Analysis of Minimax and Alpha Beta Pruning Algorithms used on the game Gobblet

Varun Nandu
Northeastern University
nandu.v@husky.neu.edu

Abstract

Artificial Intelligence is used to generate intelligent behaviors primarily in non-player characteristics, often simulating human like intelligence. It is an important and growing field of study, with practical applications ranging from search to speech recognition. One good way to study AI is through games. In this project I have used different decision algorithms like Minimax and Alpha-Beta Pruning on the game Gobblet. The performance of these algorithms have been analyzed by using different depths limits to be searched for decision making.

I. INTRODUCTION

Gobblet is a board game for two players. This game is a complex version of Tic Tac Toe game. The game uses a 4x4 board and the goal is to create a consecutive 4 pegs in a row, column or diagonal.

Rules for the game are:

• Each player has 12 pegs in 4 different sizes ordered in 3 stacks. The player can place on the board only the top peg from each stack.

• A player can move pegs that are already on the board in his turn.

• A player can cover his opponent peg with a bigger peg. While a peg is covered it cannot be moved.

• It is not allowed to cover an opponent peg with a new peg (only with the pegs that are currently on the board), unless this move will block a winning move.

• A player may uncover his opponent pegs in his turn but if a row, column or diagonal is created for his opponent by this move, the player lost the game.

In this project, the game will be played between a player (Human) and computer. The difficulty of the game can be set by the user at the start of the game. The challenge here is, the computer is required to use intelligence against each move by the player considering the difficulty chosen by the player and come up with an optimal move. The moves will be determined based on the algorithm being used. The game is deterministic, turn-taking, two player, zero-sum game. The state of this game can be properly represented and the agents are restricted to certain number of actions whose outcomes can be defined by rules.

Since each square in this game can consist of player one's token, player two's token or empty with each player token having upto four sizes. Thus each square can contain a maximum of 4 stack of pieces. This gives us a crude upper bound of 2^{64} possible positions. Thus this is an adversarial search problem.

II. ADVERSERIAL SEARCH

An adversarial (game) search problem can be defined as a kind of search problem with the following elements:

• S0: The initial state, which specifies how the game is set up at the start.

• PLAYER(s): Defines which player has the move in a state.

• ACTIONS(s): Returns the set of legal moves in a state.

• RESULT(s, a): The transition model, which defines the result of a move.

• TERMINAL-TEST(s): A terminal test, which is true when the game is over and false

• TERMINAL STATES otherwise. States where the game has ended are called terminal states.

• UTILITY(s, p): A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal states for a player p.

The initial state, ACTIONS function, and RESULT function define the game tree which is a tree where the nodes are game states and the edges are moves. In Gobblet there are two players. We will call player 1 as Max and player 2 as Min. At every move Max will try to improve his chances of winning whereas Min will try to improve his chances of winning thus reducing the Max's chances of winning. We can also say that Max will think of a move that increases his chances of winning while Min will think of a move that decreases the chances of Max winning. To compute the next move of a player, we will use algorithms which will aid in decision making by considering the opponents future moves that will be in turn based on the players future moves and so on, using the utility function. These algorithms are Minimax and

Alpha-Beta Pruning.

III. MINIMAX ALGORITHM

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds. The minimax algorithm performs a complete depth-first exploration of the game tree. The basic algorithm for minimax is as follows:

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        bestValue := −∞
        for each child of node
            v := minimax(child, depth − 1, FALSE)
            bestValue := max(bestValue, v)
        return bestValue
    else (* minimizing player *)
        bestValue := +∞
        for each child of node
            v := minimax(child, depth − 1, TRUE)
            bestValue := min(bestValue, v)
        return bestValue
```

Since we can't search the entire tree, we only consider the first few levels. In this manner we pretend that the nodes below a certain level are leaves, and determine their value using the analysis function.

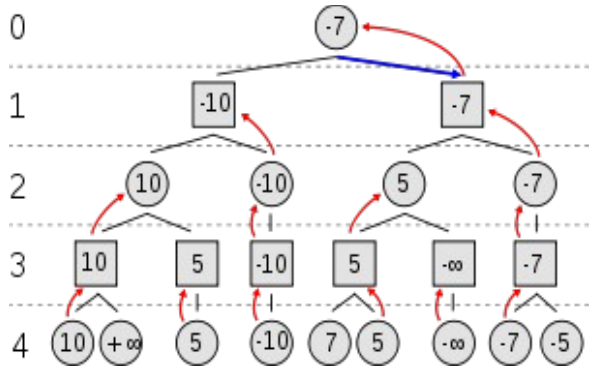


Figure 1: Minimax Algorithm search tree for two players

If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O(bm).

IV. ALPHA-BETA PRUNING

Minimax algorithms is Complete Search algorithm. This means they'll search the full space (at least the one applicable to the current state) trying every possible play to find the best one. The Alpha-Beta Prune is a clever Branch and Bound algorithm that prunes branches of the game tree that are guaranteed to be worse than the current best. Thus it seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the 'more promising' subtree and a deeper search can be performed in the same time. The alpha-beta pruning algorithm is as follows:

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        v := -∞
        for each child of node
            v := max(v, alphabeta(child, depth – 1, α, β,
                        FALSE))
            α := max(α, v)
            if β ≤ α
                break (* β cut-off *)
        return v
    else
        v := ∞
        for each child of node
            v := min(v, alphabeta(child, depth – 1, α,
                        β,TRUE))
            β := min(β, v)
            if β ≤ α
                break (* α cut-off *)
        return v
```

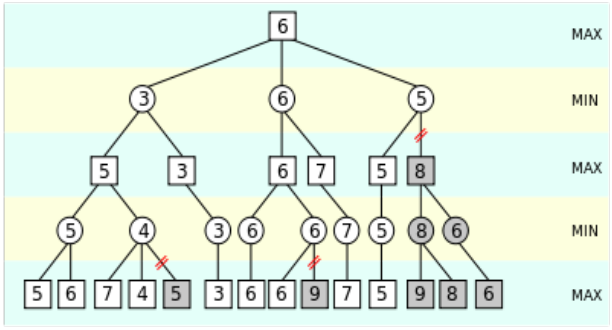


Figure 2: Alpha-beta pruning search tree. The shaded branches are the pruned as they won't contribute to the final result.

The algorithm maintains two values, alpha and beta, which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of respectively. Initially alpha is negative infinity and beta is positive infinity, i.e. both players start with their lowest possible score. It can happen that when choosing a certain branch of a certain node the minimum score that the minimizing player is assured of becomes less than the maximum score that the maximizing player is assured of (beta <= alpha). If this is the case, the parent node should not choose this node, because it will make the score for the parent node worse. Therefore, the other branches of the node do not have to be explored.

V. IMPLEMENTATION

For this project the code for Gobblet game has been written in Python 2.7. The game begins by asking for the difficulty of the opponent (computer) player wherein the other player is human. The difficulty level decide the depth up to which the algorithm searches (0 = Easy, 1 = Medium, 2 = Hard and 3 = Expert). The more the depth the more plies the algorithm checks for. The code considers 1 depth as 1 move of each player. Thus, 2 depth means it considers the future possible moves of each player twice. The game then informs that Human player plays first. There are two operations available for Human Player. He can either insert a new piece or update the position of an existing piece. Valid insert and update operations are checked using insertCheck and updateCheck functions respectively. The player who plays first has guaranteed chance of winning if it is a perfect player or say is the optimal player. The game then asks for the algorithm to be used by the computer to compute its next move. The algorithms are minimax and alpha-beta pruning algorithms. All of the algorithms used in this project make use of an evaluation function. The evaluation function is used to calculate the how good the player position is set at the current position of the game. The function would go through the positions of each player on the board and would give a certain score for the current state. The positions that may affect the calculation of score are numerous. The most common and the most essential check is if at the current state any of the player is

winning. The other important check is if there are any threats. For example, if there are three tokens of the same player in a row and both the ends are empty and if that player is an optimal player, he is guaranteed to win in the next move. Along with these there are few more main checks. Those are :

- Four In a Row (1000000)
- Three in a row (50000)
- Two in a row (10000)
- Higher Size Pieces (5000)

The evaluation function scores the value for a particular state by considering the sum of all these scores for a player and the sum of the scores for the opponent score and returning the difference. Also more score has been given to the center positions so that the center positions are chosen over the edges which have more chances of converting to a win.

VI. RESULTS

The algorithms specified above have been used to calculate the time taken for the computer when the computer human plays first. Time calculated is in seconds. The test board on which these tests are performed is given below:

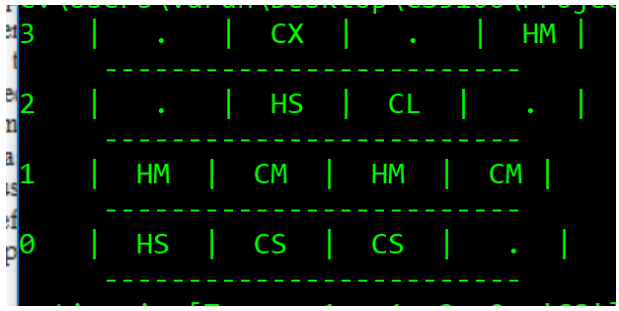


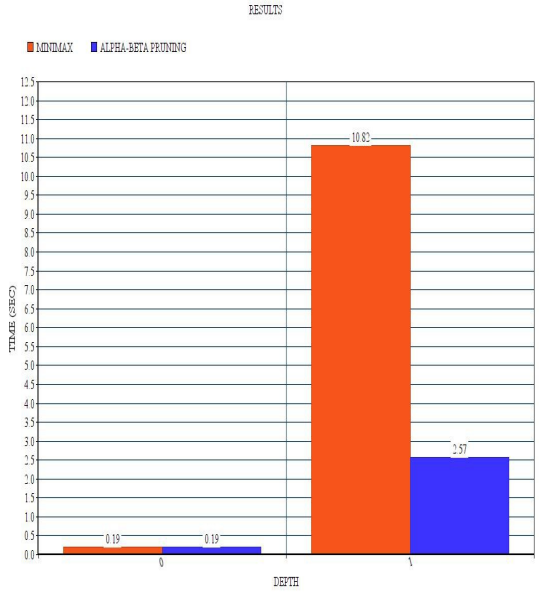
Figure 3: Test Board

These time are noted for each level of difficulty chosen. The time taken at each difficulty level is shown in table as follows:

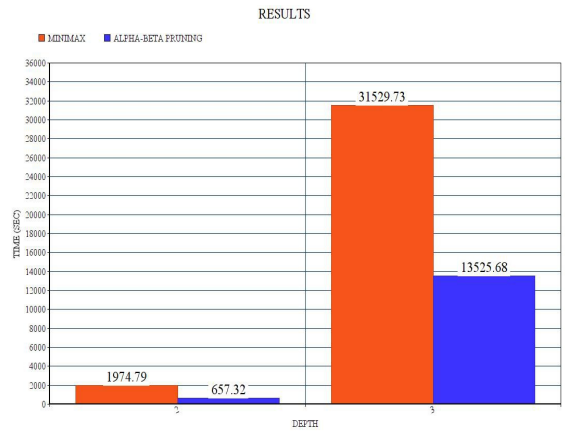
DEPTH	ALGORITHMS	
	MINIMAX ALGORITHM	ALPHA-BETA PRUNING
0	0.19	0.19
1	10.82	2.57
2	1974.79	657.32
3	31529.73	13525.68

Table 1: Test Results

The graph for the same is given below:



Graph 1: Results for Depth 0 and 1



Graph 2: Results for Depth 2 and 3

As one can see, the time taken at easy difficulty level (depth = 0) is the same for Minimax as well as Alpha-Beta Pruning Algorithm because it only evaluates the very next move of the opponent and does not consider the opponents opponent move. Thus calculation is considerably reduced. However one interesting observation is that Alpha-Beta Pruning performs way better than Minimax algorithm as the depth goes on increasing. Other interesting observation is that as the depth increases, the time taken by computer to make a move increases exponentially. This is because the state space is too large and a mere depth of 2 requires computer to take at least 10 min to make a move in the faster Alpha-Beta Pruning Algorithm.

VII. CONCLUSION

The use of minimax and alpha-beta pruning algorithms for Gobblet produced near optimal results. Use of proper general rules helped choose a better position in the given state and use of good evaluation functions proved to be useful in getting near optimal results for the computer player. We found that both the algorithms performance decreased equally when the depth of the search was increased to obtain better results. This is because in the worst case when you can insert in every square and update every square from every square, the maximum branching factor available was 304. Thus every time the depth is increased the time increases by a factor of 304. This calculation has to be performed for computer as well as human every step. Both the algorithms resulted in same moves being returned for a given depth. However, alpha-beta pruning resulted in better performance over the Minimax algorithm and ratio of the performance increased when the depth of the search increased.

VIII. FUTURE WORK

For the future I would like to better the evaluation function so that it helps pruning the branches that won't help in the final result. Also because of the large state space I feel like another approach like reinforcement learning is better suited to solving this game. With enough training exploration of such a huge state space wont be needed and computer might be able to play optimal moves which are even 10 steps ahead of the opponent.

IX. REFRENCES

- Russell, S., Norvig, P., & Intelligence, A. (2010). A modern approach. Artificial Intelligence. Prentice-Hall, 161.
- https://en.wikipedia.org/wiki/Gobblet for game information
- https://boardgamegeek.com/thread/170389/gobblet-strategy for game strategies
- http://www.blueorangegames.com/components/com_games/files/games_rules/gobblet_rules.pdf gobblet rules from original gobblet site