Comp 6721 - Artificial Intelligence - Project 2 project report

Federico O'Reilly Regueiro - 40012304

November 28th, 2016

1 Intro

The present project touches on the use of minimax, alpha-beta pruning and heuristics for game-playing. In the context of Reversi, the state-space grows rather quickly as a function of board-size. For every square, we can have either empty, white or black; yielding a state space of size 3^n, where n is the number of squares on a side of the board.

Typically, Reversi is played on an 8×8 board. This yields a whopping $3^{64} = 3,433,683,820,292,512,484,657,849,089,281$ possible board configurations[1]. Obviously, this is not a state space that can be searched with brute-force. For this project, we use a variation of minimax search optimization called negamax.

2 Negamax

The only difference between minimax and negamax is that negamax alternates node value signs for board-configuration evaluations depending on whether each configuration corresponds to a min board or a max board. There is no operational difference with the way that minimax works yet negamax facilitates a slightly cleaner implementation and streamlines the code somewhat.

```
56    public Position doMiniMax() {
57
58        if (root.children.isEmpty()) return null;
59        double bestVal = root.children.get(0).value;
60        Position bestMove = root.children.get(0).move;
61        for (int i = 1; i < root.children.size(); i++){
62            Node child = root.children.get(i);
63            if (-child.value > bestVal) {
64                bestVal = -child.value;
65                bestMove = child.move;
66            }
```

[1] Of course, a vast majority of them are not legal configurations.

```
67          }
68
69          return bestMove;
70      }
```

However, as we've seen in class, there is still a need to prune the search tree greatly in order to evaluate only pertinent board configurations. This is the reason $\alpha\beta$ pruning is so important, see line 84.

```
71      public void evaluateNegamax(Node n, double alpha, double beta) {
72
73          // evaluate all leaves first, traverse away from the root
74          // then use leaves' value to decide internal nodes
75
76          if(!n.children.isEmpty()) {
77              for (int i = 0; i < n.children.size(); i++){
78                  Node child = n.children.get(i);
79                  evaluateNegamax(child, -beta, -alpha);
80                  alpha = (-child.value > alpha)
81                          ? -child.value : alpha;
82                  n.value = alpha;
83                  if (alpha >= beta) {
84                      break;
85                  }
86              }
87          }
88          // evaluate leaves
89          else {
90              n.value = h.evaluateBoard(n.board, n.turn);
91          }
```

In order to use $\alpha\beta$ pruning in negamax, on each successive level k, becomes $\alpha_k = -\beta_{k-1}, \beta_k = -\alpha_{k-1}$ as can be seen in line 79.

3 Heuristic

As a heuristic, I have used a weighted sum of four different attributes of a board-configuration: corners, token stability, mobility and token parity. Tuning the sum is reliant on laboured trial and error and I could not find a satisfactory tuning for the evaluation of each different attribute.

3.a The simplest metric - token parity or piece count

The goal of reversi is to have more tokens on the board than the opponent. In this regard it is clear that token differential could be a good metric for measuring the goodness of any given configuration. However, Reversi is characterized by a rapidly changing board due to the constant token reversals, so piece differential can only be a metric as endgame draws near. For all practical means, it follows

that simple token differential is also more relevant the higher the ply of the minimax / negamax algorithm, since the leaves of our search tree are closer to the end of the game. Unfortunately, in our recursive implementation, the JVM's heap is depleted at around 6 or 7 ply, which leaves little room to play with weighting which considers a more distant lookahead.

```
52    private double pieceParity(Board board, int turn){
53
54        int parity = 0;
55        for (int i = 0; i < Board.BOARD_SIZE; ++i) {
56            for (int j = 0; j < Board.BOARD_SIZE; ++j) {
57                parity += board.getPlayerAtPos(new Position(j, i));
58            }
59        }
60        return (double)parity * GamePlay.opposite(turn);
61    }
```

3.b stability

On the opposite side of the spectrum from the volatile piece-differential is piece-stability. This aspect of the heuristic takes into account how many of the player's pieces cannot be flipped anymore (namely corners and adjacent tokens) and it penalizes tokens which can be immediately flipped. Piece stability is a better predictor of the game's outcome but more so towards the latter half of the game since at the onset there are rarely any stable pieces since the most common way for pieces to gain stability is by touching corners directly or indirectly.

The evaluation of token stability is the most involved one of the ones I chose so I decided to not account for certain configurations in which stability is achieved[2] in order to avoid consuming too much time during evaluation.

```
63    private double stability(Board board, int turn) {
64
65        List<Position> opponentMoves = board.getValidMoves(turn);
66        Set<Position> flips = new HashSet<>();
67        Set<Position> stable = new HashSet<>();
68        for (int i = 0; i < opponentMoves.size(); i++) {
69            List<Position> temp = board.getFlips(opponentMoves.get(i),
                  turn);
70            for (int j = 0; j < temp.size(); j++) {
71                flips.add(temp.get(j));
72            }
73        }
74
75        Position pos;
76        for (int i = 0; i < 4; i++) {
77            pos = theCorners[i];
```

[2]Namely groups of pieces which end up being totally surrounded by the opponent.

```
78              do {
79                  Position vScanPos = pos;
80                  while (vScanPos.isValid() &&
                        board.getPlayerAtPos(vScanPos) ==
                        GamePlay.opposite(turn)) {
81                      stable.add(vScanPos);
82                      vScanPos = edges[i][0].move(vScanPos);
83                  }
84                  pos = edges[i][1].move(pos);
85              } while (pos.isValid() && board.getPlayerAtPos(pos) ==
                    GamePlay.opposite(turn)) ;
86          }
87
88          int stability = -flips.size() + (3 * stable.size());
89          return (double)stability;
90      }
```

3.c corners

As we saw previously, corners are fundamental for token-stability and are therefore an important part of board-evaluation. This part of the heuristic not only focuses on the corners themselves but also penalizes configurations which will most likely result in the opponent gaining a corner, such as configurations in which the player has a token one square away from an open corner and an opponent's token on the other side.

```
92      private int corners(Board board, int turn) {
93
94          int corners = 0;
95          for (int i = 0; i < 4; i++) {
96              if (board.getPlayerAtPos(theCorners[i]) == turn) {
97                  corners++;
98              }
99              else if (board.getPlayerAtPos(theCorners[i]) == 0) {
100                 for (int j = 0; j < 3; j++){
101                     Position in1 = in[i][j].move(theCorners[i]);
102                     Position in2 = in[i][j].move(in1);
103                     if (board.getPlayerAtPos(in1) ==
                            GamePlay.opposite(turn)
104                             && board.getPlayerAtPos(in2) == turn){
105                         corners--;
106                     }
107                 }
108             }
109         }
110         return corners;
111     }
```

3.d mobility

It is possible to pass in the game, if there are no available moves. This is, of course, not really desirable as we loose the opportunity to populate the board. Additionally, the more spaces we have for placing a token, the more likely we are to have access to a good move in the near future. Therefore, this part of the heuristic tries to maximize relative player mobility and heavily penalizes configurations under which the player will have to pass.

```
113     private double mobility(Board board, int turn) {
114
115         int myMoves =
                board.getValidMoves(GamePlay.opposite(turn)).size();
116         if (myMoves == 0) return -1000; // we don't like passing
117         int yourMoves = board.getValidMoves(turn).size();
118         double mobility = ((double)myMoves-yourMoves) /
                ((double)myMoves+yourMoves);
119         return mobility * GamePlay.opposite(turn);
120     }
```

3.e weighting

After several attempts at fine-tuning the heuristic, I could not find any series of weights which completely satisfied me. The tentative weights found were these:

```
39      // Weighting
40      double par = ply * parity;// * ((64-remainingMoves)/64);
41      double mob = 2 * mobility;// * ((64-remainingMoves)/64);
42      double cor = 8 * corners / ply;
43      double stabl = 8 * stability;
44
45      return par + mob + cor + stabl;
```

Where a factor for emphasizing end-game has been commented-out since it did not seem to improve performance.

4 test runs and performance

The heuristic was tested against my teammate's heuristic and performed somewhat similarly. The scores for different plys can be seen in figures 1 and 2. Logs indicating board configurations, number of searched nodes and scores are attached in the electronic submission.

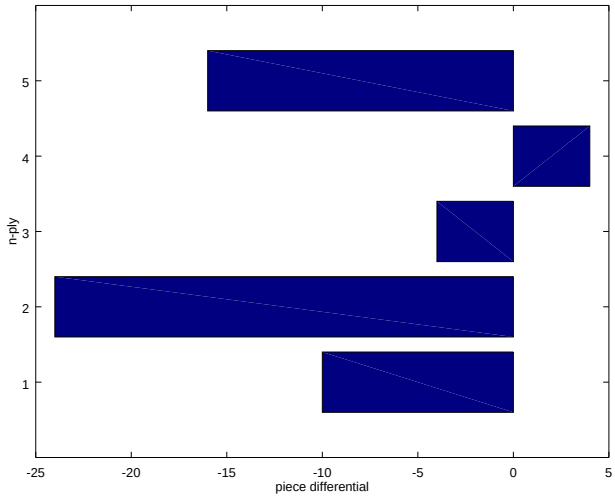

Figure 1: Teammate's PositionalHeuristic (black) vs FedeHeuristic(white) final scores as a function of ply-depth

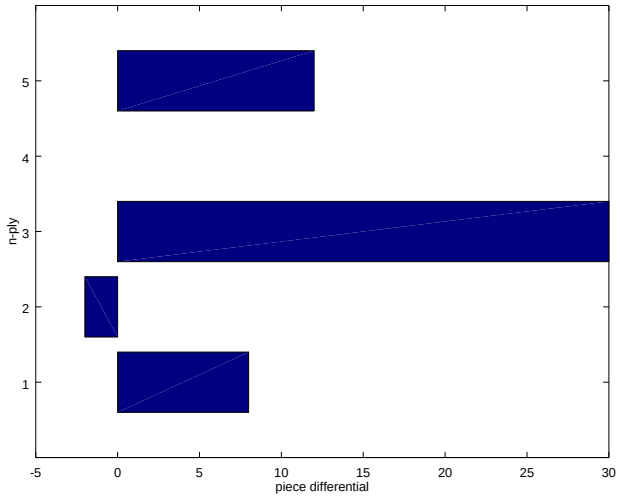

Figure 2: FedeHeuristic(black) vs teammate's PositionalHeuristic (white) final scores as a function of ply-depth