Amed Tentori

Koya Mitchell

CS 4242

04/15/2020

**Final Project Result Report**

**Introduction:**

For our project we decided to study the minimax algorithm and how it works. The minimax algorithm is a very commonly used algorithm used in decision making and game theory to find an optimal move for a player. There are two players, depicted as maximizer and minimizer. The maximizer tries to acquire the highest score possible and the minimizer tries to acquire the lowest score possible. For every board state the board has a type of score related to it. These scores are determined by some heuristic. In other words, the algorithm builds a tree to which each depth after the zero depths will be a depth of either maximizer or minimizer. From the initial state in depths zero, the tree expands to a certain depth that has the possibilities of the initial states possible moves it can make. This will continue with the children nodes of the initial state to a max depth. The idea is that the algorithm will backtrack from the terminal nodes of the tree, evaluate, and determine which child has either the lowest score (min) or highest score (max) between them. The algorithm will continue this until it tracks back to the parent node at depth zero and will decide which child node is the maximizer based on the score of the child node state. This node that is finally selected will be the best move that was determined by looking steps ahead of the game and determining which path of moves leads to the best possible move available.

Our question for this project was, “How effective and efficient is this algorithm?” So, we decided to test it out and see for ourselves firsthand. Using the game of Tic-Tac-Toe, we will create a program of the game using the minimax algorithm to select the “computers move.” We will then create another program using another way to select the computer's move and compare the programs. We will determine if the programs are making the appropriate moves and which is more consistent.

**Tic-Tac-Toe games:**

***TicTacToe.cpp:***

In this program we created artificial intelligence by using the minimax algorithm. As explained in the introduction we create a tree of all possible moves that can be made. All the board states are then evaluated to a heuristic. This heuristic helps determine the best score of the board state when the algorithm trackback from the terminal nodes back to the initial board to determine the best move. At each depth the tree selects either the max or min of the children nodes. Alpha beta pruning was used to help cut down on searching unnecessarily throughout the tree. We also made it so that the first move of the AI ‘X’ is random; this is done so that the two AI’s do not play the same game one hundred times. Below will be the minimax method that the program uses to help select a move.

***MiniMax method:***

// Apply the minimax game optimization algorithm

pair<int, pair<int, int>> MiniMax(char board[3][3], char symbol, int depth, int alpha, int beta)

{

// Initialize best move

pair<int, int> BestMove = make\_pair(-1, -1);

int BestScore = (symbol == CPU) ? LOSS : WIN;

// If we hit a terminal state (leaf node), return the best score and move

if (Full(board) || TIE != GetBoardState(board, CPU) || depth == 8)

{

BestScore = GetBoardState(board, CPU);

return make\_pair(BestScore, BestMove);

}

vector<pair<int, int>> legalmoves = GetLegalMoves(board);

for (int i = 0; i < legalmoves.size(); i++)

{

pair<int, int> curr\_move = legalmoves[i];

board[curr\_move.first][curr\_move.second] = symbol;

// Maximizing player's turn

if (symbol == CPU)

{

int score = MiniMax(board, USER, depth + 1, alpha, beta).first;

// Get the best scoring move

if (BestScore < score)

{

BestScore = score - depth \* 10;

BestMove = curr\_move;

// Check if this branch's best move is worse than the best

// option of a previously search branch. If it is, skip it

alpha = max(alpha, BestScore);

board[curr\_move.first][curr\_move.second] = BLANK;

if (beta <= alpha)

{

break;

}

}

} // Minimizing opponent's turn

else

{

int score = MiniMax(board, CPU, depth + 1, alpha, beta).first;

if (BestScore > score)

{

BestScore = score + depth \* 10;

BestMove = curr\_move;

// Check if this branch's best move is worse than the best

// option of a previously search branch. If it is, skip it

beta = min(beta, BestScore);

board[curr\_move.first][curr\_move.second] = BLANK;

if (beta <= alpha)

{

break;

}

}

}

board[curr\_move.first][curr\_move.second] = BLANK; // Undo move

}

return make\_pair(BestScore, BestMove);

}

***ManualGame.cpp:***

In this program we created artificial intelligence by manually creating the possible paths the computer can take based off of the gamestate of the board. If the computer goes first its first move is placed on a random tile to ensure more randomness. Whenever it is the computer’s turn after the first turn the computer will iterate through the if statements until it hits a statement that can be considered to not be an illegal move. It will repeat this process until the game is over resulting in either a win, loss or a tie between the player and computer. Below is a snippet of how the artificial intelligence of the computer works. When it is the computers turn it iterates through the if statements to see where the player has marked their tiles already. The computer then proceeds to make its next move the moment it hits an if statement where a move can be made. After the move is made playerTurn becomes true and it is now the player’s turn.

***Snippet of computer decision maker method:***

Board layout:

[(1,2,3),

(4,5,6),

(7,8,9)]

int Ai() //Ai decision maker

{

playerTurn == false;

if (One == 'O' && Two == 'O' && playerTurn == false && Three == '3')

{

Three = 'O';

playerTurn = true;

}

if (Four == 'O' && Five == 'O' && playerTurn == false && Six == '6')

{

Six = 'O';

playerTurn = true;

}

// continues for the rest of the possible path if there are two ‘O’s next to each other and it is not the players turn

if ((playerChoice == 1 || playerChoice == 5 || playerChoice == 9) && playerTurn == false) {

if ((One == 'X' && Five == 'X') && playerTurn == false && Nine == '9') //diagonal

{

Nine = 'O';

playerTurn = true;

}

if ((One == 'X' && Nine == 'X') && playerTurn == false && Five == '5') //diagonal

{

Five = 'O';

playerTurn = true;

}// continues until all possible game states are explored

***TTT.java:***

This program was the first used when i was thinking through ways to create a TIc-Tac-Toe game. So for this program the AI move is selected out of random. The AI makes a random move and is not allowed to take spots that are already occupied.

**Results:**

***Tic-Tac-Toe.cpp:***

I ran/tested the program one hundred times by playing the game one hundred times. Each and every time the two AI’s could only Draw. I then played the AI myself and had mixed results. I found that trying to win could only result in a draw but I found that I could let myself lose by making poor moves.

AI vs AI: (X vs O)

| Total Wins | Total Losses | Total Draws |
| --- | --- | --- |
| 0 | 0 | 100 |

AI vs User: (O vs X)

| Total Wins | Total Losses | Total Draws |
| --- | --- | --- |
| 13 | 0 | 87 |

***ManualGame.cpp:***

Player vs AI(X vsO):

Out of 100 games

| Win(against AI) | Loss(against AI) | Draw |
| --- | --- | --- |
| 10 | 35 | 55 |

Out of the one hundred games tested between a player and the computer the player won 10 times while the computer won 35 times. The rest of the games led to ties. I have noticed that if I was starting to gain an advantage over the computer the computer will always try to make the game into a draw. The only way I was able to beat the computer was to use the same specific path of placing the tiles while going first. If I use this method the player has a chance of winning every time going first. The path I took was bottom left, bottom right, then top right. Afterwards there were two choices left to place the marker to win: middle center tile or middle left tile.

***TTT.java:***

After testing this program I found the results to be completely random when It came to AI vs AI. As for the AI vs the User the AI was not making the clearest of moves. As the User could find east routes to win a game with no difficulties.

AI vs AI (X vs O):

| Total Wins | Total Losses | Total Draws |
| --- | --- | --- |
| 46 | 33 | 21 |

AI vs User(X vs O):

| Total Wins | Total Losses | Total Draws |
| --- | --- | --- |
| 12 | 77 | 11 |

**Conclusion:**

After testing and reviewing the programs we found that the results that came from TicTacToe.cpp, which uses the minimax algorithm to select a move, worked the best when it came to making a move in a game. This game of Tic-Tac-Toe was impossible to defeat. It was able to make both offensive and defensive moves. If there was a chance for the AI to win it would take it, while also being able to block the users chance of winning. We saw the AI versus another AI was only able to draw because they were looking enough moves ahead to determine which move would be the lowest chance of it lossing. The speed of the AI making a move did not seem to matter because they all produced a move pretty quickly and hard to say that one seemed faster than the other. Compared to the ManuelGame.cpp program the TicTacToe.cpp decision making was just smarter. For the ManualGame.cpp program the decision making was not calculated. It was only hard coded so the computer can only make decisions based on if the possible combinations of tile placements with turn order were coded in to begin with. Yes the ManuelGame.cpp program made obvious moves but it can also be tricked. We found a pattern that would allow you to win everytime going first versus this AI. Which isn't the worst result but it does show that the game is broken to an extent.

In conclusion, we saw that using the program using the minimax algorithm for this sort of decision making is very efficient. For Tic-Tac-Toe the AI was able to quickly select the best move. The algorithm allowed the AI to look several moves ahead and see which move would lead it to a winning state. If the AI saw that it was being placed in a position of losing it was able to defend against that if needed so. While the other methods for an AI did make a decision, it could not make a decision that acted more like an actual thought out move.