CO3015 Computer Science Project

Dissertation

Mostafa Nabi

March 16, 2020

This report covers my efforts to create a chess game as a web application. The report explains my approaches to developing it, the architecture and design of the software as well as the algorithms and data structures used. The software itself is incomplete but most of the core features are working, therefore it is still useful for anyone wishing to see how to make a chess engine as a web app and any obstacles they may face.

Contents

1 Introduction

1.1 About the Project

Chess is a fascinating and old strategy game and ideal for computer programming, as Claude Shannon says in his ground breaking paper on chess:

"The chess machine is an ideal one to start with, since: (1) the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate); (2) it is neither so simple as to be trivial nor too difficult for satisfactory solution; (3) chess is generally considered to require "thinking" for skilful play; a solution of this problem will force us either to admit the possibility of a mechanized thinking or to further restrict our concept of "thinking"; (4) the discrete structure of chess fits well into the digital nature of modern computers. [17]."

As someone who both enjoys playing chess and has always been intrigued by how computers play chess, I decided it would be an ideal project for my dissertation. There are many challenges in creating a computer chess player, such as the data structures used to represent the game state, the algorithms to decide on the next move and techniques for evaluating possible moves. In order to make my application as accessible as possible, I chose to deploy it as a web application. This comes with its own challenges, which have been covered in later sections.

1.2 Aim of the Project

The original aim of this project was to create a desktop chess game for MacOS computers written in the Swift programming language. I wished to develop the game in a development environment I was unfamiliar with. The objectives could be summarised as:

- Creating a reasonably strong chess AI, where reasonably strong means posing a challenge to more serious players.
- Develop the game using technologies not used before in an unfamiliar environment.

After having completed the two player feature of my project, before beginning work on the AI, I chose to recreate my project as a web application. The reason I did not make it as a web application to begin with was because I had experience in web development in the past and wanted to try new technologies. However I realised that in fact, I knew very little about the web technologies. This would still satisfy my original objective to use new technologies since my previous web development experience was nearly exclusively front-end development in pure Javascript.

Motivated by my keen interest for web development I chose to port my project to the web. Another major reason was to make my project more accessible, since only those with access to an apple computer could use my application, which would also have to be downloaded. As result of redoing many parts of my project my objectives changed considerably, partly because of my new interests and partly because of the significant time lost. Most noticeably the focus of my project was no longer solely the AI but also hosting and deploying a web application that the public can use. To summarise my new objectives:

- Creating an adequate chess AI, where adequate means posing a challenge to more casual/amateur players.
- Host and deploy the chess game as a web application.

Hosting and deploying a web application is considerably more difficult than creating a simple UI on a desktop application, as such I had less time to spend on the AI which is why I chose to make a weaker AI in order to focus more the web aspects of my project.

2 Research

In this section I will cover which aspects of my project required research and what sources I used to do it. I will not be covering the actual topics themselves nor why I chose them, that will be covered in their respective sections of each topic.

2.1 Chess

Chess is a game that has existed for a long time and is a common and well known game so it was not necessary to research the game itself. It has also been the focus of much research in Computer Science so there are many comprehensive resources available. Although this does have the downside of making it harder to find quality resources. The main source I used is the chess programming wiki which covers near enough all aspects of chess programming [3]. The wiki does not go into detail on how to implement, rather it aims to explain the concepts. For the most part it was enough for to figure out how to implement the techniques. The aspects of the chess app I needed to research fall under these categories:

- Representation of the chess board.
- The decision making algorithm for selecting the next best move.
- The evaluation of moves.

The board representation is the first thing that I looked into, as it would be a deciding factor in how I would do much of the rest of my project. There were two approaches I considered, the intuitive method was to use a 2D array for the board, but by doing a quick search I came across the Bitboard technique [2].

For the decision making algorithm I chose the mini-max approach, this is the standard approach to two-player zero-sum game games [5]. Another technique is to add the alpha-beta pruning algorithm on top of mini-max. In summary, alpha-beta will ignore possible moves that are known to be worse than the current best one we have, by doing this move searching is significantly sped up [1].

Move evaluation is by far the most difficult aspect of building a chess AI. There are many possibilities to take into account and there really is no one answer. Many of the possible techniques are beyond the scope of my implementation. I chose to focus on two simple techniques which coupled with a reasonable search depth is enough for an adequate AI. The simplest and most important evaluation technique is material evaluation [4]. However with material evaluation alone the AI will not have any indication of mobility, so it is important to couple it with piece-square tables [8].

2.2 Web Deployment

In order to deploy and host my web application I had to first buy a domain and then find a good service to host my server. I initially tried using Amazon Web Services(AWS) and bought my domain from there. However AWS was expensive and did not provide me with the freedom I needed. Instead I opted to rent a virtual computer and install a server myself. The service I chose was DigitalOcean, because they were cheap and gave me full control. I used a tutorial they wrote to guide me through setting up a server with node.js [9].

My chess app is written in c++ and In order to deploy it as a web application I had to wrap it inside a node.js addon. The page of the node.js documentation on c++ addons gives a good guide on how that is possible and what tools can help [15]. One of the tools is node-gyp which is a build tool for compiling node.js addons. the github page for node-gyp provides examples on how to use it [14], on which I based my own addon.

3 Requirements

The requirements will be split into two sections, the requirements for my web application which includes the user interface and that of the chess application itself.

3.1 Web Application Requirements

The requirements of the web application can be split into the user interface which defines how the user should be able to interact with my application, and the functionality that the web application needs to provide.

3.1.1 User Interface Requirements

The UI is the layer between your user and your application, as such it is important to make using it a good experience. Regardless of how good the technical aspect of the application is, if it is frustrating to use it cannot be counted as a success. The appearance of the UI should be minimalistic and uncluttered, displaying only information needed. It would also be ideal to give the user the option to hide/show information is they please. For example the user should be able to hide/show the move logs.

The UI must provide my user with the following options:

- Move chess pieces
- Set difficulty
- Undo last move
- Resign and reset a game
- Build a game from a FEN string
- Export an existing game to a FEN string
- See a log of all moves made
- Navigate to different pages (SinglePlayer, TwoPlayer, About)

A FEN string is a formal chess notation that saves the game state as a string [20]. Some of these requirements are shared by other components of my project, for example the chess app must also support the extraction of the board state into a FEN string, in truth the UI would simply communicate with the chess app to do this. However I felt this requirement was best suited here as it is something done directly by the user. The UI must be compatible with the common modern browser (Firefox, Edge, Chrome and later versions of IE).

3.1.2 Web Functionality Requirements

This section covers the aspects of the web application that the user does not directly interact with, such as the server. The application must be accessible to the public and support multiple users accessing the server simultaneously. The server must be written in node.js and be able to communicate with c++ chess application. The server will communicate with the user's browser using Web Sockets.

3.2 Chess Application Requirements

This section covers the requirements of the chess application itself, so anything to do the logic of the game.

A summary of the requirements is:

- Prevent all illegal moves
- Recognise and allow all legal moves, including special moves such as castling and en-passant.
- Provide an AI with adequate competency the user can play against
- Ensure the wait time for the AI to move is reasonable.
- Allow the user to play with and without an AI
- Support and AI with multiple difficulty settings
- Support and building of a game using FEN and extracting a game into FEN.

I will not explain the basic rules of chess in this paper, if you are unsure of them then please see this link: https://www.chess.com/learn-how-to-play-chess. An AI of 'adequate' competency means that it will provide a challenge for casual or amateur players. A reasonable wait time is an average of below 30 seconds. The difficulty of the AI is determined by the search depth of the Mini-max algorithm, by increasing users will also increase the wait time. The chess app should set an upper and lower bound for the difficulty level.

A summary of features that the chess application could support but are not requirements are:

- Timed game modes
- Ability to play against other AIs
- Special game modes like Blitz chess.

4 Design of the System

This section will cover design of the overall system, it will be split up into two sections, covering the web application and chess application. The focus of this section will not be the implementation of the components, but their relationships and how they are integrated and build up from smaller components.

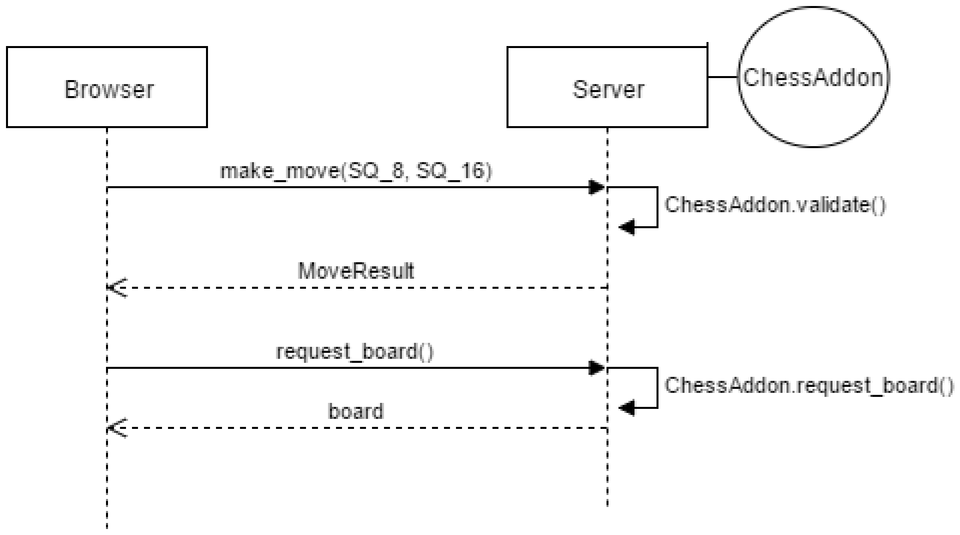
4.1 Design of Web Application

Most of the complexity of the web application is server side, the front end was kept minimal and straightforward. The front end uses the bootstrap framework for its layout and appearance. There is only one HTML page, the contents are hidden/shown using Javascript. This prevents annoying page refreshes. The structure is hard coded in HTML since it will not change and the contents are updated using Javascript. Much of the styling is done using CSS and HTML classes to separate out the styling and structure of the web page. This allows styles to be re-used and easily modifiable.

The web application follows the Model-View-Controller design pattern. The components of the web application are:

- The view which displays the UI.
- The controller which is the client side Javascript that enables user interaction and facilitates communication between the browser and server.
- The model is the server and the chess app that runs in it.

Communication the browser and server is done using WebSockets [13], unlike HTTP requests, WebSockets allow two-way communication to be done between the server and client. This is needed for when the user plays against the AI. It is done by simple message passing which is asynchronous. By passing JSON strings back and forth more it is possible to sent detailed messages easily. This is much more elegant than using pure HTTP in which case I would have to apply the users move then respond with the opponents; by using WebSockets it is possible to apply the users move, respond with the result and then wait for the AI to make a move. A simple communication between the browser would look as follows.



My web server is an Nginx server and my applications server is node.js. One of the great things about node.js is large number of package that are available to use with it, and the quality of the node package manager (npm). I use a number of packages to handle the standard functionality of a server.

- Express.js [10] is a framework for handling HTTP requests.
- node-gyp [14] build tool for building C++ addons for node.js
- nodemon [16] is a process manager tool, it automatically restarts the node server on file changes, which is extremely useful for development.
- ws [19] is a lightweight server side WebSocket implementation.

Node.js allows users to write a package.json file, this is a JSON file specifying dependencies of the project, as well as the versions of those dependencies. This allows me to easily port over the project to different servers and keep track of dependencies. The chess addon is written is c++ and using node-gyp build into a node.js addon. This way node.js can include it like it would include any other package [15].

4.2 Design of Chess Application

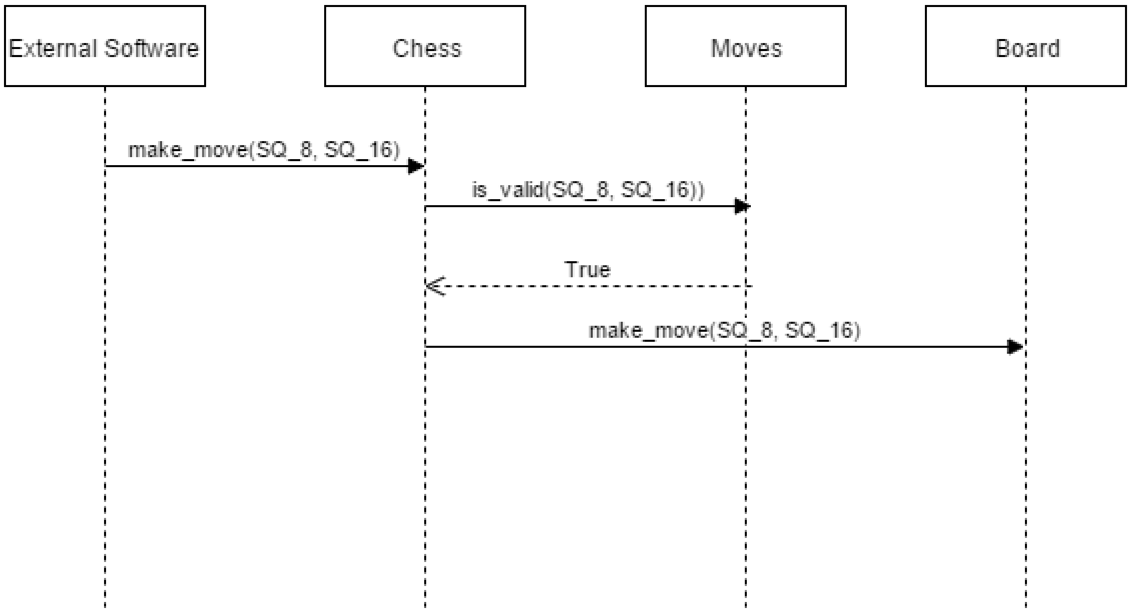
The chess application is written entirely in c++ and has no GUI. Instead it provides an API so that external software can interface with it, this way the chess logic is completely isolated from any UI. It is important to decouple the components within the chess application so that if there are changes in any one component there will be minimal impact in other areas. For example, although my project is a web application it is in-fact possible to use the chess application as it is and create a desktop UI for it. My chess app is split up into the following components:

- Chess, which is the master class

- Evaluation, which is the AI
- Moves, which checks for move legality and generates moves
- Board, which stores the game state
- Bitboard, which is a wrapper around an unsigned 64 bit integer, it provides many functions and operators for doing working with bitboards. See the Bitboard.h file for details.
- Types, this is utility component which things like data types common to the application.

The Chess component is the master class holds all the other components and provides an API for external code to interact with the application. The Evaluation and Move components are not actually classes, rather they are functions placed inside their own namespaces. The reason I have done it this way is because they do not store any state or information, rather they have all the data they need passed to them as function arguments. The Move component is where most the game logic is held. Primarily it performs move validation and calculation on a given reference of the Board. There are times when the board must be modified like in move validation to check if a move results in check, which is why I preferred to separate the game logic and board state. All functions in the Moves component take a constant reference to the Board, so it must create a local copy if it wishes to change the Board. The same applies for the Evaluation component.

The Board class is the class that stores the entire state of the game. There is no game logic in this component. The Board stores the bitboards for all the pieces and provides an interface for other components to modify and request data about the Board state. There is no checking for legality on this level since the user is not expected to interact with the Board component. Instead the user would create an instance of the Chess class and would interact only with that. See the diagram below for an overview of a user might make a move.


External Software
Chess
Moves
Board
make_move(SQ_8, SQ_16)
is_valid(SQ_8, SQ_16))
True
make_move(SQ_8, SQ_16)

5 Board Representation

I chose the bitboard technique for representing the chess board. The bitboard utilises the fact that a chess board contains 64 squares and represents it using a 64 bit integer. Where each square in the chess board is mapped to a single bit in the integer [2]. Since bitboards can only store location information, not the type of piece or its colour it is necessary to have a bitboard for every piece type [Rook, Knight, Bishop, Queen, King and Pawn] for every colour, so a total of twelve bitboards to represent a single chess board. However, because of their generic nature it is also possible to extend bitboards to not just represent the locations of pieces on the board but also the positions they can move to and positions they can attack. Take a look at the following example showing the position and move bitboard of the white queen on an empty board. The right bottom left bit is the first square and top right bit is the last square. Written as a 64-bit binary number the left-most bit is the last square.

```
Position bitboard           Move bitboard
00000000                    00010101
00000000                    00001110
00000100                    11111Q11
00000000                    00001110
00000000                    00010101
00000000                    00100100
00000000                    01000100
00000000                    10000100
```

5.1 Usage

The usage of bitboards seems more complicated than it really is at first, but becomes intuitive with a little practice. The greatest benefit of bitboards is that we can use extremely efficient bitwise operations like the bitwise:

```
AND(&), OR(|), NOR(^)
```

and bitwise shifts to answer many questions on the game state. As an example I will demonstrate two use cases for bitboards, first how to generate a simple move bitboard and then how to see if the king is in check. Note that the examples shown are code snippets and not exhaustive.

5.1.1 Sliding Piece Moves

Except for sliding pieces the move bitboards for most pieces are fairly straightforward. Sliding piece moves are the most difficult and expensive to calculate because they often have the largest possible moves and also because they can be blocked by other pieces so there are more checks that must be performed. Showing how to compute sliding piece bitboards is beyond the scope of this paper, see here:

https://chessprogramming.wikispaces.com/Sliding+Piece+Attacks

To compute the move bitboard for the White King you would just:

1. left-shift by 7,8 and 9 to get the squares in-front
2. left shift by 1 to get the square to the right
3. right shift by 1 to get the square to the left
4. right shift by 7,8,9 to get the squares behind

```
// To get the bitboard for the front squares
// king_move_bb = The resulting move bitboard
// king_pos_bb = The current positon bitboard
uint64 king_move_bb = (king_pos_bb << 7) | (king_pos_b << 8) | (king_pos_bb << 9)

00000    01110
00K00 -> 00000
00000    00000
00000    00000

// You cannot move to squares occupied by your own pieces
king_move_bb = king_move_bb ^ white_squares_bb
```

To check whether the white king is check you would:

1. Computer the move bitboard for every opponent piece and combine them
2. To a bitwise AND with bitboard for you king
3. If the result is greater than 0 then the king is in check

5.2 Drawbacks of Bitboards

While bitboards are a great technique they also have their drawbacks.

- Bitboards are harder to implement and debug then a more straightforward array implementation.
- On 32-bit computers then benefits of using 64-bit integers is reduced as the computations must be split.
- Since the bitboards are just bits, it is more complicated to answer questions such "What piece is on square x". An array could contain objects/enums which would easily answer such a question.

Although I believe the advantages are greater.

- By utilising bitwise operations which can be done in a single CPU cycle many computations are done efficiently.

- Bitboards take up fairly little memory, so it is possible to store many pre-calculated bitboards for possible moves.

6 Utility Data-Structures

The Types.h file contains many utility data structures, functions, constants and enums. It was inspired after looking the stockfish source code and the heavily inspired by them [18]. The application uses enums for modelling the board squares. The bitboard class is a wrapper around a 64-bit number, I have overwritten the default bitwise operators on it to work with the Square enum. I feel this section is better explained by looking at the code directly, see the Types.h file my source code.

7 Decision Algorithm and Evaluation Function

7.1 Minimax

Chess is a well defined game with as such it is easy to model the legal operations in an algorithm. The algorithm I chose is the mini-max algorithm [5]. The mini-max algorithm is used for two player games, where the player the algorithm represents is the 'max' player and the opponent is the 'min' player. The algorithm follows the assumption that each player will always choose the move that is best for them, which in zero-sum games will always be worst for the opponent (hence min and max). An integral component in the algorithm is the evaluation function, this assigns a numerical value to possible next moves the algorithm can choose from.

The pseudo-code below gives an idea of how the max function works. The min function is similar only it chooses the move with the lowest result.

```
function max(board, depth) {
    if(depth == 0 or board.get_moves() == 0)
        return evaluate(board)

    best_result = -INFINITY
    all_moves = board.get_all_moves()

    for move in all_moves
        board.apply_move(move)
        evaluation = min(board)
        if(evaluation > best_result)
            best_result = evaluation
    return best_result
}
```

The function can summarised in the following steps.

1. Basecase: If the depth is reached or no moves are available, return the result.
2. Set the initial result to positive INFINITY
3. For possible you apply it and pass it to the min algorithm.
4. The min algorithm will do the same pass it to max, this continues until the base case is reached.
5. The max function which was called first will return the best result it found.

The minimax algorithm is fairly straightforward and easy to implement. A more elegant solution is the negamax algorithm, which instead of having two functions performs the minimax search in one. By tracking which players move it is it can modify the evaluation result to show from the perspective of the current player. The best result for the max player is equal to the negation of the best result of the min player [6].

```
max(a, b) == -min(-a, -b)
```

Although I implemented the negamax algorithm, I found it less readable and that debugging it was more tricky. I couldn't see any real benefits is efficiency either so I chose to stick with minimax.

7.2 Evaluation

The evaluation function is by far the most complex aspect of the AI. It is fairly straightforward to create a simple one, but to make a comprehensive function is time consuming and requires a great deal of experimentation. I opted to create a simple evaluation function based on material score and piece-square tables. The material score method assigns a numerical value to all piece types and then compares the total value of your pieces and your opponent in order to get an evaluation. A common technique is to subtract the black total from the white total, then multiply the result by 1 if the turn player is white and -1 if they're black.

```
evaluation = current_turn * (white_total - black_total)
```

The material values I used are based on GM Larry Kauffman's research [11].

- pawn 100
- knight 325
- bishop 325
- rook 500
- queen 1000
- king 10,000

When looking ahead at least 3 moves this will prevent the AI from making obvious mistakes that would cause it to lose pieces. However the AI has no notion of positioning and mobility. This can be solved by adding piece-square tables. Piece-square tables are arrays of length 64 that have a one-to-one mapping with the squares on the board. Each piece for every colour has its own table. For the most part the tables for white/black pieces are mirror images of one another. By doing this the AI will be more inclined/discouraged to move pieces to certain squares. For example we can discourage knights from going to the edges and encourage centre pawns to move forward while discouraging pawns that protect the king's castle [12].

It is a good idea to have different values for material score and piece-square tables for different phases of the game. For example while we want to discourage the king from moving in the early game, but in the end game the king becomes a powerful piece and we want it to be active.

8 Testing

The application has an option to build games using the FEN notation, this can be used to test the AI by seeing how to responds to different situations. I ran a number of simple to see whether it could find and avoid checkmates.

8.1 Finding Checkmate

The following FEN string describes a checkmate in one move:

```
rnb1k1nr/pppp1ppp/5q2/2b1p3/4P3/P1N4P/1PPP1PP1/R1BQKBNR b KQkq - 7 4
```

As we can see by pasting it the AI was easily able to find that. This FEN string creates a game state where the AI can checkmate in two moves:

```
7k/6p1/2p3p1/8/4p3/rq6/8/7K b -- 103 52
```

However in this case the AI makes the first move correct but not the second one, this is because the alpha-beta algorithm looks three moves ahead, and because the opponent will always be checkmated, alpha beta prunes the tree too early and simply makes the first move, thinking it will result in checkmate. This results in it making random moves that, while not resulting in a loss, do not result in a gain either. Although I have isolated the bug, I have not had time to fix it.

Excluding bugs like the above, the AI in general can avoid mistakes and take advantage of obvious blunders made by the opponent. I asked some amateurs to play and they struggled against it, but any reasonable player (myself included) would not find it challenging to play against.

8.2 Perft Results

I have compared by perft results to those at https://chessprogramming.wikispaces.com/Perft+Results and based my perft function on theirs [7].

My results were as follows:

- Depth: 1, Moves: 20
- Depth: 2, Moves: 400
- Depth: 3, Moves: 8902
- Depth: 4, Moves: 197281
- Depth: 5, Moves: Runtime too long

My perft results seem to be correct, so I can be fairly confident there are no issues with my move generation. However after 4 moves it becomes far to long to run for it to be usable.

8.3 Server Tests

The webserver was tested by having people try it. I posted the URL on the universities chess society Facebook page. As a result many users accessed at the same time. As a result the following bugs were discovered which I could not discover alone. The chess app is written in c++ and hence not inherently asynchronous, this means the server is stalled so when multiple people access it the have to wait for one another. This is NOT how a website is meant to function. The other bug is more enigmatic, the browser uses WebSockets to connect with the server and if the user is idle for a time the connection is automatically dropped.

As a result the website is unusable by multiple people and users have to refresh the page and resign their games if they wait too long to make a move. Both these issues, while serious flaws in usability shouldn't be too hard to fix. As the issues do not require major changes to any existing implementations.

9 Critical Appraisal

9.1 Analysis of Project

The biggest issue with my project was a lack of time, caused by an overestimation of my own abilities and an underestimation of the difficulty of the tasks involved. My decision to port my existing Swift project over to a web app was done because I did not think it would be too difficult. However I did not realise until after I completed building a web UI and bought and configured a server that Javascript does not support 64-bit bitwise operations. As a result I had to build the app in c++ and wrap into a node.js addon. While I enjoyed the experience and it taught me a lot, it set me back a significant amount of time, especially considering I had never done any non-trivial coding in c++. What I should have done was create a quick and cheap prototype first and only then should I have started porting the project over if it was possible. This all happened in early March or so, more than half way into the year and I basically started the project from scratch. Had I not been too overconfident I would not have risked it, this event highlighted my inexperience in estimating the time and effort required to build software.

As for the quality of my project, I feel the foundations are well built but there are too many bugs and issues. For example my chess application is built well in terms of the architecture and data structures but there is a bug in both the castling of the AI and the evaluation function which causes it to make odd moves. The AI was one of the last components I built, so it was rushed. My server is solid and the communication with the browser is smooth, however as mentioned in the previous section it is unusable to the public. These issues are isolated and not at the core of the software but are still serious problems that have a real impact on the usability of the application. These issues can be fixed without too much difficulty but require time, which I did not have.

I also unable to satisfy numerous requirements, such as having an option to undo, resign and time a game. An accurate description would be that my project is in the beta stage, it has the all the core functionality and is playable but it is definitely not a complete project. However I do not think my decision to make it a web app was wrong, learning experience I gained from created a public website from scratch was well worth it.

9.2 Academic Impact and Personal Development

My project is not suitable for commercial purposes, given it is incomplete and not unique. The aim of the project from the outset was for it to be a learning experience for myself. Initially I simply wished to learn more about algorithms and data structures and felt chess would be an enjoyable and fairly thorough way to learn them. Half way through the project I had come to the decision that I would likely work in web development in the future and decided it would be a good idea to port my project over into a web application. This way my experience would be more relevant to future work and I would be able to easily demonstrate my project.

10 Conclusion

I would say that the aims of the project were not met, the objective was to create a usable online chess application. However, as explained, my project is not complete. The website works but cannot support multiple people, as such it cannot be called a true web application. The aim was to create a fairly weak but functioning AI. However my AI still has bugs in it. Although I believe my project was not a failure either, while it did not meet all its original aims it got pretty close; and the true objective for my project from the very beginning was to learn about algorithms and data structures which I have. While buggy, the minimax algorithm has been implemented and in all honesty, I think I did a pretty good job with the bitboards. I have also gone further and explored modern web technologies with a good amount of success. While my chess app may block my server, wrapping c++ into node.js alone was something I did not know was possible to begin with. In summary, while my project was unsuccessful as a game, it was a success as a learning experience.

Glossary

material The chess pieces, it is common to give each piece a numeric value and use it to evaluate the state of the board.. 3

piece-square tables A table where each position is mapped to a square on the board. Each position is given a numerical value to encourage /discourage a piece to go to it. . 3

zero-sum game A two player game where a win/loss for one player results in the exact opposite for the their opponent.. 3

Bibliography

[1] ChessProgrammingWiki. *Alpha–beta pruning*. URL: https://chessprogramming.wikispaces.com/Alpha-Beta.

[2] ChessProgrammingWiki. *Bitboards*. URL: https://chessprogramming.wikispaces.com/Bitboards.

[3] ChessProgrammingWiki. *ChessProgrammingWiki*. URL: https://chessprogramming.wikispaces.com.

[4] ChessProgrammingWiki. *Material Evaluation*. URL: https://chessprogramming.wikispaces.com/Material.

[5] ChessProgrammingWiki. *Minimax*. URL: https://chessprogramming.wikispaces.com/Minimax.

[6] ChessProgrammingWiki. *Negamax*. URL: https://chessprogramming.wikispaces.com/Negamax.

[7] ChessProgrammingWiki. *Perft*. URL: https://chessprogramming.wikispaces.com/Perft.

[8] ChessProgrammingWiki. *Piece-square tables*. URL: https://chessprogramming.wikispaces.com/Piece-Square+Tables.

[9] DigitalOcean. *Install node.js server*. URL: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-node-js-application-for-production-on-ubuntu-16-04.

[10] Express. *express.js framework*. URL: https://expressjs.com.

[11] Larry Kaufmann. *Centipawn values*. URL: http://chess.wikia.com/wiki/Centipawn#Larry_Kaufman.27s_Research.

[12] Tomasz Michniewski. *Simplified Evaluation Function*. URL: https://chessprogramming.wikispaces.com/Simplified+evaluation+function.

[13] Mozilla. *WebSockets*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[14] node-gyp. *node-gyp build tool*. URL: https://github.com/nodejs/node-gyp.

[15] Node.js. *Node.js: C/C++ addons*. URL: https://nodejs.org/api/addons.html.

[16] remy. *nodemon file monitor*. URL: https://github.com/remy/nodemon.

[17] Claude Shannon. "Programming a Computer for Playing Chess". In: *Philosophical Magazine, Ser.7* 41.314 (1950). URL: http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf.

[18] Stockfish. *Types*. URL: https://github.com/mcostalba/Stockfish/blob/ONE_PLY/src/types.h.

[19] websockets/ws. *node websockets*. URL: https://github.com/websockets/ws.

[20] Wikipedia. *FEN notation*. URL: https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation.