Evaluating the Capability and Limitations of a Minimax Checkers AI in MATLAB

Modeling and Simulation

Jaden Keener

April 19th, 2022

Contents

1 Abstract

The game of checkers is defined by a relatively simple set of rules, but allows for a very high degree of player strategy. While computers struggle to understand abstract concepts like strategy, they are exceedingly powerful at following sets of clearly defined rules. This project explores the possibility of creating the game of checkers and an AI capable of replicating these advanced human tactics in game, and seeks to understand some of the potential limitations of such an AI. We were successful in modeling and implementing the game of checkers and an AI capable of playing it, finding that the AI opponent could represent an extremely capable opponent at the cost of long program runtimes.

2 Introduction

Checkers is a game that is mechanically very simple, but has a high degree of depth in strategy [3]. Although the game is structured around a simple turn order and every piece moves and captures in the same manner, an experienced player can routinely best a new player through their knowledge of strategy. This creates a low ”skill-floor” where it is very easy for someone to sit at a checkers board and play the game, but a very high ”skill-ceiling” where you can expend a large amount of time practicing your strategy.

This combination of a low skill-floor and high skill-ceiling makes checkers a very interesting problem to tackle with computer AI. Computers excel at following a set of simple rules, such as those present in checkers, but infamously struggle with more abstract concepts like the strategies required to play checkers at a high level. Despite this shortcoming, we may be able to create a computer AI capable of besting the average checkers player simply by 'brute-forcing' a strategy from the simple rules of checkers.

This problem has been considered before, and an algorithm known as ”Minimax” [4] was developed. The minimax algorithm will be explored more thoroughly in the background section of this report, but combining the minimax algorithm with the ability of the computer to play many simple moves allows us to create a computer checkers opponent capable of making good moves, and possibly even beating many human players.

This approach to creating a computer opponent is not without its limitations. Even though the computer can play and evaluate moves very quickly, the number of moves possible in a game of checkers is massive. In order for our computer to choose worthwhile moves we have to evaluate not only the moves currently available, but also the moves that will be available to our opponent after each one of our possible moves, and then again all of the moves available to us after each of those possible moves available to our opponents, and so on. This report will investigate the feasibility of creating the game of Checkers and an associated AI opponent in the MATLAB programming language, and will seek to evaluate the capability, completeness, and run-time limitations of our game and AI opponent at different algorithm depths.

3 Background

We will now cover topics that are important to understanding the project. The two systems that we need to model are the game of checkers itself and the minimax algorithm which we will use to create an AI opponent. The following subsections give some high level knowledge on the topics.

3.1 Checkers

In order to create a computer opponent in checkers we first need to understand the game itself. There are many variations of checkers, so we will briefly define the rules we will use in our simulation. Checkers is a two player, turn based game played on an 8x8 checkerboard[1]. Each player is given 12 pawns at the start of the game, arranged in the pattern shown in figure 1. Our simulation has the black player and red player, with the black player playing first.

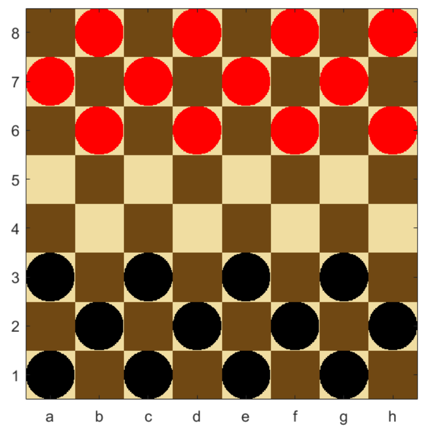


Figure 1: The setup for a game of checkers.

The rules of checkers are relatively simple. Pawns may move diagonally to any unoccupied space directly adjacent to their current position, but may only move towards the enemy side of the board. If an enemy piece is present on a space a friendly piece would normally be able to move to, AND the space immediately after the enemy piece is not occupied, then the friendly piece may jump ’over’ the enemy piece, capturing the enemy piece in the process. After a piece captures, if that same piece can now capture another opponent it may do so. This is known as a chain capture, allowing one piece to capture many opposing pieces at once. One critical rule relevant to this implementation of checkers is that when a capture move is available, it MUST be taken. If multiple captures moves are available then the player may choose between them, but they may not choose a move that does not result in a capture when a capturing move is available.

There is one more piece type in addition to pawns. When a pawn reaches the opposing edge of the board, it is upgraded to a king. King pieces follow all of the same rules as pawns except that they may move both forward and backward. There is no limit on the number of pawns that may be upgraded to kings. Our visualization will represent king pieces with a golden center.

The objective of the game is to remove all of the opposing pieces from the board. The game ends either when all opposing pieces are eliminated or when the opposing player has no legal moves available to

[1]Traditionally, the checkerboard is built such that the spaces under the pawns are dark. Our visualization will flip the checkerboard colors for improved contrast and readability.

them.

3.2 The Minimax Algorithm

As the rules of checkers are a very simple for a computer to understand, it is very easy for the computer to play a move. Making that move a good move is much more difficult. The easiest way to get the computer to play a good move is to simply have it try to play many, many possible moves and then decide which one is best. Deciding which move is best is the job of the minimax algorithm. The minimax algorithm evaluates the current state of the board by creating a sum based on the pieces present on the board. For example, black pawns may be worth +3 points with red pawns worth -3 points. Kings are similar, worth ±5 points for black and red respectively. This condenses the state of our board to a single integer.

Since the black players pieces are worth positive points, and more pieces on the board is usually a good thing, we can say that the black player is in good form when the minimax algorithm returns a positive number for the position. Likewise, the red player is at an advantage when the minimax algorithm returns a negative number. The magnitude of this number correlates to how much of an advantage the player has. The player who is trying to achieve a positive score from the minimax algorithm is known as the "maximizing player", and the player trying to achieve a negative score is the "minimizing player".

Figure 2 shows an example game state. In this state, minimax would evaluate the board as +4 (+9 from black pawns, +5 from black king, -10 from red kings). This means that the current game state is in advantage to the maximizing player (black).

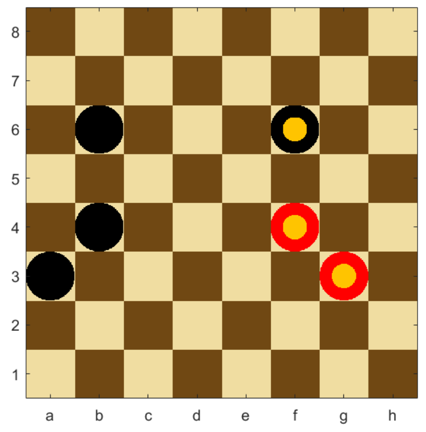


Figure 2: Example game state, minimax evaluates as +4.

By combining this minimax algorithm with the computers ability to quickly evaluate many possible moves, we can create a basic strategy for the computer to follow. In its most basic form the computer simply attempts to play every move it can, and then sees which move returns the most desirable value from the minimax algorithm. This would be an example of running the minimax algorithm with a depth of 1. Looking only a single move into the future does not provide for very good strategy however, so to see any real capability in our AI we must evaluate more than one turn. If we were to look at all of the turns immediately available to us, then all of the responses of our opponent, this would be evaluating with a depth of 2. This means that in addition to evaluating all moves immediately available to us, we must also evaluate all moves available to our opponent after every one of our possible moves.

The process for choosing a best move at algorithm depths greater than one will be discussing in the following section on our models and implementation.

4 Models and Implementation

We will now detail our relevant models and their implementations. This project requires us to model both the game of checkers itself and the minimax algorithm. The following subsections detail the models and a high level overview of the implementation for many individual functions that work together to create a functioning system. The full code for each function can be found in the appendix.

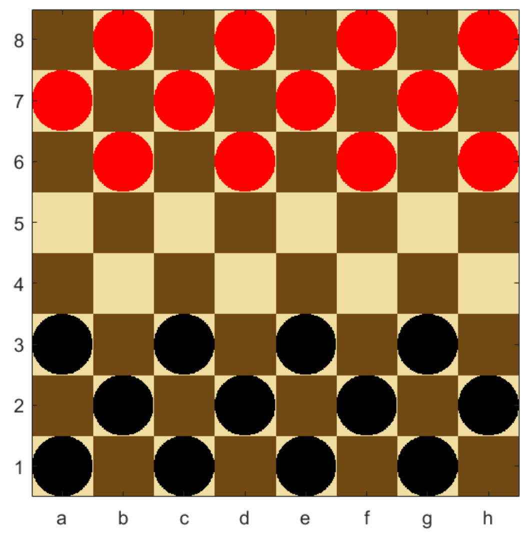
4.1 Modeling Checkers

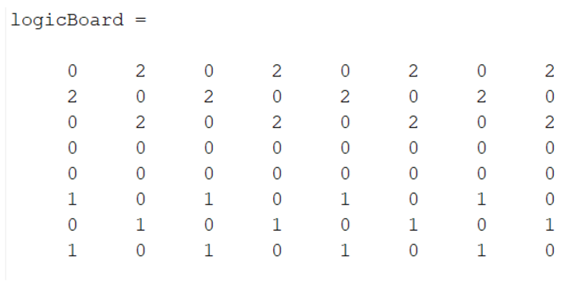
Before we may begin modeling our AI with the minimax algorithm, we must first model the game of checkers in MATLAB. This task is more difficult than it may seem. Though the rules of checkers are relatively simple, it is still quite challenging to implement in a programming language.

4.1.1 Capturing the Game State

Perhaps the most important aspect of our checkers model is modeling the current state of the game. In order to use the minimax algorithm later on, we need to capture the game state in a form that is simple to read and manipulate. We chose to model the game state as an 8x8 array, *logicBoard*, named such as it is the board that all of our game logic will be based on. As an 8x8 array *logicBoard* has the same dimensions as our checkerboard. To capture the current game state, we can simply assign each piece type a unique value and then store it in the index of *logicBoard* that corresponds to its place on the board. For example, we may define a black pawn as having a value of '1' in *logicBoard*. If there was a black pawn in the eight row first column of the board (position a1), then *logicBoard(8,1) = 1*. We can then assign unique values to our other piece types (black king, red pawn, red king) and store them in a similar manner.

We chose to define our model such that the black pawn, red pawn, black king, and red king are identified by '1', '2', '11', and '22' respectively. Figure 3 shows how *logicBoard* looks in a new-game state.





```
logicBoard =

     0     2     0     2     0     2     0     2
     2     0     2     0     2     0     2     0
     0     2     0     2     0     2     0     2
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
```

Figure 3: New-game state and the corresponding value of *logicBoard*

Now that we have defined a model for how we will store our current game state, we can create a simple function to initialize *logicBoard* for a new game. This function will therefore be named *newGame()*. This function initializes *logicBoard* as a global[2] variable and utilizes a simple for loop to place the pawns in the correct places for a new game.

4.1.2 Generating Legal Moves

Now that we have a way to store and manipulate our current game state, we can write a function to determine the legal moves available to the player. We will define a function *generateMovesPlayer()* to generate a list of all legal moves that a player may make in the current game state. This function will take only one input argument, *player*, getting all other relevant information from global variables (such as the global *logicBoard*). The *player* input argument is a boolean that tells the function which player it should generate moves for (true is player one, false is player two).

The function will store all moves into a global variable *moveList*. The *moveList* variable is a cell array with a very specific structure. Each move is described by a double array where the first index is the index of *logicBoard* where the piece we want to move resides. All following values in the move array are the spaces the piece in question has to move over or to in the move. The first index of *moveList* is unique, as it holds an array with the indices of *moveList* where capture moves are present. Such that if the third and fifth index of *moveList* are captures moves, then *moveList*{*1*} = [3, 5].

For example, take the game state shown in figure 4. Lets evaluate the moves available to player 1 in this position. First, there is an empty square to the top right of our piece. This move would be generated as [d4, e5][3], as the piece we want to move is on d4 and we want to move only one space

[2]This project makes significant use of global variables. The author came to regret this choice, and would instead recommend using MATLAB's object oriented classes and their corresponding properties and methods.

[3]In the actual program, these would be integer index values. d4/e5 are used here for example purposes.

to e5. This move would then be appended to the end of *moveList* We also need to evaluate if we can move to the top left. There is an enemy piece in this position that we can capture[4]. The move array generated for this capture would be written as [d4, c5, b6]. It first specifies the piece we want to move (d4), then specifies every step along the way, INCLUDING the index of the piece we are capturing (c5) before finally finishing on b6. This move would once again be appended to the end of *moveList*, and the index of this capture move would also need to be included in *moveList*{*1*} as previously described. This generates the final output *moveList* = {[3], [d4, e5], [d4, c5, b6]}.

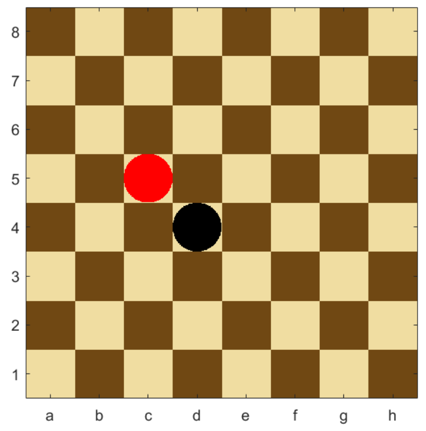


Figure 4: Example game state to illustrate the functionality of *generateMovesPlayer()*. Evaluating this position for black generates *moveList* = {[3], [d4, e5], [d4, c5, b6]}

The function works by looping through every index of *logicBoard*. If the index contains a piece that belongs to the player specified by *player*, then it evaluates what moves that piece can make. It performs this evaluation by checking the indices of *logicBoard* that are diagonally adjacent to the piece in the direction it can move. If it is an empty space, then it can move. If there is a friendly piece on the space, it can not move. If there is an enemy piece, then we perform more advanced logic to evaluate a potential capture, which will be described in the next section. It then stores these possible moves in *moveList* as previously described. At the end of *generateMovesPlayer()*, we check to see if *moveList*{*1*} is empty. If it is, that means there were no capture moves available so all moves are legal. If it is not empty, that means that the only legal moves available are those at the indices of *moveList* specified in *moveList*{*1*}, so we should ONLY return those moves. Therefore, the function will prune all moves from *moveList* except for the capture moves. The function also prunes *moveList*{*1*} before returning, such that the final value of *moveList* contains ONLY legal moves and nothing else. This pruning happens regardless of that status of *moveList*{*1*}.

4.1.3 Detecting Captures

In the previous section on generating legal moves, we discussed how capture moves are written to the *moveList* cell array. Detecting these capture moves is not trivial however, and requires extensive logic of its own. This section will discuss how capture moves are detected and evaluated.

[4]The logic for detecting captures is somewhat complex, and will be described in the next section

It is the responsibility of *generateMovesPlayer()* to first detect that a capture move is possible. If while evaluating the moves for a given piece, the function notices that there is an enemy piece adjacent to the piece in question AND the space after that enemy piece is open to jump to, then it sees this as a viable capture. This logic is relatively simple, however, we must remember to evaluate the possibility of a chain-captures. As a reminder, a friendly piece can capture multiple opponents in one move if a chain of captures presents itself. The logic for detecting these chain captures is significantly more complicated and requires recursion, so we will write a new function *evalHop()* to evaluate them.

The *evalHop()* function is designed to be called recursively to evaluate chain-captures. The function takes three input arguments. The first argument *i* holds the index of where the capturing piece WOULD be on its current chain-capture path. This allows us to evaluate potential chain-captures from any place on *logicBoard*. The second argument, *move*, holds the move as it currently exists. As this function calls itself recursively, we need to keep track of where we have already been. The *move* argument is formatted the same way as the moves in *moveList*, and is continuously built over each call of *evalHop()*. When *evalHop()* is first called *move* is the capture first detected in *generateMovesPlayer*. The third and final argument is *player* and is the same boolean used in *generateMovesPlayer()*.

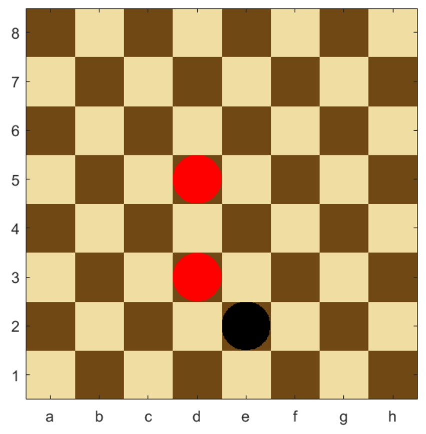


Figure 5: Example position to explain *evalHop()*.

Let's give an example of how this function works using figure 5, evaluating for player one (black). In this position *generateMovesPlayer()* will recognize an initial capture move [e2, d3, c4]. It will then detect any possible chain-captures by calling *evalHop(c4, [e2, d3, c4], true)*. As previously discussed, these arguments tell the function evaluate any potential captures from a hypothetical black piece at c4, and if a capture exists, append its moves to the current *move* [e2, d3, c4]. The function will see that a chain capture is available from c4 (capturing the red pawn on d5). After detecting this capture, the function will recursively call itself as *evalHop(e5, [e2, d3, c4, d5, e6], true)*. The function will then evaluate any potential captures from position e5. Since there are no more possible captures here, the function returns and appends the final value of *move* to *moveList*.

The recursive nature of *evalHop()* means that it is capable of evaluating multiple possible 'branches' of chain-captures. For example, if after one capture there are two possible chain-capture paths, it will evaluate and return both possible paths and append them to *moveList.*

Special care must be taken to ensure that *evalHop()* works in all scenarios. The function must take care not to jump over the bounds of the board, so exceptions are written to ensure that no captures can be made if our piece is in the first or last two rows. Additionally, extra care must be taken in order to evaluate king hops. As a king can move in any direction, it is easy for the function to get stuck in an infinite loop of attempting to capture the same piece repeatedly (going back and forth). To avoid this, we create a local variable *hopBoard* which is equivalent to *logicBoard*, except that all locations specified in *move* are set to zero. All logic decisions for chain-captures are then made using this *hopBoard* variable. This makes it so that a piece does not appear to be captured twice, without actually changing the current game state by overwriting *logicBoard.*

4.1.4 Playing Moves

Now, we must create a simple function that allows our program to play moves, *playMove().* This function takes only one argument, *move.* The *move* argument is simply one of the moves contained in *moveList.* Lets use the capture move seen in figure 4 as an example move to play. We will call *playMove([d4, c5, b6]).* The function plays this move by overwriting the value of *logicBoard(b6)* with the value of *logicBoard(d4)*, then setting*logicBoard([d4, c5])* to zero. This works for a move with any number of indices. This seamlessly integrates capturing, as the captured piece will reside on one of the indices in *move*, and will be overwritten with a zero.

The *playMove()* function is also where we evaluate for kings. After every move is played, we check to see if a pawn is on the opposing edge of the board (a black piece in row 8 or a red piece in row 1). If there is an enemy pawn on these rows, overwrite it with a king.

4.1.5 Checkers Visualization

The previously described models and functions provide us with all of the logic needed for our game of checkers to be playable. Now we may create a function to visualize the board. This function, *drawBoard()* will take no input arguments (though it does rely on the global *logicBoard*). As we stored all of the information about the current position in the *logicBoard* variable, we should be able to make a complete visualization based on it alone.

Our *drawBoard()* function will use the *imagesc()* command to create our visualization. The *imagesc()* command creates a figure where each pixels color is defined by its value in an array. The size of the array is therefore the resolution of the image. Therefore, our first task is to create the array which holds our image information. We will name this array *imageBoard.* We will use a simple series of for loops to build a checker pattern onto *imageBoard* by writing square chunks with a specific value which corresponds to the checkerboard colors. We will assign our dark squares a value of 3, and our light squares a value of 4.

Placing the pieces on the board requires us to write circular areas of *imageBoard* with specific values for red or black. We do this by creating a circular logic array, *mask*, which is the size of our checkers. We then use a for loop to step through every index of *logicBoard.* If *logicBoard* contains a piece at index *i*, then we use the circular logic array *mask* to 'paint' the piece on to the corresponding position in *imageBoard.* If the piece happens to be a king, then we use a smaller version of *mask* to paint a smaller golden circle on top of the piece.

As this process of drawing checkers is destructive to *imageBoard*, we must redraw the checkerboard pattern on every call of *drawBoard()*. The destructive nature also makes draw order important: we must draw the checkerboard pattern first, then the pieces, then the king circles. After we build *imageBoard* we can define a custom colorspace using the *colormap()* command. This allows us to choose specific RGB color values to correspond to the five[5] different integer values present in *imageBoard*. Figure 6 shows an example of all possible pieces on the board at one time. Also note that every figure in this report that contains a screenshot from the game was created via this function.

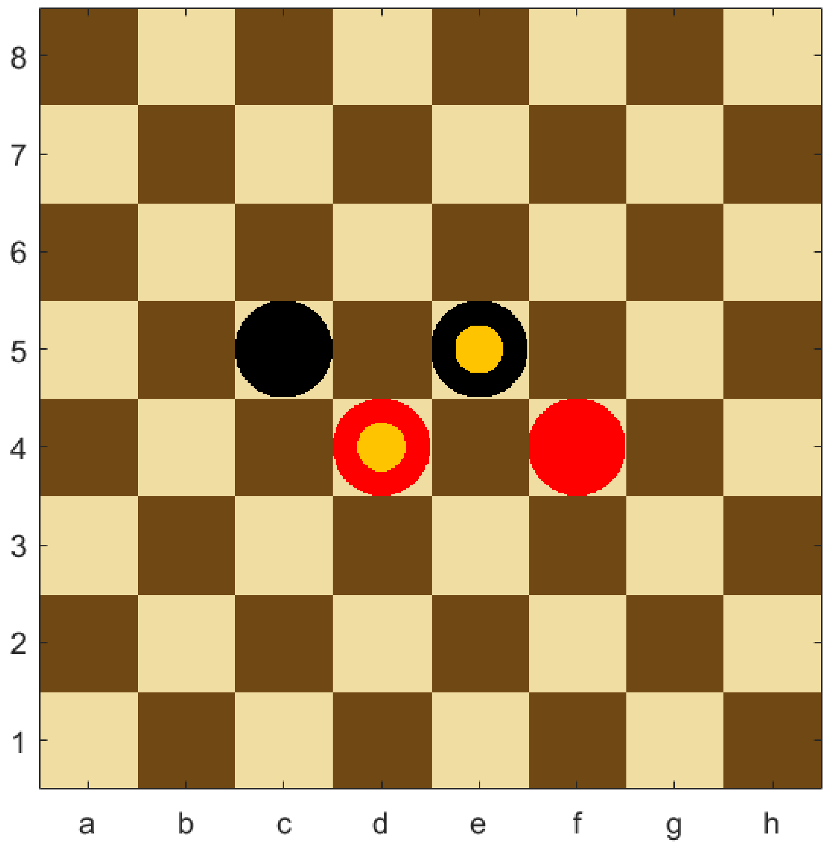


Figure 6: Every piece possible, drawn by *drawBoard()*.

One interesting note on the implementation of *drawBoard*. The *imagesc()* command uses a relative scale for the colorspace. This means that in order for our colorspace to be assigned correctly, ALL values 1 through 5 must be present in *imageBoard* at all times. For this reason, the corners of *imageBoard* will always contain one pixel of the red, black, and king gold colors. At high drawing resolutions these pixels are unnoticable, but still necessary.

[5]The values are: 1-Black, 2-Red, 3-BoardDark, 4-BoardLight, 5-KingGold

4.2 Modeling the Minimax Algorithm

The minimax algorithm was briefly discussed in the background section of this report, but we will now more thoroughly model the minimax algorithm as it applies to our checkers game. As previously described, the minimax algorithm works by evaluating the current game state (position) by assigning a point value to each piece type. However, for the minimax algorithm to be of any use to us, we must evaluate at a depth far greater than 1. Evaluating at a depth of greater than 1 requires us to define more comprehensive criteria for choosing our best move, as we now must consider how our opponent can respond.

4.2.1 Minimax at Depths Greater Than One

Choosing the best move at a algorithm depth of 1 is simple: we just choose the move that results in the highest or lowest minimax evaluation (depending on whether we are the maximizing or minimizing player). When evaluating at depths greater than 1, we must decide how the opponent would react. It would make sense for our opponent to also choose the move that results in the best minimax evaluation for them at the current position. Therefore, when evaluating at minimax depths of greater than 1 we will choose the turn that will give us the best minimax evaluation n turns in the future (where n is evaluation depth), assuming that the opposing player will choose their best possible move at every step of the process.

In order to accomplish this forward-looking evaluation, we have to evaluate from the "bottom up". Consider a minimax evaluation with depth 3. Our minimax algorithm first generates every possible position after $n = 3$ moves, but it does not evaluate any. Once it reaches its maximum depth (in this case 3), it evaluates those positions. Then, on the previous layer (depth 2) the algorithm chooses to play the move generated at depth 3 which has the best evaluation. Keep in mind that "best" evaluation depends on whether the player in the previous layer is maximizing or minimizing. Layer 2 is then assigned that value that it chose from layer 3, as that is the current evaluation. This process repeats until we reach depth 0.

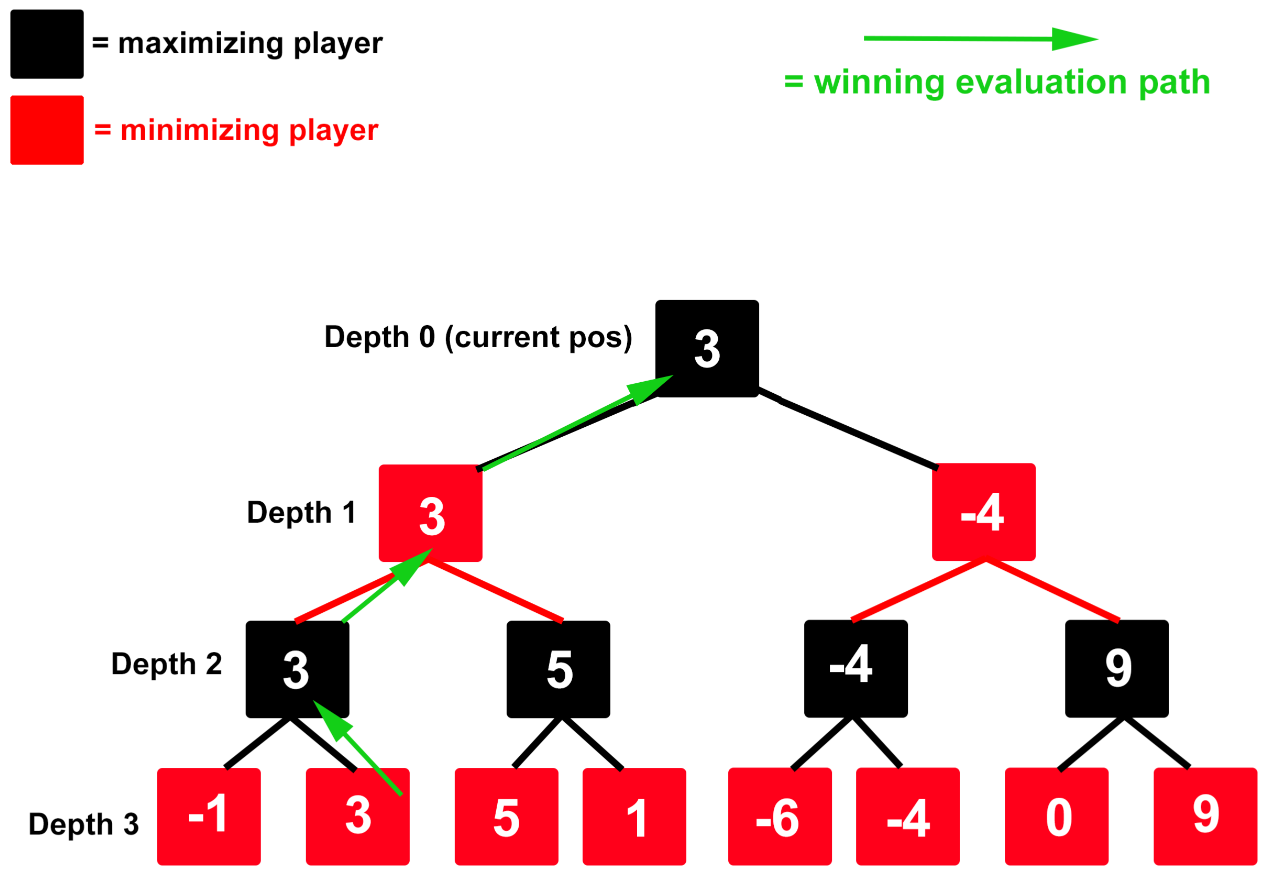


Figure 7: Minimax Algorithm Flowchart

This process can be explained more clearly through the use of a graphic. Figure 7 shows an algorithm flowchart for minimax at depth 3. Each square represents a position, and each line represents the move which results in that position. In our example, we FIRST generate every possible position after 3 moves. THEN we evaluate all of the positions at our maximum algorithm depth. NEXT, the player at depth 2 (black) chooses to play the move hat results in the position from depth 3 with the best evaluation (in this case, the move with the highest evaluation since black is maximizing). The value of the position at depth 2 is then assigned as the value it chose from depth 3. This process then repeats at depth 1, though this time the opposing player is choosing the move. This means that it will choose the move that generates the position with the LOWEST evaluation possible. Finally, depth 0 chooses the best move available to it, and plays it.

4.2.2 Minimax in MATLAB

Now that we have a model for our minimax evaluation technique, we have to implement it in MATLAB. The best solution to this problem is a recursive function. We will define a new function *minimax()*. This function accepts two inputs: the first *depth* is an integer value greater than or equal to 0, which defines the algorithm depth (how many moves the deep the algorithm should look). The second argument *maximizing* is a Boolean that tells the *minimax()* function whether it should evaluate for the maximizing (true) or minimizing (false) player.

The *minimax()* function also has two outputs, *eval* and *bestMove.* The *eval* output is simply the integer evaluation that the minimax function returns, while *bestMove* is the move[6] that *minimax()* thinks should be played.

The logic for the *minimax()* function is relatively simple. We first perform a simple check to see if *depth* is greater than zero. If it is, then we generate all of the moves available at the current position with *generateMovesPlayer()*[7]. check if we are currently maximizing or minimizing based on *maximizing.* Both branches of this check can be described similarly. We then enter a for loop where we step through every move in *moveList.* In this for loop, we play one of the moves then call *minimax(depth-1, maximizing).* This call takes us one iteration further into minimax and changes *maximizing* to the opposite value (evaluating from the opposite point of view). The output of this *minimax()* call is then compared to see if it is bigger (maximizing) or smaller (minimizing) than our current best evaluation[8] If the returned evaluation (*currentEval*) is better[9] than our previous best, then we change our evaluation to the new one (*eval = currentEval*) and assign our *bestMove* as the move that was just evaluated. After evaluating all possible moves and finding a final value for *eval* and *bestMove*, we return to the function that called us.

When *minimax()* is called with a depth of zero, then we know we have reached the end of our position tree (the last row in figure 7). At this point, we need to perform a "static evaluation" of the positions. This is where we sum the value of the pieces to generate an evaluation as described in the background section of this report. We perform a static evaluation in MATLAB by simply stepping through every index of *logicBoard* and modifying *eval* according to the piece discovered at that index (± 3 for pawns and ± 5 for kings). We then return to the function that called with our new value of *eval.*

One important note on implementation. The *minimax()* function makes use of several of our previously written functions, namely *playMove()* and *generateMovesPlayer().* These functions were written in such a way that they will always modify the global variables *logicBoard* and *moveList.* We need these global variables to be intact at various points within *minimax()* and after *minimax()* completes, so we need to store and load its value locally at several points while the function runs. This could have been avoided by better defining our functions to work without the use of global variables.

5 Results and Discussion

We will now discuss the findings of our project. As a reminder, our project is seeking to evaluate the possibility of implementing checkers and an AI opponent in MATLAB, then evaluating the proficiency and runtime of the AI opponent at different minimax algorithm depths. The following subsections break down each question in more detail, each containing our findings and a brief discussion on their meanings.

[6]This move is formatted in the same way moves were defined in *moveList*

[7]The *player* input boolean of *generateMovesPlayer()* is the same as *maximizing*

[8]The evaluation starts at negative infinity for maximizing players and positive infinity for minimizing players.

[9]The function has special handling in the case that *currentEval == bestEval.* In this case, it will randomly decide whether to update our *eval* and *bestMove*, or let them remain the same. This was added to avoid the computer playing the same moves over and over again when playing against itself.

5.1 Completeness (Playability)

First, we must evaluate the completeness of our game model. One particularly relevant metric we can use to determine if our game is complete is to give it some challenging scenarios, and see if it can successfully generate all legal moves in that position. If our game is capable of generating all legal moves in any given position, and ONLY legal moves, then we can say that it is a complete model of checkers. This is a common metric used in the development of Chess AI [1], and should be no less applicable to a checkers AI.

Figure 8 shows an example position that should be relatively challenging to calculate, black to play. This position checks the capability of our program to evaluate kings, evaluate blocked captures, evaluate chain-captures with multiple possible paths, evaluate king chain-capturing, and to evaluate forced captures. The king piece should have four legal moves in this position: d4 to b6, d4 to f2, d4 to f6 to d8, and d4 to f6 to h4. As a capture is available, and capture moves must be taken when available, the pawn piece should not have any moves (even though it has an open space adjacent at a3). Therefore, we should have a total of 4 legal moves.

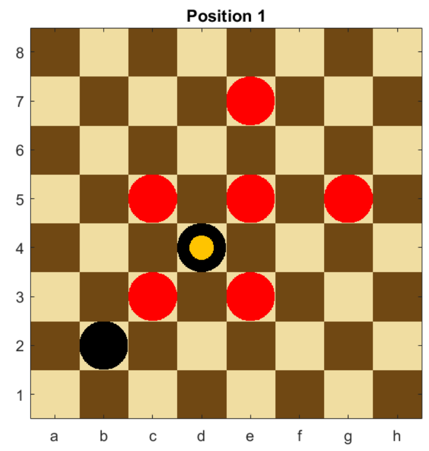
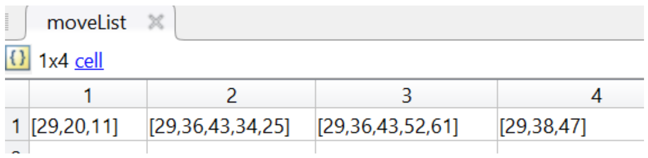


Figure 8: Test Position 1.

Figure 9 shows the moves generated for black by *generateMovesPlayer()*. We can see that we generated the correct number of moves (4), and that the moves are all as described. Note that *moveList* stores moves via array index (integers), and that it includes the indices of pieces being captured (as described in section 4.1.2).


moveList

1x4 cell

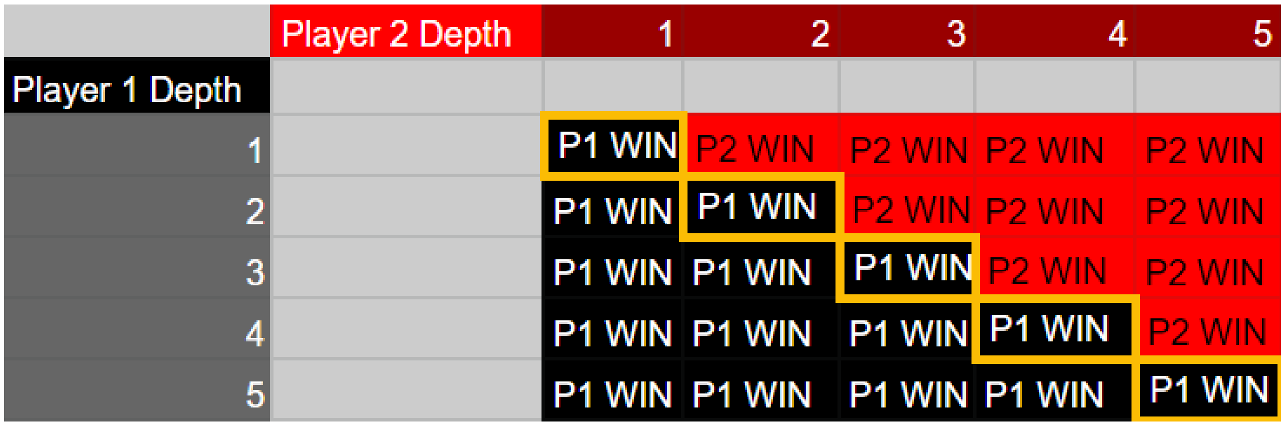
	1	2	3	4
1	[29,20,11]	[29,36,43,34,25]	[29,36,43,52,61]	[29,38,47]

Figure 9: Generated *moveList* for Test Position 1.

5.2 AI Proficiency

We can now evaluate the proficiency of our AI. Proficiency refers to the ability of our AI to perform well in checkers. Evaluating this aspect of the AI is more difficult than it may initially seem. Simply having the author playing against the AI would not be very scientific, as human ability (especially in a novice player) tends to vary wildly between games. A more scientific evaluation process (which we will use) is to have the AI play against itself at different depths. For example, having our player 1 AI at depth 2 and our player 2 AI at depth 3. As the AIs decision making progress is deterministic (due to the nature of the algorithm), we should only need to repeat this process once for each matchup. All subsequent matchups at the same depths will result in the exact same game.

Figure 10 shows the results of this testing in a tournament grid style. The depths for player 1 are rows and the depths of player 2 are columns. Then, where the row and column intersect is the result of the match. For example, if we wanted to find the results for the matchup with player one having depth 3 and player two having depth 4, we would look in cell [3,4].



	Player 2 Depth	1	2	3	4	5
Player 1 Depth						
1		P1 WIN	P2 WIN	P2 WIN	P2 WIN	P2 WIN
2		P1 WIN	P1 WIN	P2 WIN	P2 WIN	P2 WIN
3		P1 WIN	P1 WIN	P1 WIN	P2 WIN	P2 WIN
4		P1 WIN	P1 WIN	P1 WIN	P1 WIN	P2 WIN
5		P1 WIN	P1 WIN	P1 WIN	P1 WIN	P1 WIN

Figure 10: Tournament Chart of the AI Playing Against Itself at Different Depths.

The results captured in figure 10 show that when one player has a higher algorithm depth than the other, the player with the higher depth will always win. In the case that the two players have the same algorithm depth, player 1 will always win. The trend of the player with higher depth always winning is what we would expect. We expect our AI to become more proficient the further in the future it can look, so it should always beat an AI opponent who can not look as far into the future. The trend on the diagonal (tied player depths) is more interesting however. The fact that player 1 always wins when the two players have the same algorithm depth reveals an interesting fact about checkers that the first player has an inherent advantage. This seems logical, as player 1 is always one move ahead in the game, which should allow them to more quickly get into a good position.

While these results do show that the AI plays better at higher depths, it still does not show that the AI is playing well. It is possible that the AI is playing poorly at all depths, but 'better' at higher ones. In order to ensure ourselves that the AI is in fact playing at a decent level, the author will attempt to defeat the AI in battle. The author is not a great checkers player, but feels that he does represent an average human beings checkers ability. The author found that he struggled to defeat the AI even at depths as low as 2.

Playing these games be played allowed for some interesting observations about the AIs behavior. The AI has an extreme willingness to sacrifice its own pieces to achieve better position on the board. The AI is also quite good at setting up tricky forced-capture scenarios for the enemy that result in large chain captures for the AI. These are quite advanced tactics for human players, and the author feels that the AI would be a formidable opponent for even experienced players when ran at higher (4+) evaluation depths.

5.3 Minimax Runtime as a Function of Algorithm Depth

We will also evaluate the run-time of our program as it relates to algorithm depth. As the number of total possible moves expands exponentially with algorithm depth (such can be seen plainly in even the very basic flowchart example in figure 7), we should expect that runtime will follow a similar trend.

To keep things standardized between tests, we will evaluate the same position at every depth. The position used for these tests can be found in figure 11. This position was chosen mostly arbitrarily, but we tried to choose a position where it is 1) not immediately obvious what the best move is, and 2) has enough turns left in the game to evaluate at sufficient depths. We will also evaluate the position from black's point of view (*maximizing = true*), though the results should have little/no change if evaluated with *maximizing = false*.

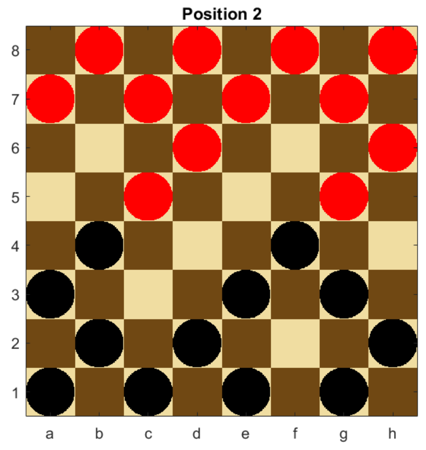


Figure 11: Position Used to Evaluate Runtime

Runtime data was collected using MATLAB's "Run and Time" feature when calling the *minimax()* function with our desired depth. We collected datapoints for all depths between 1 and 8, then plotted them. We also wrote some simple logic to keep track of the total number of moves evaluated for each depth[10], so that we can see how runtime scales with total number of moves evaluated. The results of these tests can be found plotted in figure 12

[10]The logic is just incrementing a counter every time *playMove()* is called, as that function is called every time *minimax()* evaluates a new move.

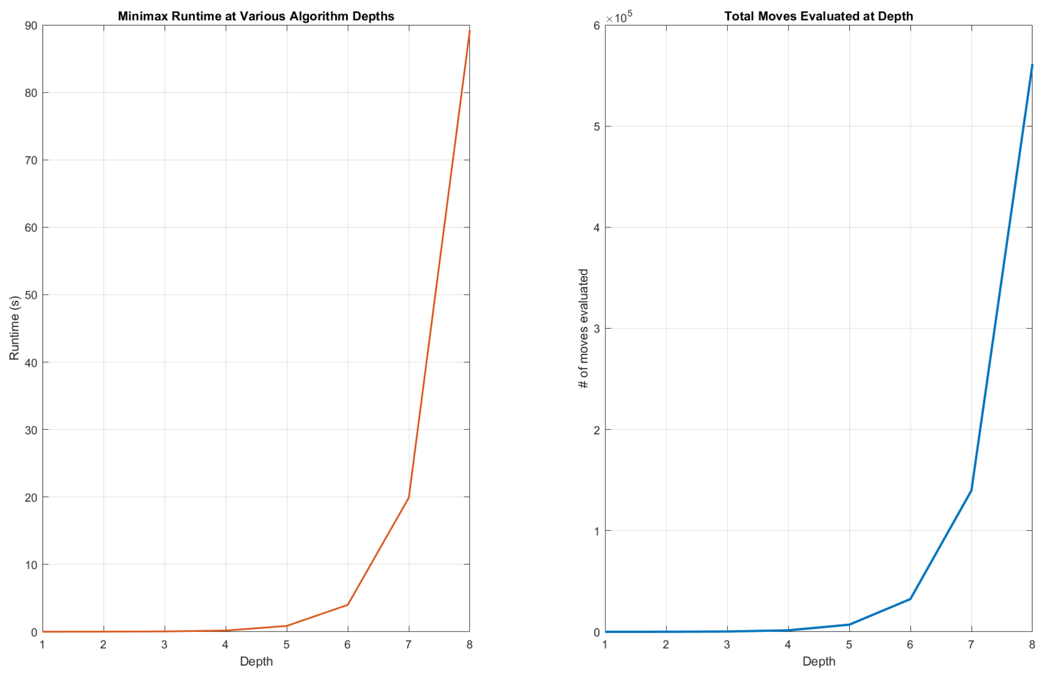


Figure 12: Plot of Runtime and Total # of Moves Evaluated at depths 1 through 8.

We can see from the figure 12 that our hypothesis was correct, and runtime does increase exponentially with algorithm depth. We also see that our runtime curve seems to follow nearly identically with our total number of moves curve. This shows that while runtime scales exponentially with respect to depth, it seems to scale linearly with number of moves. This phrasing of ”the minimax algorithm scales linearly with total number of moves evaluated” is likely more accurate than our statement of runtime scaling exponentially with depth, as in some positions it is possible that a greater depth does not result in more moves needing to be evaluated. Examples of such positions would be when the game is nearly over, and only a limited number of total moves are possible.

6 Summary

We were successfully able to model and implement the game of checkers and an AI capable of playing it in the MATLAB programming language. The ’AI’ was based off of the well known minimax algorithm. Our game implementation was shown to be robust, and our AI followed the trends we expected, namely higher proficiency and runtime at greater evaluation depths.

Interestingly, the most difficult aspect of this project was modeling and implementing the game itself. The implementation of the AI (minimax algorithm) was very simple relative to the sometimes high degree of complexity that the checkers game required. The AI model could certainly be improved much further, with one notable possible improvement being the implentation of alpha-beta pruning [2]. Alpha-beta pruning is an optimization technique for the minimax algorithm that allows the program to quickly disregard (prune) move paths that will not result in relevant evaluations. This project will not explain the logic of this process, but it if implemented it is expected that this would greatly reduce the number of total positions that must be evaluated by *minimax()*, in turn greatly reducing the runtime of the function.

This project was a very interesting exploration into how complex behaviors can arise out of relatively simple rules. When playing against the author, the AI could be seen to be utilizing what many would consider to be advanced checkers tactics. The project was also an interesting exploration into the limitations of such a technique as observed by our data collected on runtime. This tradeoff between performance and runtime is a very common problem encountered in computing.

7 References

References

[1] chessprogramming.org. *Chess Engine-Testing Positions.* URL: `https://www.chessprogramming.org/Test-Positions`. Accessed 4.18.2022.

[2] Wikipedia. *Alpha-beta Pruning.* URL: `https://en.wikipedia.org/wiki/Alpha-beta_pruning`. Accessed 4.18.2022.

[3] Wikipedia. *Checkers.* URL: `https://en.wikipedia.org/wiki/Checkers`. Accessed 4.9.2022.

[4] Wikipedia. *Minimax.* URL: `https://en.wikipedia.org/wiki/Minimax`. Accessed 4.9.2022.

A MATLAB Code for *newGame()*

The following is the complete MATLAB code for the *newGame()* function described and used in the project.

```
%% newGame()
% No input or outputs
% Initializes the important variables used by other functions in the
% checkers game.
function newGame()
clear;clf;
global logicBoard
global offsets
global repeat

% Initialize board
logicBoard = zeros(8,8);

% Initialize repeat
repeat(:,:,1) = logicBoard;
repeat(1,1,2) = 0;

% Initialize offsets used in generateMovesPlayer() and playMove()
offsets = struct;
offsets.ul = -9;
offsets.ur = 7;
offsets.bl = -7;
offsets.br = 9;

% Build the board
for r = 1:3
    for c = 1:8
        if mod(r+c, 2) == 0
            %Creating player 1 (bottom of board). Player 1 has pawns on
            %bottom three rows where r+c is even.
            logicBoard(r+5,c) = 1;
        else
            %Creating player 2 (top of board). Player 2 has pawns on
            %top three rows where r+c is odd.
            logicBoard(r,c) = 2;
        end
    end
end
end
```

B MATLAB Code for *generateMovesPlayer()*

The following is the complete MATLAB code for the *generateMovesPlayer()* function described and used in the project.

```
%% generateMovesPlayer(player)
% Generates cell array of all possible moves. Each index in moveList
% represents a possible moves. the moves are coded in vectors. the first
% index in the vector specifies the piece, and then each succesive index
% specifies its path. In cases of multiple captures, there will be several
% 'steps' on the path
%
function generateMovesPlayer(player)
global logicBoard
global moveList
global offsets

% First, create an empty moveList. The moveList is a cell array holding all
% possible moves. moveList{1} holds an array that specifies the indices of
% moveList{} where a capture is possible. In checkers you HAVE to play a
% capture move if one is available to you
%
% indices 2:end of moveList{} hold the moves as a vector. Moves are
% formatted as follows:
%
% moveList{move#} = [index of piece to move, index of space to move to]
%
% The vector will be more than 2 long if we move multiple spaces in a move,
% when capturing for example. When capturing, we move on top of the piece,
% then follow through to the open square past it. The move vector gets even
% longer for chain captures, formatted the same way.
moveList =  {[]};

for i = 1:64

% These are the indexes of the squares immediately diaganol to our piece.
ur = i+offsets.ur;
ul = i+offsets.ul;
br = i+offsets.br;
bl = i+offsets.bl;


% Note about the function below: To avoid having a bunch of edge case
% checks for when we are on the sides of the board, we use try-catch
% statements. If an index fails (because we are on a side), we just fall
% into the catch which just exits the current evaluation.
switch logicBoard(i)
```

```
% If black piece, do player ones rules
case {1, 11}
    % Only evaluate this if we want moves for player 1
    if player
        if mod(i-1, 8) ~= 0 % If we arent on the top edge of the board

        % First, evaluate the upper left square to see if we can move
        % to it or capture a piece on it
        try
        switch logicBoard(ul) %

            % If the space is empty we can move to it, so write that as
            % a possible move to the moveList
            case 0
                moveList{end+1} = [i, ul];

            % If the space has an enemy piece on it, we might be able
            % to capture
            case {2, 22}
                try
                % If the space past the enemy piece is open, we can
                % capture!
                if logicBoard(ul+offsets.ul) == 0 && mod(i-1,8) > 1
                    % Now we need to see if we can perform a chain
                    % capture. This evaluation is a little complex, so
                    % we call a different function. read evalHop() for
                    % explanation of parameters
                    evalHop(ul+offsets.ul, [i,ul,ul+offsets.ul], true)
                end
                catch
                end
        end
        catch
        end


        % Now do the same thing for the upper right square. The process
        % is all the same so comments will be sparse.
        try
        switch logicBoard(ur)
            case 0 % If open square
                moveList{end+1} = [i, ur];
            case {2, 22} % If enemy piece, see if cappable
                try
                if logicBoard(ur+offsets.ur) == 0 && mod(i-1,8) > 1
                    evalHop(ur+offsets.ur, [i,ur,ur+offsets.ur], true)
                end
                catch
                end
        end
        catch
        end
        end

        % IF OUR PIECE IS A KING: then we need to check the other two
        % directions
        if mod(i,8) ~= 0 && logicBoard(i) == 11
        try
        switch logicBoard(bl) % Check the left side jump
            case 0 % If open square
                moveList{end+1} = [i, bl];
            case {2, 22}
                try
                if logicBoard(bl+offsets.bl) == 0 && mod(i,8) < 7
                    evalHop(bl+offsets.bl, [i,bl,bl+offsets.bl], true)
                end
                catch
                end
        end
        catch
        end
        try
        switch logicBoard(br) % Check the right side jump
            case 0 % If open square
                moveList{end+1} = [i, br];
            case {2,22} % If enemy piece, see if cappable
                try
                if logicBoard(br+offsets.br) == 0 && mod(i,8) < 7
                    evalHop(br+offsets.br, [i,br,br+offsets.br], true)
                end
                catch
                end
        end
        catch
        end
        end
    end


%% Player 2 evaluation. Same as player 1 but we look down instead of up.
    case {2, 22}
        if ~player
if mod(i,8) ~= 0
try
switch logicBoard(bl) % Check the left side jump
    case 0 % If open square
        moveList{end+1} = [i, bl];
    case {1, 11}
        try
        if logicBoard(bl+offsets.bl) == 0 && mod(i,8) < 7
            evalHop(bl+offsets.bl, [i,bl,bl+offsets.bl], false)
        end
        catch
        end
end
catch
end
try
switch logicBoard(br) % Check the right side jump
    case 0 % If open square
        moveList{end+1} = [i, br];
    case {1,11} % If enemy piece, see if cappable
        try
        if logicBoard(br+offsets.br) == 0 && mod(i,8) < 7
            evalHop(br+offsets.br, [i,br,br+offsets.br], false)
        end
        catch
        end
end
catch
end
end

% Now check other direction for kings
if mod(i-1, 8) ~= 0 && logicBoard(i) == 22
try
switch logicBoard(ul) % Check the left side jump
    case 0 % If open square
        moveList{end+1} = [i, ul];
    case {1, 11}
        try
        if logicBoard(ul+offsets.ul) == 0 && mod(i-1,8) > 1
            evalHop(ul+offsets.ul, [i,ul,ul+offsets.ul], false)
        end
        catch
        end
end
catch
```

```
            end
            try
            switch logicBoard(ur) % Check the right side jump
                case 0 % If open square
                    moveList{end+1} = [i, ur];
                case {1, 11} % If enemy piece, see if cappable
                    try
                    if logicBoard(ur+offsets.ur) == 0 && mod(i-1,8) > 1
                        evalHop(ur+offsets.ur, [i,ur,ur+offsets.ur], false)
                    end
                    catch
                    end
            end
            catch
            end
            end
        end
end
end

% Now, moveList{1} stores the indices in moveList{} where forced captures
% lay. Forced captures MUST be played if available, so they are the only
% legal moves. Only pass legal moves.
if ~isempty(moveList{1})
    moveList = moveList(moveList{1});

% If there are no forced captures, chop off the first index of moveList(),
% as it does not contain any moves.
else
    moveList = moveList(2:end);
end
```

C MATLAB Code for *evalHop()*

The following is the complete MATLAB code for the *evalHop()* function described and used in the project.

```
%% evalHop
% Special function for evaluating captures. Used to detect and
% write capture chain moves to moveList.
%
% i = index of piece AFTER performing a hop
% move = array of piece location history
% player = true if p1, false if p2.
%
% This function is designed to be called recursively
```

```
%
function evalHop(i, move, player)
global logicBoard
global moveList
global offsets




% Define locations around us
ur = i+offsets.ur;
ul = i+offsets.ul;
br = i+offsets.br;
bl = i+offsets.bl;

fail = 0; % Fail keeps track of whether or not our chip has run out of
          % captures. Failing twice writes the move string to
          % moveList{}.

% We need to take some precautions. Because a king can go in
% any direction it will jump a piece, then, when checking for another
% piece to capture, it will see the same piece and capture it again.
% So on and so forth ad infinitum. For this reason, we need to
% create a temporary copy of hopBoard where all the move indexes listed
% in move() are set to zero, so that it doesnt see them as potential
% captures any more.
% We keep hopBoard(move(1)) the same because it holds useful information
% about the piece we are playing.
hopBoard = logicBoard;
hopBoard(move(2:end)) = 0;




% The try-catch statements below let us escape the horror of trying to make
% sure that the index of our hopBoard is always a positive number. Now if
% i is negative, we simply error out and catch it by adding one to our fail
% counter
switch player
    case true % For player 1

        % If there is a piece to caputure to our upper left, and we can
        % capture it without going off the board
        try
        if ((hopBoard(ul) == 2) || (hopBoard(ul) == 22)) && ...
        mod(i-1,8) > 1
        try

        % And if there is an empty spot to jump to past it.
        if hopBoard(ul+offsets.ul) == 0

            % Take the capture and see if we can cap again (recursive)
            moveWIP = [move, ul, ul+offsets.ul];
            evalHop(ul+offsets.ul, moveWIP, true)

        % In any other situation, fail additional capture
        else
            fail = fail+1;
        end
        catch
            fail = fail+1;
        end
    else
        fail = fail+1;
    end
    catch
        fail = fail+1;
    end



    % Now do the same thing, but for the upper right spot
    try
    if ((hopBoard(ur) == 2) || (hopBoard(ur) == 22)) &&...
        mod(i-1,8) > 1

        try
        if hopBoard(ur+offsets.ur) == 0

            moveWIP = [move, ur, ur+offsets.ur];
            evalHop(ur+offsets.ur, moveWIP, true)

        else
            fail = fail+1;
        end
        catch
            fail = fail+1;
        end
    else
        fail = fail+1;
    end
```

```
catch
    fail = fail+1;
end

% If there were no other available captures (fail ==2), write the
% move sequence to moveList and go home. However, only do this for
% pawns. Kings need to fail 4 times. move(1) holds the index of the
% playing piece.
if fail == 2 && hopBoard(move(1)) == 1
    moveList{end+1} = move;
    moveList{1} = [moveList{1}, length(moveList)];
end

% Now, check the other direction if our piece is a king. Kings are
% denoted by 11 or 22. move(1) holds the index of the playing piece.
if hopBoard(move(1)) == 11
try
if ((hopBoard(bl) == 2) || (hopBoard(bl) == 22)) &&...
    mod(i,8) < 7
    try
    if hopBoard(bl+offsets.bl) == 0
        moveWIP = [move, bl, bl+offsets.bl];
        evalHop(bl+offsets.bl, moveWIP, true)
    else
        fail = fail+1;
    end
    catch
        fail = fail+1;
    end
else
    fail = fail+1;
end
catch
    fail = fail+1;
end

try
if ((hopBoard(br) == 2) || (hopBoard(br) == 22)) &&...
    mod(i,8) < 7
    try
    if hopBoard(br+offsets.br) == 0
        moveWIP = [move, br, br+offsets.br];
        evalHop(br+offsets.br, moveWIP, true)
    else
        fail = fail+1;
    end
```

```
            catch
                fail = fail+1;
            end
        else
            fail = fail+1;
        end
        catch
            fail = fail+1;
        end


        if fail == 4
            moveList{end+1} = move;
            moveList{1} = [moveList{1}, length(moveList)];
        end
        end
```

```
%% Player 2 rules. Same concept as player 1 rules
    case false

        try
        if ((hopBoard(bl) == 1) || (hopBoard(bl) == 11)) &&...
            mod(i,8) < 2
            try
            if hopBoard(bl+offsets.bl) == 0
                moveWIP = [move, bl, bl+offsets.bl];
                evalHop(bl+offsets.bl, moveWIP, false)
            else
                fail = fail+1;
            end
            catch
                fail = fail+1;
            end
        else
            fail = fail+1;
        end
        catch
    fail = fail+1;
end

try
if ((hopBoard(br) == 1) || (hopBoard(br) == 11)) &&...
    mod(i,8) < 7
    try
    if hopBoard(br+offsets.br) == 0
        moveWIP = [move, br, br+offsets.br];
        evalHop(br+offsets.br, moveWIP, false)
    else
        fail = fail+1;
    end
    catch
        fail = fail+1;
    end
else
    fail = fail+1;
end
catch
    fail = fail+1;
end


if fail == 2 && hopBoard(move(1)) == 2
    moveList{end+1} = move;
    moveList{1} = [moveList{1}, length(moveList)];
end



% Now, check the other direction if our piece is a king. Kings are
% denoted by 11 or 22. move(1) holds the index of the playing piece
if hopBoard(move(1)) == 22
try
if ((hopBoard(ul) == 1) || (hopBoard(ul) == 11)) &&...
    mod(i-1,8) > 1
    try

    % And if there is an empty spot to jump to past it.
    if hopBoard(ul+offsets.ul) == 0

        % Take the capture and see if we can cap again (recursive)
        moveWIP = [move, ul, ul+offsets.ul];
        evalHop(ul+offsets.ul, moveWIP, false)
    % In any other situation, fail additional capture
    else
        fail = fail+1;
    end
    catch
        fail = fail+1;
    end
else
    fail = fail+1;
end
catch
    fail = fail+1;
end



% Now do the same thing, but for the upper right spot
try
if ((hopBoard(ur) == 1) || (hopBoard(ur) == 11)) &&...
    mod(i-1,8) > 1

    try
    if hopBoard(ur+offsets.ur) == 0

        moveWIP = [move, ur, ur+offsets.ur];
        evalHop(ur+offsets.ur, moveWIP, false)

    else
        fail = fail+1;
    end
    catch
        fail = fail+1;
    end
else
    fail = fail+1;
end
catch
    fail = fail+1;
end


if fail == 4
    moveList{end+1} = move;
    moveList{1} = [moveList{1}, length(moveList)];
end
end
```

```
end
end
```

D MATLAB Code for *playMove()*

The following is the complete MATLAB code for the *playMove()* function described and used in the project.

```
%% playMove(move)
% Plays a move by updating the logicBoard. Also upgrades pieces to kings.
%
% move is a vector with minimum length two from moveList{}
%
function playMove(move)
global logicBoard

% Play the move by dragging the piece across the board. This also
% eliminates any pieces captured.
logicBoard(move(end)) = logicBoard(move(1));
logicBoard(move(1:end-1)) = 0;

% After each move, check for kings
for c = 1:8
    if logicBoard(1,c) == 1
        logicBoard(1,c) = 11;
    end
    if logicBoard(8, c) == 2
        logicBoard(8,c) = 22;
    end
end
end
```

E MATLAB Code for *drawBoard()*

The following is the complete MATLAB code for the *drawBoard()* function described and used in the project.

```
%% drawBoard()
% draw board based on current game state. this board is stored in
% imageBoard.
function imageBoard = drawBoard()
global logicBoard
global imageBoard
res = 100;
res = res*8;
```

```
% Initialize our image array (imageBoard)
imageBoard = zeros(res);

% These arrays make light and dark squares, and are slid into imageBoard in
% the following for loop to create the checkerboard.
boardDark = 3*ones(res/8);
boardLight = 4*ones(res/8);




% Create checker background pattern
for r = 1:8 % For each row
    for c = 1:8 % and each column
        % The following operations grab an eighth of the board at a time,
        % then set it to its correct color
        if mod(r+c,2) == 0
            imageBoard(1+(r-1)*res/8:r*res/8, 1+(c-1)*res/8:c*res/8) = boardDark;
        else
            imageBoard(1+(r-1)*res/8:r*res/8, 1+(c-1)*res/8:c*res/8) = boardLight;
        end
    end
end

% This is super jank, but for the colormap to work correctly we need
% atleast one pixel of each color is on the board at all times.
% This ensures that happens.
imageBoard(1) = 5;
imageBoard(res) = 1;
imageBoard(end) = 2;




% Creating a 'circular' array mask used to paint the chips on to imageBoard
x = 1:res/8;
y = 1:res/8;
cx = res/8/2; % circle center
cy = res/8/2; % circle center
r = res/8/2; % radius
mask =((x-cx).^2 + (y'-cy).^2) < r^2  ; % Creating a mask


% Now create a another circular mask with half the radius. This is used to
% signify a piece as a king
```

```
kingMask =((x-cx).^2 + (y'-cy).^2) < (r/2)^2  ; % Creating a mask



% Now draw the chips on the board. We find piece locations from logicBoard,
% then paint them on to the corresponding positions in imageBoard.
for r = 1:8
    for c = 1:8
    switch logicBoard(r,c)
        % Draw our pawns
        case {1, 2}
            % We take the 'tile', spraypaint the checker over it, then load
            % it back into the imageBoard
            tile = imageBoard(1+(r-1)*res/8:r*res/8, 1+(c-1)*res/8:c*res/8);
            tile(mask) = logicBoard(r,c);
            imageBoard(1+(r-1)*res/8:r*res/8, 1+(c-1)*res/8:c*res/8) = tile;

        % For Kings we first paint the pawn, then paint the king gold over
        % it
        case {11, 22}
            tile = imageBoard(1+(r-1)*res/8:r*res/8, 1+(c-1)*res/8:c*res/8);
            tile(mask) = logicBoard(r,c)/11;
            tile(kingMask) = 5;
            imageBoard(1+(r-1)*res/8:r*res/8, 1+(c-1)*res/8:c*res/8) = tile;
    end
    end
end

% Build custom colormap
% color order is: player1, player2, boardDark, boardLight, king color
colors = [0, 0, 0; 255, 0, 0; 112, 72, 19;240, 221, 161; 255, 196, 0];
colors = colors./255;

% Draw our board image
colormap(colors)
imagesc(imageBoard)
% set(gca, 'YTickLabel', [], 'XTickLabel', [], 'xtick', [], 'ytick', [])
ticks = [1:8]*res/8-50;
xticks(ticks)
yticks(ticks)
xticklabels({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'})
yticklabels({'8', '7', '6', '5', '4', '3', '2', '1'})
axis square
end
```

F MATLAB Code for *minimax()*

The following is the complete MATLAB code for the *minimax()* function described and used in the project.

```matlab
%% minimax(depth, maximizing)
% This function contains our implementation of the minimax function, and
% therefore is the logic for our AI opponent
%
% depth is algorithm depth greater than or equal to 0
% maximizing is a boolean, true to evaluate from maximizing players
% perspective, false for minimizing.
%
function [eval, bestMove] = minimax(depth, maximizing)
global logicBoard
global moveList

bestMove = 0; % Placeholder value for bestMove

% If at bottom of depth, perform static evaluation of the board and return
% that value.
if depth == 0
    eval = 0;
    for i = 1:64
        switch logicBoard(i)
            case 1
                eval = eval+3;
            case 11
                eval = eval+5;
            case 2
                eval = eval-3;
            case 22
                eval = eval-5;
        end
    end
    return
end



% Lets get some moves in here. maximizing tells us player info
generateMovesPlayer(maximizing);

% Because I made the terrible mistake of using globals, store these for
% later
logicStore = logicBoard;
moves = moveList;
```

```
% If we are the maximizing player, then play all of our moves and perform
% minimax on each of those children with depth-1.
if maximizing
    maxEval = -inf;
    eval = maxEval;
    for i = 1:length(moves)
        % We need to reload our logicBoard every time because globals lol
        logicBoard = logicStore;
        playMove(moves{i})
        eval = minimax(depth-1, false);
        if eval > maxEval
            maxEval = eval;
            bestMove = moves{i};
        end
        % Lets choose randomly which move to use if they are equally good
        if (eval == maxEval) && (randi([0, 1], 1, 1) == 1)
            bestMove = moves{i};
        end

    end
    logicBoard = logicStore;
    moveList = moves;
    return

% Minimizing case
else
    minEval = inf;
    eval = minEval;
    for i = 1:length(moveList)
        logicBoard = logicStore;
        playMove(moves{i})
        eval = minimax(depth-1, true);
        if eval < minEval
            minEval = eval;
            bestMove = moves{i};
        end
        % Lets choose randomly which move to use if they are equally good
        if (eval == minEval) && (randi([0, 1], 1, 1) == 1)
            bestMove = moves{i};
        end
    end
    logicBoard = logicStore;
    moveList = moves;
    return
end
end
```