Alquerque

Danny Nicolai Larsen, Mikkel Brix Nielsen & Steffen Bach

December 9, 2021

Contents

CONTENTS

PUT MAIN TEXT HERE

1 Appendix

1.1 Move class

```
1  public class Move {
2      private int from;
3      private int to;
4      /**
5       * Creates a new move with given origin and destination.
6       * @param from the place to move the piece from.
7       * @param to the place to move the peiece to.
8       */
9      public Move(int from, int to) {
10         this.from = from;
11         this.to = to;
12     }
13 
14     /**
15      * Returns the origin of this move.
16      * @return the origin of this move.
17      */
18     public int from() {
19         return from;
20     }
21 
22     /**
23      * Returns the destination of this move.
24      * @return the destination of this move
25      */
26     public int to() {
27         return to;
28     }
29 }
```

1.2 Board class

```
1  import java.util.ArrayList;
2 
3  public class Board {
4      private char[] board;
5      private int turn;
6      private boolean isWhite;
7      private boolean isGameDone;
8      private static int finishedGames = 0;
9      private static final char EMPTY = ' ';
10 
11     /**
12      * Creates a new Alquerque board in the starting state:
13      * each player has twelve pieces in their original position, and it is↩
            white's turn.
14      */
15     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      * Returns how many objects of type Board that represents games, that ↩
             are finished games.
142      * @return how many objects of type Board that represents games, that ↩
             are finished games.
143      */
144     public static int finishedGames() {
145         return finishedGames;
146     }
147 
148     /**
149      * Returns a copy of this board
150      * @return a copy of this board
151      */
152     public Board copy() {
153         Board newBoard = new Board();
154         for (int i = 0; i < this.board.length; i++)
155             newBoard.board[i] = this.board[i];
156         newBoard.turn = this.turn;
157         newBoard.isWhite = this.isWhite;
158         newBoard.isGameDone = this.isGameDone;
159         return newBoard;
160     }
161 
162     /**
163      * Checks whether this Board is equal to other Object
164      * @param other Object to check against this board
165      * @return whether this Board is equal to other Object
166      */
167     public boolean equals(Object other){
168         if (other == null) return false;
169         else if (this == other) return true;
170         else if (!(other instanceof Board)) return false;
171         Board otherBoard = (Board) other;
172         int i = 0;
173         while(i < this.board.length && this.board[i] == otherBoard.board[i↩
                ])
174             i++;
175         return (i == this.board.length && this.turn == otherBoard.turn && ↩
                this.isGameDone == otherBoard.isGameDone);
176     }
177 
178     /**
179      * Returns a hashCode compised of this boards attributes
180      * @return a hashCode comprised of this boards attributes
```

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

1.3 MainTest

This was just made for testing purposes

```
1  import java.util.Locale;
2  import java.util.Scanner;
3  public class MainTest {
4      public static Board myBoard = new Board();
5      public static Board yourBoard = new Board();
6      public static Scanner reader = new Scanner(System.in);
7      public static final char EMPTY = ' ';
8  
9      public static void main(String[] args) {
10 
11         // test whether an instance of Move returns the correct to and from values
12         Move m1 = new Move(19,13);
13         System.out.println("Move 1 excpected: 19, 13 - Got: "  + m1.from() + ", " + m1.to());
14         Move m2 = new Move(2,25);
15         System.out.println("Move 2 excpected: 2, 25 - Got: " + m2.from() + ", " + m2.to());
```

```
16          Move m3 = new Move(23,46);
17          System.out.println("Move 3 excpected: 23, 46 - Got: " + m3.from()
                + ", " + m3.to());
18          Move m4 = new Move(-12,40);
19          System.out.println("Move 4 excpected: -12, 40 - Got: " + m4.from()
                 + ", " + m4.to());
20          Move m5 = new Move(-91,-108);
21          System.out.println("Move 4 excpected: -91, -108 - Got: " + m5.from
                () + ", " + m5.to());
22
23          // test to see if legal moves prints the correct moves
24
25          printBoard(myBoard);
26          int from = 0;
27          int to = 0;
28          boolean isWhite = true;
29          do {
30              do {
31                  System.out.println("It's " + ((isWhite) ? "white to move,
                        these are the legal moves:" : "black to move these are
                         the legal moves:"));
32                  for (int i = 0; i < myBoard.legalMoves().length; i++)
33                      System.out.println("From: " + myBoard.legalMoves()[i].
                            from() + ", " + "to: " + myBoard.legalMoves()[i].
                            to());
34                  System.out.println("which peice do you want to move: ");
35                  from = reader.nextInt();
36                  System.out.println("where do you want to move that piece")
                        ;
37                  to = reader.nextInt();
38              } while (!myBoard.isLegal(new Move(from, to)));
39              myBoard.move(new Move(from, to));
40              isWhite = !isWhite;
41              printBoard(myBoard);
42          } while (!myBoard.isGameOver());
43
44
45          // test of white() and black()
46          System.out.println("Starting positon of the board:");
47          printBoard(myBoard);
48          System.out.println("Position of white's pieces:");
49          for (int i = 0; i < myBoard.white().length; i++)
50              System.out.print(myBoard.white()[i] + ", ");
51          System.out.println("\nPosition of black's pieces:");
52          for (int i = 0; i < myBoard.black().length; i++)
53              System.out.print(myBoard.black()[i] + ", ");
54
55          boolean isWhite = true;
56          do {
57              myBoard.move(Minimax.nextMove(myBoard,5,isWhite));
58              isWhite = !isWhite;
59          } while (!myBoard.isGameOver());
60
61          System.out.println("\n");
62          System.out.println("Other position of board: ");
63          printBoard(myBoard);
64          System.out.println("Position of white's pieces:");
65          for (int i = 0; i < myBoard.white().length; i++)
66              System.out.print(myBoard.white()[i] + ", ");
67          System.out.println("\nPosition of black's pieces:");
68          for (int i = 0; i < myBoard.black().length; i++)
69              System.out.print(myBoard.black()[i] + ", ");
```

```
70
71
72
73          // test to see if the finishedGames method works and if it is ↩
                incremented by MiniMax
74          int white = 0;
75          int black = 0;
76          int draw = 0;
77          boolean end = false;
78          boolean isWhite = true;
79          do {
80              System.out.println("Games Played: " + Board.finishedGames());
81              myBoard = new Board();
82              printBoard(myBoard);
83              do {
84                  Move nextMove = Minimax.nextMove(myBoard, 5, isWhite);
85                  myBoard.move(nextMove);
86                  printBoard(myBoard);
87                  isWhite = !isWhite;
88              } while (!myBoard.isGameOver());
89              if (myBoard.white().length == 0) {
90                  System.out.println("White won!");
91                  black++;
92              } else if (myBoard.black().length == 0) {
93                  System.out.println("White won!");
94                  white++;
95              } else {
96                  System.out.println("Draw.");
97                  draw++;
98              }
99              System.out.println("Stats:");
100             System.out.println("White's wins: " + white);
101             System.out.println("Black's wins: " + black);
102             System.out.println("Draws: " + draw);
103
104             System.out.println();
105             System.out.print("Do you want to continue with another game? (↩
                    y/n): ");
106             end = ((reader.nextLine().toLowerCase().charAt(0) == 'n') ? ↩
                    true : false);
107         } while (!end);
108
109
110         // testing hashCode and copy
111         System.out.println(myBoard.hashCode());
112         Move[] m = myBoard.legalMoves();
113         for (int i = 0; i < m.length; i++) {
114             System.out.println("From: " + m[i].from() + ", to: " + m[i].to↩
                    ());
115         }
116         Board nBoard;
117         nBoard = myBoard.copy();
118         System.out.println(nBoard.hashCode());
119
120     }
121
122     // these methods were previously developed for our own Alquerque ↩
            client.
123     /**
124      * Creates a representation of the game board with the pieces ↩
              correctly placed
125      * in the form of a two dimensional array.
```

```
126         * Precondition: Relies on method black() and white() to return valid positions numbered from 1-25
127         * @return a two dimensional array 5 x 5 with the game pieces placed correctly
128         */
129        private static char[][] boardWithPieces(Board myBoard) {
130            char[][] boardArr = new char[6][5]; //A-E & (no 0) 1-5
131            for (int j = 1; j < boardArr.length; j++)
132                for (int i = 0; i < boardArr[j].length; i++)
133                    boardArr[j][i] = EMPTY; // Fills board with empty spaces
134            for (int i = 0; i < myBoard.black().length; i++)
135                boardArr[((myBoard.black()[i] - 1) / 5) + 1][((myBoard.black()[i] - 1) % 5)] = 'B'; // Places black pieces
136            for (int i = 0; i < myBoard.white().length; i++)
137                boardArr[((myBoard.white()[i] - 1) / 5) + 1][((myBoard.white()[i] - 1) % 5)] = 'W'; // Places white pieces
138            return boardArr;
139        }
140
141        /**
142         * prints a representation of the board to the terminal
143         */
144        private static void printBoard(Board myBoard) {
145            System.out.println(); // new line
146            int i = 0, j = 1;
147            System.out.println("   A   B   C   D   E"); //upper-coordinate-line (A-E)
148            char[][] boardWithPieces = boardWithPieces(myBoard);
149            while (j < 6) {
150                System.out.print(j + " "); //left-hand coordinate (1-5)
151                while (i < 5) {
152                    System.out.print("[" + boardWithPieces[j][i] + "]");
153                    if (i < 4)
154                        System.out.print("-");
155                    i++;
156                }
157                System.out.print(" " + (j)); //right-hand coordinate (1-5)
158                System.out.println("");
159                i = 0;
160                if (j % 2 == 1 && j < 5)
161                    System.out.println("   | \\ | / | \\ | / |");
162                else if (j % 2 == 0)
163                    System.out.println("   | / | \\ | / | \\ |");
164                j++;
165            }
166            System.out.println("   A   B   C   D   E"); //bottom-coordinate-line (A-E)
167            System.out.println(""); // new line
168        }
169    }
```