1/1 Send
- Minimax Checkers Report -

Andrew Wang
Connor Byers
Nick Faria
Nick Makharinets

Table of Contents

Introduction

For our SE 101 project, we created a digital checkers board using a 32x32 LED matrix and an Arduino Mega. We first developed a player vs player gamemode by creating the basic checkers board environment on the LED matrix. Then, for the player vs computer gamemode, we developed a minimax algorithm with alpha-beta pruning in order to generate the computer's moves in C++. Due to memory limitations on the Arduino Mega, we chose to run our minimax algorithm on our computers, meaning we had to communicate between the Arduino Serial and the C++ files on our computer.

We chose to develop a functioning player vs player gamemode for our prototype. By having our basic functions ready, like canAttack() which checks whether a checkers piece can attack or movePiece() which "moves" the checkers piece on the LED matrix, we could afford to recycle these functions in the development of our minimax function. Finally, we touched up our project by fixing minor bugs and improving player-interface interactions.

Background Research

Fortunately, there is clear documentation on how to assemble the 32x32 LED Matrix and on its associated library functions. To build our hardware, we followed the tutorial found at [8]. From this link, we were also given the github repositories of the two libraries we needed to include: the RGB matrix panel library at [9] and the Adafruit GFX library at [10].

To gain a basic understanding of the minimax algorithm, we first looked at the public pseudocode on its Wikipedia page: [1]. We also watched various YouTube tutorials on how to implement the algorithm in tic-tac-toe and in other games, since there were no tutorials for checkers specifically.

A good amount of research was also done regarding the class structure and vector implementation in C++ for use with the minimax algorithm and move projection.

Finally, some investigation was done into the amount of memory available on the arduino mega, sourced from [7]. Some research was done into the efficiency of vectors containing object pointers vs vectors containing objects in order to maximize efficiency on the minimax side of things, as a very large amount of data would be pushed through said vectors.

Implementation

Our project was implemented using a combination of software running on the arduino mega and laptop simultaneously. The computationally intensive minimax move generation is processed on the laptop while the arduino handles player input and LED board interactions. This setup allows for fairly smooth execution of the overall task and modularity.

The player vs player gamemode, LED board and button controllers are all housed on the arduino itself. This allowed for simple and effective prototyping in the early stages, as the laptop and inter device interactions were not required. The software contained on the arduino mega can be divided into 3 major components:

- **Rules**, governing what moves the player can and cannot perform in game
- **Moving**, which encapsulates button input and the execution of approved moves
- **Display**, the physical representation of the game state through the LED matrix

This combination of modular components facilitated prototyping and bug fixing by dividing the functionality and workload among all group members.

The rules portion of the Arduino mega's software is based off of the American Checkers Rules [4] with some minor modifications. In order to optimize the available processing power on board and with future minimax algorithm plans in mind, a collective consensus was made to remove mandatory piece taking. This would allow for a greatly reduced computational cost which was later reallocated towards minor optimizations and communication between software. At the core of the rule functionality lie two functions, checkMove and canAttack. These functions receive move commands and return relevant integers to signify the validity of said move.

Following directly, the moving portion of the code is responsible for generating the player desired move coordinates relevant to the button input and if approved by the rules functions, executing said move on the board. Because of the extensive amount of power that the matrix required, it was necessary to wire the buttons as pullup input instead of using the standard method. As standard input, buttons are wired to the 5 volt power pin on the arduino, and pressing them completes the circuit between the power pin and ground. This caused power to be drawn away from the 5 volt pin on the arduino, and in turn did not allow the matrix to display properly. With pullup input, a button press completes the circuit from ground to the digital pin that the button is connected to, not involving the 5 volt pin whatsoever. This allows input to be read without the buttons drawing power away from the 5 volt pin on the arduino, and ensures the matrix has enough power to display properly. A 5 amp power cable that is connected to the matrix from a wall outlet is also used to provide additional power.

The rest of the movement portion plays a supporting role for the rules component. When a move is approved, the movePiece and removePiece functions work in conjunction to append the current gamestate as necessary. On the other hand, if the selected move is not valid, the user will be prompted to select another move, which will in turn pass through the validation and execution process.

Displaying the board on the LED matrix was accomplished by making use of the RGB matrix panel library that was downloaded from the website the arduino was bought from (https://github.com/adafruit/RGB-matrix-Panel). This library is extensively documented and the github includes example programs, so using it was fairly straightforward.

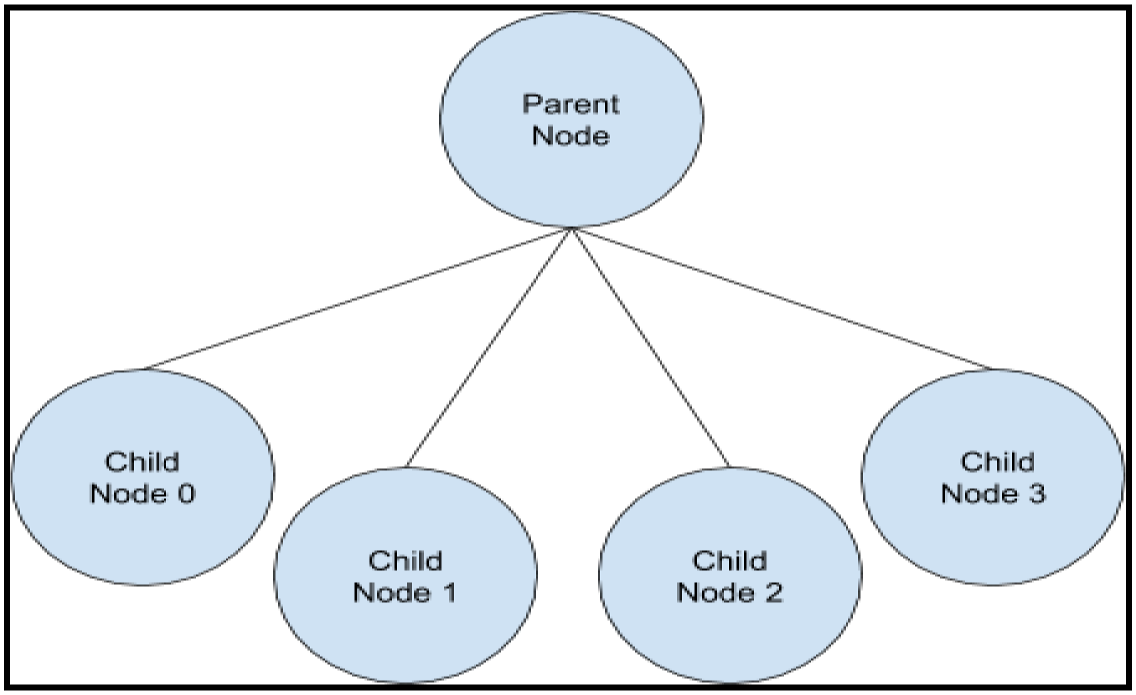
When playing on player vs computer mode, the arduino works in conjunction with the minimax algorithm written in C++ on the laptop. The choice of C++ was in part due to the previously existing C++ rule and movement code and its object oriented nature over C. All the player interactions are still processed in the same way on the mega, after which the computer is given a signal to compute its optimal move using a Minimax algorithm with alpha beta pruning. This process is repeated every turn, appending the current gamestate as required.

Again the software on the laptop can be divided into three key components:

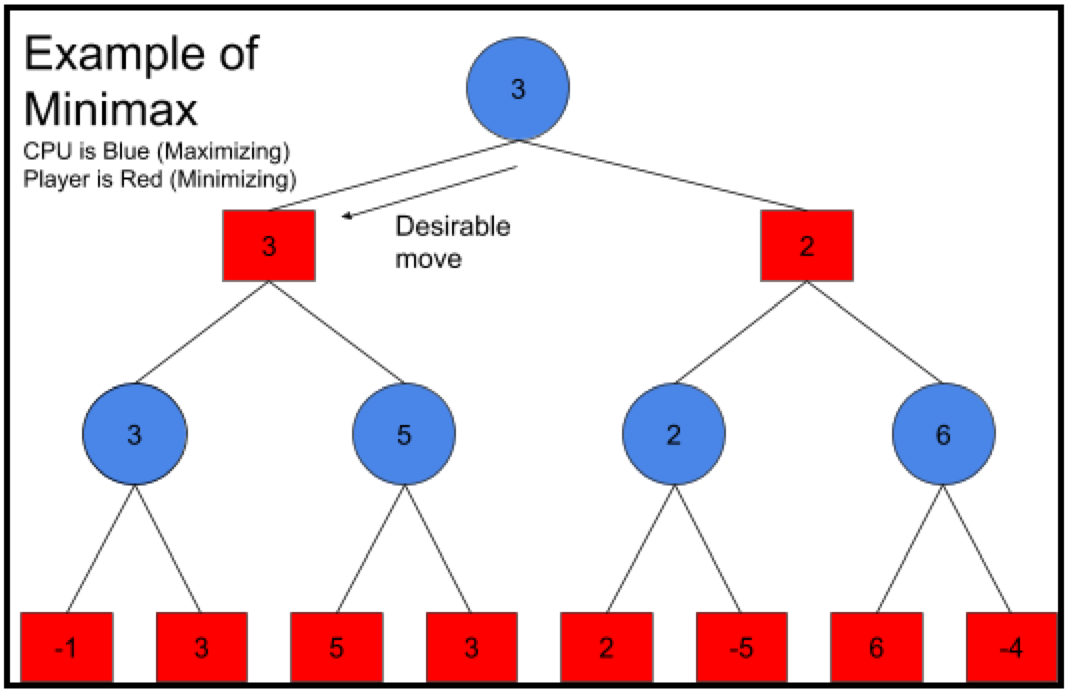
- **Minimax Algorithm**, which recursively chooses the best possible move
- **Move projection**, which projects all possible board states to be analysed
- **Evaluation**, computes the desirability of a given board relative to the player

The move projection portion works directly with the minimax algorithm, by generating child boards from any given parent board without disrupting the original board state. As a backbone for this functionality, the previously coded rule governing functions on the arduino side provide validation for the creation of a child board given a possible move for a selected piece. These functions are modified to simply return whether a move is valid or not, without moving any pieces during their execution. With all of this in mind, the process of projecting valid moves is done piece by piece. When the minimax algorithm calls the projection function it is returned all the valid board states generated from moves of the next movable piece on the board, this piece is located with the findNext function. For example, if called on the starting board state, the projection function will find the first movable piece belonging to the indicated player and store the possible child board states on the parent node. This piece by piece projection allows for a relatively small amount of child node production per call while simultaneously allowing for rapid generation of the entire selection of possible board states when required at maximum depth. To ensure that the projection function doesn't project the same pieces moves multiple times per node, each parent node contains a "place" integer pointer attribute. This is used to indicate how far along in the current board state the projection function has searched, preventing the projection of repeated board states. This also acts as a base case for the projection function, when the place pointer has reached the end of the players pieces, the function does nothing, allowing eventual termination of minimaxing. Once the findNext function has found the next piece on the board belonging to the player, it returns saves its position in the “place” pointer, removing the need for heap allocated pointer while also storing current search location for future use. Next, projection function group runs through every possible move from that pieces position, checking it against the modified version of checkMove, called validMove. If the move is valid, a copy of the parent board state is created, on which the desired move is executed using the newBoard function. Since the moves validity was previously checked using validMove, newBoard simply appends the duplicate board with the desired move. After this, the new board is pushed into the parent nodes child vector, successfully adding that move to the minimax tree. This process is repeated for each valid move given a piece for every call of project.

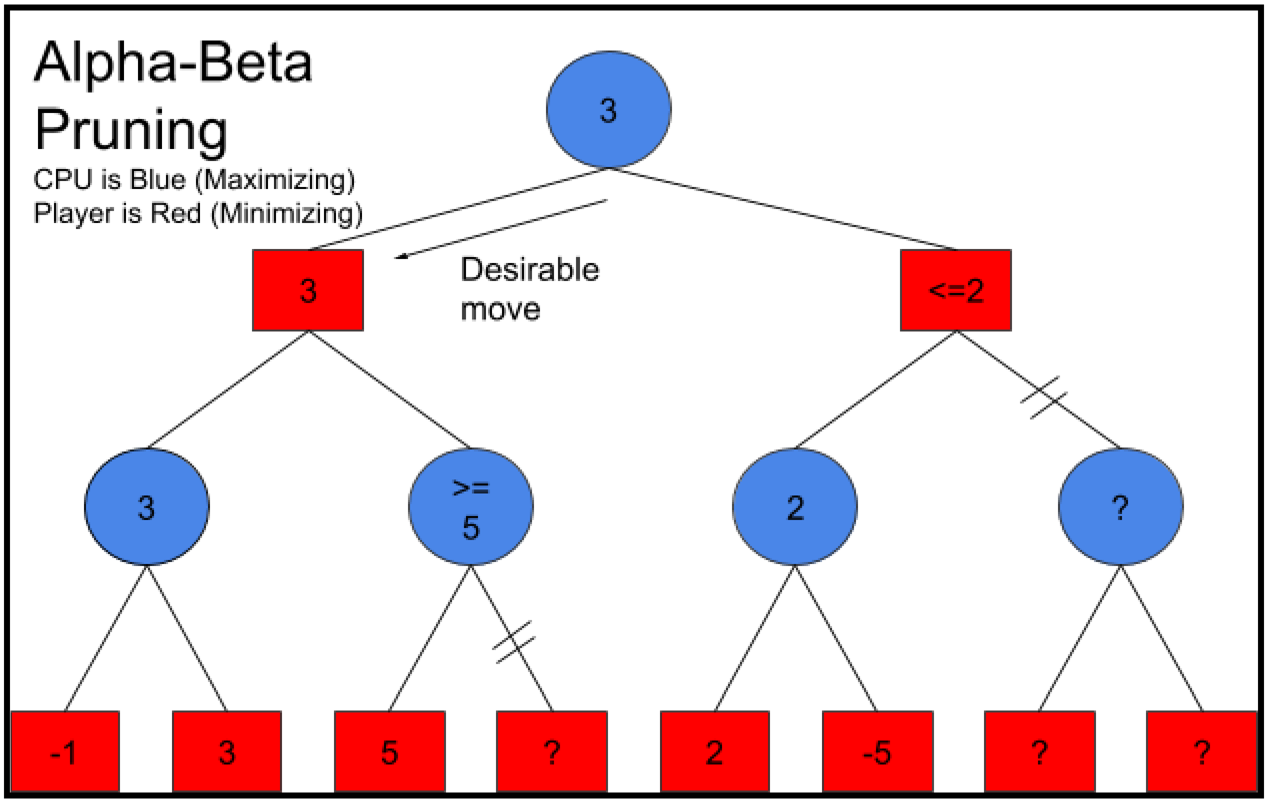
To implement our minimax algorithm, we created a class called Node whose instances would represent every “node” of the decision tree. The most important attribute in the class was the vector of type node called “children,” which stores the child nodes.



The minimax algorithm is simple to understand. First, we designated one player, the computer, as the “maximizing player” and the other, the user, as the “minimizing player.” This means that after the evaluation function is called on a board state, the maximizing player will favour boards with a higher (positive) value whereas the minimizing player will favour those with a lower (negative) value. To project the next move for the computer, we took the current board state as a parameter. From this board state, we alternated between projecting all the possible board states after the computer or the player makes a move. This was accomplished by scanning through the board for the player or the computer’s pieces. If the pieces could move, we created a new board state and stored it in a vector. Once we reached a certain predetermined “depth” (how many moves we chose to look ahead) we called the evaluate function on the final board states. From here, we performed a series of maximizing and minimizing on the evaluations, alternating as we go up the decision tree, until we received a final value.



The minimax algorithm, by nature, is computationally expensive as it runs in O(b^d) where b is the number of legal moves at each point of the tree and d is the maximum depth of the tree. Minimax, by itself, could sometimes take several seconds, or minutes, to generate a move for the computer. To help alleviate this, we implemented alpha-beta pruning, which is an improved version of the minimax algorithm that does not search through redundant branches of the decision tree. Essentially, whenever the maximum score the minimizing player is guaranteed of becomes less than the minimum score of the maximizing player, the maximizing player no longer needs to search through the node's children (and vice-versa.) In the best-case scenario, alpha-beta pruning runs at O($b^{d/2}$) if the moves are ordered in a way that makes pruning, the removal of redundant branches, more likely.



The evaluation function takes various aspects of the board in order to compute a number that represents how much of an advantage the player or the computer has. It takes into account different factors:

- Number of pieces with kings being worth more than normal pieces
- The position of the pieces on the board. A normal piece will be worth more if it's in the center or close to the other side as it has a greater probability of becoming a king
- The number of possible moves
- If the back row is full as it leaves no possibility for the other side to get a king
- If a piece can take the other side’s piece

Each of these factors have different weights that affect the evaluation differently. For example, it will value a piece not being taken more than its position on the board.

As there are different strategies for different stages in the game, there is a check if the game is in endgame. If it is, then it calculates the average distance in between each sides king pieces with all of the other sides pieces. If it has more kings and pieces than the other in the endgame, there is an incentive to decrease the average distance between pieces so that they will get closer to taking the other's pieces and therefore, closer to winning the game.

Group Members' Contributions

Andrew Wang

- Assembled the Arduino Mega, LED Matrix and buttons with Nick F.
- Created the alpha-beta pruning algorithm in C++ to generate the moves for the computer.
- Integrated Nick M's project() function and Connor's evaluate() function into the alpha-beta pruning algorithm to generate child board states and values.

[illegible]

Connor Byers

- Created the canAttack() function, working hand in hand with checkMove to validate moves and provide a smooth game play experience
- Created the entire board Evaluate() function, which computes a rating for any given board state, working in conjunction with the minimax
- Played an instrumental role along with Nick F. in connecting the laptop software with the arduino for use in the player vs ai game mode

[illegible]

Nick Makharinets

- Created the checkMove() function, a software representation of the movement rules in checkers
- Created the entire move projection architecture and implementation for use with the Minimax algorithm
- Integrated the move projection functionality with the Minimax algorithm

[illegible]

Nick Faria

- Assembled the Arduino Mega, LED Matrix and buttons with Andrew W.
- Coded visual representations for the board and prompts for use with the LED Matrix
- Worked hand and hand with Connor B. in the connection of the laptop software with that on the arduino, researching and devising implementations

[illegible]

Final Product Evaluation

We believe our final product to have completed the vast majority of its original intended functions. First, the player vs player gamemode is fully functional and also includes additional optimizations such as displaying when each player’s turn occurs, and a visually appealing menu screen to select the game mode from. Unfortunately, we were not able to complete our goal of allowing ai assisted move hints during player vs player. Despite missing this small goal, our product exceeds initial expectations and fulfils its core functionality well.

One of the greater disappointments was our inability to manage communication between the Arduino and our C++ program in both directions. This meant that, during the player vs computer game loop, instead of collecting input from the controller we made, we instead had to resort to manually inputting the board state on the C++ program and sending it to the Arduino (since communication only works from the computer to the Arduino and not the other way around.) Despite this setback, we believe our project to have accomplished most of its goal since the AI is functional and has held its ground against other online checkers-playing software in testing.

Design Trade-offs

Processing minimax() on the computer:

Unfortunately, there is only about 256 kB of memory available from the Arduino Mega. This is not enough memory to run our minimax function as the amount of checkers boards, and consequently the demand for memory, exponentially increases as it searches down the decision tree. This meant we had to write our minimax function on a separate C++ file and compile it ourselves, which adds the extra need to manage communication between the Arduino and our computers. This, however, allowed us to generate all the possible boards, and therefore allowed the computer to make the most optimal moves. If we coded everything on the Arduino IDE, then we would have sacrificed a great deal of runtime for the sake of convenience.

Removing force-takes:

In some versions of checkers, if a piece can afford to attack then it will be forced to perform the attack. Due to the nature of our minimax function, we decided to omit the force-take rule from our game. If we included the rule, then we would need to run a separate minimax function to account for the force-takes which would drastically increase the computation time.

Inconvenient button layout:

Currently our button layout on the breadboard is embedded in a series of wires, making it slightly inconvenient to play with them. As previously discussed, this was due to the fact that we had to wire our buttons as pullup input to ensure they would not draw power away from the 5 volt pin on the arduino and prevent the matrix from displaying properly. This change in wiring means there needs to be wires on each side of a button, making conveniently pressing the buttons slightly awkward.

Code cleanliness:

In order to make traceable code for easy bug fixing, some portions are not as optimized and efficient as possible. This tradeoff was mandatory due to the sheer amount of code, numerous integration steps, and the necessity to meet deadlines after unforeseen delays such as the extreme difficulty we had with minimax arduino integration.

Inability to communicate with Serial

Unfortunately, we were unable to fully manage the communication between the Arduino and our computers. This meant that we were unable to implement a player vs computer game loop in the code that runs on the arduino. We tried to alleviate this issue by making use of the one-way communication between the computer and the arduino that we were able to get working, allowing the player to manually enter their board state on the C++ program and having the LED Matrix display the moves without sending any user input back. Due to the lack of documentation found online on how to communicate with the Serial monitor using C++, we were forced to resort to this solution.

Future Work

In the future we would also like to clean the move projection and rules software, as certain trade offs were made to optimize modularity and readability in order to facilitate bug fixing. Unfortunately, this is not the most optimal process for the required operations, and thus cleaning the code up will improve efficiency while maintaining functionality.

Despite using alpha-beta pruning to speed up the minimax algorithm, the function can still take a significant amount of time to generate a move depending on the boardstate. This is most likely because the probability that a branch will be pruned is maximized when the possible moves are ordered from most to least likely. In our project, we have the option to create some form of move-ordering function to aid the pruning of the decision tree if we wish to do so in the future.

We would also like to find a solution to communicating between the Arduino and our C++ program. Perhaps we would have to convert our C++ code to a language that is more export-friendly such as Java or Python. Otherwise we would have to integrate the minimax function into our Arduino code and limit the number of children a certain node can spawn to a small constant number to prevent the program from running out of memory. Regardless, we do not see a solution to this problem that involves keeping our code in C++, unless a working serial communication library for C++ is found.

References:

Alpha–beta pruning. (2019, October 19). Retrieved from https://en.wikipedia.org/wiki/Alpha–beta_pruning.

[1]

Noamyoungerm. (1993, June 1). Finding moves with multiple jumps in checkers. Retrieved from https://gamedev.stackexchange.com/questions/53963/finding-moves-with-multiple-jumps-in-checkers.

[2]

checkers.c. (n.d.). Retrieved from http://gtkboard.sourceforge.net/doc/doxygen/checkers_c-source.html.

[3]

Arneson, E. (2019, August 10). The Game and History of Checkers. Retrieved from https://www.thesprucecrafts.com/play-checkers-using-standard-rules-409287.

[4]

https://cs.huji.ac.il/~ai/projects/old/English-Draughts.pdf

[5]

Filipek, B. (n.d.). Vector of Objects vs Vector of Pointers Updated. Retrieved from https://www.bfilipek.com/2014/05/vector-of-objects-vs-vector-of-pointers.html.

[6]

Arduino . (n.d.). Arduino Mega. Retrieved from https://store.arduino.cc/usa/mega-2560-r3.

[7]

Burgess, P. (n.d.). 32x16 and 32x32 RGB LED Matrix. Retrieved from https://learn.adafruit.com/32x16-32x32-rgb-led-matrix/.

[8]

Adafruit. (2019, September 25). adafruit/RGB-matrix-Panel. Retrieved from https://github.com/adafruit/RGB-matrix-Panel.

[9]

Adafruit. (2019, October 18). adafruit/Adafruit-GFX-Library. Retrieved from https://github.com/adafruit/Adafruit-GFX-Library.

[10]

HELLOSIGN

Audit Trail

TITLE	SE 101 Report
FILE NAME	SE 101 Report
DOCUMENT ID	308d71a710089c8a9d3fc5a5c38e2797b2a031f1
AUDIT TRAIL DATE FORMAT	MM / DD / YYYY
STATUS	● Completed

This document was requested from script.google.com

Document History

SENT	**12** / **01** / **2019** 04:43:35 UTC	Sent for signature to Nicholas Makharinets (nmakharinets@gmail.com), Connor Byers (connorbyers27@gmail.com) and Nick Faria (n.faria373@gmail.com) from 324.andrew.wang@gmail.com IP: 129.97.125.5
VIEWED	**12** / **01** / **2019** 04:46:06 UTC	Viewed by Connor Byers (connorbyers27@gmail.com) IP: 129.97.131.0
SIGNED	**12** / **01** / **2019** 04:46:21 UTC	Signed by Connor Byers (connorbyers27@gmail.com) IP: 129.97.131.0
VIEWED	**12** / **01** / **2019** 04:46:42 UTC	Viewed by Nicholas Makharinets (nmakharinets@gmail.com) IP: 129.97.125.5
SIGNED	**12** / **01** / **2019** 04:46:51 UTC	Signed by Nicholas Makharinets (nmakharinets@gmail.com) IP: 129.97.125.5

HELLOSIGN

Audit Trail

TITLE	SE 101 Report
FILE NAME	SE 101 Report
DOCUMENT ID	308d71a710089c8a9d3fc5a5c38e2797b2a031f1
AUDIT TRAIL DATE FORMAT	MM / DD / YYYY
STATUS	● Completed

This document was requested from script.google.com

Document History

VIEWED	**12** / **01** / **2019** 06:57:30 UTC	Viewed by Nick Faria (n.faria373@gmail.com) IP: 129.97.125.4
SIGNED	**12** / **01** / **2019** 06:58:13 UTC	Signed by Nick Faria (n.faria373@gmail.com) IP: 129.97.125.4
COMPLETED	**12** / **01** / **2019** 06:58:13 UTC	The document has been completed.