```java
import java.util.LinkedList;




/**
 * This class implements my own Othello AI.
 * @author Arvind Vijayakumar
 */

public class MyPlayerAlt implements OthelloPlayer {

    private OthelloSide side;
    private OthelloSide opponentSide;

    //AI's copy/account of board
    private OthelloBoard board = new OthelloBoard();


    public void init(OthelloSide side) {
        this.side = side;
        if (side == OthelloSide.BLACK)
            opponentSide = OthelloSide.WHITE;
        else
            opponentSide = OthelloSide.BLACK;
    }

    int turn = 0;
    private final int MAX_DEPTH = 6;
    public Move doMove(Move opponentsMove, long millisLeft) {
        //Reflects opponents move on AI copy of board
        //System.out.println("Start of doMove.");
        System.out.println(" \n\n\n\n||||>> Turn: " + turn + " <<||||\n\n\n\n");
        Move move = new Move();
        if(opponentsMove != null) {
            board.move(opponentsMove, opponentSide);
        }
        if(getMoveList(board, side).size() == 0) {
            return null;
        }
        //Move chosenMove = null;
        //if(chosenMove != null) { System.out.println("chosenMove" + chosenMove); }
        //else{ System.out.println("chosenMove == null"); }
        Pair<Double, Move> store = minimax(board, side, 0, MAX_DEPTH, move, -Double.MAX_VALUE,
  Double.MAX_VALUE);
        Move chosenMove = store.move;
        //int maxDepth = 8;
        //alt(board, side,  chosenMove, -Float.MAX_VALUE, Float.MAX_VALUE, 0, maxDepth);
        turn++;
        if(board.checkMove(chosenMove, side)) {
            System.out.println("ChosenMove: " + chosenMove.toString());
            //System.out.println("FINAL:theScore: " + store.score);
            System.out.println("Other Current Score: " + evaluator(board, side.opposite()));
            System.out.println("Computer Current Score: " + evaluator(board, side));
            board.move(chosenMove, side);
            return chosenMove;
        } else {
            return null;
        }
    }
    /*OthelloMove bestFound = new OthelloMove();
     int maxDepth = 8;
     minimax(board, bestFound, -Float.MAX_VALUE, Float.MAX_VALUE, maxDepth);
    //Wait for Thread to finish
     board.makeMove(bestFound);*/

    public class Pair<Double, Move> {
          public double score;
```

```
68          public Move move;
69          public Pair(double x, Move y) {
70            this.score = x;
71            this.move = y;
72          }
73        }
74
75    /**
76     * Main minimax recursive method.
77     */
78    private Pair<Double, Move> minimax(OthelloBoard board, OthelloSide side, int depth, int max_depth,
79                                       Move m, double alpha, double beta) {
80        double bestScore;
81        int turncount = turn;
82        Move bestMove;
83        //System.out.println("Start of minimax. Depth: " + depth + " Side: " + side);
84      /*  int state = board.getState();
85        if (state == OthelloBoard.DRAW)
86            return new Pair<Double, Move>(0, m);
87        if ((state == OthelloBoard.BLACK_WINS) && (side == OthelloSide.BLACK))
88            return new Pair<Double, Move>(Double.POSITIVE_INFINITY, m);
89        if ((state == OthelloBoard.WHITE_WINS) && (side == OthelloSide.WHITE))
90            return new Pair<Double, Move>(Double.POSITIVE_INFINITY, m);
91        if ((state == OthelloBoard.BLACK_WINS) && (side == OthelloSide.WHITE))
92            return new Pair<Double, Move>(Double.NEGATIVE_INFINITY, m);
93        if ((state == OthelloBoard.WHITE_WINS) && (side == OthelloSide.BLACK))
94            return new Pair<Double, Move>(Double.NEGATIVE_INFINITY, m);*/
95        if(board.isDone()) {
96            double endgame = (double) board.rawScore(side);
97            if(side == this.side) {
98                return new Pair<Double, Move>(endgame, m);
99            } else {
100               return new Pair<Double, Move>(-endgame, m);
101           }
102       }
103       if(depth == max_depth) {
104           double mdScore = evaluator(board, side);
105           return new Pair<Double, Move>(mdScore, m);
106       } else {
107           LinkedList<Move> moveList = getMoveList(board, side);
108           if(depth == 0) {
109               LinkedList<Move> corners = new LinkedList<Move>();
110               for(Move mv : moveList) {
111                   if(board.isCorner(mv)) {
112                       corners.add(mv);
113                   }
114               }
115               if(corners.size() != 0) {
116                   Move bcorner = null;
117                   double best = -Double.MAX_VALUE;
118                   for(Move ml : corners) {
119                       double temp = evalMove(board, side, ml);
120                       if(temp > best) {
121                           best = temp;
122                           bcorner = ml;
123                       }
124                   }
125                   return new Pair<Double, Move>(best, bcorner);
126               }
127           }
128           //System.out.println(moveList.toString());
129           bestScore = -Double.MAX_VALUE;
130           bestMove = new Move(1,1);
131           if(moveList.size() == 0) {
132               double mdScore = evaluator(board, side);
133               return new Pair<Double, Move>(mdScore, m);
134           } else {
135               for(int i = 0; i < moveList.size(); i++) {
```

```
136                     OthelloBoard tempBoard = board.copy();
137                     Move move = moveList.get(i);
138                     tempBoard.move(move, side);
139                     alpha = -(minimax(tempBoard, side.opposite(), depth + 1, max_depth, move, -beta, -
    alpha)).score;
140                     //System.out.println("Side: " + side);
141                     //System.out.println("alpha (before IF): " + alpha);
142                     //System.out.println("bestScore (before IF): " + bestScore);
143                     if(beta <= alpha) {
144                         return new Pair<Double, Move>(alpha, move);
145                     }
146                     if(alpha > bestScore ) {
147                         bestScore = alpha;
148                         bestMove = move;
149                         //bestMove.copy(move);
150                         //System.out.println("theScore(IF): " + alpha);
151                         //System.out.println("bestScore(IF): " + bestScore);
152                     }
153                 }
154                 return new Pair<Double, Move>(bestScore, bestMove);
155             }
156         }
157     }
158 
159     private double evaluator(OthelloBoard board, OthelloSide side) {
160         double pieceEval;
161         double stability;
162         double mobility;
163         double pieceCount;
164         double corner;
165         double nextCorner;
166         double score = 0;
167         //double n = 1;
168         //pieceEval
169         double p = pieceCount(board, side);
170         pieceCount = 10*p;
171         //stability
172         double s = stability(board, side);
173         stability = 74.396*s;
174         //mobility
175         double m = mobility(board, side);
176         //mobility = 30*m*Math.pow(0.86, turnCount);
177         mobility = 78.922*m;
178         //pieceCount
179         double pE = pieceEval(board, side);
180         //pieceEval = 60*pE*Math.pow(0.96, turnCount);
181         pieceEval = 10*pE;
182         //corner
183         double c = cornerOccupancy(board, side);
184         corner = 801.724*c;
185         //nextCorner
186         double nC = nextToCorner(board, side);
187         nextCorner = 382.026*nC;
188         //score
189         score = pieceCount + stability + mobility + pieceEval + corner + nextCorner;
190         return score;
191 
192     }
193 
194     private double cornerOccupancy(OthelloBoard board, OthelloSide side) {
195         int bTiles = 0;
196         int wTiles = 0;
197         double score = 0;
198         if(board.get(side, 0, 0)) {bTiles++;}
199         if(board.get(side, 0, 7)) {bTiles++;}
200         if(board.get(side, 7, 0)) {bTiles++;}
201         if(board.get(side, 7, 7)) {bTiles++;}
202         if(board.get(side.opposite(), 0, 0)) {wTiles++;}
```

```
203         if(board.get(side.opposite(), 0, 7)) {wTiles++;}
204         if(board.get(side.opposite(), 7, 0)) {wTiles++;}
205         if(board.get(side.opposite(), 7, 7)) {wTiles++;}
206         score = 25*(bTiles - wTiles);
207         return score;
208     }
209 
210     private double nextToCorner(OthelloBoard board, OthelloSide side) {
211         int bTiles = 0;
212         int wTiles = 0;
213         double score = 0;
214         //Top-Left Corner Area
215         if(board.get(side, 0, 1)) {bTiles++;}
216         else if(board.get(side.opposite(), 0, 1)) {wTiles++;}
217         if(board.get(side, 1, 0)) {bTiles++;}
218         else if(board.get(side.opposite(), 1, 0)) {wTiles++;}
219         if(board.get(side, 1, 1)) {bTiles++;}
220         else if(board.get(side.opposite(), 1, 1)) {wTiles++;}
221         //Top-Right Corner Area
222         if(board.get(side, 6, 0)) {bTiles++;}
223         else if(board.get(side.opposite(), 6, 0)) {wTiles++;}
224         if(board.get(side, 6, 1)) {bTiles++;}
225         else if(board.get(side.opposite(), 6, 1)) {wTiles++;}
226         if(board.get(side, 7, 1)) {bTiles++;}
227         else if(board.get(side.opposite(), 7, 1)) {wTiles++;}
228         //Bottom-Left Corner Area
229         if(board.get(side, 0, 6)) {bTiles++;}
230         else if(board.get(side.opposite(), 0, 6)) {wTiles++;}
231         if(board.get(side, 1, 6)) {bTiles++;}
232         else if(board.get(side.opposite(), 1, 6)) {wTiles++;}
233         if(board.get(side, 1, 7)) {bTiles++;}
234         else if(board.get(side.opposite(), 1, 7)) {wTiles++;}
235         //Bottom-Right Corner Area
236         if(board.get(side, 7, 6)) {bTiles++;}
237         else if(board.get(side.opposite(), 7, 6)) {wTiles++;}
238         if(board.get(side, 6, 6)) {bTiles++;}
239         else if(board.get(side.opposite(), 6, 6)) {wTiles++;}
240         if(board.get(side, 6, 7)) {bTiles++;}
241         else if(board.get(side.opposite(), 6, 7)) {wTiles++;}
242         score = -12.5*(bTiles - wTiles);
243         return score;
244     }
245 
246     private double pieceEval(OthelloBoard board, OthelloSide side) {
247         int score = 0;
248         int[][] values = { {20, -3, 11, 8, 8, 11, -3, 20},
249                            {-3, -7, -4, 1, 1, -4, -7, -3},
250                            {11, -4, 2, 2, 2, 2, -4, 11},
251                            {8, 1, 2, -3, -3, 2, 1, 8},
252                            {8, 1, 2, -3, -3, 2, 1, 8},
253                            {11, -4, 2, 2, 2, 2, -4, 11},
254                            {-3, -7, -4, 1, 1, -4, -7, -3},
255                            {20, -3, 11, 8, 8, 11, -3, 20}};
256         for(int x = 0; x < 8; x++) {
257             for(int y = 0; y < 8; y++) {
258                 if(board.get(side, x, y)) {
259                     score += values[y][x];
260                 } else if(board.get(side.opposite(), x, y)) {
261                     score -=values[y][x];
262                 }
263             }
264         }
265         return score;
266     }
267 
268     private double stability(OthelloBoard board, OthelloSide side) {
269         boolean frontierBlack = false;
270         boolean frontierWhite = false;
```

```
271         int bPieces = 0;
272         int wPieces = 0;
273         int fScore = 0;
274         LinkedList<Move> occupiedBlackList = getOccupiedPlaces(board, side);
275         LinkedList<Move> occupiedWhiteList = getOccupiedPlaces(board, side.opposite());
276         int[] xC = {1, 1, 0, -1, -1, -1, 1, 0};
277         int[] yC = {0, 1, 1, 0, -1, 1, -1, -1};
278         for(int i = 0; i < occupiedBlackList.size(); i++) {
279             for(int k = 0; k < 8; k++) {
280                 int x1 = occupiedBlackList.get(i).getX();
281                 int y1 = occupiedBlackList.get(i).getY();
282                 int x = x1 + xC[k];
283                 int y = y1 + yC[k];
284                 if( x >= 0 && y >= 0 && !(board.isCorner(x1, y1))) {
285                     if(board.get(side, x1, y1) && !(board.occupied(x, y))) {
286                         frontierBlack = true;
287                     }
288                 }
289             }
290             if(frontierBlack) {bPieces++;}
291         }
292         for(int i = 0; i < occupiedWhiteList.size(); i++) {
293             for(int k = 0; k < 8; k++) {
294                 int x1 = occupiedWhiteList.get(i).getX();
295                 int y1 = occupiedWhiteList.get(i).getY();
296                 int x = x1 + xC[k];
297                 int y = y1 + yC[k];
298                 if( x >= 0 && y >= 0 &&!(board.isCorner(x1, y1))) {
299                     if(board.get(side.opposite(), x1, y1) && !(board.occupied(x, y))) {
300                         frontierWhite = true;
301                     }
302                 }
303             }
304             if(frontierWhite) {wPieces++;}
305         }
306         if(bPieces > wPieces) {
307             fScore = -100*bPieces/(bPieces + wPieces);
308         } else if(bPieces < wPieces) {
309             fScore = 100*wPieces/(bPieces + wPieces);
310         }
311         return fScore;
312     }
313 
314     private double mobility(OthelloBoard board, OthelloSide side) {
315         int m = 0;
316         int my = getMoveList(board, side).size();
317         int opp = getMoveList(board, side.opposite()).size();
318         if(my > opp) {
319             m = 100*my/(my + opp);
320         } else if(my < opp) {
321             m = -100*opp/(my + opp);
322         }
323         return m;
324     }
325 
326     private double pieceCount(OthelloBoard board, OthelloSide side) {
327         int p = 0;
328         int my = board.rawScore(side);
329         int opp = board.rawScore(side.opposite());
330         if(my > opp) {
331             p = 100*my/(my + opp);
332         } else if(my < opp) {
333             p = -100*opp/(my + opp);
334         }
335         return p;
336     }
337 
338     // Returns a list of possible moves.
```

```
339     private LinkedList<Move> getMoveList(OthelloBoard board, OthelloSide side) {
340         Move move;
341         LinkedList<Move> moveList = new LinkedList<Move>();
342         for (int i = 0; i < 8; i++)
343             for (int j = 0; j < 8; j++) {
344                 move = new Move(i,j);
345                 if (board.checkMove(move, side)) {
346                     moveList.add(move);
347                 }
348             }
349         return moveList;
350     }
351 
352     private LinkedList<Move> getOccupiedPlaces(OthelloBoard board, OthelloSide side) {
353         LinkedList<Move> occupiedPlaces = new LinkedList<Move>();
354         Move move;
355         for(int x = 0; x < 8; x++) {
356             for(int y = 0; y < 8; y++) {
357                 if(board.get(side, x, y)) {
358                     move = new Move(x,y);
359                     occupiedPlaces.add(move);
360                 }
361             }
362         }
363         return occupiedPlaces;
364     }
365 
366     public double evalMove(OthelloBoard board, OthelloSide side, Move m) {
367        OthelloBoard temp = board.copy();
368        temp.move(m, side);
369        double score = evaluator(temp, side);
370        return score;
371     }
372 }
```