Why I think my evaluation function is reasonable

I believe my function is reasonable because it evaluates the entire board and checks for patterns that are advantageous or disadvantageous and states that set up an unblockable state.
At the beginning of the game my function considers how many possible 4 in a rows there are for each position. The ones such as middle spots will have the highest score as there are more winning combinations that utilize those spots.
After more coins have been placed and patterns begin to form on the board the evaluation will then be weighted heavily on those patterns. I weighed the score of patterns based on what I believe provide a better chance of winning.
If max is represented as 1 and 0 as empty, an example of a pattern that will be searched for is 01100 horizontally which will guarantee a win for max and give a high score.

Numerical expression

Variables visual representation

$let\ 1 = max,\ 0 = empty,\ h = horizontal,\ v = vertical,\ d = diagonal$

$h1 \equiv \#\ of\ 1100,\ 0011$
$h2 \equiv 0110,\ 1110,\ 0111,\ 11011,\ 001100$
$h3 \equiv 00110$
$h4 \equiv 01110$
W = 1111
$v \equiv 1110$

$d1 \equiv 0110$
$d2 \equiv 1011,\ 1101$

Expression

$f \equiv number\ of\ possible\ 4\ in\ a\ rows$

$\tilde{h}1 \equiv h1\ for\ min$ $w = 200000(W - \sim W)$

$$g = 50(h1 - \tilde{h}1) + 200(h2 - \tilde{h}2) + 300(h3 - \tilde{h}3) + 500(h4 - \tilde{h}4) + 100(v - \tilde{v}) + 100(d1 - \tilde{d}1) + 200(d2 - \tilde{d}2)$$

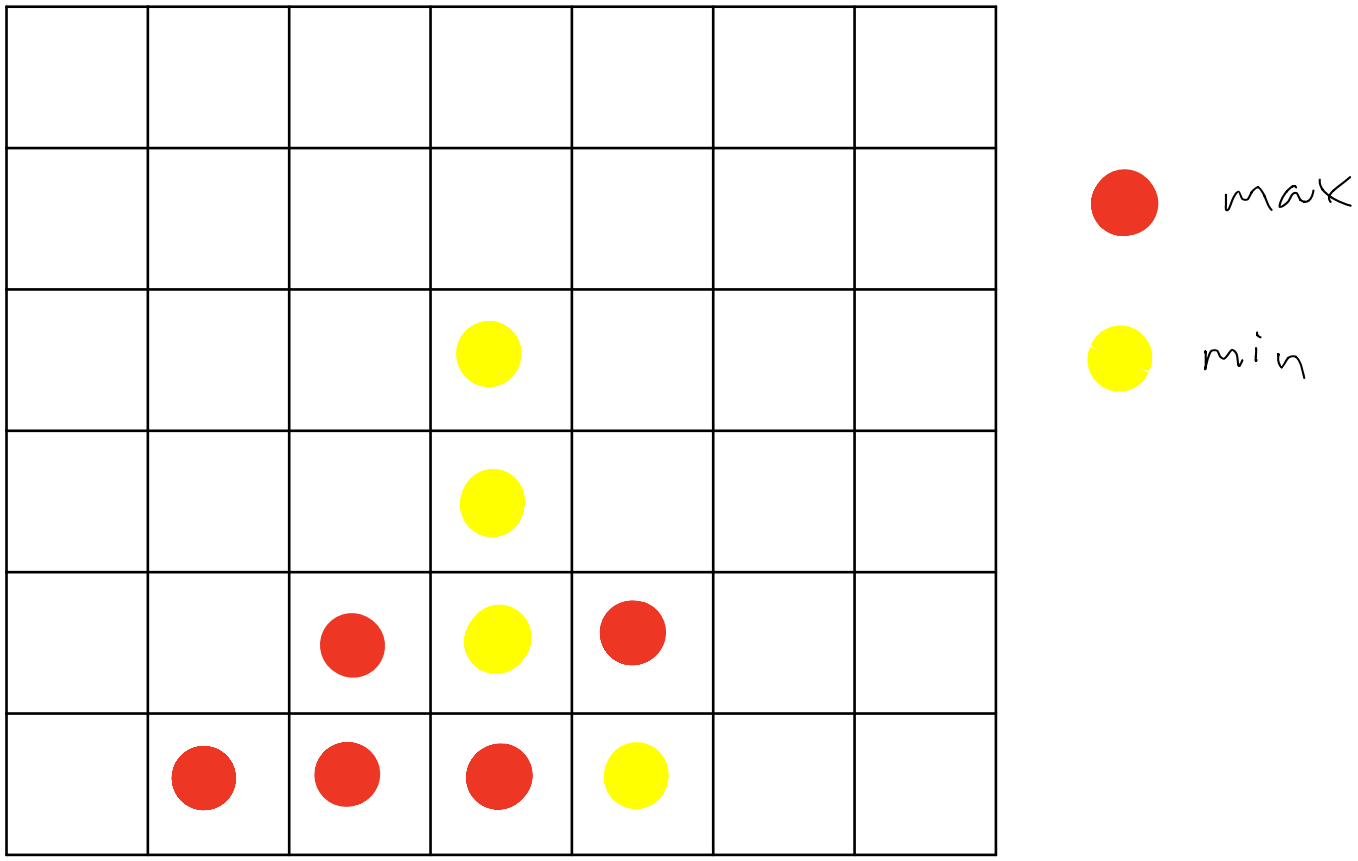
$when\ getCoins() < 12$
$eval = f + g + w$

$when\ getCoins() >= 12$
$eval = g + w$

$f \equiv$

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Playing as Max



$f = 4+5+8+7+8-10-13-13-5 = 32-41 = -9$

$g = 50(0-0) + 200(1-0) + 300(0-0) + 500(0-0) + 100(0-1) +$
$100(0-0) + 200(0-0)$

$= 200 - 100 = 100$

$eval = 100 - 9 = 91$

```
class MaxPoint implements Comparable<MaxPoint>{
       public MaxPoint(int x, int y){
           this.x = x;
           this.y = y;
       }
       int x, y;
       @Override
       public int compareTo(MaxPoint o2) {
           if (this.y > o2.y){
           return -1;
           }
           if (this.y < o2.y){
               return 1;
               }
           else {
               return 0;
           }
       }
}
class MinPoint implements Comparable<MinPoint>{
   public MinPoint(int x, int y){
       this.x = x;
       this.y = y;
   }
   int x, y;
   @Override
   public int compareTo(MinPoint o2) {
       if (this.y < o2.y){
       return -1;
       }
       if (this.y > o2.y){
           return 1;
           }
       else {
           return 0;
       }
   }
}
```

```
private PriorityQueue<MaxPoint> successor_func_max(GameStateModule state){
     PriorityQueue<MaxPoint> queue = new PriorityQueue<>();
     int value;
     for (int i = 0; i < state.getWidth(); i++){
         if(state.canMakeMove(i)){
             state.makeMove(i);
             value = eval(state);
             MaxPoint point = new MaxPoint(i, value);
             queue.add(point);
             state.unMakeMove();
         }
     }
     return queue;
 }

 private PriorityQueue<MinPoint> successor_func_min(GameStateModule state){
     PriorityQueue<MinPoint> queue = new PriorityQueue<>();
     int value;
     for (int i = 0; i < state.getWidth(); i++){
         if(state.canMakeMove(i)){
             state.makeMove(i);
             value = eval(state);
             MinPoint point = new MinPoint(i, value);
             queue.add(point);
             state.unMakeMove();
         }
     }
     return queue;
 }
```

To implement alpha beta pruning ordering I used a PriorityQueue and two variants of a Point called MinPoint(x,y) and MaxPoint(x,y) where x is column where the coin is placed and y is the value returned by the evaluation function. These point variants are used so they can be sorted in the correct order in a PriorityQueue. A MaxPoint PriorityQueue will put larger values first and a MinPoint PriorityQueue will put smaller values first.

I used two different successor functions for each player. The function will iterate through each possible move. It will make a move at position ‘i’, use my evaluation function to set ‘value’, then add a point(i, value) to the PriorityQueue.

Inside the alphabeta function. Looking at the case of Max turn.

```
      ...
       if (playerID == player) {
           value = Integer.MIN_VALUE + 1;
           int bestVal = Integer.MIN_VALUE;
           PriorityQueue<MaxPoint> queue = successor_func_max(state);
           while (queue.peek() != null) {
               MaxPoint point = queue.poll();
               int i = point.x;
               if (state.canMakeMove(i)) {
                   state.makeMove(i);
                   value = Math.max(value, alphaBeta(state, depth, alpha, beta,
opponent));
                   state.unMakeMove();
                   if (value > bestVal) {
                       bestVal = value;
                       if (depth == 1) { // top of recursion, make our move choice
                           bestMoveSeen = i;
                       }
                       bestVal = Math.max(bestVal, value);
                       alpha = Math.max(alpha, bestVal);
                   }

               }
       ...
```

When alphabeta is called a PriorityQueue will be created using the appropriate successor function. To iterate through the successors the first member of the queue will be popped. As the queue is sorted automatically, the best successor is guaranteed to be searched first. The i value is taken from the point and the function proceeds as it originally did.