**Planning and Design stage:**

Sources:

These were sources that I used when I was completing research as part of the design process

Chess as a whole:

* <https://youtu.be/STjW3eH0Cik>

Minimax algorithm:

* <https://youtu.be/5oXyibEgJr0>
* <https://youtu.be/l-hh51ncgDI>

**Research phase:**

The purpose of the this phase is to research the problem and how similar problems have been solved before. This should allow me to learn about the various techniques approaches used by others to solve this problem.

The first piece of research I conducted was watching this MIT lecture (<https://youtu.be/STjW3eH0Cik>) which was specifically about the various approaches to creating a program that can play chess.

The lecture went through the various ways that you might attempt to create a chess program. It mentioned that you could try to emulate how humans play. This would involve analysing the board my various metrics and attempting to describe the arrangement of pieces like pawns or whether or not it was a good time to castle. There is no obvious way to achieve this with computation.

Another method could be to develop a set of if-then rules. For example if is possible to move a pawn to take a piece then do that. Else if it is possible to castle do that ect. This could also be achieves by ranking every possible move be some score in order to decide. However it is unclear how to score moves effectively without looking ahead. This method would be tedious and the result would be a primitive chess playing computer that was unable to play well as it cannot look ahead any number of moves.

Another possible method is to use the British Museum algorithm which uses brute force. It keeps looking ahead following each possible move and each possible move from the resulting game state in a large tree until it reaches a game state that is won or lost. These finished game states correspond to the lead nodes on this huge decision tree. Then it can use this info select the absolute optimal move by considering how both players act rationally and the sequences of moves that cause a win or a loss.

The standard language used is to refer to this tree of moves and resulting game states as a decision tree. The tree has 2 key attributes:

* The branching factor (b) describes on (often on average) how many branches will come of a given node. In this context this means how many possible moves are there from any given game state.
* The depth (d) of the tree which describes how many layers deep the tree is. This represents how many moves ahead the chess engine is examining.

Assuming that the branching factor is consistent the number of nodes in the tree should be approximately b^d.

The lecturer gave a good explanation of why the British Museum algorithm is infeasible. He explained that chess on average have a branching factor of 35 and on in an average game each player may make 50 moves. This puts the total number of nodes at 35^100. This number could potentially be reduced if moves that are unlikely to be good are not considered. This would reduce the branching factor but at the cost of potentially missing some key moves. This puts the number of game states that would need to be analysed at 10^120 to 10^154. For context there are around 10^80 atoms in the universe. For this approach is completely impossible.

The best solution is to change the objective that the computer is trying to achieve to something else that allows it to look less moves ahead. This can be done by giving the computer the problem of aiming to achieve the best board state in X moves judged by some static evaluation function. This means that your tree only has a small depth and that each of the leaf nodes is given a score as to how favourable it is for the computer. In order to maximise the minimum score that the computer is guaranteed to get regardless of the users moves the minimax algorithm is used.

I then used this lecture multiple sources to understand how the minimax algorithm worked.

I used this source to learn more about how minimax worked: <https://www.youtube.com/watch?v=5oXyibEgJr0>

This source was extremely useful for trying to understand alpha beta pruning:

<https://www.youtube.com/watch?v=l-hh51ncgDI>

This source was used to better understand the algorithm and for the diagrams:

<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>

From the various sources I have decided to use the minimax algorithm to determine the best possible move for my chess engine to make.

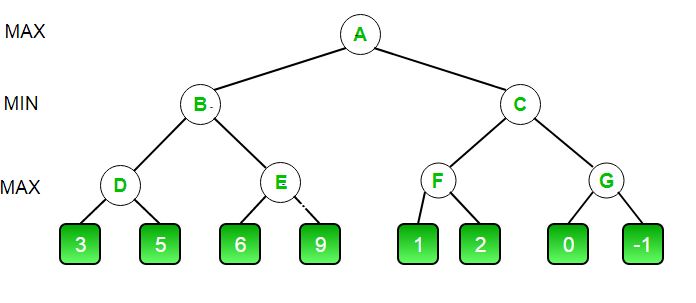
**The way the algorithm works is as follows:**

A function is required to compute a static evaluation of any board. This means it creates a score for how good this board is for a given player. This is done without looking any moves ahead. This could be done by giving each piece a value and adding up the total values of the pieces for each player. For example a pawn could be 1, a queen 9 and a castle 3. The king doesn’t need to be included as both players will always have a king. We could then create a score for how much this board state favours the computer by taking its total score and misusing the users total score. This way the computer is aiming to maximise the score in order to win and the user is aiming to minimise the score in order to win. Note a static evaluation could include other things like is the opposing king in check or a measure of how spread out the pawns are.

A tree that I will call a decision tree can then be created to represent the possible paths of a game in the near future. Nodes represent game states and a decision for one of the players of which of the many legal moves to make. Edges that connect nodes represent moves. A parent node will have many edges to show the many possible legal moves. Each of these then connect to a child node that represents the resulting game state. The game states on the leaf node are then scored using the static evaluation method.

Here is an example tree with a depth of 3 and a branching factor of 2:

(image: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>)



From here on the computer is aiming to determine what is the best possible game state that it can guarantee achieving no matter how the user plays. If the computer is the maximiser (want the highest score) and the user is the minimiser (want the lowest score) the computer will assume that the user will play optimally and act accordingly. This way the computer will be able to guarantee itself a certain score or better if the user plays sub optimally.

Working through this example assuming both players act rationally and optimally:

We need to propagate up the tree level by level to see what each player will pick. For nodes D, E, F, G it is the computer go and it want to play optimally to maximise the score. Therefore for nodes D it will select the move that leads the game state of score 5. In this way node D itself is worth 5 to the computer as it can guarantee a 5 from node D.

By this logic node E is worth 9, node F is worth 2 and node G is worth 0 (still better than -1 which favours the user).

Now the computer will assume that the user will play in an adversarial manner to prevent the computer being able to maximise its score. For example at node B the user want to minimise and so they will select the path that leads to D as they know that E is worth 9 to the computer while D is only worth 5. For C the user will select G as is worst for the computer. The computer then assumes that the user will do this and so to the computer game state B is worth 5 and game state C is worth 0. Therefore the computer will, from game state A select C as it is guaranteed a 5 or better doing this.

Hence the computer will choose B. The user will move and then the computer will re-evaluate. If the user selected E the computer gets 9 if the user selects D correctly the computer gets 5.

It is important to note that I intend to write my own version of minimax with optimisation although the standard minimax and alpha beta pruning algorithm is standardised and widely used like merge sort so I intend to use multiple sources (will be linked) to write the initial minimax algorithm. Minimax is also recursive and while it process can be represented using a tree a abstract data structure for a tree is not required.

In order to improve how the computer plays you must be able to look deeper. There are multiple ways to improve performance which I will outline and then go into detail later.

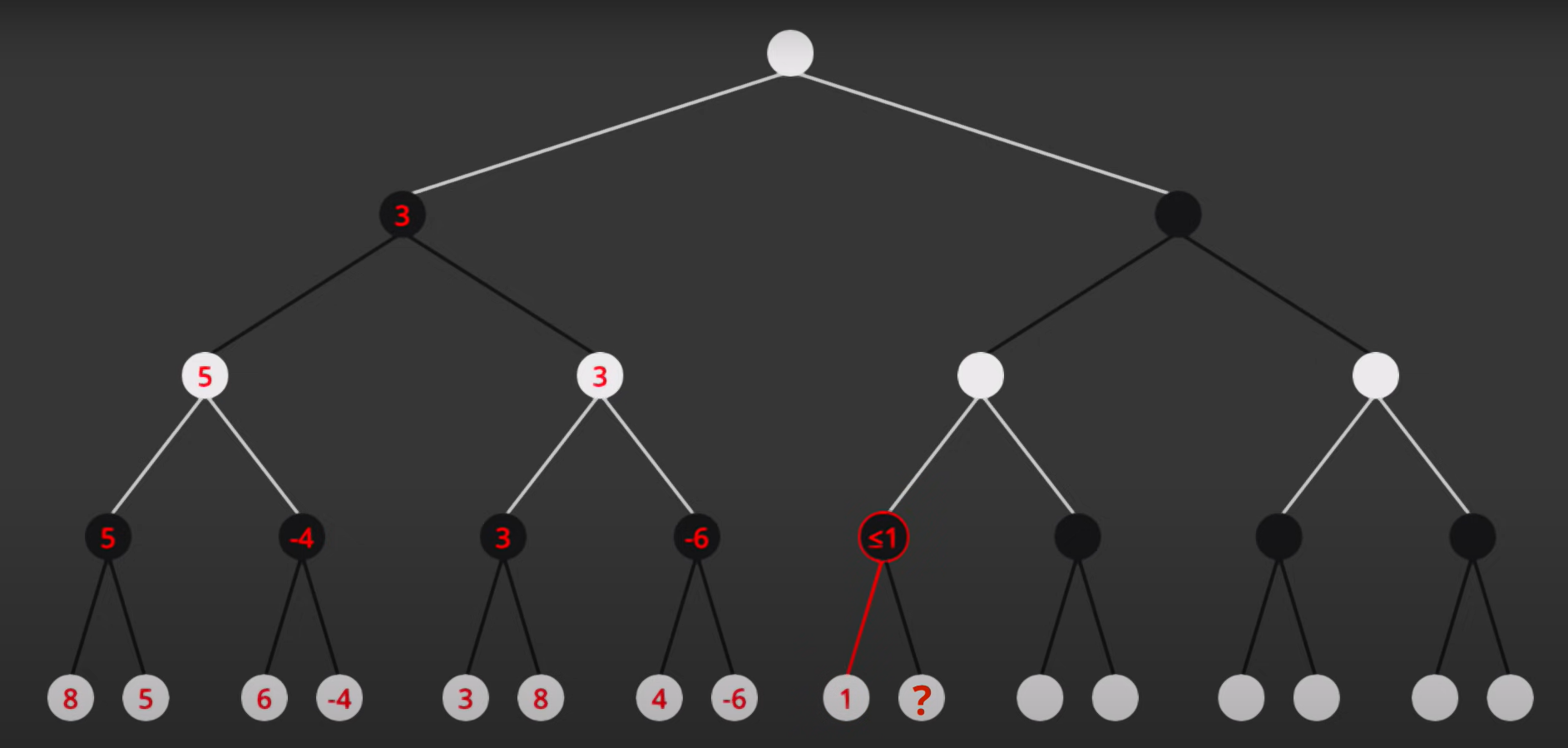
* Use alpha beta pruning. This lets you skip evaluating parts of the tree that are redundant and cannot affect your choice saving time allowing you to search deeper.
* Insurance policy: This allows you to give your algorithm until a given timestamp to complete instead of a certain depth. An insurance policy is created by generating the best move at each depth. This means that if the computer has to stop half way through a search of depth 6 for example it can still use the best answer from depth 5. The cost of this in terms of time efficiency for a search that reaches a depth of N is the time taken to complete a search of depth N-1 (so if the branching factor is 35 is increases time by 1/35)
* Another important consideration is: do I need to consider all legal moves? I originally considered the starting chess board and wondered if I needed to consider both the knight as surely at that point the board is symmetrical so I can forget about one of the nights and 4 of the pawns. However the board is not symmetrical as one side has a king and the other a queen. I could not feasibly statically evaluate moves and only consider say the 15 best. This is because a setup move can pay off in the subsequent move as it might allow a move valuable piece to be taken.
* I also learned form a few of the sources that if you examine all possible moves in the perceived order of best to worst it will improve the performance of the alpha beta pruning. This move ordering could use a static evaluation of the move or use information from lower depth minmax traversals completed as part of the insurance policy.
* Parallelization, if I can complete part of the minimax algorithm where minimax is completed on all the child game states then I can make if more efficient
* Short Term Caching: I can add caching in memory (volatile RAM) to improve time efficiency at the cost of space efficiency. Here is an example of how this may be beneficial:

Say the computer has done a depth 5 search from position A and as part of this is has needed to do a depth 4 search of A, B, C (may not happen due to pruning). If the user moves so that the computer must now complete a depth 5 search form position C then the cached depth 4 search of C means that if can save time as it doesn’t need to make insurance searches at depth 2, 3, and 4

* Asynchronous off time tasks: When the server is not busy processing a request it could be doing something else like completing a move in depth search of frequent game states or completing long term caching operations (hand in hand with below point). This can be done so that while the sever is running minimax if creates a task to add the result to a database so it doesn’t need to be recomputed. This task will then be ignored until there is off time
* Long term caching: I will need a way to represent a game state in the database for various reasons including saved games where the user left of or games stated for puzzle / challenge minigames. Why not add to the database game states that the computer has encountered. A bridging table could attach to them a frequency of encounters and the best possible move, utility score and depth searched. Then if the a better move is determined at a greater depth this can be updated. The most frequent board states (ones near the start of the game) can be recognised by a high frequency and a more in depth minimax search can determine a more optimal move. T ovoid clogging a last encountered timestamp could allow game states with a low frequency and no recent encountered to be deleted from the database. Even if a move detailed minimax search must be run say on game state A with depth 8 where child game states include B, C, D. A depth 8 search of B and a depth 7 search of C in the database could be used. At small depths if would not be worth storing the best option of interrogating the database to see if one is cached (e.g. depth 3 or less).
* The minimax algorithm could be written in a language that is harder to develop in than python but runs faster. For example I could use C or Cython to greatly improve performance
* It is important that static evaluations can be performed quickly to allow for a greater search depth (the slowest part of the algorithm is doing B^D static evaluations)
* It is important that a game state be represented by an abstract data structure that ideally has these properties:
  + If it is a class then determining legal moves of a parent and initialising many new child game static as well as statically evaluating them should be efficient. For this reason a 3d matrix may be more suitable
  + It is important that the object can be serialised into json to be sent to client side JavaScript veer a restful HTTPS API
  + It is important that the object can be converted to a form that can be stored as one or more rows in an SQL database and vice-versa.
  + It should be simple to look for the special case of check (king is attacked) or checkmate (king is attacked and no move is not check). It should also be possible to reduce the number of legal moves to ones that are not check (king could move into up to 9 squares or the attacking pieces could be blocked or taken)

**Alpha beta pruning:**

This is the only optimisation that I intend to add to my first prototype. It allow redundant parts of the tree to not be examined. If done perfectly if could reduce total computation significantly to approximately 2b^(d/2) this requires the order in which moves are examined to be the order from best to worst which we cannot know exactly but we can estimate. This reduce computation to a square root of what it was before for the same depth and allow for double the previous depth to be examined ceteris paribus time being equal.



In the above example (image <https://www.youtube.com/watch?v=l-hh51ncgDI>) a static evaluation can be skipped. The node with a question mark doesn’t need to be evaluated as white is already guaranteed a 3 or better. Therefore as black will only ever allow white 1 or less on the parent note to “?” (marked “<=1”) the node with a question mark doesn’t need to be evaluated. Whatever the value of the node with a question mark, whit can do better. Depending on depth this could save a static evaluation or a whole recursive call to minimax.

An example of how this logic is completed with alpha beta pruning is shown here:

source: <https://www.youtube.com/watch?v=l-hh51ncgDI>

(I included the code from the video for transparency that I had seen it but I don’t intent to copy it directly but I will use it to explain the algorithm as it is very standard pseudocode)

A picture containing graphical user interface

Description automatically generated

Here we see how 3 static evaluations hare been saved by pruning.

(for reference the 5 we need to complete is close to the approximation given by ½ \* B^(D/2) showing that this is a highly optimal example of pruning)

We can see that alpha and beta are arguments to the recursive function. Here is how they work:

Alpha represent the highest score so far guaranteed in any given section of the tree for the maximising player (here white)

Beta represent the lowest score so far guaranteed for the minimising player (here black) which. Beta can be thought of as the best result for white if it is whites turn to move or the worst move for white possible if it is black’s turn to move

Here is the tree after the first static evaluation is performed:

A picture containing shape

Description automatically generated

We can see that whenever the evaluation for a child node is finished the values for alpha if white and beta if black of the parent node are updated. The parent’s alpha beta values are then given to the next child that is being evaluated:

A picture containing diagram

Description automatically generated

Here is an example of pruning:

A picture containing diagram

Description automatically generated

The static evaluation 5 has been computed. Then the alpha value of the node labelled “>=5” (just a label for explanation) is changed to 5 to show that at that node the best the white player can do is 5. At this point there is enough information to see that black will never allow for this path to be taken as rather than give white the greater of 5 or ? (uncommuted static evaluation in grey) it can give white a best case of 3. Therefore it doesn’t matter what the node in grey is. This is shown as alpha is greater than beta. This tells the function to stop examining children of this node “>=5” as it means that this node gives at least a 5 and black only needs to give us a 3 or worse so his branch can be pruned.

A picture containing timeline

Description automatically generated

Similarly we can see above how a branch of the tree has been pruned. This is again because beta <= alpha (the condition is the same regardless of which player it is to move). This shows that white had a better option further up in the tree and that it would never take this path as it favours black more than is necessary (giving black a -4 or better when white could have a 3 or better)

**First Prototype:**

My first prototype should be very simple and won’t fully achieve all the design specifications. It is important however as it will aim to create some of the main mechanisms that will be needed by later versions. It should also be quick to develop so that I don’t loose motivation or bite or more than I can chew (try to do to much of the development at once). For this reason reusing code form other projects is essential. I also want to select a more simple game than chess.

Here is the idea:

I will make a tic tac toe / noughts and crosses game that will allow a user to play against a computer. The user interface will not be the focus of attention and so it doesn’t need to be overly complicated.

Here is my first and then my improved second iteration of the user interface:

Done on paper insert images here

I will implement this 2nd version with HTML and JavaScript with very little in the way of CSS

The current board will be stored as a 3x3 array in JavaScript.

The user will always go first and will always be X

The board widget will be implemented as a table tag with 3 rows and 3 columns, this allows each individual cell to be a separate tag allowing me to see which cell has been clicked.

Validation will be needed when a square is clicked:

* It is impossible to give as input a square that is not one of these 9
* If game over do nothing
* Else if filled do nothing

To make a move this is what will happen (post validation):

* Disable the board (make it ignore further user input)
* Update the 3x3 2d array in js
* Update the board the user sees
* Make a request to the server to identify if the user has won
  + If user has won or the game is a draw do nothing else, leave the board disabled, make a message to the user accordingly
* Else request to the server providing the board currently. The server will respond with the new board after it has moves and weather or not it has won. Moves left are also given and if 0 the game is won by the computer.
  + If user has lost or the game is a draw do nothing else, leave the board disabled, make a message to the user accordingly
* Else update the 3x3 array stored in js
* Update the board widget
* Change the message to show it is the user’s turn
* Re-enable the board widget to allow for the users input

Note how some of these operations like determining is the game is won could easily be done In client side JavaScript. I have chosen not to do this as in chess that is a more complex operation that the backend will handle. It is overkill to have the backend decide if a move is valid so this will be done in the client side JavaScript. In the final thing I intend for the server to give the frontend an array of all the users legal moves after it makes a move to avoid unnecessary api calls but to also avoid logic being done in the front end.

A full duplex bi-directional connection is not required and so a form of WebSocket is not required. Instead a restful API will be used using the HTTPS protocol.

The backend (server side code) will be written in python. It will setup a restful API and host the websites files using a simple flask server.

It would be easy to have the backend memorise a process of steps that would guarantee a win or draw but instead I will use the British Museum algorithm to generate a complete tree of depth 9. Minimax will be used to navigate this tree but no static evaluations are needed as all leaf nodes represent a win (score of 1) a draw (score of 0) or a loss (score of -1).

A tree of depth 9 with a branching factor or 9 decreasing by 1 each time will have a total number of static evaluation of approximately 360,000. However as the user is starting only a depth of 8 is required reducing static evaluation to 40,000.

So to be clear I will need to perform minimax at a depths (d) of: 8, 6, 4 and 2 and the branching factor (b) is decreasing by 1 each time.

This means the total computation (measured by total leaf nodes in each minimax call) will be:

8! + 6! + 4! + 2! = 41066 which is mostly calculating the first move (first move is 40320 steps).

The backend will require these functions:

(note I am not including the actual code to make the HTTPS restful API as this is very library and language specific, instead I am including all the logical functions)

* Determine if a game is over, still going, won or lost
* A representation of the board using a 3x3 2d array
* Minimax recursive function with alpha beta pruning to determine the computers best move
  + A method to determine all legal moves from a given game state (empty squares in array)
  + A method to make a copy of the current game state which has been changed to reflect a move being made

I want to follow the industry standard of test driven development for the chess engine. I will write a unit test for the various internal functions that the program uses: DESIGN

* Test if various boards are correctly determined to be wins, losses draws or unfinished
* The thing that is less obvious is how to test its tic tac toe ability. Here are the options I thought of:`
  + Test that when random inputs are given the program is able to win and test that when it is against a tic tac toe algorithm (meant to represent optimal play) it can draw.
  + I could manually ply a bunch of games with it and ensure it is working then write the array of moves I made down and use it as a test to ensure the program makes the same moves as before

The best option I can think of is to make a function to score each possible legal move in a given game state. Therefore I can ensure that the minimax function always returns a move that is optimal or draws with other equivalent moves (due to symmetry)

I did also consider exploiting the symmetry or tic tac toe in order to reduce the total number of legal moves down to a subset of distinct moves.

For example lets number squares form top to bottom left to right with numbers 1 to 9 (top left is 1, middle is 5, bottom right is 9).

In this game state:

A picture containing shoji

Description automatically generated

Only squares 1 and 2 or equivalent need to be examined. This is because there is rotational, vertical, horizontal and diagonal symmetry.

In this game state:

Calendar

Description automatically generated with medium confidence

Only 1, 4,5,7 and 8 need to be examined due to vertical symmetry. I have decided to ignore this as 8! Is a manageable number for a computer to compute and in chess I cannot take advantage of symmetry at any point due to the king and queen.

Therefor I have the freedom to make the play experience more enjoyable by having the computer take a random move if there are multiple that are equivalent.

I have created a logging module to increase the transparency of my code. It allows me to log to both the console and a log file. It also includes a decorator that I can use to easily augment my functions to add logging to them to show their behaviour. I have optionally turned of the logging to console part to avoid filling the console with noise as I am unable to filter by importance level so that debug and info logs don’t make it to the console, only warning error or critical.

I also invested a small amount of time trying to get web sockets working. I found it difficult as it is less commonly used than alternatives and so the documentation and online troubleshooting is less helpful. In doing so I also realised that I only ever need this communication to happen between the front and backend

My experiment with WebSocket was fruitful as I determined the exact nature of the connection between the front and backend

Diagram

Description automatically generated with medium confidence

This is over the top for noughts and crosses as it is more complex due to all the logic being bone by the backend including validation. This is unnecessary here as the logic it simple enough that it can be done client side. In my chess website I want to ensure that all the more complex logic is the sole responsibility of the backend.

I am still unsure about where to store the information of what the initial board state is. This could be in a json file that the frontend requests of it could be hard coded in variables in the JavaScript. Either way the client side JavaScript will need to know what legal moves the user can make and that they all result in the game continuing. Each of the 9 possible moves the user could try to make (by clicking the square) has one of these properties:

* Is illegal as the square is full
* Results in the game continuing (the only outcome that requires the server to be queried for its move)
* Results in the user winning
* Results in the user loosing
* Results in a draw

This property will be stored for each move in a 3x3 array. It will start of before the users first move as all moves resulting in the game continuing. Then the user will make a move. The client side JavaScript will determine that move is valid so it will be displayed and added to its copy of the board, it will also determine that the game should continue. It will send the server a copy of the board showing the users move. The server will then respond with a new board showing it counter move and a new array of the properties of the 9 tiles that the user could try to move to (as listed above) this would be used for the validation of the users next move.

I have created the webserver and I have it hosting 3 near empty files. A html file with the line ‘hi from html’ a css file that makes the background colour red and a JavaScript file that creates an alert “hello from JavaScript”. Currently everything is loading and working.

I have had some time to think it through and even just the tiny start on this prototype has helped my see what logic should be completed on the frontend and the backend and how to connect them.

My initial idea for the communication between the frontend and the backed has some benefits. For instance it means that the frontend can validate a move without having to know anything about how the game of chess works. However the idea of the frontend being provided with a list of legal moves the user could make upon the computer making its move doesn’t make much sense. For one in chess there could be multiple pieces that can move to a given square and there are all together more legal moves. Looking back I do think that a bidirectional communication would be better that trying to do all the information exchange in one http request

For example I could use some kind of socket setup to have it work this way:

User makes move (trigger / event) 🡪

client asks server if it is valid (could include the from and to square) 🡪 server responds with whether or not it is valid

🡪 client asks if the game is over after the users move 🡪 server responds with details

🡪 client requests servers counter move 🡪 server responds with counter move

(🡪 client asks if game is over 🡪 server responds with details )

(repeat)

Perhaps this could be simplified so that there are less distinct requests and more is done in each request. The obvious alternative is to change my original idea so that more of the logic is done client side (validation of legal moves and determining if game over etcetera). For now I will try to stick with this (trying to imagine what is the best API for chess and not tic tac toe). At least the is game over can be one end point.

After my first attempt where I got stuck I will try again with web sockets but I might use a different library. This website covered a variety of other sources of how to use web sockets (<https://www.fullstackpython.com/websockets.html>)

I looked at this repository and will try to use it to help me get started: <https://github.com/miguelgrinberg/Flask-SocketIO/tree/main/example>

This GitHub code repository is very useful as it provides examples of using web sockets including some more advanced features. It shows how the backend and the frontend can exchange data in json form with web sockets and it includes more complex concepts like sessions, emitting vs broadcasting and rooms. I could use rooms to allow 2 online players to play chess. The example shows jQuery and a sockets library on the frontend and a flask WebSocket backend

So I will retry using WebSocket. The incentive to doing so is it will allow for a lower latency connection as a persistent full duplex direct TCPIP connection is created after a HTTP handshake. This allows for low latency bi directional messages to be shared. Low latency means that I can have the frontend and backend make many requests to each other without performance lags (the alternative would be many https request which is slow or doing it all in one http request).

I have decided it wouldn’t be worth the effort to create a test to see how well it plays nights and crosses. Instead I will play the minimax computer and check that even if I play optimally it cannot loose. I will also not worry about the api connection between front and backend being similar enough to how I intend to do it with chess. What’s important is that I can get it working with a similar approach to the one I intent to use for chess.

I have completed my frontend. It allows the user to go first as X and the server is O. It includes validation and a reset button. All the validation and determining if the game is over is completed by client side JavaScript. I am using WebSocket only to allow the server to communicate its move to the client. The server hosts the website and allows this WebSocket communication but doesn’t yet have the ability to determine the appropriate move. At the moment it waits for me to type in a move before responding with it. I am currently using global variables and some pretty basic functions. This is ok as the first prototype only needs to be functional. These improvements can be made later.

I will now try to create a version of the British Museum algorithm in order to have the computer determine its own moves. It should in theory start by picking a path that guarantees is a draw or better. Then if the user slips up and it has an opportunity to guarantee a win it should be able to win.

I have written my min max function to allow the server to decide the best move. It is working but not picking an optimal move. It can be beaten which shouldn’t be possible. In order to debug it I need to find ways to make the highly complex recursive function more transparent. One way to do this is to start with a half finished game where there are less moves left and run British museum minimax on this. I can use a modified copy of the function in a Jupiter notebook so that it generates a tree during execution. This will allow me to better see the logic error.

I have created a jupyter notebook and began to better understand the problem. I have discovered an issue that is likely not the only issue but needs to be fixed. The is\_game\_over functions in the server side python code and the client side JavaScript is flawed as I forgot to add a check for winning be diagonals. I will now fix this.

I thought that perhaps even after this fix there was still an issue with the server’s game over function. I thought it may be inaccurately scoring game states that are over (incorrectly determining a win or loss). This function appeared to be working when tested for some sample inputs in the notebook. I will now try to see the functions decision making by visualising the decision tree using a library for graphs.

I have tried adding a decorator to the function to add print statements to illustrate when functions are called and when they finish, it is still hard to tell what is happening. From this I noticed that the alpha values were changing in a strange way but the beta values never were. I then checked the code and found this logic error:

            beta = min(alpha, evaluation)

where the first argument to the min function should be beta not alpha. Let me try this fix.

This seems to have been the issue. It now seems to always win or draw when I play it now.

It is all fully working now, I had to modify some of the frontend validation as it was still failing to recognise diagonals but not it works.

**First Prototype Done:**

I have now completed my first prototype. It clearly doesn’t meet all the success criteria and it clearly isn’t chess. I think it was necessary as it is better to do more prototypes with a shorter gap / jump in progress between them. This ensures that motivation is maintained and ensures that I don’t get stuck feeling like I have bitten of more than I can chew.

I used an algorithm to determine the best possible move form a given game state. I called it British Museum minimax. This is because it featured the approach of fully searching the decision tree all the way down to wins and losses (no static evaluations needed). This general approach is called the British Museum algorithm. To then analyse the tree I had constructed efficiently I used a variation of minimax to back propagate for the lead nodes to determine the best move. This is the reason for the name. Modifications on the standard minimax included alpha beta pruning, no depth argument or static evaluations and most importantly the algorithm actually returned both the score of a given game state and the child game state with the best score (gives you the best move as an output to the function).

**Second Prototype:**

I would now like to create a second prototype. It will reuse a minimal amount of the previously created code. I intend to explore new frameworks and features that are technically difficult to use but will be appropriate for the final version. This may include using a JavaScript framework to improve the creation process for the GUI (raw html and js will not be suitable for the final product). I may also use SQL to add memorization of determined moves allowing for more detailed searching later (limited in application until I add puzzles or save games but still useful).

I will aim to solve a problem that is between chess and tic tac toe in complexity. It should force me to perform static evaluations rather than use the British Museum approach.

I am currently considering what this should be. I am thinking of doing something like draughts of a game with only a 6\*6 square and where you start with 12 pawns instead of the full sized board with all its pieces.

I have looked at the rules of draughts and there seem to be special moves and rules. I would prefer to avoid these for now. I will make the game as follows

The board is a 6X6 grid. Each player starts with 6 pawns arranged or the first 3 rows with 3 in each square of their colour in each row. This is similar to draughts but the fewer pieces and smaller board should lead to a lower branching factor. The pawns can move like in normal chess and take other pawns:

* Can move forward 1 or 2 into an empty square
* Can take diagonally

I will not be adding these special rules:

* En Passant: If your opponent’s pawn just moved forward 2 squares into a vacant space your pawn can take it horizontally if it is on the left or right next square
* When a pawn reaches the other side it can be promoted into a queen, bishop, rook, knight

I can later increase the problem complexity be adding in more pieces or special rules.

The objective of the game is to remove all your opponents pieces. If this hasn’t happened in 50 turns (after the black player’s 25th turn) the pieces are counted to determine a winner.

I will implement this in a similar way to tic tac toe. I will use a one page website that will use web sockets to communicate with a python backend.

I will use VUE js to create an interactive component which will manage the user interface and validation. I will again implement the board as a table tag with rows and cells, I should be able to detect when a cell is clicked and which cell it is.

I will use WebSocket as the API to connect the front and backend. The client and server may need to interact in the following ways:

* The client requesting the server’s move and the server responding with it.
* The client updating the server as to a change to the difficulty setting (will be stored in the session object)
* The client notifying the server that the user wishes to end the game
* After 50 moves reached, the server recognises that before or after its move the game is over or if prompted by the client as the user wants to end the game
* The client checking if a move is legal

Some things I want to try include:

* Storing the current state of the game server side in the flask session object so the client only needs to request information or notify changes
* After looking into it I think I could use promises to use web sockets in a way more similar to a http request

Another use for WebSocket:

The attraction of WebSocket is the speed. I don’t need bi-directional communication as my server will only ever respond to the client’s requests. The issue that WebSocket presents is that the server responds user another event. For example the client may emit ‘server\_move\_request’ with come relevant information and then the server will emit ‘server\_move\_responce’ in response. This must be handled by a different handler and so beaks the logical flow of execution into 2 functions. This makes it hard to, for example, have a function that want to check if a move is valid and then do something else like make the move. This is because the function cannot create a promise or call a function that returns the servers response and then continue with execution with that information (like http). For this reason where this property is desired I could try to adapt my use of WebSocket to be more like http.

For example

Async Function request\_legal\_moves():

Socket.emit(‘request\_legal\_moves’, game\_data)

Await socket.on(‘response\_legal\_moves’)

…

// somehow return servers responce

However I would need to experiment with this as it is a bit more complicated. This is because ‘response\_legal\_moves’ is an event that calls a call back function when it occurs and gives the callback function the servers response as an argument.

Maybe I could do this

Async function request\_legal\_moves(callback):

Socket.on(‘response\_legal\_moves’)(callback)

Socket.emit(‘request\_legal\_moves’, some\_game\_data)

I continued to look into this as after experimenting with WebSocket during the tic tac toe prototype I realised that I naturally found it hard to not be able to await a promise to see the server’s response. I kept looking up if there are solutions to this problem and all I could see is websites that I had already seen that state that you should use WebSocket over HTTP when you need continuous updates on something or when you need the 2 way connection (e.g. a chat app needs the server to be able to send the new messages to the client to allow them to be displayed).

I don’t need any of these things. I wanted to use WebSocket because of its low latency. This would allow me to make more requests and allow logic to be done in the backend without noticeable lags on the frontend.

I used python to measure how long I had to wait for a response after a http request.

def get\_latency(url):

total = 0

for \_ in range(5):

start = time()

requests.get(url)

end = time()

total += end-start

mean = total / 5

return mean

get\_latency("https://www.google.com/")

0.14106497764587403

get\_latency("https://www.google.com/")

0.13407492637634277

get\_latency("https://www.google.com/")

0.13257269859313964

get\_latency("https://www.google.com/")

0.13857812881469728

get\_latency('https://www.youtube.com/')

0.2924633026123047

get\_latency('https://www.youtube.com/')

0.31210713386535643

get\_latency('https://www.youtube.com/')

0.3238521099090576

get\_latency('https://edhrec.com/')

0.08779067993164062

get\_latency('https://edhrec.com/')

0.08545541763305664

get\_latency('https://edhrec.com/')

0.08265228271484375

get\_latency('https://stackoverflow.com/')

0.31110215187072754

get\_latency('https://stackoverflow.com/')

0.30382657051086426

get\_latency('https://stackoverflow.com/')

0.30450005531311036

get\_latency('https://en.wikipedia.org/wiki/Main\_Page')

0.11946406364440917

get\_latency('https://en.wikipedia.org/wiki/Main\_Page')

0.11989774703979492

get\_latency('https://en.wikipedia.org/wiki/Main\_Page')

0.11942543983459472

These numbers will include the time taken for the server to process the request but still it shows that on a fast network http is fast enough to manage. I also have decided that I can take steps to reduce unnecessary requests. Ultimately I intend to focus on performance optimisation later on. This should not be the focus of prototypes.

I will now try to sketch the webpage: