# Connect 4 AI

David Friedman Sridatt Bhamidipati

March 2, 2016

## Contents

## 1 Preface

This is a project for ECS 170, Artificial Intelligence, at UC Davis. We wrote AI using minimax and alphabeta pruning in order to play and win connect 4.

## 2 Evaluation

### 2.1 Functions

Our evaluation function is a combination of two evaluation functions. Before getting into them we must first describe what a winning group is, as we will use this terminology throughout. A winning group is any combination of four slots on the board in one row, column, or diagonal. There are 69 total winning rows in the game. A winning group is owned when only one player has pieces in that group.

#### 2.1.1 Number owned in winning group

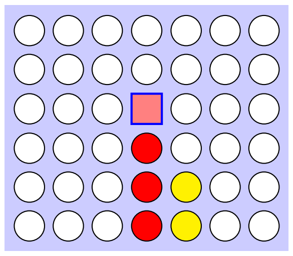
Our first evaluation function is a a count of all winning groups we own, weighted by the number we own in that group minus the count of all winning groups the opponent owns, again weighted by the number they own in that group. The equation for this is:

$$f_1(x) = \sum_{n \in s_{1,r}} numOwned_1(n) - \sum_{n \in s_{2,r}} numOwned_2(n)$$

where $numOwned_{r,n}$ represents the number owned by player $r$ in the winning group n, and $s_r$ represents the winning groups soley owned by player $r$.

#### 2.1.2 Odd/Even Threats

Our second evaluation function is more complicated. First it iterates over a list of all possible winning groups, and finds the threats. A threat is a slot that would complete a 4 in a row for either team. It then iterates over each of these threats and tallies them, categorizing them based on the team who owns the threat and the threats polarity (odd/even). Even threats are threats that have an even number of (necessarily) empty spaces above them, while odd threats are threats with an odd number of empty spaces above them (figure 1). The function ignores a threat if there is a threat below it from the opposite team with opposite polarity (the reasoning will be explained in the section 2.2). The function then checks if player 1 has more odd threats than player 2, if player 2 has two more odd threats than player one, or, if in the case they both have the same number, if player 2 has at least one even threat. If the former is satisfied it returns 100, signifying that player one is in the lead. For the latter two it returns $-100$, signifying the opposite, and if none of the cases are satisfied we


(a) Even red threat

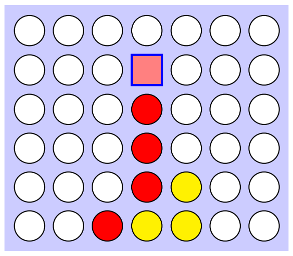

(b) Odd red threat

Figure 1: Types of threats (threat marked with squares)

return 0. The equation looks like the following piecewise:

$$f_2(x, y) = \begin{cases} 100 & odd(x) > odd(y) \\ -100 & odd(y) > odd(x) + 1 \\ -100 & odd(x) = odd(y) \text{ and } even(y) > 0 \\ 0 & otherwise \end{cases}$$

where x is all threats by player 1, and y is all threats by player 2.

## 2.2 Reasoning

Our first evaluation function is straightforward. We want to value states where we have more control of the winning positions more. If we control a winning position (only our pieces are in it), we can use it to win. If the enemy controls it they can use it to win. Additionally, if more of our/their pieces are in a winning position, we need less moves to complete that win, which is favourable.

Our second evaluation function is more complicated. It relies on the fact that red (player 1) will only ever move when there are an even number of pieces on the board, and yellow (player 2) will only ever move when there are an odd number of pieces on the board. Additionally it relies on the fact that since columns have an even number of slots, red generally has more control over odd positions than even. This is because red plays during all odd turns, and all columns are an even number, meaning once a column has been filled it is generally reds turn again. This means red has an easier time taking odd threats, and yellow has an easier time taking even threats. Therefore, it is logical that red can generally block all of yellows threats and play its own winning threat if it has more of these odd threats. Yellow on the other hand needs to have two more odd threats, as if it only has one more red may use its advantage in playing odd rows to block all these threats before yellow can win. The last case is because if they have an equal number of threats, once red has blocked all of yellows odd threats, yellow will generally have an easier time grabbing an even row, and thus winning with its even threat.

As mentionedin setion 2.1.2, we need to be explain *why* we remove threats above another threat if the threat is from the other player and has a different

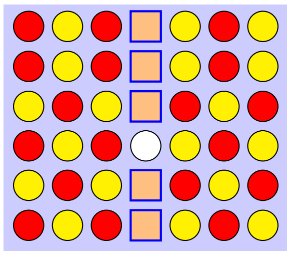


Figure 2: Useless yellow threat

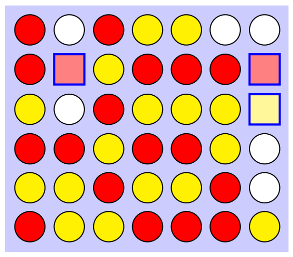
parity. This is quite simple. Take figure 2 as an example. Here orange squares represent threats both players own. Note that yellow's lowest threat is even. However, in order to take it the slot beneath must be filled. Because there are an even number of pieces in play, it is reds turn. Red has an odd threat in the first row. Therefore yellow's even threat may never be taken, as red must play below first and win. Additionally if we for now ignore all but yellows first even threat and reds even threat above it, we can see that the same thing would occur. Therefore in general threats of opposite parity to lower threats by the other player are useless. We can extend this to also mark shared threats by yellow on odd levels as useless for yellow, and shared even threats as useless for red, but we leave this distinction out of our code to keep the runtime of our evaluation fast.

## 2.3 Worked Example

We will use figure 3a as our example game state. First we will run our second evaluation function, as it is much simpler to run in this case. Threats have been marked their respective team colors. Red has two odd threats, and yellow has one even threat, however one of reds threats is useless because yellow's threat is below it with different parity. However, we still have $odd(x) > odd(y)$, and our evaluation function returns 100.

Our second heuristic is a bit lengthier to run through, as there are a total of 69 winning positions we need to consider. Here, however, only 8 winning positions are not invalid due to having two different players in them, and of those two (the rightmost column starting from the top and the second from top) have no pieces in them. We can easily count the number owned by red and yellow in each group, and we see that the result is $7 - 6 = 1$.

Both our evaluation functions favour red, and as seen in figure 3b, we have guessed correctly as red does win.


(a) Red to win

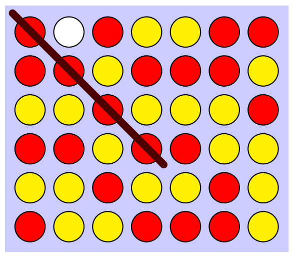

(b) Red wins

Figure 3: Example

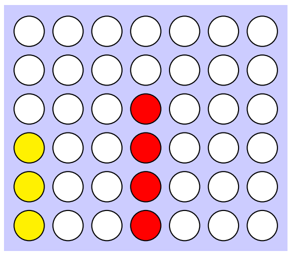
## 2.4 Abandonded function - Islands

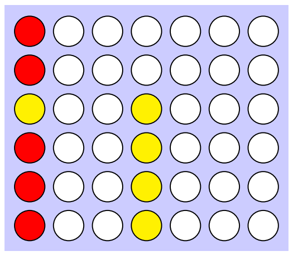
Our last idea for an evaluation function was to count the number of islands created by player 1 and player 2 on the board, compare them, and return a reasonable evaluation value. An island is essentially a connected component in an undirected graph, and we viewed it in a perspective where having more of our same coins clumped together was advantageous to us because we can then make moves from various angles and end up forcing victory over our opponent. Thus, given that the number of player 1's coins and player 2's coins on the board are about the same, having the lesser number of islands could be more advantageous. We solved this by implementing a depth first search, which we would run on each unvisited cell. We proceeded to count up the number of islands for each player. Finally, it was important that we returned the difference of the number of islands rather than the value of the least island count, since we took into consideration that the evaluation should not be extremely high if the difference was small. While it seemed to perform decently, it was much much slower than our other evaluation functions (due in part to the lack of easy ways to perform it using bit manipulation), so impaired the depth our algorithm could reach, and was ultinmately scrapped.

## 3 MiniMax Against poor AI

We win every game in the section below. We played around half of the games as player one, and around half of the games as player two (Our player is indicated below each board).

### 3.1 Against StupidAI

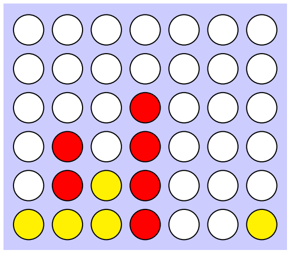

(a) As player 1


(b) As player 2

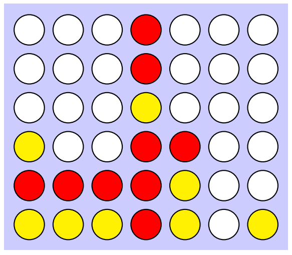
If MiniMax goes first see 4a otherwise see 4b. Games depend only on who goes first (we use no randomness so seed has no effect, last three game plays identical so omitted for brevity). We win all 5 games.

## 3.2 Against RandomAI

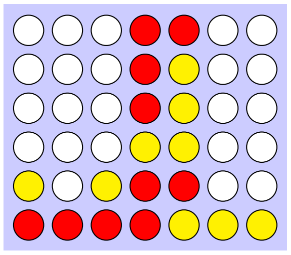
Against the random AI we win all 5 games, both as player 1 and as player 2.



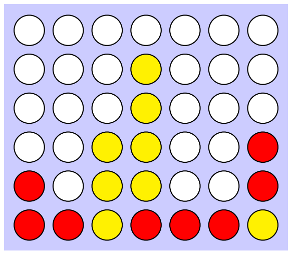
(a) As player 1



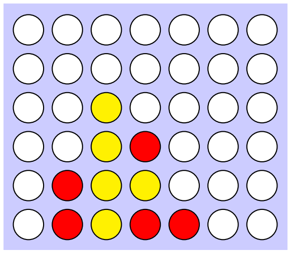
(b) As player 1



(c) As player 1



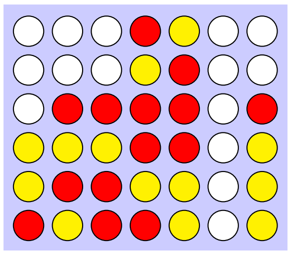
(d) As player 2

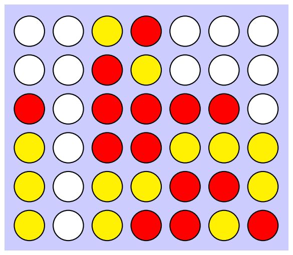


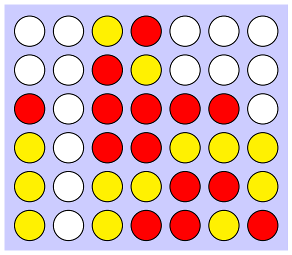
(e) As player 2

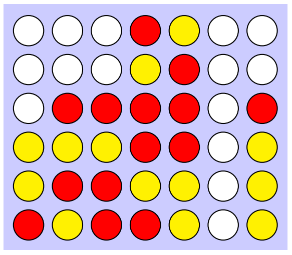
## 3.3 Against MonteCarloAI

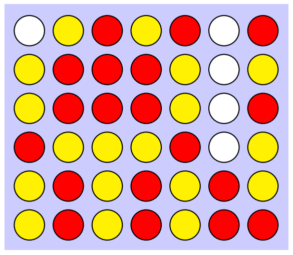
Against Monte Carlo AI we win every game. Shown below are 10 games against MonteCarloAI, half as player 1 and half as player 2. We win all 10 games.

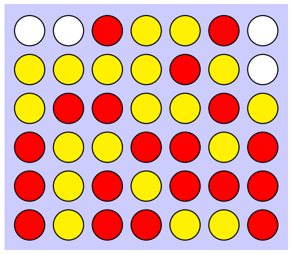

(a) As player 1

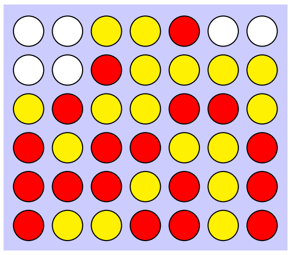

(b) As player 1

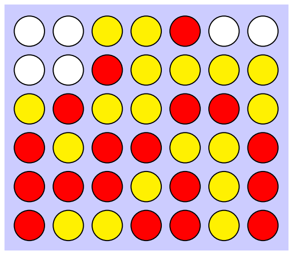

(c) As player 1

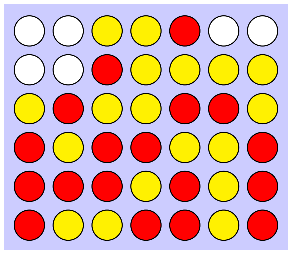

(d) As player 1

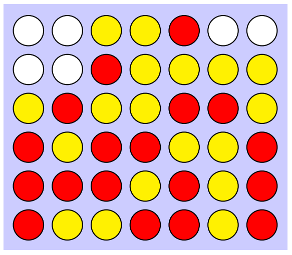

(e) As player 1


(f) As player 2


(g) As player 2


(h) As player 2

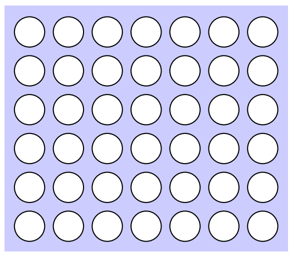

(i) As player 2

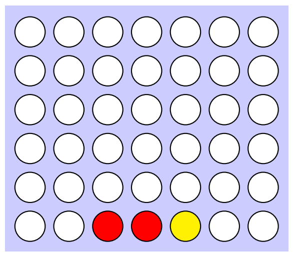

(j) As player 2

## 4 Successor Function

We perform two different types of ordering: static and dynamic. The static ordering is very simple. We initialize an array storing what we have determined to be an ordering that usually results in a cutoff. First we try the center, then the two columns around it, then the ones next to that, then the edges. This gives a decent performance boost because central columns are part of more winning groups than non-central edges.

The dynamic ordering is done using the concept of a *killer heuristic*. The basic idea is that moves at the same level of the game tree are often quite similar, and what helped us prune at one node may help prune at a sibling. This is clearly the case in connect 4. Take figure 7b for example. If we started our search at the empty board state shown in 7a we would arrive at this state at a depth of 3 in our search. However we could arrive at this state in two different ways. No matter which way we use to arrive the best move choice will remain the same. By storing this best move in an array of length 42 (the number of moves possible) at index 3 (our current depth) we can make it so the next time we arrive at this state we may be able to immediately find the best choice and possibly prune the rest of the search tree by simply trying the stored best move. Although other states may be encountered between arriving at two instances of the same state, the optimal move still tends to remain the same for similar states. For example in figure 7b, for any placement of the second red piece (holding the central piece constant) except the central column, yellows best move is always the central column.


(a) Initial state


(b) Three ply search

### 4.1 Abandoned Successor functions/modifications

In addition to the scheme used, we also attempted utilizing caching to allow for quick pruning of the game tree. We set up a simple LRU cache with a limited number of entries, and tried running our alpha-beta algorithm on it for different cache sizes. Ultimately, having no cache was faster than having a cache, probably due to our poor hash generation, as we used a simple hash function that essentially just performed a bitwise and on the bitboard representing player 1 and 2, and then returned the lower 32 bits. In retrospect it is obvious why this failed, as states that fill the same slots but with different colors are extremely common in our game tree, and this would cause a very large number of colisions when retrieving from the hashtable.

There are a number of alternate hashing schemes that may have worked better, such as simply returning the long that encodes player 1's board, or extending Java's hashmap to use 64 bit keys, in which case we could encode every state as a unique 63 bit sequence (42 bits for blacks board, then 3 bits per column for the height of the column). Alternatively we could use a technique often used in chess playing AI, the Zobrist hash, which calculates a random number for each possible state of a single slot (excluding empty slots), and then loops over the board and xors the corresponding random numbers for each slot together. It may seem slow at first (we have to loop over 42 different slots initially) but the key lies in computing subsequent hashes. Since only one state has changed, we can simply xor the changed squares value with our old hash to get a new one. If we take a move back we simply xor the random value of that move, which will return the hash to its previous value. This hash function generally gives decent results.