Alquerque

Danny Nicolai Larsen, Mikkel Brix Nielsen & Steffen Bach

December 9, 2021

Contents

CONTENTS

PUT MAIN TEXT HERE

1 Appendix

1.1 Move class

```
public class Move {
    private int from;
    private int to;
    /**
     * Creates a new move with given origin and destination.
     * @param from the place to move the piece from.
     * @param to the place to move the peiece to.
     */
    public Move(int from, int to) {
        this.from = from;
        this.to = to;
    }

    /**
     * Returns the origin of this move.
     * @return the origin of this move.
     */
    public int from() {
        return from;
    }

    /**
     * Returns the destination of this move.
     * @return the destination of this move
     */
    public int to() {
        return to;
    }
}
```

1.2 Board class

```
import java.util.ArrayList;

public class Board {
    private char[] board;
    private int turn;
    private boolean isWhite;
    private boolean isGameDone;
    private static int finishedGames = 0;
    private static final char EMPTY = ' ';

    /**
     * Creates a new Alquerque board in the starting state:
     * each player has twelve pieces in their original position, and it is white's turn.
     */
    public Board() {
```

```
16          turn = 1;
17          board = new char[26];
18          for (int i = 1; i < 26; i++) {
19              if (i < 13)
20                  board[i] = 'B';
21              else if (i == 13)
22                  board[i] = EMPTY;
23              else
24                  board[i] = 'W';
25          }
26          isWhite = (turn % 2 == 1);
27          isGameDone = false;
28      }
29
30      /**
31       * Returns the positions of all black pieces on the board.
32       *
33       * @return the positions of all black pieces on the board.
34       */
35      public int[] black() {
36          ArrayList<Integer> blackPieces = new ArrayList<Integer>();
37          for (int i = 1; i <= 25; i++)
38              if (this.board[i] == 'B')
39                  blackPieces.add(i);
40          int[] black = new int[blackPieces.size()];
41          for (int i = 0; i < blackPieces.size(); i++)
42              black[i] = blackPieces.get(i);
43          return black;
44      }
45
46      /**
47       * Returns the positions of all white pieces on the board.
48       * @return the positions of all white pieces on the board.
49       */
50      public int[] white() {
51          ArrayList<Integer> whitePieces = new ArrayList<Integer>();
52          for (int i = 1; i <= 25; i++)
53              if (this.board[i] == 'W')
54                  whitePieces.add(i);
55          int[] white = new int[whitePieces.size()];
56          for (int i = 0; i < whitePieces.size(); i++)
57              white[i] = whitePieces.get(i);
58          return white;
59      }
60
61      /**
62       * Moves a piece and updates the board correspondingly.
63       * Precondition: move must be a legal between 1 and 25
64       * @param move the move to simulate.
65       */
66      public void move(Move move) {
67          board[move.to()] = board[move.from()];
68          board[move.from()] = EMPTY;
69          if (isTakeMove(move))    //if the move is a take, the taken piece ↩
                is removed
70              board[(move.to() + move.from()) / 2] = EMPTY; //calculates ↩
                    average position value and removes piece
71          // Updates who's turn it is
72          this.turn++;
73          isWhite = (turn % 2 == 1);
74          // updates finishedGames after eachmove
75         if (isGameOver() && !isGameDone) {
```

```
76              finishedGames++;
77              isGameDone = true;
78          }
79      }
80
81      /**
82       * Checks whether a move is legal.
83       * Precondition: move must be an int from 1 through 25
84       * @param move move input to evaluate.
85       */
86      public boolean isLegal(Move move) {
87          if (board[move.to()] != EMPTY)  // Checks whether the player tries to move from an empty cell
88              return false;
89          else if ((isWhite && board[move.from()] != 'W') || (!isWhite && board[move.from()] != 'B'))
90              // Checks if the player tries to move the opponents piece
91              return false;
92          else if (fileDiff(move) > 2)
93              // Checks if the player tries to move to a column that is too far away, which prevents moves rolling over from one row to the next
94              return false;
95          else if (!isTakeMove(move)){ // Logic for regular moves
96              if ((isWhite && (pieceDiff(move) < -6 || pieceDiff(move) > -4)) ||
97                      (!isWhite && (pieceDiff(move) < 4 || pieceDiff(move) > 6)))
98                  // Checks if direction is correct and if it is within the range of allowed cells to move to
99                  return false;
100             else if (move.from() % 2 == 0 && move.to() % 2 == 0)
101                 // Check for moves on even cells (To confirm that it follows the lines on the board)
102                 return false;
103         } else if (isTakeMove(move)) { // Logic for moves that take another piece
104             if (Math.abs(pieceDiff(move)) != 2 && Math.abs(pieceDiff(move)) != 8 &&
105                     Math.abs(pieceDiff(move)) != 10 && Math.abs(pieceDiff(move)) != 12)
106                 // Checks if the move is to the specified allowed cells for a take move
107                 return false;
108             else if (move.from() % 2 == 0 && Math.abs(pieceDiff(move)) != 10 && Math.abs(pieceDiff(move)) != 2)
109                 // Checks for moves on even cells (to confirm it follows the lines on the board)
110                 return false;
111         }
112         return true;
113     }
114
115     /**
116      * Returns an array of all legal moves for this board
117      * @return an array of all legal moves for this board
118      */
119     public Move[] legalMoves() {
120         ArrayList<Move> legalList = new ArrayList<Move>();
121         for (int i = 1; i < board.length; i++)
122             if (board[i] != EMPTY)
123                 for (int j = 1; j < board.length; j++)
```

```
124                     if (isLegal(new Move(i,j)))
125                         legalList.add(new Move(i,j));
126         Move[] legalMoves = new Move[legalList.size()];
127         for (int i = 0; i < legalList.size(); i++)
128             legalMoves[i] = legalList.get(i);
129         return legalMoves;
130     }
131 
132     /**
133      * Returns if the game is over
134      * @return if the game is over
135      */
136     public boolean isGameOver() {
137         return (white().length == 0 || black().length == 0 || legalMoves()↩
                .length == 0);
138     }
139 
140 
141     /**
142      * Returns how many objects of type Board that represents games, that ↩
              are finished games.
143      * @return how many objects of type Board that represents games, that ↩
              are finished games.
144      */
145     public static int finishedGames() {
146         return finishedGames;
147     }
148 
149     /**
150      * Returns a copy of this board
151      * @return a copy of this board
152      */
153     public Board copy() {
154         Board newBoard = new Board();
155         for (int i = 0; i < this.board.length; i++)
156             newBoard.board[i] = this.board[i];
157         newBoard.turn = this.turn;
158         newBoard.isWhite = this.isWhite;
159         newBoard.isGameDone = this.isGameDone;
160         return newBoard;
161     }
162 
163     /**
164      * Checks whether this Board is equal to other Object
165      * @param other Object to check against this board
166      * @return whether this Board is equal to other Object
167      */
168     public boolean equals(Object other){
169         if (other == null) return false;
170         else if (this == other) return true;
171         else if (!(other instanceof Board)) return false;
172         Board otherBoard = (Board) other;
173         int i = 0;
174         while(i < this.board.length && this.board[i] == otherBoard.board[i↩
                ])
175             i++;
176         return (i == this.board.length && this.turn == otherBoard.turn && ↩
                this.isGameDone == otherBoard.isGameDone);
177     }
178 
179 
180     /**
```

```
181         * Returns a hashCode compised of this boards attributes
182         * @return a hashCode comprised of this boards attributes
183         */
184        public int hashCode() {
185            return (this.board.hashCode() + this.turn*31);
186        }
187
188        /*
189         * Auxillerary methods to check how far there are between the columns in the move
190         */
191        private int fileDiff(Move move){
192            return Math.abs(((move.from() - 1) % 5 + 1) - ((move.to() - 1) % 5 + 1));
193        }
194
195        /*
196         * Auxillerary method to check how far there is between two pieces
197         */
198        private int pieceDiff(Move move) {
199            return (move.to() - move.from());
200        }
201
202        /*
203         * checks whether the move is a take move
204         */
205        private boolean isTakeMove(Move move) {
206            return ((Math.abs(pieceDiff(move)) > 6 || Math.abs(pieceDiff(move)) < 4) &&
207                    ((isWhite && board[(move.to() + move.from()) / 2] == 'B') || //checks if opponent piece is taken
208                            (!isWhite && board[(move.to() + move.from()) / 2] == 'W'))); //checks if opponent piece is taken
209        }
210    }
```

1.3 MainTest

This was just made for testing purposes

```
1   import java.util.Scanner;
2   public class MainTest {
3       public static Board myBoard = new Board();
4       public static Board yourBoard = new Board();
5       public static Scanner reader = new Scanner(System.in);
6       public static final char EMPTY = ' ';
7
8       public static void main(String[] args) {
9
10
11
12          // test to see if legal moves prints the correct moves
13          /*
14          printBoard(myBoard);
15          int from = 0;
16          int to = 0;
```

```
17          boolean isWhite = true;
18          do {
19              do {
20                  System.out.println("It's " + ((isWhite) ? "white to move, ↩
                        these are the legal moves:" : "black to move these are↩
                         the legal moves:"));
21                  for (int i = 0; i < myBoard.legalMoves().length; i++)
22                      System.out.println("From: " + myBoard.legalMoves()[i].↩
                            from() + ", " + "to: " + myBoard.legalMoves()[i].↩
                            to());
23                  System.out.println("which peice do you want to move: ");
24                  from = reader.nextInt();
25                  System.out.println("where do you want to move that piece")↩
                        ;
26                  to = reader.nextInt();
27              } while (!myBoard.isLegal(new Move(from, to)));
28              myBoard.move(new Move(from, to));
29              isWhite = !isWhite;
30              printBoard(myBoard);
31          } while (!myBoard.isGameOver());
32           */
33
34
35
36
37          /*
38          // test of white() and black()
39          System.out.println("Starting positon of the board:");
40          printBoard(myBoard);
41          System.out.println("Position of white's pieces:");
42          for (int i = 0; i < myBoard.white().length; i++)
43              System.out.print(myBoard.white()[i] + ", ");
44          System.out.println("\nPosition of black's pieces:");
45          for (int i = 0; i < myBoard.black().length; i++)
46              System.out.print(myBoard.black()[i] + ", ");
47
48          boolean isWhite = true;
49          do {
50              myBoard.move(Minimax.nextMove(myBoard,5,isWhite));
51              isWhite = !isWhite;
52          } while (!myBoard.isGameOver());
53
54          System.out.println("\n");
55          System.out.println("Other position of board: ");
56          printBoard(myBoard);
57          System.out.println("Position of white's pieces:");
58          for (int i = 0; i < myBoard.white().length; i++)
59              System.out.print(myBoard.white()[i] + ", ");
60          System.out.println("\nPosition of black's pieces:");
61          for (int i = 0; i < myBoard.black().length; i++)
62              System.out.print(myBoard.black()[i] + ", ");
63           */
64
65          /*
66          // test to see if the finishedGames method works and if it is ↩
                incremented by MiniMax
67          int white = 0;
68          int black = 0;
69          int draw = 0;
70          boolean end = false;
71          boolean isWhite = true;
72          do {
```

```
73              System.out.println("Games Played: " + Board.finishedGames());
74              myBoard = new Board();
75              printBoard(myBoard);
76              do {
77                  Move nextMove = Minimax.nextMove(myBoard, 5, isWhite);
78                  myBoard.move(nextMove);
79                  printBoard(myBoard);
80                  isWhite = !isWhite;
81              } while (!myBoard.isGameOver());
82              if (myBoard.white().length == 0) {
83                  System.out.println("White won!");
84                  black++;
85              } else if (myBoard.black().length == 0) {
86                  System.out.println("White won!");
87                  white++;
88              } else {
89                  System.out.println("Draw.");
90                  draw++;
91              }
92              System.out.println("Stats:");
93              System.out.println("White's wins: " + white);
94              System.out.println("Black's wins: " + black);
95              System.out.println("Draws: " + draw);
96
97              System.out.println();
98              System.out.print("Do you want to continue with another game? (↩
                    y/n): ");
99              end = ((reader.nextLine().toLowerCase().charAt(0) == 'n') ? ↩
                    true : false);
100         } while (!end);
101         */
102
103         // testing hashCode and copy
104         /*
105         System.out.println(myBoard.hashCode());
106         Move[] m = myBoard.legalMoves();
107         for (int i = 0; i < m.length; i++) {
108             System.out.println("From: " + m[i].from() + ", to: " + m[i].to↩
                    ());
109         }
110         nBoard = myBoard.copy();
111         System.out.println(nBoard.hashCode());
112         */
113     }
114
115     // these methods were previously developed for our own Alquerque ↩
            client.
116     /**
117      * Creates a representation of the game board with the pieces ↩
            correctly placed
118      * in the form of a two dimensional array.
119      * Precondition: Relies on method black() and white() to return valid ↩
            positions numbered from 1-25
120      * @return a two dimensional array 5 x 5 with the game pieces placed ↩
            correctly
121      */
122     private static char[][] boardWithPieces(Board myBoard) {
123         char[][] boardArr = new char[6][5]; //A-E & (no 0) 1-5
124         for (int j = 1; j < boardArr.length; j++)
125             for (int i = 0; i < boardArr[j].length; i++)
126                 boardArr[j][i] = EMPTY; // Fills board with empty spaces
127         for (int i = 0; i < myBoard.black().length; i++)
```

```
128             boardArr[((myBoard.black()[i] - 1) / 5) + 1][((myBoard.black()↩
                    [i] - 1) % 5)] = 'B'; // Places black pieces
129         for (int i = 0; i < myBoard.white().length; i++)
130             boardArr[((myBoard.white()[i] - 1) / 5) + 1][((myBoard.white()↩
                    [i] - 1) % 5)] = 'W'; // Places white pieces
131         return boardArr;
132     }
133 
134     /**
135      * prints a representation of the board to the terminal
136      */
137     private static void printBoard(Board myBoard) {
138         System.out.println(); // new line
139         int i = 0, j = 1;
140         System.out.println("   A   B   C   D   E"); //upper-coordinate-↩
                line (A-E)
141         char[][] boardWithPieces = boardWithPieces(myBoard);
142         while (j < 6) {
143             System.out.print(j + " "); //left-hand coordinate (1-5)
144             while (i < 5) {
145                 System.out.print("[" + boardWithPieces[j][i] + "]");
146                 if (i < 4)
147                     System.out.print("-");
148                 i++;
149             }
150             System.out.print(" " + (j)); //right-hand coordinate (1-5)
151             System.out.println("");
152             i = 0;
153             if (j % 2 == 1 && j < 5)
154                 System.out.println("   | \\ | / | \\ | / |");
155             else if (j % 2 == 0)
156                 System.out.println("   | / | \\ | / | \\ |");
157             j++;
158         }
159         System.out.println("   A   B   C   D   E"); //bottom-coordinate-↩
                line (A-E)
160         System.out.println(""); // new line
161     }
162 }
```