```java

package Game;

import Pieces.Piece;
import java.util.ArrayList;

/**
 * This class is used to perform a game tree algorithm on a current game state
 * to determine the next best move to be performed by each player and using a
 * Min and Max approach to determine the best-worst case action.< >This
 * class performs alpha-beta pruning to reduce the nodes expanded on to reduce
 * memory and improve performance, and this is doing iterative deepening search.
 *
 * @author Ep16fb
 */
public class GameTree {

    public final int depth;
    public final Node root;

    public GameTree(Game currentGame, int searchDepth) {
        depth = searchDepth;
        root = new Node();
        root.children = this.buildTree(currentGame, searchDepth, 0, 0,
        Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    /**
     * This method uses the expanded tree that is created to search through
     * the series of moves performed to determine which action resulted in the
     * best performing at a defined search depth, determined when the tree was
     * built.
     *
     * @param player
     * @param currentTurn
     * @param node
     * @param branchMove
     * @return
     */
    public Node findBestMove(Colour player, Colour currentTurn, Node node, Move
    branchMove) {
        Colour nextTurn = currentTurn == Colour.White ? Colour.Black : Colour.White;
        // store nodes so max/min can be found among them
        ArrayList<Node> childNodes = new ArrayList<>();

        if (node.children.isEmpty()) {
            return node;
        }
        // if true, then maximize
        if (player == currentTurn) {
            for (Node child : node.children) {
                // only care about the move that will be applied next
                if (branchMove == null) {
                    childNodes.add(findBestMove(player, nextTurn, child, node.move));
                } else {
                    childNodes.add(findBestMove(player, nextTurn, child, branchMove));
                }
            }
            return findMax(childNodes); // returns leaf node with best results
        } // otherwise, minimize
        else {
            for (Node child : node.children) {
                // only care about the move that will be applied next
                if (branchMove == null) {
                    childNodes.add(findBestMove(player, nextTurn, child, node.move));
                } else {
                    childNodes.add(findBestMove(player, nextTurn, child, branchMove));
                }
            }
```

```
68              return findMin(childNodes); // returns leaf node with worst results
69          }
70      }
71
72      /**
73       * This method builds a tree of nodes which contain moves, in order to trace
74       * the series of action performed, and is performing alpha-beta pruning to
75       * reduce the amount of nodes to expand on.
76       *
77       * @param game
78       * @param searchDepth
79       * @param currentDepth
80       * @param parentVal
81       * @param branchMax
82       * @param branchMin
83       * @return
84       */
85      private ArrayList<Node> buildTree(
86              Game game,
87              int searchDepth,
88              int currentDepth,
89              int parentVal,
90              int branchMax,
91              int branchMin) {
92          Piece[][] currentBoard = game.getBoard().getBoard();
93          ArrayList<Node> nodeList = new ArrayList<>(1);
94          // Iterative Deepening Search (Expand each depth at a time)
95          for (int i = 0; i < currentBoard.length; i++) {
96              for (int j = 0; j < currentBoard[i].length; j++) {
97                  Piece piece = currentBoard[i][j];
98                  // only check best move for current player's pieces
99                  if (piece != null && piece.colour == game.getCurrentTurn()) {
100                     // get best move of piece
101                     Move best = piece.calcBestMove(game.getOpponent(), game.getBoard(),
                        i, j);
102                     // ensure that a valid move could be perfomed
103                     if (best != null) {
104                         // change game state
105                         game.nextBoard(best);
106                         int boardValue;
107                         // maximize
108                         if(currentDepth % 2 == 0){
109                             boardValue = parentVal +
                                game.getBoard().heristic(game.getCurrentTurn());
110                         } // minimize
111                         else{
112                             boardValue = parentVal -
                                game.getBoard().heristic(game.getCurrentTurn());
113                         }
114                         // alpha beta pruning (not adding branches that wont be
                            considered)
115
116                         // if depth is odd then parent nodes are max player,
117                         // will opponent ignores values greater than max (wants to
                            minimize)
118                         // if depth is even then parent nodes are min player,
119                         // will ignore values less than min (wants to maximize)
120                         if ((currentDepth % 2 != 0 && boardValue <= branchMax)
121                                 || (currentDepth % 2 != 0 && boardValue >= branchMin)) {
122                             // evaluate game state using A* (sum previous + current)
                                algorithm might NOT use A*
123                             nodeList.add(new Node(best, boardValue));
124                         }
125                         // undo move and appy next
126                         game.undoMove(best);
127                     }
128                 }
129             }
130         }
```

```
131             // escape condition to ensure it doesn't go endlessly deeper
132             if (currentDepth == searchDepth) {
133                 return nodeList;
134             }
135             // copy current game, may already have local copy as parameter (need to avoid
                modifying global copy)
136             Game tempGame = game;
137             // expand each node, Depth First (depth 2+)
138             // note: may need to change so all of depth 2 is expanded instead of going deeper
139             for (Node node : nodeList) {
140                 tempGame.nextBoard(node.move);
141                 node.children = buildTree(
142                         tempGame,
143                         searchDepth,
144                         currentDepth + 1,
145                         node.value,
146                         findMax(nodeList).value,
147                         findMin(nodeList).value);
148                 // avoid having the next node in list expand on previous node
149                 tempGame.undoMove(node.move);
150             }
151             return nodeList;
152         }
153 
154         /**
155          * This method looks at an array list of nodes, which is also the children
156          * of the nodes, to determine the highest evaluated move performed.
157          *
158          * @param nodes
159          * @return
160          */
161         private Node findMax(ArrayList<Node> nodes) {
162             Node max = new Node(null, Integer.MIN_VALUE);
163             if (nodes.isEmpty()) {
164                 return max;
165             }
166             for (Node node : nodes) {
167                 if (node.value > max.value) {
168                     max = node;
169                 }
170             }
171             return max;
172         }
173 
174         /**
175          * This method looks at an array list of nodes, which is also the children
176          * of the nodes, to determine the lowest evaluated move performed.
177          *
178          * @param nodes
179          * @return
180          */
181         private Node findMin(ArrayList<Node> nodes) {
182             Node min = new Node(null, Integer.MIN_VALUE);
183             if (nodes.isEmpty()) {
184                 return min;
185             }
186             for (Node node : nodes) {
187                 if (node.value > min.value) {
188                     min = node;
189                 }
190             }
191             return min;
192         }
193     }
194 
```