Collaborative Deep Dive on Game Trees

Berwin Lan, Sree Chalimadugu, Wesley Soo-Hoo

July 11, 2023

1 Introduction

Our team was interested in learning about game trees and exploring the tree algorithms for a game like tic-tac-toe, which involves two players. We wanted to delve into some of the main algorithmic components of game theory, such as the Minimax algorithm, which aims to maximize the minimum gain, or minimize the possible loss for a worst case, maximum loss scenario. This is very prevalent in game theory, which is why we were interested in learning about the Minimax algorithm for this collaborative deep dive.

After we learned about how the Minimax algorithm works to help a player make the ideal move in tic-tac-toe, we wanted to implement some optimizations to our Minimax algorithm through alpha/beta pruning. This helped us expand our knowledge beyond the baseline algorithm and see what kinds of strategies are used to better optimize and improve an algorithm in game theory.

2 Preliminaries

A *tree* is a connected undirected graph with no simple circuits. A specific type of tree that we are concerned with in this deep dive is a *binary tree*, which is a tree where every node in it has at most 2 children, which we can designate as a right or left child [Rosen, 2003].

In combinatorial game theory, a *game tree* is a type of tree that represents the sequence of a two-player zero-sum game flow, with all possible game states in a particular game – in our case, a two-player game of tic-tac-toe. The vertices in a game tree represent the states that a game can be in as it progresses, and the edges represent the possible legal moves between the states (vertices) [Rosen, 2003].

3 Game Trees

3.1 Algorithms

In game theory, there are several algorithms that are widely used in order to optimize the decision made by a player. We focused on the Minimax algorithm and its optimizations, which we will explain in the following sections.

3.2 Minimax Algorithm

The Minimax algorithm represents the two players as a maximizer and minimizer, in which the maximizer tries to score as high as possible, and the minimizer tries to score as low as possible [codecademy, 2022]. At the core, the goal of Minimax is to find the optimal move for a player

and "maximize the minimum gain" – in other words, assuming that the opponent tries to minimize your gains, the algorithm will try to maximize your gains based on the opponent's move, which is assumed to be played optimally [for Geeks, 2022a]. This assumption is based off of a worst-case scenario, where if the opponent does not play optimally, the problem is simplified.

With the Minimax algorithm, the game of tic-tac-toe can be represented by a game tree, with the current game state and the possible moves to be made are represented. A helpful representation of tic-tac-toe as a game tree is shown in Figure 1. The *leaf nodes*, which are the nodes without any children, represent the end conditions, which are the wins or losses. When the Minimax algorithm is run, we can evaluate a game state even if it is not a leaf, and rather at an intermediate state in a game, by either taking the minimum or maximum child node on each layer.

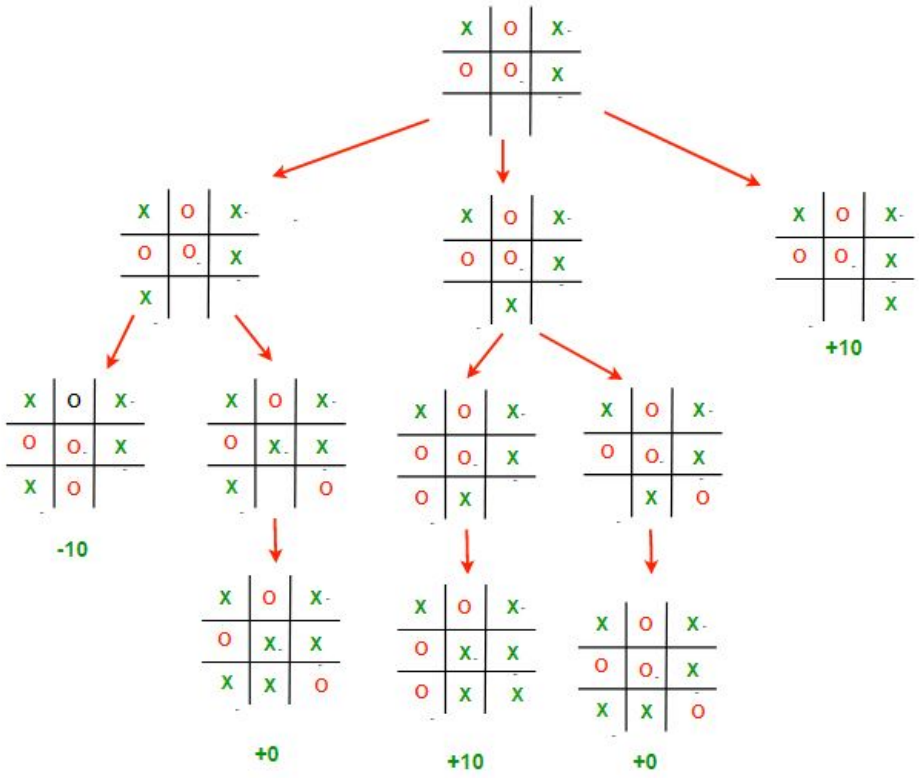

Figure 1: Game tree representation of a game of tic-tac-toe [for Geeks, 2022b]

The Minimax algorithm is a backtracking technique, where it tries all possible moves, and then backtracks recursively to come to a decision. So, when the computer plays as the maximizer, it traverses the entire tree in a depth-first search, alternating between the maximizer and the minimizer on each layer to find the maximum value possible at the top level. At each level, the computer chooses the minimum or maximum node, based on if it is the maximizer or minimizer at that level. At the end, this means that the top level decision (the current turn) will choose the best result if both players play optimally.

3.3 Alpha-Beta Pruning

If we want to make our existing algorithm more efficient, we can implement alpha-beta pruning, which is an improvement to the Minimax algorithm. This optimization allows us to cut off any branches that don't change the decision making or outcome of the game.

Alpha-beta pruning involves two variables – alpha and beta – and we track them to decide when to "prune" a part of the tree. Alpha represents the best (largest) score seen so far for the maximizer player, and beta represents the best (smallest) score for the minimizer player. We initially set alpha as negative infinity and beta as positive infinity to start each player with their worst possible score, after which the variables are updated as the game progresses [Krivokuća, 2022].

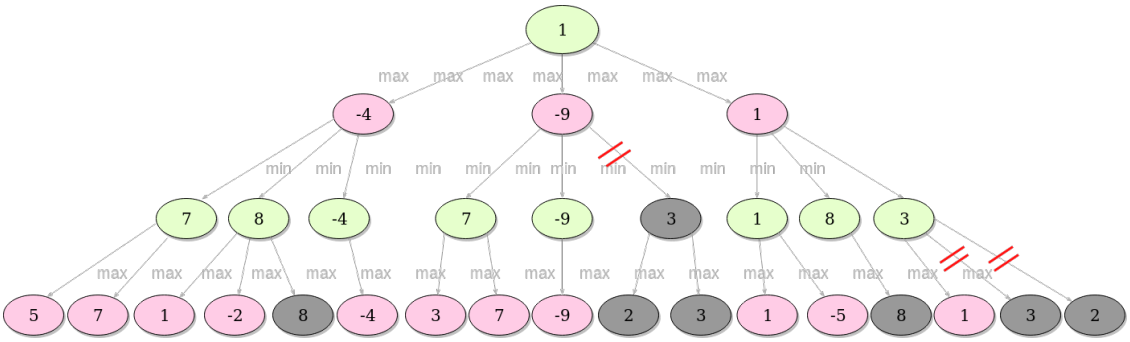

Figure 2: On each step, the alpha-beta pruning algorithm checks whether the next move is worse than the current best move. If so, all the children of the next move are cut off, and that branch is not evaluated down to the leaves.

We will use Figure 2 as a sample game tree to explain how alpha-beta pruning works. In this tree, the edges below green nodes represent layers where the maximizer is playing, and the edges below pink nodes represent layers where the minimizer is playing. Gray nodes are where leaf nodes to the right on the same layer can be pruned, and the pruning itself is represented by the two red parallel lines on the pruned edges.

As an example, consider the minimizer playing on the leftmost pink node with value -4. It will search each of its child nodes left-to-right, starting with the green node of value 7. That node's value is determined by taking the maximum of its children (5 and 7) since that layer of edges is a maximizer according to the Minimax algorithm. Since 7 is the smallest value encountered thus far, beta is set to 7, representing the best possible score. Then, we backtrack and consider the green node to the right of the green node of value 7. Its value is not known yet, but we know that it will have the maximum value of any of its children. Then, we search its child nodes from left-to-right, so we look at 1, -2, and 8 in that order. When we encounter 8, we note that it is greater than the green 7 node one layer closer to the root. Therefore, because the parent of the gray node has the value of the maximum child seen so far, its value at this time is 8, which is greater than the current value of beta. Regardless of what values the nodes right of the gray node have, none of them will change the decision or outcome of the game:

- If the value is less than 7 (the current value of beta), then the maximizer layer will ignore those values because they are less than the current value of beta.

- If the value is greater than 8 (which beta will be set to, from the gray node), it is also greater than the green node with value 7. Either way, the value of its green parent node will be greater than 7 in the green layer. When the minimizer selects the minimum value from the nodes in the green layer, it will not select the node greater than 7 no matter how *much* it is bigger than 7.

Thus, we can prune off all the nodes that we *would* have looked at to the right of 8 (in this case, there are none).

A more interesting case is the one in the middle, which we will explain briefly using the same principles as above. We start as the maximizer playing in a game state represented by the root node with value 1. We explore the leftmost branch using a depth-first search to calculate the leftmost pink child node's value of -4, set alpha $= -4$, and then move to do the same for the second pink node. We consider that node's children (once again with a depth-first search), and the node's value will be the minimum of its children's values. The first node's value is 7 because the maximum value of its children is 7, and the pink node's temporary value is also 7 because it is the minimum value seen so far. Then, we calculate the value of the next green node by looking at the maximum value of *its* child, which is -9 (it only has one child). We compare that to the current minimum value among the green child nodes, 7, and it is smaller, so the parent pink node's value is set to -9. Compared to the current alpha value of -4 from the other pink node, -9 is smaller. Because that is a maximizing layer, the node of value -9 will not be chosen when there is a bigger node (the one with -4). Thus, we can ignore any nodes to the right of the green -9, like the gray 3 and its children, for the same reasons as the previous example.

- If the value is more than -9, it will be ignored by the minimizer on that layer because it is not less than -9.
- If the value is less than -9, then the pink node will be set to its value. However, the pink node will then be ignored by the maximizer layer because -4 is greater.

Because it doesn't matter what comes after a node that is less than alpha or greater than beta, we can prune all unsearched nodes to its right with the same branch. In a larger search space with greater depth and breadth than tic-tac-toe, this ends up saving a lot of time, which we further discuss in the next section.

With the alpha-beta pruning method, we are able to significantly reduce the amount of time it takes for the Minimax algorithm to execute, as we are able to eliminate entire branches of the search tree, and rather focus the search time to the remaining sub-tree.

3.4 Runtime comparison between algorithms

We have claimed that alpha-beta pruning significantly improves the runtime of the Minimax algorithm, so we ran each algorithm 5 times and took the average runtime of each computer move. The results are shown in Figure 3, and we can see that alpha-beta pruning drastically improves the

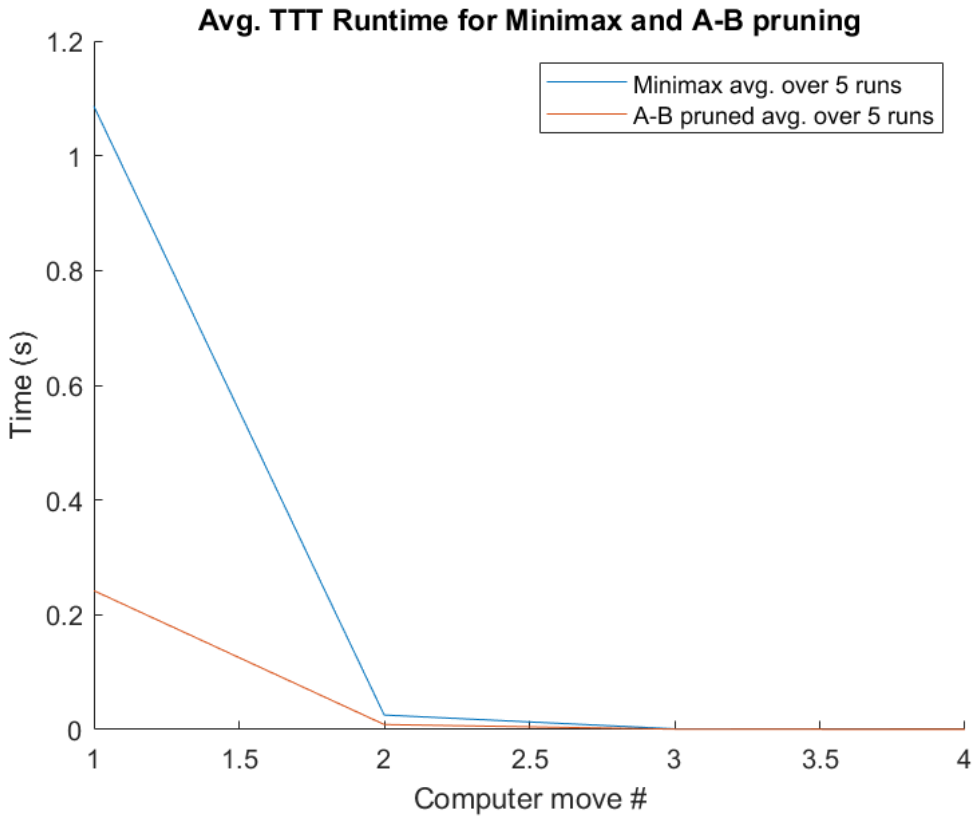

Figure 3: The alpha-beta pruned Minimax algorithm is significantly faster than the original Minimax implementation, especially at earlier stages of the game.

runtime of the computer in a game of tic-tac-toe. This is most dramatic at the beginning, where the alpha-beta pruning resulted in 3-4x faster decision making. As expected, the difference between the two algorithms' runtimes decreases as the board fills in and there are fewer possible branches in the tree to check.

3.5 Other Considerations

If the size of a game tree is too large, it is difficult to visit all of the states in a short amount of time, so oftentimes, a maximum recursion depth can be specified with a heuristic algorithm to estimate the value of a node without traversing all the way to its leaf. Heuristic algorithms use traditional knowledge about a game to make an educated guess about which player has the advantage [Roberts, 2003]. For example, in a game of chess, one simple heuristic can be the material difference between the two players, as a player with a queen would likely be ahead of a player with a rook.

Caching can also be used to optimize the Minimax algorithm. In our code, the `minimax()` function gets called recursively at every node in the tree. While the tree must be traversed entirely the first time around, the results can be cached so that the second time the function is run, instead of re-traversing the entire tree, each node's score can be pulled from the cache. This will dramatically

decrease the runtime of the Minimax function starting from the second turn in the algorithm. This does not affect the tic-tac-toe game much, as after the first turn, the tree is already relatively small, but in a much more complex game, such as chess, the game tree is significantly deep at every turn [for Geeks, 2022b].

References

[codecademy, 2022] codecademy (2022). Artificial intelligence decision making: Minimax.

[for Geeks, 2022a] for Geeks, G. (2022a). Minimax algorithm in game theory (introduction).

[for Geeks, 2022b] for Geeks, G. (2022b). Minimax algorithm in game theory (tic-tac-toe ai – finding optimal move).

[Krivokuća, 2022] Krivokuća, M. (2022). Minimax with alpha-beta pruning in python.

[Roberts, 2003] Roberts (2003). Strategies and tactics for intelligent search.

[Rosen, 2003] Rosen, K. H. (2003). *Discrete Mathematics and its Applications*. McGraw Hill, fifth edition.

4 Appendix: minimax.py

```
# Helper functions for running the Minimax algorithm

import play, boardhelpers

# Computer is +1 Maximizer
def minimax(board, recursion_depth=0):
    """
    Returns:
        tuple (next_move, next_score)
            next_move: Two-ple with (row: int, col: int)
            next_score: an int representing the score of the next move to be played
    """
    # Setting initial conditions for players
    if recursion_depth % 2 == 0:
        # max player
        current_player = 1
    else:
        # min player
        current_player = -1

    # get all possible moves: recursion

    # Base case: winner / all moves played
    if boardhelpers.eval_board(board) != 0 or boardhelpers.count_turns(board) == 9:
        return ((None, None), (boardhelpers.eval_board(board) * 10) / boardhelpers.count_turns(board))

    # Otherwise, determine next move
    moves = play.get_possible_moves(board) # gets all possible places to put the next move
    next_scores = {}  # dict where k:v is move:score
    for move in moves:
        try:
            next_board = play.make_move(board, move, current_player)
        except Exception:
            continue
        _, next_scores[move] = minimax(next_board, recursion_depth + 1)

    # Using next_scores, pick the min or max move
    if current_player == 1:  # MAX player
        next_move = max(next_scores, key=next_scores.get)
        next_score = next_scores[next_move]
    else:    # MIN player
        next_move = min(next_scores, key=next_scores.get)
        next_score = next_scores[next_move]
    return (next_move, next_score)

def computer_turn(board):
    next_move, next_score = minimax(board)
    if next_move[0] is None:
        return board, True

    next_board = play.make_move(board, next_move, 1)
```

```
    return next_board, boardhelpers.eval_board(next_board)
```

5 Appendix: abpruning.py

```
# Alpha-beta pruning
import boardhelpers, play, math

# Computer is +1 Maximizer
def minimax_alpha_beta(board, recursion_depth=0, alpha=-math.inf, beta=math.inf):
    """
    Returns:
        tuple (next_move, next_score)
            next_move: Two-ple with (row: int, col: int)
            next_score: an int representing the score of the next move to be played
    """
    # Setting initial conditions for players
    if recursion_depth % 2 == 0:
        # max player
        current_player = 1
    else:
        # min player
        current_player = -1

    # get all possible moves: recursion
    # Base case: winner / all moves played
    if boardhelpers.eval_board(board) != 0 or boardhelpers.count_turns(board) == 9:
        return ((None, None), (boardhelpers.eval_board(board) * 10) / boardhelpers.count_turns(board))

    # Otherwise, determine next move
    moves = play.get_possible_moves(board) # gets all possible places to put the next move
    next_scores = {}  # dict where k:v is move:score
    for move in moves:
        try:
            next_board = play.make_move(board, move, current_player)
        except Exception:
            continue
        _, score = minimax_alpha_beta(next_board, recursion_depth + 1, alpha, beta)
        next_scores[move] = score
        if current_player == 1:
            alpha = max(score, alpha)
            if score > beta:
                break
        else:
            beta = min(score, beta)
            if score < alpha:
                break

    # Using next_scores, pick the min or max move
    if current_player == 1:  # MAX player
        next_move = max(next_scores, key=next_scores.get)
        next_score = next_scores[next_move]
    else:    # MIN player
        next_move = min(next_scores, key=next_scores.get)
        next_score = next_scores[next_move]
    return (next_move, next_score)
```

```
def computer_turn_alpha_beta(board):
    next_move, next_score = minimax_alpha_beta(board)
    if next_move[0] is None:
        return board, True

    next_board = play.make_move(board, next_move, 1)
    return next_board, boardhelpers.eval_board(next_board)
```

6 Appendix: boardhelpers.py

```
# Helper functions for the game board.

# This function evaluates the state of the current board and returns a score.
# Returns 0 if no one won, or the number associated with the player who won
def eval_board(board):
    # Check if horizontal is won
    for row in board:
        if sum(row) == 3:
            return 1
        elif sum(row) == -3:
            return -1

    # check if vertical is won
    for i in range(len(board[0])):
        if sum([row[i] for row in board]) == 3:
            return 1
        elif sum([row[i] for row in board]) == -3:
            return -1

    # check diagonals
    if sum(board[i][i] for i in range(3)) == 3:
        return 1
    elif sum(board[i][i] for i in range(3)) == -3:
        return -1

    if sum(board[2-i][i] for i in range(3)) == 3:
        return 1
    elif sum(board[2-i][i] for i in range(3)) == -3:
        return -1

    return 0

def draw_board(board):
    print("\n")
    print("\n-----\n".join(["|".join([("X" if cell == 1 else ("O" if cell == -1 else " ")) for cell in r
    print("\n")

# Counts the number of turns that have already gone
def count_turns(board):
    return sum(sum([abs(i) for i in row]) for row in board)

# moves are defined as the square that you will place your marker in as a (row, col) tuple
def get_possible_moves(board):
    moves = []
    for row in range(len(board)):
        for col in range(len(board[0])):
            if board[row][col] == 0:
                moves.append((row, col))
    return moves
```

7 Appendix: play.py

```
# Helper functions for running the game.
from copy import deepcopy
import boardhelpers

# moves are defined as the square that you will place your marker in as a (row, col) tuple
def get_possible_moves(board):
    moves = []
    for row in range(len(board)):
        for col in range(len(board[0])):
            if board[row][col] == 0:
                moves.append((row, col))
    return moves

def make_move(board, move, player):
    # Returns the board after the move
    row, col = move
    new_board = deepcopy(board)
    if new_board[row][col] != 0:
        raise Exception(f"Trying to overwrite a filled square. board[{row}][{col}] = {board[row][col]}")
    new_board[row][col] = player
    return new_board

def player_turn(board):
    row = int(input("Row: "))
    col = int(input("Col: "))

    next_board = make_move(board, (row, col), -1)
    over = (boardhelpers.eval_board(next_board) != 0)
    return next_board, over
```

8 Appendix: main.py

```
import minimax, abpruning, boardhelpers, play, time

def main():

    algorithms:dict = {"1": minimax.computer_turn,
                       "2": abpruning.computer_turn_alpha_beta}

    gametype = algorithms[input("Enter 1 for Minimax, 2 for alpha-beta pruning: ")]

    board = [
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]

    over = False

    while boardhelpers.eval_board(board) == 0 and boardhelpers.count_turns(board) < 9:
        if over:
            break

        boardhelpers.draw_board(board)
        board, over = play.player_turn(board)

        if over:
            break

        boardhelpers.draw_board(board)
        start = time.time()
        board, over = gametype(board)
        end = time.time()
        print(f"Evaluation time: {round(end - start, 6)} sec.")

    boardhelpers.draw_board(board)
    print("Game over")


if __name__ == '__main__':
    main()
```