CECS 451

Tic Tac Toe with Alpha/Beta Pruning

Bich Tram Pham
Department of Computer Science
CSU Long Beach
Bich-Tram.Pham01@student.csulb.edu

Dimpal Shah
Department of Computer Science
CSU Long Beach
Dimpal.Shah@student.csulb.edu

Jocelyn Gonzalez
Department of Computer Science
CSU Long Beach
Jocelyn.Gonzalez03@student.csulb.edu

Jose Jimenez
Department of Computer Science
CSU Long Beach
Jose.Jimenez06@student.csulb.edu

Ruby Nguyen
Department of Computer Science
CSU Long Beach
Ruby.Nguyen01@student.csulb.edu

Abstract

Tic Tac Toe is a fairly common game in the field of Artificial Intelligence for its simulation in the process of gauging the performance of a computerized agent in finding the best strategies that maximize a goal. As such, we are interested in designing a computer program that can simulate this traditional game between a human and a computer and see for ourselves the behavior of the computerized agent in its attempt to win the game. Correspondingly, to create a decent agent, we will be using the Minimax implementation, a very suitable algorithm for game theory, along with Alpha/Beta Pruning algorithm which can pair really well with Minimax and help to improve the performance of the agent in a two-players game.

1 Introduction

Tic Tac Toe is a simple yet interesting two-players game where the human player and their opponent, an AI bot in this case, are initially presented with a 3 by 3 matrix board that is filled with empty spaces. The rules of the game are very straightforward. The human player and the bot will each take their turn and choose a position on the board to mark 'X' and 'O' respectively. The game will end when whoever gets their marks three in a row first, be it horizontally, vertically, or diagonally on the board; or when neither the human player nor the bot can achieve such a condition, which results as a draw.

Minimax Algorithm. A popular decision-making method in game theory and artificial intelligence is the minimax algorithm. Its goal is to assist a computer in deciding how to proceed in a game, given that the adversary is likewise playing effectively. The algorithm works by building a game tree that depicts every play that the opponent and the computer might possibly make. The edges of the tree represent potential moves, and each node represents a state of the game. The algorithm then assumes that both players will play optimally and recursively simulates the rest of the game, evaluating each conceivable move. At each level of the tree, it alternates between increasing the computer's score and decreasing the opponent's score; hence, the term "minimax".

Alpha/Beta Pruning. A search algorithm used in the minimax algorithm to reduce the number of nodes that need to be evaluated in a game tree, and thus speed up the Minimax's search by a significant factor. In the minimax algorithm, the ideal move is often determined by evaluating each node in the game tree. However, evaluating each node in the tree can be computationally expensive due to the fact that the number of nodes in the tree can increase exponentially with tree depth. Branches of the game tree that are unlikely to result in a better solution than the best one so far are pruned as part of the alpha-beta pruning process. It accomplishes this by keeping track of two values, **Alpha** and **Beta**, which stand for the best score each player has currently achieved, respectively maximizing player and minimizing player.

The algorithm can prune a branch of the tree if it discovers a move that results in a score that is lower than the best score at the moment since the opponent will never choose that move. Similar to this, if the algorithm discovers a move that results in a score higher than the best score currently available, it can stop evaluating additional moves in that branch because the opponent will never make a decision that results in a worse result. The alpha-beta pruning technique can significantly reduce the number of nodes that must be examined without impacting the outcome of the minimax algorithm by reducing branches of the game tree in this fashion.

2 Methods

We are building a Tic Tac Toe game with an AI bot using the minimax algorithm with alpha-beta pruning. The code defines a board with 9 available spaces, and creates a class named abp that includes functions to handle the game logic.

When the player or bot selects X or O, the gameBoard() function shows the board with those symbols in the appropriate locations. The checkingForWinner() function determines if a player or bot wins based on three consecutive numbers in a vertical, horizontal, or diagonal row using nested lists. The emptySpaces() function keeps track of how many empty spaces are still on the board.

The player() function handles input errors and prompts the user to choose a location on the board. The input error handling ensures that the user enters an integer from one to ten and does not select an occupied spot on the game board. If the user does commit an error, the program will notify the user of the specific error that was committed and re-prompt the user to select an open spot on the game board. The bot() function determines the best move for the bot using the minimax algorithm and alpha-beta pruning. The bot() function uses the abpmm() function to recursively search the game tree and choose the best course of action. The abpmm() function uses the Alpha/Beta pruning Minimax algorithm, which is applied to assist the bot in determining its move in the game. The function requires these five inputs:

1) **Board** - The game's current situation
2) **Depth** - The searchable depth of the game tree, this indicates how far the algorithm should look while deciding on its next move
3) **Alpha** - The maximizer's top score so far, the initial value being set at negative infinity
4) **Beta** - The minimizer's top score so far, the initial value being set at positive infinity
5) **Player** - Depending on whose turn it is, set to 1 (human player) or -1 (the bot)

The function starts out by determining if the current board has a winner, a score of 10 is returned if the maximizer won, a score of -10 is returned if the minimizer won, or a score of 0 is returned if it's a draw.

```
# checking which player won and returning the score accordingly
if abp.checkingForWinner(board, 1) is True:
    return [None, None, 10]
elif abp.checkingForWinner(board, -1) is True:
    return [None, None, -10]
#returns a score of zero if the game has reached maximum depth
elif depth == 0:
    return [None, None, 0]
```

Figure 1: First step of abpmm() function - checking if the current state has a winner

If no winner is found and the maximum depth has not been reached, the function repeatedly calls itself, switching between players who maximize and minimize, until the maximum depth is attained. The function examines each potential move and assigns a score to each move throughout each recursive call. The optimal move for the current player is then determined using the score. In the end, the function returns the best move and its corresponding score.

```
# Checks to see where the player can make a move and evaluates the resulting board
for cell in abp.emptySpaces(board):
    # putting the player on the board
    board[cell[0]][cell[1]] = player
    # recursively calling the abpmm function to check the board state
    score = abp.abpmm(board, depth - 1, alpha, beta, -player)
    # removing the player from the position on the board
    board[cell[0]][cell[1]] = 0
```

Figure 2: Second of abpmm() function - recursively call the abpmm() to reach maximum depth

```
# If the current player is maximizing, updating decisions to perfrom alpha beta pruining
# to find the best position the bot thinks the player is going to make
if player == 1:
    if score[2] > best_score:
        best_score = score[2]
        best_row = cell[0]
        best_col = cell[1]
        alpha = max(alpha, best_score)
    if beta <= alpha:
        break
# else the current player is minimizing, updating decisions to perfrom alpha beta pruining
# to find the best position the bot thinks it is going to make
else:
    if score[2] < best_score:
        best_score = score[2]
        best_row = cell[0]
        best_col = cell[1]
        beta = min(beta, best_score)
    if beta <= alpha:
        break
```

Figure 3: Alpha/Beta Pruning Algorith to predict the best move the bot could make

3 Results

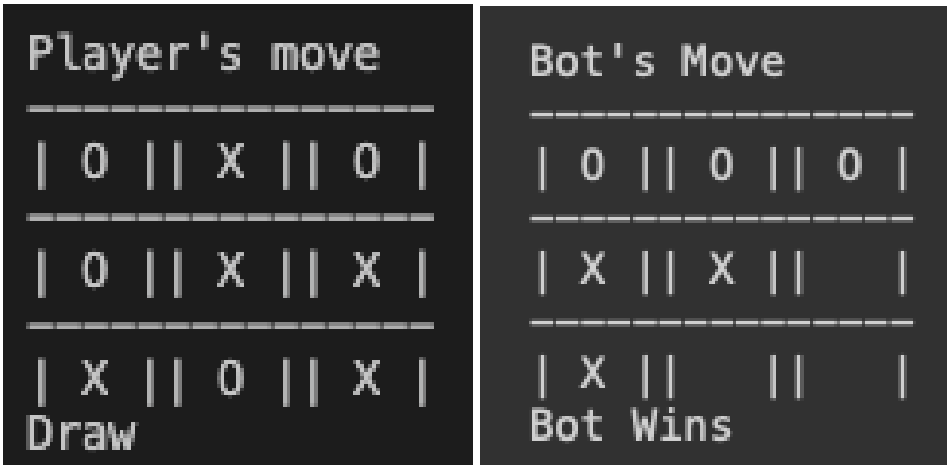

Figure 4: Outcome of Tic Tac Toe Program

	Human 1	Human 2	Human 3
Human Wins	0	0	0
Bot Wins	7	6	8
Draws	3	4	2
Total	10	10	10

Table 1: Results of 30 Test Runs of Tic Tac Toe Program

In the game of Tic Tac Toe, the outcomes of playing against a computer-controlled player utilizing the Alpha/Beta pruning algorithm are limited to either the computer's victory or a draw, as shown in Figure 1. After 30 test runs with three different human users, we saw the bot won 70% of the games, while the user had no wins. The results also concluded with 30% draws from the 30 test runs. The computer's ability to select the most optimal moves to win the game renders it virtually unbeatable, and any victory by a human player may indicate a flaw in the algorithm's implementation.

However, the effectiveness of the algorithm is contingent upon the game's design and the quality of its evaluation function. A poorly designed or overly complex evaluation function can impede the algorithm's ability to effectively prune the game tree, thereby causing a significant increase in computational time and resource usage.

Despite this, the use of the alpha beta pruning algorithm in Tic Tac Toe can enhance the overall performance of the computer-controlled player, resulting in a more challenging and engaging gameplay experience. Therefore, game developers and enthusiasts must carefully consider the game's design and evaluation function when implementing the algorithm to optimize its effectiveness.

4 Conclusion

This project creates a computer program that allows a user to play the classic game of Tic Tac Toe. To do so, we built a computerized agent, or bot, that plays against the user using two popular AI algorithms: minimax and alpha beta pruning. After the game board and user components were built, the minimax and alpha beta pruning algorithms were implemented in the abpmm() function that is embedded inside the bot component. The abpmm() function determines the optimal move for the computerized agent and executes it.

Through this project, we saw the effectiveness of the minimax algorithm against a human user and how the alpha beta pruning algorithm adds efficiency to the computer agent.

Reference

[1] B. T. Pham, D. Shah, J. Gonzalez, J. Jimenez, R. Nguyen. (2023, May 1). *CECS-451-Final-Project* [Online]. Available: https://github.com/rubynguyen2505/CECS451-Final-Project

[2] D. Higginbotham, "An exhaustive explanation of minimax, a staple AI algorithm," Flying Machine Studios, 10-Jan-2012. [Online]. Available: https://www.flyingmachinestudios.com/programming/minimax/#:~:text=The%20minimax%20algorithm%20is%20used,to%20see%20all%20possible%20moves. [Accessed: 01-May-2023].

[3] "Games," Introduction to Artificial Intelligence 2017. [Online]. Available: https://materiaalit.github.io/intro-to-ai-17/part2/. [Accessed: 01-May-2023].

[4] "Python oops concepts," *GeeksforGeeks*, 14-Jun-2022. [Online]. Available: https://www.geeksforgeeks.org/python-oops-concepts. [Accessed: 30-Apr-2023].

[5] R. D. Vishwakarma, C. E. Gudumotu. (2023, Feb. 28). *CECS451-ISA/Lab_02_28_2023/alpha_beta_pruning* [Online]. Available: https://github.com/rahvis/CECS451-ISA/tree/main/Lab_02_28_2023/alpha_beta_pruning