithm was faster when it encountered a board state that was in the cache.

I also added a test to verify that the output produced by the new improved parallel minimax algorithm was the same when the completing the search in parallel and when completing the search without any concurrency.

The final test I added were to test the time limited minimax algorithm by checking that bots given move time to explore the tree did better than those given less time.

These tests allowed me to verify that my minimax algorithm was still working after I modified it. It was immediately apparent if the change caused the algorithm to crash or produce random moves.

I also created some new tests for the game class:

```
import unittest
import pickle

from .game import Game

from assorted import NotComputerTurn, NotUserTurn
from chess_functions import King, Queen, Rook, Bishop, Pawn, Board_State, Vector
from move_engine import Move_Engine_Prime

class Test_Case(unittest.TestCase):
    # this test checks that a game object can be preserved when it is encoded to binary and then loaded back
    def test_save_and_restore(self):
        game = Game(
            echo=False,
            time=10,
        )

        # make a load of changes to the game object
        game.implement_user_move(from_square="A2", to_square="A4")
        game.implement_computer_move(best_move_function=lambda *arg, **kwargs: [0, [Vector(0, 6), Vector(0, -2)]])

        game.implement_user_move(from_square="B2", to_square="B4")
        game.implement_computer_move(best_move_function=lambda *arg, **kwargs: [0, [Vector(1, 6), Vector(0, -2)]])

        game.implement_user_move(from_square="C2", to_square="C4")
        game.implement_computer_move(best_move_function=lambda *arg, **kwargs: [0, [Vector(2, 6), Vector(0, -2)]])

        original_game = game
```

```

pickle_file_path =
"chess_game/test_data/test_save_and_restore/saved_game.game"

# dump it to binary and save to a file
with open(pickle_file_path, "wb") as file:
    file.write(
        pickle.dumps(
            game
        )
    )

# read the binary and load it back into an object
with open(pickle_file_path, "rb") as file:
    reloaded_game = pickle.loads(
        file.read()
    )

# check that the games are the same
self.assertTrue(
    original_game == reloaded_game,
    "The original game was not the same as the saved and reloaded
game"
)

```

These were used to check that I could save a game object to binary and then restore it without its contents being changed.

```

def test_whose_go(self):
    # this function tests that the validation present can stop a
player going twice in a row
    game = Game(time=2)

    game.implement_user_move(from_square="A2", to_square="A4")

    def try_to_make_user_move(*args, **kwargs):
        game.implement_user_move(from_square="B2", to_square="B4")

    def try_to_make_computer(*args, **kwargs):
        game.implement_computer_move()

    self.assertRaises(
        NotUserTurn,
        try_to_make_user_move,
        "The user shouldn't be able to make 2 consecutive turns"
    )

    game.implement_computer_move()

```

```

self.assertRaises(
    NotComputerTurn,
    try_to_make_computer,
    "The computer shouldn't be able to make 2 consecutive turns"
)

```

I also created the above test to ensure the validation of the Game class was working and that it wouldn't let the wrong player go. To do this I asserted that it raised the appropriate errors.

Here are some screenshots as evidence that the old tests and my new tests are able to run successfully:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
PS C:\Users\henry\Documents\computing coursework\prototype 6> ./run_tests_fast.ps1
-----
Ran 80 tests in 10.801s
OK

```

Move Number	Moving color	New Utility	Future Utility	Time Taken	No Legal moves	Move Description	White Pieces Taken	Black Pieces Taken
1	white (x)	-50	-10	0.03 sec	20	WP moved from F8 to B4	WP WB	-
2	Black (x)	-50	10.01 sec	20	BN moved from B8 to C6	WP WB	-	
3	white (x)	-50	0.04 sec	21	WN moved From G1 to H3	WP WB	-	
4	Black (x)	-155	-95	0.04 sec	22	BN moved From E8 to D4, taking piece WP	WP WB	-
5	white (x)	-125	0.03 sec	20	WP moved from E2 to E3	WP WB	-	
6	Black (x)	-175	-125	17.88 sec	25	BN moved From G8 to F6	WP WB	-
7	white (x)	-155	0.04 sec	29	WN moved From D8 to D7	WP WB	-	
8	Black (x)	-505	-180	22.57 sec	27	BN moved from B4 to D3, taking piece WB	WP WB	-
9	white (x)	-505	-	0.03 sec	3	CHECK: w in check	WP WB	-
10	Black (x)	-780	-495	17.07 sec	29	W moved From D3 to E2	WP WB	-
11	white (x)	-780	-	0.03 sec	4	CHECK: w in check	WP WB	-
12	Black (x)	-875	-600	13.26 sec	25	BN moved from C1 to A2, taking piece WP	WP WP WB WB	-
13	white (x)	-895	-	0.03 sec	24	WP moved From C2 to C3	WP WP WB WB	-
14	Black (x)	-935	-660	11.88 sec	24	BN moved From D2 to D4	WP WP WB WB	-
15	white (x)	-660	-	0.03 sec	25	WR moved From A1 to A2, taking piece BN	WP WP WB WB	-
16	Black (x)	-935	-635	18.76 sec	28	BB moved From C8 to H3, taking piece WN	WP WP WB BN	BN
17	white (x)	-935	-	0.03 sec	30	W moved From D1 to E1	WP WP WB BN	BN
18	Black (x)	-1070	-955	39.21 sec	32	BB moved From H3 to G2, taking piece WP	WP WP WB BN WN	BN
19	white (x)	-1070	-	0.04 sec	32	WQ moved From D1 to G4	WP WP WB BN WN	BN
20	Black (x)	-1960	-1630	4.45 sec	30	BN moved From D2 to D4, taking piece WQ	WP WP WB BN WN WQ	BN
21	white (x)	-1980	-	0.03 sec	25	WP moved From F2 to F3	WP WP WB BN WN WQ	BN
22	Black (x)	-2460	-2125	23.19 sec	33	BB moved From G2 to H1, taking piece WS	WP WP WR BN WN WQ	BN
23	white (x)	-2345	-	0.03 sec	24	W moved From D1 to D2, taking piece WP	WP WP WR BN WN WQ	BP BN
24	Black (x)	-2845	-2510	8.35 sec	30	BR moved From A8 to A7, taking piece WB	WP WP WR BR BN WN WQ	BP BN
25	white (x)	-2845	-	0.01 sec	12	WK moved From D1 to D1, taking piece WB	WP WP WR BR BN WN WQ	BP BN
26	Black (x)	-2960	-2950	14.44 sec	35	BB moved From H1 to F3, taking piece WP	WP WP WR BR BN WN WQ	BP BN
27	white (x)	-2960	-	0.01 sec	-	CHECK: w in check	WP WP WR BR BN WN WQ	BP BN
28	Black (x)	-3010	-2970	10.99 sec	38	WK moved From D1 to D2, taking piece WP	WP WP WR BR BN WN WQ	BP BN
29	white (x)	-2990	-	0.01 sec	7	BN moved From D4 to H2, taking piece WP	WP WP WR BR BN WN WQ	BP BN
30	Black (x)	-3033	-2990	5.81 sec	36	WP moved From D2 to D3	WP WP WR BR BN WN WQ	BP BN
31	white (x)	-3045	-	0.01 sec	8	W moved From D1 to D2	WP WP WR BR BN WN WQ	BP BN
32	Black (x)	-3155	-3135	7.69 sec	38	BN moved From G4 to E3, taking piece WP	WP WP WR BR BN WN WQ	BP BN
33	white (x)	-3155	-	0.01 sec	4	CHECK: w in check	WP WP WR BR BN WN WQ	BP BN
34	Black (x)	-3155	-3135	10.16 sec	41	W moved From E3 to D2	WP WP WR BR BN WN WQ	BP BN
35	white (x)	-3155	-	0.01 sec	6	BN moved From E3 to F5	WP WP WR BR BN WN WQ	BP BN
36	Black (x)	-3165	-3155	7.15 sec	39	W moved From A7 to A2	WP WP WR BR BN WN WQ	BP BN
37	white (x)	-3165	-	0.01 sec	-	CHECK: w in check	WP WP WR BR BN WN WQ	BP BN
38	Black (x)	-3195	-3150	5.81 sec	46	W moved From E7 to E5	WP WP WR BR BN WN WQ	BP BN
39	white (x)	-3195	-	0.01 sec	4	BN moved From B1 to D2	WP WP WR BR BN WN WQ	BP BN
40	Black (x)	-3150	-3155	14.88 sec	52	BD moved From C8 to C6	WP WP WR BR BN WN WQ	BP BN
41	white (x)	-3155	-	0.01 sec	3	W moved From C3 to C4	WP WP WR BR BN WN WQ	BP BN
42	Black (x)	-3470	-3450	14.87 sec	58	BB moved From G5 to D2, taking piece WN	WP WP WR BR BN WN WQ	BP BN
43	white (x)	-3450	-	0.01 sec	1	CHECK: w in check	WP WP WR BR BN WN WQ	BP BN
44	Black (x)	-1000000	-1000000	8.12 sec	61	WK moved From C1 to B1	WP WP WR BR BN WN WQ	BP BN
-	-	-	-	-	-	WB moved From D2 to C2	WP WP WR BR BN WN WQ	BP BN
						CHECKmate! B wins	-	-

The dots represent completed test. The first image show all 80 of the fast test have been completed. The second image shows the last test to finish form the slow minimax test (which I ran overnight). As the game output function has repeatedly cleared the terminal we cannot see the dots for the rest of the test. I could tell that the tests were finished as there was no python task in the task manager and as this test hadn't been cleared away as another test was completed.

In the process of getting these test to run successfully I identified various issues due to initially failing test which were later fixed

I then tested the website and the program as a whole (black box testing) by playing games of chess against it. This helped me identify more issues that I was able to fix.

Here are the main issues that were detected by the automated unit tests:

Timed Minimax not working:

From the automated testing of the timed minimax function I realised that the function was not consistently interrupting itself and sticking to the provided maximum time.

I considered this a failed test as the expected outcome was the at the time column in the automated unit test would show a consistent amount of time taken by each bot.

Here is the screenshot evidence of the failed test:

This shows a test of a 2 second bot vs a 6 second bot. We can see that there is significant variance in the times shown and that on average, they are longer than the permitted max time:

Moves History:									
Move Number	Moving Color	New Utility	Future Utility	Time Taken	No legal moves	Move Description	White Pieces Taken	Black Pieces Taken	
1	White (+)	50	50	4.73 sec	20	WN moved from B1 to C3	WP	BN	
2	Black (-)	0	0	14.33 sec	20	BN moved from B8 to C6	WP	BN	
3	White (+)	50	50	3.61 sec	22	WN moved from G1 to F3	WP	BN	
4	Black (-)	0	0	9.49 sec	22	BN moved from E8 to E6	WP	BN	
5	White (+)	40	40	3.34 sec	24	WP moved from D2 to D4	WP	BN	
6	Black (-)	-90	-90	10.15 sec	24	BN moved from C6 to D4, taking piece WP	WP	BN	
7	White (+)	260	260	4.29 sec	32	WQ moved from D1 to D4, taking piece BN	WP	BN	
8	Black (-)	220	220	12.17 sec	21	BP moved from D7 to D5	WP	BN	
9	White (+)	550	550	6.46 sec	47	WQ moved from D4 to F6, taking piece BN	WP	BN BN	
10	Black (-)	-365	-365	14.22 sec	29	BP moved from E7 to E5, taking piece WQ	WP	BN BN	
11	White (+)	325	325	3.43 sec	31	WN moved from C3 to D5, taking piece WQ	WP	BN BN	
12	Black (-)	-585	-585	11.68 sec	28	BQ moved from D8 to D5, taking piece WN	WP	BN BN	
13	White (+)	-545	-545	4.84 sec	25	WP moved from E2 to E4	WP	BN BN	
14	Black (-)	-665	-645	10.09 sec	46	BQ moved from D5 to E4, taking piece WP	WP	BN BN	
-	-	-	-	-	-	CHECK: W in check	-	-	
15	White (+)	-645	-645	4.49 sec	4	WB moved from C8 to E3	WP MP WQ	BN BN BN	
16	Black (-)	-975	-975	9.31 sec	44	BP moved from E4 to F3, taking piece WN	WP MP WQ	BN BN BN	
17	White (+)	-90	-90	6.19 sec	30	WP moved from G2 to F3, taking piece BQ	WP MP WQ	BN BN BN BQ	
18	Black (-)	-110	-110	12.34 sec	26	BB moved from C8 to E6	WP MP WQ	BN BN BN BQ	

```
Time Taken
4.73 sec
14.33 sec
3.61 sec
9.02 sec
3.34 sec
10.15 sec
4.29 sec
12.17 sec
6.46 sec
14.22 sec
3.99 sec
11.68 sec
4.84 sec
10.09 sec
-
4.49 sec
9.31 sec
6.19 sec
12.34 sec
-
```

This was because I not checking the time and interrupting the function frequently enough. When I completed a depth 3 call, I was only checking the time after pre-sorting the moves and after each depth 2 recursive call was completed. To fix this I added a max time argument to the main minimax method of the original Move Engine class.

```
def minimax(self, board_state: Board_State, depth, alpha=-ARBITRARILY_LARGE_VALUE+1, beta=ARBITRARILY_LARGE_VALUE+1,
variable_depth: bool | None = None, part_of_presort=False,
legal_moves_to_examine: Iterable | None = None, max_time=None) ->
tuple[int, tuple[Vector, Vector]]:
```

It had a default value of None which meant that there was no max time. This meant that previous code that called the minimax function without a max time parameter was unaffected. I then added a relevant function to interrupt the minimax function:

```
# this function is used to ensure that the minimax function can be self
interrupting
def is_time_expired(max_time):
    if max_time is None:
        return False
    # print("⌚", end="")
    return perf_counter() >= max_time
```

and regularly checked if the max time had been exceeded throughout the minimax function:

```
# self interrupting
if is_time_expired(max_time):
    raise TimeOutError()
```

I also passed on this max time argument on to all recursive calls

```
# recursive minimax call with depth n-1 to evaluate child
score, _ = self.minimax(
    board_state=child_board_state,
    depth=depth-1,
    alpha=alpha,
    beta=beta,
    # is_time_expired=is_time_expired,
    max_time=max_time,
)
```

This meant that the minimax timed call now more reliably stuck to its designated max time.

Here is a screen shot of a unit test to show that the issue was fixed:

Move Number	Moving Color	New Utility	Future Utility	Time Taken	No legal moves	Move Description	White Pieces Taken	Black Pieces Taken
1	White (+)	50	50	5.16 sec	20	MN moved from B1 to C3		
2	Black (-)	0	0	9.15 sec	20	BN moved from D1 to F5		
3	White (+)	50	50	1.14 sec	22	MN moved from G1 to F3		
4	Black (-)	0	0	9.21 sec	22	BN moved from G8 to F6		
5	White (+)	40	40	5.64 sec	24	WP moved from D2 to D4		
6	Black (-)	-90	-90	9.25 sec	24	BN moved from C6 to D4, taking piece WP	WP	
7	White (+)	260	260	5.38 sec	32	WQ moved from D1 to D4, taking piece BN	WP	
8	Black (-)	220	220	9.17 sec	21	BP moved from D7 to D5	WP	
9	White (+)	550	550	6.87 sec	47	HQ moved from D5 to D3, taking piece BN	WP	
10	Black (-)	-365	-365	9.25 sec	35	BP moved from D4 to F6, taking piece HQ	WP HQ	
11	White (+)	-235	-235	1.24 sec	31	MN moved from C3 to D5, taking piece BP	WP MN	
12	Black (-)	-585	-585	9.23 sec	28	BQ moved from D8 to D5, taking piece WN	WP MN HQ	
13	White (+)	-545	-545	5.36 sec	25	WP moved from E2 to E4	WP MN HQ	
14	Black (-)	-665	-645	9.19 sec	46	BQ moved from D5 to E4, taking piece WP	WP WP MN HQ	
-						CHECK: W in check		
15	White (+)	-645	-645	5.1 sec	4	WB moved from C6 to E3	WP WP MN HQ	
16	Black (-)	-975	-576	9.25 sec	44	DQ moved from E4 to F3, taking piece WN	WP WP MN MN HQ	
17	White (+)	-98	-90	6.16 sec	30	WP moved from G2 to F3, taking piece BQ	WP WP MN MN HQ	
18	Black (-)	-110	-110	9.14 sec	26	BB moved from C8 to E6	WP WP MN MN HQ	
19	White (+)	-25	-25	5.09 sec	32	WB moved from E3 to A7, taking piece BP	WP WP MN MN HQ	
20	Black (-)	-340	-340	9.14 sec	31	BR moved from A8 to A7, taking piece WB	WP WP MN MN HQ	
21	White (+)	-320	-320	5.36 sec	23	WB moved from F1 to D3	WP WP WB MN MN HQ	
22	Black (-)	-435	-435	9.27 sec	33	BR moved from A8 to D2, taking pieces WP	WP WP WP MN MN HQ	
23	White (+)	65	65	1.01 sec	26	MN moved from A1 to A2, taking piece BR	WP WP WP MN MN HQ	
24	Black (-)	-418	-418	9.14 sec	27	BB moved from E6 to A2, taking piece WR	WP WP WP WR MN MN HQ	
25	White (+)	-325	-325	5.12 sec	22	WB moved from D3 to H7, taking piece BP	WP WP WP WR WB MN MN HQ	
26	Black (-)	-640	-630	9.13 sec	22	BR moved from H8 to H7, taking piece WB	WP WP WP WR WB WB MN MN HQ	
27	White (+)	-630	-630	5.06 sec	13	MK moved from E1 to F1	WP WP WP WR WB WB MN MN HQ	
28	Black (-)	-745	-745	9.06 sec	26	BR moved from H7 to H2, taking piece WP	WP WP WP WR KB KB MN MN HQ	
29	White (+)	-245	-265	5.07 sec	19	WR moved from H1 to H2, taking piece BN	WP WP WP WP WR KB KB MN MN HQ	
8	.	.	BK	.				
7	.	.	BP	BP	.			
6				
5				
4				
3				
2	BB	WP	WP	.				
1	.	WP	.	.				
	.	WK	.	.				
	(A	B	C	D	E	F	G	H)

Time Taken	N
5.16 sec	
9.19 sec	
5.14 sec	
9.21 sec	
5.64 sec	
9.25 sec	
5.38 sec	
9.17 sec	
6.87 sec	
9.22 sec	
5.24 sec	
9.23 sec	
5.36 sec	
9.19 sec	
-	
5.1 sec	
9.25 sec	
6.16 sec	
9.14 sec	
5.09 sec	
9.14 sec	
5.36 sec	
9.27 sec	
5.31 sec	
9.14 sec	
5.12 sec	
9.13 sec	
5.06 sec	
9.06 sec	
5.07 sec	
9.03 sec	

The above screenshot were for a game where a 5 second bot played against a 9 second bot. The times column show that now, the bots are able to stick to their specified time with only a few small exceptions

Some test were written as I wrote the code or even before. This test driven development meant that I aimed to write the code to make the test work. For instance when I created the test to check if game objects could be pickled:

```
def test_save_and_restore(self):
```

it failed the first time. This prompted me to define a `__eq__` method for the `Game_Webiste` class which caused it to then work.

Tests completed on final product:

Test No.	Aspect Being Tested	Type of test	Test Data Used	Expected Result

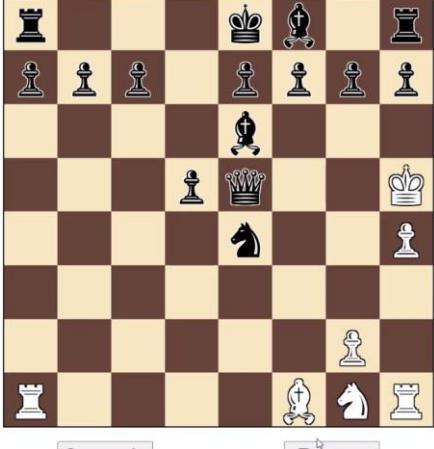
1	The chess board and website renders correctly		Valid	I loaded the page in my browser	The webpage load and the chess board is drawn
Actual Result	Evidence			Success?	
As expected	<p style="text-align: center;">Your Go:</p> 			Full Success	

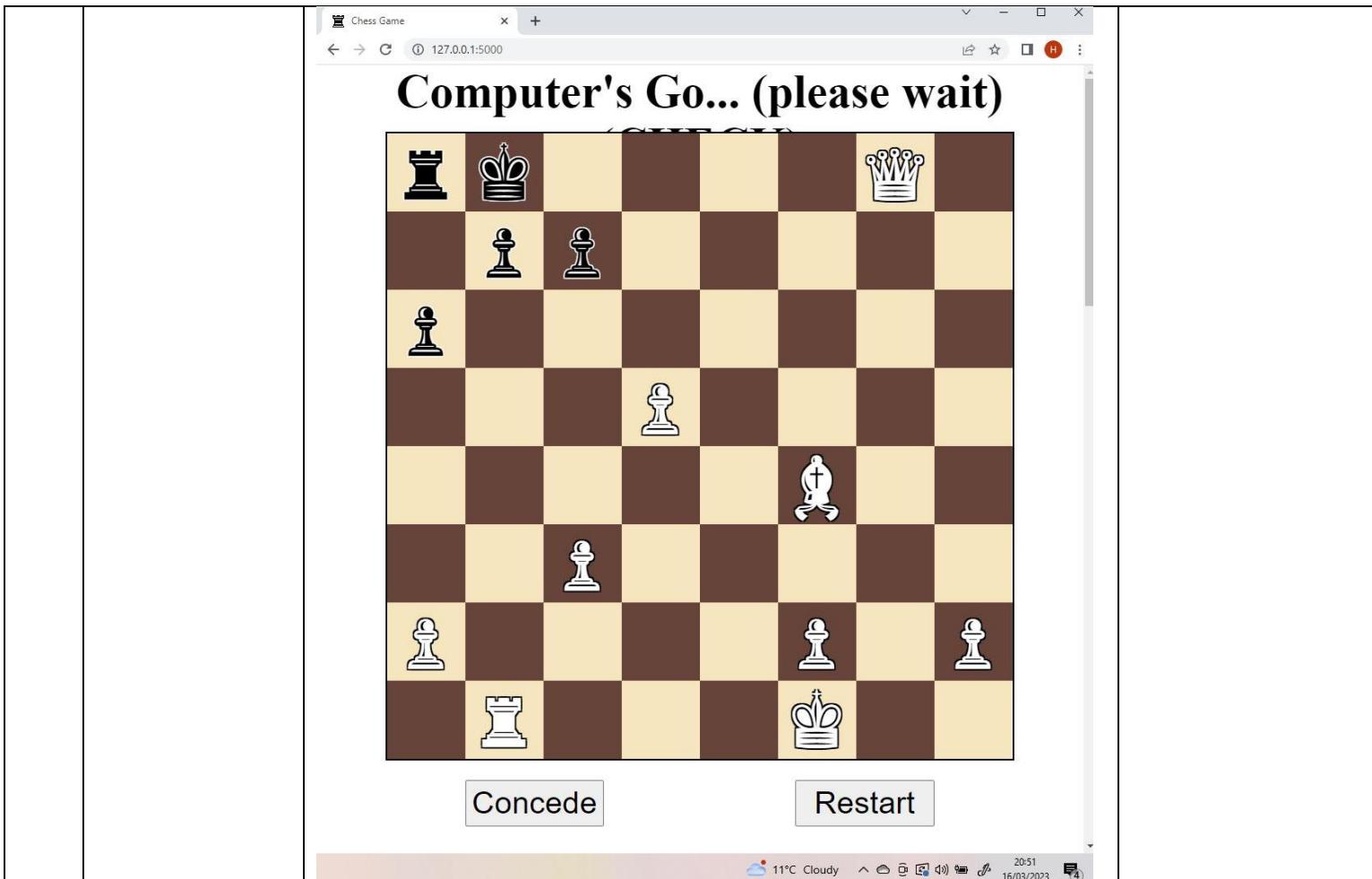
Test No.	Aspect Being Tested		Type of test	Test Data Used	Expected Result
2	I can, the user can move my pieces and will the computer respond with a move		Valid	I clicked a piece to move it forward	I should be able to move my pieces and I should see the computer move its pieces
Test No.	Aspect Being Tested	Type of test	Evidence		Success?
As expected	<p style="text-align: center;">Your Go:</p> 			Full Success	

Test No.	Aspect Being Tested	Type of test	Test Data Used	Expected Result
3	I should be able to make all legal moves and shouldn't be able to make any illegal moves	Valid (I can make legal moves) and then Invalid (I can't make illegal moves)	I will try to move various pieces	I should be able to try to move a variety of pieces and they should move as expected, no

Actual Result	Evidence	Success?
I was not able to move 2 pieces correctly. My pawns couldn't move forward 2 and I couldn't take pieces with a knight	<p>Screenshot where it won't let me move a pawn forward 2: (clicking even when the highlight was wrong did nothing)</p> <p>Your Go:</p>  <p>Concede Restart</p> <p>Screenshot where it won't let me take with a knight: Clicking to take the pawn anyway did nothing</p> <p>Your Go:</p>  <p>Concede Restart</p>	FAILURE

--	--	--

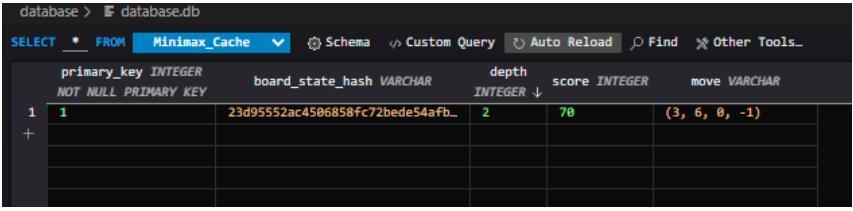
Test No.	Aspect Being Tested	Type of test	Test Data Used	Expected Result
4	Test that the game over functionality works: I will test trying to loose to the program and then trying to beat it in checkmate. I will check that it works as expected and produces the correct output	Valid extreme	I tried to loose and then I tried to win	When the game is over (for both winners), the board should be disabled and the title message should explain that the game is over.
Actual Result		Evidence		
When I lost, the outputs were as expected, when I tried to win, the game crashed when the computer was near to loosing. This is a huge issue as it left the webpage broken as it was waiting for the server move and server had had an error when it tried to for fill the move request		<p>Here is the screenshot of me loosing where everything is as expected.</p> <p>Checkmate: you lost, better luck next time</p>  <p>Evidence of the issue when I tried to win</p>		



		Minimax Cache						
		primary_key	board_state_hash	depth	score	move		
1	4009	20155971464957826	2	660	(0, 7, 6, 0)			
2	4008	927926736545287814	1	660	(1, 3, 0, 3)			
3	4007	642675859063811645	2	660	(0, 7, 6, 0)			
4	4006	536481991241824822	1	660	(1, 2, 0, 4)			
5	4005	1921958880228850792	2	660	(0, 7, 6, 0)			
6	4004	77461767507845130	1	660	(1, 1, 0, 5)			
7	4003	210271792303748139	2	660	(0, 7, 6, 0)			
8	4002	588953250176716216	1	660	(1, 0, 0, 6)			
9	4001	88510193952885238	2	1930	(2, 6, 0, -2)			
10	4000	42375408853353621	1	1935	(6, 7, -6, 0)			
11	3999	1200441462563016436	1	1930	(6, 7, -6, 0)			
12	3998	1241187857639879510	4	1000001	NJUL			
13	3997	2234337633661957847	2	1445	(5, 3, -1, -1)			
14	3996	1787843202595186267	1	1415	(0, 5, 0, -1)			
15	3995	1808701355710456777	1	1415	(0, 5, 0, -1)			
16	3994	1621117848468354948	1	1415	(0, 5, 0, -1)			
17	3993	705120737705654377	1	1425	(0, 5, 0, -1)			
18	3992	10526095857934726541	1	1425	(0, 5, 0, -1)			
19	3991	1925017816109203929	1	1425	(0, 5, 0, -1)			
20	3990	1062452198426405282	1	1425	(0, 5, 0, -1)			
21	3989	1050549739365111308	1	1430	(0, 5, 0, -1)			
22	3988	1999234254579433030	1	1430	(0, 5, 0, -1)			
23	3987	8889799019049903518	1	1430	(0, 5, 0, -1)			
24	3986	5336526071469211	1	1435	(0, 5, 0, -1)			
25	3985	56321036847866432	1	1435	(0, 5, 0, -1)			
26	3984	2286327569406795915	1	1435	(0, 5, 0, -1)			
27	3983	1771359184405357857	1	1435	(0, 5, 0, -1)			
28	3982	2176131602641102354	1	1435	(0, 5, 0, -1)			
29	3981	2019421152020824394	1	1435	(0, 5, 0, -1)			
30	3980	329063785855372918	1	1435	(0, 5, 0, -1)			
31	3979	1596717603520619216	1	1435	(0, 5, 0, -1)			
32	3978	1228768202841809288	1	1435	(0, 5, 0, -1)			
33	3977	392528724704671255	1	1435	(0, 5, 0, -1)			
34	3976	923328485226979813	1	1435	(0, 5, 0, -1)			
35	3975	188642309682774574	1	1435	(0, 5, 0, -1)			
36	3970	730624838680332488	1	1435	(0, 7, 1, 0)			
37	3969	15530148380624312285	1	1435	(0, 7, 1, 0)			
38	3968	373384974879282048	1	1435	(0, 7, 1, 0)			
39	3967	1914255768898845572	1	1435	(0, 7, 1, 0)			
40	3966	161624395896537251	1	1435	(0, 7, 1, 0)			
41	3965	147947750119026085	1	1430	(0, 6, 1, 1)			
42	3964	240466518708162348	1	1430	(0, 6, 1, 1)			
43	3963	192126072362578477	1	1430	(0, 6, 1, 1)			
44	3962	2140803127151520757	1	1430	(0, 6, 1, 1)			
45	3961	2046583137276653598	1	1430	(0, 6, 1, 1)			

Above you can see an erroneous result in the database

Test No.	Aspect Being Tested	Type of test	Test Data Used	Expected Result
5	The database cache is being build up after each minimax search the computer completes	Valid	I played the AI on various difficulties and checked that database as I did	I should add a small amount of depth 1 cache to the database on a low difficulty. It should produce a large amount of depth 1 cache with some depth 2 cache at a high difficulty
Actual Result	Evidence	Success?		
When operating concurrently on a depth 2 call, the algorithm didn't add all its depth 1	Below is a screenshot of the database's content after it searched the board state at depth 2. We can see that it hasn't logged the depth 1	Failure		

	cache to the database cahce: 	
--	---	--

Knights cannot take pieces

One of the issue that was identified by test 3 was that my knights couldn't take enemy pieces. I believe that I must have broken the code at some point at certainly could take pieces at the end of prototype 2.

I created an automated unit test where I replicated the chess game that I had played and asserted that the move I was expecting (knight takes a piece) was in the legal move:

```
# this tests for a bug I encountered where a knight couldn't take a piece,
# I created this test to identify the bug
# I then modified the knight pieces class and used this test to show
that the bug was fixed
def test_troubleshoot_bug_legal_moves(self):
    # replicate the previous situation
    board_state = Board_State()
    board_state = board_state.make_move(
        Vector(1, 0), Vector(1, 2)
    )
    board_state = board_state.make_move(
        Vector(1, 7), Vector(1, -2)
    )
    board_state = board_state.make_move(
        Vector(6, 0), Vector(-1, 2)
    )
    board_state = board_state.make_move(
        Vector(6, 7), Vector(-1, -2)
    )
    board_state = board_state.make_move(
        Vector(4, 1), Vector(0, 1)
    )
    board_state = board_state.make_move(
        Vector(3, 6), Vector(0, -2)
    )

    # board_state.print_board()
```

```

    legal_moves = board_state.generate_legal_moves()

    # check that the knight can make the move
    self.assertTrue(
        (Vector(2, 2), Vector(1, 2)) in legal_moves
    )

```

This test failed which confirmed that I could recreate the logic error. I then fixed the issue by modifying the below code from the Knight Class

```

def generate_movement_vectors(self, pieces_matrix, position_vector):
    # this function yields all 8 possible vectors
    def possible_movement_vectors():
        vectors = (Vector(2, 1), Vector(1, 2))
        # for each x multiplier, y multiplier and vector combination
        for i_multiplier, j_multiplier, vector in iter_product((-1,
1), (-1, 1), vectors):
            # yield corresponding vector
            yield Vector(
                vector.i * i_multiplier,
                vector.j * j_multiplier
            )
        # iterate through movement vectors
        for movement_vector in possible_movement_vectors():
            # get resultant
            resultant_vector = position_vector + movement_vector
            # look at contents of square
            contents =
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix)

            # if square is empty yield vector

            # old code that caused the logic error
            # if contents == "empty":
            #     yield movement_vector

            # corrected code to fix knight bug
            if contents in ("empty", "enemy"):
                yield movement_vector

```

You can see at the bottom that I was only returning the movement vector if the contents of the to square was empty. The below code fixed this issue by also allowing the 2 square to contain an enemy piece.

Here is some screenshot evidence that the bug is fixed meaning the knights can now take pieces:

Computer's Go... (please wait)



Concede

Restart

Your Go:



Concede

Restart

Your Go:



Pawns cannot move forward 2 places:

I identified that I couldn't move more than one Pawn forward 2 spaces as part of failed test number 3. Here is how I fixed the issue.:.

I identified the issue and was able to fix the code:

```
STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
    (
        pieces_mod.Rook(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Queen(color="B"),
        pieces_mod.King(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Rook(color="B")
    ),
    # (pieces_mod.Pawn(color="B"),)*8,
    tuple(pieces_mod.Pawn(color="B") for _ in range(8)),
    (None,)*8,
    (None,)*8,
    (None,)*8,
```

```
(None,)*8,
# (pieces_mod.Pawn(color="W"),)*8,
tuple(pieces_mod.Pawn(color="W") for _ in range(8)),
(
    pieces_mod.Rook(color="W"),
    pieces_mod.Knight(color="W"),
    pieces_mod.Bishop(color="W"),
    pieces_mod.Queen(color="W"),
    pieces_mod.King(color="W"),
    pieces_mod.Bishop(color="W"),
    pieces_mod.Knight(color="W"),
    pieces_mod.Rook(color="W")
)
)
```

The problematic lines of code are commented out and the fixes are below. It turned out that the issue was that the line:

```
(pieces_mod.Pawn(color="B"),)*8
```

Was not making 8 individual pawn objects with their own different last move property.

This meant that all the pawns were the same object passed by reference and so once one piece had moved forward 2, none of them were able to move forward 2.

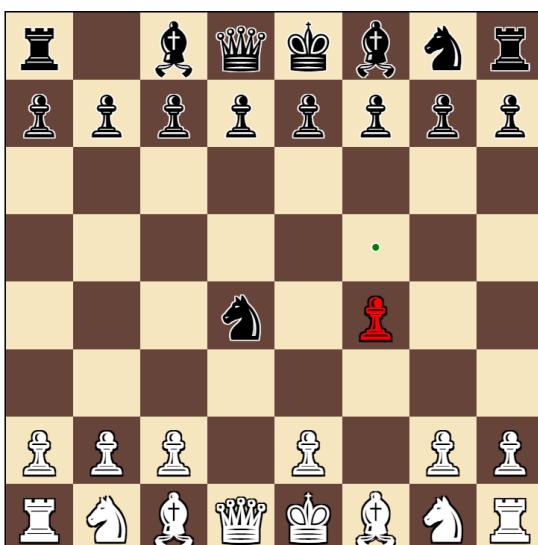
```
tuple(pieces_mod.Pawn(color="W") for _ in range(8)),
```

The above replacement line of code fixed this by making 8 distinct pawn objects.

And here is evidence that the issue is now fixed and that the pawns can each individually move forward 2, but only the first time:

Your Go:



Computer's Go... (please wait)**Concede****Restart****Your Go:****Concede****Restart**[Concurrent workers not caching](#)

In failed test 4, I identified an issue with the worker functions in the parallel minimax module not each individually adding their searches to the database.

This means that a parallel depth 2 search of the starting positions produced only 1 cache item with a depth of 2 and no additional depth 1 cache.

This would have prevented the chess program being able to learn (one of the items of my design criteria).

I realised that I had forgotten to add the caching to these worker classes. As a result I changed them to add database caching.

```
# # this object represents a job that a concurrent worker should complete,
# # I used an object as a function cannot be sent between workers
# class Minimax_Sub_Job:
#     # contractor: configure bot by setting these parameters as
# properties
#     def __init__(self, move_engine, board_state, depth, args, kwargs,
max_time=None) -> None:
#         self.move_engine: Move_Engine = move_engine
#         self.board_state: Board_State = board_state

#         # # not sure why depth sometimes is a single length tuple
containing an int
#         # here is a quick fix
#         if isinstance(depth, tuple):
#             if len(depth) == 1:
#                 depth = depth[0]
#             assert isinstance(depth, int)

#         self.depth = depth
#         self.max_time = max_time

#         # these are other erroneous argument that could be passed to the
minimax call
#         self.args = args
#         self.kwargs = kwargs

#     def __call__(self, legal_moves_sub_array):
#         # perform part of a minimax search using a sub set of legal
moves

#         # print(f"running minimax job with legal moves sub array:
(hash={hash(str(legal_moves_sub_array))})")
#         # print(f"STARTING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
#         # print(legal_moves_sub_array),

#         result = self.move_engine.minimax(
#             board_state=self.board_state,
#             depth=self.depth,
#             legal_moves_to_examine=legal_moves_sub_array,
#             # is_time_expired = self.is_time_expired,
#             max_time=self.max_time,
#             *self.args,
#             **self.kwargs
#         )

#         # print(f"FINISHING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
```

```
#         # print(f"Minimax sub job finished
(hash={hash(str(legal_moves_sub_array))})")
#         return result

# this object represents a job that a concurrent worker should complete,
# I used an object as a function cannot be sent between workers
class Minimax_Sub_Job:
    # contractor: configure bot by setting these parameters as properties
    def __init__(self, board_state, depth, args, kwargs, additional_depth,
cache_allowed, max_time=None) -> None:

        # create a cache manager as necessary
        self.cache_allowed = cache_allowed
        self.cache_manager = DB_Cache() if cache_allowed else None

        # create a move engine as necessary
        self.move_engine: Move_Engine = Move_Engine(
            cache_manager=self.cache_manager,
            cache_allowed=cache_allowed,
            additional_depth=additional_depth,
        )
        # assert isinstance(self.move_engine, Move_Engine),
f"Minimax_Sub_Job.__init__    self.move_engine of unexpected type
{type(self.move_engine)}\n{self.move_engine!r}"

        # assign parameters as properties
        self.board_state: Board_State = board_state

        self.depth = depth
        self.max_time = max_time

        # these are other erroneous argument that could be passed to the
minimax call
        self.args = args
        self.kwargs = kwargs

    def __call__(self, legal_moves_sub_array):
        # perform part of a minimax search using a sub set of legal moves

        # print(f"running minimax job with legal moves sub array:
(hash={hash(str(legal_moves_sub_array))})")
        # print(f"STARTING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
        # print(legal_moves_sub_array)d_state,
```

```

        # result = self.move_engine.minimax(
        # assert isinstance(self.move_engine, Move_Engine),
f"Minimax_Sub_Job.__call__    self.move_engine of unexpected type
{type(self.move_engine)}\n{self.move_engine!r}"

        if self.cache_allowed:
            with self.cache_manager:
                result = self.move_engine.minimax(
                    board_state=self.board_state,
                    depth=self.depth,
                    legal_moves_to_examine=legal_moves_sub_array,
                    # is_time_expired = self.is_time_expired,
                    max_time=self.max_time,
                    *self.args,
                    **self.kwargs
                )
        else:
            result = self.move_engine.minimax(
                board_state=self.board_state,
                depth=self.depth,
                legal_moves_to_examine=legal_moves_sub_array,
                # is_time_expired = self.is_time_expired,
                max_time=self.max_time,
                *self.args,
                **self.kwargs
            )

        # print(f"FINISHING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
        # print(f"Minimax sub job finished
(hash={hash(str(legal_moves_sub_array))})")
        return result

# # this bot is responsible for performing a low depth search of a move to
be used to presort the moves
# class Presort_Moves_Sub_Job:
#     # construct object
#     # def __init__(self, depth, board_state: Board_State, move_engine:
Move_Engine,max_time=None):
#         # cache manager parameter may not be needed if the cache manager
object is also bound to the move engine
#         self.depth = depth
#         self.board_state = board_state
#         self.move_engine = move_engine

#
#         self.is_time_expired = is_time_expired
#         self.max_time = max_time

```

```
#     def __call__(self, move):

#
#         # self interrupting
#         if is_time_expired(self.max_time):
#             raise TimeOutError()

#
#         # score a child, use minimax first call to add caching
#         child = self.board_state.make_move(*move)

#
#         child_score, _ = self.move_engine.minimax(
#             board_state=child,
#             depth=self.depth,
#             # is_time_expired = self.is_time_expired,
#             max_time=self.max_time,
#         )

#
#         return (child_score, move)

# this bot is responsible for performing a low depth search of a move to
# be used to presort the moves
class Presort_Moves_Sub_Job:
    # construct object
    def __init__(self, depth, board_state: Board_State, cache_allowed,
max_time=None) -> None:
        # def __init__(self, depth, board_state: Board_State, move_engine:
Move_Engine, cache_allowed, cache_manager, max_time=None) -> None:
            # cache manager parameter may not be needed if the cache manager
object is also bound to the move engine

        # assign parameters as properties
        self.depth = depth
        self.board_state = board_state

        # create a cache manager as necessary
        self.cache_allowed = cache_allowed
        self.cache_manager = DB_Cache() if cache_allowed else None

        # create a move engine object
        self.move_engine = Move_Engine(
            presort_moves=True,
            additional_depth=0,
            cache_allowed=self.cache_allowed,
            cache_manager=self.cache_manager,
```

```
# cache_allowed=self.cache_allowed,
# cache_manager=self.cache_manager,
)

# self.cache_allowed = cache_allowed
# self.cache_manager = cache_manager

self.is_time_expired = is_time_expired
self.max_time = max_time

def __call__(self, move):
    # cache manager already built into the minimax move engine first
call method

    # print(f"Presort job using cache
manager: {self.cache_manager!r}")
    # with self.cache_manager:
    #     child = self.board_state.make_move(*move)
    #     child_score, _ = self.move_engine.minimax_first_call(
    #         board_state = child,
    #         depth = self.depth
    #     )
    #     # print("Closing cache")
    #     result = (child_score, move)
    # return result

    # self interrupting
    if is_time_expired(self.max_time):
        raise TimeOutError()

    # score a child, use minimax first call to add cache manager being
opened twice
    # no need to say no parallel as a basic move engine is used
    child = self.board_state.make_move(*move)
    if self.cache_allowed:
        with self.cache_manager:
            # child_score, _ = self.move_engine.minimax_first_call(
            child_score, _ = self.move_engine.minimax(
                board_state=child,
                depth=self.depth,
                # is_time_expired = self.is_time_expired,
                max_time=self.max_time,
            )
    else:
        # child_score, _ = self.move_engine.minimax_first_call(
        child_score, _ = self.move_engine.minimax(
            board_state=child,
```

```
        depth=self.depth,
        # is_time_expired = self.is_time_expired,
        max_time=self.max_time,
    )

    return (child_score, move)
```

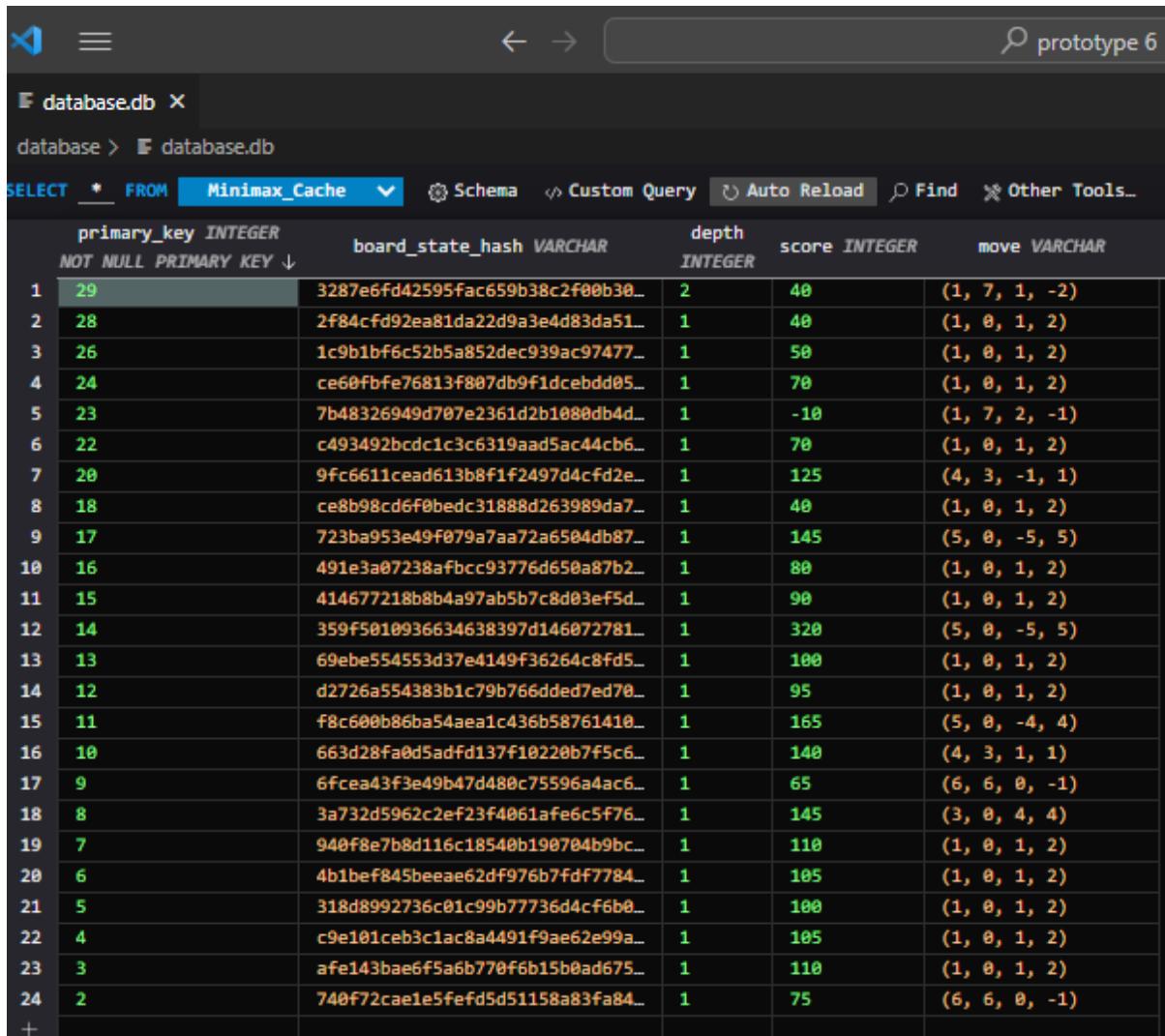
You can see the old Job classes without caching are commented out. The new Job classes that replaced them were able to concurrently access the database in order to cache their results.

Here is a screenshot of the database after the fix to show that the issue was corrected. This shows that minimax cache items created in extreme difficulty (enough time to perform a depth 2 search). We can see the expected outcome which is a depth 2 search was performed and all the depth 1 searches that were needed to perform it were also saved to the database:

	primary_key	INTEGER NOT NULL PRIMARY KEY	board_state_hash	VARCHAR	depth	score	INTEGER	move	VARCHAR
					INTEGER				
77	85		fe9b0e6996b58f9d4437cb03040e70...		1	10		(0, 6, 0, -1)	
78	86		4b883f187b6a518dedd1d3a0543de2...		1	60		(0, 6, 0, -1)	
79	87		55322e780e82c1aaea57dbb70fc07d...		1	10		(0, 6, 0, -1)	
80	88		5b4adda6944ccc3a1f28b5ff8bbe1c...		1	20		(1, 7, 1, -2)	
81	89		eb00462846620af47393fab1002772...		1	60		(0, 6, 0, -1)	
82	90		733fe27cabae50d6f28969434b7d91...		1	30		(1, 7, 1, -2)	
83	91		f1d1b732e3d5f3e23545a76c4b5fc3...		1	10		(0, 6, 0, -1)	
84	92		d474d72160956a584d5decce451143...		1	55		(0, 6, 0, -1)	
85	93		6a4a025d5a8539a113698c89f41fa8...		1	10		(0, 6, 0, -1)	
86	94		27e314a3d1be3f613c061e3522e3ed...		1	55		(0, 6, 0, -1)	
87	95		e153bfdaa56e9f84524b7b2dc53b71...		1	50		(0, 6, 0, -1)	
88	96		bd5bf1312d5c15568b9082bd7ea0c0...		1	5		(0, 6, 0, -1)	
89	97		b61eec1a1fd3c49cc60c55d6c50521...		1	50		(0, 6, 0, -1)	
90	98		d98f115feaadb9b5a027f7b6a6ce4...		1	5		(0, 6, 0, -1)	
91	99		87240dc36634c6cdæ67e8ae519f80...		1	0		(1, 7, 1, -2)	
92	100		559467f2d7bbf9ecc2501d0e2bc951...		1	10		(1, 7, 1, -2)	
93	101		8f120a6ad3b11b39e34bf5bab918d...		1	50		(0, 6, 0, -1)	
94	102		7497bb5a1514bdadd991a99dac3f54...		1	0		(0, 6, 0, -1)	
95	103		23cd8188f8b09a1e0c716bc813c5b6...		1	50		(0, 6, 0, -1)	
96	104		3f045f8102a80e43669cd2206ed458...		1	0		(0, 6, 0, -1)	
97	105		f3f34083b8871895f130b96944ff8e...		1	45		(0, 6, 0, -1)	
98	106		f8295653df0f8140dac2874fd439c3...		1	0		(0, 6, 0, -1)	
99	107		6fccba2f3cbcd96daef68ce1b58217...		1	45		(0, 6, 0, -1)	
100	108		5d04bef62f08a553050f5bb7641114...		1	0		(0, 6, 0, -1)	
101	109		7df5107535239eb71cf4d5ced057dc...		1	40		(0, 6, 0, -1)	
102	110		2ec93821aa73614e9b0f8631cc5fcd...		1	-5		(0, 6, 0, -1)	
103	111		e03db5b5e8f91b456ac56ab5278156...		1	0		(1, 7, 1, -2)	
104	112		dfdb78d52214218ecd823566140472...		1	40		(0, 6, 0, -1)	
105	113		f2f2ec033ecc0785c2a0e67e2641bb...		1	10		(1, 7, 1, -2)	
106	114		eb89f54f52f9a983a62203f50d1404...		1	-5		(0, 6, 0, -1)	
107	115		272e825bf56f55fb7d997cc77b4789...		1	40		(0, 6, 0, -1)	
108	116		0c6b85a43e4f90620a3ae8518ac3d6...		1	40		(0, 6, 0, -1)	
109	117		c5426e7aa74c31a0337741235ef595...		1	-10		(0, 6, 0, -1)	
110	118		e167c45e847460c86df44684c68722...		1	35		(0, 6, 0, -1)	
111	119		84408f20b3f268857141fbb89b0eb3...		1	-10		(0, 6, 0, -1)	
112	120		8dc5059c0685538326ec1b53399068...		1	35		(0, 6, 0, -1)	
113	121		46a04a13e9fc8d0f60f7e3e74bba76...		1	-10		(0, 6, 0, -1)	
114	122		96986b1cb3e2ddfe92fa8d543c8c61...		1	30		(0, 6, 0, -1)	
115	123		0a7635dcc85741142d1eac4ba5db3a...		1	-10		(0, 6, 0, -1)	
116	124		7a48aa11a58f6101e8b909250aa0a6...		1	0		(1, 7, 1, -2)	
117	125		bc8366c6c8ae76dac1d3939bcc5fa...		1	10		(1, 7, 1, -2)	
118	126		ac27311a510758b82ee963179362e...		1	30		(0, 6, 0, -1)	
119	127		334c1a34bd745b95415bf71aa4504d...		2	90		(3, 3, -1, 1)	

You can see the large amount of depth 1 cache used as part of the depth 2 search and then the single unit of depth 2 cache.

Here is second screen shot of a different move that also shows that the issue is fixed.



The screenshot shows a database browser interface with the following details:

- Database:** database.db
- Table:** Minimax_Cache
- Columns:** primary_key (INTEGER, NOT NULL, PRIMARY KEY), board_state_hash (VARCHAR), depth (INTEGER), score (INTEGER), move (VARCHAR).
- Rows:** 24 rows are listed, each containing a primary key value (ranging from 1 to 24), a unique board state hash, a search depth (either 1 or 2), a score (e.g., 40, 50, 70, 125, 145, 165, 180, 205, 220), and a move (e.g., (1, 7, 1, -2), (1, 0, 1, 2)).

The above database content shows the cache produced by 30 second of thinking by the computer (after I made the first move). As you can now see, the depth 2 search result has been saved to the cache as well as the depth 1 search that were completed in the process.

This means that if instead, 6 users on medium difficulty gave the computer 6 lots of 5 seconds to think about a move, it should now still be able to complete a depth 2 search. This is because it could make progress and by completing a group of depth 1 searches, then stop and start where it left off later. This is why it is important that intermediate searches are preserved as they allow the AI to learn

Erroneous Results:

In failed test 5, I found an example of an edge case where my minimax function produced an invalid output:

The screenshot shows a chess board with a checkmate position. The board state is as follows:

- White King (B) is at e1.
- White Queen (W) is at h1.
- White Rook (B) is at a1.
- White Bishop (W) is at c1.
- White Knight (W) is at g1.
- White Pawn (B) is at d2.
- White Pawn (B) is at e2.
- White Pawn (B) is at f2.
- White Pawn (B) is at g2.
- White Pawn (B) is at h2.
- Black King (W) is at e8.
- Black Queen (B) is at h8.
- Black Rook (B) is at a8.
- Black Bishop (W) is at c8.
- Black Knight (W) is at g8.
- Black Pawn (W) is at d7.
- Black Pawn (W) is at e7.
- Black Pawn (W) is at f7.
- Black Pawn (W) is at g7.
- Black Pawn (W) is at h7.

Below the board, there are buttons for "Concede" and "Restart".

On the left, a database query results table is shown:

primary_key INTEGER	board_state_hash	depth INTEGER	score INTEGER	move VARCHAR
4000	381557515464957826	2	668	(b, 7, 6, 0)
4008	92792678545287814	1	668	(1, 3, 0, 3)
4007	64267585963801645	2	668	(b, 6, 0, 0)
4006	53648191241824822	1	668	(1, 2, 0, 4)
4005	192108880228050792	2	668	(b, 7, 6, 0)
4004	77657526878403638	1	668	(1, 1, 0, 2)
4003	235000000000000000	2	668	(b, 7, 6, 0)
4002	50895250476736216	1	668	(1, 0, 0, 6)
4001	8510339528805238	2	1938	(2, 6, 0, -2)
4000	423754885353353621	1	1935	(b, 7, -6, 0)
3999	120844146256161636	3	1930	(b, 7, -6, 0)
3998	1241187857639879518	4	1000001	NULL
3997	231000000000000000	2	1000001	(b, 5, -2, -1)
3996	178784390559138637	1	1435	(b, 5, 0, -2)
3995	108876355710465777	1	1435	(b, 5, 0, -1)
3994	1621117848466354948	1	1415	(b, 5, 0, -1)
3993	785120777795654377	1	1425	(b, 5, 0, -1)
3992	185269585795472641	1	1425	(b, 5, 0, -1)
3991	192501781409203929	1	1425	(b, 5, 0, -1)
3990	300000000000000000	1	1435	(b, 5, 0, -1)
3989	185454573935111198	1	1430	(b, 5, 0, -1)
3988	199923254579430380	1	1430	(b, 5, 0, -1)
3987	888879991994993518	1	1430	(b, 5, 0, -1)
3986	5136520871469212	1	1435	(b, 5, 0, -1)
3985	56362108487066432	1	1435	(b, 5, 0, -1)
3984	200303250000000000	1	1435	(b, 5, 0, -1)
3983	177657526878403638	1	1435	(b, 5, 0, -1)
3982	217613606461380354	1	1435	(b, 5, 0, -1)
3981	2019421525826824394	1	1435	(b, 5, 0, -1)
3980	32980078559372910	1	1435	(b, 5, 0, -1)
3979	159871598502619216	1	1435	(b, 5, 0, -1)
3978	122876180301841899208	1	1435	(b, 5, 0, -1)
3977	320000000000000000	1	1435	(b, 5, 0, -1)
3976	913150000000000000	1	1435	(b, 5, 0, -1)
3975	1886423963827374574	1	1435	(b, 5, 0, -1)
3974	738624838600312825	1	1435	(b, 7, 1, 0)
3973	1553043824324132285	1	1435	(b, 7, 1, 0)
3972	3735849487928206	1	1435	(b, 7, 1, 0)
3971	191425378388945172	1	1435	(b, 7, 1, 0)
3970	155555555555555555	1	1435	(b, 7, 1, 0)
3969	147474750119876085	1	1430	(b, 5, 1, 1)
3968	24846610788162348	1	1430	(b, 6, 1, 1)
3963	1921268723262578477	1	1430	(b, 6, 1, 1)
3962	214880127153120975	1	1430	(b, 6, 1, 1)
3961	2046513172726635598	1	1430	(b, 6, 1, 1)

Here you can see that I am playing the chess AI and have got it into check. The AI failed to respond with a move and failed to declare an that the game was over. This was due to an internal server error from the following invalid minimax result:

12	3998	1241187857639879510	4	1000001	NULL
----	------	---------------------	---	---------	------

The value 1_000_001 is used as a starting value for alpha and is an invalid score to return.

I was able to replicate the error by loading the data from the web client into a unit test and repeating the search:

```
def test_specific_bug_null_move(self):
    pieces_matrix = [
        [Rook("B"), King("B"), None, None, None, None, None, Queen("W"),
         None],
        [None, Pawn("B"), Pawn("B"), None, None, None, None, None],
        [Pawn("B"), None, None, None, None, None, None, None],
        [None, None, None, Pawn("W"), None, None, None, None],
        [None, None, None, None, None, Bishop("W"), None, None],
        [None, None, Pawn("W"), None, None, None, None, None],
        [Pawn("W"), None, None, None, None, Pawn("W"), None,
         Pawn("W")],
        [None, Rook("W"), None, None, None, King("W"), None, None]
    ]
    next_to_go = "B"

    board_state = Board_State(next_to_go, pieces_matrix)
    print()
```

```
board_state.print_board()

move_engine = Move_Engine_Prime()
move_engine.cache_allowed = False
move_engine.cache_manager = None
move_engine.parallel = False

# self.assertRaises(
#     AssertionError,
#     lambda: move_engine(board_state, depth=4)
# )

def absolute(x): return x if x >= 0 else 0-x

# _, move = move_engine(board_state, depth=1)
# self.assertTrue(move is not None)
# board_state = board_state.make_move(*move)
# print()
# board_state.print_board()

# _, move = move_engine(board_state, depth=1)
# self.assertTrue(move is not None)
# board_state = board_state.make_move(*move)
# print()
# board_state.print_board()

# _, move = move_engine(board_state, depth=1)
# self.assertTrue(move is not None)
# board_state = board_state.make_move(*move)
# print()
# board_state.print_board()

print(board_state.is_game_over_for_next_to_go())
print(list(board_state.generate_legal_moves()))

score, move = move_engine(board_state, depth=4)

self.assertTrue(
    absolute(score) < 1_000_001 and move is not None,
    msg=repr({"score": score, "move": move})
)
```

The following test helped me reproduce the error but not diagnose it.

I then checked that the game wasn't over in another test (the black king should have one move)

```
# this test is to ensure that the game over function can correctly
identify a game over situation
# The program crashed in a way that hasn't happened since (I must have
been tinkering with the minimax code)
# One of my assertion statements triggered and identified that the
legal move variable was none.
# this ensured that the program could successfully identify that the
game wasn't over and that the king has one move
# to correct the believed source of the bug, I ensured timed calls
always perform at least a depth 1 search to ensure that a move is always
produced.

def test_troubleshoot_bug_game_over(self):
    pieces_matrix = deserialize_client_pieces_matrix(
        json.loads(
            """
                [[[null,null],["B","♔"],[null,null],[null,null],[null,
                null],[null,null],[null,null],["B","♕"]],[["W","♔"],[null,null],[null,n
                ull],[null,null],[null,null],[null,null],[null,null],["W","♕"]],[["B",
                "♔"],[["B","♕"],[["B","♔"]],[[null,null],
                [null,null],[null,null],[null,null],[null,null],[null,null],[null,
                null],[[[null,null],[null,null],[null,null],[null,null],[null,null],[n
                ull,null],[null,null],[null,null],[null,null],[null,null],[null,null],[n
                ull,null],[null,null],[null,null],[null,null],[null,null],[null,null]]]
            """
        )
    )

    board_state = Board_State(
        next_to_go="B",
        pieces_matrix=pieces_matrix,
        check_encountered=True
    )

    # board_state.print_board()

    legal_moves = list(board_state.generate_legal_moves())

    # print({"legal_moves": legal_moves})

    self.assertEqual(
        legal_moves,
        [(Vector(i=1, j=7), Vector(i=1, j=0))]
    )

```

```
over, _ = board_state.is_game_over_for_next_to_go()
# print(over)

self.assertTrue(not over)
```

This test work and passed without issue. This meant that the issue was with the minimax function as my chess engine could successfully identify that there was one remaining legal move.

I didn't figure out what caused the issue but I added checks within my code as part of a defensive design strategy to prevent the issue causing unexpected behaviour.

I added the following code to the Timed_Move_Engine:

```
# while there is time left, keep searching to a greater depth
while not time_used_up():
    # print(f"Iterating again: time_used_up() -- {time_used_up()}")
    try:
        # deepest_result =
        self.minimax_first_call_parallel(board_state=board_state, depth=depth,
                                         is_time_expired=time_used_up())

        # search the tree in parallel if needed
        result =
        self.minimax_first_call_parallel(board_state=board_state, depth=depth,
                                         max_time=self.max_time, variable_depth=0)
        score, move = result

        # if the move or score are invalid then raise the
        appropriate error
        if absolute(score) == 1_000_001 or move is None:
            raise UnexplainedErroneousMinimaxResultError()

    except UnexplainedErroneousMinimaxResultError:
        # I don't know the cause of the error but I can catch it
        here and prevent it causing issues and producing unexpected behaviour
        break
    except TimeOutError:
        # print(f"Breaking: time_used_up() -- {time_used_up()}")
        break
    else:
        # print(f"Completes depth={depth} so incrementing depth")
        # print(f"Result at depth {depth}: {deepest_result}")
```

```

        # if neither the time is used up or an erroneous result
was produced, keep searching at a greater depth
        deepest_result = result
        depth += 1
    
```

The idea was that I would detect this invalid result and not use it, using the previous result.

I also added code to the cache manager to prevent invalid results from being added to the database cache:

```

def add_to_cache(self, board_state: Board_State, score, depth, move):
    # this function adds to the cache
    # this function assumes that no move valuable cache already exists
    # (it overwrites all other cache)
    assert self.engaged, "Context manager must be used"

    # this is used to tackle any bugs elsewhere in the program
    # it ensures that no invalid cache items are added to the database
    def absolute(x): return x if x >= 0 else 0-x
    if absolute(score) > 1_000_000 or (move is None and depth > 0):
        # don't add erroneous data to cache
        return None
    
```

These 2 combined changes fixed the issue and prevented the unexpected behaviour from causing the program to crash or from producing an invalid output.

I have played many more chess games since and have not encountered this bug again. I tried beating the chess program and I was able to, this made me confident that the issue has been fixed.

Here is screenshot evidence that I can now beat the chess program without this issue occurring.

Stakeholder Feedback:

I received feedback on my iteration from my stakeholders. Here is a summary of the feedback:

GUI:

- They thought that the user interface was clear
- They could clearly see all the pieces on the board
- They were able to intuitively move a piece
- They liked the highlighting as it made it clear where the pieces could move and which pieces the opponent was moving
- They thought that it was clear how to change the difficulty and could see a noticeable change to how long the computer was thinking
- They thought it was clear what reset and concede buttons did
- They thought that the information conveyed by the pieces taken and move history tables was clear

Special Features:

- They appreciated that I had implemented promotion of pawns into queens
- They thought that the feature to reload the game state after closing the tab was really useful and very convenient

Reliability (after the fixes from my testing):

- They didn't encounter any bugs or logic errors.
- The website was able to properly handle check and when the game was over
- The website prevented the user making invalid moves and the computer only made valid moves

Suggestions for improvement:

- In landscape, the white space on the left and right of the chess board could be better utilised. For example you could put the move history there.
- In addition, there is no hint to say that you should scroll down for extra content (the move history and pieces taken table)

Evaluation (just of prototype 3):

Overall I think that this prototype has been very successful. Despite many of the test of the website failing on the first try, all issues were fixed. My users also didn't have any issues with the program producing invalid outputs or bugs or any kind. Once I had fixed some key bugs, their feedback was that the program was very reliable.

I believe that this iteration meets all the design objectives in their entirety. I therefore believe that this final prototype has met all items of the success criteria that were listed high-priority, desirable and essential. Considering the time constraints I am very proud of this and believe that the final iteration will serve my stakeholder's needs well.

Evaluation: Evaluative Testing:Post-Development Testing

To evidence that I had completed my test I used video clips. This was appropriate as it would have been difficult to show how the game changed in response to each click with screenshots:

Here is the link to the external folder that contains these video clips:

https://drive.google.com/drive/folders/1iYk_W6cmezXJgGBniHZUgc_YrKSGTV7z?usp=sharing

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
1	Different Difficulty Settings	Needed for stakeholder (casual chess player). Success criteria item 1	Valid	Clicked radio buttons then looked for a change in response time for the computer move	The response time should change so that it is approximately the same as the listed time (e.g. medium

				is 5 seconds) plus an extra second delay.
Actual Result	Evidence		Success?	
This did cause a notable change in response time for the computer move.	<p>I can change the radio button easily as shown. The response time change was noticeable. See video clip 1.</p> <p>Change Difficulty: (AI thinking time)</p> <ul style="list-style-type: none"> <input type="radio"/> Trivial (1 sec) <input type="radio"/> Easy (2 sec) <input checked="" type="radio"/> Medium (5 sec) <input type="radio"/> Hard (10 sec) <input type="radio"/> Challenge (15 sec) <input type="radio"/> Extreme (30 sec) <input type="radio"/> Legendary (60 sec) <p>Change Difficulty: (AI thinking time)</p> <ul style="list-style-type: none"> <input type="radio"/> Trivial (1 sec) <input type="radio"/> Easy (2 sec) <input type="radio"/> Medium (5 sec) <input type="radio"/> Hard (10 sec) <input type="radio"/> Challenge (15 sec) <input checked="" type="radio"/> Extreme (30 sec) <input type="radio"/> Legendary (60 sec) 		Full Success	

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
2	Client Side Validation	Test that I cannot input an invalid move or move the opponents pieces on my turn.: Success Criteria items: 2 and 28	Invalid	Clicked on my piece than an invalid square, then tried to click and move the opponent's pieces	The invalid click events are ignored, they will only affect the highlighting.
Actual Result	Evidence		Success?		
The result was as expected, bogus click events were ignored	See clip 2		Full Success		

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
3	Client Side Validation	To meet success criteria items 3 and 29, the user cannot move when it is not the user's turn. The move that the computer produces should then be valid	Invalid	Set difficulty to high and then try to move while it is the opponents turn. The computer then made a move which was valid	These clicks will be ignored and there will be no corresponding output. The computer's move should be a valid legal move
Actual Result	Evidence		Success?		
The expected result, invalid inputs were ignored. The move	See clip 3		Full Success		

	that the computer made was valid		
--	----------------------------------	--	--

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
4	Check: testing that a message is used to show check and testing that the user cannot input invalid moves that cause their king to be in check	To meet success criteria items 4 and 11 and to ensure that the program has the relevant and does not allow invalid inputs	Valid (for message) And Invalid (try to make invalid move)	I play the game until I had moved my king into check	The computer should make use of the main title to show that it is my go and I am in check. It should show with highlighting that I cannot make moves that violate the rules of check. When I try to make these moves anyway, they should be ignored
Actual Result	Evidence				Success?
The title message identified check when if occurred. I couldn't move in a way that caused check. When I was in check, the highlighting showed the correct legal moves, I couldn't input an invalid legal move	See clip 4				Full Success

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
5	All aspects of highlighting in both the user and computer move. As well as checking that the user and computer can move	This will indicate if my usability feature to show the moves and legal moves with highlighting is working. I will also see if the computer can make a move and if it is shown with highlighting This corresponds to items: 5,6,7,8 from the success criteria	Valid	I implemented user moves step by step to see the highlighting at each part of the process.	I can click one of my pieces, it will be highlighted red and everywhere it can move to will be highlighted green. The computer move will be highlighted before it happens. The moving piece will be red and the square it moves to will be green.
Actual Result	Evidence				Success?
Exactly as expected	See clip 5				Full success

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
----------	---------------------	-----------------------	--------------	----------------	-----------------

6	<p>The Chess AI should be capable of making smart moves, it should also be able to learn over time.</p> <p>I will also try to show that I have improved the algorithms efficiency by using multiple codes</p>	Tests success criteria item 9, 26, 27	Valid	I will make a series of moves to lay a trap where it trades a knight for a pawn	on trivial difficulty it should fall for the trap. Then on extreme difficulty and all later games it should avoid the trap. I should be able to show this with the database.
Actual Result	Evidence	Success?			
As expected, at a greater difficulty it avoided the trap. This cache then allowed a low difficulty to also avoid the trap	See clip 6	Full Success			

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
7	Computer's ability to put me in checkmate. And the appropriate game over output	This will test that the AI is trying to actually win and will test how the program uses outputs to clearly show when the game is over. This corresponds to success criteria items 10, 12, 13	valid	I will try to put myself in checkmate	The computer should be able to put my king in checkmate. The main title should then show that I am in check mate. The board should then be disabled
Actual Result	Evidence	Success?			
As expected, the AI took the option of putting me in checkmate over other options like trying to take more pieces. Then the title indicate the game was over and the board was disabled	See clip 7	Full Success			

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
8	Bulk testing of many features. I will test: The move history and pieces lost tables work; the reset and concede buttons work	This will test various features are working as intended. It will test that the program meets success criteria item: 15, 30, 31, 32	Valid	Checking the contents of the tables and clicking the buttons	The tables should be able to accurately keep track of the chess game (moves and pieces taken) as I make moves. The buttons should be able to end the game and then reset it.

	and that reloading the webpage restores the game. I will also check that the server connection is fast enough			This includes resetting the move and pieces tables
	Actual Result	Evidence		Success?
	As expected the tables and buttons all worked	See clip 8		Full success

Test No.	Aspect Being Tested	Justification of test	Type of test	Test Data Used	Expected Result
9	Bulk testing of many features. I will test: that reloading the webpage restores the game. I will also check that the server connection is fast enough. I will also verify that the program works in different aspect ratios.	This will test various features are working as intended. It will test that the program meets success criteria item: 17, 19, 25	Valid	I started a new game, reloaded the tab. Then show the response speed and then different aspect ratios	I expect the response speed to be fast. The game should be reloaded when the tab is reopened. The website should work on: <ul style="list-style-type: none">• PC / Laptop• Tablet Landscape• Phone portrait
	Actual Result	Evidence		Success?	
	The reloading of the game worked fully. The latency from the server was low (response times were fast). Some aspect ratios worked perfectly while others struggled to fully render the move history table but could render everything else.	Clip 9		Partial success Some aspect ratios for mobile devices struggled to render the move history table. It was either too large or not fully shown. Other aspect ratios worked perfectly well. Test objectives 17 and 19 were fully achieved	

Usability Survey:

I had an extensive discussion with a stake holder that plays casual chess on a phone regularly. I asked him to play around with the final product and comment on how well it met his needs:

He said:

"The website has an overall clear appearance and it is clear what the pieces are.

Doing what I would do intuitively, I will click a pawn. It now shows where the pawn can move and after clicking it, it has now moved.

The highlighting makes it clear which black piece has moved and how it has moved.

I now realise that scrolling down there is extra information

It is clear how to change difficulty

The pieces taken and move history are clear. I understand exactly what they show.

I have made another move as expected.

I will now try to take a piece. I like how the piece is highlighted green.

I like the choice of cream and brown colours for the board, it makes the pieces distinct.

I will try conceding.

The game has stopped .

I will try to restart.

It is clear what the buttons are.

I will try closing the tab, the game is still there. That is really convenient neat as you can just continue the game

A possible improvement could be to make use of the space either side of the chess board in landscape

"

I found this feedback very useful. It acts as good evidence to suggest that I have succeeded in making the website clear and accessible (success criteria item 24).

I will make sure to use a few of the points that came up in the questionnaire:

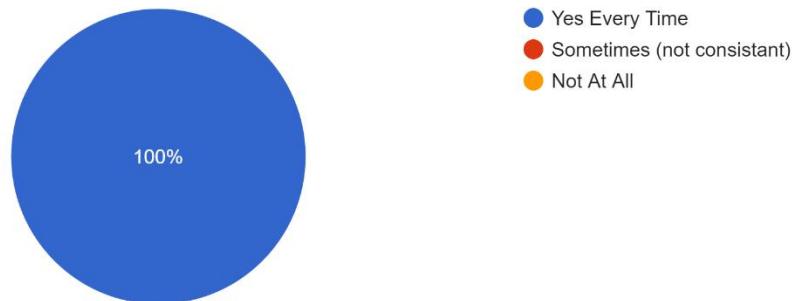
- I will ask if it is intuitive: what to do when the website loads.
- I will ask if people immediately noticed the difficulty setting and tables below the buttons.

I used a google form to conduct a questionnaire of stakeholder to get their feedback on the usability features of the chess program:

Usability Survey Results

Did the webpage for the chess game load successfully?

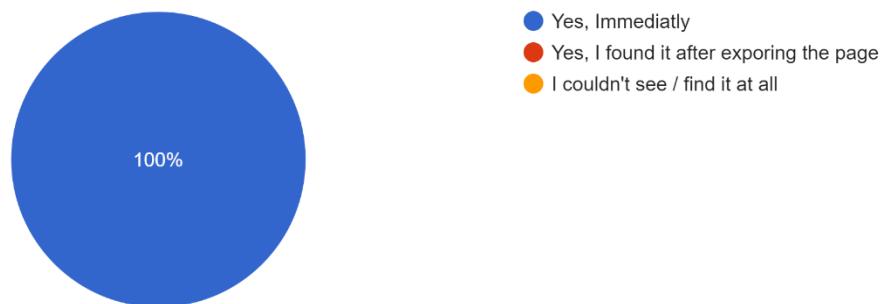
6 responses



This shows that the webpage is reliable.

Could you see the chess board aswell as the main title and Reset & Concede buttons?

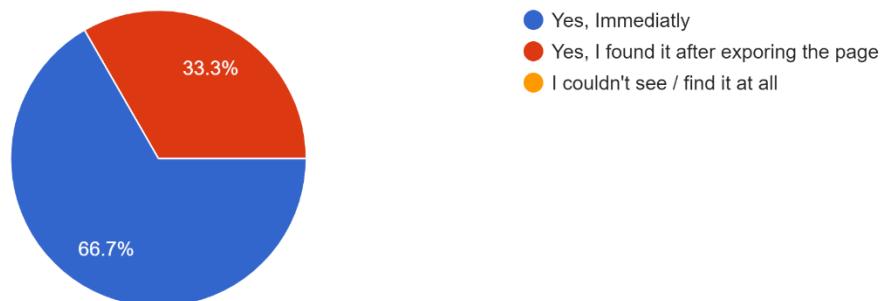
6 responses



This is expected as I put these elements in immediate view as they were the most important.

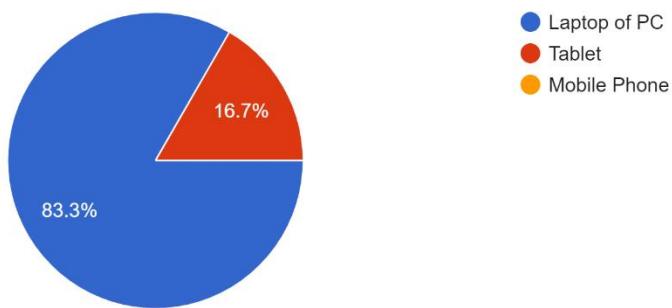
Could you see the how to change the difficulty aswell as that pieces taken and move history tables

6 responses



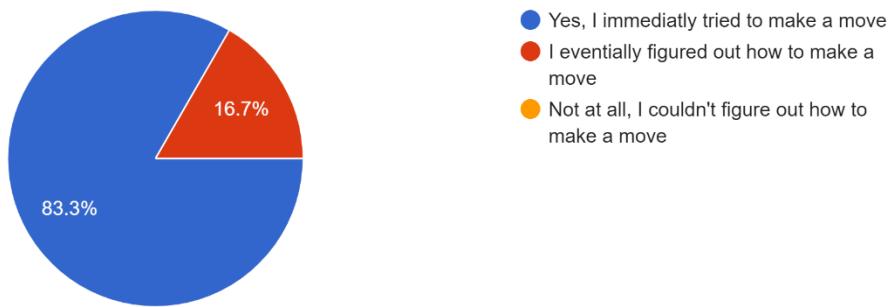
This feedback shows that the program would benefit from some kind of text or arrow to show that the user could scroll down for more information and settings.

What device did you load the website on? (select the most applicable option)
6 responses



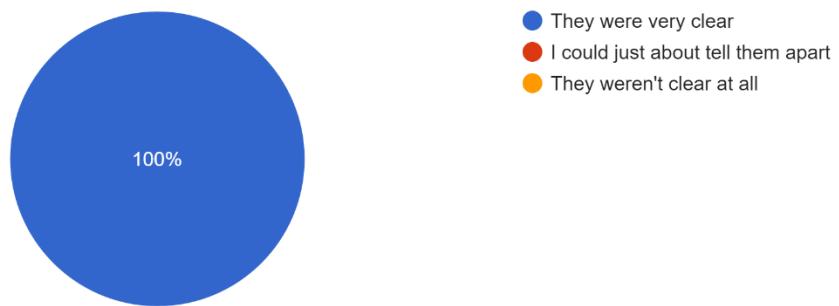
This likely shows that the stakeholders I was able to collect data from are not a representative sample of the whole population as they are mostly using laptops to play with the website. A larger sample may show that more users will use tablets.

Was it intuitive to figure out what you should do when the page loaded?
6 responses



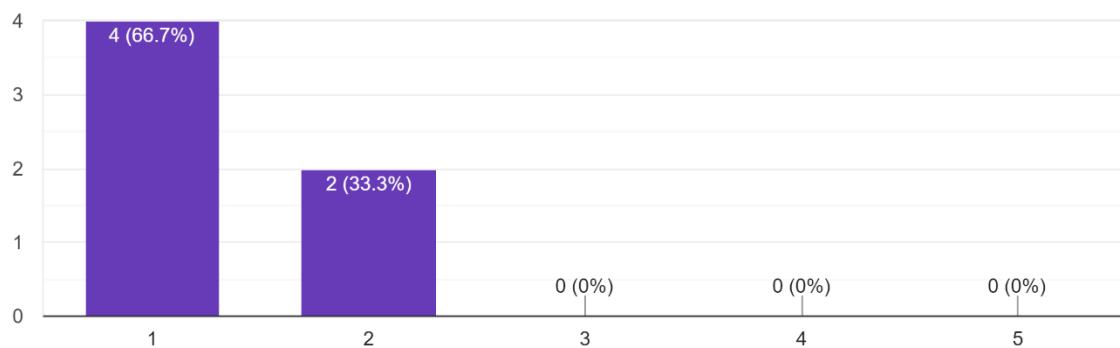
This question shows that the input method for making a move was intuitive. This is likely partly due to the interface is similar to many other chess websites and programs that these stakeholders may have already encountered. It is encouraging as it shows that the program is easy to use.

Were the pieces clear (could you see which pieces they were and their color) on the chess board?
6 responses



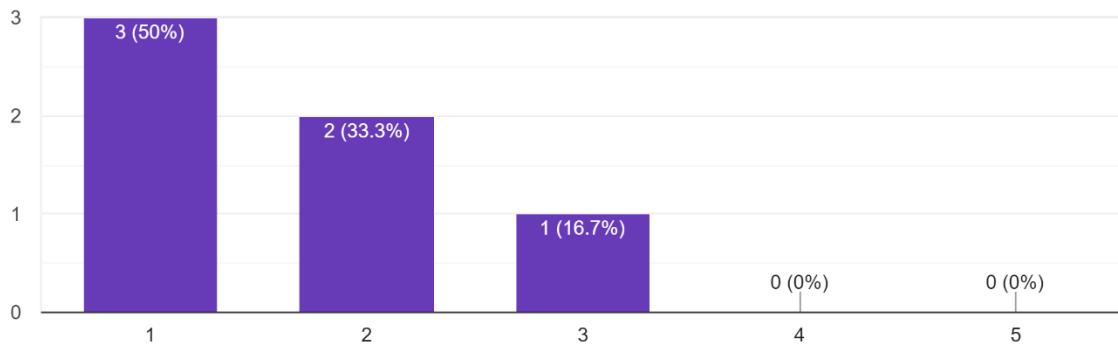
This response shows that the cream and brown colours used for the board, combined with the text shadow were able to contrast the pieces against the board. This made them very clear to users.

On a scale of 1 to 5 (1 is invaluable): How useful did you find the different difficulty settings?
6 responses



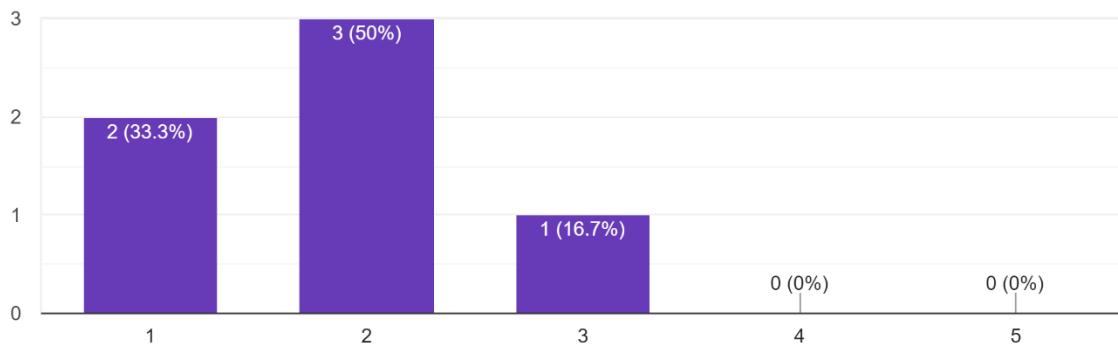
This shows that having a range of different difficulty settings is a feature that is appreciated by casual chess players (the stakeholder / target niche). This allows people to play an easy or challenging game of chess, whatever their level. It is likely that the main issue that some people had with the difficulty setting was that each additional difficulty took longer to decide the computer's move.

On a scale of 1 to 5 (1 is invaluable): How useful did you find the reset and concede buttons?
6 responses



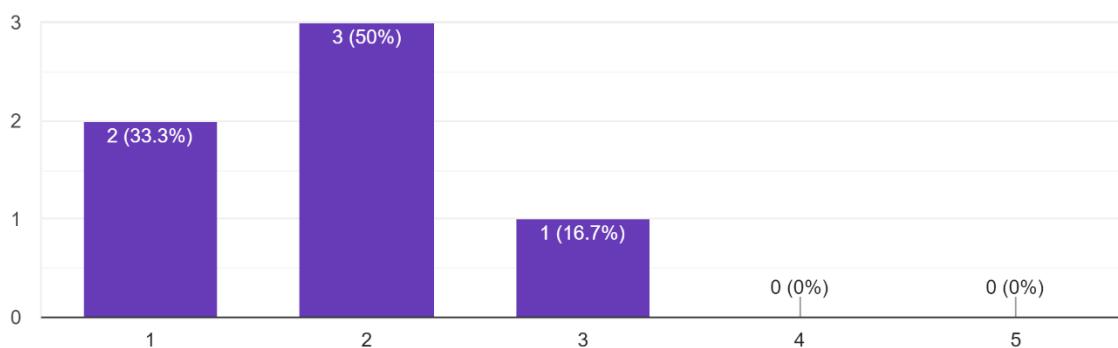
This shows that the ability to reset the games was also appreciated. It seems that some people didn't see a use for the concede game button.

On a scale of 1 to 5 (1 is invaluable): How useful did you find the pieces taken table?
6 responses



On a scale of 1 to 5 (1 is invaluable): How useful did you find the move history table?

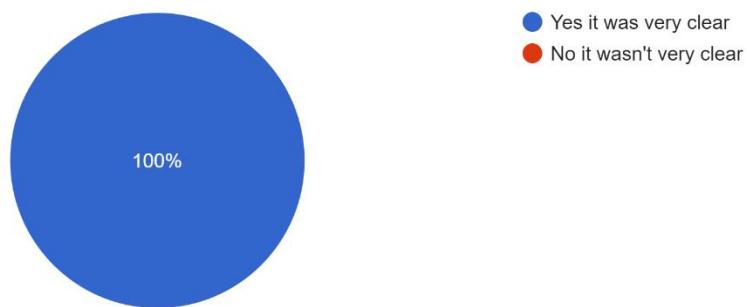
6 responses



These results suggest that people had some issues with these tables. It is likely that the fact that they had to know to scroll down contributed to this. In addition the lack of chess squares laid on the board (e.g. so that the user could easily find the square B5) means that some player cannot make full use of the information in the move history table.

Was all the text clear and readable?

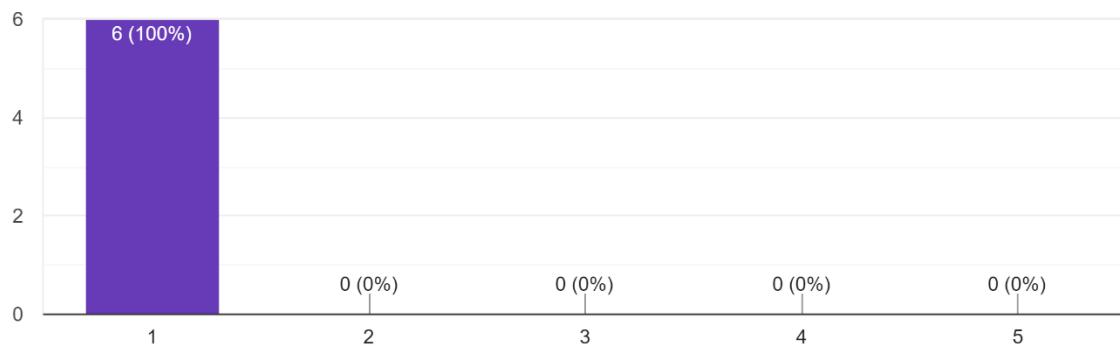
6 responses



This is very encouraging. It shows that efforts to maximise contrast and make font sized big enough have allowed users to clearly see what is going on.

On a scale of 1 to 5 (1 is invaluable): How useful did you find the highlighting (red and green) that was used to show chess moves?

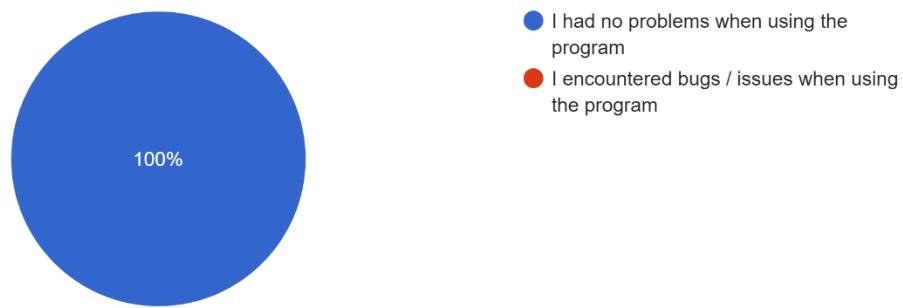
6 responses



This shows that the highlighting is by far the most popular feature. Users seemed to like how it made moving pieces intuitive. It also helps new players who are not yet aware of where all the pieces can move. The use of highlighting to show the computer move seemed to also be appreciated as it drew the user's eye and made the computer's move clearer.

How reliable was the program? Did it crash or cause any errors?

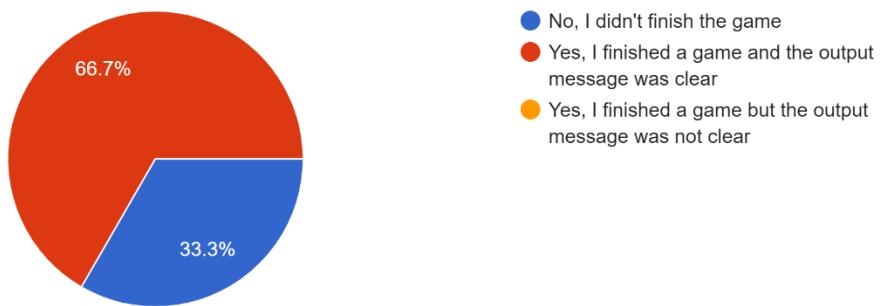
6 responses



This shows that my testing and validation has paid off as the program is now very robust. None of the users had issues where the program crashed. This is inline with the results of my post development testing.

Did you finish a chess game? if so was the resulting output (declaring the game over) clear?

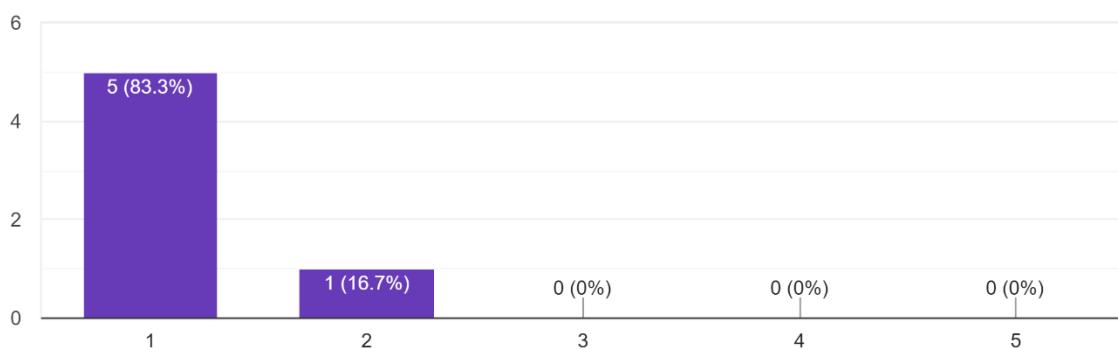
6 responses



As many of the stakeholders were only able to test the program for a limited time, some were not able to fully finish a chess game. However the 4 who were able indicated that they felt that the resulting message and the disabled board was clear and intuitive.

On a scale of 1 to 5 (1 is really enjoyable): Did you enjoy using the program?

6 responses



This feedback is very encouraging. It shows that I was able to meet my stakeholders main need: entertainment. All of my stakeholders enjoyed that game. This indicates that the game meets the needs of casual chess players well.

Breakdown of success criteria (what has been achieved):

Number	Feature name	Importance	Achieved	Evidence	Evaluation
1	Difficulty settings	essential	Fully	Post development test 1 (and also test 6)	I used radio buttons to input the difficulty settings. I created 7 different difficulties that affected how long the AI had to explore the decision tree. I felt this was important as casual chess players could vary in skill and may not always want a challenge.

2	Validation client side	Essential	Fully	Post development test 2 (and also test 3)	As soon as the website opens, the computer waits for the user's move. The user is always white and has unlimited time to think about their moves.
3	Validation client side	Essential	Fully	Post development test 3 (and also test 2)	I used both client and server side validation to ensure only the right player can make a move at any given point in the game
4	Validation client side	Essential	Fully	Post development test 4	I showed how the highlights showed that the user couldn't make some moves. I then showed that any inputs what tried to make these invalid moves were ignored
5	Intuitive Move Input with dot markings	High priority	Fully	Post development test 5	I implemented this feature fully and it was well received by users. It makes it clear where the user can and can't move a piece. It also helps users who are not that familiar with the game of chess.
6	Graphically show the move	Essential	Fully	Post development test 5	Once the move is inputted, the new board state is shown to the user (with their piece moved to the new square). There is a purposeful delay before the computer move is added to ensure that I don't have too much happening at once. This lets me clearly show the user that their move has been implemented
7	Computer Move output	Essential	Fully	Post development test 5	The computer is able to move the black pieces on the board to successfully act as an opponent to the user.
8	Highlighting	High priority	Fully	Post development test 5	Highlighting for a short time shows what piece the computer is moving and where the piece is going. Stakeholders found this feature helped them see exactly what move the computer had made.
9	Chess engine mid game	Essential	Fully	Post development test 6	The Chess AI was able to make "smart" decisions when it was able to sufficiently search the tree. This allowed it to avoid losing its pieces and to take my valuable pieces
10	Chess engine late game	Essential	Fully	Post development test 7	The Chess AI is able to pursue checkmate in the late game, once it already had a significant piece advantage. This help the user experience as there AI opponent is really trying to win. This should make the game more interesting.
11	Alert after each move for check	Essential	Fully	Post development test 4	I made use of a large title to show which users turn it was as well as to make it clear, when a player was in check. I believe this provided some necessary direction to users (about whose turn is was).

12	Proper recognition of game over and its properties	Essential	Fully	Post development test 7	I again used the main title element to output to the users that the game was over. I also disabled the board. This allowed users to still see all the pieces, without being able to move them.
13	Chess engine aware of different game over outcomes	Essential	Fully	Post development test 7	It is able so successfully decide to pursue winning by checkmate over taking valuable pieces. It will also avoid losing by checkmate at all cost. It is aware of stalemate (normal and 3 repeat stalemates). It will try to avoid a stalemate when it is winning and cause one when it is losing
14	Undo button for last user move	Low priority	Not Implemented		This would have been a nice feature as it could have helped users who miss-clicked. In the end I already planned to implement so many other features which meant this feature would only be implemented if I had time to spare.
15	Log of history of moves in game	Ideal / desirable	Fully	Post development test 8	This feature works fully. It logs the user's and computer's moves in a table. This includes whenever a piece is taken. The feature could be improved by adding numbers and letters down the side of the chess board to allow new players to locate squares that it refers to.
16	Step back through log	Low priority			This functionality would have required significant time to implement as additional complexity would have to be added to both the front end and the backend.
17	Fast and reliable connection between front and backend	Essential	Fully	Post development test 9	The server and client browser are connected by WebSocket connections which have extremely fast due to using a direct TCP/IP connection. This meant that the final product didn't feature any lagging.
18	Allowing users to login & sign up	Low priority	Not implemented		This feature would have required huge amount of additional work. I would have needed to update the database; front end and backend validation; made a new python module to handle users and make significant changes to the user interface to allow for a login system. As I was able to implement a save game feature without users, it hardly seemed worth the steep time and opportunity cost
19	Reloading unfinished game	Ideal / Desirable	Fully	Post development test 9	This functionality was added using cookies and a database. I am proud of this feature as I think that it is technically impressive and very useful.
20	Puzzles	Low priority	Not implemented		This feature was written of early in the design process for being too much work.

21	Draughts	Last add, very low priority	Not implemented		This feature was written off early in the design process for being too much work and deviating from chess
22	Allow users to start with a custom game layout	Low priority	Not implemented		This would have been a nice feature to add. I could have allowed the user to be black and go second. I think that overall it makes sense for the user to go first as the computer has many advantages (such as looking at more possible board states). As such, I don't think that much was lost in not implementing this feature
23	Leader board and trophy system	After interview I no longer want to implement this (lowest possible priority)	Not implemented		I wrote this feature off early in the analysis section. I believe that the idea of a leader board is more suited to competitive players than casual players.
24	Accessibility through website	Essential	Fully	I cite the usability interview and questionnaire as evidence as they show that the website is clear and intuitive	I aimed to ensure that the layout of the page and color, size and font of the text were as clear as possible. I prioritised this over aesthetics (hence the background color is white). I think that maximising clarity will benefit all stakeholders and the program's usability as a whole. I particularly think that it will benefit older and less tech-savvy stakeholders.
25	Mobile access	Desirable	Partially	Post development test 9	I think that the efforts I did make to achieve this feature were still important. The final product should work well enough on most aspect ratios. This should allow the elderly who struggle to use PCs to play on tablets. This should also allow people on the go to play on their phones.
26	Efficient adversarial AI	Highly desirable	Fully	See detailed explanation of improvements in prototype 3 (such as parallelisation). The use of parallelisation and database cache was shown in post development test 6	I am proud of my effort here. While I still am not able to quickly perform a depth 3 search, I brought the time needed to perform a depth 2 search of the starting positions from around 25 seconds to 12 seconds. I think that making depth 2 more possible is important as this lets the computer apply game theory to anticipate the user and make interesting decisions (like when we saw it avoid the trap).

27	The AI adversary feels as organic and real as possible	Highly desirable	Fully	Post development test 6	I think that allows with making depth 2 searches more possible, and use of a database allowed allows the chess AI to learn make the AI a more real and dynamic opponent that is more fun to play against.
28	Validation of users move	Essential	Fully	Post development test 2	I added both client and server side validation to ensure that the user couldn't make an invalid move.
29	Valid computer move	Essential	Fully	Post development test 3	I added checks within my code to catch erroneous and invalid moves made by the minimax algorithm and prevent them reaching the user and causing unexpected behaviour.
30	Pieces Taken output	Desirable	Fully	Post development test 8	I added this feature as a way for users to see at a glance who is winning. This can be helpful information that a casual player might want.
31	Restart game button	Essential	Fully	Post development test 8	The restart button allows the user to play a fresh new chess game. This can be done when the game is ongoing or when it is over. This allows users to start again if they want to (for example if they lose their queen).
32	Concede game button	Essential	Fully	Post development test 8	The idea of this button was that it could stop the game progressing while still allowing the user to see all the pieces and the move history. This would stop the computer's move coming through and changing the board state. Some players may also want the ability to concede (as they can in a real chess game).

Limitations and Maintenance:

This program is only a prototype and has various limitations that could get in the way of it being fully deployed. These include limitations of the program itself and difficulties in maintaining the system at scale.

Limitations with the program itself include:

- One of the main limitations of the program is that the database is currently all stored within one file. This would pose scalability problems if the program were to be used by many people. The solution would be to use a more scalable database system. For instance, the database could be implemented using a cloud based system such as AWS. This would make the database far more scalable and able to hold many more unfinished games
- Another limitation is that I didn't implement all the moves of chess. I did implement pawn promotion but didn't implement moves like en passant and castling. If this chess program

were to become as popular as some of its competitors like Lichess, it would need to have these moves implemented.

- Exponential time complexity $O(2^n)$ of the minimax search algorithm limits the depth to which the decision tree can be searched. This functional caps the difficulty of the chess AI. A different technique (e.g. machine learning rather than heuristics) would need to be used to create a chess AI like stockfish that can challenge expert and pro chess players. I have tried to work to maximise the challenge by improving the efficiency of my minimax function.

As part of maintenance, certain issues may arise:

- For one, the database may accumulate to many unfinished games. While each one is only around 4KB of binary data, this could slow that database and present an unnecessary cost if it got out of hand in future.
- Another issue is that the database may gather so many cached results that has a very strong opening at all difficulties. This could erode the concept of difficulty settings and prove a problem for new players.
- One issue that a future maintenance team would have to deal with would be technical debt. This describes the cost of committing to a given system, language, protocol or framework. In the future, Flask or WebSocket or SQL-Alchemy could become deprecated frameworks that are no longer supported. This would make maintaining the system more difficult and costly. New errors could be created from a no longer maintained framework or users trying to use the program in a way for which it wasn't designed (e.g. with a new type of browser that the website isn't set up for)
- Another difficulty for a future maintenance team would be quick and dirty fixes to problems. These are fixes that do not address the root cause of the problem and instead aim to prevent the problem from having a negative effect. This could be seen as bad practice as problems that are swept under the rug are allowed to build up. I only added 1 such fix in the form of my fix to the erroneous minimax result encountered in testing in prototype 3. In this case I made such a fix as the problem was complex to solve and I had limited time available. If I had added many such fixes to the program (I didn't) this could make maintenance difficult. This is because it adds clutter and additional complexity to the code. It also makes it more likely that future changes will be difficult as they will cause many of these temporary fixes to fail.

Some of the key ways in which this chess program could be improved:

- I could create a help feature to make the program move usable for stakeholders that are not as tech-savvy or for users that are confused about what to do.
- I could have added the login system (success criteria item 18) to the program. This would have allowed users to save their games to their profile, allowing them to continue the game on any device. This could have also allowed for other features like a leader board (success criteria item 23)
- I could have added more complex moves including castling and en passant (I did add pawn promotion to prototype 3). This would allow a user to play a full game of chess with all the proper rules. This would also prevent them needing to change their game plan (e.g. many chess players plan to castle).
- I could implement the stakeholder feedback about the user interface. Specifically I could make better use of the white space that is either side of the chess board in landscape. I could also add a note or arrow to indicate that there is additional content (the difficulty settings, moves history and pieces taken table) if you scroll down.

Appendix:

Sources:

(I have directly sighted some sights for a diagram or pieces of information already in this document)

URL	How I used the source	Date
https://www.chessprogramming.org/Simplified_Evaluation_Function	I used the pieces matrices from this website to make my static evaluation function (heuristic used)	October 2022
https://stackoverflow.com/	I used this source to help me understand how to debug an issue that someone else had also had. The answer system could usually directly answer my question or direct me to another source that could help.	August 2022 till the end
https://www.w3schools.com/	I used this website to help me with HTML, CSS and JavaScript	Since August 2022 till the end
https://youtu.be/STjW3eHOCik	I used this lecture to help me understand the minimax algorithm and how you might go about improving it	August 2022
https://youtu.be/I-hh51ncgDl	I used this video to help me understand how alpha beta pruning works and to get a better understanding of the minimax algorithm	August 2022
https://youtube.com/playlist?list=PL4cUxeGkC9hYYGbV60Vq3IXYNfDk8At1	I used these tutorials to help me understand how to use the VUE js framework (not used in final product)	December 2022
https://cdnjs.cloudflare.com/	I used Cloudflare to access a free copy of the WebSocket and crypto (used for hash function) library for use in my webpage. I copied the following script tags directly from the website:	Since August 2022
	<pre> <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js" integrity="sha512- 9mpsATI0KClwt+xVZfbcf21J8IFBAwsubJ6mI3rtULwyM3fBmQFzj0It4tGqxLOGQwGfJdk/G+fA NnxFq9/cew==" crossorigin="anonymous" referrerpolicy="no-referrer"></script> <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto- js/4.1.1/crypto-js.min.js" integrity="sha512- E8Q5vWZ0eCLGk4km3hxSsNmGwbLtSCSUcewDQPQWZF6pEU8G1T8a5FF32w01i8ftdMhssTrF/0h yGiwonTcXA=="</pre> <p style="text-align: center;">crossorigin="anonymous" referrerpolicy="no-referrer"></script></p>	
https://nodejs.org/en	I used this site to download the node runtime so that I could install Vue.js	December 2022

https://vuejs.org/guide/quick-start.html#creating-a-vue-application	I used this site to provide additional sources for how to make a Vue.js app	December 2022
https://mermaid.live/	I used this online editor and mermaid code to make some of my diagrams	January 2023
https://drawio-app.com/	I used the free version of this app to make some of the diagrams	October 2022
https://sourceforge.net/projects/freeplane/	I used Freeplane to make some of the diagrams	October 2022
https://www.tutorialspoint.com/javascript/index.htm	I used some of these tutorials for reference when creating the JavaScript code	September 2022
https://www.sqlalchemy.org/	I used the official source for documentation of the SQLAlchemy library	January 2022
https://marshmallow.readthedocs.io/en/stable/	I used the official source for documentation of the Marshmallow library	January 2022
https://flask.palletsprojects.com/en/2.2.x/	I used the official source for documentation of the Flask library	January 2022
https://flask-socketio.readthedocs.io/	I used the official source for documentation of the flask-socketio library	January 2022

Code Listing:

Prototype 1:

directory structure:

```
PS C:\Users\henry\Documents\computing coursework\code list\prototype 1> tree /f
Folder PATH listing for volume OS
Volume serial number is 4ED8-B070
C:.
    app.py
    logger.py

    game_engine
        game_engine.py

    static
        main.js
        style.css

    templates
        index.html
```

app.py

```
# imports
# built in libraries
import flask
# import os
from flask_socketio import SocketIO

# my local code
# import logger as logger_module
import game_engine as ge

# setup flask app
app = flask.Flask(__name__)

# setup sockets
socketio = SocketIO(app, async_mode=None)
# RuntimeError: You need to use the gevent-websocket server
# issue solved by running with 'python -m flask run'

# define socket handler for move request
# decorator on move request sent from client
@socketio.on("server_move_request")
def handle_server_move_request(msg):
    # unpack json data and load into object
    game_state: ge.Game_State = ge.Game_State(
        board_positions=msg['board_positions'],
        to_go_next=msg["next_to_go"],
        moves_left=msg['moves_left']
    )
    # use game engine module's main function to determine the best child state
    # (game state following best move)
    best_child: ge.Game_State = ge.main(game_state)

    # code used for debugging to print out the move
    # print('selecting move:')
    # for row in best_child.board_positions:
    #     print(row)

    # now use emit to give a response in the form of an updated board matrix
    socketio.emit(
        "server_move_response",
        {"board_positions": best_child.board_positions}
    )
```

```
# define flask api route handlers
# this endpoint responds with the index.html page which will then load and
request files from the static folder.
# this is the main endpoint that hosts the
@app.route('/', methods=['GET'])
def index():
    return flask.render_template('index.html')

# this function simply runs the app and configures the port that is used.
def run_app():
    app.run(host='127.0.0.1', port=5000, debug=True)

# this code runs the main app function only if this file is run directly and
not if it is imported
if __name__ == '__main__':
    run_app()
```

logger.py

```
# code from: https://realpython.com/python-logging/
import logging

def logging_decorator_factory(lgr):
    def decorator(function):
        name = function.__name__

        def wrapper(*args, **kwargs):
            lgr.info(f"Beginning execution of function: {name} with
arguments {args, kwargs}")
            try:
                result = function(*args, **kwargs)
            except Exception as e:
                lgr.error(f"execution of function: {name} with
arguments {args, kwargs} failed with exception {str(e)[:80]}")
                raise
            else:
                out = str(result).replace("\n", " ").replace("\t", " ")[:200]
                lgr.info(f" function {name} finished and returned: {out}")
            return result
    return decorator
```

```
wrapper.__name__ = name

        return wrapper
    return decorator


# def setup_logger(name, file_path='log.log', c_level='INFO', f_level='INFO'):
def setup_logger(name, log_file_dir, level="INFO", clear=True):
    # clear log file:
    if clear:
        with open(log_file_dir, 'w') as file:
            file.write("")

    # Create a custom logger
    logger = logging.getLogger(name)
    # Create handlers
    c_handler = logging.StreamHandler()

    f_handler = logging.FileHandler(log_file_dir)
    # f_handler.setLevel(file_level)
    # c_handler.setLevel(console_level)
    logger.setLevel(level)

    # Create formatters and add it to handlers
    c_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
    f_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
    c_handler.setFormatter(c_format)
    f_handler.setFormatter(f_format)

    # Add handlers to the logger
    logger.addHandler(c_handler)
    logger.addHandler(f_handler)

    logger.info(f"This logger named {name} created")

    return logger


if __name__ == "__main__":
    setup_logger(name=__name__)
```

game_engine.py

```
# the data class library is used to minimize boiler plate code in defining
__init__ and __eq__ methods for classes
from dataclasses import dataclass
# this is used to make a complete and distinct copy of a 2d array so changes
# to the original don't affect the copy
from copy import deepcopy

# this is a class to represent the game state
# its attributes cannot be altered after it is initialized (immutable).
# instead changes to the game state, such as a move, are handled as new game
state objects
@dataclass(frozen=True)
class Game_State:
    # these variables hold the data relevant to a game state
    board_positions: list[list]
    moves_left: int
    to_go_next: str

    # used for debugging so that the game state can be represented in the
    console.
    def print_board(self):
        for row in self.board_positions:
            print("|".join(["." if e == "" else e for e in row]))


    def is_game_over(self):
        """Returns False, none for still going and True, 1/0/-1 for over.
        1 is user wins, 0 is draw, -1 is user loses"""
        # internal function as it only used here
        # yields all of the sequences of 3 squares
        def gen_triplets():
            # yield rows
            yield from self.board_positions
            # yield columns
            for col in range(3):
                yield [self.board_positions[row][col] for row in range(3)]
            # yield diagonals
            yield [
                self.board_positions[0][0],
                self.board_positions[1][1],
                self.board_positions[2][2],
            ]
            yield [
                self.board_positions[2][0],
                self.board_positions[1][1],
                self.board_positions[0][2],
```

```

        ]
    # if there is a sequence that is 3 in a row return True and 1 or -1
    for win or loss (user perspective)
        if ["X"]*3 in gen_triplets():
            return True, 1
        if ["O"]*3 in gen_triplets():
            return True, -1
        # note so long as user takes all odd turns 9,7,...3,1 they will have
        last go so redundant in theory
        # but I left it in incase the code needs to be extended in future
        if self.moves_left == 0:
            return True, 0
        # if not already determined otherwise then the game isn't over
        return False, None

    # this function iterates through and yields the game states that could
    result form all possible moves on the current game state
    def gen_child_game_states(self):
        """precondition of game not being over"""
        # for each square
        for i, row in enumerate(self.board_positions):
            for j, square in enumerate(row):
                # if the square is empty
                if square == "":
                    # make a copy of the boards state where the next to move
                    moved there
                    new_board_positions = deepcopy(self.board_positions)
                    new_board_positions[i][j] = self.to_go_next

                    # yield this as a new game state where the user is next to
                    yield Game_State(
                        board_positions=new_board_positions,
                        moves_left=self.moves_left-1,
                        to_go_next="X" if self.to_go_next == "0" else "0"
                    )
                # else continue to next iteration

    # this is so called as it used the minimax algorithm along with alpha beta
    pruning to navigate the decision tree
    # the british museum algorithm is an approach where you explore the tree fully
    # until all terminal nodes are game states where the game is over
    # this function returns 2 values, the score and game state object for the best
    child game state (corresponds to best move)
    def british_museum_minimax(game_state: Game_State, maximizing_player:bool,
                                alpha, beta):
        # computer want 0 or -1 so minimizer
        # this is the base case to the recursive function (stop at terminal nodes)
        over, outcome = game_state.is_game_over()

```

```
if over:
    return outcome, game_state

# recursive case
best_child= None

# even though the initial call is for the servers move (minimizer),
recursive calls may be form maximizer
# I will explain the logic in more depth for the maximizing player
if maximizing_player:
    # meant to represent negative infinity (arbitrary bad so it is
replaced by the best evaluation)
    max_evaluation = -100
    # for each possible game state
    for child in game_state.gen_child_game_states():
        # recursive class to function to evaluate child node
        # now minimizing player
        # index for fist element as I only care about score not the actual
game state
        evaluation = british_museum_minimax(
            game_state=child,
            maximizing_player=not maximizing_player,
            alpha=alpha,
            beta=beta,
        )[0]
        # max_evaluation = max(max_evaluation, evaluation)
        # if this evaluation is the best so far
        if evaluation > max_evaluation:
            # update the max evaluation and best child (only for this
call)
            max_evaluation = evaluation
            best_child = child
            # update the alpha value to represent the best that the
maximizing player can get
            alpha = max(alpha, evaluation)

            # if the minimizing player can do better (for them less) than
alpha then they have a better option
            # this means that that this won't be the best child of the earlier
call
            # (the minimizer wouldn't need to give the maximizer such a good
score)
            if beta <= alpha:
                break
            # return the best child game state and its score
            return max_evaluation, best_child
        # similar for minimizer
    else:
```

```
min_evaluation = 100
for child in game_state.gen_child_game_states():
    evaluation = british_museum_minimax(
        game_state=child,
        maximizing_player=not maximizing_player,
        alpha=alpha,
        beta=beta,
    )[0]
    # min_evaluation = min(min_evaluation, evaluation)
    if evaluation < min_evaluation:
        min_evaluation = evaluation
        best_child = child
        beta = min(beta, evaluation)
    if beta <= alpha:
        break
return min_evaluation, best_child

# this main function is an easier wrapper for the app module to use
# it only determines and returns the best child (servers best move)
def main(game_state: Game_State):
    # output for debugging
    print("selecting best move from this game state:")
    game_state.print_board()

    # call the recursive function to determine the best child (and score for
    # debugging)
    # start with arbitrarily low alpha value and high beta value
    score, best_child = british_museum_minimax(
        game_state=game_state,
        maximizing_player=False,
        alpha=-100,
        beta=+100
    )
    # more output for debugging
    print("best move is:")
    best_child.print_board()
    print(f"with a guaranteed score of {score} or better (minimizer)")

    # return the best child
    return best_child
```

[index.html](#)

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<!-- standard metadata to help browsers -->
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Tic Tac Toe</title>

<!-- import javascript -->
<!-- <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"
integrity="sha512-
894YE6QWD5I59HgZOGReFYm4dnWc1Qt5NtvYSaNcOP+u1T9qYdvdihz0PPSiiqn/+3e7Jo4EaG7Tu
bfWGUrMQ==" crossorigin="anonymous" referrerpolicy="no-referrer"></script> -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js"
integrity="sha512-
9mpsATI0KC1wt+xVZfbcf2lJ8IFBAwsubJ6mI3rtULwyM3fBmQFzj0It4tGqxLOGQwGfJdk/G+fANn
xfq9/cew==" crossorigin="anonymous" referrerpolicy="no-referrer"></script>

<!-- load in the javascript from a url that the server will determine as
the html is served -->
<script src="{{ url_for('static', filename='main.js') }}"></script>

<!-- import css styles -->
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}>
</head>
<body>
<!-- i used the center tag to make the CSS simpler -->
<center>
    <h1>Tic Tac Toe</h1>
    <h2>You go first:</h2>
    <!-- this table will be the 3x3 board -->
    <table id="board">
        <tbody>
            <tr>
                <!-- each square has an ID so I can see which one is clicked -
            ->
                <td id="sq_11"></td>
                <td id="sq_12"></td>
                <td id="sq_13"></td>
            </tr>
            <tr>
                <td id="sq_21"></td>
                <td id="sq_22"></td>
                <td id="sq_23"></td>
            </tr>
            <tr>
                <td id="sq_31"></td>
```

```

        <td id="sq_32"></td>
        <td id="sq_33"></td>
    </tr>
</tbody>
</table>

<!-- this is the button for resetting the game --&gt;
&lt;button id="reset_btn"&gt;Reset Game&lt;/button&gt;
&lt;/center&gt;

&lt;/body&gt;
&lt;/html&gt;
</pre>

```

main.js

```

// create a socket object from the online socket library
let socket = io();

// global constants to show game state
let moves_left = 9;
let next_to_go= "X";
let board_positions= [
    ['', '', ''],
    ['', '', ''],
    ['', '', '']
];

// this function takes a row and column 1 to 3 and gives the html element
function get_square(row, col){
    // console.log('get_square called');
    // return document.querySelector(`#sq_${row.toString() + col.toString()}`);
    return document.getElementById(`sq_${row.toString() + col.toString()}`);
};

// this returns an array of all rows, columns and diagonals on the board for examination
function get_triplets(){
    triplets = []
    // push all rows
    for(i=0; i<=2; i++){
        triplets.push(board_positions[i])
    }
    // push all columns
    for(j=0; j<=2; j++){

```

```
column = []
for (i = 0; i <= 2; i++) {
    column.push(board_positions[i][j])
}
triplets.push(column);
}
// add diagonals
triplets.push([board_positions[0][0], board_positions[1][1],
board_positions[2][2]])
triplets.push([board_positions[0][2], board_positions[1][1],
board_positions[2][0]])
return triplets
}

// changes the next_to_go to the next player based on current player
function toggle_next_to_go(){
    // console.log('toggle_next_to_go called');
    if(next_to_go === "X"){
        next_to_go = "O"
    }
    else {
        next_to_go = "X"
    }
};

// this function updates the table tag in html so the contents reflects the
// board positions array.
function update_widget(){
    // console.log("update_widget called");
    // console.log({board_positions})
    var row; var col;
    for (let i = 0; i <= 2; i++) {
        for (let j = 0; j <= 2; j++) {
            row=i+1; col=j+1;
            square = get_square(row, col);
            // console.log("changing content of this tag")
            // console.log(`square ${row.toString()} ${col.toString()}`)
            // console.log({square})
            // console.log(`to ${board_positions[i][j]}`)
            // square.innerText = board_positions[i][j];
            square.textContent = board_positions[i][j];
            // square.textContent = "Test";
        };
    };
}

// checks if the target square is empty and returns a boolean
function is_legal_move(row, col){
```

```
// console.log("is_legal_move called");
// console.log({row})
// console.log({col})
i=row-1; j=col-1;
return board_positions[i][j] === ''
};

// this function is run once the users move has been validated and is
responsible for executing
// it orchestrates the various functions that must be called in the process of
making a move
function make_move(row, col){
    // console.log("make_move called");
    i=row-1; j=col-1;
    // alter board position array to make the move
    board_positions[i][j] = next_to_go;
    // change who is next to go
    toggle_next_to_go();
    // console.log({next_to_go})
    // there is now one less move left in the game
    moves_left += -1;
    // update the visual widget in html
    update_widget();

    // if the game is over handle else request the server's move
    if(is_game_over()){
        handle_game_over();
        // alert("game over")
    }
    else {
        request_server_move();
    };
};

// this is the method that is run whenever a square is clicked
function square_click(row, col){
    // console.log("square_click called");

    // validate that the square clicked is a legal move for the user to make
    // if the game isn't over
    if (!(is_game_over())){
        // and if the next to go is X, the user
        if(next_to_go === "X"){
            // and if clicked square is a legal move
            if(is_legal_move(row, col)){
                // run the make move function
                make_move(row, col);
            };
        };
    };
}
```

```
        }
        // else do nothing and ignore the click input as if the square tag
        were disabled.
        // else {
        //     console.log({next_to_go})
        //     console.log("user move ignored as server next to go")
        // }
    };
};

// this function emits a request to the server for its move
function request_server_move(){
    // console.log("request_server_move called");
    // it uses socket emit
    // is provides the server with the relevant game data
    socket.emit(
        'server_move_request',
        {
            board_positions,
            next_to_go,
            moves_left
        }
    );
};

// this function is responsible for determining if the current game is over.
// this is done by examining moves left to detect draws and triplet sequences
// to detect wins
// this function only returns a boolean value and not how the game is over
function is_game_over(){
    // console.log("is_game_over called");
    // if no moves left then the game must be over
    if(moves_left === 0){
        return true;
    };

    // get the relevant triplet sequences
    triplets = get_triplets();
    console.log({triplets})

    // iterate through the 8 triplets
    // for(i=0; i<=8; i++){
    for(i=0; i<=7; i++){
        triplet = triplets[i];
        // using 2 equals comparison on purpose
        // i am using JSON.stringify to compare arrays to see if they are 3 in
        a row
    };
}
```

```
        if (JSON.stringify(triplet) == JSON.stringify(["X", "X", "X"]) ||
JSON.stringify(triplet) == JSON.stringify(["0", "0", "0"])){
            return true;
        };
    };
    // if not already determined that the game is over then it must not be
over
    return false;
};

// this function is run once the game is over
// it determines how the game is over and takes displays the appropriate
message
function handle_game_over(){
    // precondition game is over so function is_game_over returned true
    // determine who won or if it was a draw
    // 1 is user (X) won, 0 is draw, -1 is user (X) lost
    // console.log("handle_game_over called");

    // variable initialized with 0 to represent draw
    winner = 0;

    // get triplets and iterate through them
    triplets = get_triplets();
    // for (i = 0; i <= 8; i++) {
    for (i = 0; i <= 7; i++) {
        triplet = triplets[i];
        // for each triplet make a comparison to check if the triplet means
that the user has won or lost
        // using 2 equals comparrison on purpose
        if (JSON.stringify(triplet) == JSON.stringify(["X", "X", "X"])) {
            winner = 1;
            // no real need to break out of the loop as their can only be one
or zero winning or loosing sequence
            // but it makes the code more efficient and readable
            break;
        }
        else if (JSON.stringify(triplet) == JSON.stringify(["0", "0", "0"])){
            winner = -1
            break;
        };
    };
    // if there were no winning sequences then the outcome will remain the
default: 0 for draw

    // display the appropriate message to the user in each case
    if(winner === 1){
        alert("congratulations you won");
    }
}
```

```
}

else if(winner === 0){
    alert("the game was a draw");
}
else{
    alert("unfortunately you lost");
};

// no need to disable the game as the is_game_over check will fail
whenever the user clicks a square
};

// this function is called to handle a response from the server and execute
the server's move
// the global board positions will have already been updated with the ones
returned by the server
function handle_server_move(){
    // update the html to show the user the updated game state
    update_widget();
    // decrement the moves left
    moves_left += -1;
    // toggle who is next to go to the user
    toggle_next_to_go();
    // check if the servers move is a winning one and if so handle
    appropriately
    if (is_game_over()) {
        handle_game_over();
        // alert("game over")
    };
    // no else needed to enable the user to have a turn as game over
    validation included before user's turn
};

// tie the handler method to the socket event.
socket.on('server_move_response', function(msg){
    // console.log({msg})
    // update global for board positions with the new one that the server has
    sent
    board_positions = msg.board_positions
    // call handler function
    handle_server_move();
    // i believe all the handler function have to return false
    return false;
})

// this function is run when the user clicks the reset the game button

function reset_game(){
    // reset globals
}
```

```
// if the user has just in the last second moved and then clicked reset  
before the computers move  
// a logic error could occur where the game resets and then the server  
responds with its move  
// this is prevented here by ensuring that a server move isn't pending  
  
// // LOGIC ERROR HERE, FAILS TO RESET WHEN DRAW  
// // if(next_to_go === "0"){  
// //     return false  
// // }  
  
// corrected code after reset button test  
if(next_to_go === "0" && !(is_game_over())){  
    return false  
}  
  
moves_left = 9;  
next_to_go = "X";  
board_positions = [  
    ['', '', ''],  
    ['', '', ''],  
    ['', '', '']  
];  
// update the html to reflect the board position matrix  
update_widget();  
}  
  
// code to be executed when the page loads  
window.addEventListener('load', function(){  
    // for testing (if loading with a non blank game state)  
    // update_widget();  
  
    // the following blocks of code bind the appropriate handler method to  
each square  
    // I tried to use iteration but it results in a logic error that I  
struggled to debug  
    // I therefore adopted the the less elegant manual approach  
    // the looping code is still here commented out as I have tried to make it  
work  
  
    // for(i=1; i<=3; i++){  
    //     for(j=1; j<=3; j++){  
    //         get_square(i, j).addEventListener('click', function () {  
square_click(i, j);});  
    //     }  
    // }  
    get_square(1,1).addEventListener('click', function(){square_click(1, 1)});  
    get_square(1,2).addEventListener('click', function(){square_click(1, 2)});
```

```
get_square(1,3).addEventListener('click', function(){square_click(1, 3)});  
get_square(2,1).addEventListener('click', function(){square_click(2, 1)});  
get_square(2,2).addEventListener('click', function(){square_click(2, 2)});  
get_square(2,3).addEventListener('click', function(){square_click(2, 3)});  
get_square(3,1).addEventListener('click', function(){square_click(3, 1)});  
get_square(3,2).addEventListener('click', function(){square_click(3, 2)});  
get_square(3,3).addEventListener('click', function(){square_click(3, 3)});  
  
// tie the reset the game handler function to the reset button on click  
document.getElementById('reset_btn').addEventListener('click', reset_game)  
});
```

Style.css

```
body {  
    background-color: lightgray;  
}  
  
h1, h2 {  
    text-align: center;  
}  
  
table{  
    /* width: 50vw;  
    height: 50vw; */  
    /* border: 3px solid black; */  
    /* table-layout: fixed; */  
    padding: 10px;  
    margin: 15px;  
    /* margin: auto; */  
}  
  
td{  
    border: 5px solid black;  
    /* overflow: hidden; */  
  
    /* laptop */  
    width: 21vh;  
    height: 21vh;  
  
    /* ipad air */  
    /* width: 24vw;  
    height: 24vw; */  
    font-size: 80px;  
    text-align: center  
}
```

```
button{
    margin-left: auto;
    margin-right: auto;
    width: 150;
    height: 50px;
    text-align: center;
    /* background-color: blue; */
}
```

Prototype 2: Vue page

ChessGame.vue

```
<template>
    <h1>Chess Game V2</h1>
    <div :style="cssVars" class="chess_board">
        <table>
            <tr v-for="[row_num, row] in board.entries()" :key="row">
                <td v-for="[col_num, square] in row.entries()" :key="square"
@click="square_click(row_num, col_num)">
                    <!-- {{row_num}}, {{col_num}} -->
                    <!-- {{square !== null? square: ""}} -->

                    <!-- playing around with white pieces -->
                    <!-- https://stackoverflow.com/questions/4772906/css-is-it-possible-to-add-a-black-outline-around-each-character-in-text -->
                    <span style="color:white; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-if="square == '♟'">♟</span>
                    <!-- <span style="color:black" v-if="square == '♙'">♙</span> -->
                    <!-- <span style="color:white" v-if="square == '♙'">♙</span> -->

                    <!-- <span style="color:black; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-else-if="square == '♟'">♟</span> -->
                    <span style="color:black" v-else-if="square == '♟'">♟</span>
                    <!-- <span v-else></span> -->
                </td>
            </tr>
        </table>
    </div>
```

```
<h2>Turn {{turn_num}}: {{next_to_go == 'user'? "your turn": "computer's turn"}}</h2>

<div class="option_button">
    <button>Concede</button>
    <button>Reset</button>
</div>

<div class="pieces_left_table">
    <table>
        <tr>
            <th>White Pieces Left: {{pieces_left.black}}/6</th>
            <th>Black Pieces Left: {{pieces_left.white}}/6</th>
        </tr>
        <tr>
            <td>{{ '♟'.repeat(pieces_left.white) }}</td>
            <td>{{ '♞'.repeat(pieces_left.black) }}</td>
        </tr>
    </table>
</div>

<h2>Previous Moves:</h2>

<div class="previous_moves_table">
    <table>
        <tr>
            <th>White previous moves:</th>
            <th>Black previous moves:</th>
        </tr>
        <tr>
            <td>
                <div v-for="move in previous_moves" :key="move.num">
                    <span v-if="move.player=='white'">
                        Move {{move.num}}: ♜ {{ move.from }} to {{ move.to }}
                    </span>
                </div>
            </td>
            <td>
                <div v-for="move in previous_moves" :key="move.num">
                    <span v-if="move.player=='black'">
                        Move {{move.num}}: ♞ {{ move.from }} to {{ move.to }}
                    </span>
                </div>
            </td>
        </tr>
    </table>
</div>
```

```
</tr>
</table>
</div>

</template>

<script>
    import {handle_square_click} from '@/assets/scripts/external.js'
    // import * from '@/assets/scripts/external.js'
    const white_sq_color = '#f5e6bf';
    const black_sq_color = '#66443a';
    // white pawn: '♟'; black pawn: '♙'
    const pawn_white = '♙';
    const pawn_black = '♟';

    export default {
        name: "ChessGame",
        ready_for_next_move: false,
        // pawn characters
        // https://en.wikipedia.org/wiki/Chess_symbols_in_Unicode
        data(){return{
            board: [
                Array(8).fill(null),
                Array(8).fill(pawn_black),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(pawn_white),
                Array(8).fill(null)
            ],
            next_to_go: 'user',
            turn_num: 1,
            pieces_left: {
                black: 2,
                white: 3
            },
            previous_moves: [
                { num: 1, player: "white", from: "B1", to: "B3" },
                { num: 2, player: "black", from: "A5", to: "A4" },
                { num: 3, player: "white", from: "B3", to: "A4" },
                { num: 4, player: "black", from: "E5", to: "E3" },
            ],
        }},
        methods: {
            square_click(row, col) {
                // console.log(`square click: (${row},${col})`)
                handle_square_click(row, col)
            }
        }
    }
}
```

```
        }
    },
    // https://www.telerik.com/blogs/passing-variables-to-css-on-a-vue-
component
computed: {
    cssVars(){return{
        '--black_sq_color': black_sq_color,
        '--white_sq_color': white_sq_color,
    };}
}
</script>

<style scoped>
/* :root {
    https://www.vectorstock.com/royalty-free-vector/chess-field-in-beige-
and-brown-colors-vector-24923385
    https://imagecolorpicker.com/en
    --black_sq_color: #66443a;
    --white_sq_color: #f5e6bf;
} */


h1 {
    text-align: center;
    text-decoration: underline;
}
h2 {
    text-align: center;
    font-size: 32px;
}
table {
    margin: auto;
    table-layout: fixed;
    text-align: center;
}

.option_button {
    margin: auto;
    text-align: center;
    /* margin-left: 0.125vw;
    margin-right: 0.125vw; */
    margin-left: 20px;
    margin-right: 20px;
}

.chess_board td {
    height: 10.5vh;
    width: 10.5vh;
```

```
/* dimension so it is all in view for ipad air portrait  (820 * 1180)
*/
/* height: 12vw;
width: 12vw; */
text-align: center;
/* border: 3px black solid; */
font-size: 50px;
/* font-weight: 100; */

}
.chess_board table {
    border: 1px black solid;
}

/* these odd and even rules dictate the color of chess board squares */
.chess_board tr:nth-child(2n-1) > td:nth-child(2n-1) {
background-color: var(--white_sq_color);
    /* color: black; */
}
.chess_board tr:nth-child(2n-1) > td:nth-child(2n) {
    background-color: var(--black_sq_color);
    /* color: white; */
}
.chess_board tr:nth-child(2n) > td:nth-child(2n-1) {
    background-color: var(--black_sq_color);
    /* color: white; */
}
.chess_board tr:nth-child(2n) > td:nth-child(2n) {
    background-color: var(--white_sq_color);
    /* color: black; */
}

.previous_moves_table th, td {
    width: 35vw;
    /* border: 3px solid black; */
    font-size: 28px;
}
.pieces_left_table th, td {
    width: 35vw;
    /* border: 3px solid black; */
    font-size: 28px;
}
</style>
```

[external.js](#)

```
function handle_square_click(row, col){  
    console.log(`square click: (${row},${col})`)  
}  
  
export {  
    handle_square_click,  
}
```

Prototype 2: Final Code

Directory Structure:

```
PS C:\Users\henry\Documents\computing coursework\code list\prototype 2 final>  
tree /f  
Folder PATH listing for volume OS  
Volume serial number is 4ED8-B070  
C:  
|   assorted.py  
|   board_state.py  
|   console_chess.py  
|   game.py  
|   minimax.py  
|   pieces.py  
|   test_board_state.py  
|   test_minimax.py  
|   test_pieces.py  
|   test_vector.py  
|   vector.py  
  
+-test_data  
|   +-board_state  
|       color_in_check.yaml  
|       game_over.yaml  
|       generate_all_pieces.yaml  
|       generate_legal_moves.yaml  
|       generate_pieces_of_color.yaml  
|       piece_at_vector.yaml  
  
|   +-pieces  
|       test_board_populated.yaml  
|       test_empty_board.yaml  
  
|   +-vector  
|       from_square.yaml  
|       vector_add.yaml  
|       vector_in_board.yaml  
|       vector_multiply.yaml  
|       vector_to_square.yaml
```

```
|__test_reports
    test_depth_1_advanced_vs_randotron.csv
    test_depth_1_vanilla_vs_depth_1_variable_check.csv
    test_depth_1_vanilla_vs_randotron.csv
    test_depth_2_advanced_vs_randotron.csv
    test_depth_2_vs_depth_1.csv
    test_depth_3_vs_depth_1.csv
    test_depth_3_vs_depth_2.csv
```

assorted.py

```
# this file is just a file of short assorted constants and functions that are
general in use

# It only contains small functions as I have tried to group large, similar
functions logically in there own file

# this is used in the static evaluation and minimax process. It is used to
represent infinity in a way that still allows comparrison
ARBITRARILY_LARGE_VALUE = 1_000_000

# this function is relatively redundant but allows for print statements in
debugging
# in later iteration this may be replaced with logging.
# it is useful as it allows for DEBUG print statements without needing to
remove them when finished
DEBUG = True
def dev_print(*args, **kwargs):
    if DEBUG:
        print(*args, **kwargs)

# this is an exception that allows for the game data to be bound to it
# this allows for the relevant chess game that caused the error to be examined
afterwards
# it is a normal exception except the constructor has been modified to save
the game data as a property
class __ChessExceptionTemplate__(Exception):
    def __init__(self, *args, **kwargs) -> None:
        # none if key not present
        self.game = kwargs.pop("game", None)

        super().__init__(*args, **kwargs)

# these are custom exceptions.
# they contain no logic but have distinct types allowing for targeted error
handling
class InvalidMove(__ChessExceptionTemplate__):
```

```
    pass
class NotUserTurn(__ChessExceptionTemplate__):
    pass
```

vector.py

```
# import dataclass to reduce boilerplate code
from dataclasses import dataclass


# frozen = true means that the objects will be immutable
@dataclass(frozen=True)
class Vector():
    # 2d vector has properties i and j
    i: int
    j: int

    # code to allow for + - and * operators to be used with vectors
    def __add__(self, other):
        assert isinstance(other, Vector), "both objects must be instances of
the Vector class"
        return Vector(
            i=self.i + other.i,
            j=self.j + other.j
        )
    def __sub__(self, other):
        assert isinstance(other, Vector), "both objects must be instances of
the Vector class"
        return Vector(
            i=self.i - other.i,
            j=self.j - other.j
        )

    def __mul__(self, multiplier: int):
        return Vector(
            i=self.i * multiplier,
            j=self.j * multiplier
        )

    # check if a vector is in board
    def in_board(self):
        """Assumes that the current represented vector is a position vector
        checks if it points to a square that isn't in the chess board"""
        return self.i in range(8) and self.j in range(8)

    # alternative way to create instance, construct from chess square
    @classmethod
    def construct_from_square(cls, to_sqr):
```

```

"""Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1,
3)"""
to_sqr = to_sqr.upper()
letter, number = to_sqr

# map letters and numbers to 0 to 7 and create new vector object
return cls(
    i=ord(letter.upper()) - ord("A"),
    j=int(number)-1
)

# this function is the reverse and converts a position vector to a square
def to_square(self) -> str:
    letter = chr(self.i + ord("A"))
    number = self.j+1
    return f"{letter}{number}"

# this function checks if 2 vectors are equal
def __eq__(self, other) -> bool:
    try:
        # assert same subclass like rook
        assert isinstance(other, type(self))
        assert self.i == other.i
        assert self.j == other.j

    except AssertionError:
        return False
    else:
        return True

# used to put my objects in set
def __hash__(self):
    return hash((self.i, self.j))

```

[pieces.py](#)

```

# to do: make value and value matrix unchangeable of private
# https://stackoverflow.com/questions/31457855/cant-instantiate-abstract-
# class-with-abstract-methods

# import libraries and other local modules
import abc
from itertools import chain as iter_chain, product as iter_product
from typing import Callable

from vector import Vector
from assorted import ARBITRARILY_LARGE_VALUE

```

```
# Here is an abstract base class for piece,  
# it dictates that all child object have the specified abstract attributes  
else and error will occur  
# this ensures that all piece objects have the same interface  
  
class Piece(abc.ABC):  
  
    # required  
    # color is public  
    color: str | None  
    # value and value matrix is protected  
    # value is inherent value  
    _value: int  
    # value matrix is additional value based on location  
    _value_matrix: tuple[tuple[float]]  
  
    # must be given before init  
    @abc.abstractproperty  
    def _value_matrix(): pass  
  
    @abc.abstractproperty  
    def _value(): pass  
  
    @abc.abstractproperty  
    def color(): pass  
  
    @abc.abstractmethod  
    def symbol(self) -> str:  
        """uses color to determine the appropriate symbol"""  
  
    # not needed as abstract method as some classes will now override  
    def __init__(self, color):  
        self.color = color  
        self.last_move = None  
  
    # this should use the position vector and value matrix to get the value of  
    # the piece  
    def get_value(self, position_vector: Vector):  
        # flip if black as matrices are all for white pieces  
        if self.color == "W":  
            row, column = 7-position_vector.j, position_vector.i  
        else:  
            row, column = position_vector.j, position_vector.i  
  
        # return sum of inherent value + value relative to position on board  
        return self._value + self._value_matrix[row][column]
```

```
# this function should yield all the movement vector tha the piece can
move by
    # this doesn't account for check and is based on rules specific to each
piece as well an checking if a vector is outside the board

@abc.abstractmethod
def generate_movement_vectors(self, pieces_matrix, position_vector):
    pass

# when str(piece) called give the symbol

def __str__(self):
    # return f"{self.color}{self.symbol}"
    return self.symbol()

# standard repr method
def __repr__(self):
    return f"{type(self).__name__}(color='{self.color}')"

# logic that would be otherwise repeated in many of the child classes
# determines the contents of a given square
def square_contains(self, square):
    """returns 'enemy' 'ally' or None for empty"""
    # check if empty
    if square is None:
        return "empty"
    # else the square must contain a piece, so examine its color
    if square.color == self.color:
        return "ally"
    else:
        return "enemy"

# again reduces repeated logic
# checks the result of a position vector
# if not illegal (out of board) the square contents is returned
def examine_position_vector(self, position_vector: Vector, pieces_matrix):
    """returns 'enemy' 'ally' 'empty' or 'illegal' """
    # check if the vector is out of the board
    if not position_vector.in_board():
        return 'illegal'
    # for the rest of the code I can assume the vector is in board

    # else get the square at that vector
    row, column = 7-position_vector.j, position_vector.i
    square = pieces_matrix[row][column]

    # examine its contents
    return self.square_contains(square)
```

```
# equality operator
def __eq__(self, other):
    try:
        # assert same subclass like rook
        assert isinstance(other, type(self))
        assert self.color == other.color

        # i am not checking that pieces had the same last move as I want
        # to compare kings without for the check function
        # assert self.last_move == other.last_moves

        # value and value_matrix should never be changes
    except AssertionError:
        return False
    else:
        return True

# making pieces hashable allows for pieces matrices to be hashed and
# allows for pieces and pieces matrices to be put in sets,
# also essential for piping data between python interpreter instances
# (different threads) for multitasking
def __hash__(self):
    return hash((self.symbol(), self.color, self.last_move))

# this class inherits from Piece an so it inherits some logic and some
# requirements as to how its interface should be
# as many child classes are similar I will explain this one in depth and then
# only explain notable features of others
class Pawn(Piece):
    # defining abstract properties, needed before init
    _value = 100
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [50, 50, 50, 50, 50, 50, 50, 50],
        [10, 10, 20, 30, 30, 20, 10, 10],
        [5, 5, 10, 25, 25, 10, 5, 5],
        [0, 0, 0, 20, 20, 0, 0, 0],
        [5, -5, -10, 0, 0, -10, -5, 5],
        [5, 10, 10, -20, -20, 10, 10, 5],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]
    color = None

    # define symbol method (str method)
    def symbol(self): return f"{self.color}P"

    # override init constructor
```

```
def __init__(self, color):
    # perform super's instructor
    super().__init__(color)

    # but in addition...
    # decide the vectors that the piece can move now as it is based on
color
    multiplier = 1 if color == "W" else -1

    # method defined here as it only used here
    # decided if the pawn is allowed to move foreward 2 based on square
contents and last move
    def can_move_foreword_2(square):
        return square is None and self.last_move is None

    # tuple contains pairs of vector and contition that must be met
    # (in the form of a function that takes square and returns a boolean)
    self.movement_vector_and_condition: tuple[Vector, Callable] = (
        # v(0, 1) for foreword
        (Vector(0, multiplier), lambda square:
self.square_contains(square) == "empty"),
        # v(0, 2) for foreword as first move
        (Vector(0, 2*multiplier), can_move_foreword_2),
        # v(-1, 1) and v(1, 1) for take
        (Vector(1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
        (Vector(-1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
    )

    # generate movement vectors
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # iterate through movement vectors and conditions
    for movement_vector, condition in self.movement_vector_and_condition:
        # get resultant vector
        resultant_vector = position_vector + movement_vector
        # if vector_out of range continue
        if not resultant_vector.in_board():
            continue

        # get the contents of the square corresponding to the resultant
        row, column = 7-resultant_vector.j, resultant_vector.i
        piece = pieces_matrix[row][column]

        # if the condition is met, yield the vector
        if condition(piece):
            yield movement_vector
```

```

class Knight(Piece):
    _value = 320
    _value_matrix: tuple[tuple[float]] = [
        [-50, -40, -30, -30, -30, -30, -40, -50],
        [-40, -20, 0, 0, 0, 0, -20, -40],
        [-30, 0, 10, 15, 15, 10, 0, -30],
        [-30, 5, 15, 20, 20, 15, 5, -30],
        [-30, 0, 15, 20, 20, 15, 0, -30],
        [-30, 5, 10, 15, 15, 10, 5, -30],
        [-40, -20, 0, 5, 5, 0, -20, -40],
        [-50, -40, -30, -30, -30, -30, -40, -50]
    ]
    color = None

    # n for knight as king takes k
    def symbol(self): return f"{self.color}N"

    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # this function yields all 8 possible vectors
        def possible_movement_vectors():
            vectors = (Vector(2, 1), Vector(1, 2))
            # for each x multiplier, y multiplier and vector combination
            for i_multiplier, j_multiplier, vector in iter_product((-1, 1), (-1, 1), vectors):
                # yield corresponding vector
                yield Vector(
                    vector.i * i_multiplier,
                    vector.j * j_multiplier
                )
        # iterate through movement vectors
        for movement_vector in possible_movement_vectors():
            # get resultant
            resultant_vector = position_vector + movement_vector
            # look at contents of square
            contents =
            self.examine_position_vector(position_vector=resultant_vector,
            pieces_matrix=pieces_matrix)
            # if square is empty yield vector
            if contents == "empty":
                yield movement_vector


class Bishop(Piece):
    _value = 330
    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -10, -10, -10, -10, -20],
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 10, 10, 5, 0, -10],
        [-10, 5, 5, 10, 10, 5, 5, -10],

```

```
[-10, 0, 10, 10, 10, 10, 0, -10],
[-10, 10, 10, 10, 10, 10, 10, -10],
[-10, 5, 0, 0, 0, 0, 5, -10],
[-20, -10, -10, -10, -10, -10, -10, -20]
]
color = None

def symbol(self): return f"{self.color}B"

def generate_movement_vectors(self, pieces_matrix, position_vector):
    # sourcery skip: use-itertools-product

    # repeat for all 4 vector directions
    for i, j in iter_product((1, -1), (1, -1)):
        unit_vector = Vector(i, j)
        # iterate through length multipliers
        for multiplier in range(1, 8):
            # get movement and resultant vectors
            movement_vector = unit_vector * multiplier
            resultant_vector = position_vector + movement_vector

            # examine the contents of the square and use switch case to
            decide behaviour
            match
            self.examine_position_vector(position_vector=resultant_vector,
            pieces_matrix=pieces_matrix):
                case 'illegal':
                    # if vector extends out of the board stop extending
                    break
                case 'ally':
                    # break of of for loop (not just match case)
                    # as cannot hop over piece so don't explore longer
                    vectors in same direction
                    break
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                    # break of of for loop (not just match case)
                    # as cannot hop over piece so don't explore longer
                    vectors in same direction
                    break
                case 'empty':
                    # is valid
                    yield movement_vector
                    # and keep exploring, don't break

class Rook(Piece):
    _value = 500
```

```

_value_matrix: tuple[tuple[float]] = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [5, 10, 10, 10, 10, 10, 10, 5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [0, 0, 0, 5, 5, 0, 0, 0]
]
color = None

# r for rook
def symbol(self): return f"{self.color}R"

# this code is very similar in structure to that of a bishop just with
different direction vectors
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # sourcery skip: use-itertools-product
    unit_vectors = (
        Vector(0, 1),
        Vector(0, -1),
        Vector(1, 0),
        Vector(-1, 0),
    )
    # for unit_vector, multiplier in iter_product(unit_vectors, range(1,
8)):
        for unit_vector in unit_vectors:
            for multiplier in range(1, 8):
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector

                # note cases that contain only break are not redundant, they
break the outer for loop
                match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                    case 'illegal':
                        # if vector extends out of the board stop extending
                        break
                    case 'ally':
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'enemy':
                        # this is a valid move

```

```

        yield movement_vector
        # break of of for loop (not just match case)
        # as cannot hop over piece so don't explore longer
vectors in same direction
        break
    case 'empty':
        # is valid
        yield movement_vector
        # and keep exploring, don't break

class Queen(Piece):
    _value = 900
    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -5, -5, -10, -10, -20],
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 5, 5, 5, 0, -10],
        [-5, 0, 5, 5, 5, 5, 0, -5],
        [0, 0, 5, 5, 5, 5, 0, -5],
        [-10, 5, 5, 5, 5, 5, 0, -10],
        [-10, 0, 5, 0, 0, 0, 0, -10],
        [-20, -10, -10, -5, -5, -10, -10, -20]
    ]
    color = None

    def symbol(self): return f"{self.color}Q"

    # this code also uses a similar structure to the rook or bishop
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1, 0, 1)) if i != 0 and j != 0)
        unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1, 0, 1)) if i != 0 or j != 0)

        # for unit_vector, multiplier in iter_product(unit_vectors, range(1, 8)):
        for unit_vector in unit_vectors:
            for multiplier in range(1, 8):
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector
                match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                    case 'illegal':
                        # if vector extends out of the board stop extending
                        break
                    case 'ally':
                        # break of of for loop (not just match case)

```

```

        # as cannot hop over piece so don't explore longer
vectors in same direction
        break
    case 'enemy':
        # this is a valid move
        yield movement_vector
        # break of of for loop (not just match case)
        # as cannot hop over piece so don't explore longer
vectors in same direction
        break
    case 'empty':
        # is valid
        yield movement_vector
        # and keep exploring, don't break

class King(Piece):
    # not needed as static eval doesn't add up kings value
    _value = ARBITRARILY_LARGE_VALUE

    # there are 2 matrices to represent the early and late game for the king
    value_matrix_early: tuple[tuple[float]] = [
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-20, -30, -30, -40, -40, -30, -30, -20],
        [-10, -20, -20, -20, -20, -20, -20, -10],
        [20, 20, 0, 0, 0, 20, 20],
        [20, 30, 10, 0, 0, 10, 30, 20]
    ]
    value_matrix_late = [
        [-50, -40, -30, -20, -20, -30, -40, -50],
        [-30, -20, -10, 0, 0, -10, -20, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -30, 0, 0, 0, 0, -30, -30],
        [-50, -30, -30, -30, -30, -30, -30, -50]
    ]
    color = None

    # initially value matrix is the early one
    _value_matrix: tuple[tuple[float]] = value_matrix_early

    # def __init__(self, *args, **kwargs):
    #     self._value_matrix = self.value_matrix_early
    #     super().__init__(self, *args, **kwargs)

```

```
def symbol(self): return f"{self.color}K"

# based on total pieces, changes the value matrix
def update_value_matrix(self, pieces_matrix):
    # counts each empty square as 0 and each full one as 1 then sums them
    up to get total pieces
    # if total pieces less than or equal to 10: then late game
    if sum(int(isinstance(square, Piece)) for square in
iter_chain(pieces_matrix)) <= 10:
        self.value_matrix = self.value_matrix_late
    # else early game
    else:
        self.value_matrix = self.value_matrix_early

# this generates the movement vectors for the king
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # take the opportunity to update the value matrix
    self.update_value_matrix(pieces_matrix)

    # all 8 movement vectors
    unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1,
0, 1)) if i != 0 or j != 0)

    # for each movement vector, get the resultant vector
    for movement_vector in unit_vectors:
        resultant_vector = position_vector + movement_vector

        # examine contents of square and use switch case to decide
        behaviour
        match
            self.examine_position_vector(position_vector=resultant_vector,
            pieces_matrix=pieces_matrix):
                case 'illegal':
                    continue
                case 'ally':
                    continue
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                case 'empty':
                    # is valid
                    yield movement_vector

# used by other modules to convert symbol to piece
PIECE_TYPES = {
    'P': Pawn,
    'N': Knight,
```

```

'B': Bishop,
'R': Rook,
'K': King,
'Q': Queen
}

# if __name__ == '__main__':
# ensures that all classes are valid (not missing any abstract properties)
# whenever the module is imported
Pawn('W')
Knight('W')
Bishop('W')
Rook('W')
Queen('W')
King('W')

```

board_state.py

```

from copy import deepcopy
from dataclasses import dataclass
from itertools import product as iter_product

import pieces as pieces_mod
from assorted import ARBITRARILY_LARGE_VALUE
from vector import Vector

# this is a pieces matrix for the starting position in chess
# white is at the bottom as it is from the user's perspective and I am
# currently assuming the user is white.
# I can change this in the frontend later
STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
    (
        pieces_mod.Rook(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Queen(color="B"),
        pieces_mod.King(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Rook(color="B")
    ),
    (pieces_mod.Pawn(color="B"),)*8,
    (None,)*8,
    (None,)*8,
    (None,)*8,
    (pieces_mod.Pawn(color="W"),)*8,
    (
        pieces_mod.Rook(color="W"),

```

```

        pieces_mod.Knight(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Queen(color="W"),
        pieces_mod.King(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Rook(color="W")
    )
)

# frozen makes each instance of the board_state class is immutable
@dataclass(frozen=True)
class Board_State():
    next_to_go: str = "W"
    # the pieces matrix keeps track of the board positions and the pieces
    pieces_matrix: tuple[tuple[pieces_mod.Piece]] = STARTING_POSITIONS

    # this function outputs the board in a user friendly way
    # 8| BR  BN  BB  BQ  BK  BB  BN  BR
    # 7|   .   BP  BP  BP  BP  BP  BP
    # 6|   .   .   .   .   .   .   .
    # 5| BP  BP  .   .   .   .   .
    # 4|   .   .   .   WP  .   WP  .
    # 3|   .   .   .   .   .   .   .
    # 2| WP  WP  WP  .   WP  .   WP  WP
    # 1| WR  WN  WB  WQ  WK  WB  WN  WR
    #   (A  B  C  D  E  F  G  H )

    def print_board(self):
        # convert each piece to a symbol BP and replace none with dots
        # add numbers to left and add letters at the bottom
        numbers = range(8, 0, -1)
        letters = [chr(i) for i in range(ord("A"), ord("H")+1)]
        for row, number in zip(self.pieces_matrix, numbers):
            # clean up row of pieces
            symbols_row = map(lambda piece: piece.symbol() if piece else ".",
row)
            pretty_row = f"{number}| {' '.join(symbols_row)}"
            print(pretty_row)
        print(f" ({' '.join(letters)})")

    def get_piece_at_vector(self, vector: Vector):
        # this function exists as it is a really common operation.
        # due to vector 0, 0 pointing the the bottom left not top left of the
        # 2d array,
        # some correction is needed
        column, row = vector.i, 7-vector.j
        return self.pieces_matrix[row][column]

```

```
# this function should yield all the pieces on the board
def generate_all_pieces(self):
    # nested loop for i and j to iterate through all possible vectors
    for i, j in iter_product(range(8), range(8)):
        position_vector = Vector(i,j)
        # get the contents of the corresponding square
        piece = self.get_piece_at_vector(position_vector)

        # skip if none: skip if square empty
        if piece:
            yield piece, position_vector

# this yields all pieces of a given color:
# used when examining legal moves of a given player
def generate_pieces_of_color(self, color=None):
    # be default give pieces of player next to go
    if color is None:
        color = self.next_to_go

    # return all piece and position vector pairs
    # filtered by piece must match in color
    yield from filter(
        # lambda piece, _: piece.color == color,
        lambda piece_and_vector: piece_and_vector[0].color == color,
        self.generate_all_pieces()
    )

# determines if a given player is in check based on the player's color
def color_in_check(self, color=None):
    # default is to check if the player next to go is in check
    if color is None:
        color = self.next_to_go

    # let is now be A's turn
    # I use this player a and b model to keep track of the logic here
    color_a = color
    color_b = "W" if color == "B" else "B"

    # we will examine all the movement vectors of B's pieces
    # if any of them could take the A's King then currently A is in check
    as their king is threatened by 1 or more pieces (which could take it next
    turn)
    # for each of b's pieces
    for piece, position_v in self.generate_pieces_of_color(color=color_b):
        movement_vs = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=position_v
        )
```

```

# for each movement vector that the piece could make
for movement_v in movement_vs:
    resultant = position_v + movement_v
    # check the contents of the square
    to_square = self.get_piece_at_vector(resultant)
    # As_move_threatens_king_A = isinstance(to_square,
pieces_mod.King) and to_square.color == color_a
        # if the contents is A's king then the a is in check.
        As_move_threatens_king_A = (to_square ==
pieces_mod.King(color=color_a))
            # if As_move_threatens_king_A then return true
            if As_move_threatens_king_A:
                return True
    # if none of B's pieces were threatening to take A's king then A isn't
in check
    return False

# this function is a generator to be iterated through.
# it is responsible for yielding every possible move that a given player
can make
# yields this as a position and a movement vector
def generate_legal_moves(self):
    # iterate through all pieces belonging to player next to go
    for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
        movement_vectors = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=piece_position_vector
        )
        # iterate through movement vectors of this piece
        for movement_vector in movement_vectors:
            # examine resulting child game state
            child_game_state =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
                # determine if current player next to go (different to next to
go of child game state) is in check
                is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
                    # only yield the move if it doesn't result in check
                    if not is_check_after_move:
                        yield piece_position_vector, movement_vector

    # determines if the game is over
    # returns over, winner
    def is_game_over_for_next_to_go(self):
        # sourcery skip: remove-unnecessary-else, swap-if-else-branches

```

```

# in all cases, the game is over if a player has no legal moves left
if not list(self.generate_legal_moves()):
    # if b in check
    if self.color_in_check():
        # checkmate for b, a wins
        winner = "W" if self.next_to_go == "B" else "B"
        return True, winner
    else:
        # stalemate
        return True, None
# game not over
return False, None

# this function is responsible for generating a static evaluation for a
given board-state
# it should be used by a maximiser or minimiser
# starting position should have an evaluation of 0
def static_evaluation(self):
    # if over give an appropriate score for win loss or draw
    over, winner = self.is_game_over_for_next_to_go()
    if over:
        match winner:
            case None: multiplier = 0
            case "W": multiplier = 1
            case "B": multiplier = -1
    # return winner * ARBITRARILY_LARGE_VALUE
    return multiplier * ARBITRARILY_LARGE_VALUE

# this function takes a piece as an argument and uses its color to
decide if its value should be positive or negative
def get_piece_value(piece: pieces_mod.Piece, position_vector: Vector):
    # this function assumes white is maximizer and so white pieces
have a positive score and black negative
    match piece.color:
        case "W": multiplier = 1
        case "B": multiplier = (-1)
    value = multiplier * piece.get_value(position_vector)
    # print(f"{piece.symbol()} at {position_vector.to_square} has
value {value}")
    return value

# for each piece, get the value (+/-)
values = map(
    lambda x: get_piece_value(*x),
    self.generate_all_pieces()
)
# sum values for static eval
return sum(values)

```

```

def make_move(self, from_position_vector: Vector, movement_vector: Vector):
    resultant_vector = from_position_vector + movement_vector

    # make a copy of the position vector, deep copy is used to ensure no
    parts are shared be reference
    new_pieces_matrix = deepcopy(self.pieces_matrix)

    # convert to list
    new_pieces_matrix = list(map(list, new_pieces_matrix))

    # look at square with position vector
    row, col = 7-from_position_vector.j, from_position_vector.i
    # get piece thats moving
    piece: pieces_mod.Piece = new_pieces_matrix[row][col]
    # set from square to blank
    new_pieces_matrix[row][col] = None

    # update piece to keep track of its last move
    piece.last_move = movement_vector

    # set to square to this piece
    row, col = 7-resultant_vector.j, resultant_vector.i
    new_pieces_matrix[row][col] = piece

    # convert back to tuple
    new_pieces_matrix = tuple(map(tuple, new_pieces_matrix))

    # update next to go
    new_next_to_go = "W" if self.next_to_go == "B" else "B"

    # return new board state instance
    return Board_State(
        next_to_go=new_next_to_go,
        pieces_matrix=new_pieces_matrix
    )

```

[game.py](#)

```

# import other modules
from board_state import Board_State
from minimax import minimax
from vector import Vector

```

```
from assorted import ARBITRARILY_LARGE_VALUE, dev_print, NotUserTurn,
InvalidMove

# the game class is used to keep track of a chess game between a user and the
computer
class Game(object):
    # it keeps track of:
    # the player's color's
    player_color_key: dict
    # the difficulty or depth of the game
    depth: int
    # the current board state
    board_state: Board_State
    # the number of moves so far
    move_counter: int
    # a table of game state hashes and there frequency
    piece_matrix_occurrence_hash_table: list

    # this adds a new game state to the frequency table
    def add_new_piece_matrix_to_hash_table(self, piece_matrix):
        # the pieces matrix is hashed
        matrix_hash = hash(piece_matrix)

        # if this pieces matrix has been encountered before, add 1 to the
        frequency
        if matrix_hash in self.piece_matrix_occurrence_hash_table.keys():
            self.piece_matrix_occurrence_hash_table[matrix_hash] += 1
        # else set frequency to 1
        else:
            self.piece_matrix_occurrence_hash_table[matrix_hash] = 1

    # determines if there is a 3 repeat stalemate
    def is_3_board_repeats_in_game_history(self):
        # if any of the board states have been repeated 3 or more times:
        stalemate
        return any(value >= 3 for value in
self.piece_matrix_occurrence_hash_table.values())

    # constructor for game object
    def __init__(self, depth=2, user_color="W") -> None:
        # based on user's color, determine color key
        self.player_color_key = {
            "W": 1 if user_color=="W" else -1,
            "B": -1 if user_color=="W" else 1
        }
        # set depth property from parameters
        self.depth = depth
```

```
# set attributes for game at start
self.board_state = Board_State()
self.move_counter = 0
self.piece_matrix_occurrence_hash_table = {}

# this function validates if the user's move is allowed and if so, makes it
def implement_user_move(self, from_square, to_square) -> None:
    # check that the user is allowed to move
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    if which_player_next_to_go != 1:
        raise NotUserTurn(game=self)

    # unpack move into vector form
    # invalid square syntaxes will cause a value error here
    try:
        position_vector = Vector.construct_from_square(from_square)
        movement_vector = Vector.construct_from_square(to_square) -
position_vector
    except Exception:
        raise ValueError("Square's not in valid format")

    # if the move is not in the set of legal moves, raise and appropriate exception
    if (position_vector, movement_vector) not in
self.board_state.generate_legal_moves():
        raise InvalidMove(game=self)

    # implement move
    self.board_state =
self.board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)
    # adjust other properties that keep track of the game
    self.move_counter += 1
    self.add_new_piece_matrix_to_hash_table(self.board_state.pieces_matrix
)

    return (position_vector, movement_vector),
self.board_state.static_evaluation()

# this function determines if the game is over and if so, what is the nature of the outcome
def check_game_over(self):
    # returns: over: bool, winning_player: (1/-1), classification: str
    # check for 3 repeat stalemate
    if self.is_3_board_repeats_in_game_history():
```

```

        return True, None, "Stalemate"

    # determine if board state is over for next player
    over, winner = self.board_state.is_game_over_for_next_to_go()
    # switch case statement to determine the appropriate values to be
    returned in each case
    match (over, winner):
        case False, _:
            victory_classification = None
            winning_player = None
        case True, None:
            victory_classification = "Stalemate"
            winning_player = None
        case True, winner:
            victory_classification = "Checkmate"
            winning_player = self.player_color_key[winner]
    # return appropriate values
    return over, winning_player, victory_classification

# this function determines and implements the computer move
def implement_computer_move(self, best_move_function=None):
    # for use with testing bots, a best move function can be provided for
    use, but minimax if the default

    # get next to go player (1/-1)
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    # check that is it the computer's turn
    if which_player_next_to_go != -1:
        raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

    # if no function provided, default to minimax
    if best_move_function is None:
        score, best_child, best_move = minimax(
            board_state = self.board_state,
            # assume white / user always maximizer
            # is_maximizer = (self.board_state.next_to_go == "W"),
            is_maximizer = False,
            # depth is based of difficulty of game based on depth
parameter
            depth = self.depth,
            # default values for alpha and beta
            alpha = (-1)*ARBITRARILY_LARGE_VALUE,
            beta = ARBITRARILY_LARGE_VALUE
        )
    else:

```

```
# otherwise use provided function,
# the provided function should take game as an argument and then
return data in the same format as the minimax function
score, best_child, best_move = best_move_function(self)

# adjust properties that keep track of the game state
self.board_state = best_child
self.move_counter += 1
self.add_new_piece_matrix_to_hash_table(self.board_state.pieces_matrix
)

# incase is it wanted for a print out ect, return move and score
return best_move, score
```

console_chess.py

```
from game import Game
from asserts import InvalidMove

# create new chess game
# difficulty set to depth 2
game = Game(depth=2)

# this function informs the user of the details when the game is over
def handle_game_over(winner, classification):
    print(f"The game is over, the {'user' if winner==1 else 'computer'} has
won in a {classification}")

# print out the starting board
game.board_state.print_board()
print()

# keep game going until loop broken
while True:
    # user goes first
    print("Your go USER:")

    # while loop and error checking used to ensure move input
    while True:
        try:
            print("Please enter move in 2 parts")
            from_square = input("From square: ")
            to_square = input("To square: ")
            # check
            game.implement_user_move(from_square, to_square)
        except InvalidMove as e:
            print(e)
```

```
except InvalidMove:  
    print("This isn't a legal move, try again")  
except ValueError:  
    print("This isn't valid input, try again")  
else:  
    # if it worked break out of the loop  
    break  
  
# if move results in check then output this  
if game.board_state.color_in_check():  
    print("CHECK!")  
# print out the current board_state  
game.board_state.print_board()  
print()  
  
# check if game over after user's move  
over, winner, classification = game.check_game_over()  
# if over, handle it.  
if over:  
    handle_game_over(winner=winner, classification=classification)  
    break  
  
# alternate, it is now the computers go  
print("Computer's go: ")  
  
# get the computers move  
move, _ = game.implement_computer_move()  
  
# print out the board again  
game.board_state.print_board()  
  
# print out the computer's move in terms of squares  
position_vector, movement_vector = move  
resultant_vector = position_vector + movement_vector  
piece_symbol =  
game.board_state.get_piece_at_vector(resultant_vector).symbol()  
print(f"Computer Moved {piece_symbol}: {position_vector.to_square()} to  
{resultant_vector.to_square()}")  
  
# print check if applicable  
if game.board_state.color_in_check():  
    print("CHECK!")  
  
# check if the game is over, if so handle it  
over, winner, classification = game.check_game_over()  
if over:  
    handle_game_over(winner=winner, classification=classification)  
    break
```

```
# create a new line to separate for the user's next move
print()
```

test_vector.py

```
import ddt
import unittest

# from vector.vector import Vector
from vector import Vector

# function for path to vector related test data
def test_path(file_name):
    return f"./test_data/vector/{file_name}.yaml"

# augment my test case class with ddt decorator
@ddt.ddt
class Test_Case(unittest.TestCase):

    # performs many checks of construct from square
    @ddt.file_data(test_path("from_square"))
    def test_square_to_vector(self, square, expected_vector):
        self.assertEqual(
            Vector.construct_from_square(square),
            Vector(*expected_vector),
            msg=f"\nVector.construct_from_square('{square}') != Vector(i={expected_vector[0]}, j={expected_vector[1]})"
        )

    # performs many checks of adding vectors
    @ddt.file_data(test_path("vector_add"))
    def test_add_vectors(self, vector_1, vector_2, expected_vector):
        self.assertEqual(
            Vector(*vector_1) + Vector(*vector_2),
            Vector(*expected_vector),
            msg=f"\nVector(*{vector_1}) +
Vector(*{vector_2}) != Vector(*{expected_vector})"
        )

    # performs many checks of multiplying vectors
    @ddt.file_data(test_path("vector_multiply"))
    def test_multiply_vectors(self, vector, multiplier, expected):
        self.assertEqual(
            Vector(*vector) * multiplier,
            Vector(*expected)
        )

    # many tests of vector in board
```

```

@ddt.file_data(test_path("vector_in_board"))
def test_in_board(self, vector, expected):
    self.assertEqual(
        Vector(*vector).in_board(),
        expected,
        msg=f"\nVector({vector}).in_board() != {expected}"
    )

# many tests of vector out of board
@ddt.file_data(test_path('vector_to_square'))
def test_to_square(self, vector, expected):
    self.assertEqual(
        Vector(*vector).to_square(),
        expected
    )

# if __name__ == '__main__':
#     unittest.main()

```

test_pieces.py

```

import unittest
import ddt

import pieces
# as vector is already tested we can use it here and assume it won't cause any
logic errors
from vector import Vector

def test_dir(file_name): return f"test_data/pieces/{file_name}.yaml"

EMPTY_PIECES_MATRIX = ((None,)*8,)*8

@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test is based on testing the movement vectors of a piece places
    at some position in an empty board are as expected
    @ddt.file_data(test_dir('test_empty_board'))
    def test_empty_board(self, piece_type, square, expected_move_squares):
        # deserialize
        position_vector = Vector.construct_from_square(square)
        piece: pieces.Piece = pieces.PIECE_TYPES[piece_type]('W')
        movement_vectors =
piece.generate_movement_vectors(pieces_matrix=EMPTY_PIECES_MATRIX,
position_vector=position_vector)
        resultant_squares = list(
            map(

```

```
        lambda movement_vector:
(movement_vector+position_vector).to_square(),
movement_vectors
    )
)
# assert as expected
# can use sets to prevent order being an issue as vectors are hashable
self.assertEqual(
    set(resultant_squares),
    set(expected_move_squares),
    msg=f"\n\nactual movement squares
{sorted(resultant_squares)} != expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
)

# this test assert that that a pieces movement vectors are as expected
when the piece is surrounded by other pieces
@ddt.file_data(test_dir('test_board_populated'))
def test_board_populated(self, pieces_matrix, square,
expected_piece_symbol, expected_move_squares):
    # deserialize
    def list_map(function, iterable): return list(map(function, iterable))

    def descriptor_to_piece(descriptor) -> pieces.Piece:
        # converts WN to knight object with a color attribute of white
        if descriptor is None:
            return None

        color, symbol = descriptor
        piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
        return piece_type(color=color)

    def row_of_symbols_to_pieces(row): return
list_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)

    position_vector = Vector.construct_from_square(square)
    row, column = 7-position_vector.j, position_vector.i

    # assert piece as expected
    piece: pieces.Piece = pieces_matrix[row][column]
    self.assertEqual(
        piece.symbol(),
        expected_piece_symbol,
        msg=f"\nPiece at square {square} was not the expected piece"
    )
```

```

        # assert movement vectors as expected
        movement_vectors =
piece.generate_movement_vectors(pieces_matrix=pieces_matrix,
position_vector=position_vector)
        resultant_squares = list(
            map(
                lambda movement_vector:
(movement_vector+position_vector).to_square(),
                movement_vectors
            )
        )
        self.assertEqual(
            set(resultant_squares),
            set(expected_move_squares),
            msg=f"\n\nactual movement squares
{sorted(resultant_squares)} != expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
        )

if __name__ == '__main__':
    unittest.main()

```

test_board_state.py

```

import unittest
import ddt

from board_state import Board_State
# tested so assumed correct
import pieces
from vector import Vector

def test_dir(file_name): return f"test_data/board_state/{file_name}.yaml"

# code used to deserialize
# code repeated from test_pieces, opportunity to reduce redundancy
def list_map(function, iterable): return list(map(function, iterable))
def tuple_map(function, iterable): return tuple(map(function, iterable))

def descriptor_to_piece(descriptor) -> pieces.Piece:
    # converts WN to knight object with a color attribute of white
    if descriptor is None:
        return None

    color, symbol = descriptor
    piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
    return piece_type(color=color)

```

```
def deserialize_pieces_matrix(pieces_matrix, next_to_go="W") -> Board_State:
    def row_of_symbols_to_pieces(row):
        return list_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)
    board_state: Board_State = Board_State(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)

    return board_state

@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test isn't data driven,
    # it tests that the static evaluation is 0 for starting positions
    def test_static_eval_starting_positions(self):
        self.assertEqual(
            Board_State().static_evaluation(),
            0
        )

    # this test is testing that a list of all pieces and there position
    vectors can be generated
    @ddt.file_data(test_dir('generate_all_pieces'))
    def test_generate_all_pieces(self, pieces_matrix, pieces_and_squares):
        pieces_and_squares = tuple_map(tuple, pieces_and_squares)

        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)
        # use set as order irrelevant
        all_pieces: set[pieces.Piece, Vector] =
set(board_state.generate_all_pieces())

        # convert square to vector, allowed as this is tested
        def deserialize(data_unit):
            # unpack test data unit
            descriptor, square = data_unit
            # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
            return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

        def serialize(data_unit):
            piece, vector = data_unit
            # return [piece.symbol(), vector.to_square()]


```

```
        return (piece.symbol(), vector.to_square())

    all_pieces_expected: set[pieces.Piece, Vector] = set(map(deserialize,
pieces_and_squares))
        # legal_moves_expected: list[pieces.Piece, Vector] = sorted(
        #     map(deserialize, pieces_and_squares),
        #     key=repr
        # )

        self.assertEqual(
            all_pieces,
            all_pieces_expected,
            msg=f"\nactual {list_map(serialize, all_pieces)} != expected
{list_map(serialize, all_pieces_expected)}"
        )

    # this test is to test the function responsible for getting the piece at a
given position vector
    @ddt.file_data(test_dir('piece_at_vector'))
    def test_piece_at_vector(self, pieces_matrix, vectors_and_expected_piece):
        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)

        # could use all method and one assert but this would have been less
readable, also hard to make useful message,
        # wanting different messages implies multiple asserts should be
completed
        for vector, expected_piece in vectors_and_expected_piece:
            # deserialize vector / cast to Vector
            vector: Vector = Vector(*vector)
            # repeat for piece
            expected_piece: pieces.Piece = descriptor_to_piece(expected_piece)

            # actual
            piece: piece.Piece = board_state.get_piece_at_vector(vector)
            msg = f"Piece at vector {repr(vector)} is {repr(piece)} not
expected piece {repr(expected_piece)}"

        # this test ensures that all the pieces belonging to a specific color and
there position vectors can be identified
        @ddt.file_data(test_dir('generate_pieces_of_color'))
        def test_generate_pieces_of_color(self, pieces_matrix, color,
pieces_and_squares):
            pieces_and_squares = tuple_map(tuple, pieces_and_squares)

            board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)
            # use set as order irrelevant
```

```
legal_moves_actual: set[pieces.Piece, Vector] =
set(board_state.generate_pieces_of_color(color))

# convert square to vector, allowed as this is tested
def deserialize(data_unit):
    # unpack test data unit
    descriptor, square = data_unit
    # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
    return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

def serialize(data_unit):
    piece, vector = data_unit
    # return [piece.symbol(), vector.to_square()]
    return (piece.symbol(), vector.to_square())

legal_moves_expected: set[pieces.Piece, Vector] = set(map(deserialize,
pieces_and_squares))

self.assertEqual(
    legal_moves_actual,
    legal_moves_expected,
    msg=f"\nactual {list_map(serialize,
legal_moves_actual)} != expected {list_map(serialize,
legal_moves_expected)}"
)

# this test ensures that the chess engine can determine if a specified
player is currently in check
@ddt.file_data(test_dir('color_in_check'))
def test_color_in_check(self, pieces_matrix, white_in_check,
black_in_check):
    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)

    self.assertEqual(
        board_state.color_in_check("W"),
        white_in_check,
        msg=f"white {'should' if white_in_check else 'should not'} be in
check but {'is' if board_state.color_in_check('W') else 'is not'}"
    )
    self.assertEqual(
        board_state.color_in_check("B"),
        black_in_check,
        msg=f"black {'should' if black_in_check else 'should not'} be in
check but {'is' if board_state.color_in_check('B') else 'is not'}"
    )
```

```
# this test ensures that a game over situation can be identified and its
nature discerned
@ddt.file_data(test_dir("game_over"))
def test_game_over(self, pieces_matrix, expected_over, expected_outcome,
next_to_go):
    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix,
next_to_go=next_to_go)

    self.assertEqual(
        board_state.is_game_over_for_next_to_go(),
        (expected_over, expected_outcome),
        msg=f"\nactual {board_state.is_game_over_for_next_to_go()} !=\nexpected {(expected_over, expected_outcome)}"
    )

# this is one of the most important functions to test
# this function determines all the possible legal moves a player can make
with their pieces, accounting for check
# this test checks this for many inputs
@ddt.file_data(test_dir("generate_legal_moves"))
def test_generate_legal_moves(self, pieces_matrix, next_to_go,
expected_legal_moves):
    board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix=pieces_matrix, next_to_go=next_to_go)
    actual_legal_moves = set(board_state.generate_legal_moves())

    # convert square to vector, allowed as this is tested
    def deserialize_expected_legal_moves():
        # unpack test data unit
        for test_datum in expected_legal_moves:
            from_square, all_to_squares = test_datum
            for to_square in all_to_squares:
                position_vector: Vector =
Vector.construct_from_square(from_square)
                    movement_vector: Vector =
Vector.construct_from_square(to_square) - position_vector

                yield (position_vector, movement_vector)

    expected_legal_moves = set(deserialize_expected_legal_moves())
    self.assertEqual(
        actual_legal_moves,
        expected_legal_moves,
    )

if __name__ == '__main__':
    unittest.main()
```

test_minimax.py

```
# this test is responsible for testing various mutations of the minimax
# function and how they play, it is not a data driven test

# imports
from random import choice as random_choice
import unittest
# from functools import wraps
import multiprocessing
import os.path
import csv

from game import Game
from minimax import minimax
from assorted import ARBITRARILY_LARGE_VALUE
# from board_state import Board_State
# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given
# function a given number of times
# def repeat_decorator_factory(times):
#     def decorator(func):
#         @wraps(func)
#         def wrapper(*args, **kwargs):
#             for _ in range(times):
#                 func(*args, **kwargs)
#         return wrapper
#     return decorator

# this is a utility function that maps a function across an iterable but also
# converts the result to a list data structure
def list_map(func, iter):
    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a message
# it converts pieces to symbols
def map_pieces_matrix_to_symbols(pieces_matrix):
    return list_map(
        lambda row: list_map(
            lambda square: square.symbol() if square else None,
            row
        ),
        pieces_matrix
    )
```

```
# this function updates a CSV file with the moves and scores of a chess game
# for graphical analysis in excel
def csv_write_move_score(file_path, move, score):
    # convert move to a pair of squares
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector

    from_square = position_vector.to_square()
    to_square = resultant_vector.to_square()

    # if file doesn't exist, create it and add the headers
    if not os.path.exists(file_path):
        with open(file_path, "w", newline="") as file:
            writer = csv.writer(file, delimiter=",")
            writer.writerow(("from_square", "to_square", "score"))

    # add data as a new row
    with open(file_path, "a", newline="") as file:
        writer = csv.writer(file, delimiter=",")
        writer.writerow((from_square, to_square, score))

# this contains the majority of the logic to do a bot vs bot test with the
# game class
# it is a component as it isn't the whole test
def minimax_test_component(description, good_bot, bad_bot, success_criteria,
write_to_csv=False):
    # print(f"CALL minimax_test_component(description={description},
    write_to_csv={write_to_csv})")
    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,
    # the test is designed to assert that good bot wins (and or draws in some
    cases)

    # generate csv path
    if write_to_csv:
        # not sure why but the description sometimes contains an erroneous
        colon, this is caught and removed
        # was able to locate bug to here, as it is a test I added a quick fix
        # bug located, some description strings included them
        csv_path = f"test_reports/{description}.csv".replace(" ",
        "_").replace(":", "")

        # prevent adding to a previous trial
        if os.path.exists(csv_path):
            # overwrite so it is blank
            with open(csv_path, "w") as file:
                file.write("")
```

```
# start a new blank game
# depth irrelevant as computer move function passed as parameter
game: Game = Game()

# keep them making moves until return statement breaks loop
while True:
    # get move choice from bad bot
    _, _, move_choice = bad_bot(game)

    # serialised to is can be passed as a user move (reusing game class)
    position_vector, movement_vector = move_choice
    resultant_vector = position_vector + movement_vector
    from_square, to_square = position_vector.to_square(),
    resultant_vector.to_square()

    # implement bad bot move and update csv
    move, score = game.implement_user_move(from_square=from_square,
    to_square=to_square)
    if write_to_csv:
        csv_write_move_score(
            file_path=csv_path,
            move=move,
            score=score
        )

    # see if this move causes the test to succeed or fail or keep going
    success, msg, board_state = success_criteria(game,
description=description)
    if success is not None:
        return success, msg, board_state

    # providing good bot function, implement good bot move and update csv
    move, score =
game.implement_computer_move(best_move_function=good_bot)
    if write_to_csv:
        csv_write_move_score(
            file_path=csv_path,
            move=move,
            score=game.board_state.static_evaluation()
        )

    # again check if this affects the test
    success, msg, board_state = success_criteria(game,
description=description)
    if success is not None:
        return success, msg, board_state
```

```

        # # if needed provide console output to clarify that slow bot hasn't
crashed
        # if game.move_counter % 10 == 0 or depth >= 3:
        # print(f"Moves {game.move_counter}: static evaluation ->
{game.board_state.static_evaluation()}, Minimax evaluation -> {score} by turn
{description}")

# below are some function that have been programmed as classes with a __call__
method.
# these are basically fancy functions that CAN BE HASHED.
# I had to manually do this under the hood hashing as it is needed to allow
communication between the threads
# a job must be hashable to be piped to a thread (separate python instance)

class Random_Bot():
    # picks a random move
    def __call__(self, game):
        # determine move at random
        legal_moves = list(game.board_state.generate_legal_moves())
        assert len(legal_moves) != 0
        # match minimax output structure
        # score, best_child, best_move
        return None, None, random_choice(legal_moves)
    def __hash__(self) -> int:
        return hash("I am random bot, I am a unique singleton so each instance
can share a hash")

# picks a good move
# has constructor to allow for configuration
class Good_Bot():
    # configure for depth and allow variable depth
    def __init__(self, depth, check_extra_depth):
        self.depth = depth
        self.check_extra_depth = check_extra_depth

    # make minimax function call given config
    def __call__(self, game):
        return minimax(
            board_state=game.board_state,
            is_maximizer=(game.board_state.next_to_go == "W"),
            depth=self.depth,
            alpha=(-1)*ARBITRARILY_LARGE_VALUE,
            beta=ARBITRARILY_LARGE_VALUE,
            check_extra_depth=self.check_extra_depth

```

```

        )
def __hash__(self) -> int:
    return hash(f"Good_Bot(depth={self.depth},
check_extra_depth={self.check_extra_depth})")

# used to look at a game and decide if the test should finish
class Success_Criteria():
    # constructor allow config for stalemates to sill allow test to pass
    def __init__(self, allow_stalemate_3_states_repeated: bool):
        self.allow_stalemate_3_states_repeated =
allow_stalemate_3_states_repeated
    def __call__(self, game: Game, description):
        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do
        match game.check_game_over():

            # if 3 repeat stalemate, check with config wether is is allows
            case True, None, "Stalemate":
                # game.board_state.print_board()
                if game.is_3_board_repeats_in_game_history and
self.allow_stalemate_3_states_repeated:
                    # game.board_state.print_board()
                    # print(f"Success: Stalemate at {game.moves} moves in test
{description}: 3 repeat board states, outcome specify included in allowed
outcomes")
                    return True, f"Success: Stalemate at {game.move_counter}
moves in test {description}: 3 repeat board states, outcome specify included
in allowed outcomes",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
                else:
                    # game.board_state.print_board()
                    # print(f"FAILURE: ({description}) stalemate caused (3
repeats? -> {game.is_3_board_repeats_in_game_history()})")
                    return False, f"FAILURE: ({description}) stalemate caused
(3 repeats? -> {game.is_3_board_repeats_in_game_history()})",

            map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
                # good bot loss causes test to fail
            case True, 1, "Checkmate":
                # game.board_state.print_board()
                # print(f"Failure: ({description}) computer lost")
                return False, f"Failure: ({description}) computer lost",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
                # good bot win causes test to pass
            case True, -1, "Checkmate":
                # game.board_state.print_board()
                # print(f"SUCCESS: ({description}) Game has finished and been
won in {game.move_counter} moves")

```

```
        return True, f"SUCCESS: ({description}) Game has finished and
been won in {game.move_counter} moves",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
    # if the game isn't over, return success as none and test will
continue
    case False, _, _:
        return None, None, None

def hash(self):
    return

hash(f"Success_Criteria(allow_stalemate_3_states_repeated={self.allow_stalemat
e_3_states_repeated})")

# given a test package (config for one test), carry it out
def execute_test_job(test_data_package):
    # deal with unexplained bug where argument is tuple / list length 1
    # containing relevant dict (quick fix as only a test)
    # I was able to identify that this is where it occurs and add a correction
    # but I am not sure what the cause of the bug is
    if any(isinstance(test_data_package, some_type) for some_type in (tuple,
list)):
        if len(test_data_package) == 1:
            test_data_package = test_data_package[0]

    # print(f"test_data_package --> {test_data_package}")

    # really simple, call minimax test component providing all keys in package
    # as keyword arguments
    return minimax_test_component(**test_data_package)

# this function takes an iterable of hashable test_packages
# it all 8 logical cores on my computer to multitask to finish the test sooner
def pool_jobs(test_data):
    # counts logical cores
    # my CPU is a 10th gen i7
    # it has 4 cores and 8 logical cores due to hyper threading
    # with 4-8 workers I can use 100% of my CPU
    cores = multiprocessing.cpu_count()

    # create a pool
    with multiprocessing.Pool(cores) as pool:
        # map the execute_test_job function across the set of test packages
        # using multitasking
        # return the result
        return pool.map(
```

```
        func = execute_test_job,
        iterable = test_data
    )

# test case contains unit tests
# multitasking only occurs within a test, tests are themselves executed
sequentially
# I could pool all tests into one test function but this way multiple failures
can occur in different tests
# (one single test would stop at first failure)

class Test_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and
    checks the results with a unittest
    # a failure is correctly identified to correspond to the function that
    called this function
    # reduces repeated logic
    def check_test_results(self, test_results):
        for success, msg, final_pieces_matrix in test_results:
            # print()
            # for row in final_pieces_matrix:
            #     row = " ".join(map(
            #         lambda square: str(square).replace("None", " . "),
            #         row
            #     ))
            #     print(row)
            # print(msg)

            # i choose to iterate rather than assert all as this allows me to
            have the appropriate message on failure
            self.assertTrue(
                success,
                msg=msg
            )

    # tests basic minimax vs random moves
    def test_vanilla_depth_1_vs_randomron(self):
        # 10 trials as outcome is linked to a random behaviour
        trials = 10

        # test package generated to include relevant data and logic (bots and
        success criteria)
        test_data = {
            "description": "test: depth 1 vanilla vs randotron",
            "good_bot": Good_Bot(depth=1, check_extra_depth=False),
            "bad_bot": Random_Bot(),
```

```

        "success_criteria":  

Success_Criteria(allow_stalemate_3_states_repeated=True),  

        "write_to_csv": False  

    }  
  

# only trial will write to a csv  

test_data_but_to_csv = test_data.copy()  

test_data_but_to_csv["write_to_csv"] = True  
  

self.check_test_results(  

    pool_jobs(  

        (trials-1) * [test_data] + [test_data_but_to_csv]  

    )  

)  
  

def test_advanced_depth_1_vs_randotron(self):  

    trials = 10  
  

    test_data = {  

        "description": "test: depth 1 advanced vs randotron",  

        "good_bot": Good_Bot(depth=1, check_extra_depth=True),  

        "bad_bot": Random_Bot(),  

        "success_criteria":  

Success_Criteria(allow_stalemate_3_states_repeated=True),  

        "write_to_csv": False  

    }  
  

# only trial will write to a csv  

test_data_but_to_csv = test_data.copy()  

test_data_but_to_csv["write_to_csv"] = True  
  

self.check_test_results(  

    pool_jobs(  

        (trials-1) * [test_data] + [test_data_but_to_csv]  

    )  

)  
  

def test_depth_2_vs_randotron(self):  

    trials = 10  
  

    test_data = {  

        "description": "test: depth 2 advanced vs randotron",  

        "good_bot": Good_Bot(depth=2, check_extra_depth=True),  

        "bad_bot": Random_Bot(),  

        "success_criteria":  

Success_Criteria(allow_stalemate_3_states_repeated=True),  

        "write_to_csv": False  

    }
}

```

```
        "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
        "write_to_csv": False  
    }  
  
    # only trial will write to a csv  
    test_data_but_to_csv = test_data.copy()  
    test_data_but_to_csv["write_to_csv"] = True  
  
    self.check_test_results(  
        pool_jobs(  
            (trials-1) * [test_data] + [test_data_but_to_csv]  
        )  
    )  
  
def test_depth_1_advanced_vs_depth_1_vanilla(self):  
    # only one trial needed as outcome is deterministic  
    trials = 1  
  
    test_data = {  
        "description": "test: depth 1 vanilla vs depth 1 variable check",  
        "good_bot": Good_Bot(depth=1, check_extra_depth=True),  
        "bad_bot": Good_Bot(depth=1, check_extra_depth=False),  
        # they aren't different enough in efficacy to guarantee no draws  
        "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=True),  
        "write_to_csv": False  
    }  
  
    # only trial will write to a csv  
    test_data_but_to_csv = test_data.copy()  
    test_data_but_to_csv["write_to_csv"] = True  
  
    self.check_test_results(  
        pool_jobs(  
            (trials-1) * [test_data] + [test_data_but_to_csv]  
        )  
    )  
  
def test_depth_2_vs_depth_1(self):  
    trials = 1  
  
    test_data = {  
        "description": "test: depth 2 vs depth 1",  
        "good_bot": Good_Bot(depth=2, check_extra_depth=True),  
        "bad_bot": Good_Bot(depth=1, check_extra_depth=True),
```

```
        "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
        "write_to_csv": False  
    }  
  
    # only trial will write to a csv  
    test_data_but_to_csv = test_data.copy()  
    test_data_but_to_csv["write_to_csv"] = True  
  
    self.check_test_results(  
        pool_jobs(  
            (trials-1) * [test_data] + [test_data_but_to_csv]  
        )  
    )  
  
def test_depth_3_vs_depth_2(self):  
    trials = 1  
  
    test_data = {  
        "description": "test: depth 3 vs depth 2",  
        "good_bot": Good_Bot(depth=3, check_extra_depth=True),  
        "bad_bot": Good_Bot(depth=2, check_extra_depth=True),  
        "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
        "write_to_csv": False  
    }  
  
    # only trial will write to a csv  
    test_data_but_to_csv = test_data.copy()  
    test_data_but_to_csv["write_to_csv"] = True  
  
    self.check_test_results(  
        pool_jobs(  
            (trials-1) * [test_data] + [test_data_but_to_csv]  
        )  
    )  
  
def test_depth_3_vs_depth_1(self):  
    trials = 1  
  
    test_data = {  
        "description": "test: depth 3 vs depth 1",  
        "good_bot": Good_Bot(depth=3, check_extra_depth=True),  
        "bad_bot": Good_Bot(depth=1, check_extra_depth=True),  
        "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
        "write_to_csv": False  
    }
```

```

# only trial will write to a csv
test_data_but_to_csv = test_data.copy()
test_data_but_to_csv["write_to_csv"] = True

self.check_test_results(
    pool_jobs(
        (trials-1) * [test_data] + [test_data_but_to_csv]
    )
)

def test_depth_3_vs_randotron(self):
    trials = 4

    test_data = {
        "description": "test: depth 3 vs randotron",
        "good_bot": Good_Bot(depth=3, check_extra_depth=True),
        "bad_bot": Random_Bot(),
        "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
        "write_to_csv": False
    }

    # only trial will write to a csv
    test_data_but_to_csv = test_data.copy()
    test_data_but_to_csv["write_to_csv"] = True

    self.check_test_results(
        pool_jobs(
            (trials-1) * [test_data] + [test_data_but_to_csv]
        )
    )

if __name__ == '__main__':
    unittest.main()

```

[test_data\vector\from_square.yaml](#)

```

test1:
    square: 'G3'
    expected_vector: [6, 2]

test2:
    square: 'A8'
    expected_vector: [0, 7]

```

```
test3:  
    square: 'H1'  
    expected_vector: [7, 0]  
  
test4:  
    square: 'C4'  
    expected_vector: [2, 3]  
  
test5:  
    square: 'F6'  
    expected_vector: [5, 5]  
  
# # invalid, used to check the unittest is working  
# test6:  
#     square: 'A5'  
#     expected_vector: [5, 5]
```

test_data\vector\vector_add.yaml

```
test1:  
    vector_1: [1, 4]  
    vector_2: [4, 6]  
    expected_vector: [5, 10]  
  
test2:  
    vector_1: [1, 7]  
    vector_2: [4, 6]  
    expected_vector: [5, 13]  
  
test3:  
    vector_1: [1, 3]  
    vector_2: [0, 6]  
    expected_vector: [1, 9]  
  
test4:  
    vector_1: [-1, 32]  
    vector_2: [3, 6]  
    expected_vector: [2, 38]  
  
test5:  
    vector_1: [6, 0]  
    vector_2: [6, 7]  
    expected_vector: [12, 7]  
  
# # invalid test
```

```
# test6:  
#   vector_1: [6, 0]  
#   vector_2: [6, 7]  
#   expected_vector: [0, 7]
```

test data\vector\vector in board.yaml

```
test1:  
  vector: [0, 0]  
  expected: True
```

```
test2:  
  vector: [7, 0]  
  expected: True
```

```
test3:  
  vector: [0, 7]  
  expected: True
```

```
test4:  
  vector: [7, 7]  
  expected: True
```

```
test5:  
  vector: [4, 6]  
  expected: True
```

```
test6:  
  vector: [3, 2]  
  expected: True
```

```
test7:  
  vector: [-1, -1]  
  expected: False
```

```
test8:  
  vector: [-5, 4]  
  expected: False
```

```
test9:  
  vector: [8, 7]  
  expected: False
```

```
test10:  
  vector: [-4, 4]
```

```
expected: False

test11:
    vector: [10, 4]
    expected: False

# # adding comment inside causes logic error and false pass on test
# # invalid delete me
# test:
#     vector: [4, 6]
#     expected: False
```

test_data\vector\vector_multiply.yaml

```
test1:
    vector: [1, 1]
    multiplier: 1
    expected: [1, 1]
test2:
    vector: [1, 1]
    multiplier: -1
    expected: [-1, -1]
test3:
    vector: [1, 1]
    multiplier: 5
    expected: [5, 5]
test4:
    vector: [5, 0]
    multiplier: 0
    expected: [0, 0]
```

test_data\vector\vector_to_square.yaml

```
test1:
    vector: [0, 0]
    expected: A1
test2:
    vector: [5, 7]
    expected: F8
test3:
    vector: [6, 2]
    expected: G3
test4:
    vector: [0, 6]
```

```
expected: A7
test5:
vector: [7, 7]
expected: H8
```

test data\pieces\test board populated.yaml

```
test1:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
]
square: B1
expected_piece_symbol: WP
expected_move_squares: [
    A2, C2, B2, B3
]
test2:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BQ, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D4
expected_piece_symbol: BQ
expected_move_squares: [
    D5,
    E4, F4, G4,
    E3, F2, G1,
    D3, D2, D1,
    C4,
    C5, B6, A7
]
test3:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
```

```
[null, null, null, null, BR, null, null, null],
[null, BR, null, BR, null, null, WB, null],
[null, null, BP, null, null, null, BP, null],
[null, WR, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null]
]
square: D4
expected_piece_symbol: BR
expected_move_squares: [
    D5,
    E4, F4, G4,
    D3, D2, D1,
    C4,
]
test4:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, WP, null, WP, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D7
expected_piece_symbol: BP
expected_move_squares: [
    D6, D5,
    C6, E6
]
test5:
pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D8
expected_piece_symbol: BK
expected_move_squares: [
    C7, D7, E7,
    C8, E8
]
```

```

test6:
pieces_matrix: [
    [BR,    BK,    BB,    BK,    BQ,    BB,    BK,    BR ],
    [BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP ],
    [null, null, null, null, null, null, null, null],
    [WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP ],
    [WR,    WN,    WB,    WK,    WQ,    WB,    WN,    WR ]
]
square: B1
expected_piece_symbol: WN
expected_move_squares: [A3, C3]

test7:
pieces_matrix: [
    [BR,    BN,    BB,    BK,    BQ,    BB,    BN,    BR ],
    [BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP ],
    [null, null, null, null, null, null, null, null],
    [WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP ],
    [WR,    WN,    WB,    WK,    WQ,    WB,    WN,    WR ]
]
square: G1
expected_piece_symbol: WN
expected_move_squares: [F3, H3]

```

test_data\pieces\test_empty_board.yaml

```

test1:
piece_type: 'N'
square: E4
expected_move_squares: [
    D2, F2, D6, F6, C5, C3, G5, G3
]
test2:
piece_type: Q
square: C3
expected_move_squares: [
    A1, B2, D4, E5, F6, G7, H8,
    C1, C2, C4, C5, C6, C7, C8,
    A3, B3, D3, E3, F3, G3, H3,
    A5, B4, D2, E1
]
test3:
piece_type: K
square: B2

```

```

expected_move_squares: [
    C1, C2, C3,
    B1, B3,
    A1, A2, A3,
]
test4:
  piece_type: P
  square: E3
  expected_move_squares: [
    E4, E5
  ]
test5:
  piece_type: R
  square: F6
  expected_move_squares: [
    F1, F2, F3, F4, F5, F7, F8,
    A6, B6, C6, D6, E6, G6, H6
  ]
test6:
  piece_type: B
  square: D7
  expected_move_squares: [
    C8, E6, F5, G4, H3,
    E8, C6, B5, A4,
  ]
test7:
  piece_type: K
  square: D8
  expected_move_squares: [
    C7, D7, E7,
    C8, E8
  ]

```

test data\board state\color in check.yaml

```

test1:
  pieces_matrix: [
    [BR, BK, BB, BK, BQ, BB, BK, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WK, WB, WK, WQ, WB, WK, WR ]
  ]
  white_in_check: false
  black_in_check: false
test2:

```

```
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, WK, WQ, BK],
    [null, null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null, null],
]

white_in_check: false
black_in_check: true

test3:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null, BQ],
    [null, null, null, null, null, null, null, null, null],
    [null, null, null, WP, WP, null, null, null, null],
    [null, null, null, WB, WK, WB, null, null, null],
]
white_in_check: true
black_in_check: false

test4:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, WK, null, BK],
    [null, null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null, WR]
]
white_in_check: false
black_in_check: true

test5:
pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
```

```
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: false
test6:
pieces_matrix: [
    [null, null, BK, null, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true
test7:
pieces_matrix: [
    [null, null, null, null, BK, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true
test8:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, WP, BK, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: true
black_in_check: true
test9:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
```

```
[null, null, BK, WP, null, null, null, null],
[null, null, null, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
white_in_check: true
black_in_check: true
```

test data\board state\game over.yaml

```
test1:
  next_to_go: W
  pieces_matrix: [
    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP ],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP ],
    [WR,   WK,   WB,   WK,   WQ,   WB,   WK,   WR ]
  ]
  expected_over: false
  expected_outcome: null
test2:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK,   null, null, null, null],
    [null, null, null, WP,   null, null, null, null],
    [null, null, null, WK,   null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_over: true
  expected_outcome: null
test3:
  next_to_go: B
  pieces_matrix: [
    [WR,   null, null, null, null, null, BK,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, WK ],
]
expected_over: true
expected_outcome: W
test4:
next_to_go: B
pieces_matrix: [
    [null, null, null, BK,    null, null, null, null],
    [null, null, null, WP,    null, null, null, null],
    [null, null, WP,     WK,    null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
expected_over: true
expected_outcome: null
test5:
next_to_go: B
pieces_matrix: [
    [null, null, null, BK,    null, null, null, null],
    [null, null, WP,     WP,    null, null, null, null],
    [null, null, null, WK,    null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
expected_over: true
expected_outcome: W
test6:
next_to_go: W
pieces_matrix: [
    [BR,    null, null, null, null, BR,    BK,    null],
    [BP,    null, null, null, null, BP,    BP,    null],
    [null, BP,    null, null, null, null, null, null],
    [null, null, BQ,    null, null, null, null, null],
    [null, null, null, null, null, null, null, WQ ],
    [null, null, null, null, null, null, WP,    null],
    [WP,    null, null, null, null, WP,    WB,    null],
    [WR,    null, null, null, null, null, null, WR ],
]
expected_over: false
```

```

expected_outcome: null
test7:
  next_to_go: B
  pieces_matrix: [
    [BR,  null,  null,  null,  null,  BR,  BK,  WQ  ],
    [BP,  null,  null,  null,  null,  BP,  BP,  null],
    [null,  BP,  null,  null,  null,  null,  null,  null],
    [null,  null,  BQ,  null,  null,  null,  null,  null],
    [null,  null,  null,  null,  null,  null,  null,  null],
    [null,  null,  null,  null,  null,  null,  null,  null],
    [null,  null,  null,  null,  null,  null,  WP,  null],
    [WP,  null,  null,  null,  null,  WP,  WB,  null],
    [WR,  null,  null,  null,  null,  null,  null,  WR  ],
  ]
  expected_over: true
  expected_outcome: W

```

test data\board state\generate all pieces.yaml

```

test1:
  pieces_matrix: [
    [null,  null,  null,  null,  null,  null,  null,  null],
    [BP,  null,  BP,  null,  null,  null,  null,  null],
    [null,  WP,  null,  null,  null,  null,  null,  null]
  ]
  pieces_and_squares: [
    [WP,  B1],
    [BP,  A2],
    [BP,  C2]
  ]

test2:
  pieces_matrix: [
    [null,  null,  null,  null,  null,  null,  null,  null],
    [null,  null,  null,  null,  null,  null,  null,  null],
    [null,  null,  null,  BP,  null,  null,  null,  null],
    [null,  null,  null,  null,  BR,  null,  null,  null],
    [null,  BR,  null,  BR,  null,  null,  WB,  null],
    [null,  null,  BP,  null,  null,  null,  BP,  null],
    [null,  WR,  null,  null,  null,  null,  null,  null],
    [null,  null,  null,  null,  null,  null,  null,  null]
  ]
  pieces_and_squares: [

```

```
[WR, B2],
[BR, B4],
[BP, C3],
[BR, D4],
[BP, D6],
[BP, G3],
[WB, G4],
[BR, E5]
]
```

test_data\board state\generate legal moves.yaml

```
test1:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_legal_moves: []
test2:
  next_to_go: W
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_legal_moves: [
    [B1, [B2, B3, A2, C2]]
  ]
test3:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[BP, null, BP, null, null, null, null, null],
[null, WP, null, null, null, null, null, null]
]
expected_legal_moves: [
    [A2, [A1, B1]],
    [C2, [C1, B1]],
]
test4:
next_to_go: W
pieces_matrix: [
    [BR, BN, BB, BK, BQ, BB, BN, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WN, WB, WK, WQ, WB, WN, WR ]
]
expected_legal_moves: [
    [A2, [A3, A4]],
    [B2, [B3, B4]],
    [C2, [C3, C4]],
    [D2, [D3, D4]],
    [E2, [E3, E4]],
    [F2, [F3, F4]],
    [G2, [G3, G4]],
    [H2, [H3, H4]],
    [B1, [A3, C3]],
    [G1, [F3, H3]]
]
test5:
next_to_go: W
pieces_matrix: [
    [BK, BP, null, null, null, null, WP, WK],
    [BP, BP, null, null, null, null, WP, WP],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WN, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
expected_legal_moves: [
    [B2, [A4, C4, D1, D3]],
]
```

```
[G4, [
    H3, F5, E6, D7, C8,
    H5, F3, E2, D1
]],
]
```

test data\board state\generate pieces of color.yaml

```
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]
  color: W
  pieces_and_squares: [
    [WP, B1]
  ]
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]
  color: B
  pieces_and_squares: [
    [BP, A2],
    [BP, C2]
  ]
test3:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
```

```

[null, null, null, null, null, null, null, null]
]
color: W
pieces_and_squares: [
  [WR, B2],
  [WB, G4]
]
test4:
pieces_matrix: [
  [null, null, null, null, null, null, null, null],
  [null, null, null, null, null, null, null, null],
  [null, null, null, BP, null, null, null, null],
  [null, null, null, null, BR, null, null, null],
  [null, BR, null, BR, null, null, WB, null],
  [null, null, BP, null, null, null, BP, null],
  [null, WR, null, null, null, null, null, null],
  [null, null, null, null, null, null, null, null]
]
color: B
pieces_and_squares: [
  [BR, B4],
  [BP, C3],
  [BR, D4],
  [BP, D6],
  [BP, G3],
  [BR, E5]
]

```

test data\board state\piece at vector.yaml

```

test1:
pieces_matrix: [
  [BR, BN, BB, BK, BQ, BB, BN, BR ],
  [BP, BP, BP, BP, BP, BP, BP, BP ],
  [null, null, null, null, null, null, null, null],
  [WP, WP, WP, WP, WP, WP, WP, WP ],
  [WR, WN, WB, WK, WQ, WB, WN, WR ]
]
vectors_and_expected_piece: [
  [[0, 0], WR],
  [[1, 1], WP],
  [[2, 4], null],
  [[7, 3], null],
  [[5, 7], BB],

```

```
[[2, 6], BP]
]

test2:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
vectors_and_expected_piece: [
    [[0, 0], null],
    [[1, 1], WR],
    [[1, 3], BR],
    [[0, 7], null],
    [[7, 0], null],
    [[7, 7], null],
    [[3, 3], BR],
    [[3, 5], BP],
    [[6, 3], WB],
    [[6, 1], null]
]
```

Prototype 3: Final code

Directory Structure

```
PS C:\Users\henry\Documents\computing coursework\code list\prototype 3> tree
/f
Folder PATH listing for volume OS
Volume serial number is 4ED8-B070
C:.
|   app.py
|   __init__.py
|
+-assorted
    chess_exceptions.py
    general.py
    safe_hash.py
    __init__.py
|
+-chess_functions
```

```
board_state.py
pieces.py
test_fast_board_state.py
test_fast_pieces.py
test_fast_vector.py
vector.py
__init__.py

test_data
└── board_state
    color_in_check.yaml
    game_over.yaml
    generate_all_pieces.yaml
    generate_legal_moves.yaml
    generate_pieces_of_color.yaml
    piece_at_vector.yaml

└── pieces
    test_board_populated.yaml
    test_empty_board.yaml

└── vector
    from_square.yaml
    vector_add.yaml
    vector_in_board.yaml
    vector_multiply.yaml
    vector_to_square.yaml

chess_game
└── console_chess.py
    game.py
    game_web.py
    test_fast_game.py
    __init__.py

test_data
└── test_save_and_restore
    saved_game.game

database
└── create_database.py
    database.db
    handle_games.py
    models.py
    __init__.py

move_engine
└── cache_managers.py
```

```
minimax.py
minimax_parallel.py
parallel_minimax_testing.py
test_slow_timed_minimax_engine.py
__init__.py

test_reports
    test_depth_2_vs_depth_1
        test_1.csv
        test_1.games

    test_depth_3_vs_depth_2
        test_1.csv
        test_1.games

schemas
    cache_item_schema.py
    socket_schemas.py
    __init__.py

test_reports
    test_bots_by_depth
        test_depth_2_vs_depth_1.csv
        test_depth_3_vs_depth_2.csv

    test_bots_by_time
        test_10s_timed_bot_vs_2s_timed_bot.csv
        test_10s_timed_bot_vs_5s_timed_bot.csv
        test_12s_timed_bot_vs_2s_timed_bot.csv
        test_13s_timed_bot_vs_5s_timed_bot.csv
        test_14.0s_timed_bot_vs_10s_timed_bot.csv
        test_15s_timed_bot_vs_5s_timed_bot.csv
        test_17s_timed_bot_vs_2s_timed_bot.csv
        test_20s_timed_bot_vs_10s_timed_bot.csv
        test_20s_timed_bot_vs_5s_timed_bot.csv
        test_21.0s_timed_bot_vs_15s_timed_bot.csv
        test_28.0s_timed_bot_vs_20s_timed_bot.csv
        test_30s_timed_bot_vs_15s_timed_bot.csv
        test_40s_timed_bot_vs_10s_timed_bot.csv
        test_40s_timed_bot_vs_20s_timed_bot.csv
        test_60s_timed_bot_vs_30s_timed_bot.csv
        test_6s_timed_bot_vs_2s_timed_bot.csv
        test_7s_timed_bot_vs_2s_timed_bot.csv
        test_9s_timed_bot_vs_5s_timed_bot.csv

    test_depth_vs_randomtron
        test_depth_1_num_1.csv
        test_depth_1_num_2.csv
```

```

|           test_depth_1_num_3.csv
|           test_depth_2_num_1.csv
|           test_depth_2_num_2.csv
|           test_depth_2_num_3.csv
|
└── website
    ├── flask_server.py
    ├── secret_key.key
    └── __init__.py
|
└── static
    ├── chess_class.js
    ├── favicon.ico
    ├── initial_game_data.js
    ├── initial_game_data.json
    ├── main.js
    ├── style.css
    └── vector.js
|
└── templates
    └── chess_game.html

```

PS C:\Users\henry\Documents\computing coursework\code list\prototype 3>

init .py

(the file is empty but its existing allows to do module imports)

app.py

```

from website import app
from database import persistent_DB_engine, volatile_RAM_engine, end_engines

# variables to decide the URL of the website
host = '127.0.0.1'
port_num = 5000
url = f"http://[{host}]:{port_num}"

# this function runs the flask server (hosting the website)
# to close the server, use ctrl + c to create a keyboard interrupt,
# this will safely destroy all engine connections to the database
def run_app():
    print(url)
    try:
        app.run(
            host=host,
            port=port_num,
            debug=True
        )
    
```

```

        except KeyboardInterrupt:
            end_engines([persistent_DB_engine, volatile_RAM_engine])

# use python -m flask run to stop it raising warnings about how it should be
deployed
if __name__ == "__main__":
    run_app()

```

assorted__init__.py

```

# this file decides that components from within this folder are exported as
part of the assorted module
from .general import ARBITRARILY_LARGE_VALUE, cache_decorator, dev_print
from .safe_hash import safe_hash
from .chess_exceptions import TimeOutError, InvalidMove, NotUserTurn,
NotComputerTurn, UnexplainedErroneousMinimaxResultError

```

assorted\chess_exceptions.py

```

# this error is used to cause the timed minimax function
# raising and then catching this error allows for the algorithm to be self-
interrupting
class TimeOutError(Exception):
    pass

# this is an exception that allows for the game data to be bound to it
# this allows for the relevant chess game that caused the error to be examined
# afterwards
# it is a normal exception except the constructor has been modified to save
the game data as a property
class __ChessExceptionTemplate__(Exception):
    def __init__(self, *args, **kwargs) -> None:
        # none if key not present
        self.game = kwargs.pop("game", None)

    super().__init__(*args, **kwargs)

# these are custom exceptions.
# these are used primarily by the game class
# they contain no logic but have distinct types allowing for targeted error
handling
class InvalidMove(__ChessExceptionTemplate__):

```

```
pass

class NotUserTurn(__ChessExceptionTemplate__):
    pass

class NotComputerTurn(__ChessExceptionTemplate__):
    pass

class UnexplainedErroneousMinimaxResultError(__ChessExceptionTemplate__):
    pass
```

assorted\general.py

```
# this file is just a file of short assorted constants and functions that are
general in use
# It only contains small functions as I have tried to group large, similar
functions logically in there own file

# from functools import lru_cache
# cache_decorator = lru_cache(maxsize=10000)
def cache_decorator(func): return func

# this is used in the static evaluation and minimax process. It is used to
represent infinity in a way that still allows comparrison
ARBITRARILY_LARGE_VALUE = 1_000_000

# this function is relatively redundant but allows for print statements in
debugging
# in later iteration this may be replaced with logging.
# it is useful as it allows for DEBUG print statements without needing to
remove them when finished
DEBUG = False
def dev_print(*args, **kwargs):
    if DEBUG:
        print(*args, **kwargs)
```

assorted\safe hash.py

```
from hashlib import sha256
```

```
# this hash function produces an integer hash of a python object
# it is necessary as the in built hash function produces different hashes for
# the same object,
# this function always produces the same hash for a given object, this allows
# the hash to be stored in a database for later use
def safe_hash(item):
    # https://stackoverflow.com/questions/30585108/disable-hash-randomization-
    # from-within-python-program

    # convert item (usually tuple) to string
    encoded_string = repr(item).encode("utf-8")
    # produce a hexadecimal string hash of the object
    hex_hash = sha256(encoded_string).hexdigest()
    # convert this to an integer
    # int_hash = int(hex_hash, 16)
    # return int_hash
    return hex_hash
```

chess functions__init__.py

```
# this file decided which function and classes from within this folder should
# be exported as part of the chess functions module.

from .board_state import Board_State, random_board_state
from .vector import Vector
from .pieces import PIECE_TYPES, Piece, Pawn, Knight, Bishop, Rook, King,
Queen
```

chess functions\vector.py

```
# import dataclass to reduce boilerplate code
from dataclasses import dataclass

from assorted import cache_decorator, safe_hash

# frozen = true means that the objects will be immutable
@dataclass(frozen=True)
class Vector():
    # 2d vector has properties i and j
    i: int
    j: int
```

```
# code to allow for + - and * operators to be used with vectors
@cache_decorator
def __add__(self, other):
    assert isinstance(other, Vector), "both objects must be instances of
the Vector class"
    return Vector(
        i=self.i + other.i,
        j=self.j + other.j
    )

@cache_decorator
def __sub__(self, other):
    assert isinstance(other, Vector), "both objects must be instances of
the Vector class"
    return Vector(
        i=self.i - other.i,
        j=self.j - other.j
    )

@cache_decorator
def __mul__(self, multiplier: int | float):
    return Vector(
        i=int(self.i * multiplier),
        j=int(self.j * multiplier)
    )

# check if a vector is in board
@cache_decorator
def in_board(self):
    """Assumes that the current represented vector is a position vector
    checks if it points to a square that isn't in the chess board"""
    return self.i in range(8) and self.j in range(8)

# alternative way to create instance, construct from chess square
@classmethod
@cache_decorator
def construct_from_square(cls, to_sqr):
    """Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1,
3)"""
    to_sqr = to_sqr.upper()
    letter, number = to_sqr

    # map letters and numbers to 0 to 7 and create new vector object
    return cls(
        i=ord(letter.upper()) - ord("A"),
        j=int(number)-1
    )
```

```

# this function is the reverse and converts a position vector to a square
@cache_decorator
def to_square(self) -> str:
    letter = chr(self.i + ord("A"))
    if letter == "@":
        print(repr(self))
    number = self.j+1
    return f"{letter}{number}"

@cache_decorator
def magnitude_and_unit_vector(self):
    def sqrt(x):
        return x**(1/2)
    def square(x):
        return x**2

    magnitude: float = sqrt(square(self.i) + square(self.j))
    # unit_vector: Vector = self.__mul__(1/magnitude)
    # return magnitude, unit_vector

    return magnitude

# this function checks if 2 vectors are equal
def __eq__(self, other) -> bool:
    try:
        # assert same subclass like rook
        assert isinstance(other, type(self))
        assert self.i == other.i
        assert self.j == other.j

    except AssertionError:
        return False
    else:
        return True

# used to put my objects in set
def __hash__(self):
    return hash(safe_hash(
        (self.i, self.j)
    ))

```

chess functions\pieces.py

```

# to do: make value and value matrix unchangeable of private
# https://stackoverflow.com/questions/31457855/cant-instantiate-abstract-
class-with-abstract-methods

```

```
# import libraries and other local modules
import abc
from itertools import chain as iter_chain, product as iter_product
from typing import Callable
import json

# from vector import Vector
from .vector import Vector
# from ..vector import Vector
# from chess_functions.vector import Vector

from asserted import ARBITRARILY_LARGE_VALUE, safe_hash, cache_decorator

# Here is an abstract base class for piece,
# it dictates that all child object have the specified abstract attributes
# else an error will occur
# this ensures that all piece objects have the same interface

class Abstract_Piece_Interface(abc.ABC):
    # must be given before init
    @abc.abstractproperty
    def _value_matrix(): pass

    @abc.abstractproperty
    def _value(): pass

    @abc.abstractproperty
    def color(): pass

    @abc.abstractmethod
    def symbol(self) -> str:
        """uses color to determine the appropriate symbol"""

    # this function should yield all the movement vector tha the piece can
    move by
    # this doesn't account for check and is based on rules specific to each
    piece as well as checking if a vector is outside the board
    @abc.abstractmethod
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        pass

class Piece():
    # not needed as abstract method as some classes will now override
    def __init__(self, color):
        self.color = color
        self.last_move = None
```

```
# this should use the position vector and value matrix to get the value of
the piece
@cache_decorator
def get_value(self, position_vector: Vector):
    # flip if black as matrices are all for white pieces
    if self.color == "W":
        row, column = 7-position_vector.j, position_vector.i
    else:
        row, column = position_vector.j, position_vector.i

    # return sum of inherent value + value relative to positon on board
    return self._value + self._value_matrix[row][column]

# when str(piece) called give the symbol
@cache_decorator
def __str__(self):
    # return f"{self.color}{self.symbol}"
    return self.symbol()

# standard repr method
@cache_decorator
def __repr__(self):
    return f"{type(self).__name__}(color='{self.color}')"

# logic that would be otherwise repeated in many of the child classes
# determines the contents of a given square
@cache_decorator
def square_contains(self, square):
    """returns 'enemy' 'ally' or None for empty"""
    # check if empty
    if square is None:
        return "empty"
    # else the square must contain a piece, so examine its color
    if square.color == self.color:
        return "ally"
    else:
        return "enemy"

# again reduces repeated logic
# checks the result of a position vector
# if not illegal (out of board) the square contents is returned
@cache_decorator
def examine_position_vector(self, position_vector: Vector, pieces_matrix):
    """returns 'enemy' 'ally' 'empty' or 'illegal' """
    # check if the vector is out of the board
    if not position_vector.in_board():
        return 'illegal'
    # for the rest of the code I can assume the vector is in board
```

```
# else get the square at that vector
row, column = 7-position_vector.j, position_vector.i
square = pieces_matrix[row][column]

# examine its contents
return self.square_contains(square)

# equality operator
def __eq__(self, other):
    try:
        # assert same subclass like rook
        assert isinstance(other, type(self))
        assert self.color == other.color

        # i am not checking that pieces had the same last move as I want
        # to compare kings without for the check function
        # assert self.last_move == other.last_moves

        # value and value_matrix should never be changes
    except AssertionError:
        return False
    else:
        return True

# making pieces hashable allows for pieces matrices to be hashed and
# allows for pieces and pieces matrices to be put in sets,
# also essential for piping data between python interpreter instances
# (different threads) for multitasking
def __hash__(self):
    return hash(safe_hash((self.symbol(), self.color, self.last_move)))

# def __json__(self):
#     return {
#         'type': type(self).__name__,
#         'color': self.color,
#         'last_move': self.last_move,
#     }

# def to_json(self):
#     return json.dumps(self.__json__())

# @classmethod
# def from_json(cls, json_data):
#     data = json.loads(json_data)
#     piece_type = data['type']
#     color = data['color']
```

```
#     last_move = data['last_move']

#     # instantiate the appropriate subclass of Piece based on the type
field
#     if piece_type == 'Pawn':
#         return Pawn(color, last_move)
#     elif piece_type == 'Rook':
#         return Rook(color, last_move)
#     elif piece_type == 'Knight':
#         return Knight(color, last_move)
#     elif piece_type == 'Bishop':
#         return Bishop(color, last_move)
#     elif piece_type == 'Queen':
#         return Queen(color, last_move)
#     elif piece_type == 'King':
#         return King(color, last_move)
#     else:
#         raise ValueError(f"Unknown Piece type: {piece_type}")

# restored from v2.4 still has issue with pawn moving foreword 2

# # this class inherits from Piece an so it inherits some logic and some
requirements as to how its interface should be
# # as many child classes are similar I will explain this one in depth and
then only explain notable features of others
# class Pawn(Piece):
#     # defining abstract properties, needed before init
#     _value = 100
#     _value_matrix: tuple[tuple[float]] = [
#         [0, 0, 0, 0, 0, 0, 0, 0],
#         [50, 50, 50, 50, 50, 50, 50, 50],
#         [10, 10, 20, 30, 30, 20, 10, 10],
#         [5, 5, 10, 25, 25, 10, 5, 5],
#         [0, 0, 0, 20, 20, 0, 0, 0],
#         [5, -5, -10, 0, 0, -10, -5, 5],
#         [5, 10, 10, -20, -20, 10, 10, 5],
#         [0, 0, 0, 0, 0, 0, 0, 0]
#     ]
#     color = None

#     # define symbol method (str method)
#     def symbol(self): return f"{self.color}P"

#     # override init constructor
#     def __init__(self, color):
#         # perform super's instructor
```

```
#         super().__init__(color)

#         # but in addition...
#         # decide the vectors that the piece can move now as it is based on
color
#         multiplier = 1 if color == "W" else -1

#         # method defined here as it only used here
#         # decided if the pawn is allowed to move foreward 2 based on square
contents and last move
#         def can_move_foreword_2(square):
#             return square is None and self.last_move is None

#         # tuple contains pairs of vector and contition that must be met
#         # (in the form of a function that takes square and returns a
boolean)
#         self.movement_vector_and_condition: tuple[Vector, Callable] = (
#             # v(0, 1) for foreword
#             (Vector(0, multiplier), lambda square:
self.square_contains(square) == "empty"),
#                 # v(0, 2) for foreword as first move
#                 (Vector(0, 2*multiplier), can_move_foreword_2),
#                 # v(-1, 1) and v(1, 1) for take
#                 (Vector(1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
#                     (Vector(-1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
#                 )

#         # generate movement vectors
#         def generate_movement_vectors(self, pieces_matrix, position_vector):
#             # iterate through movement vectors and conditions
#             for movement_vector, condition in
self.movement_vector_and_condition:
#                 # get resultant vector
#                 resultant_vector = position_vector + movement_vector
#                 # if vector_out of range continue
#                 if not resultant_vector.in_board():
#                     continue

#                 # get the contents of the square corresponding to the resultant
row, column = 7-resultant_vector.j, resultant_vector.i
#                 piece = pieces_matrix[row][column]

#                 # if the condition is met, yield the vector
#                 if condition(piece):
#                     yield movement_vector
```

```
# this class inherits from Piece an so it inherits some logic and some
requirements as to how its interface should be
# as many child classes are similar I will explain this one in depth and then
only explain notable features of others
class Pawn(Piece, Abstract_Piece_Interface):
    # defining abstract properties, needed before init
    _value = 100
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [50, 50, 50, 50, 50, 50, 50, 50],
        [10, 10, 20, 30, 30, 20, 10, 10],
        [5, 5, 10, 25, 25, 10, 5, 5],
        [0, 0, 0, 20, 20, 0, 0, 0],
        [5, -5, -10, 0, 0, -10, -5, 5],
        [5, 10, 10, -20, -20, 10, 10, 5],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]
    color = None

    # define symbol method (str method)
    def symbol(self): return f"{self.color}P"

    def does_square_contain_enemy(self, square):
        return self.square_contains(square) == "enemy"

    def does_square_contain_empty(self, square):
        return self.square_contains(square) == "empty"

    def can_move_foreword_2(self, to_square, middle_square):
        if self.last_move is not None:
            return False
        if not self.does_square_contain_empty(middle_square):
            return False
        if not self.does_square_contain_empty(to_square):
            return False

        return True
    # override init constructor
    def __init__(self, color):
        # perform super's instructor
        super().__init__(color)

        # but in addition...
        # decide the vectors that the piece can move now as it is based on
color
        multiplier = 1 if color == "W" else -1
```

```
# method defined here as it only used here
# decided if the pawn is allowed to move foreword 2 based on square
contents and last move

# def can_move_foreword_2(to_square, middle_square):
#     return (to_square is not None) and (middle_square is not None)
and (self.last_move is None)

# def does_square_contain_enemy(square):
#     return self.square_contains(square) == "enemy"

# def does_square_contain_empty(square):
#     return self.square_contains(square) == "empty"

# tuple contains pairs of vector and condition that must be met
# (in the form of a function that takes square and returns a boolean)
self.movement_vector_and_condition: tuple[Vector, Callable] = (
    # v(0, 1) for foreword
    # (Vector(0, multiplier), lambda square:
self.square_contains(square) == "empty"),
    (Vector(0, multiplier), self.does_square_contain_empty),

    # now fixed
    # v(0, 2) for foreword as first move
    (Vector(0, 2*multiplier), self.can_move_foreword_2),

    # v(-1, 1) and v(1, 1) for take
    # (Vector(1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
    (Vector(1, multiplier), self.does_square_contain_enemy),
    # (Vector(-1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
    (Vector(-1, multiplier), self.does_square_contain_enemy),
)

# generate movement vectors
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # iterate through movement vectors and conditions
    for movement_vector, condition in self.movement_vector_and_condition:
        # get resultant vector
        resultant_vector = position_vector + movement_vector
        # if vector_out of range continue
        if not resultant_vector.in_board():
            continue

        # get the contents of the square corresponding to the resultant
        row, column = 7-resultant_vector.j, resultant_vector.i
```

```

        to_square = pieces_matrix[row][column]

        # if the condition is met, yield the vector

        if movement_vector not in (Vector(0, 2), Vector(0, -2)):
            if condition(to_square):
                yield movement_vector
        else:
            # must be moving foreword 2
            middle_square_vector = position_vector + movement_vector *
(1/2)

            row, column = 7-middle_square_vector.j, middle_square_vector.i
            middle_square = pieces_matrix[row][column]

            if condition(to_square=to_square,
middle_square=middle_square):
                yield movement_vector


class Knight(Piece, Abstract_Piece_Interface):
    _value = 320
    _value_matrix: tuple[tuple[float]] = [
        [-50, -40, -30, -30, -30, -40, -50],
        [-40, -20, 0, 0, 0, 0, -20, -40],
        [-30, 0, 10, 15, 15, 10, 0, -30],
        [-30, 5, 15, 20, 20, 15, 5, -30],
        [-30, 0, 15, 20, 20, 15, 0, -30],
        [-30, 5, 10, 15, 15, 10, 5, -30],
        [-40, -20, 0, 5, 5, 0, -20, -40],
        [-50, -40, -30, -30, -30, -40, -50]
    ]
    color = None

    # n for knight as king takes k
    def symbol(self): return f"{self.color}N"

    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # this function yields all 8 possible vectors
        def possible_movement_vectors():
            vectors = (Vector(2, 1), Vector(1, 2))
            # for each x multiplier, y multiplier and vector combination
            for i_multiplier, j_multiplier, vector in iter_product((-1, 1), (-1, 1), vectors):
                # yield corresponding vector
                yield Vector(
                    vector.i * i_multiplier,

```

```
        vector.j * j_multiplier
    )
# iterate through movement vectors
for movement_vector in possible_movement_vectors():
    # get resultant
    resultant_vector = position_vector + movement_vector
    # look at contents of square
    contents =
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix)

    # if square is empty yield vector

    # # old code that caused the logic error
    # if contents == "empty":
    #     yield movement_vector

    # corrected code to fix knight bug
if contents in ("empty", "enemy"):
    yield movement_vector


class Bishop(Piece, Abstract_Piece_Interface):
    _value = 330
    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -10, -10, -10, -10, -20],
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 10, 10, 5, 0, -10],
        [-10, 5, 5, 10, 10, 5, 5, -10],
        [-10, 0, 10, 10, 10, 10, 0, -10],
        [-10, 10, 10, 10, 10, 10, 10, -10],
        [-10, 5, 0, 0, 0, 0, 5, -10],
        [-20, -10, -10, -10, -10, -10, -10, -20]
    ]
    color = None

    def symbol(self): return f"{self.color}B"

    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # sourcery skip: use-itertools-product
        # repeat for all 4 vector directions
        for i, j in iter_product((1, -1), (1, -1)):
            unit_vector = Vector(i, j)
```

```
# iterate through length multipliers
for multiplier in range(1, 8):
    # get movement and resultant vectors
    movement_vector = unit_vector * multiplier
    resultant_vector = position_vector + movement_vector

    # examine the contents of the square and use switch case to
decide behaviour
    match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
        case 'illegal':
            # if vector extends out of the board stop extending
            break
        case 'ally':
            # break of of for loop (not just match case)
            # as cannot hop over piece so don't explore longer
vectors in same direction
            break
        case 'enemy':
            # this is a valid move
            yield movement_vector
            # break of of for loop (not just match case)
            # as cannot hop over piece so don't explore longer
vectors in same direction
            break
        case 'empty':
            # is valid
            yield movement_vector
            # and keep exploring, don't break

class Rook(Piece, Abstract_Piece_Interface):
    _value = 500
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [5, 10, 10, 10, 10, 10, 10, 5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [0, 0, 0, 5, 5, 0, 0, 0]
    ]
    color = None

    # r for rook
    def symbol(self): return f"{self.color}R"
```

```
# this code is very similar in structure to that of a bishop just with
different direction vectors
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # sourcery skip: use-itertools-product
    unit_vectors = (
        Vector(0, 1),
        Vector(0, -1),
        Vector(1, 0),
        Vector(-1, 0),
    )
    # for unit_vector, multiplier in iter_product(unit_vectors, range(1,
8)):
        for unit_vector in unit_vectors:
            for multiplier in range(1, 8):
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector

                # note cases that contain only break are not redundant, they
break the outer for loop
                match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                    case 'illegal':
                        # if vector extends out of the board stop extending
                        break
                    case 'ally':
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'enemy':
                        # this is a valid move
                        yield movement_vector
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'empty':
                        # is valid
                        yield movement_vector
                        # and keep exploring, don't break

class Queen(Piece, Abstract_Piece_Interface):
    _value = 900
    _value_matrix: tuple[tuple[float]] = [
```

```

[-20, -10, -10, -5, -5, -10, -10, -20],
[-10, 0, 0, 0, 0, 0, 0, -10],
[-10, 0, 5, 5, 5, 5, 0, -10],
[-5, 0, 5, 5, 5, 5, 0, -5],
[0, 0, 5, 5, 5, 5, 0, -5],
[-10, 5, 5, 5, 5, 5, 0, -10],
[-10, 0, 5, 0, 0, 0, 0, -10],
[-20, -10, -10, -5, -5, -10, -10, -20]
]
color = None

def symbol(self): return f"{self.color}Q"

# this code also uses a similar structure to the rook or bishop
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1, 0, 1)) if i != 0 and j != 0)
    unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1, 0, 1)) if i != 0 or j != 0)

    # for unit_vector, multiplier in iter_product(unit_vectors, range(1, 8)):
    for unit_vector in unit_vectors:
        for multiplier in range(1, 8):
            movement_vector = unit_vector * multiplier
            resultant_vector = position_vector + movement_vector
            match
                case 'illegal':
                    # if vector extends out of the board stop extending
                    break
                case 'ally':
                    # break of of for loop (not just match case)
                    # as cannot hop over piece so don't explore longer
                    vectors in same direction
                    break
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                    # break of of for loop (not just match case)
                    # as cannot hop over piece so don't explore longer
                    vectors in same direction
                    break
                case 'empty':
                    # is valid
                    yield movement_vector
                    # and keep exploring, don't break

```

```
class King(Piece, Abstract_Piece_Interface):
    # not needed as static eval doesn't add up kings value
    _value = ARBITRARILY_LARGE_VALUE

    # there are 2 matrices to represent the early and late game for the king
    value_matrix_early: tuple[tuple[float]] = [
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-20, -30, -30, -40, -40, -30, -30, -20],
        [-10, -20, -20, -20, -20, -20, -20, -10],
        [20, 20, 0, 0, 0, 20, 20],
        [20, 30, 10, 0, 0, 10, 30, 20]
    ]
    value_matrix_late = [
        [-50, -40, -30, -20, -20, -30, -40, -50],
        [-30, -20, -10, 0, 0, -10, -20, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -30, 0, 0, 0, -30, -30],
        [-50, -30, -30, -30, -30, -30, -30, -50]
    ]
    color = None

    # initially value matrix is the early one
    _value_matrix: tuple[tuple[float]] = value_matrix_early

    # def __init__(self, *args, **kwargs):
    #     self._value_matrix = self.value_matrix_early
    #     super().__init__(self, *args, **kwargs)

    def symbol(self): return f"{self.color}K"

    # based on total pieces, changes the value matrix
    def update_value_matrix(self, pieces_matrix):
        # counts each empty square as 0 and each full one as 1 then sums them
        # up to get total pieces
        # if total pieces less than or equal to 10: then late game
        if sum(int(isinstance(square, Piece)) for square in
iter_chain(pieces_matrix)) <= 10:
            self.value_matrix = self.value_matrix_late
        # else early game
        else:
            self.value_matrix = self.value_matrix_early
```

```
# this generates the movement vectors for the king
def generate_movement_vectors(self, pieces_matrix, position_vector):
    # take the opportunity to update the value matrix
    self.update_value_matrix(pieces_matrix)

    # all 8 movement vectors
    unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1, 0, 1)) if i != 0 or j != 0)

    # for each movement vector, get the resultant vector
    for movement_vector in unit_vectors:
        resultant_vector = position_vector + movement_vector

        # examine contents of square and use switch case to decide
        behaviour
        match
            self.examine_position_vector(position_vector=resultant_vector,
            pieces_matrix=pieces_matrix):
                case 'illegal':
                    continue
                case 'ally':
                    continue
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                case 'empty':
                    # is valid
                    yield movement_vector

# used by other modules to convert symbol to piece
PIECE_TYPES = {
    'P': Pawn,
    'N': Knight,
    'B': Bishop,
    'R': Rook,
    'K': King,
    'Q': Queen
}

# if __name__ == '__main__':
# ensures that all classes are valid (not missing any abstract properties)
# whenever the module is imported
Pawn('W')
Knight('W')
Bishop('W')
Rook('W')
Queen('W')
```

```
King('W')
```

chess functions\board state.py

```
from copy import deepcopy
from itertools import product as iter_product
from random import choice as random_choice

from . import pieces as pieces_mod
from assorted import ARBITRARILY_LARGE_VALUE, safe_hash, cache_decorator
from .vector import Vector

# this is a pieces matrix for the starting position in chess
# white is at the bottom as it is from the user's perspective and I am
# currently assuming the user is white.
# I can change this in the frontend later

STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
    (
        pieces_mod.Rook(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Queen(color="B"),
        pieces_mod.King(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Rook(color="B")
    ),
    # (pieces_mod.Pawn(color="B"),)*8,
    tuple(pieces_mod.Pawn(color="B") for _ in range(8)),
    (None,)*8,
    (None,)*8,
    (None,)*8,
    (None,)*8,
    # (pieces_mod.Pawn(color="W"),)*8,
    tuple(pieces_mod.Pawn(color="W") for _ in range(8)),
    (
        pieces_mod.Rook(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Queen(color="W"),
        pieces_mod.King(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Knight(color="W"),
    )
)
```

```
        pieces_mod.Rook(color="W")
    )
)

# frozen makes each instance of the board_state class is immutable
# @dataclass(frozen=True)
class Board_State():
    # a table of game state hashes and there frequency
    pieces_matrix_frequency: tuple

    check_encountered: bool

    next_to_go: str
    # the pieces matrix keeps track of the board positions and the pieces
    pieces_matrix: tuple[tuple[pieces_mod.Piece]]


    def __init__(self, next_to_go="W", pieces_matrix=STARTING_POSITIONS,
pieces_matrix_frequency={}, check_encountered=None) -> None:
        self.next_to_go = next_to_go
        self.pieces_matrix = pieces_matrix
        self.pieces_matrix_frequency = pieces_matrix_frequency

        if check_encountered is None:
            check_encountered = self.color_in_check()
        self.check_encountered = check_encountered
        self.three_repeat_stalemates_enabled = True

    # this adds a new game state to the frequency table
    def new_piece_matrix_frequency_table(self):
        # the pieces matrix is hashed
        matrix_hash = hash(str(self.pieces_matrix))
        copy_pieces_matrix_frequency = self.pieces_matrix_frequency.copy()

        # if this pieces matrix has been encountered before, add 1 to the
frequency
        if matrix_hash in self.pieces_matrix_frequency.keys():
            copy_pieces_matrix_frequency[matrix_hash] += 1
        # else set frequency to 1
        else:
            copy_pieces_matrix_frequency[matrix_hash] = 1

    return copy_pieces_matrix_frequency

    # determines if there is a 3 repeat stalemate
    def is_3_board_repeats_in_game_history(self):
```

```
# if any of the board states have been repeated 3 or more times:  
stalemate  
    if not self.three_repeat_stalemates_enabled:  
        return False  
    return any(value >= 3 for value in  
self.pieces_matrix_frequency.values())  
  
@property  
def moves_counter(self):  
    return sum(self.pieces_matrix_frequency.values())  
  
@property  
def number_legal_moves(self):  
    return len(list(  
        self.generate_legal_moves()  
    ))  
  
@property  
def number_total_pieces(self):  
    return len(list(  
        self.generate_all_pieces()  
    ))  
  
# @property  
def generate_pieces_taken_by_color(self, color):  
    # https://stackoverflow.com/questions/8106227/difference-between-two-lists-with-duplicates-in-python  
    all_remaining_pieces = list(map(  
        lambda piece_and_vector: piece_and_vector[0],  
        self.generate_all_pieces(),  
    ))  
  
    # cannot be a taken king  
    starting_quantities = {  
        pieces_mod.Pawn(color): 8,  
        pieces_mod.Rook(color): 2,  
        pieces_mod.Bishop(color): 2,  
        pieces_mod.Knight(color): 2,  
        pieces_mod.Queen(color): 1,  
    }  
  
    for piece, starting_count in starting_quantities.items():  
        missing = starting_count - all_remaining_pieces.count(piece)  
        # yield from [str(piece)]*missing  
        yield from [piece]*missing  
  
    # this function outputs the board in a user friendly way
```

```

# 8| BR BN BB BQ BK BB BN BR
# 7| . . BP BP BP BP BP BP
# 6| . . . . . . . .
# 5| BP BP . . . . . .
# 4| . . . WP . WP . .
# 3| . . . . . . . .
# 2| WP WP WP . WP . WP WP
# 1| WR WN WB WQ WK WB WN WR
#   (A B C D E F G H )

def print_board(self):
    # convert each piece to a symbol BP and replace none with dots
    # add numbers to left and add letters at the bottom
    numbers = range(8, 0, -1)
    letters = [chr(i) for i in range(ord("A"), ord("H")+1)]
    for row, number in zip(self.pieces_matrix, numbers):
        # clean up row of pieces
        symbols_row = map(lambda piece: piece.symbol() if piece else ".",
row)
        pretty_row = f"{number}| {' '.join(symbols_row)}"
        print(pretty_row)
    print(f" ({' '.join(letters)})")

    # print()
    # pieces_taken = list(self.pieces_taken)
    # if len(pieces_taken) > 0:
    #     print("Pieces taken:")
    #     print(", ".join(
    #         pieces_taken
    #     ))
    # else:
    #     print("No pieces taken")

@cache_decorator
def get_piece_at_vector(self, vector: Vector):
    # this function exists as it is a really common operation.
    # due to vector 0, 0 pointing the the bottom left not top left of the
2d array,
    # some correction is needed
    column, row = vector.i, 7-vector.j
    return self.pieces_matrix[row][column]

# this function should yield all the pieces on the board

def generate_all_pieces(self):
    # nested loop for i and j to iterate through all possible vectors
    for i, j in iter_product(range(8), range(8)):
        position_vector = Vector(i,j)

```

```
# get the contents of the corresponding square
piece = self.get_piece_at_vector(position_vector)

# skip if none: skip if square empty
if piece:
    yield piece, position_vector

# this yields all pieces of a given color:
# used when examining legal moves of a given player

# causes minimax test to fail
## @CACHE_DECORATOR
def generate_pieces_of_color(self, color=None):
    # be default give pieces of player next to go
    if color is None:
        color = self.next_to_go

    # return all piece and position vector pairs
    # filtered by piece must match in color
    yield from filter(
        # lambda piece, _: piece.color == color,
        lambda piece_and_vector: piece_and_vector[0].color == color,
        self.generate_all_pieces()
    )

# determines if a given player is in check based on the player's color
@cache_decorator
def color_in_check(self, color=None):
    # default is to check if the player next to go is in check
    if color is None:
        color = self.next_to_go

    # let is now be A's turn
    # I use this player a and b model to keep track of the logic here
    color_a = color
    color_b = "W" if color == "B" else "B"

    # we will examine all the movement vectors of B's pieces
    # if any of them could take the A's King then currently A is in check
as their king is threatened by 1 or more pieces (which could take it next
turn)
    # for each of b's pieces
    for piece, position_v in self.generate_pieces_of_color(color=color_b):
        movement_vs = piece.generate_movement_vectors(
            pieces_matrix=self.pieces_matrix,
            position_vector=position_v
        )
        # for each movement vector that the piece could make
```

```

        for movement_v in movement_vs:
            resultant = position_v + movement_v
            # check the contents of the square
            to_square = self.get_piece_at_vector(resultant)
            # As_move_threatens_king_A = isinstance(to_square,
pieces_mod.King) and to_square.color == color_a
                # if the contents is A's king then the a is in check.
                As_move_threatens_king_A = (to_square ==
pieces_mod.King(color=color_a))
                    # if As_move_threatens_king_A then return true
                    if As_move_threatens_king_A:
                        return True
            # if none of B's pieces were threatening to take A's king then A isn't
in check
            return False

        # this function is a generator to be iterated through.
        # it is responsible for yielding every possible move that a given player
can make
        # yields this as a position and a movement vector

        # causes minimax test to fail
        ## @CACHE_DECORATOR
        def generate_legal_moves(self):
            # iterate through all pieces belonging to player next to go
            for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
                movement_vectors = piece.generate_movement_vectors(
                    pieces_matrix=self.pieces_matrix,
                    position_vector=piece_position_vector
                )
                # iterate through movement vectors of this piece
                for movement_vector in movement_vectors:
                    # examine resulting child game state
                    child_game_state =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
                        # determine if current player next to go (different to next to
go of child game state) is in check
                        is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
                            # only yield the move if it doesn't result in check
                            if not is_check_after_move:
                                yield piece_position_vector, movement_vector

        # determines if the game is over
        # returns over, winner
        @cache_decorator

```

```

def is_game_over_for_next_to_go(self):
    # sourcery skip: remove-unnecessary-else, swap-if-else-branches
    # check for 3 repeat stalemate
    if self.is_3_board_repeats_in_game_history():
        return True, None

    # in all cases, the game is over if a player has no legal moves left
    if not list(self.generate_legal_moves()):
        # if b in check
        if self.color_in_check():
            # checkmate for b, a wins
            winner = "W" if self.next_to_go == "B" else "B"
            return True, winner
        else:
            # stalemate
            return True, None
    # game not over
    return False, None

# this function is responsible for generating a static evaluation for a
given board-state
# it should be used by a maximiser or minimiser
# starting position should have an evaluation of 0
@cache_decorator
def static_evaluation(self):
    # if over give an appropriate score for win loss or draw
    over, winner = self.is_game_over_for_next_to_go()
    if over:
        match winner:
            case None: multiplier = 0
            case "W": multiplier = 1
            case "B": multiplier = -1
        # return winner * ARBITRARILY_LARGE_VALUE
        return multiplier * ARBITRARILY_LARGE_VALUE

    # this function takes a piece as an argument and uses its color to
    # decide if its value should be positive or negative
    def get_piece_value(piece: pieces_mod.Piece, position_vector: Vector):
        # this function assumes white is maximizer and so white pieces
        # have a positive score and black negative
        match piece.color:
            case "W": multiplier = 1
            case "B": multiplier = (-1)
        value = multiplier * piece.get_value(position_vector)
        # print(f"{piece.symbol()} at {position_vector.to_square} has
        # value {value}")
        return value

```

```
# for each piece, get the value (+/-)
values = map(
    lambda x: get_piece_value(*x),
    self.generate_all_pieces()
)
# sum values for static eval
return sum(values)

@cache_decorator
def make_move(self, from_position_vector: Vector, movement_vector: Vector):
    resultant_vector = from_position_vector + movement_vector

    # make a copy of the position vector, deep copy is used to ensure no
    parts are shared be reference
    new_pieces_matrix = deepcopy(self.pieces_matrix)

    # convert to list
    new_pieces_matrix = list(map(list, new_pieces_matrix))

    # look at square with position vector
    row, col = 7-from_position_vector.j, from_position_vector.i
    # get piece thants moving
    piece: pieces_mod.Piece = new_pieces_matrix[row][col]
    # set from square to blank
    new_pieces_matrix[row][col] = None

    # update piece to keep track of its last move
    # print(f"Setting piece {piece!r} at {from_position_vector!r} to have
    a last move property of {movement_vector!r}")
    piece.last_move = movement_vector

    if piece == pieces_mod.Pawn("W") and resultant_vector.j == 7:
        piece = pieces_mod.Queen("W")

    if piece == pieces_mod.Pawn("B") and resultant_vector.j == 0:
        piece = pieces_mod.Queen("B")

    # set to square to this piece
    row, col = 7-resultant_vector.j, resultant_vector.i
    new_pieces_matrix[row][col] = piece

    # convert back to tuple
    new_pieces_matrix = tuple(map(tuple, new_pieces_matrix))

    # update next to go
    new_next_to_go = "W" if self.next_to_go == "B" else "B"
```

```
new_pieces_matrix_frequency = self.new_piece_matrix_frequency_table()

# return new board state instance
return Board_State(
    next_to_go=new_next_to_go,
    pieces_matrix=new_pieces_matrix,
    pieces_matrix_frequency = new_pieces_matrix_frequency,
    check_encountered=self.check_encountered
)

@cache_decorator
# this function created a string to represent this object
def __repr__(self) -> str:
    return f"Board_State(pieces_matrix={self.pieces_matrix!r},
next_to_go='{self.next_to_go}', pieces_matrix_frequency={self.pieces_matrix_frequency})"

# this function encodes MOST of the information that makes this object
unique in a number.
# this doesn't include move history, this should make the cache more
useful at the cost of making the program worse at avoiding 3 repeat stalemated
def database_hash(self):
    return safe_hash(
        (
            "Board_State",
            self.pieces_matrix,
            self.next_to_go,
            # consider board states the same even if they have a different
history?
            # self.pieces_matrix_frequency
        )
    )

def __hash__(self) -> int:
    return hash(self.database_hash())

# this function produces a random board state
# this is done by selecting and then implementing a legal move at random up to
so many moves
# it is used in unittest
def random_board_state(moves: int):
    board_state = Board_State()
    # keep trying to get a non over board state
    while True:
        try:
```

```

        # iteratively pick a move at random and implement it until the
right number of move is reached
        for _ in range(moves):
            legal_moves = list(board_state.generate_legal_moves())
            assert len(legal_moves) > 0
            random_move = random_choice(legal_moves)
            board_state = board_state.make_move(*random_move)

    except AssertionError:
        # if no legal moves (over) then try again
        continue
    else:
        return board_state

```

chess functions\test fast vector.py

```

import ddt
import unittest

from .vector import Vector


# function for path to vector related test data
def test_path(file_name):
    return f"./test_data/vector/{file_name}.yaml"

# augment my test case class with ddt decorator
@ddt.ddt
class Test_Case(unittest.TestCase):

    # performs many checks of construct from square
    @ddt.file_data(test_path("from_square"))
    def test_square_to_vector(self, square, expected_vector):
        self.assertEqual(
            Vector.construct_from_square(square),
            Vector(*expected_vector),
            msg=f"\nVector.construct_from_square('{square}') != Vector(i={expected_vector[0]}, j={expected_vector[1]})"
        )

    # performs many checks of adding vectors
    @ddt.file_data(test_path("vector_add"))
    def test_add_vectors(self, vector_1, vector_2, expected_vector):
        self.assertEqual(
            Vector(*vector_1) + Vector(*vector_2),
            Vector(*expected_vector),

```

```

msg=f"\nVector(*{vector_1}) +
Vector(*{vector_2}) != Vector(*{expected_vector})"
)

# performs many checks of multiplying vectors
@ddt.file_data(test_path("vector_multiply"))
def test_multiply_vectors(self, vector, multiplier, expected):
    self.assertEqual(
        Vector(*vector) * multiplier,
        Vector(*expected)
    )

# many tests of vector in board
@ddt.file_data(test_path("vector_in_board"))
def test_in_board(self, vector, expected):
    self.assertEqual(
        Vector(*vector).in_board(),
        expected,
        msg=f"\nVector(*{vector}).in_board() != {expected}"
    )

# many tests of vector out of board
@ddt.file_data(test_path('vector_to_square'))
def test_to_square(self, vector, expected):
    self.assertEqual(
        Vector(*vector).to_square(),
        expected
    )

if __name__ == '__main__':
    unittest.main()
chess functions\test_fast_pieces.py

```

```

import unittest
import ddt

from . import pieces
# as vector is already tested we can use it here and assume it won't cause any
logic errors
from .vector import Vector

def test_dir(file_name): return f"test_data/pieces/{file_name}.yaml"

EMPTY_PIECES_MATRIX = ((None,)*8,)*8

@ddt.ddt
class Test_Case(unittest.TestCase):

```

```

# this test is based on testing the movement vectors of a piece places
at some position in an empty board are as expected
@ddt.file_data(test_dir('test_empty_board'))
def test_empty_board(self, piece_type, square, expected_move_squares):
    # deserialize
    position_vector = Vector.construct_from_square(square)
    piece: pieces.Piece = pieces.PIECE_TYPES[piece_type]('W')
    movement_vectors =
piece.generate_movement_vectors(pieces_matrix=EMPTY_PIECES_MATRIX,
position_vector=position_vector)
    resultant_squares = list(
        map(
            lambda movement_vector:
(movement_vector+position_vector).to_square(),
            movement_vectors
        )
    )
    # assert as expected
    # can use sets to prevent order being an issue as vectors are hashable
    self.assertEqual(
        set(resultant_squares),
        set(expected_move_squares),
        msg=f"\n\nactual movement squares
{sorted(resultant_squares)} != expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
    )

# this test assert that that a pieces movement vectors are as expected
when the piece is surrounded by other pieces
@ddt.file_data(test_dir('test_board_populated'))
def test_board_populated(self, pieces_matrix, square,
expected_piece_symbol, expected_move_squares):
    # deserialize
    def list_map(function, iterable): return list(map(function, iterable))
    def tuple_map(function, iterable): return tuple(map(function,
iterable))

    def descriptor_to_piece(descriptor) -> pieces.Piece:
        # converts WN to knight object with a color attribute of white
        if descriptor is None:
            return None

        color, symbol = descriptor
        piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
        return piece_type(color=color)

    def row_of_symbols_to_pieces(row): return
tuple_map(descriptor_to_piece, row)

```

```

# update pieces_matrix replacing piece descriptors to piece objects
pieces_matrix = tuple_map(row_of_symbols_to_pieces, pieces_matrix)

position_vector = Vector.construct_from_square(square)
row, column = 7-position_vector.j, position_vector.i

# assert piece as expected
piece: pieces.Piece = pieces_matrix[row][column]
self.assertEqual(
    piece.symbol(),
    expected_piece_symbol,
    msg=f"\nPiece at square {square} was not the expected piece"
)

# assert movement vectors as expected
movement_vectors =
piece.generate_movement_vectors(pieces_matrix=pieces_matrix,
position_vector=position_vector)
resultant_squares = list(
    map(
        lambda movement_vector:
(movement_vector+position_vector).to_square(),
        movement_vectors
    )
)
self.assertEqual(
    set(resultant_squares),
    set(expected_move_squares),
    msg=f"\n\nactual movement squares
{sorted(resultant_squares)} != expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
)

if __name__ == '__main__':
    unittest.main()

```

chess functions\test fast board state.py

```

import unittest
import ddt
from random import randint
import json

from .board_state import Board_State, random_board_state
from .pieces import Pawn

```

```
# tested so assumed correct
from . import pieces
from .vector import Vector

from schemas import deserialize_pieces_matrix as
deserialize_client_pieces_matrix

def test_dir(file_name): return f"test_data/board_state/{file_name}.yaml"

# code used to deserialize
# code repeated from test_pieces, opportunity to reduce redundancy
def list_map(function, iterable): return list(map(function, iterable))
def tuple_map(function, iterable): return tuple(map(function, iterable))

def descriptor_to_piece(descriptor) -> pieces.Piece:
    # converts WN to knight object with a color attribute of white
    if descriptor is None:
        return None

    color, symbol = descriptor
    piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
    return piece_type(color=color)

def deserialize_pieces_matrix(pieces_matrix, next_to_go="W") -> Board_State:
    def row_of_symbols_to_pieces(row):
        return tuple_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = tuple_map(row_of_symbols_to_pieces, pieces_matrix)
    board_state: Board_State = Board_State(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)

    return board_state

@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test isn't data driven,
    # it tests that the static evaluation is 0 for starting positions
    def test_static_eval_starting_positions(self):
        self.assertEqual(
            Board_State().static_evaluation(),
            0
        )

    def test_for_specific_issue(self):
        # print("\nRunning test test_for_specific_issue")
```

```
board_state = Board_State()

piece: Pawn = board_state.get_piece_at_vector(Vector(1, 1))
self.assertTrue(
    piece.last_move is None,
    f"Piece hasn't been moved yet so last_move property should be
None\nInstead it is {piece.last_move}"
)

board_state = board_state.make_move(Vector(0, 1), Vector(0, 2))
board_state = board_state.make_move(Vector(0, 6), Vector(0, -2))

piece: Pawn = board_state.get_piece_at_vector(Vector(1, 1))
self.assertTrue(
    piece.last_move is None,
    f"Piece hasn't been moved yet so last_move property should be
None\nInstead it is {piece.last_move}"
)

movement_vectors = list(piece.generate_movement_vectors(
    position_vector=Vector(1, 1),
    pieces_matrix=board_state.pieces_matrix,
))
# movement_vectors = sorted(movement_vectors, key=repr)

# print("\n")
# board_state.print_board()
# print("\n")

# print({"movement_vectors": movement_vectors})
self.assertTrue(
    Vector(0, 1) in movement_vectors and Vector(0, 2) in
movement_vectors,
    "Piece should be able to move foreword by either 1 or 2 squares"
)

actual_legal_moves = list(board_state.generate_legal_moves())
actual_legal_moves = list(map(list, actual_legal_moves))

self.assertTrue(
    [Vector(1, 1), Vector(0, 2)] in actual_legal_moves
)

# expected_legal_moves = [
#     [Vector(1, 1), Vector(0, 2)],
#     [Vector(2, 1), Vector(0, 2)],
#     [Vector(3, 1), Vector(0, 2)],
```

```
#      [Vector(4, 1), Vector(0, 2)],
#      [Vector(5, 1), Vector(0, 2)],
#      [Vector(6, 1), Vector(0, 2)],
#      [Vector(7, 1), Vector(0, 2)],
#      [Vector(1, 6), Vector(0, -2)],
#      [Vector(2, 6), Vector(0, -2)],
#      [Vector(3, 6), Vector(0, -2)],
#      [Vector(4, 6), Vector(0, -2)],
#      [Vector(5, 6), Vector(0, -2)],
#      [Vector(6, 6), Vector(0, -2)],
#      [Vector(7, 6), Vector(0, -2)],
#      [Vector(1, 1), Vector(0, 1)],
#      [Vector(2, 1), Vector(0, 1)],
#      [Vector(3, 1), Vector(0, 1)],
#      [Vector(4, 1), Vector(0, 1)],
#      [Vector(5, 1), Vector(0, 1)],
#      [Vector(6, 1), Vector(0, 1)],
#      [Vector(7, 1), Vector(0, 1)],
#      [Vector(1, 6), Vector(0, -1)],
#      [Vector(2, 6), Vector(0, -1)],
#      [Vector(3, 6), Vector(0, -1)],
#      [Vector(4, 6), Vector(0, -1)],
#      [Vector(5, 6), Vector(0, -1)],
#      [Vector(6, 6), Vector(0, -1)],
#      [Vector(7, 6), Vector(0, -1)],
#      [Vector(0, 3), Vector(0, 1)],
#      [Vector(0, 4), Vector(0, -1)],
#      [Vector(1, 0), Vector(1, 2)],
#      [Vector(1, 0), Vector(-1, 2)],
#      [Vector(6, 0), Vector(1, 2)],
#      [Vector(6, 0), Vector(-1, 2)],
#      [Vector(1, 7), Vector(1, -2)],
#      [Vector(1, 7), Vector(-1, -2)],
#      [Vector(6, 7), Vector(1, -2)],
#      [Vector(6, 7), Vector(-1, -2)],
#      [Vector(0, 0), Vector(0, 1)],
#      [Vector(0, 0), Vector(0, 2)],
#      [Vector(7, 0), Vector(0, 1)],
#      [Vector(7, 0), Vector(0, 2)],
#      [Vector(0, 7), Vector(0, -1)],
#      [Vector(0, 7), Vector(0, -2)],
#      [Vector(7, 7), Vector(0, -1)],
#      [Vector(7, 7), Vector(0, -2)],
#  ]
# expected_legal_moves = list(map(list, expected_legal_moves))
```

```

        # actual_legal_moves = sorted(actual_legal_moves, key=lambda x:
repr(x))
        # expected_legal_moves = sorted(expected_legal_moves, key=lambda x:
repr(x))

        # print("\n")
        # print({"expected_legal_moves": expected_legal_moves})
        # print({"actual_legal_moves": actual_legal_moves})
        # self.assertEqual(
        #     actual_legal_moves,
        #     expected_legal_moves,
        #     "The actual legal moves are not the same as the expected legal
moves"
        # )

# this test was written as I was confused about the source of a bug
# I thought some move resultant vectors were some how outside the board
# instead I was using a negative movement vector
def test_legal_moves_in_board(self):
    # print("\nRunning test_legal_moves_in_board")
    def gen_random_board_states():
        # multiplier = 10
        multiplier = 3
        for _ in range(4*multiplier):
            yield random_board_state(moves=randint(0, 10))
        for _ in range(2*multiplier):
            yield random_board_state(moves=randint(0, 20))
        for _ in range(multiplier):
            yield random_board_state(moves=randint(0, 40))

        for board_state in gen_random_board_states():
            for to_v, movement_v in board_state.generate_legal_moves():
                resultant_v: Vector = to_v + movement_v
                self.assertTrue(
                    resultant_v.in_board(),
                    "All resultant vectors from legal moves should be in
board"
                )
    )

# this test is testing that a list of all pieces and there position
vectors can be generated
@ddt.file_data(test_dir('generate_all_pieces'))
def test_generate_all_pieces(self, pieces_matrix, pieces_and_squares):
    pieces_and_squares = tuple_map(tuple, pieces_and_squares)

    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)

```

```
# use set as order irrelevant
all_pieces: set[pieces.Piece, Vector] =
set(board_state.generate_all_pieces())


# convert square to vector, allowed as this is tested
def deserialize(data_unit):
    # unpack test data unit
    descriptor, square = data_unit
    # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
    return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

def serialize(data_unit):
    piece, vector = data_unit
    # return [piece.symbol(), vector.to_square()]
    return (piece.symbol(), vector.to_square())


all_pieces_expected: set[pieces.Piece, Vector] = set(map(deserialize,
pieces_and_squares))
# legal_moves_expected: list[pieces.Piece, Vector] = sorted(
#     map(deserialize, pieces_and_squares),
#     key=repr
# )

self.assertEqual(
    all_pieces,
    all_pieces_expected,
    msg=f"\nactual {list_map(serialize, all_pieces)} != expected
{list_map(serialize, all_pieces_expected)}"
)

# this test is to test the function responsible for getting the piece at a
given position vector
@ddt.file_data(test_dir('piece_at_vector'))
def test_piece_at_vector(self, pieces_matrix, vectors_and_expected_piece):
    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)

    # could use all method and one assert but this would have been less
    readable, also hard to make useful message,
    # wanting different messages implies multiple asserts should be
completed
    for vector, expected_piece in vectors_and_expected_piece:
        # deserialize vector / cast to Vector
        vector: Vector = Vector(*vector)
        # repeat for piece
        expected_piece: pieces.Piece = descriptor_to_piece(expected_piece)
```

```
# actual
piece: piece.Piece = board_state.get_piece_at_vector(vector)
msg = f"Piece at vector {repr(vector)} is {repr(piece)} not
expected piece {repr(expected_piece)}"

# this test ensures that all the pieces belonging to a specific color and
there position vectors can be identified
@ddt.file_data(test_dir('generate_pieces_of_color'))
def test_generate_pieces_of_color(self, pieces_matrix, color,
pieces_and_squares):
    pieces_and_squares = tuple_map(tuple, pieces_and_squares)

    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)
    # use set as order irrelevant
    legal_moves_actual: set[pieces.Piece, Vector] =
set(board_state.generate_pieces_of_color(color))

    # convert square to vector, allowed as this is tested
    def deserialize(data_unit):
        # unpack test data unit
        descriptor, square = data_unit
        # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
        return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

    def serialize(data_unit):
        piece, vector = data_unit
        # return [piece.symbol(), vector.to_square()]
        return (piece.symbol(), vector.to_square())

    legal_moves_expected: set[pieces.Piece, Vector] = set(map(deserialize,
pieces_and_squares))

    self.assertEqual(
        legal_moves_actual,
        legal_moves_expected,
        msg=f"\nactual {list_map(serialize,
legal_moves_actual)} != expected {list_map(serialize,
legal_moves_expected)}"
    )

    # this test ensures that the chess engine can determine if a specified
player is currently in check
@ddt.file_data(test_dir('color_in_check'))
```

```

def test_color_in_check(self, pieces_matrix, white_in_check,
black_in_check):
    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)

    self.assertEqual(
        board_state.color_in_check("W"),
        white_in_check,
        msg=f"white {'should' if white_in_check else 'should not'} be in
check but {'is' if board_state.color_in_check('W') else 'is not'}"
    )
    self.assertEqual(
        board_state.color_in_check("B"),
        black_in_check,
        msg=f"black {'should' if black_in_check else 'should not'} be in
check but {'is' if board_state.color_in_check('B') else 'is not'}"
    )

# this test ensures that a game over situation can be identified and its
nature discerned
@ddt.file_data(test_dir("game_over"))
def test_game_over(self, pieces_matrix, expected_over, expected_outcome,
next_to_go):
    board_state: Board_State = deserialize_pieces_matrix(pieces_matrix,
next_to_go=next_to_go)

    self.assertEqual(
        board_state.is_game_over_for_next_to_go(),
        (expected_over, expected_outcome),
        msg=f"\nactual {board_state.is_game_over_for_next_to_go()} !=\nexpected {(expected_over, expected_outcome)}"
    )

# this is one of the most important functions to test
# this function determines all the possible legal moves a player can make
with their pieces, accounting for check
# this test checks this for many inputs
@ddt.file_data(test_dir("generate_legal_moves"))
def test_generate_legal_moves(self, pieces_matrix, next_to_go,
expected_legal_moves):
    board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix=pieces_matrix, next_to_go=next_to_go)

    # print("board_state.get_piece_at_vector(Vector(6, 2))")
    # print(board_state.get_piece_at_vector(Vector(6, 2)))

    # print(Vector(6, 2).to_square())

    actual_legal_moves = set(board_state.generate_legal_moves())

```

```
# convert square to vector, allowed as this is tested
def deserialize_expected_legal_moves():
    # unpack test data unit
    for test_datum in expected_legal_moves:
        from_square, all_to_squares = test_datum
        for to_square in all_to_squares:
            position_vector: Vector =
                Vector.construct_from_square(from_square)
            movement_vector: Vector =
                Vector.construct_from_square(to_square) - position_vector

            yield (position_vector, movement_vector)

expected_legal_moves = set(deserialize_expected_legal_moves())

# print({"actual_legal_moves": actual_legal_moves})
# print({"expected_legal_moves": expected_legal_moves})

self.assertEqual(
    actual_legal_moves,
    expected_legal_moves,
    "\nActual (first) != Expected (second)"
)

# this tests for a bug I encountered where a knight couldn't take a piece,
# I created this test to identify the bug
# I then modified the knight pieces class and used this test to show that
the bug was fixed
def test_troubleshoot_bug_legal_moves(self):
    # replicate the previous situation
    board_state = Board_State()
    board_state = board_state.make_move(
        Vector(1, 0), Vector(1, 2)
    )
    board_state = board_state.make_move(
        Vector(1, 7), Vector(1, -2)
    )
    board_state = board_state.make_move(
        Vector(6, 0), Vector(-1, 2)
    )
    board_state = board_state.make_move(
        Vector(6, 7), Vector(-1, -2)
    )
    board_state = board_state.make_move(
        Vector(4, 1), Vector(0, 1)
    )
    board_state = board_state.make_move(
```



```
legal_moves = list(board_state.generate_legal_moves())

# print({"legal_moves": legal_moves})

self.assertEqual(
    legal_moves,
    [(Vector(i=1, j=7), Vector(i=1, j=0))]
)

over, _ = board_state.is_game_over_for_next_to_go()
# print(over)

self.assertTrue(not over)

if __name__ == '__main__':
    unittest.main()
```

chess functions\test_data\vector\from_square.yaml

```
test1:
    square: 'G3'
    expected_vector: [6, 2]

test2:
    square: 'A8'
    expected_vector: [0, 7]

test3:
    square: 'H1'
    expected_vector: [7, 0]

test4:
    square: 'C4'
    expected_vector: [2, 3]

test5:
    square: 'F6'
    expected_vector: [5, 5]

# # invalid data to check the tests work
# test6:
#     square: 'A5'
#     expected_vector: [5, 5]
```

chess functions\test_data\vector\vector_add.yaml

```
test1:  
  vector_1: [1, 4]  
  vector_2: [4, 6]  
  expected_vector: [5, 10]  
  
test2:  
  vector_1: [1, 7]  
  vector_2: [4, 6]  
  expected_vector: [5, 13]  
  
test3:  
  vector_1: [1, 3]  
  vector_2: [0, 6]  
  expected_vector: [1, 9]  
  
test4:  
  vector_1: [-1, 32]  
  vector_2: [3, 6]  
  expected_vector: [2, 38]  
  
test5:  
  vector_1: [6, 0]  
  vector_2: [6, 7]  
  expected_vector: [12, 7]  
  
# # invalid delete me  
# test6:  
#   vector_1: [6, 0]  
#   vector_2: [6, 7]  
#   expected_vector: [0, 7]
```

chess functions\test_data\vector\vector_in_board.yaml

```
test1:  
  vector: [0, 0]  
  expected: True  
  
test2:
```

```
vector: [7, 0]
expected: True
test3:
vector: [0, 7]
expected: True
test4:
vector: [7, 7]
expected: True

test5:
vector: [4, 6]
expected: True

test6:
vector: [3, 2]
expected: True

test7:
vector: [-1, -1]
expected: False

test8:
vector: [-5, 4]
expected: False

test9:
vector: [8, 7]
expected: False

test10:
vector: [-4, 4]
expected: False

test11:
vector: [10, 4]
expected: False

# # adding comment inside causes logic error and false pass on test
# # invalid delete me
# test:
#   vector: [4, 6]
#   expected: False
```

[chess functions\test_data\vector\vector_multiply.yaml](#)

test1:

```
vector: [1, 1]
multiplier: 1
expected: [1, 1]
test2:
vector: [1, 1]
multiplier: -1
expected: [-1, -1]
test3:
vector: [1, 1]
multiplier: 5
expected: [5, 5]
test4:
vector: [5, 0]
multiplier: 0
expected: [0, 0]
```

chess functions\test data\vector\vector to square.yaml

```
test1:
vector: [0, 0]
expected: A1
test2:
vector: [5, 7]
expected: F8
test3:
vector: [6, 2]
expected: G3
test4:
vector: [0, 6]
expected: A7
test5:
vector: [7, 7]
expected: H8
```

chess functions\test data\pieces\test board populated.yaml

```
test1:
pieces_matrix: [
[null, null, null, null, null, null, null, null],
[BP, null, BP, null, null, null, null, null],
```

```
[null, WP, null, null, null, null, null, null]
]
square: B1
expected_piece_symbol: WP
expected_move_squares: [
    A2, C2, B2, B3
]
test2:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BQ, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D4
expected_piece_symbol: BQ
expected_move_squares: [
    D5,
    E4, F4, G4,
    E3, F2, G1,
    D3, D2, D1,
    C4,
    C5, B6, A7
]
test3:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D4
expected_piece_symbol: BR
expected_move_squares: [
    D5,
    E4, F4, G4,
    D3, D2, D1,
    C4,
]
test4:
```

```
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, WP, null, WP, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D7
expected_piece_symbol: BP
expected_move_squares: [
    D6, D5,
    C6, E6
]
test5:
pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
square: D8
expected_piece_symbol: BK
expected_move_squares: [
    C7, D7, E7,
    C8, E8
]
test6:
pieces_matrix: [
    [BR, BK, BB, BK, BQ, BB, BK, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WN, WB, WK, WQ, WB, WN, WR ]
]
square: B1
expected_piece_symbol: WN
expected_move_squares: [A3, C3]
test7:
```

```

pieces_matrix: [
    [BR,    BN,    BB,    BK,    BQ,    BB,    BN,    BR  ],
    [BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP  ],
    [null, null, null, null, null, null, null, null],
    [WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP  ],
    [WR,    WN,    WB,    WK,    WQ,    WB,    WN,    WR  ]
]
square: G1
expected_piece_symbol: WN
expected_move_squares: [F3, H3]

```

chess functions\test data\pieces\test empty board.yaml

```

test1:
  piece_type: 'N'
  square: E4
  expected_move_squares: [
    D2, F2, D6, F6, C5, C3, G5, G3
  ]
test2:
  piece_type: Q
  square: C3
  expected_move_squares: [
    A1, B2, D4, E5, F6, G7, H8,
    C1, C2, C4, C5, C6, C7, C8,
    A3, B3, D3, E3, F3, G3, H3,
    A5, B4, D2, E1
  ]
test3:
  piece_type: K
  square: B2
  expected_move_squares: [
    C1, C2, C3,
    B1, B3,
    A1, A2, A3,
  ]
test4:
  piece_type: P
  square: E3
  expected_move_squares: [
    E4, E5
  ]
test5:
  piece_type: R

```

```

square: F6
expected_move_squares: [
    F1, F2, F3, F4, F5, F7, F8,
    A6, B6, C6, D6, E6, G6, H6
]
test6:
piece_type: B
square: D7
expected_move_squares: [
    C8, E6, F5, G4, H3,
    E8, C6, B5, A4,
]
test7:
piece_type: K
square: D8
expected_move_squares: [
    C7, D7, E7,
    C8, E8
]

```

chess functions\test_data\board state\color in check.yaml

```

test1:
pieces_matrix: [
    [BR, BK, BB, BK, BQ, BB, BK, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],
    [null, null, null, null, null, null, null, null],
    [WP, WP, WP, WP, WP, WP, WP, WP ],
    [WR, WK, WB, WK, WQ, WB, WK, WR ]
]
white_in_check: false
black_in_check: false
test2:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, WK, WQ, BK],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true

```

```
test3:  
  pieces_matrix: [  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null, BQ],  
    [null, null, null, null, null, null, null, null, null],  
    [null, null, null, WP, WP, null, null, null, null],  
    [null, null, null, WB, WK, WB, null, null, null],  
  ]  
  white_in_check: true  
  black_in_check: false  
  
test4:  
  pieces_matrix: [  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, WK, null, BK],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, WR]  
  ]  
  white_in_check: false  
  black_in_check: true  
  
test5:  
  pieces_matrix: [  
    [null, null, null, BK, null, null, null, null],  
    [null, null, null, WP, null, null, null, null],  
    [null, null, null, WK, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
  ]  
  white_in_check: false  
  black_in_check: false  
  
test6:  
  pieces_matrix: [  
    [null, null, BK, null, null, null, null, null],  
    [null, null, null, WP, null, null, null, null],  
    [null, null, null, WK, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
    [null, null, null, null, null, null, null, null],  
  ]
```

```
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true
test7:
pieces_matrix: [
    [null, null, null, null, BK, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: false
black_in_check: true
test8:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, WP, BK, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: true
black_in_check: true
test9:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, BK, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
white_in_check: true
black_in_check: true
```

chess functions\test_data\board state\game_over.yaml

```
test1:
    next_to_go: W
    pieces_matrix: [
        [BR,     BK,     BB,     BK,     BQ,     BB,     BK,     BR  ],
        [BP,     BP,     BP,     BP,     BP,     BP,     BP,     BP  ],
        [null,   null,   null,   null,   null,   null,   null,   null],
        [WP,     WP,     WP,     WP,     WP,     WP,     WP,     WP  ],
        [WR,     WK,     WB,     WK,     WQ,     WB,     WK,     WR  ]
    ]
    expected_over: false
    expected_outcome: null

test2:
    next_to_go: B
    pieces_matrix: [
        [null,   null,   null,   BK,     null,   null,   null,   null],
        [null,   null,   null,   WP,     null,   null,   null,   null],
        [null,   null,   null,   WK,     null,   null,   null,   null],
        [null,   null,   null,   null,   null,   null,   null,   null]
    ]
    expected_over: true
    expected_outcome: null

test3:
    next_to_go: B
    pieces_matrix: [
        [WR,     null,   null,   null,   null,   null,   BK,     null],
        [null,   WR,     null,   null,   null,   null,   null,   null],
        [null,   null,   null,   null,   null,   null,   null,   WK  ],
        [null,   null,   null,   null,   null,   null,   null,   null]
    ]
    expected_over: true
    expected_outcome: W

test4:
    next_to_go: B
    pieces_matrix: [
        [null,   null,   null,   BK,     null,   null,   null,   null],
        [null,   null,   null,   WP,     null,   null,   null,   null]
```

```
[null, null, WP, WK, null, null, null, null],
[null, null, null, null, null, null, null, null],
]

expected_over: true
expected_outcome: null

test5:
next_to_go: B
pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, WP, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]

expected_over: true
expected_outcome: W

test6:
next_to_go: W
pieces_matrix: [
    [BR, null, null, null, null, null, BR, BK, null],
    [BP, null, null, null, null, BP, BP, null],
    [null, BP, null, null, null, null, null, null],
    [null, null, BQ, null, null, null, null, null],
    [null, null, null, null, null, null, null, WQ ],
    [null, null, null, null, null, null, WP, null],
    [WP, null, null, null, null, WP, WB, null],
    [WR, null, null, null, null, null, null, WR ],
]

expected_over: false
expected_outcome: null

test7:
next_to_go: B
pieces_matrix: [
    [BR, null, null, null, null, BR, BK, WQ ],
    [BP, null, null, null, null, BP, BP, null],
    [null, BP, null, null, null, null, null, null],
    [null, null, BQ, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, WP, null],
    [WP, null, null, null, null, WP, WB, null],
    [WR, null, null, null, null, null, null, WR ],
]
```

```
]
expected_over: true
expected_outcome: W
```

chess functions\test_data\board_state\generate_all_pieces.yaml

```
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
  ]
  pieces_and_squares: [
    [WP, B1],
    [BP, A2],
    [BP, C2]
  ]
]

test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  pieces_and_squares: [
    [WR, B2],
    [BR, B4],
    [BP, C3],
    [BR, D4],
    [BP, D6],
    [BP, G3],
    [WB, G4],
    [BR, E5]
  ]
]
```

chess functions\test_data\board_state\generate_legal_moves.yaml

```
test1:
```

```
next_to_go: B
pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
]
expected_legal_moves: []
test2:
next_to_go: W
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
]
expected_legal_moves: [
    [B1, [B2, B3, A2, C2]]
]
test3:
next_to_go: B
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP, null, BP, null, null, null, null, null],
    [null, WP, null, null, null, null, null, null]
]
expected_legal_moves: [
    [A2, [A1, B1]],
    [C2, [C1, B1]],
]
test4:
next_to_go: W
pieces_matrix: [
    [BR, BN, BB, BK, BQ, BB, BN, BR],
```

```

[BP,    BP,    BP,    BP,    BP,    BP,    BP,    BP  ],
[null, null, null, null, null, null, null, null],
[WP,    WP,    WP,    WP,    WP,    WP,    WP,    WP  ],
[WR,    WN,    WB,    WK,    WQ,    WB,    WN,    WR  ]
]
expected_legal_moves: [
[A2, [A3, A4]],
[B2, [B3, B4]],
[C2, [C3, C4]],
[D2, [D3, D4]],
[E2, [E3, E4]],
[F2, [F3, F4]],
[G2, [G3, G4]],
[H2, [H3, H4]],
[B1, [A3, C3]],
[G1, [F3, H3]]
]
test5:
next_to_go: W
pieces_matrix: [
[BK, BP, null, null, null, null, WP, WK],
[BP, BP, null, null, null, null, WP, WP],
[null, null, null, BP, null, null, null, null],
[null, null, null, null, BR, null, null, null],
[null, BR, null, BR, null, null, WB, null],
[null, null, BP, null, null, null, BP, null],
[null, WN, null, null, null, null, null, null],
[null, null, null, null, null, null, null, null]
]
expected_legal_moves: [
[B2, [A4, C4, D1, D3]],
[G4, [
H3, F5, E6, D7, C8,
H5, F3, E2, D1
]],
]
]

# # specific test due to error in testing
# # FAILURE EXPECTED IT THAT C3 to D5 IS MISSED
# test6:
#   next_to_go: W
#   pieces_matrix: [
#     [BR, null, BB, BK, BQ, BB, null, BR ],
#     [BP, BP, BP, BP, BP, BP, BP, BP ],
#     [null, null, BN, null, null, BN, null, null],

```

```

#      [null, null, null, null, null, null, null, null],
#      [null, null, null, null, null, null, null, null],
#      [null, null, WN, null, WP, WN, null, null],
#      [WP,   WP,   WP,   WP,   null,   WP,   WP,   WP ],
#      [WR,   null,   WB,   WK,   WQ,   WB,   null,   WR ]
#
# ]
# expected_legal_moves: [
#     [A2, [A3, A4]],
#     [B2, [B3, B4]],
#     [D2, [D3, D4]],
#     [G2, [G3, G4]],
#     [H2, [H3, H4]],
#     [E3, [E4]],
#     [F3, [D4, E5, G1, G5, H4]],
#     [C3, [A4, B1, B5, E2, E4, D5]],
#     [A1, [B1]],
#     [H1, [G1]],
#     [E1, [E2]]
# ]

```

chess functions\test_data\board state\generate pieces of color.yaml

```

test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]
  color: W
  pieces_and_squares: [
    [WP, B1]
  ]
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]

```

```

color: B
pieces_and_squares: [
    [BP, A2],
    [BP, C2]
]

test3:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
color: W
pieces_and_squares: [
    [WR, B2],
    [WB, G4]
]
test4:
pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, null, null, BR, null, null, null],
    [null, BR, null, BR, null, null, WB, null],
    [null, null, BP, null, null, null, BP, null],
    [null, WR, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
]
color: B
pieces_and_squares: [
    [BR, B4],
    [BP, C3],
    [BR, D4],
    [BP, D6],
    [BP, G3],
    [BR, E5]
]

```

chess_functions\test_data\board_state\piece_at_vector.yaml

```

test1:
pieces_matrix: [
    [BR, BN, BB, BK, BQ, BB, BN, BR ],
    [BP, BP, BP, BP, BP, BP, BP, BP ],

```

```

[null, null, null, null, null, null, null, null],
[WP, WP, WP, WP, WP, WP, WP, WP ],
[WR, WN, WB, WK, WQ, WB, WN, WR ]
]
vectors_and_expected_piece: [
  [[0, 0], WR],
  [[1, 1], WP],
  [[2, 4], null],
  [[7, 3], null],
  [[5, 7], BB],
  [[2, 6], BP]
]

test2:
pieces_matrix: [
  [null, null, null, null, null, null, null, null],
  [null, null, null, null, null, null, null, null],
  [null, null, null, BP, null, null, null, null],
  [null, null, null, null, BR, null, null, null],
  [null, BR, null, BR, null, null, WB, null],
  [null, null, BP, null, null, null, BP, null],
  [null, WR, null, null, null, null, null, null],
  [null, null, null, null, null, null, null, null]
]
vectors_and_expected_piece: [
  [[0, 0], null],
  [[1, 1], WR],
  [[1, 3], BR],
  [[0, 7], null],
  [[7, 0], null],
  [[7, 7], null],
  [[3, 3], BR],
  [[3, 5], BP],
  [[6, 3], WB],
  [[6, 1], null]
]

```

chess game\ init .py

```

from .game import Game
# from .game_with_difficulty import Game_With_Difficulty
from .game_web import Game_Website

```

chess game\console_chess.py

```

from .game import Game

```

```
from assorted import InvalidMove

# create new chess game
# difficulty set to depth 2
game = Game(time=2, echo=True)

# this function informs the user of the details when the game is over
def handle_game_over(winner, classification):
    print(f"The game is over, the {'user' if winner==1 else 'computer'} has
won in a {classification}")

# print out the starting board
game.board_state.print_board()
print()

def prettify_legal_moves(legal_moves):
    return ", ".join(
        map(
            lambda vs: f"{vs[0].to_square()}>{vs[1].to_square()}" ,
            legal_moves
        )
    )

# keep game going until loop broken
try:
    while True:
        # user goes first
        print("Your go USER:")

        # while loop and error checking used to ensure move input
        while True:
            try:
                print("Please enter move in 2 parts")
                from_square = input("From square: ")
                to_square = input("To square: ")
                # check
                game.implement_user_move(from_square, to_square)
            except InvalidMove:
                print("This isn't a legal move, try again")
                pretty_moves =
prettify_legal_moves(game.board_state.generate_legal_moves())
                print(f"Legal moves are: {pretty_moves}")
            except ValueError:
                print("This isn't valid input, try again")
            else:
                # if it worked break out of the loop
                break

```

```
# if move results in check then output this
if game.board_state.color_in_check():
    print("CHECK!")
# print out the current board_state
# game.board_state.print_board()
# print()

# check if game over after user's move
over, winner, classification = game.check_game_over()
# if over, handle it.
if over:
    handle_game_over(winner=winner, classification=classification)
    break

# alternate, it is now the computers go
print("Computer's go: ")

# get the computers move
move, _ = game.implement_computer_move()

# print out the board again
# game.board_state.print_board()

# print out the computer's move in terms of squares
position_vector, movement_vector = move
resultant_vector = position_vector + movement_vector
piece_symbol =
game.board_state.get_piece_at_vector(resultant_vector).symbol()
print(f"Computer Moved {piece_symbol}: {position_vector.to_square()}"
      to {resultant_vector.to_square()}")

# print check if applicable
if game.board_state.color_in_check():
    print("CHECK!")

# check if the game is over, if so handle it
over, winner, classification = game.check_game_over()
if over:
    handle_game_over(winner=winner, classification=classification)
    break
# create a new line to separate for the user's next move
print()
except KeyboardInterrupt:
    print("\nGame has been ended")
```

chess_game\game_web.py

```
# this class is a variant of the game class that is to be used in a web game
# of chess
# it originally inherited from the game class but then it became to different
# so I no longer made it inherit from the game class

# from .game import Game

from assorted import safe_hash
from chess_functions import Board_State, Vector
from move_engine import Move_Engine_Timed
from schemas import serialize_piece, serialize_move

# import other modules
from chess_functions import Board_State, Vector
from move_engine import Move_Engine_Timed
from assorted import NotUserTurn, NotComputerTurn, InvalidMove, safe_hash

# # this function clears the console
# def clear_console():
#     # https://stackoverflow.com/questions/517970/how-to-clear-the-
# interpreter-console
#     os.system('cls')

# # this function allows for a functions performance to be measured
# def time_function(function):
#     start = perf_counter()
#     result = function()
#     end = perf_counter()
#     time_delta = end - start
#     return result, time_delta

# the game class is used to keep track of a chess game between a user and the
# computer
class Game_Website(object):
    # # it keeps track of:
    # # the player's color's
    # player_color_key: dict
    # # the difficulty or depth of the game
    # depth: int
    # # the current board state
    # board_state: Board_State
    # # the number of moves so far
    # move_counter: int
```

```
# move_engine: Move_Engine
# game_history_output: tuple[str]

# constructor for game object
# def __init__(self, time, user_color="W", echo=True) -> None:
def __init__(self, difficulty="medium", user_color="W") -> None:
    self.difficulty = difficulty.strip().lower()

    # based on user's color, determine color key
    self.player_color_key = {
        "W": 1 if user_color == "W" else -1,
        "B": -1 if user_color == "W" else 1
    }
    # set depth property from parameters

    # set attributes for game at start
    # start with a blank board
    self.board_state = Board_State()
    self.move_counter = 0
    # use the timed move engine to find moves
    self.move_engine = Move_Engine_Timed()
    self.move_history_output = []

# the time function is treated as a property to emulate how it was before
when it was a property (before difficulty)
@property
def time(self):
    match self.difficulty:
        case "really_easy": return 1
        case "easy": return 2
        case "medium": return 5
        case "hard": return 10
        case "really_hard": return 15
        case "extreme": return 30
        case "legendary": return 60
    raise ValueError(f"Difficulty {self.difficulty} not recognised")

# this functions adds a move to the move history,
# it doesn't make the move
def add_move_to_history(self, move):
    # should be called pre move
    # color_moving = "White" if self.board_state.next_to_go == "W" else
    "Black"
    # print({"move": move})

    # unpack move and determine squares and relevant pieces
    position_vector: Vector = move[0]
```

```

        resultant_vector = move[0] + move[1]
        moving_piece, taken_piece =
self.board_state.get_piece_at_vector(position_vector),
self.board_state.get_piece_at_vector(resultant_vector)
        from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()

        # increment move count
        self.move_counter += 1

        # generate string move description using this data
        if taken_piece is None:
            # move_description = f"Move {self.move_counter}: {color_moving} moved {moving_piece} from {from_square} to {to_square}"
            # move_description = f"{self.move_counter}: {color_moving} moved {moving_piece} from {from_square} to {to_square}"
            move_description = f"{self.move_counter}: {moving_piece} from {from_square} to {to_square}"
        else:
            # move_description = f"Move {self.move_counter}: {color_moving} moved {moving_piece} from {from_square} to {to_square}, taking piece {taken_piece}"
            # move_description = f"{self.move_counter}: {color_moving} moved {moving_piece} from {from_square} to {to_square}, taking piece {taken_piece}"
            move_description = f"{self.move_counter}: {moving_piece} from {from_square} to {to_square}, taking piece {taken_piece}"

        # append this string to the move history list
        self.move_history_output.append(move_description)

# this function makes a move on the board state and logs this in the game history
def make_move(self, move):
    assert not self.board_state.is_game_over_for_next_to_go()[0], "Cannot make move if game is over"
    assert move is not None, "Cannot move as move parameter not valid"

    self.add_move_to_history(move)

    # adjust properties that keep track of the game state
    self.board_state = self.board_state.make_move(*move)

# this function validates if the user's move is allowed and if so, makes it
# input form javascript is a move in terms of vectors

```

```

def implement_user_move(self, move: list[Vector]) -> None:
    assert not self.board_state.is_game_over_for_next_to_go()[0], "Cannot
make move if game is over"

    # check that the user is allowed to move
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    # check it is the user's go
    if which_player_next_to_go != 1:
        raise NotUserTurn(game=self)

    # if the move is not in the set of legal moves, raise an appropriate
exception
    if move not in self.board_state.generate_legal_moves():
        self.board_state.print_board()
        print(f"Move {move} not in legal moves")
        raise InvalidMove(game=self)

    # make the move
    self.make_move(move)

    # no return data needed
    # return move, self.board_state.static_evaluation()

# this function determines if the game is over and if so, what is the
nature of the outcome
def check_game_over(self):
    # returns: over: bool, winning_player: (W / B), classification: str

    # determine if board state is over for next player
    over, winner = self.board_state.is_game_over_for_next_to_go()
    # switch case statement to determine the appropriate values to be
returned in each case
    match (over, winner):
        case False, _:
            victory_classification = None
            winning_player = None
        case True, None:
            if self.board_state.is_3_board_repeats_in_game_history():
                victory_classification = "stalemate board repeat"
            else:
                victory_classification = "stalemate no legal moves"
                winning_player = None
        case True, winner:
            victory_classification = "checkmate"
            winning_player = self.player_color_key[winner]

```

```
# return appropriate values
return over, winning_player, victory_classification

# this function is used to determine the AI's move
# the time delta can be explicitly given or it can use the time provided
manually as a parameter
def best_move_function(self, time):
    if time is None:
        time = self.time

    # returns just the move
    return self.move_engine(
        board_state=self.board_state,
        # depth is based of difficulty of game based on depth parameter
        time=time,
    )[1]

    # this function is used to generate a description of the move that the
computer has completed for the client side GUI
    # it is needed to provide highlighting ect.
    # it doesn't make the computer move
def generate_computer_move_description(self, move, previous_board_state):

    # determine various qualities about the move
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector

    moved_piece =
previous_board_state.get_piece_at_vector(position_vector)
    taken_piece =
previous_board_state.get_piece_at_vector(resultant_vector)

    from_square = position_vector.to_square()
    to_square = resultant_vector.to_square()

    # return dictionary of this data ready for json serialization
    # to do this, ensure all objects such as pieces and vectors are
serialized.
    move_description = {
        "moved_piece": serialize_piece(moved_piece),
        "taken_piece": serialize_piece(taken_piece),
        "from_square": from_square,
        "to_square": to_square,
        "move": serialize_move(move)
    }

    return move_description
```

```
# this function completed the computer's move in a given time delta and
# returns a description
def implement_computer_move_and_report(self, time=None):
    previous_board_state = self.board_state

    # get next to go player (1/-1)
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )
    # check that is it the computer's turn
    if which_player_next_to_go != -1:
        raise NotComputerTurn()
        # raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

    # get the computer's move
    best_move = self.best_move_function(time=time)

    # defensive design to detect issues, ensure a move has been calculated
    assert best_move is not None

    # implement the move and return a description
    self.make_move(best_move)

    return self.generate_computer_move_description(best_move,
previous_board_state)

def __eq__(self, other: object) -> bool:
    # this method used the hash function to determine if 2 of these
    objects are the same
    if not isinstance(other, Game_Website):
        return False

    return hash(self) == hash(other)

def __hash__(self):
    # this function encodes all the data that makes a game unique in a
    hash to be stored in a database
    return hash(safe_hash((
        "Game_Website",
        self.player_color_key,
        self.time,
        self.board_state,
        self.move_counter,
        # self.move_engine,
        self.move_history_output,
```

```
        self.difficulty
    )))

# class Game_With_Difficulty(Game):
#     def __init__(self, difficulty="medium", user_color="W") -> None:
#         self.difficulty = difficulty.strip().lower()

#         # based on user's color, determine color key
#         self.player_color_key = {
#             "W": 1 if user_color == "W" else -1,
#             "B": -1 if user_color == "W" else 1
#         }
#         # set depth property from parameters
#         self.echo = False

#         # set attributes for game at start
#         self.board_state = Board_State()
#         self.move_counter = 0
#         self.move_engine = Move_Engine_Timed()
#         self.game_history_output = ()

#     @property
#     def time(self):
#         # return 0.5
#         match self.difficulty:
#             case "really easy": return 1
#             case "easy": return 2
#             case "medium": return 6
#             case "hard": return 10
#             case "really hard": return 15

#     def implement_user_move(self, position_vector: Vector, movement_vector: Vector) -> None:
#         resultant_vector: Vector = position_vector + movement_vector
#         from_square, to_square = position_vector.to_square(),
#         resultant_vector.to_square()

#         return super().implement_user_move(from_square, to_square,
#         time_delta=None, estimated_utility=None)

#     def implement_computer_move(self):
```

```

#         previous_board_state: Board_State = self.board_state
#         move, _ = super().implement_computer_move(best_move_function=None)
#         position_vector, movement_vector = move
#         resultant_vector = position_vector + movement_vector

#         moved_piece =
previous_board_state.get_piece_at_vector(position_vector)
#         taken_piece =
previous_board_state.get_piece_at_vector(resultant_vector)

#         from_square = position_vector.to_square()
#         to_square = resultant_vector.to_square()

#         move_description = {
#             "moved_piece": serialize_piece(moved_piece),
#             "taken_piece": serialize_piece(taken_piece),
#             "from_square": from_square,
#             "to_square": to_square,
#             "move": serialize_move(move)
#         }

#         return move_description

#     def __hash__(self):
#         return safe_hash(
#             self.player_color_key,
#             self.time,
#             self.echo,
#             self.board_state,
#             self.move_counter,
#             self.move_engine,
#             self.game_history_output,
#             self.difficulty
#         )

```

chess_game\game.py

```

# import other modules
from chess_functions import Board_State, Vector
from move_engine import Move_Engine_Timed
from assorted import NotUserTurn, NotComputerTurn, InvalidMove, safe_hash

from time import perf_counter
import os

```

```
# this function wipes the console to allow the updated board to be displayed
def clear_console():
    # https://stackoverflow.com/questions/517970/how-to-clear-the-
interpreter-console
    os.system('cls')

# this function allows a function's execution time to be measured
def time_function(function):
    start = perf_counter()
    result = function()
    end = perf_counter()
    time_delta = end - start
    return result, time_delta

# the game class is used to keep track of a chess game between a user and the
computer
class Game(object):
    # constructor for game object
    def __init__(self, time=10, user_color="W", echo=False) -> None:
        # based on user's color, determine color key
        self.player_color_key = {
            "W": 1 if user_color=="W" else -1,
            "B": -1 if user_color=="W" else 1
        }
        # set depth property from parameters
        self.time = time
        self.echo = echo

        # set attributes for game at start
        self.board_state = Board_State()
        self.move_counter = 0
        self.move_engine = Move_Engine_Timed()
        self.game_history_output = ()

    # hash function describes all unique properties of the game as a unique
    number
    def __hash__(self):
        return hash(safe_hash(
            "Game",
            self.player_color_key,
            self.time,
            self.echo,
            self.board_state,
            self.move_counter,
            self.move_engine,
            self.game_history_output,
```

```
)))

# thi function determines if 2 game objects are equal
def __eq__(self, other: object) -> bool:
    if not isinstance(other, Game):
        return False

    return hash(other) == hash(self)

@staticmethod
def create_row(moving_player, time_delta, move_count, new_utility,
future_utility, move_description, white_pieces_taken, black_pieces_taken,
number_legal_moves) -> str:
    # print("(moving_player, time_delta, move_count, new_utility,
future_utility, move_description, white_pieces_taken, black_pieces_taken,
number_legal_moves)")
    # print((moving_player, time_delta, move_count, new_utility,
future_utility, move_description, white_pieces_taken, black_pieces_taken,
number_legal_moves))

        # this function creates formatted string that represents a row in a
table and contains all the data provided as parameters
    if time_delta is None:
        time_delta = ""
    if future_utility is None:
        future_utility = ""
    return f"| {move_count:^14} | {moving_player:^15} | {new_utility:^15} | {future_utility:^15} | {time_delta:^15} | {number_legal_moves:^15} | {move_description:<40} | {white_pieces_taken:<35} | {black_pieces_taken:<35} |"

def print_game_history(self):
    # this function clears the console and then print out a table that
contains the history of the game

    clear_console()
    # https://www.geeksforgeeks.org/string-alignment-in-python-f-string/
    print("Moves History:")
    print()
    # print table header
    # print(self.create_row('Moving Color', 'Move Count', 'Time Taken
(sec)', 'New Utility', 'Move', 'White Pieces Taken', 'Black Pieces Taken'))
    print(
        self.create_row(
            moving_player="Moving Color",
            time_delta="Time Taken",
            new_utility="New Utility",
```

```
        future_utility="Future Utility",
        move_description="Move Description",
        white_pieces_taken="White Pieces Taken",
        black_pieces_taken="Black Pieces Taken",
        move_count="Move Number",
        number_legal_moves="NO legal moves"
    )
)
# print each row
for item in self.game_history_output:
    print(item)
print()
# print the board
self.board_state.print_board()

def make_move(self, move, time_delta, future_utility):
    # if echo, gather some data before the move
    if self.echo:
        color_moving = "White (+)" if self.board_state.next_to_go == "W"
    else "Black (-)"
        # print({"move": move})
        position_vector: Vector = move[0]
        resultant_vector: Vector = move[0] + move[1]
        moving_piece, taken_piece =
self.board_state.get_piece_at_vector(position_vector),
self.board_state.get_piece_at_vector(resultant_vector)
        from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()
        no_legal_moves = self.board_state.number_legal_moves

        # adjust properties that keep track of the game state
        # implement the move on the board state
        self.board_state = self.board_state.make_move(*move)
        # print(f"self.board_state.next_to_go --> {self.board_state.next_to_go}")
        self.move_counter += 1

        # if echo, gather move data, format and create a row and then print
        # out the move history
        if self.echo:
            # find various data points
            utility_score = self.board_state.static_evaluation()
            move_number = self.move_counter

            if taken_piece is None:
                move_description = f"{moving_piece} moved from {from_square} to {to_square}"
            else:
```

```
        move_description = f"{moving_piece} moved from {from_square}  
to {to_square}, taking piece {taken_piece}"  
  
    def pieces_missing_characters(color):  
        return list(map(  
            str,  
            self.board_state.generate_pieces_taken_by_color(color)  
        ))  
  
    white_pieces_taken = " ".join(pieces_missing_characters("W"))  
    # white_pieces_taken = "(No pieces taken)" if white_pieces_taken  
== "" else white_pieces_taken  
    black_pieces_taken = " ".join(pieces_missing_characters("B"))  
    # black_pieces_taken = "(No pieces taken)" if black_pieces_taken  
== "" else black_pieces_taken  
  
    if time_delta is not None:  
        time_delta = f"round(time_delta, 2)} sec"  
    # else:  
    #     time_delta = ""  
  
    # create a row  
    new_row = self.create_row(  
        moving_player=color_moving,  
        time_delta=time_delta,  
        move_count=move_number,  
        new_utility=utility_score,  
        move_description=move_description,  
        white_pieces_taken=white_pieces_taken,  
        black_pieces_taken=black_pieces_taken,  
        number_legal_moves=no_legal_moves,  
        future_utility=future_utility,  
    )  
  
    # add the row to the history  
    self.game_history_output = tuple(list(self.game_history_output) +  
    [new_row])  
  
    # create a now row to show check if appropriate  
    if self.board_state.color_in_check():  
        check_msg = f"CHECK: {self.board_state.next_to_go} in check"  
        new_row = self.create_row(  
            moving_player="-",  
            time_delta="-",  
            move_count="-",  
            new_utility="-",  
            move_description=check_msg,  
            white_pieces_taken="-",  
        )
```

```

        black_pieces_taken="-",
        number_legal_moves="-",
        future_utility="-",
    )
    self.game_history_output =
tuple(list(self.game_history_output) + [new_row])

# create a new row to show game over if appropriate
over, winner = self.board_state.is_game_over_for_next_to_go()
if over:
    how = "Stalemate" if winner is None else "Checkmate"
    winning_color = "White" if winner == "W" else "B"

    if how == "Stalemate":
        if self.board_state.is_3_board_repeats_in_game_history():
            game_over_msg = "Stalemate: both players draw (3
repeats of the board state)"
        else:
            game_over_msg = "Stalemate: both players draw (no
legal moves and not in check)"
    else:
        game_over_msg = f"Checkmate: {winning_color} wins"

    new_row = self.create_row(
        moving_player="-",
        time_delta="-",
        move_count="-",
        new_utility="-",
        move_description=game_over_msg,
        white_pieces_taken="-",
        black_pieces_taken="-",
        number_legal_moves="-",
        future_utility="-",
    )
    self.game_history_output =
tuple(list(self.game_history_output) + [new_row])

# print the game history
self.print_game_history()

# this function validates if the user's move is allowed and if so, makes
it
def implement_user_move(self, from_square, to_square, time_delta=None,
estimated_utility=None) -> None:
    # check that the user is allowed to move
    which_player_next_to_go = self.player_color_key.get(
        self.board_state.next_to_go
    )

```

```
if which_player_next_to_go != 1:
    raise NotUserTurn(game=self)

# unpack move into vector form
# invalid square syntaxes will cause a value error here
try:
    position_vector = Vector.construct_from_square(from_square)
    movement_vector = Vector.construct_from_square(to_square) -
position_vector
except Exception:
    raise ValueError("Square's not in valid format")

move = (position_vector, movement_vector)

# if the move is not in the set of legal moves, raise and appropriate
exception
if move not in self.board_state.generate_legal_moves():
    raise InvalidMove(game=self)

self.make_move(move, time_delta, estimated_utility)

return move, self.board_state.static_evaluation()

# this function determines if the game is over and if so, what is the
nature of the outcome
def check_game_over(self) -> list[bool, str, str]:
    # returns: over: bool, winning_player: (1/-1), classification: str

    # determine if board state is over for next player
    over, winner = self.board_state.is_game_over_for_next_to_go()
    # switch case statement to determine the appropriate values to be
    returned in each case
    match (over, winner):
        case False, _:
            victory_classification = None
            winning_player = None
        case True, None:
            if self.board_state.is_3_board_repeats_in_game_history():
                victory_classification = "stalemate board repeat"
            else:
                victory_classification = "stalemate no legal moves"
                winning_player = None
        case True, winner:
            victory_classification = "checkmate"
            winning_player = self.player_color_key[winner]
    # return appropriate values
    return over, winning_player, victory_classification
```

```

# this function determines and implements the computer move
def implement_computer_move(self, best_move_function=None):
    # for use with testing bots, a best move function can be provided for
    # use, but minimax if the default

        # get next to go player (1/-1)
        which_player_next_to_go = self.player_color_key.get(
            self.board_state.next_to_go
        )
        # check that is it the computer's turn
        if which_player_next_to_go != -1:
            raise NotComputerTurn()
            # raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

        # if no function provided, default to minimax
        if best_move_function is None:
            def best_move_function(self):
                return self.move_engine(
                    board_state = self.board_state,
                    # depth is based of difficulty of game based on depth
parameter
                    time = self.time,
                )

            # otherwise use provided function,
            # the provided function should take game as an argument and then
return data in the same format as the minimax function
            result, time_delta = time_function(
                lambda: best_move_function(self)
            )
            # print({"result": result})
            score, best_move = result
            # best_move, score = result

            # print({"best_move": best_move})
            # print({"score": score})

            assert best_move is not None

            self.make_move(best_move, time_delta, score)

        # incase is it wanted for a print out ect, return move and score
        return best_move, score

```

chess game\test fast game.py

```
import unittest
import pickle

from .game import Game

from assorted import NotComputerTurn, NotUserTurn
from chess_functions import King, Queen, Rook, Bishop, Pawn, Board_State,
Vector
from move_engine import Move_Engine_Prime

class Test_Case(unittest.TestCase):
    # this test checks that a game object can be preserved when it is encoded
    to binary and then loaded back
    def test_save_and_restore(self):
        game = Game(
            echo=False,
            time=10,
        )

        # make a load of changes to the game object
        game.implement_user_move(from_square="A2", to_square="A4")
        game.implement_computer_move(best_move_function=lambda *arg, **kwargs:
[0, [Vector(0, 6), Vector(0, -2)]])

        game.implement_user_move(from_square="B2", to_square="B4")
        game.implement_computer_move(best_move_function=lambda *arg, **kwargs:
[0, [Vector(1, 6), Vector(0, -2)]])

        game.implement_user_move(from_square="C2", to_square="C4")
        game.implement_computer_move(best_move_function=lambda *arg, **kwargs:
[0, [Vector(2, 6), Vector(0, -2)]])

        original_game = game

        pickle_file_path =
"chess_game/test_data/test_save_and_restore/saved_game.game"

        # dump it to binary and save to a file
        with open(pickle_file_path, "wb") as file:
            file.write(
                pickle.dumps(
                    game
                )
            )

        # read the binary and load it back into an object
        with open(pickle_file_path, "rb") as file:
            reloaded_game = pickle.loads(
                file.read()
            )
```

```
# check that the games are the same
self.assertTrue(
    original_game == reloaded_game,
    "The original game was not the same as the saved and reloaded
game")
)

def test_whose_go(self):
    # this function tests that the validation present can stop a player
    going twice in a row
    game = Game(time=2)

    game.implement_user_move(from_square="A2", to_square="A4")

    def try_to_make_user_move(*args, **kwargs):
        game.implement_user_move(from_square="B2", to_square="B4")

    def try_to_make_computer(*args, **kwargs):
        game.implement_computer_move()

    self.assertRaises(
        NotUserTurn,
        try_to_make_user_move,
        "The user shouldn't be able to make 2 consecutive turns"
    )

    game.implement_computer_move()

    self.assertRaises(
        NotComputerTurn,
        try_to_make_computer,
        "The computer shouldn't be able to make 2 consecutive turns"
    )

# @unittest.skip("Test takes too long, depth ")
# def test_specific_bug_null_move(self):
#     pieces_matrix = [
#         [Rook("B"), King("B"), None, None, None, None, Queen("W"),
None],
#         [None, Pawn("B"), Pawn("B"), None, None, None, None],
#         [Pawn("B"), None, None, None, None, None, None],
#         [None, None, None, Pawn("W"), None, None, None],
#         [None, None, None, None, Bishop("W"), None, None],
#         [None, None, Pawn("W"), None, None, None, None],
#         [Pawn("W"), None, None, None, None, Pawn("W"), None, Pawn("W")],
#         [None, Rook("W"), None, None, None, King("W"), None, None]
#     ]
```

```
#     next_to_go = "B"

#     board_state = Board_State(next_to_go, pieces_matrix)
#     print()
#     board_state.print_board()

#     move_engine = Move_Engine_Prime()
#     move_engine.cache_allowed = False
#     move_engine.cache_manager = None
#     move_engine.parallel = False

#     # self.assertRaises(
#     #     #     AssertionError,
#     #     #     lambda: move_engine(board_state, depth=4)
#     #     # )

#     def absolute(x): return x if x >= 0 else 0-x

#     # _, move = move_engine(board_state, depth=1)
#     # self.assertTrue(move is not None)
#     # board_state = board_state.make_move(*move)
#     # print()
#     # board_state.print_board()

#     # _, move = move_engine(board_state, depth=1)
#     # self.assertTrue(move is not None)
#     # board_state = board_state.make_move(*move)
#     # print()
#     # board_state.print_board()

#     # _, move = move_engine(board_state, depth=1)
#     # self.assertTrue(move is not None)
#     # board_state = board_state.make_move(*move)
#     # print()
#     # board_state.print_board()

#     print(board_state.is_game_over_for_next_to_go())
#     print(list(board_state.generate_legal_moves()))

#     score, move = move_engine(board_state, depth=4)

#     self.assertTrue(
```

```
#           absolute(score) < 1_000_001 and move is not None,
#           msg=repr({"score": score, "move": move})
#       )

if __name__ == '__main__':
    unittest.main()
```

database\ init .py

```
# this file does 2 things:
# it decides which objects should be exported as part of the database module
# it initialise the database (creates a connection and creates all tables)

# objects to export
from .create_database import create_session, end_session, create_engines,
end_engines
from .models import Minimax_Cache_Item, create_tables
from .handle_games import get_saved_game, save_game

# initialise database
engines_dict = create_engines(echo=False)

create_tables(engines_dict)

persistent_DB_engine = engines_dict["persistent_DB_engine"]
volatile_RAM_engine = engines_dict["volatile_RAM_engine"]
```

database\create_database.py

```
# external modules used
import os
import sqlalchemy as sqla
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.pool import QueuePool

# path to database file
DB_path = os.path.join(os.getcwd(), 'database', 'database.db')

# this function creates 2 database engines, one in RAM and one is secondary
storage
def create_engines(echo):
    # echo means, should all the queries be printed out
```

```
# caution with below code, could cause unexpected behavior if not thread
safe, further research needed
# I don't know much about threading in python, I find it confusing due to
complexities around the global interpreter lock
# as a result I had to just try various arguments to create these engines
in a way that allowed me to use them with my flask server
persistent_DB_engine = sqla.create_engine(
    'sqlite:///` + DB_path,
    echo=echo,
    poolclass=QueuePool,
    connect_args={'check_same_thread': False}
)

volatile_RAM_engine = sqla.create_engine(
    "sqlite:///memory:",
    echo=echo,
    poolclass=QueuePool,
    connect_args={'check_same_thread': False}
)
return {
    "persistent_DB_engine": persistent_DB_engine,
    "volatile_RAM_engine": volatile_RAM_engine
}

# this function safely terminates the database connection
def end_engines(engines):
    for engine in engines.values():
        engine.dispose()

# this function creates a session to access the database
def create_session(engine):
    return scoped_session(sessionmaker(bind=engine))

# this function safely ends a session
def end_session(session):
    session.commit()
    session.close()
```

database\handle_games.py

```
# import external modules and local models
import pickle

from .models import Saved_Game

# cannot import for type hint due to high risk of circular imports
```

```
# from chess_game import Game_Website

# this method queries the database for a game by cookie id
# and then deserializes the binary data to restore the game
def get_saved_game(cookie_key, session):

    # query database for game by cookie key
    result: Saved_Game = session.query(Saved_Game) \
        .where(
            # Saved_Game.game_hash == str(hash(game_hash))
            Saved_Game.cookie_key == str(cookie_key)
        ) \
        .first()

    # if the result is none (possible if browser has cookie that is invalid)
    then return none
    if result is None:
        print(f"get_saved_game game not in database so returning none")
        print(f"Couldn't retrieve game with cookie_key: {cookie_key}")
        return None

    # now deserialize the game from the binary data
    game = pickle.loads(
        result.raw_game_data
    )
    # print(f>this game recovered from database under
    cookie_key: {cookie_key})
    # game.board_state.print_board()

    # print(f"game recovered from database: {hash(game)}")
    # return the game
    return game

# this function saves a game to the database under a certain cookie ID.
def save_game(game, cookie_key, session):
    # print("saving this game into database")
    # game.board_state.print_board()
    # print(f"saving game in database game with hash {hash(game)} under
    cookie_key: {cookie_key}")

    # delete any old games with the same cookie id
    session.query(Saved_Game) \
        .where(
            # Saved_Game.game_hash == str(hash(game_hash))
            Saved_Game.cookie_key == str(cookie_key)
        ) \
        .delete()
```

```
# create a new entry in the database and commit
session.add(
    Saved_Game(game, cookie_key)
)
session.commit()
```

database\models.py

```
# import external modules
import sqlalchemy as sqla
from sqlalchemy.ext.declarative import declarative_base
import pickle

# cannot be used for type hints as it has a high risk of causing a circular
import
# from chess_game import Game_Website

# create a base object with which I will make my database tables (part of the
ORM)
Base = declarative_base()

# create an object and a table (ORM used) to represent a cached minimax call
class Minimax_Cache_Item(Base):

    # here is the metadata and the columns of the database
    __tablename__ = "Minimax_Cache"

    primary_key = sqla.Column(sqla.Integer, primary_key=True)

    board_state_hash = sqla.Column(sqla.String())
    depth = sqla.Column(sqla.INT())
    score = sqla.Column(sqla.INT())
    # 4 character move string encoded by the 4 digits of the from and movement
vectors
    move = sqla.Column(sqla.String())

    # a single entry can be created by initializing an object with this class
    def __init__(self, board_state_hash: str, depth, move: str, score: int):
        # hash board state
        self.board_state_hash = str(board_state_hash)
        self.depth = depth
        self.move = move
        self.score = score

    # string description of object
    def __repr__(self) -> str:
```

```

        return
f"Minimax_Cache_Item(board_state_hash='{self.board_state_hash}', "
depth={self.depth}, move='{self.move}', score={self.score})"

# this object represents an entry in a database table that saves games for
later reloading
class Saved_Game(Base):
    # metadata as well as columns defined here
    __tablename__ = "Saved_Games"

    primary_key = sqla.Column(sqla.Integer, primary_key=True)

    # game_hash = sqla.Column(sqla.String())
    cookie_key = sqla.Column(sqla.String())
    raw_game_data = sqla.Column(sqla.BINARY())

    # initializing an object from this class allow a new entry in the database
    to be made
    def __init__(self, game, cookie_key):
        self.cookie_key = str(cookie_key)
        self.raw_game_data = bytes(
            pickle.dumps(
                game
            )
        )

    # string description of object
    def __repr__(self) -> str:
        return f"Saved_Game(cookie_key='{self.cookie_key}')"

# create indexes on cookie key and board state hash as these are the columns
that I will use to search for a specific entry
sqla.schema.Index("board_state_hash", Minimax_Cache_Item.board_state_hash)
sqla.schema.Index("cookie_key", Saved_Game.cookie_key)

# for each engine, build these tables
def create_tables(engines_dict):
    for engine in engines_dict.values():
        Base.metadata.create_all(engine, checkfirst=True)

move_engine\__init__.py
from .minimax_parallel import Move_Engine_Prime, DB_Cache, Move_Engine_Timed
from .minimax import Move_Engine
from .cache_managers import RAM_cache, JSON_Cache

```

```
move_engine\cache_managers.py
# import from external modules
import os
import json
from time import perf_counter

# import local modules
from assorted import safe_hash
from chess_functions import Board_State, Vector
from database import create_session, end_session, persistent_DB_engine,
volatile_RAM_engine, Minimax_Cache_Item
from schemas import minimax_cache_item_schema

# create an blank object with no logic that can be used with a context manager
class Blank_Context():
    def __enter__(self, *args, **kwargs):
        # def __exit__(self, exception_type, exception_value, traceback):
        pass

    def __exit__(self, *args, **kwargs):
        pass

# ram cache doesn't need access to a context manager so it uses the blank one
class RAM_cache(Blank_Context):
    # cache is stored in a local dictionary property in the ram cache object
    def __init__(self) -> None:
        self.memoization_cache = dict()

    def cache_size(self):
        return len(self.memoization_cache)

    # the result of a minimax call is added to the cache
    def add_to_cache(self, board_state: Board_State, depth, score, move):
        # moves are stored as tuples of integers
        def serialize_move(move):
            if move is None:
                return None

            return (
                (move[0].i, move[0].j),
                (move[1].i, move[1].j)
            )
        # ensure that the cache exists in the memoization_cache property
        assert self.memoization_cache is not None

        # can use a board_state as a key as it is hashable
```

```

# the board state hash is used as the key within the dictionary (which
acts as a hash table)
    board_state_hash = board_state.database_hash()

        # print(f"CALL: add_to_cache(self, board_state={hash(board_state)}"
depth={depth}, score={score}, move={move})")
        # print(f"self.memoization_cache.get(board_state_hash) is None    --"
>      {self.memoization_cache.get(board_state_hash) is None}")

            # decide if the cache should be updated based on whether the record
already existed and if so, its depth
            if self.memoization_cache.get(board_state_hash) is None:
                needs_update = True
            else:
                best_depth_in_cache =
self.memoization_cache[board_state_hash]["depth"]
                    # print(f"best_depth_in_cache < depth    --"
>          {best_depth_in_cache} < {depth}    --->    {best_depth_in_cache < depth}")
                needs_update = best_depth_in_cache < depth

            # print(f"Cache:  {self.memoization_cache}")

        # if need an update then add the record (lesser depth cache
overwritten)
        if needs_update:
            new_data_item = {
                "depth": depth,
                "score": score,
                "move": serialize_move(move)
            }

            self.memoization_cache[board_state_hash] = new_data_item
            # if depth >= 1:
            # print(f"New data {board_state_hash} added to
cache:  {new_data_item}")
            # print(f"new_cache_size:  {self.cache_size()}")
            # print(f"Cache needed updating:  {needs_update}")

        # this function retrieves an item from cache
def search_cache(self, board_state, depth):
    # move is converted from tuple of integers back to vectors when
deserialized
    def deserialize_move(serialised_move):
        if serialised_move is None:
            return None

        return (
            Vector(*serialised_move[0]),

```

```

        Vector(*serialised_move[1])
    )

assert self.memoization_cache is not None

# get the hash ov hte board state
# board_state_hash = hash(board_state)
board_state_hash = board_state.database_hash()

# if not in cache, return none
if self.memoization_cache.get(board_state_hash) is None:
    # if depth >= 2:
    #     print(f"item {board_state_hash} not in cache
(depth={depth}))")
        return None
# else check the depth is adequate and return if appropriate
else:
    best_depth_in_cache =
self.memoization_cache[board_state_hash]["depth"]
    record_useful = best_depth_in_cache >= depth
    if record_useful:
        # if depth >= 2:
        # print(f"RAM_Cache used for
{board_state_hash}: {self.memoization_cache[board_state_hash]}")

        data_item = self.memoization_cache[board_state_hash].copy()
        data_item["move"] = deserialize_move(data_item["move"])

        return data_item
    else:
        # if depth >= 2:
        # print(f"cache not useful for item
{board_state_hash}: {self.memoization_cache[board_state_hash]}, required
depth is {depth})")
        # print(f"cache searched but not used")
        return None

# this function creates a persistent cache in a json file
class JSON_Cache(RAM_cache):
    # when the object is created, load the contents of the json file into the
memoization cache variable
    def __init__(self, file_path=None) -> None:
        # print("Initializing json cache")
        if file_path is not None:
            self.file_path = file_path
        else:
            self.file_path = r"./database/minimax_cache.json"

```

```
# default
self.memoization_cache = {}

# try get data from file
if os.path.exists(self.file_path):
    with open(self.file_path, "r") as file:
        content = file.read()
try:
    assert content != "", "Json file was blank / empty"
    cache_data = json.loads(content)

    # fix that in json, board state hash key is sting
    new_items = map(
        lambda item: (int(item[0]), item[1]),
        cache_data.items()
    )
    cache_data = dict(tuple(new_items))

    self.memoization_cache = cache_data
except Exception as e:
    # print(e)
    # print("Failed to load json cache data so using blank cache")
    pass

# inherited ram cache method add to and retrieve cache form the
memoization cache variable

# write the memoization cache dictionary to a json file
def save(self):
    print(f"Saving cache (size={self.cache_size()}) to json file")
    with open(self.file_path, "w") as file:
        file.write(
            json.dumps(
                self.memoization_cache
            )
        )

# do this when the cache manager is closed by the context manager
def __exit__(self, *args, **kwargs):
    self.save()

# this json cache doesn't save automatically
class JSON_Cache_Manual_Save(JSON_Cache):
    def __exit__(self, *args, **kwargs):
        pass

# database cache connects to the database to store minimax cached results
```

```
# it uses a database in RAM for small cache and a persistent database for
searches at a greater depth
class DB_Cache():

    # by default, depth 0 uses ram cache and greater depth uses persistent
    # cache
    def __init__(self, min_DB_depth=1):
        # set the min depth parameter as a property and then define other
        properties with starting values
        self.min_DB_depth = min_DB_depth

        self._RAM_cache = None
        self._DB_session = None
        self.engaged = False
        # a cached depth 3 call can be used if a depth 2 call is needed
        self.allow_greater_depth = True

        self.time_delta_DB = 0
        self.time_delta_schemas = 0
        self.checks_to_cache = 0
        self.retrieved_item = 0
        self.items_added = 0

    def __enter__(self, *args, **kwargs):
        # when the context manager is used and the boot up sequence runs (this
        method)
        # a database session is created (using the database module)

        # assert all(e is None for e in (self._RAM_cache, self._DB_session))
        assert not self.engaged, "Must not be engaged (entered) already to
        enter"
        self.engaged = True

        # these variables keep track of various data points while the cache is
        used within a context manager
        self.time_delta_DB = 0
        self.time_delta_schemas = 0
        self.checks_to_cache = 0
        self.retrieved_item = 0
        self.items_added = 0

        # this allows for the cache manager to be entered multiple times on
        accident without making multiple sessions
        # self._RAM_cache = RAM_cache()
        # self._persistent_DB_session =
        scoped_session(sessionmaker(bind=persistent_DB_engine))
        # self._volatile_RAM_session =
        scoped_session(sessionmaker(bind=volatile_RAM_engine))
        self._persistent_DB_session = create_session(persistent_DB_engine)
```

```
self._volatile_RAM_session = create_session(volatile_RAM_engine)

def __exit__(self, *args, **kwargs):
    # when the context manager runs the close function of the cache
    manager (this method)
    # the session objects are closed and discarded and tracker variables
    are set back to 0

    assert self.engaged, "Must be engaged (entered) to exit"
    self.engaged = False

    end_session(self._persistent_DB_session)
    self._persistent_DB_session = None

    end_session(self._volatile_RAM_session)
    self._volatile_RAM_session = None

    # print(f"Cache session lasted {round(self.time_delta_DB,
    2)}+{round(self.time_delta_schemas, 2)} sec (DB + schema time):
    {self.checks_to_cache} checks made, {self.retrieved_item} items retrieved,
    {self.items_added} items added")

    self.time_delta_DB = 0
    self.time_delta_schemas = 0
    self.checks_to_cache = 0
    self.retrieved_item = 0
    self.items_added = 0

def _search_DB_cache(self, session, board_state, depth):
    # this function hashes the board state and searches the database for
    cache corresponding to that hash

    # board_state_hash = str(hash(board_state))
    board_state_hash = board_state.database_hash()

    start = perf_counter()

    # cache must have a matching hash and a sufficient depth
    # ORM allows for queries to be written in a pythonic way
    if self.allow_greater_depth:
        result = session.query(Minimax_Cache_Item) \
            .where(
                Minimax_Cache_Item.board_state_hash == board_state_hash
            ) \
            .where(
                Minimax_Cache_Item.depth >= depth
            ) \
            .first()
```

```
        else:
            result = session.query(Minimax_Cache_Item)\n                .where(\n                    Minimax_Cache_Item.board_state_hash == board_state_hash\n                )\n                .where(\n                    Minimax_Cache_Item.depth == depth\n                )\n                .first()\n\n            stop = perf_counter()\n            self.time_delta_schemas += stop - start\n\n            # if not item found then return none\n            if result is None:\n                return None\n\n            start = perf_counter()\n\n            # use the schema object to deserialize the cached item before\n            returning it.\n            result = minimax_cache_item_schema.dump(result)\n\n            stop = perf_counter()\n            self.time_delta_schemas += stop - start\n\n            return result\n\n    def _add_to_DB_cache(self, session, board_state: Board_State, score,\n    depth, move):\n        # this function adds an item to the cache\n\n        start = perf_counter()\n\n        # board state is hashed\n        # print("adding item to database cache")\n        # board_state_hash = str(hash(board_state))\n        board_state_hash = board_state.database_hash()\n\n        # if depth >= 2:\n        #     print(f"Adding item {board_state_hash} at depth {depth} to\n        cache")\n\n        # new database entry item created with schema\n        new_item = minimax_cache_item_schema.load(\n            dict(\n                board_state_hash=board_state_hash,\n                score=score,\n\n
```

```
        depth=depth,
        move=move
    )
)

stop = perf_counter()
self.time_delta_schemas += stop - start

start = perf_counter()

# other searches at a lesses depth are deleted
# (assumes cache already checked and the other cache is worse)
session.query(Minimax_Cache_Item) \
    .where(
        Minimax_Cache_Item.board_state_hash == board_state_hash
)\ \
    .delete()

# print(f"move    -->  {move!r}")

# print(f"New item:  {new_item!r}")

# the item is added to the database and committed
session.add(new_item)
session.commit()

stop = perf_counter()
self.time_delta_DB += stop - start

def search_cache(self, board_state: Board_State, depth):
    # this function searches the cache
    assert self.engaged, "Context manager must be used"

    self.checks_to_cache += 1

    # not sure why depth sometimes is a single length tuple containing an
int
    # here is a quick fix
    if isinstance(depth, tuple):
        if len(depth) == 1:
            depth = depth[0]

    # print(f"(depth, self.min_DB_depth)    -->    {(depth,
self.min_DB_depth)}")

    # decide which session to use based on depth
    if depth < self.min_DB_depth:
```

```
        session = self._volatile_RAM_session
        # return self._RAM_cache.search_cache(board_state=board_state,
depth=depth)
    else:
        session = self._persistent_DB_session
        # return self._search_DB_cache(board_state=board_state,
depth=depth)

        # search the cache with this session and then return the result
        result = self._search_DB_cache(session=session,
board_state=board_state, depth=depth)

        if result is not None:
            self.retrieved_item += 1

    return result

def add_to_cache(self, board_state: Board_State, score, depth, move):
    # this function adds to the cache
    # this function assumes that no move valuable cache already exists (it
overwrites all other cache)
    assert self.engaged, "Context manager must be used"

    # this is used to tackle any bugs elsewhere in the program
    # it ensures that no invalid cache items are added to the database
    def absolute(x): return x if x >= 0 else 0-x
    if absolute(score) > 1_000_000 or (move is None and depth > 0):
        # don't add erroneous data to cache
        return None

    # decides that session to use based on depth
    if depth < self.min_DB_depth:
        session = self._volatile_RAM_session
        # self._RAM_cache.add_to_cache(board_state=board_state,
score=score, depth=depth, move=move)
    else:
        session = self._persistent_DB_session
        # self._add_to_DB_cache(board_state=board_state, score=score,
depth=depth, move=move)

    # the caches item is added to the database
    self._add_to_DB_cache(session=session, board_state=board_state,
score=score, depth=depth, move=move)

    self.items_added += 1
```

```

# the hash function describes only the data that makes this cache object
unique
def __hash__(self) -> int:
    hash(safe_hash(
        (
            "DB_Cache",
            # self._RAM_cache,
            # self._DB_session,
            self.min_DB_depth,
            self.engaged,
            self.allow_greater_depth,
            # self.time_delta_DB
            # self.time_delta_schemas
            # self.checks_to_cache
            # self.retrieved_item
            # self.items_added
        )
    )))

```

move engine\minimax_parallel.py

```

# import external and local modules

# import sys
import multiprocessing
from time import perf_counter

from .minimax import Move_Engine, is_time_expired

from .cache_managers import RAM_cache, DB_Cache

from chess_functions import Board_State
from assorted import ARBITRARILY_LARGE_VALUE, TimeOutError,
UnexplainedErroneousMinimaxResultError

# import time
def get_cores():
    return multiprocessing.cpu_count()
    # return multiprocessing.cpu_count()//2

# Classes with the name of JOB are essentially functions that represent a
# general task that can be completed concurrently
# they are classes as functions cannot be easily piped between threads

```

```
# # this object represents a job that a concurrent worker should complete,
# # I used an object as a function cannot be sent between workers
# class Minimax_Sub_Job:
#     # contractor: configure bot by setting these parameters as properties
#     def __init__(self, move_engine, board_state, depth, args, kwargs,
max_time=None) -> None:
#         self.move_engine: Move_Engine = move_engine
#         self.board_state: Board_State = board_state

#         # # not sure why depth sometimes is a single length tuple containing
an int
#         # # here is a quick fix
#         if isinstance(depth, tuple):
#             if len(depth) == 1:
#                 depth = depth[0]
#             assert isinstance(depth, int)

#         self.depth = depth
#         self.max_time = max_time

#         # these are other erroneous argument that could be passed to the
minimax call
#         self.args = args
#         self.kwargs = kwargs

#     def __call__(self, legal_moves_sub_array):
#         # perform part of a minimax search using a sub set of legal moves

#         # print(f"running minimax job with legal moves sub array:
(hash={hash(str(legal_moves_sub_array))})")
#         # print(f"STARTING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
#         # print(legal_moves_sub_array),

#         result = self.move_engine.minimax(
#             board_state=self.board_state,
#             depth=self.depth,
#             legal_moves_to_examine=legal_moves_sub_array,
#             # is_time_expired = self.is_time_expired,
#             max_time=self.max_time,
#             *self.args,
#             **self.kwargs
#         )
# 
```

```
#         # print(f"FINISHING sub job on moves
segment:  {hash(str(legal_moves_sub_array))}")
#         # print(f"Minimax sub job finished
(hash={hash(str(legal_moves_sub_array))})")
#         return result

# this object represents a job that a concurrent worker should complete,
# I used an object as a function cannot be sent between workers
class Minimax_Sub_Job:
    # contractor: configure bot by setting these parameters as properties
    def __init__(self, board_state, depth, args, kwargs, additional_depth,
cache_allowed, max_time=None) -> None:

        # create a cache manager as necessary
        self.cache_allowed = cache_allowed
        self.cache_manager = DB_Cache() if cache_allowed else None

        # create a move engine as necessary
        self.move_engine: Move_Engine = Move_Engine(
            cache_manager=self.cache_manager,
            cache_allowed=cache_allowed,
            additional_depth=additional_depth,
        )
        # assert isinstance(self.move_engine, Move_Engine),
f"Minimax_Sub_Job.__init__      self.move_engine of unexpected type
{type(self.move_engine)}\n{self.move_engine!r}"

        # assign parameters as properties
        self.board_state: Board_State = board_state

        self.depth = depth
        self.max_time = max_time

        # these are other erroneous argument that could be passed to the
minimax call
        self.args = args
        self.kwargs = kwargs

    def __call__(self, legal_moves_sub_array):
        # perform part of a minimax search using a sub set of legal moves

        # print(f"running minimax job with legal moves sub array:
(hash={hash(str(legal_moves_sub_array))})")
        # print(f"STARTING sub job on moves
segment:  {hash(str(legal_moves_sub_array))}")
```

```
# print(legal_moves_sub_array)d_state,
```

```
# result = self.move_engine.minimax(
# assert isinstance(self.move_engine, Move_Engine),
f"Minimax_Sub_Job.__call__      self.move_engine of unexpected type
{type(self.move_engine)}\n{self.move_engine!r}"
```

```
if self.cache_allowed:
    with self.cache_manager:
        result = self.move_engine.minimax(
            board_state=self.board_state,
            depth=self.depth,
            legal_moves_to_examine=legal_moves_sub_array,
            # is_time_expired = self.is_time_expired,
            max_time=self.max_time,
            *self.args,
            **self.kwargs
        )
else:
    result = self.move_engine.minimax(
        board_state=self.board_state,
        depth=self.depth,
        legal_moves_to_examine=legal_moves_sub_array,
        # is_time_expired = self.is_time_expired,
        max_time=self.max_time,
        *self.args,
        **self.kwargs
    )

    # print(f"FINISHING sub job on moves
segment: {hash(str(legal_moves_sub_array))}")
    # print(f"Minimax sub job finished
(hash={hash(str(legal_moves_sub_array))})")
    return result
```

```
# # this bot is responsible for performing a low depth search of a move to be
used to presort the moves
# class Presort_Moves_Sub_Job:
#     # construct object
#     # def __init__(self, depth, board_state: Board_State, move_engine:
Move_Engine,max_time=None) -> None:
#         # cache manager parameter may not be needed if the cache manager
object is also bound to the move engine
#         self.depth = depth
#         self.board_state = board_state
#         self.move_engine = move_engine
```

```
#             self.is_time_expired = is_time_expired
#             self.max_time = max_time

#     def __call__(self, move):

#         # self interrupting
#         if is_time_expired(self.max_time):
#             raise TimeOutError()

#         # score a child, use minimax first call to add caching
#         child = self.board_state.make_move(*move)

#         child_score, _ = self.move_engine.minimax(
#             board_state=child,
#             depth=self.depth,
#             # is_time_expired = self.is_time_expired,
#             max_time=self.max_time,
#         )

#         return (child_score, move)

# this bot is responsible for performing a low depth search of a move to be
# used to presort the moves
class Presort_Moves_Sub_Job:
    # construct object
    def __init__(self, depth, board_state: Board_State, cache_allowed,
max_time=None) -> None:
        # def __init__(self, depth, board_state: Board_State, move_engine:
Move_Engine, cache_allowed, cache_manager, max_time=None) -> None:
            # cache manager parameter may not be needed if the cache manager
object is also bound to the move engine

        # assign parameters as properties
        self.depth = depth
        self.board_state = board_state

        # create a cache manager as necessary
        self.cache_allowed = cache_allowed
        self.cache_manager = DB_Cache() if cache_allowed else None

        # create a move engine object
        self.move_engine = Move_Engine(
            presort_moves=True,
            additional_depth=0,
```

```
        cache_allowed=self.cache_allowed,
        cache_manager=self.cache_manager,
        # cache_allowed=self.cache_allowed,
        # cache_manager=self.cache_manager,
    )

# self.cache_allowed = cache_allowed
# self.cache_manager = cache_manager

self.is_time_expired = is_time_expired
self.max_time = max_time

def __call__(self, move):
    # cache manager already built into the minimax move engine first call
method

    # print(f"Presort job using cache manager: {self.cache_manager!r}")
    # with self.cache_manager:
    #     child = self.board_state.make_move(*move)
    #     child_score, _ = self.move_engine.minimax_first_call(
    #         board_state=child,
    #         depth=self.depth
    #     )
    #     # print("Closing cache")
    #     result = (child_score, move)
    # return result

    # self interrupting
    if is_time_expired(self.max_time):
        raise TimeOutError()

    # score a child, use minimax first call to add cache manager being
opened twice
    # no need to say no parallel as a basic move engine is used
    child = self.board_state.make_move(*move)
    if self.cache_allowed:
        with self.cache_manager:
            # child_score, _ = self.move_engine.minimax_first_call(
            child_score, _ = self.move_engine.minimax(
                board_state=child,
                depth=self.depth,
                # is_time_expired = self.is_time_expired,
                max_time=self.max_time,
            )
    else:
        # child_score, _ = self.move_engine.minimax_first_call(
        child_score, _ = self.move_engine.minimax(
```

```
        board_state=child,
        depth=self.depth,
        # is_time_expired = self.is_time_expired,
        max_time=self.max_time,
    )

    return (child_score, move)

# this is the main class to complete a minimax search in parallel.
# some methods inherited from parent class while some are overwritten
class Parallel_Move_Engine(Move_Engine):
    # construct the engine using parameters
    def __init__(self, cache_manager: RAM_cache | None = DB_Cache(),
    cache_allowed: bool = True, parallel=True, min_parallel_depth=2,
    workers=get_cores(), **kwargs) -> None:
        self.parallel = parallel
        if parallel and cache_allowed:
            assert isinstance(cache_manager, DB_Cache), "Only DB cache can be
used in parallel"
            self.min_parallel_depth = min_parallel_depth
            self.workers = workers

        # use parent class construction
        super().__init__(cache_manager=cache_manager,
        cache_allowed=cache_allowed, **kwargs)

    def should_use_parallel(self, board_state: Board_State, depth):
        return depth >= self.min_parallel_depth and self.parallel

    # this function breaks up the legal moves array into many sub arrays that
    have a similar distribution of good and bad moves
    def break_up_legal_moves_to_segments(self, legal_moves: list,
    number_sub_arrays):
        # print("Legal moves:")
        # print(legal_moves)

        legal_move_sub_arrays = [[] for _ in range(number_sub_arrays)]

        # repeatedly add next best move to the end of the sub array
        # iterating through the sub arrays to give each one a similar
        distribution of move quality
        i = 0
        while legal_moves:
            legal_move_sub_arrays[i].append(
                legal_moves.pop(0)
            )
```

```
i = (i+1) % number_sub_arrays

# reverse sub arrays so in order of best to worst
legal_move_sub_arrays = list(map(
    lambda e: list(reversed(e)),
    legal_move_sub_arrays
))

# print("Broken down")
# for sub_array in legal_move_sub_arrays:
#     print(sub_array)

return legal_move_sub_arrays

# this function runs the presort moves process in parallel

def generate_move_child_in_parallel(self, board_state: Board_State, depth: int, is_maximizer, give_child=True, max_time=None):
    # print("generate_move_child_in_parallel called")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # explore_depth = max(depth-3, 0)
    explore_depth = max(depth-2, 0)

    if (not self.parallel):
        yield from super().generate_move_child(
            board_state=board_state,
            depth=depth,
            is_maximizer=is_maximizer,
            give_child=give_child,
            # is_time_expired=is_time_expired
            max_time=max_time
        )
    else:
        # if can do presort concurrently
        # # print("getting legal moves")
        legal_moves = list(board_state.generate_legal_moves())

        # print("setting up job")
        # create job
        job = Presort_Moves_Sub_Job(
            depth=explore_depth,
            board_state=board_state,
            cache_allowed = self.cache_allowed,
```

```
        max_time=max_time,
    )

    # concurrently map the job onto the iterable of legal moves
    # print("generate_move_child_in_parallel: Using multiprocessing
pool to presort in parallel")
    with multiprocessing.Pool(self.workers) as pool:
        moves_and_scores = pool.map(
            func=job,
            iterable=legal_moves
        )

    # print("generate_move_child_in_parallel: finished with
multiprocessing pool")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # sort by score
    moves_and_scores = sorted(
        moves_and_scores,
        key=lambda triplet: triplet[0],
        reverse=is_maximizer
    )
    # use map to get rid of score

    def move_and_blank_child(score_and_move):
        _, move = score_and_move
        return (move, None)

    moves_scores_children = map(move_and_blank_child,
moves_and_scores)

    # print("children sorted by score")

    yield from moves_scores_children

# this function performs a minimax search in parallel

def parallel_minimax(self, board_state, depth, max_time=None, *args,
**kwargs):
    # print("Call parallel_minimax")
    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()
```

```
# print("parallel_minimax, checking cache")
# if cache can be used then check if this search has already been
complete
if self.cache_allowed:
    result = self.cache_manager.search_cache(
        board_state=board_state,
        depth=depth
    )
    if result is not None:
        # print("no call needed as cache is sufficient")
        return result["score"], result["move"]

# print("Parallel minimax called")

# assert isinstance(depth, int)

# assert depth >= 2, "depth must be greater or equal to 2 to do
parallelization"

is_maximizer = self.color_maximizer_key.get(board_state.next_to_go)

# print("started getting moves_and_child_sorted")
# print("getting legal moves, presorted")

# generate presorted moves iterable
# print("parallel_minimax: calling generate_move_child_in_parallel")
moves_sorted = list(self.generate_move_child_in_parallel(
    board_state=board_state,
    depth=depth,
    is_maximizer=is_maximizer,
    give_child=False,
    # is_time_expired=is_time_expired,
    max_time=max_time,
))

# print("parallel_minimax: finished generate_move_child_in_parallel")

# print("finished getting moves_and_child_sorted")
# print("moves_sorted: ")
# print(moves_sorted)

# break up these pre sorted legal moves into segments for each sub job
to work on
# print("parallel_minimax, breaking up moves into sub arrays")
legal_move_sub_arrays = self.break_up_legal_moves_to_segments(
    legal_moves=moves_sorted,
    number_sub_arrays=self.workers
)
```

```
# print("parallel_minimax: finished break_up_legal_moves_to_segments")

# self interrupting
if is_time_expired(max_time):
    raise TimeOutError()

# print("legal moves sub arrays generated")

# print("Printing legal moves sub arrays")
# print(legal_move_sub_arrays)

# print("Defining job for workers")

# print("parallel_minimax, defining sub job")
# use these legal moves to do many simultaneous minimax operations

# I updated the sub job classes so that I don't need to pass a cache
manager object to them
# this was causing issues as the database connection that the database
cache manager contained couldn't be piped between workers

# # create a job function
# job = Minimax_Sub_Job(
#     # move_engine=self,
#     move_engine=Move_Engine(

#         cache_allowed=False,
#         cache_manager=None,
#         # cache_manager=cache_manager,
#         # cache_allowed=cache_allowed,

#         # is this the miracle fix?
#         additional_depth=self.additional_depth,

#         use_validator=self.use_validator,

#     ),
#     board_state=board_state,
#     depth=depth,

#     # is_time_expired = is_time_expired,
#     max_time=max_time,

#     args=args,
#     kwargs=kwargs,
# )
```

```
# create a job function

job = Minimax_Sub_Job(
    board_state=board_state,
    depth=depth,
    max_time=max_time,
    additional_depth=self.additional_depth,
    cache_allowed=self.cache_allowed,
    args=args,
    kwargs=kwargs,
)

# print(f"Using pool to complete jobs in parallel (pool size = {cores}))"

# map this job function onto the legal moves sub array in parallel
# print("parallel_minimax: Using multiprocessing pool")
with multiprocessing.Pool(self.workers) as pool:
    minimax_sub_job_results = pool.map(
        func=job,
        iterable=legal_move_sub_arrays
    )

# self interrupting
if is_time_expired(max_time):
    raise TimeOutError()

# print("parallel_minimax: multiprocessing finished")

# set a best move and score variable with starting values
best_move = None
if is_maximizer:
    best_score = 0-(ARBITRARILY_LARGE_VALUE + 1)
else:
    best_score = ARBITRARILY_LARGE_VALUE + 1

# select best move
for result in minimax_sub_job_results:
    score, move = result

    # if this move is better then update the best move and score
variables
        if (is_maximizer and score > best_score) or (not is_maximizer and
score < best_score):
            best_score = score
            best_move = move
```

```
# add the result to the cache
if self.cache_allowed:
    self.cache_manager.add_to_cache(
        board_state=board_state,
        depth=depth,
        move=best_move,
        score=best_score
    )

# print("finished getting best outcome")

return best_score, best_move

# this function is able to perform handle the initial non recursive call
for a parallel minimax search

def minimax_first_call_parallel(self, board_state: Board_State, depth,
max_time=None, *args, **kwargs):
    # print(f"CALL    minimax_first_call_parallel")
    # print(f"CALL minimax_first_call_parallel(self,
board_state={hash(board_state)}, depth={depth}, *args, **kwargs)")
    # assert isinstance(depth, int)

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    should_use_parallel =
self.should_use_parallel(board_state=board_state, depth=depth)
    # print(f"minimax_first_call_parallel:  should_use_parallel  --> {should_use_parallel}")

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

# I find the below code clearer, but unfortunately it won't work as
the functions cannot be pickled and sent/piped between the workers

# if should_use_parallel:
#     def chosen_minimax_function():
#         return self.parallel_minimax(
#             board_state=board_state,
#             depth=depth,
#             cache_allowed=self.cache_allowed,
#             cache_manager=self.cache_manager,
#             # is_time_expired = is_time_expired,
#             max_time=max_time,
```

```
#                 *args,
#                 **kwargs
#             )
# else:
#     def chosen_minimax():
#         return self.minimax(
#             board_state=board_state,
#             depth=depth,
#             max_time=max_time,
#             *args, **kwargs
#         )

# if self.cache_allowed:
#     with self.cache_manager:
#         result = chosen_minimax()

# else:
#     result = chosen_minimax()

# the below code simply make 2 decisions,
# should parallel minimax be used or vanilla minimax?
# should cache be used?
# it is unfortunate that the above function based solution didn't work
as be bellow code features lots of repetition
if should_use_parallel:
    if self.cache_allowed:
        with self.cache_manager:
            result = self.parallel_minimax(
                board_state=board_state,
                depth=depth,
                max_time=max_time,
                *args,
                **kwargs
            )
    else:
        result = self.parallel_minimax(
            board_state=board_state,
            depth=depth,
            max_time=max_time,
            *args,
            **kwargs
        )
else:
    if self.cache_allowed:
        with self.cache_manager:
            result = self.minimax(
                board_state=board_state,
```

```
        depth=depth,
        max_time=max_time,
        *args, **kwargs
    )
else:
    result = self.minimax(
        board_state=board_state,
        depth=depth,
        max_time=max_time,
        *args, **kwargs
    )

if self.use_validator:
    self.validator(result=result, board_state=board_state)

return result

# when the object is called, use minimax_first_call_parallel to handle the
call
def __call__(self, *args, **kwargs):
    return self.minimax_first_call_parallel(*args, **kwargs)

# this move engine class represents be best configuration of the move engine
# it also updates some functions form parallel minimax (improves upon it)
class Move_Engine_Prime(Parallel_Move_Engine):
    def __init__(self) -> None:
        super().__init__(
            parallel=True,
            cache_allowed=True,
            cache_manager=DB_Cache(min_DB_depth=1),
            additional_depth=1,
            # additional_depth=0,
            presort_moves=True,
            color_maximizer_key={"W": True, "B": False},
            use_validator=False,
            # workers = get_cores()
            workers=5
        )
        self.depth = 2

    # when called, use standard depth if no depth parameter provided
    def __call__(self, board_state: Board_State, depth=None, *args, **kwargs):
        if depth is None:
            return super().__call__(board_state=board_state, depth=self.depth,
*args, **kwargs)
        else:
            return super().__call__(board_state=board_state, depth=depth,
*args, **kwargs)
```

```
# this function returns a boolean to decide if parallel processing is
needed.

# it takes the board state as a parameter as the evaluation could depend
on the legal moves to analyze
def should_use_parallel(self, board_state: Board_State, depth):
    # if not allowed to use parallel or hte game is over then return false
    if not self.parallel:
        return False

    over, _ = board_state.is_game_over_for_next_to_go()
    if over:
        return False

    # otherwise begin by estimating the depth that the decision tree will
be searched to
    if board_state.check_encountered:
        likely_depth = depth + 0.5 * self.additional_depth
    else:
        likely_depth = depth

    # then estimate the branching factor by looking at the average legal
moves available to both players
    board_state_a = board_state
    legal_moves_a = list(board_state.generate_legal_moves())
    some_legal_move_a = legal_moves_a[0]

    board_state_b = board_state_a.make_move(*some_legal_move_a)

    branching_factor_a = len(legal_moves_a)
    branching_factor_b = len(list(board_state_b.generate_legal_moves()))


    likely_branching_factor = 1/2 * (branching_factor_a +
branching_factor_b)

    # use an exponential formula to create and estimate for static
evaluations
    estimated_static_eval = likely_branching_factor ** likely_depth
    # print({"estimated_static_eval": estimated_static_eval})

    # if the number of evaluations exceeds that of depth 2 on the starting
positions, then use parallel
    return (estimated_static_eval >= 400)

def break_up_legal_moves_to_segments(self, legal_moves: list,
number_sub_arrays):
    # use the original break up legal moves function as it is better
```

```
return super().break_up_legal_moves_to_segments(legal_moves,
number_sub_arrays)

# the below code was experimental and turned out to not improve
performance,
# it divided the scored legal move up differently into sub arrays
# it put the best few in the first sub array and then the next best
few in the next and so on
# this was worse than the previous function. this is because creating
a distribution or good and bad from best to worst improved the pruning

# legal_move_sub_arrays = [[] for _ in range(number_sub_arrays)]

# number_legal_moves = len(legal_moves)
# number_workers = self.workers

# moves_per_worker = number_legal_moves // number_workers

# workers_with_extra = number_legal_moves % number_workers
# workers_without_extra = number_workers - workers_with_extra

# moves_per_sub_array = [moves_per_worker] * workers_without_extra +
[moves_per_worker+1] * workers_with_extra
# assert sum(moves_per_sub_array) == number_legal_moves

# # print(moves_per_sub_array)

# # >>> x = [1,2,3,4,5]
# # >>> x[:2]
# # [1, 2]
# # >>> x[2:]
# # [3, 4, 5]

# for worker_index in range(number_workers):
#     moves_in_array = moves_per_sub_array[worker_index]
#     legal_move_sub_arrays[worker_index] =
legal_moves[:moves_in_array]
#     legal_moves = legal_moves[moves_in_array:]

# # print("Broken down")
# # for sub_array in legal_move_sub_arrays:
# #     print(sub_array)

# yield from legal_move_sub_arrays

# this minimax algorithm explores the decision tree to a given depth before
stopping
```

```
class Move_Engine_Timed(Move_Engine_Prime):
    # explore for a certain amount of time
    def timed_call(self, board_state: Board_State, time):
        # so that internal jobs can access the time used up indicator
        # self.max_time = perf_counter() + time
        # take 3 seconds to close threads

        time_delta_allowed = time

        # time_delta_allowed = max(time - 3, 0)
        # time_delta_allowed = max(time - 10, 0)
        # time_delta_allowed = max(time - 4, 0.5)
        # print({"time_delta_allowed": time_delta_allowed})

        # stop at max time
        self.max_time = perf_counter() + time_delta_allowed

    def time_used_up():
        return perf_counter() >= self.max_time

    # need minimum of depth 1 as depth 0 doesn't give a move
    # ensure that even if time = 0, at least a depth 1 search is completed
    # deepest_result = self.minimax_first_call(board_state=board_state,
depth=1)
    deepest_result = self.minimax_first_call(board_state=board_state,
depth=1, variable_depth=0)
    depth = 2

    def absolute(x): return x if x >= 0 else 0-x
    # while True:

        # while there is time left, keep searching to a greater depth
        while not time_used_up():
            # print(f"Iterating again: time_used_up() --> {time_used_up()}")
            try:
                # deepest_result =
                self.minimax_first_call_parallel(board_state=board_state, depth=depth,
is_time_expired=time_used_up)

                    # search the tree in parallel if needed
                    result =
self.minimax_first_call_parallel(board_state=board_state, depth=depth,
max_time=self.max_time, variable_depth=0)
                    score, move = result

                # if the move or score are invalid then raise the appropriate
error
```

```
        if absolute(score) == 1_000_001 or move is None:
            raise UnexplainedErroneousMinimaxResultError()

    except UnexplainedErroneousMinimaxResultError:
        # I don't know the cause of the error but I can catch it here
        and prevent it causing issues and producing unexpected behaviour
        break
    except TimeOutError:
        # print(f"Breaking: time_used_up() --> {time_used_up()}")
        break
    else:
        # print(f"Completes depth={depth} so incrementing depth")
        # print(f"Result at depth {depth}: {deepest_result}")

        # if neither the time is used up or an erroneous result was
        produced, keep searching at a greater depth
        deepest_result = result
        depth += 1

    # return the best move
move, _ = deepest_result
assert move is not None

# print(f"Returning result at greatest depth={depth}")
return deepest_result, depth

def __call__(self, board_state: Board_State, time, *args, **kwargs):
    # use the timed_call handler function

    assert not board_state.is_game_over_for_next_to_go()[0]

    # start = perf_counter()
    deepest_result, _ = self.timed_call(board_state=board_state,
time=time)
    # end = perf_counter()

    # time_delta = end - start
    # print(f"timed minimax set to {time} seconds actually took
{round(time_delta, 3)} seconds")
    return deepest_result

# the below function was used to take benchmarks of the timed minimax
algorithm in order to gauge how accurately it stuck to its time limit
def check_timed():
    board_state = Board_State()
    move_engine = Move_Engine_Timed()
    time = 0
```

```

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 1)
#     time += perf_counter() - start
#     print(f"At total time {time} sec: depth={depth}; result={result}")

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 2)
#     time += perf_counter() - start
#     print(f"At total time {time} sec: depth={depth}; result={result}")

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 5)
#     time += perf_counter() - start
#     print(f"At total time {time} sec: depth={depth}; result={result}")

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 10)
#     time += perf_counter() - start
#     print(f"At total time {time} sec: depth={depth}; result={result}")

# for _ in range(6):
#     start = perf_counter()
#     result, depth = move_engine(board_state, 15)
#     time += perf_counter() - start
#     print(f"At total time {time} sec: depth={depth}; result={result}")

while True:
    start = perf_counter()
    result, depth = move_engine(board_state, 20)
    time += perf_counter() - start
    print(f"At total time {time} sec: depth={depth}; result={result}")

if __name__ == "__main__":
    check_timed()

```

[move_engine\minimax.py](#)

```

# import local modules
# cannot import game as causes circular import, if necessary put in same file
# from .board_state import Board_State
# from .assorted import ARBITRARILY_LARGE_VALUE, TimeOutError

```

```
# from .vector import Vector
# from . import cache_managers as cm

from chess_functions import Board_State
from assorted import ARBITRARILY_LARGE_VALUE, TimeOutError
from chess_functions import Vector
from . import cache_managers as cm

from collections.abc import Iterable
# import multiprocessing
from time import perf_counter

# this function is used to ensure that the minimax function can be self
interrupting
def is_time_expired(max_time):
    if max_time is None:
        return False
    # print("⌚", end="")
    return perf_counter() >= max_time

# def time_preemptor(max_time):
#     if max_time is None:
#         if perf_counter() >= max_time:
#             raise TimeOutError()
#     else:
#         print("⌚", end="")

# this function is used to print out the results fo a function
# def print_decorator(function):
#     name = function.__name__
#     def wrapper(*args, **kwargs):
#         result = function(*args, **kwargs)
#         print(f"{name}(args={args!r}, kwargs={kwargs!r}) --> {result!r}")
#         return result
#     return wrapper

# the minimax function grew too large and so the function had to be broken up
into a callable object with many methods
# this allowed the function to be broken up further into sub functions to make
the problem easier to tackle.
# this also reduced the number of parameters needed when calling the function
as some are given as configuration parameters to the constructor
class Move_Engine():
    def __init__(self, cache_manager: cm.RAM_cache | None = cm.JSON_Cache(),
cache_allowed: bool = True, additional_depth=1, presort_moves: bool = True,
color_maximizer_key: dict | None = None, use_validator: bool = False) -> None:
        # setup object by setting parameters provided as properties
```

```
        self.additional_depth = additional_depth
        # self.variable_depth = additional_depth >= 1
        self.use_validator = use_validator

        # if parallelize:
        #     self.presort_moves = True
        #     self.parallelize = True
        # else:
        #     self.presort_moves = presort_moves
        #     self.parallelize = False

        self.presort_moves = presort_moves

        if color_maximizer_key is not None:
            self.color_maximizer_key = color_maximizer_key
        else:
            self.color_maximizer_key = {"W": True, "B": False}

        # self.cache_allowed = cache_allowed
        # if cache_allowed:
        #     self.cache_manager = JSON_cache()
        #     # print("Reusing json cache singleton object")
        #     self.cache_manager = singleton_JSON_cache
        #     # self.cache_manager = RAM_cache()
        # else:
        #     self.cache_manager = None

        if cache_allowed:
            assert cache_manager is not None, f"Cache manager is None, not provided"
            # assert isinstance(cache_manager, cm.RAM_cache), f"Cache manager is of type {type(cache_manager)!r} not of type RAM_cache"
            self.cache_allowed = True
            self.cache_manager = cache_manager
        else:
            self.cache_allowed = False
            self.cache_manager = None

        self.static_evaluations_presort = 0
        self.static_evaluations_non_presort = 0

        # boolean function, extra depth
@property
def variable_depth(self):
    return self.additional_depth > 0

# property counter to keep track of static evaluations
```

```
@property
def static_evaluations(self):
    return self.static_evaluations_presort +
self.static_evaluations_non_presort

# this function returns an iterable that yields child nodes to be examined
# moves will be pre sorted for a depth n call using depth n-2
def generate_move_child(self, board_state: Board_State, depth: int,
is_maximizer, give_child=True, max_time=None):
    # returns move, child
    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # if presorting th moves it not possible, then simply yield child
nodes from legal moves in order
    if (not self.presort_moves) or (depth <= 1):
        for move in board_state.generate_legal_moves():
            # if is_time_expired():
            if is_time_expired(max_time):
                raise TimeOutError()

            child_board_state: Board_State = board_state.make_move(*move)
            if give_child:
                yield move, child_board_state
            else:
                yield move, None

    # else is needed as I used yield above not return
else:
    # gather triplets of all legal moves as well as corresponding
moves and scores
    moves_scores_children = []
    appropriate_depth = depth - 2

    # iterate through legal moves
    for move in board_state.generate_legal_moves():
        # self interrupting
        if is_time_expired(max_time):
            raise TimeOutError()

        # get the child board state
        child_board_state = board_state.make_move(*move)

        # score the child board state
        score, _ = self.minimax(
            board_state=child_board_state,
```

```
        depth=appropriate_depth,
        variable_depth=False,
        part_of_presort=True
    )
    if give_child:
        moves_scores_children.append([score, move,
child_board_state])
    else:
        moves_scores_children.append([score, move, None])

    # sort by score
    moves_scores_children = sorted(
        moves_scores_children,
        key=lambda triplet: triplet[0],
        reverse=is_maximizer
    )
    # use map to get rid of score
    moves_scores_children = map(lambda triplet: triplet[1:],
moves_scores_children)

    # moves_scores_children = list(moves_scores_children)
    # moves_scores_children_printable = list(map(
    #     lambda x: (x[0], hash(x[1])),
    #     moves_scores_children
    # ))
    # print("Inspecting children:")
    # print(moves_scores_children_printable)

    yield from moves_scores_children
    # for move, child in moves_scores_children:
    #     # print(f"vanilla generate_move_child about to yield: {({move,
child})}")
    #     yield (move, child)

# this is code extracted out of the minimax function
# it contains a base case to stop the recursive function
# it also contains a recursive case for when check is encountered
(variable depth)
def pseudo_base_case(self, board_state: Board_State, alpha, beta,
part_of_presort: bool):
    # returns score: int, move: (V, V) | None
    # used then depth == 0 or game over

    is_over, _ = board_state.is_game_over_for_next_to_go()
    in_check = board_state.color_in_check()

    # base case has a secret extra recursive case
```

```

if (self.variable_depth) and (not is_over) and (in_check):
    # print("Check extra depth search ")
    score, move = self.minimax(
        board_state=board_state,
        depth = self.additional_depth,
        alpha=alpha,
        beta=beta,
        variable_depth=False
    )
    return score, move, self.additional_depth
# else, use true base case and perform a static evaluation
# increment counters for further testing
else:
    if part_of_presort:
        self.static_evaluations_presort += 1
    else:
        self.static_evaluations_non_presort += 1
    return board_state.static_evaluation(), None, 0

# this is the minimax function
# it is recursive and is able to search the decision tree to find a move
def minimax(self, board_state: Board_State, depth, alpha=-ARBITRARILY_LARGE_VALUE+1, beta=ARBITRARILY_LARGE_VALUE+1, variable_depth: bool | None = None, part_of_presort=False, legal_moves_to_examine: Iterable | None = None, max_time=None) -> tuple[int, tuple[Vector, Vector]]:
    # print(f"Analysing this board state at depth {depth}: {hash(board_state)}")
    # print(repr(board_state))
    # over, _ = board_state.is_game_over_for_next_to_go()
    # print({"over": over})
    # legal_moves = list(board_state.generate_legal_moves())
    # print({"legal_moves": legal_moves})
    # print()
    # board_state.print_board()

    # self interrupting
    if is_time_expired(max_time):
        raise TimeOutError()

    # not sure why depth sometimes is a single length tuple containing an int
    # here is a quick fix
    if isinstance(depth, tuple):
        if len(depth) == 1:
            depth = depth[0]

    # if depth >= 1:

```

```
# print(f"STARTED: minimax(board_state={hash(board_state)},  
depth={depth})")  
# print(f"Finished: minimax(board_state={hash(board_state)},  
depth={depth}) returned ({}, {})")  
  
# returns score, move  
if variable_depth is None:  
    variable_depth = self.variable_depth  
  
# if cache allowed, then search the cache  
# if a cached result can be found then end the function  
  
# if self.cache_allowed and depth >= 1:  
if self.cache_allowed:  
    cached_result =  
self.cache_manager.search_cache(board_state=board_state, depth=depth)  
    # print(f"Checked cache {cached_result}")  
    if cached_result is not None:  
        # if depth >= 1:  
            # print(f"Finished: minimax(board_state={hash(board_state)},  
depth={depth}) returned ({cached_result['score']}, {cached_result['move']})")  
        return cached_result["score"], cached_result["move"]  
  
# call base case:  
# call base case and add the result to the cache, if cache allowed  
over, _ = board_state.is_game_over_for_next_to_go()  
if depth == 0 or over:  
    best_score, best_move, final_depth = self.pseudo_base_case(  
        board_state=board_state,  
        alpha=alpha,  
        beta=beta,  
        part_of_presort=part_of_presort  
    )  
    # if best_move is not None:  
    #     best_child_board_state = board_state.make_move(*best_move)  
    # else:  
    #     best_child_board_state = None  
  
    # print("Base case, trying to add to cache")  
    if self.cache_allowed:  
        self.cache_manager.add_to_cache(  
            board_state=board_state,  
            depth=final_depth,  
            score=best_score,  
            move=best_move
```

```
)  
  
        # print(f"Base Case:  minimax(board_state={board_state!r},  
depth={depth}) returned: score={score}, move={move!r}")  
  
        # if depth >= 1:  
        # print(f"Finished: minimax(board_state={hash(board_state)},  
depth={depth}) returned ({best_score}, {best_move})")  
        return best_score, best_move  
  
  
# set up initial variables for the functions final return value  
best_move: tuple[Vector] | None = None  
# best_child_board_state: Board_State | None = None  
best_score: int | None = None  
is_maximizer = self.color_maximizer_key.get(board_state.next_to_go)  
  
# if no legal moves array provided then generate legal moves and child  
nodes from board state  
if legal_moves_to_examine is None:  
    legal_moves_to_examine =  
self.generate_move_child(board_state=board_state, depth=depth,  
is_maximizer=is_maximizer, max_time=max_time)  
  
  
# legal_moves_to_examine = list(legal_moves_to_examine)  
# print(repr(legal_moves_to_examine))  
# assert len(legal_moves_to_examine) > 0  
  
  
if is_maximizer:  
    # all valid scores are better than this, including loss  
    best_score = 0-(ARBITRARILY_LARGE_VALUE+1)  
  
    # print("VANILLA: list legal_moves_to_examine:")  
    # print(list(legal_moves_to_examine))  
  
    # iterate through the legal moves  
    for move, child_board_state in legal_moves_to_examine:  
        # self interrupting  
        if is_time_expired(max_time):  
            raise TimeOutError()  
  
        # if child node not already found, get the child node  
        if child_board_state is None:  
            child_board_state = board_state.make_move(*move)
```

```
# print(f"testing move {move!r}")

# recursive minimax call with depth n-1 to evaluate child
score, _ = self.minimax(
    board_state=child_board_state,
    depth=depth-1,
    alpha=alpha,
    beta=beta,
    # is_time_expired=is_time_expired,
    max_time=max_time,
)
# if depth == 2:
#     print(f"Move {move} examined, its score was {score}")

# update the best score, best move variables as well as alpha
and beta
if score > best_score:
    # print("best score beaten")
    alpha = max(alpha, score)

best_score = score

# print(f"updating best move to {best_move}")
best_move = move
# best_child_board_state = child_board_state

# if beta <= alpha:
#     if depth == 2:
#         print(f"Pruning at a best score of {best_score}")
#         break
# if beta <= alpha and depth<2:
#     break

# if the best that the minimizer can do (higher up the tree)
is already better than this, they will never pick this branch so stop
exploring
if beta <= alpha:
    break

# repeat this process for the minimizer
else:
    best_score = ARBITRARILY_LARGE_VALUE+1

for move, child_board_state in legal_moves_to_examine:
    # if is_time_expired():
    if is_time_expired(max_time):
        raise TimeOutError()
```

```
if child_board_state is None:
    child_board_state = board_state.make_move(*move)
# print(f"testing move {move!r}")
score, _ = self.minimax(
    board_state=child_board_state,
    depth=depth-1,
    alpha=alpha,
    beta=beta,
    # is_time_expired=is_time_expired,
    max_time=max_time,
)
# if depth == 2:
#     print(f"Move {move} examined, its score was {score}")
if score < best_score:
    # print("best score beaten")
    beta = min(beta, score)

best_score = score

# print(f"updating best move to {best_move}")
best_move = move
# best_child_board_state = child_board_state

# if beta <= alpha:
#     if depth == 2:
#         print(f"Pruning at a best score of {best_score}")
#         break
# if beta <= alpha and depth<2:
#     break
if beta <= alpha:
    break

# add the result to the cache
# print("recursive case trying to add to cache")
if self.cache_allowed:
    self.cache_manager.add_to_cache(
        board_state=board_state,
        depth=depth,
        score=best_score,
        move=best_move
    )

# print(f"Recursive Case: minimax(board_state={board_state!r},
depth={depth}) returned: score={score}, move={move!r}")
# if depth >= 1:
#     # print(f"Finished: minimax(board_state={hash(board_state)},
depth={depth}) returned ({best_score}, {best_move})")
```

```
# return the result
return best_score, best_move

# this function checks the output of the minimax function, it can be
enables to prevent unintended behaviour
def validator(self, result, board_state: Board_State):
    try:
        assert len(result) == 2, "result should be of length 2 (score and
move)"

        score, move = result
        assert isinstance(score, int), "score should be of type int"

        assert len(move) == 2, "move should be a length 2 array (position
vector and movement vector)"
        assert all(isinstance(v, Vector) for v in move), "both elements in
move should be of type vector"

        position_vector, move_vector = move
        resultant_vector = position_vector + move_vector

        assert all(v.in_board() for v in (position_vector,
resultant_vector)), "both position vector and movement vector must be in
board"

        assert move in board_state.generate_legal_moves(), "move must be
in the board state legal move generator"

    except AssertionError as e:
        raise ValueError(e)
    # else:
    #     # print("Validator completed all checks")

# this function is called for the first call (not any recursive calls),
# it sets up the cache as necessary and also uses the validator if
required
def minimax_first_call(self, board_state: Board_State, depth, *args,
**kwargs):
    if self.cache_allowed:
        with self.cache_manager:
            result = self.minimax(board_state=board_state, depth=depth,
*args, **kwargs)
    else:
        result = self.minimax(board_state=board_state, depth=depth, *args,
**kwargs)

    if self.use_validator:
```

```

        self.validator(result=result, board_state=board_state)

    return result

# the object can be called like a function in order to call the minimax
first call function
def __call__(self, *args, **kwargs):
    return self.minimax_first_call(*args, **kwargs)

# this function allows for the minimax function to be analyze to measure
static evaluation ect.
def benchmark_minimax(self, *args, **kwargs):
    def reset_counts():
        self.static_evaluations_non_presort = 0
        self.static_evaluations_presort = 0

    # reset all counts, time the call, collate data and return it with
minimax result
    reset_counts()

    start_time = perf_counter()
    result = self.minimax_first_call(*args, **kwargs)
    end_time = perf_counter()

    time_taken = end_time - start_time

    benchmark_data = {
        "time_duration": time_taken,
        "evals_presort": self.static_evaluations_presort,
        "evals_non_presort": self.static_evaluations_non_presort
    }

    reset_counts()

    return result, benchmark_data

```

move_engine\parallel minimax testing.py

```

# this file is not part of the final solution
# it was used to benchmark the parallel move engine to ensure that it is
faster.

import random
from minimax_parallel import Move_Engine_Prime, DB_Cache

```

```
# from chess_functions import Board_State
from assorted import random_board_state

from random import choice as random_choice, randint as random_int
from time import perf_counter
from math import sqrt
from itertools import product as iter_product


def mean_and_standard_deviation(times):
    f = len(times)
    if f == 0:
        return None, None
    sum_x = sum(times)
    sum_x_squared = sum(time**2 for time in times)

    mean = sum_x / f
    variance = (sum_x_squared/f) - mean**2
    standard_deviation = sqrt(variance)

    return round(mean, 2), round(standard_deviation, 2)

def run_benchmark():

    # move_engine = Move_Engine_Prime()
    # board_state = random_board_state(30)
    # print("Performing minimax at variable depth on this board state:")
    # board_state.print_board()

    # print("\n")
    # for depth in range(6):

        # start = perf_counter()
        # result = move_engine(board_state, depth)
        # time_delta = perf_counter() - start

        # print(f"\nRESULT: move_engine(board_state, {depth}) finished in
{time_delta} --> {result}\n")

    # cache_manager = DB_Cache(min_DB_depth=1)

    move_engine_4_workers = Move_Engine_Prime(cache_allowed=False, workers=4)
    move_engine_8_workers = Move_Engine_Prime(cache_allowed=False, workers=8)
    move_engine_linear = Move_Engine_Prime(cache_allowed=False,
parallel=False)

    num_workers_move_engine_times_array = (
        (4, move_engine_4_workers, []),
        (8, move_engine_8_workers, []),
        (1, move_engine_linear, [])
    )
```

```
(8, move_engine_8_workers, []),  
    (1, move_engine_linear, []),  
)  
  
trials = 10  
depth = 3  
  
print(f"Completing time trial testing for  
{len(num_workers_move_engine_times_array)} variables across {trials} trials")  
print(f"controlling depth at {depth} and disabling caching")  
print()  
try:  
    for trial in range(1, trials+1):  
        board_state = random_board_state(random_int(0, 30))  
  
        for workers, move_engine, times in  
num_workers_move_engine_times_array:  
            start = perf_counter()  
            move_engine(board_state, depth)  
            end = perf_counter()  
  
            time_delta = end - start  
            times.append(time_delta)  
  
            print(f"Trial {trial} completed for {workers} workers in  
{round(time_delta, 2)} sec")  
    except KeyboardInterrupt:  
        pass  
  
    for workers, _, times in num_workers_move_engine_times_array:  
        mean, standard_deviation = mean_and_standard_deviation(times)  
        print()  
        print(f"For {trials} trials, {workers} workers:")  
        print(f"The mean time was: {mean} sec with a standard deviation of:  
{standard_deviation}sec")  
        print()  
  
# def main():  
#     # cache_manager = DB_Cache(min_DB_depth=0)  
#     cache_manager = DB_Cache(min_DB_depth=1)  
#     move_engine_with_cache = Move_Engine_Prime(cache_allowed=True,  
cache_manager=cache_manager)  
#     move_engine_no_cache = Move_Engine_Prime(cache_allowed=False)  
  
#     trials = 5  
#     depth = 4  
#     moves_range = range(4, 10)
```

```
#     times_with_cache = []
#     times_no_cache = []
#     for trial in range(1, trials+1):
#         board_state = random_board_state(
#             random_choice(moves_range)
#         )
#         print(f"trial {trial}, analysing the following board
state: {hash(board_state)}")
#         board_state.print_board()
#         print("\n")

#         start = perf_counter()
#         result_with_cache = move_engine_with_cache(board_state, depth)
#         end = perf_counter()
#         time_delta_with_cache = end - start
#         times_with_cache.append(time_delta_with_cache)

#         start = perf_counter()
#         result_no_cache = move_engine_no_cache(board_state, depth)
#         end = perf_counter()
#         time_delta_no_cache = end - start
#         times_no_cache.append(time_delta_no_cache)

#         error_msg = f"Error: Minimax functions returned results of different
score: cache != no cache ---> {result_with_cache[0]} !=
{result_no_cache[0]}"
#         # cache may use a more in depth search leading to a different score
#         # assert result_with_cache[0] == result_no_cache[0], error_msg

#         print(f"trial {trial} of board state {hash(board_state)}:    with
cache {round(time_delta_with_cache, 2)} sec,    no cache
{round(time_delta_no_cache, 2)} sec")
#         print("\n")

#         mean_cache, standard_deviation_cache =
mean_and_standard_deviation(times_with_cache)
#         print()
#         print(f"For {trials} trials, with cache")
#         print(f"The mean time was: {mean_cache} sec    with a standard deviation
of: {standard_deviation_cache}sec")
#         print()

#         mean_no_cache, standard_deviation_no_cache =
mean_and_standard_deviation(times_no_cache)
#         print()
#         print(f"For {trials} trials, with cache")
#         print(f"The mean time was: {mean_no_cache} sec    with a standard
deviation of: {standard_deviation_no_cache}sec")
```

```
#      print()

def time_function(function):
    start = perf_counter()
    result = function()
    end = perf_counter()
    time_delta = end - start
    return result, time_delta

def main():
    worker_values = [1, 2, 4, 8, 5]

    move_engines_by_workers = {
        1: Move_Engine_Prime(),
        2: Move_Engine_Prime(),
        4: Move_Engine_Prime(),
        8: Move_Engine_Prime(),
        5: Move_Engine_Prime(),
    }
    move_engines_by_workers[1].parallel = False
    move_engines_by_workers[2].workers = 2
    move_engines_by_workers[4].workers = 4
    move_engines_by_workers[8].workers = 8
    move_engines_by_workers[5].workers = 5

    for workers in worker_values:
        move_engines_by_workers[workers].cache_allowed = False
        move_engines_by_workers[workers].additional_depth = False

    times_by_workers_by_depth = {
        2: {
            1: [],
            2: [],
            4: [],
            8: [],
            5: [],
        },
        3: {
            1: [],
            2: [],
            4: [],
            8: [],
            5: [],
        },
        5: {
            1: [],
            2: [],
            4: [],
            8: [],
            5: [],
        },
    }
```

```
trials = 1
# moves_range = range(0, 50)
moves_range = (30,)

for trial in range(1, trials+1):
    moves = random_choice(moves_range)
    board_state = random_board_state(moves)
    print(f"trial {trial}, analysing the following board
state: {hash(board_state)} (moves={moves})")
    board_state.print_board()
    print("\n")

    for depth in (2, 3):
        print(f"trial {trial}, analysing board
state: {hash(board_state)} (moves={moves}, depth={depth})")

        result_scores = []

        for workers in worker_values:
            result, time_delta = time_function(lambda:
move_engines_by_workers[workers](board_state, depth))
            score, _ = result

            times_by_workers_by_depth[depth][workers].append(time_delta)
            result_scores.append(score)

        try:
            assert all(result == result_scores[0] for result in
result_scores)
        except AssertionError:
            raise ValueError(f"Result scores where different
(deterministic no caching so should be the same): {result_scores}")
        else:
            print(f"Common minimax score: {result_scores[0]}")

            print(f"trial {trial} of board state {hash(board_state)} at
depth={depth} finished")
            print()

        for depth in (2, 3):
            for workers, times_array in times_by_workers_by_depth[depth].items():
                # mean, standard_deviation =
mean_and_standard_deviation(times_array)
                mean, _ = mean_and_standard_deviation(times_array)
                print()
                print(f"For {trials} trials, with {workers} workers at depth
{depth}: the mean time was {mean} sec")
```

```

        # print(f"The mean time was {mean} sec      with a standard
deviation was {standard_deviation} sec")
        # print(f"The mean time was {mean} sec")
        # print()

def build_cache():
    cache_manager = DB_Cache(min_DB_depth=0)
    move_engine = Move_Engine_Prime(cache_manager=cache_manager)

while True:
    moves_advanced = random.randint(16, 48)
    board_state = random_board_state(moves_advanced)
    for depth in (2, 3):
        start = perf_counter()
        score, move = move_engine(board_state, depth)
        duration = perf_counter() - start

        duration = round(duration, 2)

        from_v, move_v = move
        to_v = from_v + move_v
        from_v, to_v = from_v.to_square(), to_v.to_square()

        print(f"{moves_advanced} moves in:
move_engine(<{hash(board_state)}>, {depth}) took {duration} sec --> score:
{score} with move {from_v} to {to_v}")

if __name__ == "__main__":
    main()

```

move engine\test slow timed minimax engine.py

```

# this test is responsible for testing various mutations of the minimax
function and how they play, it is not a data driven test

# imports of external modules.
from random import choice as random_choice, randint
# from itertools import product as iter_product
import pickle
import unittest
# from functools import wraps
import multiprocessing
import os
import csv
from time import perf_counter

```

```
# imports of local modules
from chess_game import Game
# from chess_functions import Board_State
# from minimax import Move_Engine
from move_engine import Move_Engine_Timed, Move_Engine_Prime
from chess_functions import random_board_state

# from assorted import ARBITRARILY_LARGE_VALUE
# from board_state import Board_State
# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given
function a given number of times
# def repeat_decorator_factory(times):
#     def decorator(func):
#         @wraps(func)
#         def wrapper(*args, **kwargs):
#             for _ in range(times):
#                 func(*args, **kwargs)
#         return wrapper
#     return decorator

# this is a utility function that maps a function across an iterable but also
converts the result to a list data structure
def list_map(func, iter):
    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a message
# it converts pieces to symbols if they are not none
def map_pieces_matrix_to_symbols(pieces_matrix):
    return list_map(
        lambda row: list_map(
            lambda square: square.symbol() if square else None,
            row
        ),
        pieces_matrix
    )

# this functions updates a CSV file with the moves and scores of a chess game
# for graphical analysis in excel
# each move in the game causes a new row to be added
def csv_write_move_score(file_path, move_counter, move, score):
    # convert move to a pair of squares
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector

    from_square = position_vector.to_square()
    to_square = resultant_vector.to_square()
```

```
# # if file doesn't exist, create it and add the headers
# if not os.path.exists(file_path):
#     with open(file_path, "w", newline="") as file:
#         writer = csv.writer(file, delimiter=",")
#         writer.writerow(("from_square", "to_square", "score"))

# add data as a new row
with open(file_path, "a", newline="") as file:
    writer = csv.writer(file, delimiter=",")
    # writer.writerow(("Move_counter", "B_score", "from_square",
# "to_square", "W_score"))
    writer.writerow((move_counter, -score, from_square, to_square, score))

# this function is used to time another function to allow for performance
# testing
def time_function(function):
    start = perf_counter()
    result = function()
    end = perf_counter()
    time_delta = end - start
    return result, time_delta

# this function aims to save games to a file as part of an unfinished feature
def save_games(games: tuple, file_path):
    with open(file_path, "wb") as file:
        file.write(
            pickle.dumps(
                games
            )
        )

# this method aims to load a game from a file as part of another unfinished
# feature
def load_games(file_path):
    if not os.path.exists(file_path):
        return dict()
    with open(file_path, "rb") as file:
        return pickle.loads(
            file.read()
        )

# the below function contains the logic to perform a test between 2 minimax
# bots to assert that the good bot is better

# this contains the majority of the logic to do a bot vs bot test with the
# game class
```

```
# it is a component as it isn't the whole test
def minimax_test_component(description, good_bot, bad_bot, success_criteria,
write_to_csv, csv_folder, csv_file_name, load_game=False, save_game=False):
    # print(f"CALL minimax_test_component(description={description},
write_to_csv={write_to_csv})")
    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,
    # the test is designed to assert that good bot wins (and or draws in some
cases)

    # if the write to csv file it enabled
    # generate csv path
    if write_to_csv:
        # not sure why but the description sometimes contains an erroneous
colon, this is caught and removed
        # was able to locate bug to here, as it is a test I added a quick fix
        # bug located, some description stings included them

        # create folders
        if not os.path.exists(csv_folder):
            os.makedirs(csv_folder)

        # create file with headers if is doesn't exist
        csv_path = f"{csv_folder}/{csv_file_name}.csv"

        # overwrite so it is blank
        with open(csv_path, "w", newline="") as file:
            writer = csv.writer(file, delimiter=",")
            writer.writerow(("Move_counter", "B_score", "from_square",
"to_square", "W_score"))

    # pickle_file_path = f"{csv_folder}/{csv_file_name}.games"
    # if load_game and os.path.exists(pickle_file_path):
    #     print("Attempting to load game")
    #     indexes = (-1, -2)
    #     games = load_games(pickle_file_path)
    #     for game in games:
    #         print(hash(game))
    #         print(game.board_state.next_to_go)
    #     for index in indexes:
    #         try:
    #             game = games[index]
    #             assert game.board_state.next_to_go == "W", "next to go
wasn't user so use other game"
    #             except Exception as e:
    #                 print(e)
    #                 continue
```

```
#         else:
#             print(f"Using game {hash(game)} at index {index}")
#             games = games[:index]
# else:

# create a new game
print("Creating new game")
game: Game = Game(echo=True)
# games = tuple()

# start a new blank game
# depth irrelevant as computer move function passed as parameter
print()
print(f"Beginning test game: {description}")

# def record_new_game(games, new_game):
#     games = tuple(list(games) + [new_game])
#     save_games(games, pickle_file_path)
#     return games

# if save_game:
#     games = record_new_game(games, game)

# keep them making moves until return statement breaks loop
while True:
    # print()
    # game.board_state.print_board()
    # print()
    # get move choice from bad bot

    # the bad bots move is implemented as the user plays as
white (advantage)
    result, time_delta = time_function(
        lambda: bad_bot(game)
    )
    score, move_choice = result
    # print({"move_choice": move_choice})

    # serialised to is can be passed as a user move (reusing game class)
position_vector, movement_vector = move_choice
resultant_vector = position_vector + movement_vector
from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()

    # implement bad bot move and update csv
    # move, score = game.implement_user_move(from_square=from_square,
to_square=to_square, time_delta=time_delta)
```

```
move, _ = game.implement_user_move(from_square=from_square,
to_square=to_square, time_delta=time_delta, estimated_utility=score)

# games = record_new_game(games, game)

if write_to_csv:
    csv_write_move_score(
        file_path=csv_path,
        move=move,
        score=game.board_state.static_evaluation(),
        move_counter=game.move_counter
    )

# piece_moved = game.board_state.get_piece_at_vector(resultant_vector)
# print(f"Move {game.move_counter}: bad bot moved {piece_moved} from
{from_square} to {to_square} with a score perceived of {score}")

# if game.board_state.color_in_check():
#     print(f"CHECK: {game.board_state.next_to_go} in check")

# see if this move causes the test to succeed or fail or keep going

# use the provided success criteria to determine if the game should be
over.
success, msg, board_state = success_criteria(game,
description=description)
# result = success_criteria(game, description=description)
# print({"result": result})
# success, msg, board_state = result

if success is not None:
    # game.board_state.print_board()
    return success, msg, board_state

# repeat for the good bot
# providing good bot function, implement good bot move and update csv
move, score =
game.implement_computer_move(best_move_function=good_bot)

# games = record_new_game(games, game)

# update CSV
if write_to_csv:
    csv_write_move_score(
        file_path=csv_path,
        move=move,
        score=game.board_state.static_evaluation(),
        move_counter=game.move_counter
```

```

        )
# unpack move in terms of to and from squares
position_vector, movement_vector = move
resultant_vector = position_vector + movement_vector
# piece_moved = game.board_state.get_piece_at_vector(resultant_vector)
from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()
    # print(f"Move {game.move_counter}: good bot moved {piece_moved} from
{from_square} to {to_square} with a score perceived of {score}")

    # if game.board_state.color_in_check():
    #     print(f"CHECK: {game.board_state.next_to_go} in check")

    # again check if this affects the test
success, msg, board_state = success_criteria(game,
description=description)
if success is not None:
    # game.board_state.print_board()
    return success, msg, board_state

    # # if needed provide console output to clarify that slow bot hasn't
crashed
    # if game.move_counter % 10 == 0 or depth >= 3:
    # print(f"Moves {game.move_counter}: static evaluation ->
{game.board_state.static_evaluation()}, Minimax evaluation -> {score} by turn
{description}")

# below are some function that have been programmed as classes with a __call__
method.
# these are basically fancy functions that CAN BE HASHED.
# I had to manually do this under the hood hashing as it is needed to allow
communication between the threads
# a job must be hashable to be piped to a thread (separate python instance)

# this is a pipe-able object that makes a random move
class Random_Bot():
    # picks a random move
    def __call__(self, game):
        # determine move at random
        legal_moves = list(game.board_state.generate_legal_moves())
        assert len(legal_moves) != 0
        # match minimax output structure
        # score, best_move
        return None, random_choice(legal_moves)

    def __hash__(self) -> int:

```

```
        return hash("I am random bot, I am a unique singleton so each instance
can share a hash")

# this bot picks a good move
# it exploration is limited by time
# has constructor to allow for configuration
class Bot_By_Time():
    # configure for depth and allow variable depth
    def __init__(self, time, cache_allowed=False):
        self.time = time
        self.move_engine = Move_Engine_Timed()
        self.move_engine.cache_allowed = cache_allowed

    # make minimax function call given config
    def __call__(self, game):
        return self.move_engine(
            board_state=game.board_state,
            time=self.time
        )
        # result = self.move_engine(
        #     board_state=game.board_state,
        #     time=self.time
        # )
        # print(f"Bot_By_Time:    result={result}")
        # return result

    def __hash__(self) -> int:
        return hash(f"Bot_By_Time(time={self.time})")

# this is a bot that has its exploration limited by depth
class Bot_By_Depth():
    # configure for depth and allow variable depth
    def __init__(self, depth, cache_allowed=False):
        self.depth = depth
        self.move_engine = Move_Engine_Prime()
        self.move_engine.cache_allowed = cache_allowed

    # make minimax function call given config
    def __call__(self, game):
        return self.move_engine(
            board_state=game.board_state,
            depth=self.depth
        )
        # result = self.move_engine(
        #     board_state=game.board_state,
        #     depth=self.depth
        # )
```

```
# print(f"Bot_By_Depth:    result={result}")
# return result

def __hash__(self) -> int:
    return hash(f"Bot_By_Depth(depth={self.depth})")

# used to look at a game and decide if the test should finish
class Success_Criteria():
    # constructor allow config for stalemates to sill allow test to pass
    def __init__(self, allow_stalemate_3_states_repeated: bool):
        self.allow_stalemate_3_states_repeated =
allow_stalemate_3_states_repeated

    def __call__(self, game: Game, description):
        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do
        match game.check_game_over():

            # if 3 repeat stalemate, check with config wether is is allows
            # case True, None, "Stalemate":
            case True, None, _:
                # game.board_state.print_board()
                if game.board_state.is_3_board_repeats_in_game_history() and
self.allow_stalemate_3_states_repeated:
                    # game.board_state.print_board()
                    # print(f"Success: Stalemate at {game.move_counter} moves
in test {description}: 3 repeat board states, outcome specify included in
allowed outcomes")
                        # return True, f"Success: Stalemate at {game.move_counter}
moves in test {description}: 3 repeat board states, outcome specify included
in allowed outcomes",
                    map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
                    return (
                        True,
                        f"Success: Stalemate at in test {description}: 3
repeat board states, outcome specify included in allowed outcomes",
                        map_pieces_matrix_to_symbols(game.board_state.pieces_m
atrix),
                    )
            else:
                # game.board_state.print_board()
                # print(f"FAILURE: ({description}) stalemate caused (3
repeats? -> {game.board_state.is_3_board_repeats_in_game_history()})")
                    return (
                        False,
                        f"FAILURE: ({description}) stalemate caused (3
repeats? -> {game.board_state.is_3_board_repeats_in_game_history()})",
```

```

        map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
    )
# good bot loss causes test to fail
# case True, 1, "Checkmate":
case True, 1, _:
    # game.board_state.print_board()
    # print(f"Failure: ({description}) computer lost")
    return (
        False,
        f"Failure: ({description}) computer lost",
        map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
    )
# good bot win causes test to pass
# case True, -1, "Checkmate":
case True, -1, _:
    # game.board_state.print_board()
    # print(f"SUCCESS: ({description}) Game has finished and been
won in {game.move_counter} moves")
    return (
        True,
        f"SUCCESS: ({description}) Game has finished and been won
in {game.move_counter} moves",
        map_pieces_matrix_to_symbols(game.board_state.pieces_matrix),
    )
# if the game isn't over, return success as none and test will
continue
case False, _, _:
    return (
        None,
        None,
        None,
    )

def hash(self):
    return
hash(f"Success_Criteria(allow_stalemate_3_states_repeated={self.allow_stalemate_3_states_repeated})")

# given a test package (config for one test), carry it out
def execute_test_job(test_data_package):
    # deal with unexplained bug where argument is tuple / list length 1
    # containing relevant dict (quick fix as only a test)
    # I was able to identify that this is where it occurs and add a correction
    # but I am not sure what the cause of the bug is

```

```
if any(isinstance(test_data_package, some_type) for some_type in (tuple, list)):
    if len(test_data_package) == 1:
        test_data_package = test_data_package[0]

    # print(f"test_data_package --> {test_data_package}")

    # really simple, call minimax test component providing all keys in package
    # as keyword arguments
    return minimax_test_component(**test_data_package)

# this pool jobs function for completing tests in parallel is not needed when
# the move engine is parallelized

# this function takes an iterable of hashable test_packages
# it all 8 logical cores on my computer to multitask to finish the test sooner
def pool_jobs(test_data):
    # counts logical cores
    # my CPU is a 10th gen i7
    # it has 4 cores and 8 logical cores due to hyper threading
    # with 4-8 workers I can use 100% of my CPU
    cores = multiprocessing.cpu_count()

    # create a pool
    with multiprocessing.Pool(cores) as pool:
        # map the execute_test_job function across the set of test packages
        # using multitasking
        # return the result
        # return pool.map(
        #     func = execute_test_job,
        #     iterable = test_data
        # )
        yield from pool.map(
            func=execute_test_job,
            iterable=test_data
        )

# test case contains unit tests
# multitasking only occurs within a test, tests are themselves executed
# sequentially
# I could pool all tests into one test function but this way multiple failures
# can occur in different tests
# (one single test would stop at first failure)

overnight = False
RANDOM_TRIAL_NUM = 3
```

```
# this function asserts that all elements in a list are the same
def test_same(some_list):
    assert len(some_list) > 0
    return all(some_list[0] == element for element in some_list)

# @unittest.skip("Takes too long")
class Test_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and
    checks the results with a unittest
    # a failure is correctly identified to correspond to the function that
    called this function
    # reduces repeated logic
    def check_test_results(self, test_results):
        # for success, msg, final_pieces_matrix in test_results:
        for success, msg, _ in test_results:
            # print()
            # for row in final_pieces_matrix:
            #     row = " ".join(map(
            #         lambda square: str(square).replace("None", ". "),
            #         row
            #     ))
            #     print(row)
            # print(msg)

            # i choose to iterate rather than assert all as this allows me to
            have the appropriate message on failure
            self.assertTrue(
                success,
                msg=msg
            )

    # this function tests that a cached call always produces the same result
    as one without caching
    # it also ensures that the second cache call for the same search is much
    faster
    # this test is completed on many random board states
    def test_cache_is_faster(self):
        trials = 40
        # trials = 5

        # create and configure move engines
        move_engine_with_cache = Move_Engine_Prime()
        move_engine_without_cache = Move_Engine_Prime()

        move_engine_without_cache.cache_allowed = False
```

```
for move_engine in (move_engine_with_cache,
move_engine_without_cache):
    move_engine.depth = 2
    move_engine.cache_manager.allow_greater_depth = False

    # repeat test for many trials
    for _ in range(trials):
        # note random_board_state as a function is inefficient as it is
        used for testing only, takes a while
        # generate a random board state
        board_state = random_board_state(60)
        # turn of 3 repeat stalemates so cache and without behave the same
        board_state.three_repeat_stalemates_enabled = False

        # board_state.print_board()

        # check result with out cache
        true_result = move_engine_without_cache(board_state)
        true_result = list(true_result)
        true_result[1] = list(true_result[1])

        # attempt 1 with cache
        first_cache_result, first_cache_time_delta = time_function(
            lambda: move_engine_with_cache(board_state)
        )
        first_cache_result = list(first_cache_result)
        first_cache_result[1] = list(first_cache_result[1])

        # check that the results were the same
        self.assertEqual(
            first_cache_result,
            true_result,
            "The first cache call should give the same result as without
cache"
        )

        # attempt 2 with cache
        second_cache_result, second_cache_time_delta = time_function(
            lambda: move_engine_with_cache(board_state)
        )

        second_cache_result = list(second_cache_result)
        second_cache_result[1] = list(second_cache_result[1])

        # check that the results were again the same
        self.assertEqual(
            first_cache_result,
            true_result,
```

```
        "The second cache call should give the same result as without
cache"
    )

    # check that with cache was much faster
    self.assertLessEqual(
        second_cache_time_delta,
        0.2 * first_cache_time_delta,
        "When using database cache, the lookup should be at least 5
times quicker than calculation"
    )

# tests basic minimax vs random moves
def test_timed_vs_randotron(self):
    trials = RANDOM_TRIAL_NUM

    # test package generated to include relevant data and logic (bots and
success criteria)
    def generate_jobs():
        for num in range(1, trials+1):
            for time in [2, 5, 10, 20]:
                yield {
                    "description": f"test: {time}s timed move engine vs
randotron",
                    "good_bot": Bot_By_Time(time=time),
                    "bad_bot": Random_Bot(),
                    "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
                    "write_to_csv": True,
                    "csv_folder":
"./test_reports/test_timed_vs_randotron",
                    "csv_file_name": f"test_timed_{time}s_num_{num}"
                }

    def generate_test_results():
        for test_specs in generate_jobs():
            yield minimax_test_component(**test_specs)

        self.check_test_results(
            generate_test_results()
    )

# tests basic minimax vs random moves
@unittest.skip("takes too long")
def test_depth_vs_randotron(self):
    trials = RANDOM_TRIAL_NUM
```

```

# test package generated to include relevant data and logic (bots
and success criteria)
def generate_jobs():
    for num in range(1, trials+1):
        # for depth in range(4):
        for depth in [1, 2]:
            yield {
                "description": f"test: depth {depth} move engine
vs randotron",
                "good_bot": Bot_By_Depth(depth=depth),
                "bad_bot": Random_Bot(),
                "success_criteria": Success_Criteria(
                    allow_stalemate_3_states_repeated=(depth<=1)
                ),
                "write_to_csv": True,
                "csv_folder":
"../test_reports/test_depth_vs_randotron",
                "csv_file_name": f"test_depth_{depth}_num_{num}"
            }

def generate_test_results():
    for test_specs in generate_jobs():
        yield minimax_test_component(**test_specs)

    self.check_test_results(
        generate_test_results()
    )

# test that bots that can explore more are better
# @unittest.skip("takes too long")
def test_bots_by_depth(self):
    # test package generated to include relevant data and logic (bots and
success criteria)
    def generate_jobs():
        # don't do depth 0s as they cannot decide a move
        for depth_greater in (3,):
            # for depth_greater in (2, 3):
                # ensure depth_a >= depth_b
                # for depth_lesser in range(1, depth_greater):
                # for depth_lesser in range(1, depth_greater+1):
                # allow_draw = (depth_lesser == depth_greater)
                depth_lesser = depth_greater - 1
                yield {
                    "description": f"test: depth {depth_greater} bot vs depth
{depth_lesser} bot",
                    "good_bot": Bot_By_Depth(depth=depth_greater),
                    "bad_bot": Bot_By_Depth(depth=depth_lesser),

```

```
        "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
                    # "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=allow_draw),  
                    "write_to_csv": True,  
                    "csv_folder": "./test_reports/test_bots_by_depth",  
                    "csv_file_name":  
f"test_depth_{depth_greater} vs depth_{depth_lesser}"  
    }  
  
    def generate_test_results():  
        for test_specs in generate_jobs():  
            yield minimax_test_component(**test_specs)  
  
    self.check_test_results(  
        generate_test_results()  
    )  
  
# test that bots that can explore more are better  
def test_bots_by_time(self):  
    # test package generated to include relevant data and logic (bots and  
success criteria)  
    def generate_jobs():  
        time_deltas = [2, 5, 10, 15, 20]  
        for time_delta_lesser in time_deltas:  
            time_delta_greater = max(1.4*time_delta_lesser,  
4+time_delta_lesser)  
                # time_delta_greater = max(2*time_delta_lesser,  
10+time_delta_lesser)  
                # time_delta_greater = max(4*time_delta_lesser,  
15+time_delta_lesser)  
                yield {  
                    "description": f"test: {time_delta_greater}s timed bot vs  
{time_delta_lesser}s timed bot",  
                    "good_bot": Bot_By_Time(time=time_delta_greater),  
                    "bad_bot": Bot_By_Time(time=time_delta_lesser),  
                    # "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=False),  
                    "success_criteria":  
Success_Criteria(allow_stalemate_3_states_repeated=True),  
                    "write_to_csv": True,  
                    "csv_folder": "./test_reports/test_bots_by_time",  
                    "csv_file_name":  
f"test_{time_delta_greater}s_timed_bot_vs_{time_delta_lesser}s_timed_bot"  
    }  
  
    def generate_test_results():  
        for test_specs in generate_jobs():
```

```
        yield minimax_test_component(**test_specs)

    self.check_test_results(
        generate_test_results()
    )

# test what the parallel move engine doesn't produce a different output
def test_deterministic_outcomes_parallel(self):
    # create move engines and configures them
    with_parallel_engine = Move_Engine_Prime()

    without_parallel_engine = Move_Engine_Prime()

    for engine in (with_parallel_engine, without_parallel_engine):
        engine.cache_allowed = False
        engine.additional_depth = 0

    with_parallel_engine.parallel = True
    without_parallel_engine.parallel = False

    def always(*args, **kwargs): return True
    def never(*args, **kwargs): return False

    with_parallel_engine.should_use_parallel = always
    without_parallel_engine.should_use_parallel = never

    # generate a large number of random board states
    def gen_random_board_states():
        for _ in range(40):
            yield random_board_state(moves=randint(0, 10))
        for _ in range(20):
            yield random_board_state(moves=randint(0, 20))
        for _ in range(10):
            yield random_board_state(moves=randint(0, 40))

    # test that for each board state, the move engines produce the same
    # deterministic outcome
    for board_state in gen_random_board_states():
        score_parallel, move_parallel = with_parallel_engine(board_state)
        score_non_parallel, move_non_parallel =
without_parallel_engine(board_state)

        fail_msg = f"Test hash={hash(board_state)}: score_parallel !="
without --> {score_parallel} != {score_non_parallel}"
        # assert score_parallel == score_non_parallel, fail_msg
        self.assertEqual(
            score_parallel, score_non_parallel, fail_msg
```

```

        )

        fail_msg = f"Test hash={hash(board_state)}: move parallel != without --> {move_parallel} != {move_non_parallel}"
        # assert move_parallel == move_non_parallel, fail_msg
        self.assertEqual(
            move_parallel, move_non_parallel, fail_msg
        )

# for trial in range(1, 101):
#     try:
#         moves = random_choice(range(10, 50))
#         board_state = random_board_state(moves)

#         score_parallel, _ = with_parallel_engine(board_state)
#         score_non_parallel, _ = without_parallel_engine(board_state)

#         fail_msg = f"Test moves={moves} hash={hash(board_state)}: score_parallel != without --> {score_parallel} != {score_non_parallel}"
#         assert score_parallel == score_non_parallel, fail_msg
#     except AssertionError as e:
#         print(f"TEST FAILURE: (trial={trial})")
#         board_state.print_board()
#         print(repr(board_state))
#         self.fail(str(e))
#     except Exception as e:
#         self.fail(f"UNEXPECTED ERROR (trial = {trial}): {str(e)}")

if __name__ == '__main__':
    unittest.main()

```

```

schemas\__init__.py
from .cache_item_schema import minimax_cache_item_schema
from .socket_schemas import serialize_legal_moves, serialize_pieces_matrix,
deserialize_move, serialize_piece, serialize_move, deserialize_pieces_matrix

from .socket_schemas import *
schemas\cache_item_schema.py

# import external and external modules
from marshmallow import Schema, fields, pre_load, pre_dump, post_load,
post_dump

from chess_functions import Vector
from database import Minimax_Cache_Item

# the vector schema describes how to serialize and deserialize a vector object
# it converts between Vector(i=a, j=b) and [a, b]

```

```
class Vector_Schema(Schema):
    i = fields.Integer(required=True)
    j = fields.Integer(required=True)

    @pre_load
    def pre_load(self, vector, **kwargs):
        data = {}
        data["i"] = vector.i
        data["j"] = vector.j
        return data

    @post_dump
    def post_dump(self, data, **kwargs):
        return Vector(
            i=data["i"],
            j=data["j"]
        )

# this move schema is for database cache
# it converts between [Vector(a, b), Vector(c, d)] and [[a,b], [c,d]]
# is used the nested Vector Schema to do this
class Move_Schema(Schema):
    from_vector = fields.Nested(Vector_Schema, required=True)
    movement_vector = fields.Nested(Vector_Schema, required=True)

    @pre_load
    def pre_load(self, vector_tuple, **kwargs):
        # print({"vector_tuple": vector_tuple})
        from_vector, movement_vector = vector_tuple
        data = {}
        data["from_vector"] = from_vector
        data["movement_vector"] = movement_vector
        return data

    @post_dump
    def post_dump(self, data, **kwargs):
        return (
            data["from_vector"],
            data["movement_vector"]
        )

# this method deserializes a move string into a pair of Vectors stored in a
# dictionary
def get_deserialized_move(move):
    # assert len(move) == 4
    # if move == "None":
    if move == None:
```

```

        return None

move = move[1:-1]
move = move.split(", ")
from_vector = Vector(
    i=move[0],
    j=move[1]
)
movement_vector = Vector(
    i=move[2],
    j=move[3]
)
return dict(
    from_vector=from_vector,
    movement_vector=movement_vector
)

# this method converts a pair of vectors stored in a dictionary to a string
# representation of the move
def get_serialised_move(move):
    if move is None:
        # return "None"
        return None

    return str((
        move["from_vector"]["i"],
        move["from_vector"]["j"],
        move["movement_vector"]["i"],
        move["movement_vector"]["j"],
    ))
}

# this class uses the move schema and the above methods to serialize and
# deserialize a move so that it can be stored in a database
class Minimax_Cache_Item_Schema(Schema):
    board_state_hash = fields.String(required=True, load_only=True)
    depth = fields.Integer(required=True)
    score = fields.Integer(required=True)
    move = fields.Nested(Move_Schema, required=True, allow_none=True)

    @post_load
    def post_load(self, data, **kwargs):
        # assert isinstance(data["depth"], int)
        # assert isinstance(data["score"], int)
        # print(f"post_load: serializing move from {data['move']} to
{get_serialised_move(data['move'])}")

        return Minimax_Cache_Item(
            board_state_hash=data["board_state_hash"],

```

```

        depth=data[ "depth"],
        score=data[ "score"],
        move=get_serialised_move(
            data[ "move"]
        )
    )

@pre_dump
def pre_dump(self, minimax_cache_item: Minimax_Cache_Item, **kwargs):
    # print(f"pre_dump: deserializing move from {minimax_cache_item.move}")
    to {get_deserialized_move(minimax_cache_item.move)}")
    return dict(
        depth=minimax_cache_item.depth,
        move=get_deserialized_move(
            minimax_cache_item.move
        ),
        score=minimax_cache_item.score
    )

# this object can be used to serialize and deserialize moves to be stored in
the database
# it is the final product of this file and the object that will be exported
minimax_cache_item_schema = Minimax_Cache_Item_Schema()

```

schemas\socket_schemas.py

```

from marshmallow import Schema, fields, pre_load, pre_dump, post_load,
post_dump
from chess_functions import Vector, Piece, PIECE_TYPES

# different schema to serialize and deserialize a vector
class Vector_Schema(Schema):
    i = fields.Integer(required=True)
    j = fields.Integer(required=True)

    @post_load
    def get_vector_from_internal_data(self, internal_data, **kwargs):
        # print({"internal_data": internal_data})
        return Vector(**internal_data)

    @pre_load
    def get_internal_data_from_list(self, list_data, **kwargs):
        return {"i": list_data[0], "j": list_data[1]}

    @post_dump
    def make_list_from_internal_data(self, internal_data, **kwargs):
        return [internal_data['i'], internal_data['j']]

```

```
letter_symbol_pairs = (
    ("P", "♟"),
    ("K", "♚"),
    ("Q", "♛"),
    ("R", "♜"),
    ("B", "♝"),
    ("N", "♞"),
)

# these functions get the symbol or letter associated with each piece by
# searching the above tuple
def get_symbol_from_letter(target_letter):
    for letter, symbol in letter_symbol_pairs:
        if letter == target_letter.strip().upper():
            return symbol
    raise ValueError(f"letter '{target_letter}' not found")

def get_letter_from_symbol(target_symbol):
    for letter, symbol in letter_symbol_pairs:
        if symbol == target_symbol.strip():
            return letter
    raise ValueError(f"symbol '{target_symbol}' not found")

# this schema serialised and deserializes between the Pieces object and format
# use in JSON sent to the client
class Piece_Schema(Schema):
    color = fields.String(required=True)
    letter = fields.String(required=True)

    @pre_load
    def pre_load(self, data, **kwargs):
        color, symbol = data
        letter = get_letter_from_symbol(symbol)
        return {
            "color": color,
            "letter": letter,
        }

    @post_load
    def post_load(self, data, **kwargs):
        color, letter = data["color"], data["letter"]
        Piece_Type = PIECE_TYPES[letter]
        return Piece_Type(color)

    @pre_dump
    def pre_dump(self, piece: Piece, **kwargs):
        color, letter = piece.symbol()
        return {
```

```
        "color": color,
        "letter": letter,
    }

@post_dump
def post_dump(self, data, **kwargs):
    color, letter = data["color"], data["letter"]
    symbol = get_symbol_from_letter(letter)
    return [color, symbol]

# these function are utility function that can map a function across a 1d or
# 2d array, transforming all the elements
def list_map(some_function, array):
    return list(map(some_function, array))

def two_d_list_map(some_function, two_d_array):
    return list_map(
        lambda one_d_array: list_map(
            some_function,
            one_d_array
        ),
        two_d_array
    )

# create objects form the schemas with which to serialize and deserialize
vector_schema = Vector_Schema()
piece_schema = Piece_Schema()

# here are functions that use thee schemas to serialize and deserialize key
# data points to and from a dictionary (JSON like) format
def serialize_legal_moves(legal_moves):
    return two_d_list_map(vector_schema.dump, legal_moves)

def deserialize_legal_moves(legal_moves):
    return two_d_list_map(vector_schema.load, legal_moves)

def serialize_move(move):
    return list_map(vector_schema.dump, move)

def deserialize_move(move):
    return list_map(vector_schema.load, move)

def serialize_piece(piece):
    if piece is None:
        return [None, None]
```

```
        else:
            return piece_schema.dump(piece)

def deserialize_piece(piece):
    # print(piece, end=";   ")
    if piece == [None, None]:
        return None
    else:
        return piece_schema.load(piece)

def serialize_pieces_matrix(pieces_matrix):
    return two_d_list_map(serialize_piece, pieces_matrix)

def deserialize_pieces_matrix(pieces_matrix):
    # print("deserialize_pieces_matrix   called")
    return two_d_list_map(deserialize_piece, pieces_matrix)
```

website__init__.py

```
from .flask_server import app
```

website\flask_server.py

```
# import external modules
import flask
import os
from flask_socketio import SocketIO
import datetime
import pickle

# import local modules
from assorted import safe_hash
from database import save_game, get_saved_game, persistent_DB_engine,
create_session, end_session
from chess_game import Game_Website
from schemas import serialize_legal_moves, deserialize_move,
serialize_pieces_matrix, serialize_piece

# fix tabs don't exist on reload bug:
# https://stackoverflow.com/questions/44531360/flask-blogging-error-table-doesnt-exist-tables-not-being-created

# establish various important directories in variables
basedir = os.getcwd()
```

```
static_folder_path = os.path.join(basedir, 'website', 'static')
template_folder_path = os.path.join(basedir, 'website', 'templates')
key_file_path = os.path.join(basedir, 'website', 'secret_key.key')

# retrieve the secret key
with open(key_file_path, "r") as file:
    secret_key = file.read()

# sanitize secret key to remove white space
secret_key = secret_key\
    .replace(" ", "")\
    .replace("\n", "")\
    .replace(chr(13), "")

# create an app object and configure it
app = flask.Flask(
    __name__,
    static_folder=static_folder_path,
    template_folder=template_folder_path,
)
# app.config['SEND_FILE_MAX_AGE_DEFAULT'] = 0
app.secret_key = secret_key

# create the socketio object (bound to the app object)
# socketio = SocketIO(app, async_mode=None)
# socketio = SocketIO(app, async_mode="gevent")
socketio = SocketIO(app, async_mode=None, threaded=True, echo=False)

@app.route("/")
def index():
    # this is the only http endpoint,
    # it renders the appropriate html file

    response = flask.make_response(
        flask.render_template("chess_game.html")
    )

    # it then adds a cookie to the response that contains a cookie key
    # corresponding to the game in the session
    expires = datetime.datetime.now() + datetime.timedelta(hours=48)

    cookie_key = flask.session["cookie_key"]
    cookie_formatted = str(cookie_key).encode("utf-8")
```

```
response.set_cookie(  
    "chess_game_cookie_key",  
    cookie_formatted,  
    expires=expires  
)  
  
# print(f"index: cookie created: {cookie_key}")  
  
# the response is returned, creating the cookie and providing the html  
file for the webpage  
return response  
  
# the below functions both get and save a Game_Website object to the flask  
session as a binary blob  
# this prevent issues that can occur when python objects are saved directly in  
the flask session  
def get_game_in_session() -> Game_Website:  
    # print(repr(flask.session["game"]))  
    # print(repr(pickle.loads(flask.session["game"])))  
    return pickle.loads(  
        flask.session["game"]  
)  
  
def set_game_in_session(game: Game_Website):  
    flask.session["game"] = pickle.dumps(  
        game  
)  
  
# def get_game_in_session() -> Game_Website:  
#     return flask.session["game"]  
  
# def set_game_in_session(game: Game_Website):  
#     flask.session["game"] = game  
  
  
# this handler function is responsible for initializing a flask session  
def create_flask_session():  
    # if flask.session.get("session_setup_complete"):  
    #     return None  
    # else:  
    #     flask.session["session_setup_complete"] = True  
  
    # it first checks the cookies  
  
    # print("Creating session")
```

```
# check cookies
# game_hash_cookie = flask.request.cookies.get("chess_game_hash")
game_cookie = flask.request.cookies.get("chess_game_cookie_key")

# print(f"In creating session, cookie chess_game_cookie_key fetched, it
contains: {game_cookie}")

# if the appropriate cookie doesn't exist, a random number is used as the
cookie key,
# and a fresh game is created
if game_cookie is None:
    print("Cookie was none / doesn't exist")
    game = Game_Website()
    cookie_key = safe_hash(os.urandom(32))

# if a cookie id was recovered
else:

    # the id is converted into a python sting that contains hexadecimal
    characters
    hex_string = game_cookie.encode("utf-8").hex()
    cookie_key = bytes.fromhex(hex_string).decode('utf-8')
    print(f"flask cookie contained hash: {cookie_key}")


    # a database session is created
    session_DB = create_session(persistent_DB_engine)

    # the game object is retrieved from the database
    # database_lookup_result: Game_Website =
get_saved_game(game_hash=game_hash, session=session)
    database_lookup_result: Game_Website =
get_saved_game(cookie_key=cookie_key, session=session_DB)
    # print("database_lookup_result")
    # print(database_lookup_result)

    # the session is disposed of
    end_session(session_DB)
    session_DB = None

    # if the game was found in the database, use it, else create a new
game
    if database_lookup_result is None:
        print(f"game: NOT successfully read from database (returned
None)")
        game = Game_Website()
    else:
        print(f"game successfully read from database")
        game = database_lookup_result
```

```
        print(f"game with hash {hash(game)} successfully read from
database")

    # now that a game object and a cookie_key have been determined, store
these in the session object
    # the index http route will create the appropriate cookie using this id

    # print(repr(game))
    set_game_in_session(game)
    # print(f"At the end of create_flask_session, setting cookie_key to
: {cookie_key}")
    flask.session["cookie_key"] = cookie_key

    # print(f"At the end of create_flask_session, game used has
ID: {hash(game)}")

# this handler function is responsible for clearing up a flask session
# this involves saving the game to the database
def close_flask_session():
    # if flask.session.get("session_setup_complete"):
    #     flask.session["session_setup_complete"] = False
    # else:
    #     return None

    # print("Closing session")
    # game: Game_Website = get_game_in_session()

    # get the game from the database
    game: Game_Website = pickle.loads(
        flask.session["game"]
    )

    # if the window was closed part way though the computer move
    # quickly decide the computers move (time=0) so that game can be resorted
on the user's turn
    if game.board_state.next_to_go == "B":
        game.implement_computer_move_and_report(0)

    # print("close_flask_session, game in session is: ")
    # game.board_state.print_board()

    # game = session["game"]

    # get the cookie id,
    cookie_key = flask.session["cookie_key"]
```

```
# make a database session and save the game to the database using the
cookie id
session_DB = create_session(persistent_DB_engine)

save_game(game=game, cookie_key=cookie_key, session=session_DB)

end_session(session_DB)
session_DB = None

@app.before_request
def handle_create_session(*args, **kwargs):
    # this runs the handler function to create a session before the first
    request (using the session created flag)
    if not flask.session.get("session_created"):
        flask.session["session_created"] = True
        # print("before_request, session:")
        # print(flask.session)
        return create_flask_session()

# unfortunately I needed to use 2 separate events to fully close the flask
# session
# the on socket disconnect event allows me to access the session data but not
# change it
# the on http request stop_and_save_game event runs after the session data is
# deleted,
# it allow me to change the session data and set the session created flag to
# false

@socketio.on("disconnect")
def handle_close_session(*args, **kwargs):
    # this runs the handler to dispose of the session when the socket
    connection breaks
    close_flask_session()

@app.route('/stop_and_save_game')
def handle_close_session(*args, **kwargs):
    # this function sets the flag to indicate that the session needs to be
    reinitialized
    flask.session["session_created"] = False
    return "session closed"
```

```
# here I have created a socket handler function
# it is a decorator as it takes a function as a parameter and returns a
modified function with additional functionality
def bind_socket_handler(event_name, respond=True):
    def decorator(function):
        request_event = f"{event_name}_request"
        response_event = f"{event_name}_response"

        # new function to be returned takes incoming payload as an argument
        def wrapper(incoming_payload: dict | None = None):
            # print(f"Handling event: {request_event}")

            # if there was an incoming payload, this is given to the original
            function as an argument after being deserialized
            if incoming_payload is None:
                result = function()
            else:
                # print({"incoming_payload": incoming_payload})
                if not isinstance(incoming_payload, dict):
                    raise TypeError(f"incoming payload of unexpected
type: {type(incoming_payload)}")

                result = function(incoming_payload)

            # if the respond flag is true, the return value of the function is
            send back to the client
            if respond:
                outgoing_payload: dict = result
                # print(f"Sending response payload by event {response_event}")
                # print({"outgoing_payload": outgoing_payload})

                if not isinstance(outgoing_payload, dict):
                    raise TypeError(f"outgoing payload of unexpected
type: {type(outgoing_payload)}")

                socketio.emit(response_event, outgoing_payload)

        # the wrapper function name is set to the name of the original
        function
        wrapper.__name__= function.__name__

        # bind wrapper function to happen when request corresponding to the
        event is received
        # use decorator in manual way to not overwrite wrapper
        socketio.on(request_event)(wrapper)
```

```
# return wrapper for any further use
return wrapper

return decorator

# the below function accesses the game object in the session
# it returns a dictionary of serialized data about this game object
def generate_game_update_data():
    game: Game_Website = get_game_in_session()

    # collect various data point from the game object and serialize as
    necessary
    next_to_go = game.board_state.next_to_go
    difficulty = game.difficulty

    legal_moves = list(game.board_state.generate_legal_moves())
    legal_moves_serialised = serialize_legal_moves(legal_moves)

    pieces_matrix = game.board_state.pieces_matrix
    pieces_matrix_serialised = serialize_pieces_matrix(pieces_matrix)

    next_to_go_in_check = game.board_state.color_in_check()

    over, winning_player, victory_classification = game.check_game_over()

    game_over_data = {
        "over": over,
        "winning_player": winning_player,
        "victory_classification": victory_classification,
    }

    pieces_missing = {}
    for color in ("B", "W"):
        pieces_missing[color] = list(map(
            serialize_piece,
            game.board_state.generate_pieces_taken_by_color(color)
        ))

    move_history = game.move_history_output

    # this is the format of the outgoing payload,
    # flask will convert it to JSON automatically
    # this payload contains all the data that the client needs to be updated
    # about the game state
    payload = {
        "difficulty": difficulty,
        "next_to_go": next_to_go,
```

```
"game_over_data": game_over_data,
"legal_moves": legal_moves_serialised,
"pieces_matrix": pieces_matrix_serialised,
"check": next_to_go_in_check,
"pieces_taken": pieces_missing,
"move_history": move_history,
}
return payload

# using my custom socket decorator greatly simplifies the process of using
# sockets

# on request get update from the client, return the generic payload of game
# data
@bind_socket_handler("get_update")
def get_update():
    """Sends all data from game object to client to update chess game
    This data is also updated after the user move and after the computer
    move"""
    result = generate_game_update_data()
    # print(f"Update payload for game with
hash {hash(get_game_in_session())}")
    # print(result)
    return result

# on request to implement the computer move
@bind_socket_handler("implement_computer_move")
def implement_computer_move():

    # get the game and validate that the computer can go
    game = get_game_in_session()
    assert game.board_state.next_to_go == "B"
    assert not game.board_state.is_game_over_for_next_to_go()[0]

    # generate the move and a dictionary of data about it for the client
    # print("generating move")
    move_description = game.implement_computer_move_and_report()

    # update the session with the new mutated game object
    set_game_in_session(game)

    # print("after computer move: game in session is:")
    # get_game_in_session().board_state.print_board()
```

```
# generate a generic update payload and add teh move description before
returning
    outgoing_payload = generate_game_update_data()
    outgoing_payload[ "computer_move_description" ] = move_description

# print(outgoing_payload)

return outgoing_payload


# when a request comes to reset the game
# (no response needed)
@bind_socket_handler("reset_game", respond=False)
def reset_game():
    # preserve the difficulty setting
    old_game = get_game_in_session()
    # difficulty preserved
    difficulty = old_game.difficulty

    # create a new game object with the old difficulty and update the session
    new_game = Game_Website(difficulty=difficulty)

    set_game_in_session(new_game)

    # return generate_game_update_data()

# takes input from client about the user's move
@bind_socket_handler("implement_user_move")
def implement_user_move(incoming_payload):
    game = get_game_in_session()

    # validate that the user can move
    assert game.board_state.next_to_go == "W"
    assert not game.board_state.is_game_over_for_next_to_go()[0]

    # print("python function called: implement_user_move")
    # deserialize the user's move into vectors
    user_move = tuple(
        deserialize_move(
            incoming_payload[ "user_move" ]
        )
    )
    # implement the move on the game object
    game.implement_user_move(user_move)

    # print("python function finished: implement_user_move")
    # update the session with the new mutated game object
    set_game_in_session(game)
```

```
# print("after user move: game in session is:")
# get_game_in_session().board_state.print_board()

# generate and return an update payload
return generate_game_update_data()

# on request to change difficulty
@bind_socket_handler("change_difficulty", respond=False)
def change_difficulty(incoming_payload):
    game = get_game_in_session()

    # get the new difficulty from the payload
    new_difficulty = incoming_payload["new_difficulty"]
    # print(f"Changing difficulty to {new_difficulty}")

    # mutate the game object and save it to the session
    game.difficulty = new_difficulty

    set_game_in_session(game)

    # doesn't return an update
    # return generate_game_update_data()

# this code below was used to generate the initial game data json file that I
no longer use
# before the reload game feature, this prevented the client needing to request
the initial game data immediately on loading

# app = None
# if __name__ == "__main__":
#     import json
#     game: Game_Website = Game_Website()

#     next_to_go = game.board_state.next_to_go
#     difficulty = game.difficulty

#     legal_moves = list(game.board_state.generate_legal_moves())
#     legal_moves_serialised = serialize_legal_moves(legal_moves)

#     pieces_matrix = game.board_state.pieces_matrix
#     pieces_matrix_serialised = serialize_pieces_matrix(pieces_matrix)

#     next_to_go_in_check = game.board_state.color_in_check()

#     over, winning_player, victory_classification = game.check_game_over()
```

```

#     game_over_data = {
#         "over": over,
#         "winning_player": winning_player,
#         "victory_classification": victory_classification,
#     }

#     pieces_missing = {}
#     for color in ("B", "W"):
#         pieces_missing[color] = list(map(
#             serialize_piece,
#             game.board_state.generate_pieces_taken_by_color(color)
#         ))

#     move_history = game.move_history_output

#     payload = {
#         "difficulty": difficulty,
#         "next_to_go": next_to_go,
#         "game_over_data": game_over_data,
#         "legal_moves": legal_moves_serialised,
#         "pieces_matrix": pieces_matrix_serialised,
#         "check": next_to_go_in_check,
#         "pieces_taken": pieces_missing,
#         "move_history": move_history,
#     }

#     with open("./website/static/initial_game_data.json", "w") as file:
#         file.write(
#             json.dumps(payload)
#     )

```

website\templates\chess_game.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chess Game</title>

```

```
<script  
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js"  
    integrity="sha512-  
9mpsATI0KClwt+xVZfbcf21J8IFBAwsubJ6mI3rtULwyM3fBmQFzj0It4tGqxLOGQwGfJdk/G+fANn  
xfq9/cew=="  
    crossorigin="anonymous" referrerpolicy="no-referrer"></script>  
  
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-  
js/4.1.1/crypto-js.min.js"  
    integrity="sha512-  
E8QSvWZ0eCLGk4km3hxSsNmGwbLsSCSUcewDQPQWF6pEU8G1T8a5fF32wO11i8ftdMhssTrF/OhyG  
WwonTcXA=="  
    crossorigin="anonymous" referrerpolicy="no-referrer"></script>  
  
<!-- <script type="application/json" src="{{ url_for('static',  
filename='initial_game_data.json') }}></script> -->  
    <script type="text/javascript" src="{{ url_for('static',  
filename='initial_game_data.js') }}></script>  
  
    <script type="text/javascript" src="{{ url_for('static',  
filename='vector.js') }}></script>  
    <script type="text/javascript" src="{{ url_for('static',  
filename='chess_class.js') }}></script>  
  
  
<link rel="stylesheet" href="{{ url_for('static',  
filename='style.css') }}>  
    <link rel="shortcut icon" href="{{ url_for('static',  
filename='favicon.ico') }}" type="image/x-icon">  
  
</head>  
<body>  
<div class="centered">  
  
    <h1 id="top_title"></h1>  
    <!-- <span class="spacer" style="height: 1vmin;"></span> -->  
    <span class="spacer"></span>  
    <div id="board"></div>  
  
    <!-- <span class="spacer" style="height: 1vmin;"></span> -->  
    <span class="spacer"></span>  
    <div class="option_button_container">  
        <!-- <button class="option_button" id="concede_button">Concede  
Game</button> -->  
            <button class="option_button" id="concede_button">Concede</button>  
            <!-- <button class="option_button" id="restart_button">Restart  
Game</button> -->  
            <button class="option_button" id="restart_button">Restart</button>
```

```
</div>

<!-- <div class="spacer"></div> -->
<span class="spacer"></span>
<span class="spacer"></span>

<div class="difficulty_form">
    <h1>Change Difficulty: (AI thinking time)</h1>
    <!-- <p>Change Difficulty:</p> -->
    <form id="difficulty_form">
        <label>
            <input type="radio" name="difficulty" value="really_easy"
onclick="handle_radio_button_click(this)">
                Trivial (1 sec)
        </label>
        <label>
            <input type="radio" name="difficulty" value="easy"
onclick="handle_radio_button_click(this)">
                Easy (2 sec)
        </label>
        <label>
            <input type="radio" name="difficulty" value="medium"
onclick="handle_radio_button_click(this)">
                Medium (5 sec)
        </label>

        <br>

        <label>
            <input type="radio" name="difficulty" value="hard"
onclick="handle_radio_button_click(this)">
                Hard (10 sec)
        </label>
        <label>
            <input type="radio" name="difficulty" value="really_hard"
onclick="handle_radio_button_click(this)">
                Challenge (15 sec)
        </label>
        <label>
            <input type="radio" name="difficulty" value="extreme"
onclick="handle_radio_button_click(this)">
                Extreme (30 sec)
        </label>

        <br>

        <label>
```

```
        <input type="radio" name="difficulty" value="legendary"
    onclick="handle_radio_button_click(this)">
            Legendary (60 sec)
        </label>
    </form>
</div>

<!-- &lt;div class="spacer"&gt;&lt;/div&gt; --&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;

&lt;div class="pieces_taken_table"&gt;
    &lt;h1&gt;Pieces Taken:&lt;/h1&gt;
    &lt;table&gt;
        &lt;tr&gt;
            &lt;td&gt;White Pieces Taken&lt;/td&gt;
            &lt;td&gt;Black Pieces Taken&lt;/td&gt;
        &lt;/tr&gt;
        &lt;tr&gt;
            &lt;th id="num_pieces_taken_white"&gt;&lt;/th&gt;
            &lt;th id="num_pieces_taken_black"&gt;&lt;/th&gt;
        &lt;/tr&gt;
        &lt;tr&gt;
            &lt;th id="which_pieces_taken_white"&gt;-&lt;/th&gt;
            &lt;th id="which_pieces_taken_black"&gt;-&lt;/th&gt;
        &lt;/tr&gt;
    &lt;/table&gt;
&lt;/div&gt;

<!-- &lt;div class="spacer"&gt;&lt;/div&gt; --&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;
&lt;span class="spacer"&gt;&lt;/span&gt;

&lt;div class="previous_moves_table"&gt;
    &lt;h1&gt;Moves History:&lt;/h1&gt;
    &lt;table id="pieces_taken_table_tag"&gt;

        &lt;tr&gt;
            &lt;td&gt;
                White previous moves:
            &lt;/td&gt;
            &lt;td&gt;
                Black previous moves:
            &lt;/td&gt;
        &lt;/tr&gt;

        &lt;!-- &lt;tr&gt;
            &lt;td&gt;</pre>
```

```

                white move
            </td>
            <td>
                black move
            </td>
        </tr> -->

    </table>
</div>
<span class="spacer"></span>
<span class="spacer"></span>

</div>
<script type="text/javascript" src="{{ url_for('static',
filename='main.js') }}></script>
</body>
</html>
```

[website\static\main.js](#)

```

// these constants define the colors of text, squares and text shadow
// throughout the program
const white_sq_bg_color = '#f5e6bf';
// const white_sq_bg_color = '#d9cba7';
const black_sq_bg_color = '#66443a';
const white_piece_color = '#ffffff';
const black_piece_color = '#000000';
// const white_shadow = '-1px -1px 0 #000, 1px -1px 0 #000, -1px 1px 0 #000,
1px 1px 0 #000'
const black_shadow = '-1px -1px 0 #fff, 1px -1px 0 #fff, -1px 1px 0 #fff, 1px
1px 0 #fff'
const white_shadow = '-2px -2px 0 #000, 2px -2px 0 #000, -2px 2px 0 #000, 2px
2px 0 #000'
// const black_shadow = '-2px -2px 0 #fff, 2px -2px 0 #fff, -2px 2px 0 #fff,
2px 2px 0 #fff'

// this function creates a series of div tags that represent the individual
// squares in the chess board
// it decides their color, position vector and hence their ID and click event
// functions
function create_board_widget() {
    let board = document.getElementById("board");
    for (let row = 0; row <= 7; row++) {
        for (let col = 0; col <= 7; col++) {
            let i = col;
            let j = 7-row;
```

```

        // if row and column are both odd or both even then white, so add
        // together and check if even
        let sum_is_even = ((row + col) % 2 == 0);
        // console.log(`square (${i},${j}), sum is even -- ${sum_is_even}`)
        let square_bg_color = sum_is_even ? white_sq_bg_color : black_sq_bg_color;
        let square = document.createElement("div");
        // square.textContent = `(${i},${j})`;
        square.id = `square_${i}${j}`;
        square.classList.add("square");
        square.style.backgroundColor = square_bg_color;
        square.addEventListener("click", function(){handle_square_click(i,j)})
        board.append(square)
    }
}
}

// this function returns the html element of a square given a position vector
function get_square_at_vector(v) {
    let [i, j] = v;
    let id = `square_${i}${j}`;
    let square = document.getElementById(id);
    return square;
}

// this function uses the pieces matrix from the board object to populate the
// board and its squares with pieces
// it must decide their color and text shadow as well
function add_pieces(board) {
    let pieces_matrix = board.pieces_matrix
    // iterate through rows and columns
    for (let row = 0; row <= 7; row++) {
        for (let col = 0; col <= 7; col++) {
            // get the vector and decide the color
            let [i, j] = [col, 7-row]
            let [color_char, symbol] = pieces_matrix[row][col];
            let color = (color_char == "W") ? white_piece_color : black_piece_color;

            // console.log(`get_square_at_vector([${i}, ${j}]) --- ${get_square_at_vector([i, j])}`)
            square = get_square_at_vector([i, j]);
            square.innerText = `(${i}, ${j})`;
        }
    }
}

```

```
// square.style.fontSize = "20px"
square.innerText = symbol;
square.style.color = color;
square.style.textShadow = (color_char == "W") ? white_shadow :
black_shadow;
square.style.fontSize = "9vmin"

}
}

// this function uses the highlighted_squares method of the board object to
add highlighting to the chess board
// this highlights clicked pieces red and squares they can move to green
function add_highlighting(board) {
    let new_highlighting = board.get_highlighted_squares()
    // console.log(`add_highlighting function called with
${new_highlighting}`)

    // iterate through the position vectors and colors form the
get_highlighted_squares method
    for (let i in new_highlighting) {
        let [vector, color] = new_highlighting[i];

        // console.log(`Adding ${color} to square at vector ${vector}`);
        square = get_square_at_vector(vector);
        // console.log({ vector })
        // console.log({ square })
        // if (square.innerText == null) {

            // if a square should be highlighted but is empty: add a center dot
            if (square.innerText.trim() == "") {
                square.innerText = ".";
                // square.style.fontSize = "20vmin";
                square.style.fontSize = "10vmin";
            }
            square.style.color = color;
            // square.style.backgroundColor = color;
        }
    }

    // this function was not needed in the end
    // if wipes the board of all pieces and highlighting
    function clear_board() {
        for (let i = 0; i <= 7; i++) {
            for (let j = 0; j <= 7; j++) {
                square = get_square_at_vector([i, j]);
            }
        }
    }
}
```

```

        // console.log(`Square at vector (${[i, j]}) is ${square}`);
        square.textContent = "";
        // square.innerText = "";
        square.style.color = "black";
        square.style.textShadow = "";
        square.style.fontSize = "9vmin"
    }
}

// this function updates the main title widget with a new title based on the
// board object
function update_main_title(board, manual_new_title=null) {

    if (manual_new_title !== null) {
        document.getElementById("top_title").innerText = manual_new_title
        return null;
    }

    let title_msg

    if (board.just_conceded) {
        // console.log({board})
        // console.log(JSON.stringify(board))
        // console.log(board.just_conceded)
        title_msg = "Game Over: You Conceded"
    }
    else {
        title_msg = `${(board.next_to_go === "W") ? "Your Go:" : "Computer's
Go... (please wait)"}${(board.check) ? "    (CHECK)" : ""}`
    }

    document.getElementById("top_title").innerText = title_msg
}

// this function adds the pieces taken data (number and specific pieces) to
// the table
function update_pieces_taken(board) {

    let num_B_taken = board.pieces_taken.B.length
    let num_W_taken = board.pieces_taken.W.length
    let B_pieces_conjugation = (num_B_taken == 1) ? "Piece": "Pieces"
    let W_pieces_conjugation = (num_W_taken == 1) ? "Piece": "Pieces"

    document.getElementById("num_pieces_taken_black").innerText = (num_B_taken
> 0) ? `${num_B_taken} ${B_pieces_conjugation} Taken` : "No Pieces Taken";
}

```

```
document.getElementById("num_pieces_taken_white").innerText = (num_W_taken > 0) ? `${num_W_taken} ${W_pieces_conjugation} Taken` : "No Pieces Taken";

    // document.getElementById("which_pieces_taken_black").innerText =
    (board.pieces_taken.B.length > 0) ? `Pieces Lost: ${board.pieces_taken.B.map(x => x[1]).join(", ")}` : "-";
    document.getElementById("which_pieces_taken_black").innerText =
    (board.pieces_taken.B.length > 0) ? `${board.pieces_taken.B.map(x => x[1]).join(", ")}` : "-";
    // document.getElementById("which_pieces_taken_white").innerText =
    (board.pieces_taken.W.length > 0) ? `Pieces Lost: ${board.pieces_taken.W.map(x => x[1]).join(", ")}` : "-";
    document.getElementById("which_pieces_taken_white").innerText =
    (board.pieces_taken.W.length > 0) ? `${board.pieces_taken.W.map(x => x[1]).join(", ")}` : "-";
}

// this function auto selects one of the difficulty radio buttons
function set_selected_difficulty(board) {
    let difficulty = board.difficulty
    // console.log({difficulty})

    document.querySelector(`input[value=${difficulty}]`).checked = true;
    // document.querySelector(`input[value="${difficulty}"]`).checked = true;
    // document.querySelector(`input[value=${difficulty}]`).checked = true;
}

// this function adds moves to the move history table
function set_widget_move_history(board) {
    let moves = board.move_history.length;

    let rows
    let move_history_output = Array.from(board.move_history)
    // depending on the number of moves made in the game, extra cells may need to be added to make the table look right
    if (moves == 0) {
        rows = 1;
        move_history_output.push("-")
        move_history_output.push("-")
    }
    else if (moves % 2 == 0) {
        rows = moves / 2
    }
    else {
        rows = (moves + 1) / 2
        move_history_output.push("-")
    }
}
```

```
// console.log({half_moves_rounded_up});
// have a row for each number in half_moves_rounded_up

// delete old elements from the table
let old_rows =
Array.from(document.querySelectorAll('.temporary_previous_moves_table_item'));

if (old_rows.length > 0) {
    for (let i in old_rows) {
        old_rows[i].removeChild(old_rows[i].firstChild);
        old_rows[i].removeChild(old_rows[i].firstChild);
        old_rows[i].parentNode.removeChild(old_rows[i]);
    }
}

// iterate through the rows and add new TD tags to add the new row to the
table
table = document.getElementById("pieces_taken_table_table_tag")
// console.log({table})

let white_move
let black_move
let new_row
let cells
for (let r=0; r<rows; r++) {
    white_move = move_history_output[2*r];
    black_move = move_history_output[1+ 2*r];

    new_row = document.createElement("tr")
    cells = [
        document.createElement("td"),
        document.createElement("td")
    ]

    cells[0].innerText = white_move;
    cells[1].innerText = black_move;

    new_row.classList.add("temporary_previous_moves_table_item")
    cells[0].classList.add("small_text")
    cells[1].classList.add("small_text")

    new_row.appendChild(cells[0]);
    new_row.appendChild(cells[1]);

    table.appendChild(new_row)
}
```

```
}

// gets the hash of a given piece of data
// https://stackoverflow.com/questions/54701686/matching-cryptojs-sha256-with-
// hashlib-sha256-for-a-json
function get_hash_of_data(data, stringify = true) {
    let data_string
    if (stringify) {
        data_string = JSON.stringify(data)
    }
    else {
        data_string = data
    }

    return CryptoJS.SHA256(data_string).toString(CryptoJS.enc.Base64);
}

// holds a table that keep track of the hashes of various items of data
// includes data item name as a key and current hash + function to find hash
as values
previous_data_hashes = {
    pieces_taken: [null, (board) => get_hash_of_data(board.pieces_taken)],
    move_history: [null, (board) => get_hash_of_data(board.move_history)],
    piece_layout: [null, (board) => get_hash_of_data([board.pieces_matrix,
board.possible_to_vectors, board.selected_from_vector])],
    highlighting: [null, (board) =>
get_hash_of_data([board.possible_to_vectors, board.selected_from_vector])],
    difficulty: [null, (board) => get_hash_of_data(board.difficulty,
stringify=false)]
};

// this function only runs the DOM update method given if the data to display
changed
function update_as_necessary(board, update_function, hashes_table_key) {
    // create a new hash
    // only if it is different from the old hash, then update the board and
the hash in the data previous_data_hashes object

    // console.log(previous_data_hashes[hashes_table_key])
    let [old_hash, compute_hash] = previous_data_hashes[hashes_table_key];
    let new_hash = compute_hash(board);

    if (new_hash !== old_hash) {
        // console.log(`hashes are different, updating ${hashes_table_key}`)
        previous_data_hashes[hashes_table_key][0] = new_hash
        update_function(board)
    }
}
```

```
// else {
//     console.log(`hashes the same, NOT updating ${hashes_table_key}`)
// }
}

// update all dom widgets as necessary, always update title
function update_board_widget(board) {
    // clear_board();
    // console.log("board.pieces_matrix")
    // console.log(board.pieces_matrix)

    // add_pieces(board);
    // add_highlighting(board);
    // update_main_title(board);
    // set_selected_difficulty(board);
    // update_pieces_taken(board);
    // set_widget_move_history(board);

    update_main_title(board);
    update_as_necessary(board, add_pieces, "piece_layout")
    update_as_necessary(board, add_highlighting, "highlighting")
    update_as_necessary(board, set_selected_difficulty, "difficulty")
    update_as_necessary(board, update_pieces_taken, "pieces_taken")
    update_as_necessary(board, set_widget_move_history, "move_history")
}

// calls teh appropriate square click function on the board and then updates
// the DOM
function handle_square_click(i, j) {

    // board is a global variable
    // console.log(`Square at (${i}, ${j}) has been clicked`);
    board.handle_square_click([i, j]);
    // console.log(board.pieces_matrix);

    // update_board_widget(board);

    // alert("about to clear board");
    // clear_board();
}

// whole file loads after html
```

```

// create chess board html element
create_board_widget();

// create board class
let board = new Chess_Board(
    update_board_widget,
    update_main_title
);

// bind buttons to methods of the board class
document.getElementById("restart_button").addEventListener("click", function
() {
    board.reset_game()
})
document.getElementById("concede_button").addEventListener("click", function
() {
    // update_main_title(null, "You Conceded The Game")
    board.concede_game()
})

// bind a change to the radio buttons to a handler method in the board class
function handle_radio_button_click(radio_button) {
    let new_difficulty = radio_button.value;
    if (new_difficulty !== board.difficulty) {
        // console.log(`Setting new difficulty to ${new_difficulty}`)
        board.change_difficulty(new_difficulty);
    }
}

// before letting the window close, send the server a warning so it can save
// the game
// also produces a pop up box to prevent accidental closing of the game
window.onbeforeunload = () => fetch('/stop_and_save_game');

```

website\static\chess_class.js

```

// utility functions for use in the program

function arrays_are_equal(a1, a2) {
    return JSON.stringify(a1) == JSON.stringify(a2)
}

function two_d_array_contains_sub_array(two_d_array, sub_array) {
    for (let i in two_d_array) {
        if (arrays_are_equal(two_d_array[i], sub_array)) {
            return true;
        }
    }
}

```

```
        }
        return false;
    }

    function assert(condition, message) {
        if (!condition) {
            message = (message === null? "Assertion error": message);
            throw new Error(message);
        }
    }

// socket object to manage socket connection
let socket = io();

// this is a factory function that takes an event and returns an asynchronous
function
// the function will send a request to the server and then await and return
the server's resonance
function async_function_factory_send_and_receive(event_name) {
    // returns an async function
    async function external_get_response(outgoing_payload = null) {

        // this promise resolves then a response from the server is received,
        the response is given
        const server_response_promise = new Promise(function (resolve) {
            socket.on(` ${event_name}_response`, function (data) {
                resolve(data);
            })
        });

        // the request is sent to the server, along with an outgoing payload
        if appropriate
            if (outgoing_payload === null) {
                socket.emit(` ${event_name}_request`);
            }
            else {
                socket.emit(
                    ` ${event_name}_request` ,
                    outgoing_payload
                );
            }
        }

        // the server's response is then awaited
        let response_payload = await server_response_promise
        // console.log({ response_payload })
    }
}
```

```
// if the response contains any actual data then it is returned
var is_null = (response_payload === null)
var is_empty_sting = (response_payload === "")
var is_empty_dict = (Object.keys(response_payload).length === 0)

if (is_null || is_empty_sting || is_empty_dict) {
    return null;
}
else {
    return response_payload
}

// the async function with this behaviour is returned to be reused
}
return external_get_response
}

// define async functions to make requests to server

// simpler functions that don't need to send and data to the server
external_get_update = async_function_factory_send_and_receive("get_update");
external_implement_computer_move_and_update =
async_function_factory_send_and_receive("implement_computer_move")
external_reset_game = async_function_factory_send_and_receive("reset_game")

// more complex functions that must crate an outgoing payload and send it to
the server, then return the response
async function external_implement_user_move_and_update(from_vector, to_vector)
{
    movement_vector = subtract_vectors(to_vector, from_vector);
    outgoing_payload = {
        "user_move": [from_vector, movement_vector],
    };

    data_exchange_handler_function =
    async_function_factory_send_and_receive("implement_user_move")
    server_data = await data_exchange_handler_function(outgoing_payload);

    return server_data;
}

async function external_change_difficulty(new_difficulty) {
    // console.log(`sending server new difficulty of ${new_difficulty}`)
    outgoing_payload = { "new_difficulty": new_difficulty }
    external_reset_game =
    async_function_factory_send_and_receive("reset_game")
    data_exchange_handler_function =
    async_function_factory_send_and_receive("change_difficulty")
```

```
    await data_exchange_handler_function(outgoing_payload);
}

// this class contains all the data and behaviour of the client side chess
game
class Chess_Board {
    // before reload feature, constructor just used initial game data
    // constructor(update_board_widget_function) {
    //     this.update_board_widget = function () {
    //         update_board_widget_function(this)
    //     }
    //
    //     this.update_from_server_data(INITIAL_GAME_DATA)
    //     this.just_reset = true
    //     this.just_conceded = false
    // };

    // new constructor sets dom manipulation as properties
    // then makes a request to the server to determine it properties
    constructor(update_board_widget_function, update_main_title_widget) {
        this.update_board_widget = function () {
            update_board_widget_function(this)
        }
        this.update_main_title_widget = function (msg) {
            update_main_title_widget(this, msg)
        }

        // this.update_from_server_data(INITIAL_GAME_DATA)
        this.direct_server_grand_update()

    };
}

// this updates the board with the server's data but also sets reset flags
to false
async direct_server_grand_update() {
    let server_data = await external_get_update();
    this.update_from_server_data(server_data);
    this.just_reset = false
    this.just_conceded = false

    assert(this.next_to_go !== "B")

}

// this function takes server data and unpacks it,
// the properties of the board object are then updated with this new
server data
// the board widget is then redrawn
```

```
update_from_server_data(server_data, user_input_disabled = false) {
    this.possible_to_vectors = [];
    this.selected_from_vector = null;
    this.user_input_disabled = user_input_disabled

    this.pieces_matrix = server_data.pieces_matrix;
    this.legal_moves = server_data.legal_moves;
    this.next_to_go = server_data.next_to_go;
    this.check = server_data.check;
    this.just_reset = false

    // console.log("server_data")
    // console.log(server_data)

    // console.log("server_data.game_over_data")
    // console.log(server_data.game_over_data)

    this.game_over = server_data.game_over_data.over;
    delete server_data.game_over_data.over;
    this.over_data = server_data.game_over_data;

    this.pieces_taken = server_data.pieces_taken;
    this.move_history = server_data.move_history;
    this.difficulty = server_data.difficulty;

    this.update_board_widget();

}

// these 3 functions are run when certain buttons are clicked

// changes difficulty property and conveys the change to the server
change_difficulty(new_difficulty) {
    this.difficulty = new_difficulty
    external_change_difficulty(new_difficulty);
}

// simply disables the game, the only option now it to restart
concede_game() {
    this.just_conceded = true
    this.user_input_disabled = true;

    this.over = true;
    this.highlighted_squares = []
    this.selected_from_vector = null
    this.update_board_widget()
}
```

```
// this function resets the board properties to the initial game data and
then informs the server of this change
reset_game() {
    let temp_difficulty = this.difficulty;
    this.update_from_server_data(INITIAL_GAME_DATA);
    this.difficulty = temp_difficulty;
    this.just_reset = true
    this.just_conceded = false
    this.update_board_widget()
    external_reset_game();
}

// this function handles implementing a user move
// it validates that the user can go
// it then updates board's properties with the new board data from the
server
async make_user_move(from_vector, to_vector) {
    assert(
        this.next_to_go == "W",
        "User must be next to go in order to implement a user move"
    )
    // console.log("make_user_move called")
    let server_data = await
external_implement_user_move_and_update(from_vector, to_vector);
    // console.log("updating the board with this server data")
    // console.log({server_data})
    this.update_from_server_data(server_data, true)

}

// this function implements a computer move
async make_computer_move() {
    console.log("make_computer_move called")

    // validates that it is the computer's turn
    assert(
        this.next_to_go == "B",
        "User must be next to go in order to implement a user move"
    )

    // created a promise to wait some amount of time (1 sec)
    const wait_before_displaying.promise = new Promise((resolve) =>
setTimeout(resolve, 1000))
    // const wait_a_second.promise = new Promise((resolve) =>
setTimeout(resolve, 0))

    // gets the new server data after computer move
    let server_data = await external_implement_computer_move_and_update();
```

```
// console.log({server_data})

// get and remove move special data
let move_description = server_data["computer_move_description"];
delete server_data["computer_move_description"];
// console.log({ move_description })

// move description unpacked
let [from_vector, movement_vector] = move_description.move
let resultant_vector = add_vectors(from_vector, movement_vector)

// no change made for a second to the user can see the board after
their move
await wait_before_displayingPromise

// if the game is reset or conceded in that time, abort the function
if (this.just_reset || this.just_conceded) {
    return null
}

// the relevant highlighting is applied to show the computer's move
this.selected_from_vector = from_vector
this.possible_to_vectors = [resultant_vector,]
this.update_board_widget()

// the move is highlighted for 0.8 seconds before the new board state
is shown
const wait_to_show_move_highlights = new Promise((resolve) =>
setTimeout(resolve, 800));
await wait_to_show_move_highlights

// again, abort function if game reset / conceded in that time
if (this.just_reset || this.just_conceded) {
    return null
}

// console.log(`Implementing computer move:`)
// console.log(JSON.stringify(move_description))
// console.log(`Updating board to:`)
// console.log(JSON.stringify(server_data.pieces_matrix))

// show the new board positions after the computer's move
this.update_from_server_data(server_data, true);

// console.log("move_description.taken_piece")
// console.log(move_description.taken_piece)
```

```
// if (arrays_are_equal(move_description.taken_piece, [null, null])) {
//     alert(`Computer moved ${move_description.moved_piece[1]} from
${move_description.from_square} to ${move_description.to_square}`)
// }
// else {
//     alert(`Computer moved ${move_description.moved_piece[1]} from
${move_description.from_square} to ${move_description.to_square} taking
${move_description.taken_piece}`)
// }

}

// this function choreographs implementing the user move and then the
computer move
// if checks if the game is over after each one
async user_move_and_computer_move_cycle(from_vector, to_vector) {
    await this.make_user_move(from_vector, to_vector)

    if (this.game_over) {
        this.handle_game_over()
        return null
    }

    // if (this.check) {
    //     alert("CHECK");
    // }

    await this.make_computer_move()

    if (this.game_over) {
        this.handle_game_over()
        return null
    }

    // if (this.check) {
    //     alert("CHECK");
    // }

    // the user can now access the board to input another move
    this.user_input_disabled = false

}

// this function displays the appropriate output when the game is over
handle_game_over() {
    // make sure the game is over
    assert(this.game_over)
```

```
let winning_player = this.over_data.winning_player;
let victory_classification = this.over_data.victory_classification;

// manually set the title message to show this
let msg;
switch (victory_classification) {
    case "checkmate":
        msg = (winning_player == 1) ? "Checkmate: congratulations, you
won!" : "Checkmate: you lost, better luck next time"
        break
    case "stalemate board repeat":
        msg = "Stalemate: Threefold Repetition (Draw)"
        break
    case "stalemate no legal moves":
        msg = "Stalemate (Draw)"
        break
    default:
        throw new error("Invalid victory classification")
}

// update the title message
this.update_main_title_widget(msg);

// the board remains disabled

// alert(msg)

// this.reset_game();
}

// this handler function responds to the user clicking a certain square
handle_square_click(vector) {

    // if user input is disabled, ignore the click
    if (this.user_input_disabled) {
        // alert("board disabled")
        // console.log("board disabled")
        return null;
    }

    // else disable the board and decide what to do
    this.user_input_disabled = true;

    // check if they have clicked a green highlighted square they could
    move to,
```

```

        // let from_square_already_selected = this.selected_from_vector !==
null;
        let is_valid_move =
two_d_array_contains_sub_array(this.possible_to_vectors, vector);
        // console.log(`Checking
if [${vector}] in ${JSON.stringify(this.possible_to_vectors)} --> result
was ${is_valid_move}`);

        // if they have, cause the appropriate user then computer move
if (is_valid_move) {
    // async function call
    // console.log("about to call make_user_move")
    // this.make_user_move(this.selected_from_vector, vector);
    this.user_move_and_computer_move_cycle(this.selected_from_vector,
vector);
}

// else if not valid or no piece already selected then reselect
else {
    // get piece at vector clicked
    let [i, j] = vector;
    let [row, col] = [7 - j, i];
    let [color, _] = this.pieces_matrix[row][col];

    // if the piece is one of the users then highlight red
    // if (color == this.next_to_go) {
    if (color == "W") {
        this.selected_from_vector = vector;
    }
    else {
        this.selected_from_vector = null;
    }

    // then iterate through the legal moves and identify any squares
that the user could move the selected piece to
    // if there are any, highlight green
    if (color == this.next_to_go) {
        let position_vector; let movement_vector; let
resultant_vector;
        this.possible_to_vectors = [];
        for (let i in this.legal_moves) {
            [position_vector, movement_vector] = this.legal_moves[i];
            if (arrays_are_equal(position_vector, vector)) {
                resultant_vector = add_vectors(position_vector,
movement_vector);
                this.possible_to_vectors.push(resultant_vector);
            }
        }
    }
}

```

```
        }
        // update the board to show any highlighting changes
        this.update_board_widget()
    }
    this.user_input_disabled = false
}

// this method returns an array of position vectors of squares and their
color
// it highlights the selected from vector red and all selected to vectors
green
get_highlighted_squares() {
    // returns a 2d array of vector and then color
    let highlighted_squares = [];
    if (this.selected_from_vector !== null) {
        highlighted_squares.push([
            this.selected_from_vector,
            "red"
        ]);
        for (let i in this.possible_to_vectors) {
            highlighted_squares.push([
                this.possible_to_vectors[i],
                "green"
            ]);
        }
    }
    // console.log({highlighted_squares})
    return highlighted_squares;
}
}
```

website\static\vector.js

```
// the following functions execute add or subtract operations on a pair of
// vectors

function add_vectors(v1, v2) {
    // v1 + v2
    return [
        v1[0] + v2[0],
        v1[1] + v2[1],
    ]
}

function subtract_vectors(v1, v2) {
    // v1 - v2
    return [
        v1[0] - v2[0],
    ]
}
```

```
v1[1] - v2[1],  
]  
}  
  
// use arrays_are_equal method instead  
// function vectors_are_equal(v1, v2) {  
//     return JSON.stringify(v1.map(Number)) == JSON.stringify(v2.map(Number))  
// }
```

website\static\initial_game_data.js

website\static\style.css

```
body {  
    width: 100%;  
    height: 100%;  
    margin: 0 auto;  
    text-align: center;  
    font-size: 4vmin;  
}
```

```
#board {
    width: 80vmin;
    height: 80vmin;

    /* width: 800px;
    height: 800px; */

    /* margin-top: 10vmin;
    margin-bottom: 5vmin; */

    /* margin-top: 0;
    margin-bottom: 0; */

    /* margin-top: 2vmin;
    margin-bottom: 2vmin; */

    margin-left: auto;
    margin-right: auto;

    display: flex;
    flex-wrap: wrap;

    border: 2px solid black;
    border-collapse: collapse;
}

.square {
    width: 10vmin;
    height: 10vmin;
    /* width: 100px;
    height: 100px; */
    display: flex;
    align-items: center;
    justify-content: center;
    font-size: 9vmin;
    /* text-shadow: -1px -1px 0 #000, 1px -1px 0 #000, -1px 1px 0 #000, 1px
    1px 0 #000; */
}

#top_title {
    /* height: 8vh; */
    height: 6vmin;
    width: 80vw;
```

```
font-size: 6vmin;

margin: 0 auto;

text-align: center;

/* border: 2px solid red; */

padding-top: 0;
margin-top: 0;
border-top: 0;
/* clip-path: polygon(0 0, 100% 0, 100% 30%, 0 100%); */

}

/* .button_wrapper { */
.button_wrapper {
    display: flex;
    align-items: center;
    justify-content: center;
    flex-direction: column;
}

.option_button_container {
    display: flex;
    justify-content: space-between;
    /* width: 100vmin; */
    width: 60vmin;
    /* margin-top: 2vh; */
}

.option_button {
    /* height: 6vmin;
    width: 8vmin;
    min-height: 4.5ch;
    min-width: 8ch;
    font-size: 5vmin; */

    /* height: 6vmin;
    width: 8vmin;
    min-width: 14ch;
    font-size: 5vmin; */

    height: 6vmin;
    width: 8vmin;
    min-width: 8ch;
    font-size: 4vmin;
}
```

```
text-align: center;
/* font-size: 5vmin; */
/* margin-top: 2vh; */
/* border: 2px solid green; */

/* margin-left: 5vw;
margin-right: 5vw; */
}

.option_button:first-child {
    margin-left: 0;
    /* margin-right: 2ch; */
}
}

.option_button:last-child {
    /* margin-left: 2ch; */
    margin-right: 0;
}

.pieces_taken_table th, td {
    width: 35vw;
    border: 2px solid black;
    font-size: 28px;
}

.spacer {
    display: block;
    /* height: 5vh; */
    /* height: 3vmin; */
    height: 2.5vmin;
    width: 100%;
    margin: 0;
    padding: 0;
    border: none;
    overflow: hidden;
    font-size: 0;
    line-height: 0;
}
/*
.small_text {
    font-size: 4vmin;
} */

#difficulty_form form {
```

```
display: inline-flex;
flex-direction: row;
}

.difficulty_form div{
/* width: 80vh; */

width: 80vmin;
/* width: 100vmin; */

/* background-color: orange; */
/* border: 2px solid orange; */
}

.difficulty_form label {
margin-right: 6vw;
width: 10vh;
height: 5vh;
/* font-size: 4vh; */
font-size: 4vmin;
/* margin: 5vh, auto; */
}

h1 {
font-size: 4vmin;
padding-top: 0;
padding-bottom: 0;
margin-top: 0;
margin-bottom: 0;
border-top: 0;
border-bottom: 0;
}

#which_pieces_taken_white {
background-color: #66443a;
font-size: 4vmin;
color: white;
text-shadow: -2px -2px 0 #000,
2px -2px 0 #000,
-2px 2px 0 #000,
2px 2px 0 #000;
}

#which_pieces_taken_black {
background-color: #fae09f;
font-size: 4vmin;
color: black;
```

```
/* text-shadow: -1px -1px 0 #ffff,
   1px -1px 0 #ffff,
   -1px 1px 0 #ffff,
   1px 1px 0 #ffff; */

/* body {background-color: red} */
```

database\database.db

Here are some screenshots of some example content of the database:

	primary_key	INTEGER NOT NULL PRIMARY KEY	cookie_key	VARCHAR	raw_game_data	BINARY
1	1		89c9f3e8710d7e3f32...		x'8004951813000000...	
+						

≡ database.db ×

database > ≡ database.db

SELECT * FROM Minimax_Cache

Schema | Custom Query | Auto Reload | Find | Other Tools...

	primary_key INTEGER NOT NULL PRIMARY KEY	board_state_hash VARCHAR	depth INTEGER ↓	score INTEGER	move VARCHAR
1	26	23d95552ac4506858f...	2	40	(1, 7, 1, -2)
2	84	b7f4055a25006733d6...	2	115	(2, 0, 4, 4)
3	127	334c1a34bd745b9541...	2	90	(3, 3, -1, 1)
4	132	7d029641913fe457e9...	2	75	(3, 3, 1, 1)
5	2	fed0eab068456e9031...	1	105	(1, 0, 1, 2)
6	3	8974c934e73bd086aa...	1	110	(1, 0, 1, 2)
7	4	c57af64508b9b89514...	1	105	(1, 0, 1, 2)
8	6	f46dcc35db9c0b1f9d...	1	110	(1, 0, 1, 2)
9	8	87fef8df405322c633...	1	100	(1, 0, 1, 2)
10	9	b5666306846ff9f65c...	1	95	(1, 0, 1, 2)
11	10	f0d8a669d15b49110e...	1	100	(1, 0, 1, 2)
12	11	2437687865aa87a3c4...	1	95	(1, 0, 1, 2)
13	12	02cf3c5e6a2bf8e014...	1	320	(2, 0, 5, 5)
14	13	dc2fe5d0aceaa52da9...	1	145	(2, 0, 5, 5)
15	14	0b635134bd82fe4a89...	1	90	(1, 0, 1, 2)
16	15	454533408f5ee3afba...	1	80	(1, 0, 1, 2)
17	16	8c3790219b4c8a1f5e...	1	70	(1, 0, 1, 2)
18	17	12dd3c300e92a51f5d...	1	50	(1, 0, 1, 2)
19	19	43117fc70bf489f9e4...	1	70	(1, 0, 1, 2)
20	23	47c27f01825c4caa9c...	1	40	(1, 0, 1, 2)
21	25	83dfdca81bc6fb3cef...	1	40	(1, 0, 1, 2)
22	27	df1231213f1bcfd90c...	1	5	(6, 6, 1, -1)
23	28	385c13e87bd725adf9...	1	180	(1, 0, 2, 1)
24	29	5a2a0f7ec6862ea727...	1	-185	(6, 7, 1, -2)
25	30	6aa0dbe1f32e935e24...	1	115	(1, 7, 1, -2)
26	31	04ae16228476e9fff1...	1	100	(0, 6, 0, -1)
27	32	4e7995220143c3e048...	1	100	(0, 6, 0, -1)
28	33	d6e2de19ef15321021...	1	95	(0, 6, 0, -1)
29	34	2e177561306551e17a...	1	90	(0, 6, 0, -1)
30	35	06936fcfa4f4207b19c...	1	70	(0, 6, 0, -1)
31	36	4b9a42d6f640d3b355...	1	90	(1, 7, 1, -2)
32	37	9b93e135095c62a1f0...	1	70	(0, 6, 0, -1)