# Project 2: Checkers

Paul Beggs

GitHub Link

September 19, 2025

## 1 Evaluation Function

This basic evaluation function works by simply counting how many pieces each player has on the board, and then returning the difference between the current player's piece count and the opponent's piece count. Hence, it returns either a positive or negative integer, including zero if the counts are equal. The code snippet is provided in Listing 1.

```java
public class BasicEval implements ToIntFunction<Checkerboard> {
    @Override
    public int applyAsInt(Checkerboard value) {
        PlayerColor current = value.getCurrentPlayer();
        PlayerColor opponent = current.opponent();
        int currentCount = value.numPiecesOf(current);
        int opponentCount = value.numPiecesOf(opponent);
        return currentCount - opponentCount;
    }
}
```

Listing 1: Basic Evaluation Function

I've found that this evaluation function does not work well in practice. Since it does not give enough value to extraneous factors that can influence the outcome, such as piece positioning and potential future moves, it often leads to suboptimal play. For example, consider the board states from Figure 1: Here, black can move any piece, but decides to move the piece at h6 to g5, causing an immediate capture by red on the next turn. I'm not sure what may be causing this behavior, but I suspect it may be due to the fact that the evaluation function does not account for potential captures or threats, leading to poor decision-making in certain scenarios.

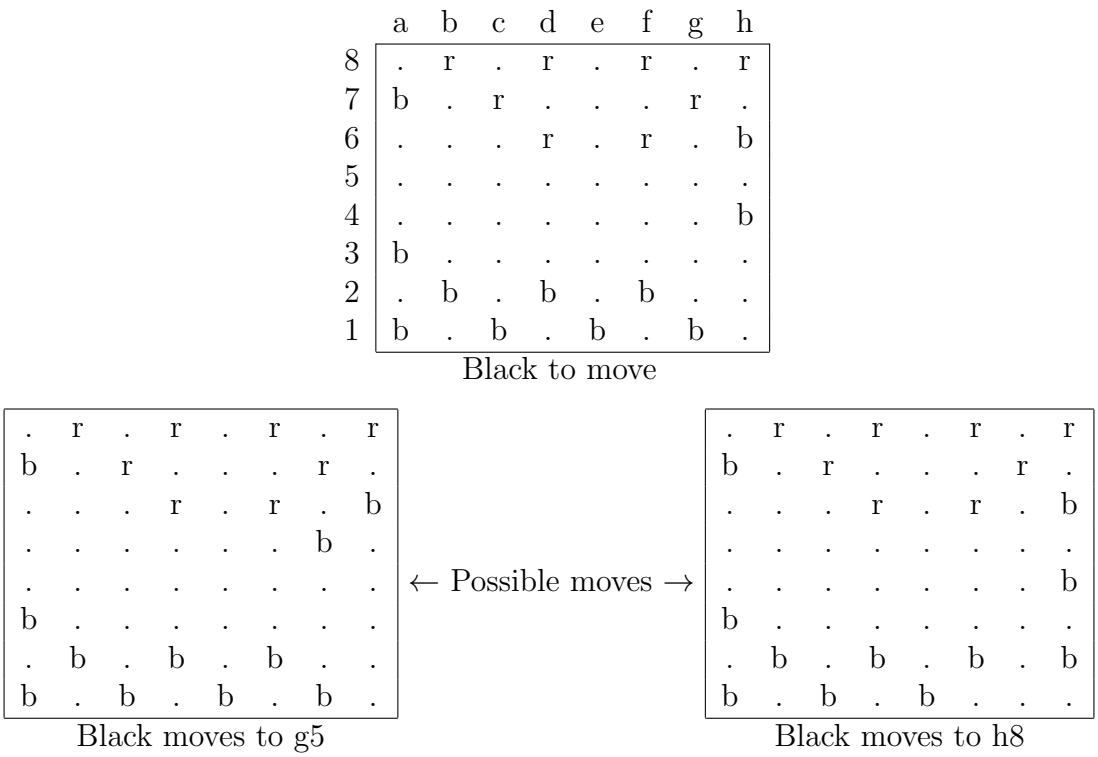


Figure 1: Board States

## 2 Searcher Implementation

Due to the lack of foresight in our evaluation function, our searcher implementation is also prone to errors. As they say, trash in, trash out. However, I believe that with a more sophisticated evaluation function, the `NegaMax` searcher would perform significantly better. The code for the `NegaMax` searcher is provided in Listing 2.

When running `AutoCheckers` with both players running `NegaMax` with depth limits 2 and 7 for `Player1` and `Player2` respectively, I found that `Player2` tended to lose more often than not (5 game difference between the two players over 32 games).

```
1  public class NegaMax extends CheckersSearcher {
2      private int numNodes;
3
4      public NegaMax(ToIntFunction<Checkerboard> e) {
5          super(e);
6      }
7
8      @Override
9      public int numNodesExpanded() {
10         return numNodes;
11     }
```

```
12
13      @Override
14      public Optional<Duple<Integer, Move>> selectMove(Checkerboard board) {
15          numNodes = 0;
16          int bestScore = -Integer.MAX_VALUE;
17          Move bestMove = null;
18          for (Move move : board.getCurrentPlayerMoves()) {
19              Checkerboard alternative = board.duplicate();
20              alternative.move(move);
21
22              int score = -maxScore(alternative, getDepthLimit() - 1);
23              if (score > bestScore) {
24                  bestScore = score;
25                  bestMove = move;
26              }
27          }
28
29          if (bestMove == null) {
30              return Optional.empty();
31          }
32
33          return Optional.of(new Duple<>(bestScore, bestMove));
34      }
35
36      private int maxScore(Checkerboard board, int depth) {
37          numNodes += 1;
38          if (depth == 0 || board.gameOver()) {
39              return getEvaluator().applyAsInt(board);
40          }
41          int maxScore = -Integer.MAX_VALUE;
42          for (Checkerboard alternative : board.getNextBoards()) {
43              int score = -maxScore(alternative, depth - 1);
44              maxScore = Math.max(maxScore, score);
45          }
46          return maxScore;
47      }
48  }
```

Listing 2: NegaMax Searcher