Optimizing the Minimax Algorithm for Connect4

By Justin Thakral

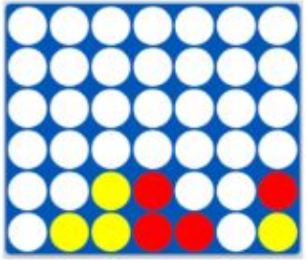
Algorithm Purpose

-Used in turn based games like connect 4, checkers, tic tac toe and chess to find the best move

-Minimax input is the game state, depth and maximizing player. For connect 4 the game state is a 2D vector, where 0 represents empty, 1 red, 2 yellow.

-Minimax output is the score for the move. Output is deterministic as the algorithm always returns the same move in the same game state.

```
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 2, 1, 0, 0, 1]
[0, 2, 2, 1, 1, 0, 2]
```

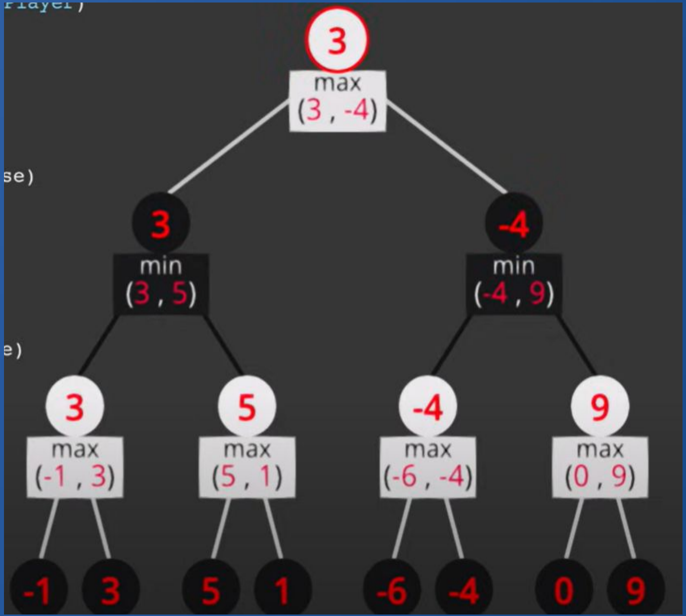


Algorithm (not optimized)

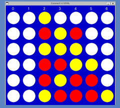
```
function minimax(position, depth, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, false)
         maxEval = max(maxEval, eval)
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, true)
         minEval = min(minEval, eval)
```



From Sebastian Lague on youtube

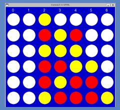


Interesting features of the Algorithm

-Minimax has solved tic tac toe, checkers and connect4 (by James D. Allen in 1988)

-Chess is not yet solved. Stockfish, a popular chess engine, has solved chess using an optimized minimax algorithm with 7 pieces remaining.

-Stockfish is so optimal a powerful computer can minimax chess with a depth of 60



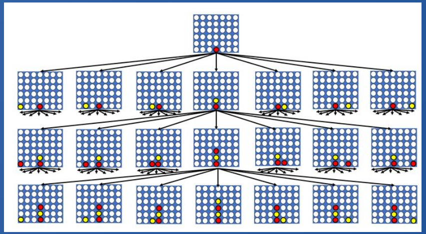
Algorithm Analysis (not optimized)

-space complexity = O(d) for the call stack

-runtime = O(b^d) where b = branches(possible moves) and d = depth(moves you are looking ahead)

-Problem Statement 1: What effect does increasing depth have on runtime? Increasing depth is very high reward (ai plays better moves) but computational VERY costly 7^6(depth) = 117649 meanwhile 7^10 = 282,475,249

-Problem Statement 2: How important is it to optimize minimax?



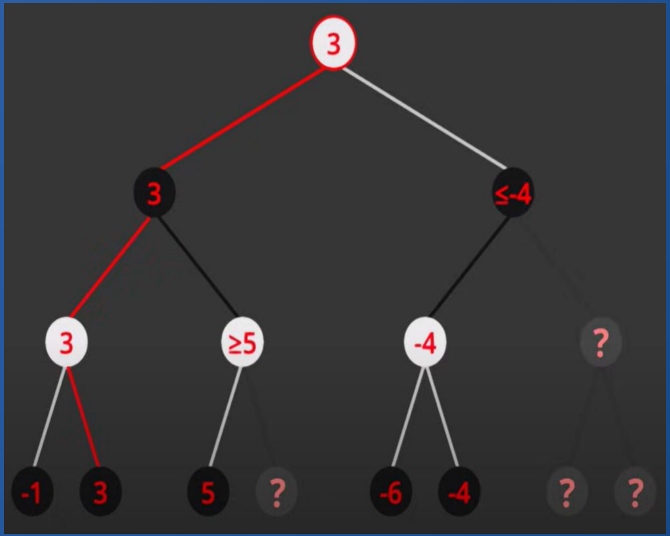
Optimization

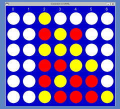
-It is extremely important to optimize the minimax algorithm so you can run highers depths.

-Alpha Beta Pruning: Carry two bounds alpha (best seen for maximizer) and beta (best for minimizer) through recursion. As soon as alpha ≥ beta, prune the remaining siblings: none can improve the outcome, so skip them

Mid Row First - Order columns as [3, 2, 4, 1, 5, 0, 6] so center is tried before edges. Center moves are more likely to connect prune branches sooner.

-Early Win Detection - If a move leads to a win, stop looking instantly as it is automatically the most optimal move.

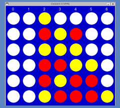




Experimental Plan

-To prove how much of an impact optimizations have I could run the raw minimax algorithm vs the improved one I created. I tested a few different game states that each optimization would do well on.

-To prove the success of the AI I had my friends and family play against it. I also inputted random game states/moves against it.

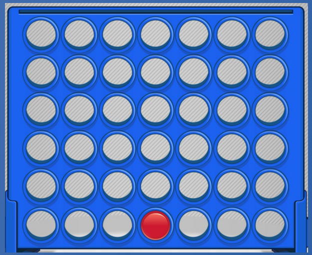


Results

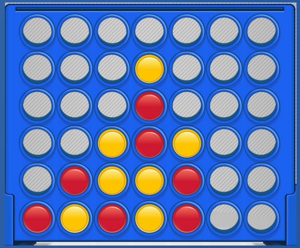
-Optimizations had a major impact on decreasing runtime.

-Even at a depth of 7 nobody beat it but few games had ties. It swiftly beat random columns in under 10 moves.

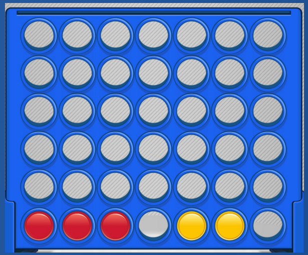
firstBoard

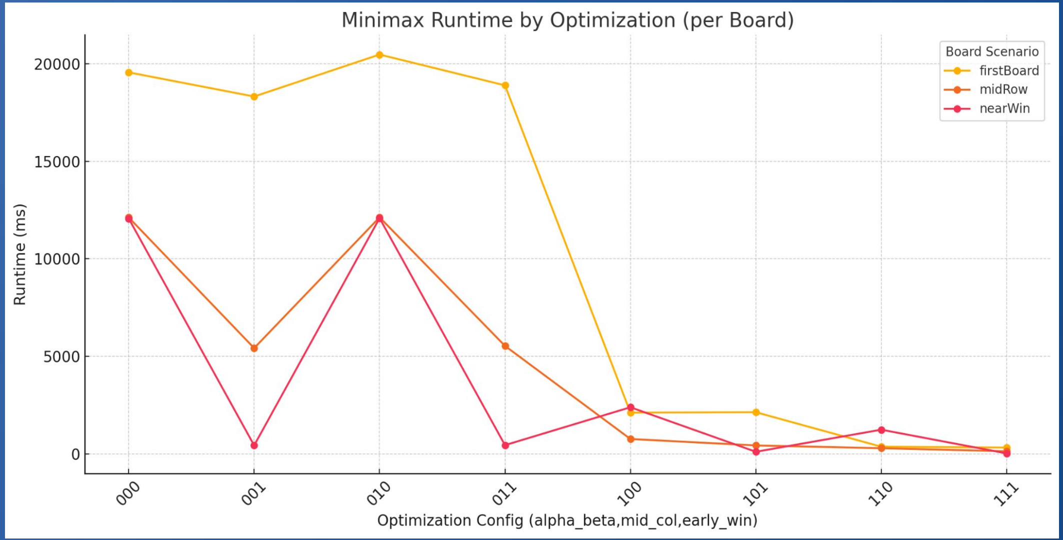


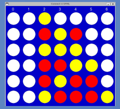
midRow



nearWin







Implementation

-I wrote my code in c++ and used SFML to draw the board(#include <SFML/Graphics.hpp>)

-The data structure is a 2D vector which is a matrix.

-In this project, I also use an important matrix algorithm, the sliding window and important operation, directional transformation.

```
{0, 1},   // Horizontal (right)        (0,0)
{1, 0},   // Vertical (down)           (1,1)
{1, 1},   //  Diagonal down-right      (2,2)
{-1, 1}   // Diagonal up-right         (3,3)
```

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)

Demo

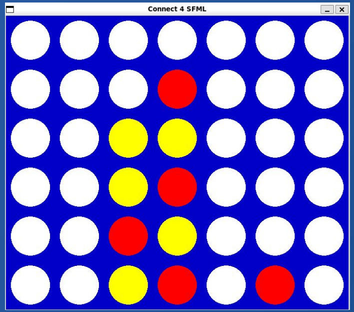
Tester Demo

```
dully@Justin:/mnt/c/Users/justi/OneDrive/Desktop/MyProjects/cs375/thakral_j_finalProject$ ./tester midRow.txt
Benchmark written to results.csv
```

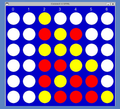
results.csv

```
alpha_beta,mid_col,early_win,runtime_ms,config
0,0,0,12065,"0,0,0"
0,0,1,429,"0,0,1"
0,1,0,12087,"0,1,0"
0,1,1,442,"0,1,1"
1,0,0,2385,"1,0,0"
1,0,1,101,"1,0,1"
1,1,0,1235,"1,1,0"
1,1,1,18,"1,1,1"
```

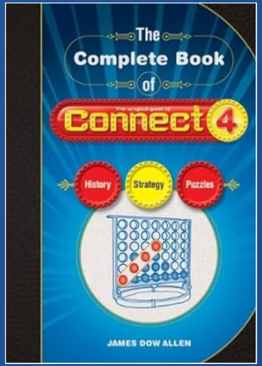
Game Demo




```
MiniMax runtime: 1782 ms
AI moves to column 2
Col 3: -21
Col 2: 97
Col 4: 9
Col 1: -10
Col 5: -21
Col 0: -49
Col 6: -15
MiniMax runtime: 1997 ms
AI moves to column 2
```



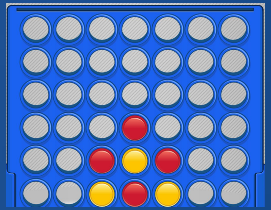
Limitations/Future Work



-Currently, my program is on a depth of 9 giving response in roughly 1 second. On a depth of above 16, moves take an hour to calculate.

-By anazlyzing how James. D Allen solved connect 4, we can improve algo

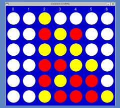
-He used techniques I used like Alpha-Beta pruning, mid row first and early win detection. Also techniques I did not use like the ones below



-Symmetry Reduction - Playing column 0 is equivalent to playing column 6 in a symmetric position, avoiding computing mirror positions

-Precomputation - Start states and endstates can be cached in a look up chart

-Forward Pruning - Avoids searching irrelevant branches.



Conclusion

-Overall, Minimax is a recursive exponentially growing algorithm that is used to find the **MOST** optimal move in turn based games.

-By using optimizations techniques like alpha-Beta pruning, mid row first and early win detection I heavily reduced runtime. So optimizing the minimax algorithm is extremely important because the raw minimax has runtime O(b^d)

-Adding in all the optimizations James. D Allen included could allow me to create the perfect AI that will play the most optimal move.