**CS 4346- Project #2 Report**

**By Russell Sullivan**

**The Problem Description:**

Connect Four is a two-player, turn-based strategy game where participants alternately drop player discs into a vertical grid. The objective is to connect four of one's own discs in a horizontal, vertical, or diagonal line before the opponent does.

This project implements a game-playing program that allows a human or AI player to compete against an AI opponent. The AI uses an adversarial search called the minmax algorithm enhanced with Alpha-Beta pruning to decide its moves.

The goal is to demonstrate how to model the game as a search problem, implement an efficient adversarial search, and compare different evaluation strategies under varying search depths.

**The Domain:**

The Domain of the problem, games, is best characterized by a finite, deterministic, perfect-information environment. Each move in the game of Connect-Four is deterministic: not allowing randomness. Both players are able to always observe the entire state of the Board. In this game, players use a Board object which is represented as a 6x7 matrix, allowing for a maximum of 42 total moves. The state space for Connect Four has an upper-bound of 342 and a lower bound of about 1.6 x 1013.[1]

**Methodologies:**

The minimaxAB performs a limited depth-first min-max recursive search with alpha-beta pruning and hueristic evaluations for determining the ‘best’ move. The minMaxAB() function receives integer arguments:

*depth*: representing the cutoff depth/height of the simulated game tree

*alpha*: representing the best value the maximizer can currently promise

*beta*: representing the best value the minimizer can currently promise

The minMaxAB() function also receives a boolean value, called *playerTypeFlag*, which indicates whether this call is for MAX\_PLAYER (true) or MIN\_PLAYER (false).

The minMaxAB() function’s recursive base case (or terminal case) is to return an integer gathered from the heuristic evaluation() function if, and only if, a call to minMaxAB() reaches conditions:

*Depth == 0 or*

*Max Player Wins or*

*Min Player Wins or*

*Board is full (Draw)*

Once one of these conditions has been met, a heuristic evaluation function returns a value or ‘score’ indicating a strong/weak game path depending on the evaluation function’s set priorities. The evaluation functions (within Heuristics.h):

*//* ***evaluateWithCenterBias()***

*// description: prioritize control of the center by assigning a weight based on distance from the center.*

*// receives: a constant reference to the Board object*

*// returns: the score ( positive if MAX\_PLAYER, negative otherwise )*

*// runtime: linear O(R\*C=42)*

*// memory: ~ 20 bytes of local storage*

*//* ***evaluateWithSparseBias()***

*// description: prioritize playing into less-filled columns.*

*// receives: a constant reference to the Board object*

*// returns: the score*

*// runtime: linear O(R\*C=42)*

*// memory: dynamically allocates vector<int>(C=7) ~50 bytes*

Essentially, this minMaxAB() creates a game tree with all possible moves and then filters out the best scores (representing the highest likelihood of winning) from an initial move 0-6. This ‘initial move’ is known by minMaxAB()’s calling function: bestMove().

bestMove() role is to find and return the optimal ‘player’ move for each column. It begins by setting the worst-case initial values:

*MAX\_PLAYER -> -infinity (or the smallest value of an int)*

*and for MIN\_PLAYER -> +infinity (or the largest value of an int)*

*Then, bestMove() loops through each column c from 0 .. 6:*

*Check if col is full;*

*Simulate the move;*

*Call miniMaxAB() to score the move;*

*If the move is an improvement, set as bestMove;*

*Undo the move;*

*Return the best Move*

bestMove(), lastly, returns the optimal move representing the highest likelyhood of winning the game. This function is called within the main game loop and is used by both min / max ‘players.’

**Source Code Implementation:**

The source code for this project was developed under a series of tests on various hypothesis having to do with decision making based on different recursive heuristic evaluation(s). Each algorithm used was tested on pencil and paper before being tested using the g++ compiler.

Important features in this project include the use of depth-controlled minMax with Alpha-Beta pruning, modular heuristics, efficient evaluations, easy tuning, and dual-depth command-line interface for user-friendly testing of the two cutoff depths.

**Source Code:**

The latest of this code can be found on my gitHub [here](https://github.com/bigbadcyborg/ai-connect-four).

**A Copy of the program Run:**

Ctrl+Click [here](analysis-output/README.md) for instructions to run the program and develop the tables to ./analysis-output/data/

**Analysis of the program:**

*The following lines are hyperlinks (CTRL+Click) to each result case ./analysis-output/data/analysis-output-main.exe-X-Y , where*

*X represents depth in Max(depth = X ) and*

*Y represents depth in Min(depth=Y)*

*let EV#1 be evaluateWithCenterBias()*

*let EV#2 be evaluateWithSparseBias()*

EV#1 Max( depth = 2 ) vs EV#2 Min( depth = 2 ): MAX Wins

[EV#1 Max( depth = 2 ) vs EV#2 Min( depth = 4 ):](analysis-output/data/analysis-output-main.exe-2-4.txt) MAX Wins

[EV#1 Max( depth = 2 ) vs EV#2 Min( depth = 8 ):](analysis-output/data/analysis-output-main.exe-2-8.txt) MAX Wins

[EV#1 Max( depth = 4 ) vs EV#2 Min( depth = 2 ):](analysis-output/data/analysis-output-main.exe-4-2.txt) MAX Wins

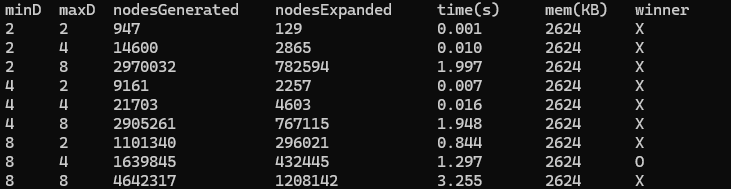
[EV#1 Max( depth = 4 ) vs EV#2 Min( depth = 4 ):](analysis-output/data/analysis-output-main.exe-4-4.txt) MAX Wins

[EV#1 Max( depth = 4 ) vs EV#2 Min( depth = 8 ):](analysis-output/data/analysis-output-main.exe-4-8.txt) MAX Wins

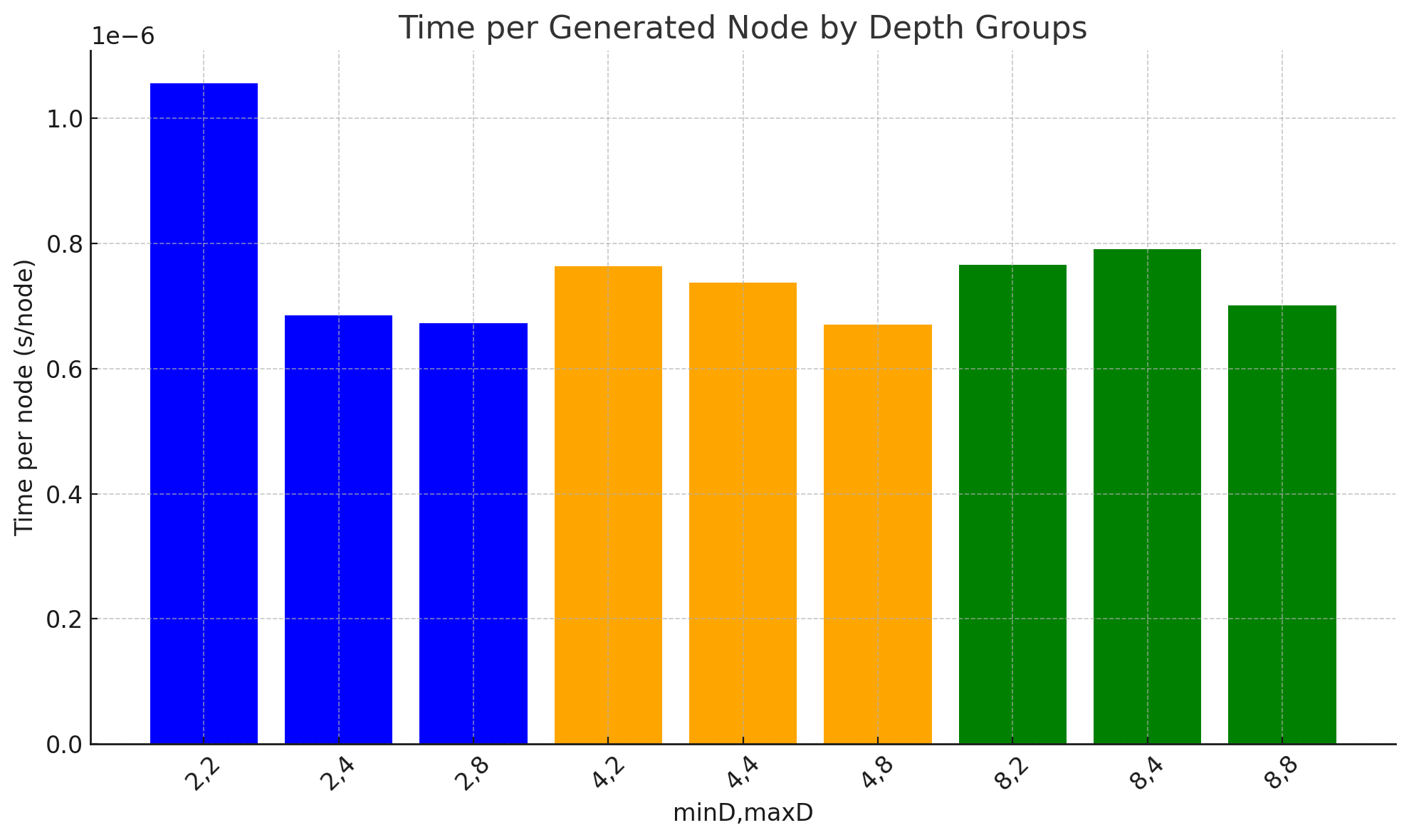
[EV#1 Max( depth = 8 ) vs EV#2 Min( depth = 2 ):](analysis-output/data/analysis-output-main.exe-8-2.txt) MAX Wins

[EV#1 Max( depth = 8 ) vs EV#2 Min( depth = 4 ):](analysis-output/data/analysis-output-main.exe-8-4.txt) **MIN Wins**

[EV#1 Max( depth = 8 ) vs EV#2 Min( depth = 8 ):](analysis-output/data/analysis-output-main.exe-8-8.txt) MAX Wins

**Tabulation of Results:**

**Analysis of the results:**

The *runtime* results can be calculated by dividing *time* by *nodesGen*erated*.* The average time comes out to .6 microseconds (μs) per node. The timing curve is a remarkably straight line.

In this script, no virtual memory is allocated.

The main RAM memory for each game execution is ~2.6 MB because the OS allocates this region for the thread’s stack until it is exhausted.

After several million nodes generated/expanded- the threads memory of only 2.6 MB was no where near being exhausted.

This reflects a very memory-efficient and time-efficient algorithm.

Max wins in 8 out of 9 scenarios. This is expected as Max has the advantage of first turn which results in deeper look-ahead.

The one scenario (minDepth=8, maxDepth=4) when Min wins because of its capitalization of larger search depth.

**Conclusion:**

In this project, I not only implemented two different evaluation heuristics (center-bias and sparse-bias), but also developed a robust instrumentation harness to measure the impact of search depth and heuristic choice on performance. The combination of alpha-beta pruning and our fixed-size board representation proved to be very memory-efficient, keeping the process's RSS at a constant ~2.6 MB. Runtime analysis showed a consistent per-node cost (~0.6 μs/node) and showed that nodes grow exponentially as search depth increases. MAX won eight out of nine depth pairings, with MIN only winning when given greater depth.

**References:**

[1]https://web.stanford.edu/class/aa228/reports/2019/final106.pdf?utm\_source=chatgpt.com

[2]https://www.youtube.com/watch?v=l-hh51ncgDI&ab\_channel=SebastianLague

[3]<https://visualgo.net/en/recursion?slide=1>

[4]<https://stackoverflow.com/questions>

[5]https://www.geeksforgeeks.org/