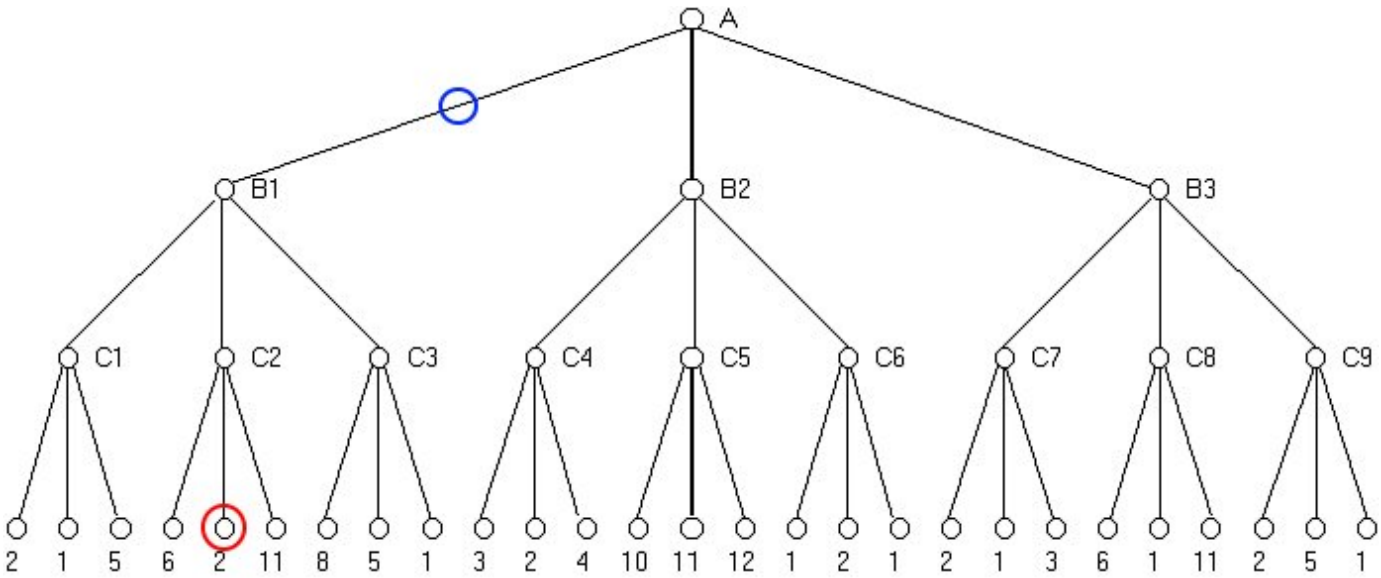
Connect M Report

Alexander Bachmann & Joshua LaRocca

Data Structures

We chose to use three C++ standard library vectors as our primary data structures: one 2D vector for our game board and two 1D vectors to store the moves taken by each player. The vectors that contain the sequence of human and AI moves were created simply out of curiosity and verification. When our friends managed to beat our AI, though rare, it was very exciting and we wanted to be able to recreate that winning sequence to verify their win (the minimax algorithm will always choose the same sequence of moves if the same sequence are chosen by the human thus allowing for games to be recreated exactly as they occurred).

Another data structure we used, though small, was the C++ standard library pair. Using pairs served as a solution to a problem we encountered: as we traversed deeper into the tree, traveling further away from the current game board, we lost track of the initial column number associated with each future game board's heuristic value. For example, we would use a pair to hold the heuristic value from the red node and the column number from the blue branch.



Heuristic Evaluation

Our heuristic evaluation is simple, though quite effective and produces some interesting behavior which will be discussed later. The current game board being evaluated, which is multiple turns ahead of the actual game, is passed in as an argument. From that game board, the number of potential winning game states remaining for each player is calculated; the difference is then returned so long as the game board that was passed in does not contain a winning state. If a winning state is detected and in favor of the AI, the evaluation value returned is strongly *favorable*, else if the winning state is in favor of the human, a strongly *disfavorable* value is returned.

```python
# Pythonic Pseudocode

def heuristic_evaluation(game_board):

    eval = 0
    self_num_winning_moves = 0
    opponent_num_winning_moves = 0

    self_num_winning_moves = num_horizontal_wins() + num_vertical_wins() +
num_diagonal_wins()
    opponent_num_winning_moves = num_horizontal_wins() +
num_vertical_wins() + num_diagonal_wins()

    eval = self_num_winning_moves - opponent_num_winning_moves

    if(is_game_over()):

        if(AI_winning_state_found):
            eval = 10000

        elif(human_winning_state_found):
            eval = -10001

    return eval
```

Minimax

To save memory, we designed a minimax algorithm that does not create many new game boards, but instead explores future states by mutating the same game board using disk pushing and popping functions. A terminal state is reached when either the explored game board has a winning state or the desired depth has been reached; depth meaning how many turns the AI is searching into the future. When a terminal state is reached, the heuristic evaluation of the game board is passed up recursively through the tree. Two for loops (one for each player) are used to traverse the tree. With each call to minimax, the current player's turn is boolean-toggled, thus changing which for loop is executed. The current player's turn determines whether alpha or beta is evaluated during the call. Each iteration of the loop adds a disk and recursively calls the minimax function; on the way out of the recursive call that same disk is popped, then alpha or beta is evaluated and returned. On the AI turn, alpha is returned and on the human turn, beta is returned. The alpha-beta pruning occurs when beta is less than or equal to alpha and is executed by a single break statement, exiting the current for loop and ceasing exploration of that branch.

```
# Pythonic Pseudocode

def minimax(game_board, depth, maximizing_player_turn, alpha, beta):

    if(depth == 0 or is_game_over()):
        return heuristic_evaluation(game_board)

    if(maximizing_player_turn):

        for each column:
            add_disk_to_column()

            eval = minimax(game_board, depth - 1, false, alpha, beta)
            alpha = max(alpha, eval)

            pop_disk_from_column()

            if(beta <= alpha)
                break

        return alpha

    else:

        for each column:
            add_disk_to_column()

            eval = minimax(game_board, depth - 1, true, alpha, beta)
            beta = min(beta, eval)

            pop_disk_from_column()

            if(beta <= alpha):
                break

        return beta
```

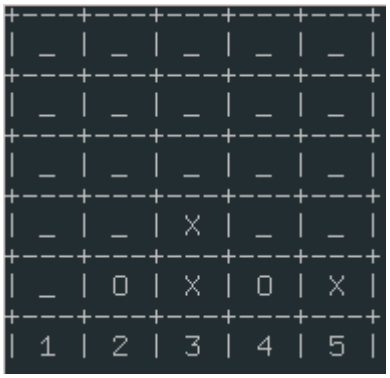
Observed Behavior

After observing many games, we began noticing interesting and repeated behaviors. For all of the following screen shots, the human moves are the X's and the AI moves are the O's.

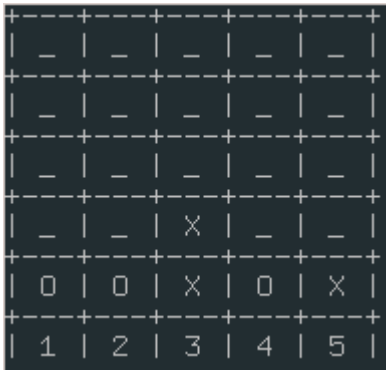
Giving up prematurely

If the AI knows that in the future it will lose (the AI assumes the human will play perfectly), it will choose to take more control of the board instead of choosing to block potential opponent wins.

Given this game state on a connect 3, 5 x 5 board:



If the human human chooses column 3,

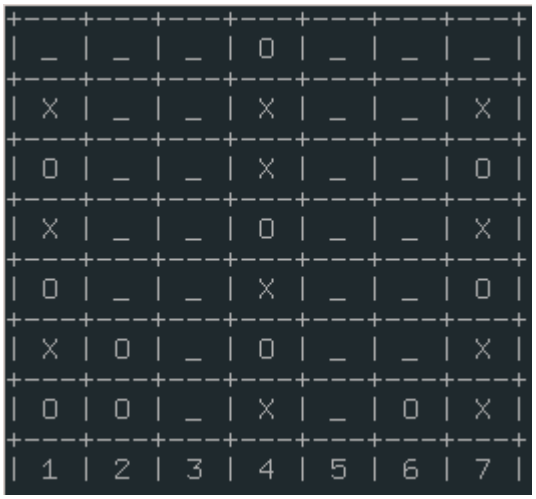


then the AI will choose column 1

At first glance, this decision by the AI seems to not make any sense, until we realize that even if the AI chose to block column 3, the human could choose column 4 which again leads to another winning state the following turn by the human choosing either column 2 *or* 4.

Tactics

Tower Building



The heuristic evaluation function prioritizes increasing the difference between the number of winning game states the AI has remaining and the number of winning game states the human has remaining. Early on, we noticed that the AI strongly prefers to build towers of disks when there is not a clear contiguous line it is pursuing since more winning states are denied for the opponent by building vertically. This tactic makes it very difficult for the AI's human opponents to keep track of diagonal threats, which was by far the most common way human players were defeated.

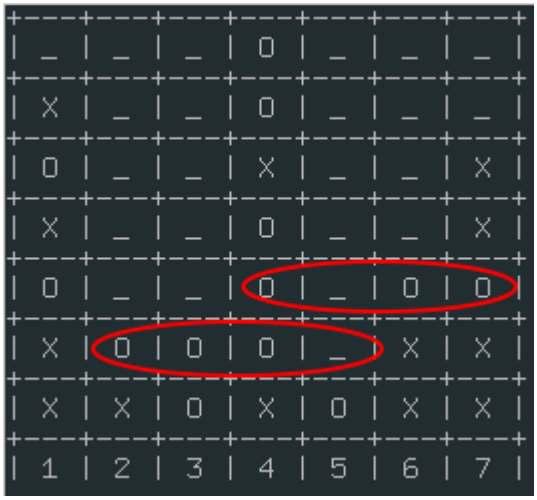
Engulfing

Another tactic that was observed is the AI's proclivity to surround its opponent's disks, rendering all of the human's previous turns useless.

"Playing with food"

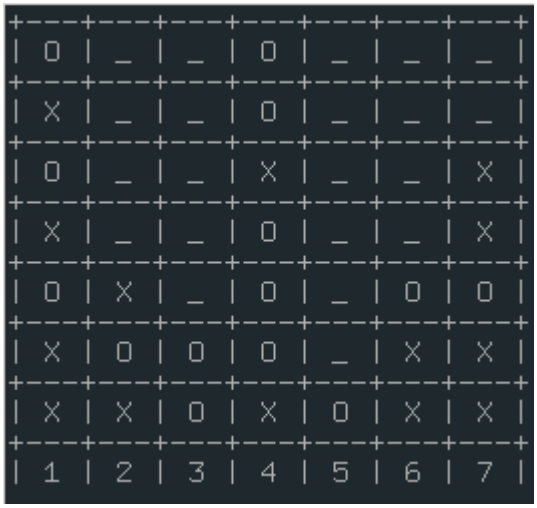
Quite humorously, the AI will choose to "play with its food" meaning that if the AI has winning states, it will not complete the game until it only has one winning state remaining.

Given this game state of a connect 4, 7 x 7 board:



the AI has a winning state no matter the choice of its opponent. If the human blocks at column 5, the AI will win the next turn at column 5.

If the human chooses column 2 (not attempting to block the win from the AI),



then the AI will continue taking more control of the board (by choosing column 1) instead of finishing the match.

Conclusion

While building our AI, we gained a better understanding of how to build a relatively strong connect M AI. The improvements in a minimax AI's skill will come, almost exclusively, from the heuristic evaluation

function. Our heuristic evaluation function assesses a gameboard then returns an integer value indicating how favorable that gameboard is. Our initial heuristic function was flawed: it only counted the number of winning states a gameboard had for the AI, which meant that the AI would choose a board that favored itself but would also completely disregard if that choice put its opponent in a better position than itself. Our solution to our AI's indifference was to calculate the *difference* between the number of winning states for each player, meaning that the AI now cared about making the opponent's gameboard disfavorable. We noticed a significant increase in skill when the AI began trying to put its opponent in a worse position than itself.

Additional improvements to our function may have been achievable by having the AI prioritize creating contiguous lines of disks, denying opponent contiguous lines, or making already contiguous lines longer. Our recommendation for building a strong minimax search AI is to focus on making the heuristic evaluation function as robust as possible.