Comparing minimax and its variations to solve checkers

CS7IS2 Project (2020-2021)

Prabhjot Singh, Prabhjot Kaur, Karan Bhardwaj

singhp1@tcd.ie, kaurp@tcd.ie, bhardwak@tcd.ie

Abstract. Minimax algorithm is the most commonly used AI technique to solve board games which is based on DFS search on game tree. variations of minimax are generally employed to improve the efficiency of AI in terms of the time taken or computations performed to make a move. These variations don't necessarily optimize the choice of move made by the algorithm, but aim to reduce computational complexity and hence, enabling AI algorithms to work with deeper game trees in resource constrained machines. This project aims to provide a comparison of variations of Minimax while keeping minimax algorithm as baseline.

1 Introduction

Game solving using artificial intelligence have consistently been one of the most interesting topics of discussion in AI since the computers were discovered. Milestones of artificial intelligence performing better than actual humans have always been an intrigue as well as a measure of thresholds in the field of AI.
In this project we aimed to explore methods used to solve board games. Although everyone in our team being a beginner in the field, we specifically wanted to pick a problem with moderate complexity and explore the methods available to solve it. This would provide us the foundation for the learning we expected from CS7IS2.

We chose to investigate checkers because of the following reasons:

- ➢ Every piece in checkers has same privileges, this reduces a lot of functional complexity.
- ➢ Every piece will have between 0 to 4 possible moves, this reduces complexity in terms of computations involved, and makes it possible for us to implement the solutions even in resource-constrained environments.

This project started with a research around two classes of algorithms Minimax and Proof number Search (PN). but as PN algorithms are based on searching a node to the win, it did not pose an optimal solution for game play along with being resource and computationally intensive. We narrowed the scope of our project to study variations of minimax and comparing their performance to present a meaningful study of comparisons between minimax, which is our baseline and three common variations of minimax that apply optimization to its performance, namely 1) alpha beta pruning, randomized alpha beta and 3) NegaScout.

We present an overview of this report in recorded presentation in this OneDrive link.

2 Related Work

Board games are one of the oldest branches of Artificial Intelligence (Shannon and Turing in 1950). These games represent a pure and quite abstract form of competition involved in 2 players which requires a form of "intelligence". In such games, states can be represented easily, and each possible action is well defined. Implementing AI on such problems are effective way to understand and evaluate AI based algorithms, as all the individual states are fully accessible. As characteristics of the opponent are unknown thus it is a contingency problem. Search trees that are required to solve board games can become astronomically large.
Algorithms used in board games should have following properties:

1. use good evaluation functions for in-between
2. look ahead as many moves as possible
3. delete irrelevant branches of the game tree, states

In the paper "**General Board Game Playing for Education and Research in Generic AI Game Learning**" by Wolfgang Konen Computer Science Institute TH Koln – Cologne University of Applied Sciences ¨ Gummersbach, Germany, importance of GBG(general board game) study and research work is highlighted in order to understand and analyse AI algorithms. This paper helped us to understand the basics of developing GPG with AI and gave insights to the algorithm such as MinMax. The games on which the AI is evaluated in this paper is 2048 and Hex gameboard. This paper helps researchers in game learning to quickly test their new ideas or to examine how well their AI agents generalize on a large variety of board games.
A Comparative Study of Game Tree Searching Methods
In 2014, a journal ((IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 5, No. 5, 2014) published on Egypt highlight the Comparisons between Game Tree Searching Methods. In complex Board problems many algorithms have been discussed to find the next-move such sequential algorithms such as MiniMax, NegaMax, Negascout, SSS* and B* as well as parallel algorithms such as Parallel Alpha-Beta algorithm. Almost all the board games use game tree with the following components-

- Each node represents a game state.
- The root represents the current game state.
- All the branches for a given node represent all the legal moves for that node.

Evaluation function determines score for the path taken to the specified branch. There exist a big set of methods for sequential game tree. The paper has underlined sequential game tree algorithms categorized into depth first and breadth search algorithm.
This paper gave the example of Tic-Tac-Toe game with MInMax , NegaMax, Alpha-Beta, NegaScout Algorithm. These algorithms are different in the terms of Moving orders. These all techniques are categorized under Brute- Force algorithms in Depth- First Search. MinMax tries to evaluate the best move for AI. It maximizes the AI Score along with taking care to minimize human score. Other mentioned algorithms are enhanced in some ways and limited in other ways in order to increase efficiency and decrease computation level of MinMax.
Important Difference between the brute-force algorithms and the selectivity algorithms is that Selectivity algorithms in Depth-First Search doesn't depend on fixed depth to stop looking in each branch. The most common techniques in this category are Quiescence Search and Forward Pruning. Quiescence Search based on the idea of variable depth searching. The algorithm follows the normal fixed depth in most branches and pruning cut of the irrelevant computations. Many algorithms implemented the idea of this technique, including N-Best Selective Search, ProbCut and Multi-ProCut.
Parallelism in Game Tree search implementation requires multiprocessors and multi core computers. There are number of algorithms categorised in this section such as A, Parallel Alpha-Beta, Parallel PVS, YBWC and Jamboree and Dynamic Tree Splitting. Some of the algorithms are same as sequence algorithms such as alpha beta but is the parallel version.
After discussion of various algorithms in the paper which includes sequential and parallel algorithms, we chose to continue our deeper dive with sequential algorithms as parallel algorithms require better CPU and GPU resources for parallel computations. Inside sequential algorithms narrowed our search to Brute- Force algorithms in Depth- First Search. We chose to evaluate and compare these algorithms against each other with respect to various components such as time, number of steps, depth etc.

Previous work related to Checkers:
We found a paper submitted by **Elmer R. Escandon´ ∗, Student Member and Joseph Campion∗ (IEEE Members).** The paper highlights the success of AI against average human player on Checkers Board game. They compared the simple MinMax algorithm with varying depths. Their studies proved that AI was successful with the depth of 5-6 but struggled hard with the depth of 3-4. They also compared the time with respect to the depth of the model. the efficiency of the AI was evaluated in terms of run time to complete the minimax algorithm and success of the AI against human beings.
The results based in this paper shows the exponential rise of time with respect to the depth of MinMax algorithm and Succes rate direct proportionality with respect to depth.
In this paper we chose MinMax algorithm as our baseline and compared other sequential brute force algorithms- NegaScout, AlphaBeta, Randomized AlphaBeta.

In the paper “Rminimax: An Optimally Randomized MINIMAX Algorithm by Silvia García Díez, Jérôme Laforge, and Marco Saerens”,importance of Randomized Alpha beta is highlighted. This algorithm has proved to be an essential component of board games as behaviour of AI can be turned predictable for regular player. To avoid the predictability there is a requirement of randomization element in AI model. Thus, it is a crucial part to evaluate the success rate of Randomized Algorithm. In this paper we have applied randomization on AlphaBeta algorithm.

3 Problem Definition and Algorithm

The aim of this project is to compare the best suitable alternatives to solve the game of checkers.
We use minimax as our baseline and three algorithms, namely, 1) Minimax with alpha-beta pruning, 2) Minimax with randomized alpha-beta pruning 3) NegaScout, as our models of choice to achieve this aim.

In the rest of this section, first, we explain our problem statement, and then the baseline AI and AI solutions of choice to the problem statement, which are compared in later sections. Python programming language is used to set up the environment and implement our baseline AI and our AI solutions of choice.

Game of Checkers
Checkers is a two-player board game with simple set of rules described as following:
➢ Checkers is played on 8x8 board with alternative dark and light blocks, with a dark set and a light set of pieces for the opposite players. Pieces are placed on dark blocks.
➢ normally, pieces can only move one step diagonally in forward direction. Once, a piece reaches the end (opposite side) of the board, the piece becomes a king piece and can move diagonally backward. Pieces always remain on dark blocks in board.
➢ Pieces can skip over opposite player’s pieces and while doing so, the pieces skipped are thrown out of game. The player who loses all the pieces first loses the game and the other player is declared winner.

To model our problem statement, we use environment set up in (4).

Baseline: Minimax
Minimax algorithm is one of the widely used methods to solve board games. It is based on Depth first search on *game tree,* (tree with the edge representing a possible move and node representing a state of the game). The minimax algorithm assumes that the opposite player always plays the move most favourable to them and based on this assumption, algorithm returns the best move for the current player.
This is done by alternatively maximizing and minimizing the evaluation of the game’s state as seen by AI’s player by considering possible moves at every stage and picking the best possible for the current (AI’s) player (maximizing stage) and opposite player (minimizing stage).

For example, Fig 1 is a simple example of how minimax decides the move based on search on game tree. Root node represents the current position of the AI's player, A and B are the possible moves, then the children of MIN nodes (leaf nodes) hold the evaluations at those positions. MIN node will choose the child node with minimum possible value, which is the best possible move for MIN player and MAX node will chose the max value, hence, in this example, the move chosen by the minimax algorithm will be A.
Algorithm 1 presents the pseudocode for minimax algorithm.

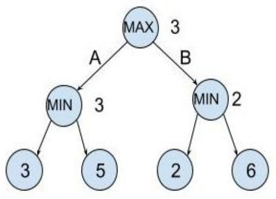

Figure 1 minimax in game tree

```
function minimax(position, depth, max_player)
    if depth is 0 and winner is None:
        return evaluation_at_this_position

    if max_player:                      //Maximizing
        maxEval -> -infinity
        best_move -> None
        for move in all_possible_moves:
            evaluation -> minimax(move, depth-1, False)
            maxEval -> max(maxEval, evaluation)
            if maxEval is evaluation:
                best_move -> move

    else:                               //Minimizing
        minEval -> +infinity
        best_move -> None
        for move in all_possible_moves:
            evaluation -> minimax(move, depth-1, True)
            minEval -> min(minEval, evaluation)
            if minEval is evaluation:
                best_move -> move
```

Algorithm 1 Pseudocode for minimax

The number of nodes searched grow exponentially with the depth when minimax algorithm is implemented in actual games as the moves possible at each MIN or MAX stage will have to be explored at every stage.

Minimax with alpha beta pruning

Alpha beta pruning is a way to optimize minimax algorithm. The optimization is based on the fact that not all the nodes need to be searched to get min and max at every layer of the game tree.
Alpha is calculated as the best possible value at maximize stage and beta is calculated as the best possible value at minimize stage. The current alpha and beta values of a node are passed to the child nodes and pruning is done based on these values. In maximizing stage, if a child node with value greater than beta is discovered, the rest of the child nodes are pruned (or the maximizing is cut off) as the best possible value has already succeeded the upper bound set by beta passed from parent. In case of minimizing stage, the cut off happens when a child node with value smaller than alpha is discovered.

Figure 2 presents a slice example of how alpha beta pruning works. Let us consider a game tree 2 a), initially alpha and beta values are unknown for every node. As we explore the first child of MIN node, we can calculate alpha to be 7 as it is the best possible value for it, this is represented in 2 b). In Fig 2 c), the calculated alpha becomes parent (MIN) node's beta value as it is the best possible value discovered so far. This value is passed to the next child node (Fig 2 d)), when the child (MAX) node discovers the value 9, which is greater than it's beta value, it cuts off the search and the MIN can now go on to discover other children or return the beta value (which is also it's final value) to the parent node if all the child nodes have been explored.

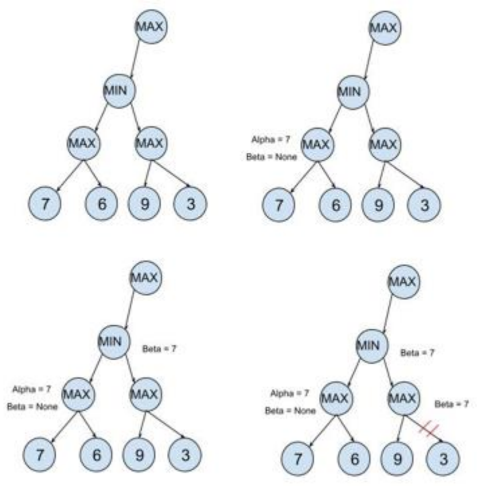

Figure 2 minimax with alpha beta pruning, a) to d) from top left to bottom right

Algorithm 2 presents pseudocode for minimax algorithm with alpha beta pruning. Alpha beta pruning is expected to greatly reduce the time of search by reducing the nodes explored. The worst case of minimax with alpha beta pruning is the actual minimax algorithm.

```
function alphaBeta(position; depth; onTurn; alpha; beta)
    if depth = 0 or endOfGame(position) then         -> leaf node
        return evaluate(position)
    end if
    bestValue = -inf - 1
    for all m in generateMoves(position; onTurn) do: . Try all moves
        value = alphaBeta(executeMove(m), depth-1, opponent(onTurn),
         -beta, -alpha)
        if value > bestV alue then
            bestValue = value
         end if
         if value > alpha then
             alpha =  value
         end if
         if alpha > beta then                  -> Pruning (cutoff)
             break
         end if
     end for
     return bestValue
 end function
```

Algorithm 2 Minimax with alpha beta pruning

Minimax with randomized alpha beta

Randomized alpha beta aims to cut down on the repetitions in the game by introducing randomness. The choice of best values at maximize and minimize stage is hindered by introducing margin to alpha and beta values. This means the move chosen is not always the best move, so the algorithm although can prove effective against a human player who can predict the repetitions, it might not compete as well with other AIs that aim to play the best move every time.

Algorithm 3 presents pseudocode for randomized alpha beta.

```
function RandomizedAlphaBeta(position; depth; onTurn)
    bestValue = -inf
    allMoves  = []
    for all m in generateMoves(position; onTurn) do:
        value   = alphaBeta(executeMove(m); depth - 1;
         opponent(onTurn);-inf;-bestValue + margin + 1)
        Add (m, value) to the list allMoves
        if value > bestValue then
            bestValue = value
        end if
     end for
     goodEnough  = select moves with value >= bestValue - margin from allMoves
     return uniformly random move from goodEnough
 end function
```

Algorithm 3 Pseudocode for randomized alpha beta

Randomized alpha beta is expected to have a performance very similar to alpha beta algorithm.

NegaScout

NegaScout algorithm is a variation of minmax with alpha beta pruning that restricts the search by setting beta temporarily to alpha + 1. If no move is found, the search is done again without the restriction.
Algorithm 4 presents pseudocode for NegaScout algorithm

```
function negaScout(position; depth; onTurn; alpha; beta)
    if depth = 0 or endOfGame(position) then
        return evaluate(position)
    end if
    bestValue  = -inf - 1
    beta2 = beta
    for all m in generateMoves(position; onT urn) do: . Try all moves
        value  = -negaScout(executeMove(m), depth-1, opponent(onTurn),-beta2, -alpha)
    if value > alpha and value < beta and m is not the first move then
         value  = -negaScout(executeMove(m), depth  1, opponent(onTurn), -beta, -alpha)
     end if
     if value > bestValue then
         bestValue =  value
     end if
     if value > alpha then
         alpha  = value
     end if
     if alpha > beta then
         break
     end if
     beta2 = alpha + 1
     end for
     return bestValue
 end function
```

Algorithm 4 Pseudocode for NegaScout algorithm

NegaScout's performance is expected to be worse than alpha beta algorithm for most cases due to the added iteration to search for the perfect value but it is still expected to outperform minimax algorithm.

4 Experimental Results

AverageTime vs Depth
X axis: depth in algorithm applied **Y axis**: Average Time Taken by AI player per single move
Evaluation vs Depth
X axis: depth in algorithm applied **Y axis**: Average number of evaulations performed by AI player per single move

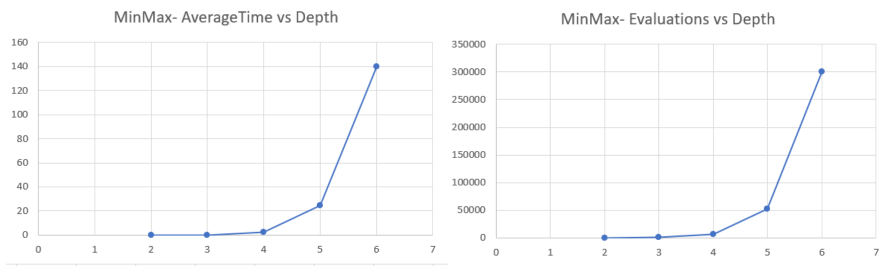

Figure 3 Average time and average numbner of evaluations per move vs Depth for Minimax

Figure 3 reveals the exponential curve function of Time and Computations performed with respect to depth for the Minimax algorithm. It can be seen from the above graphs that after depth of four minmax AI player is of no use for implementation purpose as there is a lot of time involved for a single move.

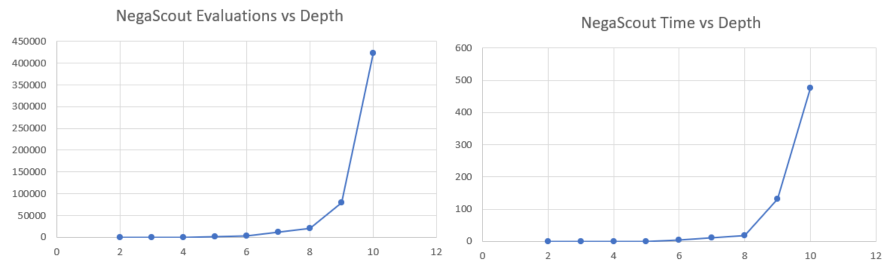

Figure 4 Average time and average numbner of evaluations per move vs Depth for NegaScout

In Figure 4, High rise in efficiency can be visualized when NegaScout algorithm is employed. This algorithm is able to perform better in the terms of time as it responds within decent time slot till the depth of 8.

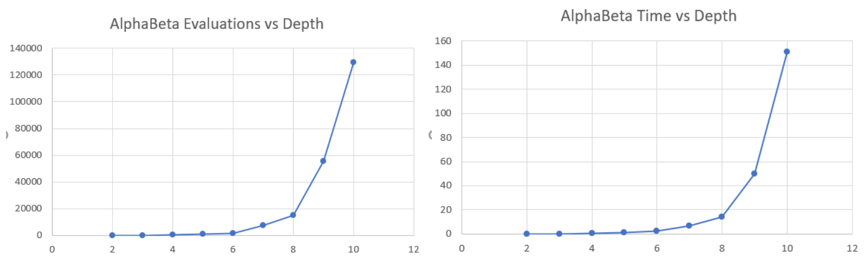

Figure 5 Average time and average numbner of evaluations per move vs Depth for Minimax with AlphaBeta pruning

Comparisons in Figure 5 reveal AlphaBeta's much greater efficiency as compared with Minimax Algorithm. It performs much better than Minimax and bit better than NegaScout (scale-y axis of NegaScout algorithm is smaller than AlphaBeta)

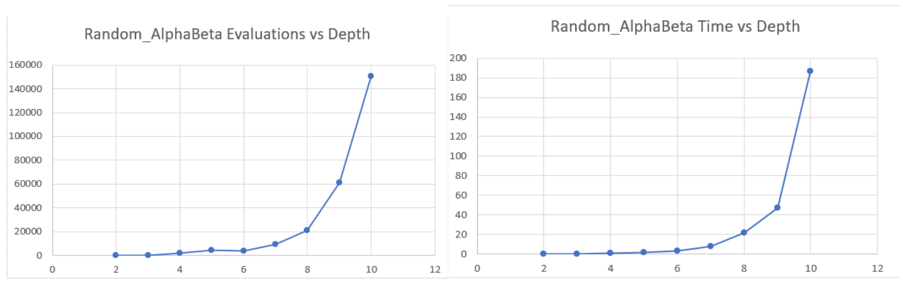

Figure 6 Average time and average numbner of evaluations per move vs Depth for Randomized AlphaBeta

Comparisons in Figure 6 show Random Alpha Beta's performance which is similar to Alpha Beta's performance in the terms of time and computation processes.

AI's against each other:
Based on the experimental analysis all the AIs performs equilaterally well in competition with each other. MinMax, AlphaBeta, NegaScout are very competitive on the same level as they all are based on Brute Search sequential algorithm. Random AlphaBeta AI's performance also proved to be very good in competing with other AI's.

5 Conclusions

This project aims to find the best way to approach game solving with AI in checkers by implementing and comparing three variations of Minimax algorithm while taking Minimax as the baseline, which are 1) Minimax with alpha beta pruning, 2) Minimax with randomized alpha beta pruning, 3) Minimax with NegaScout.
These algorithms are expected to have similar results in term of choice of the move given the depth is same, with exception of randomized alpha beta as it trades off the best choice of move to remove repetitions as repetitive moves can decrease the effectiveness of an AI against human players.

The comparisons are done based on two metrics, 1) Evaluations performed per move, 2) time taken to make a move by AI. Our comparisons reveal that all the optimizations perform better than minimax. Furthermore, randomized alpha beta and Minimax with alpha beta perform very similar as expected by our earlier analysis in section 3. Similarly, as per our expectations, NegaScout performs slightly worse than alpha beta or randomized alpha beta but still performs very much better than our baseline minimax algorithm.

References

1. A Comparative Study of Game Tree Searching Methods by Ahmed A. Elnaggar, Mostafa Abdel Aziem, Mahmoud Gadallah, Hesham El-Deeb ((IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 5, No. 5, 2014)
2. General Board Game Playing for Education and Research in Generic AI Game Learning by Wolfgang Konen
3. Minimax Checkers Playing GUI: A Foundation for AI Applications Elmer R. Escandon´ ∗ , Student Member and Joseph Campion∗ , Member, IEEE
4. https://www.youtube.com/watch?v=mYbrH1Cl3nw