Edinburgh Napier UNIVERSITY

Coursework Report

Dan Gillespie
10004946@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

1 Introduction

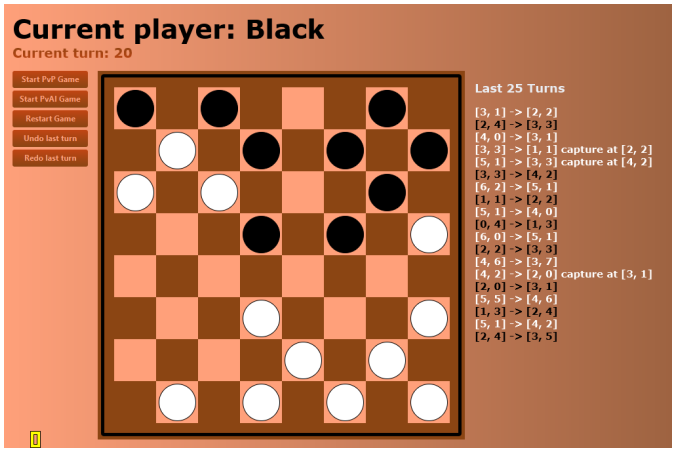
The purpose of this report is to present the design and implementation of a Checkers game, specifically with note to the data structures and algorithms used. The game was implemented using Java and any GUI elements used the JavaFX library. In deterministic games such as checkers, the program must accurately represent changes to the game state based on user input and the data structures that allow for these changes to take place are a key component of the software design. In encapsulating common data elements such as players moves and game states into data objects it allows additional functionality and game logic to be implemented easily. An example being that a collection of previous board states together inside an ArrayList allows implementation of being able to undo moves and revert the game to the previous turn.

Additionally, the game makes use of algorithms to process game data as well as make decisions. A large part of the game development was focused on implementing algorithms that help to manipulate the data structures that model the game board. Part of the validation for a player making a move involves the generation of a list of legal moves every turn, which is created by an algorithm that loops through the board and analyses every game piece location. A possibly more obvious usage for an algorithm comes in the form of an evaluation function for the games intelligent agent player. The game supports both player vs player mode and player vs AI mode, which makes use of the evaluation function to score the previously mentioned list of legal moves based on their effect on the game board. There are a number of heuristics that go into scoring a move which while might not be perfect, present results that are impressively challenging for the human player.

2 Design

2.1 Data Structures

The general software structure of the game consists of a Board class that encapsulates all data of the current game state, which is interacted with through a JavaFX graphical user interface. The GUI displays the games current board state as well as other options to reset the game, undo the last turn and fast forward turns that have been undone. The game can be played by both two players against each other, as well as a single player against an intelligent agent function that selects moves based on evaluating future board states.



Game Screen- **Graphical User Interface**

Although the GUI is not a focal point of this report, it is useful to quickly explain how it interacts with the rest of the program. By selecting one of their game pieces on the board GUI, a player can then click a square on the board where the program determines whether the player has issued a valid move. If the player issues a valid move the board will refresh displaying the new game state and either begin calculating the AI's turn or default to the next human player's turn. When the GUI is rendered, the squares on the board are bound with on-click functions that identify where on the grid the player is clicking to make their move.

From a design perspective, all moves done by either the player or AI are encapsulated behind their own data object. A move object contains the board location of the piece being moved (a row:column combination between 0-7 in each direction), the target board location, the player issuing the move and finally whether the move resulted in the capture of another player's game piece. By doing this it serves the purpose of simplifying the structure of the code by simply having to pass the single data object through functions parameters as well being able to use moves as part of collection objects such as lists, arrays and stacks.

Listing 1: Move constructor

```
1
2   public Move (int player, int fromRow, int fromColumn, int ↩
      targetRow, int targetColumn) {
3     this.player = player;
4     this.fromRow = fromRow;
5     this.fromColumn = fromColumn;
6     this.targetRow = targetRow;
7     this.targetColumn = targetColumn;
8     this.isCapture = (fromRow - targetRow == 2 || fromRow↩
      - targetRow == -2);
9   }
10
```

The data structure chosen to model the board is simple in that it is very easy to visualize. By using the built in array

structure provided by Java, it was simple to declare a 8 by 8 2D array where each element in the array has a certain value representing the state of the square on the board it is representing. Additionally, Java arrays require the size to be specified upon declaration and are not dynamic, this is useful as we do not intend to resize the array at any point.

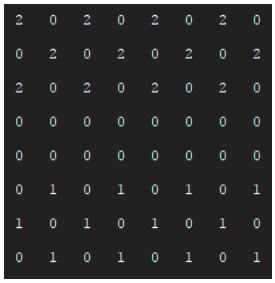
Listing 2: Board array declaration

```
1 private int[][] board = new int[8][8];
```

Accessing elements in this array can be thought of accessing values in cells of a grid, using the first accessing value as the intended row and the second as the column.

Possible cell values:

- 0 - Empty
- 1 - White game piece
- 2 - Black game piece
- 3 - White king game piece
- 4 - Black king game piece



Game Screen- **The starting game board state**

A square at board[0][3] would be 1 row and down 4 columns across and might have a value of 3 which would indicate that the square contains a white king piece. On the start of a game a function takes the empty 2D array and loops through all its elements to set the default values associating with the starting board of a game of checkers. On having moves issued on it, a board object will change the values in the array to represent the change in the board. In addition to the board array state, the greater board object also contains various other data and data structure to model the game state.

In addition to basic values such as the current player and how many turns have been carried out, there is an ArrayList data structure that contains a collection of all of the moves until this point that have been issued on the board. This is done so that once the game has ended, all of the moves taken in the game are displayed to the user in the order they were carried out. Whenever a move is added to this arraylist it is added to the last index of the array-list, meaning iterating from the starting index over the list displays the sequential order.

Listing 3: ArrayLists that will act as stacks for the undo and redo features

```
1    private ArrayList<Board> gameStates;
2    private ArrayList<Board> gameStatesRedo;
```

Each time a Board object has a Move issued to it, it changes state. Part of the required functionality for interacting with the game board is that players must have the ability to both undo the previous turn multiple times as well as redo them. This can be implemented fairly simply by having two ArrayLists being used functionally as stacks. ArrayLists were chosen due to their dynamic properties when adding elements to them. Unlike standard java array types, the size of an ArrayList does not need to be declared and the functionality for adding and removing elements is abstracted behind the ArrayList type. ArrayLists are very multi-purpose in that they come with methods that lets them be used like other data objects. The stack is a well known data structure that only allows the adding or removal of elements to one side of the collection. By limiting our usage of ArrayList functionality we can use the ArrayList as if it were a stack structure by just removing and adding elements to one side of the list. In the example below it can be seen that for the 'gameStates' ArrayList the elements are only added on the end of the list, but for the 'gameStatesRedo' ArrayList they are added and removed from the 0 index of the collection.

Much like how the moves issued on a board were stored in the board object, every time a move is issued, a clone of the previous board object is added to one of the stacks. When filling up this stack has the purpose of containing all of the game states up until that point, the the last element of the ArrayList is considered the top of the stack. When a player wishes to undo the previous turn, the GUI class that contains the current in-scope board obtains an object copy of the board to the second previously declared stack containing what can be considered 'future turns'. Additionally the first element in the previous turns stack is 'popped' and added to the 'future turns' stack. If the player continues to undo turns, each time it will pop the previous board and add it to the future stack.

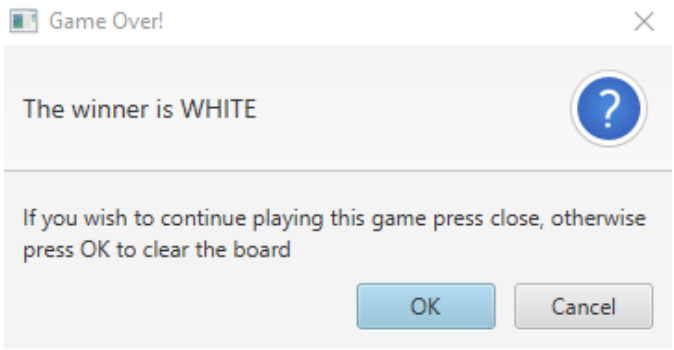
Listing 4: Functions moving board states between stacks

```
1
2    private void backOneTurn() {
3        if (gameStatesRedo.isEmpty()) {
4            gameStatesRedo.add(currentBoard.cloneBoard());
5        }
6        this.currentBoard = gameStates.get(gameStates.size()-1)↩
         .cloneBoard();
7        gameStatesRedo.add(0, gameStates.get(gameStates.size()↩
         -1).cloneBoard());
8        if (gameStates.size() != 1) {
9            gameStates.remove(gameStates.size()-1);
10       }
11       paintBoard();
12   }
13
14   private void forwardOneTurn() {
15       gameStates.add(gameStatesRedo.get(0).cloneBoard());
16       this.currentBoard = gameStatesRedo.get(0);
17       gameStatesRedo.remove(0);
18       paintBoard();
19   }
```

The top of the stack can be considered the game board in scope when undoing turns. When redoing turns the opposite operation comes into effect, except in this case the 'top' of the stack used to represent 'future' turns is the element 0. Each board is popped from the 0 element and added into scope at the top of the other stack which is in turn rendered by the GUI. If at any time during undoing turns a player decides to make a turn, the 'future' turns list is cleared as they are no longer relevant to the game anymore.

Once the game has been concluded, players will be greeted

by a prompt window, giving them the option to restart the game or to continue on the same game and be able to undo and redo turns from any point that they wish.



End-game screen

2.2 Algorithms

Making moves in the game is handled by an inner function inside the board class, by passing it a move as a parameter it alters the state of board's 2D array to the result of the applied move.

Listing 5: The doMove function that processes player moves

```
1   private void doMove(int[][] board, Move move) {
2       board[move.targetRow][move.targetColumn] = board[move↩
        .fromRow][move.fromColumn];
3       board[move.fromRow][move.fromColumn] = EMPTY;
4
5       if (move.targetRow == 0) {
6           if (board[move.targetRow][move.targetColumn] == ↩
        WHITE) {
7               board[move.targetRow][move.targetColumn] = ↩
        WHITE_KING;
8           }
9       }
10      if (move.targetRow == 7) {
11          if (board[move.targetRow][move.targetColumn] == ↩
        BLACK) {
12              board[move.targetRow][move.targetColumn] = ↩
        BLACK_KING;
13          }
14      }
15      if (move.isCapture()) {
16          int middleRow = (move.fromRow + move.targetRow) ↩
        / 2;
17          int middleColumn = (move.fromColumn + move.↩
        targetColumn) / 2;
18          board[middleRow][middleColumn] = EMPTY;
19      }
20  }
```

The validation for whether a move is valid is handled by comparing any potential moves to a list of legal moves generated each turn for the current player in scope. The algorithm that calculates all of the current legal moves iterates over each square on the board and based on the value of each element, further iterates around the surrounding squares creating potential move objects that are passed into an evaluation function to determine whether they are legal. Legal moves are moves that adhere to the rules of the game, ie. basic game pieces can only move in certain directions and be moved by certain players. The only way a player can complete their turn is by correctly selecting a move on the GUI that is contained in the legal moves generated by this algorithm.

Listing 6: Function to check whether a move is valid

```
1   private boolean isLegalMove(int[][] boardState, int player, int ↩
      fromRow, int fromColumn, int targetRow, int targetColumn↩
      ) {
2
3       if (targetRow < 0 || targetRow >= 8 || targetColumn < 0↩
        || targetColumn >= 8) {
4           return false;
5       }
6       if (boardState[targetRow][targetColumn] != EMPTY) {
7           return false;
8       }
9       if (player == WHITE) {
10          return boardState[fromRow][fromColumn] != WHITE ||↩
        targetRow <= fromRow;
11      }
12      else {
13          return boardState[fromRow][fromColumn] != BLACK ||↩
        targetRow >= fromRow;
14      }
15  }
```

A further usage of this algorithm and part of the purpose of its design corresponds to how the AI turns are evaluated. In single player vs AI games whenever the human player finishes their turn, the evaluation function that determines the best possible move for the AI player to make, uses the list of legal moves to score potential game states given the effect the moves would have on the board.

Listing 7: Snippet of the evaluation function that scores the move based on number of player pieces while also adding additional weight if the player is moving their piece agressively towards the opponents side of the board

```
1
2   int[][] copy = getCurrentBoardClone();
3   doMove(copy, move);
4
5   for (int r = 0; r < 8; r++) {
6       for (int c = 0; c < 8; c++) {
7           if (copy[r][c] == actingPlayer) {
8               moveScore = moveScore + 10;
9               if (actingPlayer == WHITE && r < 4) {
10                  moveScore = moveScore + 5;
11              }
12              else if (actingPlayer == BLACK && r > 3) {
13                  moveScore = moveScore + 5;
14              }
15          }
16      }
17  }
```

For each Move object in the list, the algorithm scores the Move based on various other heuristics such as:

- Whether the move is a capture
- Upon capture would there be a second or more captures
- The move upgraded a game piece to a king
- Aggressive positioning is rewarded
- Putting pieces in capturable positions

By collating a score for each move on its effect it allows the AI to choose the move that maximises its chance of moving the game in their favour.

```
[4, 6] -> [5, 5] Score: 55
[4, 6] -> [3, 5] Score: 25
[5, 3] -> [4, 4] Score: 25
[5, 3] -> [4, 2] Score: 25
[6, 0] -> [5, 1] Score: 55
[6, 2] -> [5, 1] Score: 55
[6, 4] -> [5, 5] Score: 55
[6, 6] -> [5, 5] Score: 55
```

Legal Moves- **with their evaluated Move scores**

In appendix A, the evaluation functions can be seen. Some of the key components of a successful algorithm are its time and space complexity. In the referenced appendix, the code has notations to include the estimate of time and space complexity.

Listing 8: Snippet of the code that runs the evaluation function on every legal move

```
1  public Move calculateBestMove() {
2
3      ArrayList<Move> legalMoves = getLegalMoves(currentPlayer); //Constant run time: O(1)
4      Move bestMove = legalMoves.get(0);
5      int bestMoveScore = getMoveScore(legalMoves.get(0));
6      for (Move m : legalMoves) {     //Linear run time: O(n)
7          int moveScore = getMoveScore(m);
8          System.out.println(m.toString() + " Score: " + moveScore);
9          if (moveScore > bestMoveScore) {
10             bestMoveScore = moveScore;
11             bestMove = m;
12         }
13     }
14     return bestMove;
15 }
```

As seen above, the function getLegalMoves is observed as having a time complexity of O(1). One of the common operations in this game is a nested for loop that loops over all of the squares in the board as seen below.

Listing 9: Common operation

```
1 for (int r = 0; r < 8; r++) {
2     for (int c = 0; c < 8; c++) {
3         //do something
4     }
5 }
```

While being a for-loop, this operation has a constant running time as both loops are bound by the constant of 8 iterations [1]. The entire sequence of calculating the best move results in running this iteration multiple times per move. While having a constant time and space complexity it is worth noting that its effects are likely not completely omittable. Since the 'calculateBestMove' function iterates through all the legal moves we should first calculate the estimated complexity of the function being run on each move.

Referencing Appendix A: getMoveScore function,

- Line 22: getCurrentBoardClone() invokes a deep copy of the current board state 2D array - Constant Time O(1)
- Line 24: doMove() carries out move on copy board - Constant Time O(1)
- Line 26-53: Double nested for loop as mentioned before - Constant Time O(1)
- Line 55-69: Regardless of which block of code is ran based on the if statement, the same invocation of getLegalMoves() function is called which we have defined as having a O(1) complexity. However the result is iterated over based on the size of the result which has a Complexity of O(n).

By simplifying all of these we get a complexity of O(n) for the getMoveScore function. In practice however it is likely this function has close to constant run time as the variation in amount of results from getLegalMoves is quite low.

Going up another level to the parent function 'calculateBestMove', the only O(n) component of the function can be easily identified is being the loop over the legal moves again. This would result in the total space time complexity for the entire running of the evaluation function for a turn at

$$O(n^2) \tag{1}$$

After investigation it seems that as the game advances towards the end, the number of legal moves increases for each player as pieces become more free, this results in larger compute times for this algorithm.

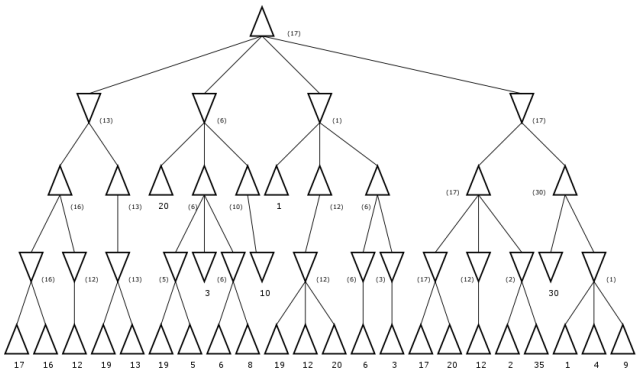
3 Enhancements

There are a number of improvements that could be made that would enhance both the efficiency of the program as well as the game experience and difficulty of the AI player. While the current method of undoing or redoing turns serves its purpose, the same functionality could be accomplished by having an 'undo move' function that would use the same stack data structure manipulation to accomplish the task. By using the already existing list of moves that have been executed on the board, the last move could be passed through a function that reverses the game state to the previous turn and also adding the move to a 'future moves' list. This would result in replacing the entire set of lists that handles cloning/replacing/storing the board states every single turn with something that already exists implemented with a simple function.

Additionally the evaluation function that the AI uses to choose the best move for their turn could be improved to take into account more board conditions. Currently the function scores the board based on a few basic parameters but by adding additional patterns for the function to look for would improve how well the AI performs against the player. Currently the function seems to favour moving king pieces over trying to improve the position of standard pieces. By adding extra weight to the score to incentivise aggressive play by moving pieces to the opponent's side of the board or by upgrading default pieces it would increase the performance of the function and the difficulty of opponent the player would experience.

By continually improving the evaluation function it gets closer to perfectly representing the value of a board state, at which point it would be beneficial to implement a more recursive

version of the current evaluation function. Following a move being evaluated, the board that it creates can be further scored by taking the legal moves on the new board and scoring them. In the case that this board has the opponent as the current player, a board that contains moves with a lower evaluation score would be ideal. By continuing this recursively, we are essentially building a large decision tree that contains the most optimal path to victory for the player. We calculate every single possible sequence of moves that both players can make and play the optimum sequence that limits your opponent's chance of winning. The concept of this algorithm is known as the minimax algorithm[2] and its implementation is focused on maximising your potential move scores, while minimising the opponents.



Minimax decision tree - **each node represents a move and its corresponding evaluation function score[3]**

On this decision tree, each node branches off into the resulting that would be legal having issued the parent move, it creates an extremely large tree depending on how deep you are willing to go.

However implementation of this would cause problems which would mean concessions would have to be made: the space and time complexity would be massive due to incredible amount of possible moves and game states. By limiting the depth of the search of future game states as well as implementing a feature known as alpha beta pruning[4] we could cut down on run times and storage space considerably so that using this algorithm is much more viable. Alpha beta pruning works in conjunction with the minimax function and is designed to reduce the amount of nodes that are evaluated by making logical decisions based on the current state of the search. The algorithm will stop searching a branch once it finds a game move is worse than one that has been previously found resulting in the effect of pruning away branches of the tree and resulting in lower time and space complexities.

Furthermore, using a database of containing optimal moves mapping late game situations would drastically improve the performance of the agent[5]. As the end of the game approaches there is a smaller number of potential game states to calculate compared to the start of the game. In having a reference to the optimal play sequences contained in the database it would ensure that the agent would not have to rely on the evaluation function and would play close to perfectly once past a certain point of the game.

By expanding the options of the game to include varying degrees of AI difficulty, it would potentially improve the player's perception of the game, having a better learning curve if they are an amateur player. This would likely be used as an excuse to refactor the evaluating function to its own class definition, where it can be expanded, and out of the board class.

4 Personal Evaluation

It can be very useful to model data objects in a way that is easy to visualise them in real life and that was the intention with the majority of data structures and objects used in creating this game. The 2D array that represents the game board can be thought of almost exactly like the real life checkers board with squares identified by their coordinates on the grid which in this case are the accessors to the array for the value contained in that square. This makes it very easy to have a high level view of the game logic that makes it simple to implement extra features such as the undo and redo functionality.

Similarly, the Move class that encapsulates the data used in making a move allows for simple and clean functions which can be reused by features that were not intended to do so. An example of this would be that when the intelligent agent is evaluating moves it can use the same functionality by issuing moves on a clone of the current board so as to not alter state of the current board in scope. One of the biggest issues faced in design was exactly this: trying to evaluate the effect of moves without creating a clone of the board object meant that any operations would by reference change the original object. This was solved by having a function inside the Board class that returns the result of a Board class constructor that copies current board values and copies arrays with new copies of lists without reference to the original object.

```
1   public Board cloneBoard() {
2       return new Board(getCurrentBoardClone(), ↩
        getMoveSequenceClone(), currentPlayer, totalTurns);
3   }
4
5   private ArrayList<Move> getMoveSequenceClone() {
6       return new ArrayList<>(this.moveSequence);
7   }
8
9   private int[][] getCurrentBoardClone() {
10      int[][] copy = new int[8][8];
11      for (int i = 0; i < 8; i++) {
12          System.arraycopy(board[i], 0, copy[i], 0, 8);
13      }
14      return copy;
15  }
```

By having a way of cloning the current board it allowed the evaluation function to create mock board objects to evaluate and the GUI class to store instances of previous board states used in the undo and redo functionality.

Originally the idea was to assign a value to each move upon object initialisation that would be a representation of its output from the evaluation function. The intention behind this was to make it easy to compare moves and hence allow the evaluating agent to select the best move quicker. However, during development it became apparent how many Move objects were being created that were not ever required to be evaluated, an example being when obtaining a list of legal moves for the current turn, each move that is created to validate if it is legal would have the evaluation function applied

to it. This seemed like a huge waste of calculation and memory space that could be avoided by just having a function that evaluated a single Move object rather than implementing the evaluation into the Move object constructor.

In using effective version control software such as Git it also allowed development to be very creative and experiment with implementations of features without concern for losing progress. It could be beneficial to use branching to develop new features when using Git, however it was decided against as unlike a situation where the game was developed as part of a group or a series of feature sprints, it was developed alone and all development was fairly linear.

As far as performance is concerned, playing the game is fairly challenging against the intelligent agent and it effectively picks complicated moves that would go unseen to an amateur player. One of the most difficult things in improving the evaluating function is that as it gets more complicated it can be very hard to see the effect of any changes. Outside of having very specific game states and monitoring and modeling the functions response to the state it becomes increasingly hard to improving the function, especially without an expert level of understanding of the game. By watching expert players play the game against other players or by testing the evaluating agent against expert players, data could be gathered on how the agent can optimally score the effect of a move based on a better understanding of what a valuable board state is.

5 Conclusion

By presenting and analysing all of the implementations of algorithms and data structures that went into developing a game of checkers, it provides an interesting insight into software design and modeling real life processes. The goal when designing an application that simulates a real life game is to make them game function almost identically and this is done through careful consideration of the data structures used. By using structures such as a 2D array to model the game board it became easy to build features upon and create a game that was both fun to play and challenging. The real challenge when developing a game like this comes not in the modeling of the game data but in the implementation of evaluation functions that conduct the decision making of the intelligent agent player. By calculating the highest scoring legal move for an AI player through heuristics designed to evaluate a moves effect on a board, the simulation that a human player is playing against a expert player can be achieved.

References

[1] geeksforgeeks, "Analysis of algorithms — set 4 (analysis of loops."

[2] flyingmachinestudios, "An exhaustive explanation of minimax, a staple ai algorithm."

[3] cs.lmu.edu, "Adversarial search: Games."

[4] U. of Cornell, "Minimax search and alpha-beta pruning."

[5] U. of Alberta, "Endgame databases."

6 Appendices

A Evaluation function

The evaluation function that scores a moves potential effect on a game board

```
1
2   public Move calculateBestMove() {
3
4       ArrayList<Move> legalMoves = getLegalMoves(currentPlayer); //Constant run time: O(1)
5       Move bestMove = legalMoves.get(0);
6       int bestMoveScore = getMoveScore(legalMoves.get(0));
7       for (Move m : legalMoves) {     //Linear run time: O(n)
8           int moveScore = getMoveScore(m);
9           System.out.println(m.toString() + " Score: " + moveScore);
10          if (moveScore > bestMoveScore) {
11              bestMoveScore = moveScore;
12              bestMove = m;
13          }
14      }
15      return bestMove;
16  }
17
18  private int getMoveScore(Move move) {
19      int actingPlayer = move.getPlayer();
20      int opponent = actingPlayer == WHITE ? BLACK : WHITE;
21      int moveScore = 0;
22      int[][] copy = getCurrentBoardClone(); //Constant run time: O(1)
23
24      doMove(copy, move); //Constant run time: O(1)
25
26      for (int r = 0; r < 8; r++) {   //Constant run time: O(1)
27          for (int c = 0; c < 8; c++) {  //Constant run time: O(1)
28              if (copy[r][c] == actingPlayer) {
29                  moveScore = moveScore + 10;
30                  if (actingPlayer == WHITE && r < 4) {
31                      moveScore = moveScore + 5;
32                  }
33                  else if (actingPlayer == BLACK && r > 3) {
34                      moveScore = moveScore + 5;
35                  }
36              }
37              if (copy[r][c] == actingPlayer + 2) {
38                  moveScore = moveScore + 20;
39              }
40              if (copy[r][c] == opponent) {
41                  moveScore = moveScore - 10;
42                  if (opponent == WHITE && r < 4) {
43                      moveScore = moveScore - 5;
44                  }
45                  else if (opponent == BLACK && r > 3) {
46                      moveScore = moveScore - 5;
47                  }
48              }
49              if (copy[r][c] == opponent + 2) {
50                  moveScore  = moveScore - 20;
51              }
52          }
53      }
54
55      if (move.isCapture()) {
56          ArrayList<Move> legalMoves = getLegalMoves(copy, currentPlayer);  //Constant run time: O(1)
57          for (Move m : legalMoves) { //Linear run time: O(n)
58              if (m.isCapture()) {
59                  moveScore = moveScore + 20;
60              }
61          }
62      } else {
63          ArrayList<Move> legalMoves = getLegalMoves(copy, opponent);  //Constant run time: O(1)
64          for (Move m : legalMoves) {  //Linear run time: O(n)
65              if (m.isCapture()) {
66                  moveScore = moveScore - 30;
67              }
68          }
69      }
70      if (becomesKing(move)) {
71          moveScore = moveScore + 20;
72      }
73      return moveScore;
74  }
75
```