Table of Contents

ABSTRACT

An 8 * 8 matrix represents the game of Chess. It is a recreational and competitive board game played between two players. It is an abstract strategy game that required strategy and tactics to win the game. Computers can play this game using Game Theory. Various algorithms used in Game of Chess are Minmax, Alpha-Beta Pruning, Null Move Heuristic, Quiescence Searching, static board evaluation function, genetic algorithms. Neural Network can also be used in playing Game of Chess.
Game theory is a branch of Mathematics, where a player's action is explained mathematically, and these actions result from some other player action. Games are modelled as Adversarial search and heuristic function, which helps solve games via AI.

Adversarial Search is a search algorithm where two players play against each other, and both try to nullify the other's action. Different opponent search for a solution in the same search space. Each opponent will try to plan ahead in time.

A heuristic is a technique that tries to solve the problem faster than any classical methods. It tries to find an approximate optimal solution when classical methods cannot. It evaluates the information at each level and figures out the next branch to follow.

Adversarial Search and heuristic function utilize Game tree. A Game tree is a tree where each node represents a game state, and each edge represents player move.
Any algorithm performance is measured in four ways:

- **Completeness:** Guaranteed solution, exist or not.
- **Optimality:** Does the algorithm can find an optimal solution.
- **Time Complexity:** Time taken by an algorithm to find a solution.
- **Space Complexity:** Space taken by algorithm while finding a solution.

INTRODUCTION

Chess in a quite old game and royal likely to came from Western world of India through Persia in 6th AD Century. It is a symbol of Intelligence since very beginning.

Claude Shannon in 1950 published a first paper 'Programming a computer for playing Chess'. It was the first paper in its league. Then Alan Turing became the first person to develop a program to play Chess, based on paper in 1951. Then many other joined this league Dietrich Prinz in 1952, Paul Stein and Mark wells in 1956, John McCarthy in 1956. In 1966 first chess match was played between computer programs. In 1988, Deep Thought shares a first place in Software Toolworks Championship.

What is Chess?

Chess is a competitive and recreational board game played between two players. It is played on 8 by 8 chess board using an abstract strategy. One player plays with black piece and another player will play using white piece. There are total of 16 pieces with each player, and it includes 8 pawns, 2 Rooks, 2 Knights, 2 bishops, 1 king and 1 Queen. Player will win the game, if the opponent King is captured. Game can also end in draw state i.e., nobody wins. Game can be played using various Game Theory Algorithm. Artificial Intelligence in a one of the implementations of Game Theory.

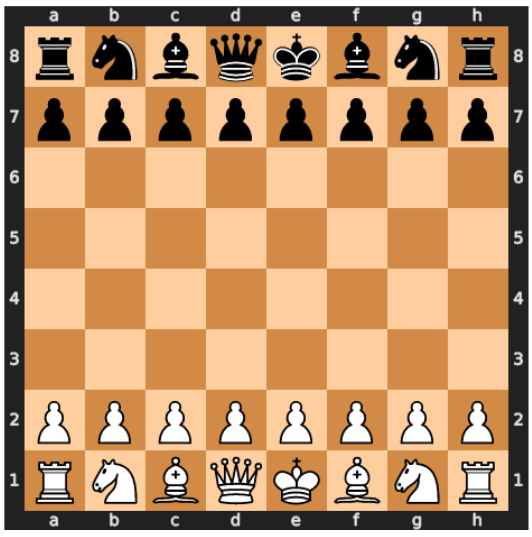


Fig. 1-1 Chess Board Representation

What is Game Theory?

Game theory is a branch of mathematics used to model the strategic interaction between different rational agents in a context with predefined rules and outcome. It can be divided into 5 categories:

- Cooperative and Non-Cooperative Games
- Symmetric and Asymmetric Games
- Perfect and Imperfect Game
- Simultaneous and Sequential Games
- Zero-Sum and Non-Zero-Sum Games

Different aspect of Game Theory can be used in Artificial Intelligence.

What is Artificial Intelligence?

Intelligence in Machine is known as Artificial Intelligence. This technology is trying to provide machines ability to behave like Human. The scientist has been able to develop software that helps Machine perform some mundane tasks. Nowadays, Robots can clean the floor, serve food in hotels, make coffee, order groceries based on user inputs. Some masterful jobs like robots help doctors during operation, lift and move bulky goods in industry, and make products in the factory production line. Additionally, the scientist has been developing software that can beat humans in games, solve the complex puzzle in significantly less time. Though machines are still very far from behaving like humans, if we consider the technology advancement curve, one can easily say that soon machines will be able to.

What is an Evaluation function?

Evaluation functions are crucial piece in Algorithm. They are heuristic function that determines relative value of position. It takes chess board state as an input and return the estimate about who

is winning and by how much. To write evaluation function, we need to be aware of numeric value of each pieces in board. Ideally, a bishop and knight are worth three pawn, a rook is worth five pawn, and queen is worth nine pawn. These relations won't be applicable in real game. We have to consider the piece position and other piece position.

DIFFERENT ALGORITHMS USED IN CHESS

In this paper we will analyze the different algorithm used in Chess. We will learn mathematical modelling, steps involved, Time and Space Complexity analysis, Completeness and Optimality of an algorithm.

Minmax algorithm

The core of Chess playing Chess in minmax. Minmax usually associates Black piece with MAX, and white piece with MIN, and always evaluates from the white point of view.

The Minmax algorithm is an Adversarial Search algorithm in Game theory. It utilizes game tree and includes two player MIN and MAX. Both players try to nullify the action of other. Max tries to maximize the result whereas MIN tries to minimize the result. Both players play alternatively, under the assumption that both are playing optimally. Optimal play means both players are playing as per rule i.e., MIN is minimizing the result and MAX is maximizing the result.

The minmax algorithm utilizes Depth First Search approach to find the result. Additionally, it also utilizes backtracking and recursion. Algorithm will traverse till terminal node and then it will backtrack while comparing all child values. It will select the minimum or maximum value, based on whose turn it is. It will then propagate the value back to their parent.

It uses static evaluation function to determine the value at each leaf node. It takes the advantage of Zero-Sum Game.

Pseudo Code

```
function minmax (node, depth, is_max_player)
        if depth is 0 or node is a leaf node then ## This is the base case
                return static evaluation of node.
        if is_max_player then
            max_val = -infinity
           for each child of node do
                node_val = minmax (child, depth -1, false) ## This is the recursion
                max_val = max (max_val, node_val) ## here we are backtracking
                undo the last move
            return max_val.
        else
            min_val = infinity
            for each child of node do
                node_val = minmax (child, depth -1, false) ## This is the recursion
                min_val = max (min_val, node_val) ## here we are backtracking
                undo the last move

            return min_val.
```

Python Implementation

Use this link to check full implementation.

```python
def minmax( board_instance, max_depth, current_depth, is_max_player, nodes_per_depth ):

    # This if else code block is only used for analysis of algorithm, by counting number of nodes explored
    if max_depth - current_depth in nodes_per_depth:
        nodes_per_depth[max_depth - current_depth] += 1
    else:
        nodes_per_depth[max_depth - current_depth] = 1

    # This is the base case, depth == 0 means it is a leaf node
    if current_depth == 0:
        leaf_node_score = static_eval(board_instance)
        return (leaf_node_score, nodes_per_depth)

    if is_max_player:

        # set absurdly high negative value such that none of the static evaluation result less than this value
        best_score = -100000

        for legal_move in board_instance.legal_moves:
            move =  chess.Move.from_uci(str(legal_move))

            # pushing the current move to the board
            board_instance.push(move)


            #calculating node score, if the current node will be the leaf node, then score will be calculated by static evaluation;
            #score will be calculated by finding max value between node score and current best score.
            node_score, nodes_per_depth = minmax(board_instance, max_depth, current_depth - 1, False, nodes_per_depth)

            # calculating the max value for the particular node
            best_score = max(best_score, node_score)

            # undoing the last move, so that we can evaluate next legal moves
            board_instance.pop()

        return (best_score, nodes_per_depth)
    else:

        # set absurdly high positive value such that none of the static evaluation result more than this value
        best_score = 100000

        for legal_move in board_instance.legal_moves:
            move =  chess.Move.from_uci(str(legal_move))

            # pushing the current move to the board
            board_instance.push(move)


            # calculating node score, if the current node will be the leaf node, then score will be calculated by static evaluation;
            # score will be calculated by finding min value between node score and current best score.
            node_score, nodes_per_depth = minmax(board_instance, max_depth, current_depth - 1, True, nodes_per_depth)

            # calculating the min value for the particular node
            best_score = min(best_score, node_score)

            # undoing the last move, so that we can evaluate next legal moves
            board_instance.pop()

        return (best_score, nodes_per_depth)
```

Fig. 1-2 minmax implementation in python

Game Play Analysis

In this section, we will allow two minmax algorithm to compete against each other. We will calculate the total time taken by each move and number of nodes explored per depth. We will only explore till depth = 3 because of system restriction. We will allow 3 move each for Black and White piece.

WHITE Turn

- **Move in UCI format:** b1c3
- **Nodes per depth:** {0: 20, 1: 400, 2: 8902, 3: 197281}
- **Time taken by Move:** 27.045262575149536

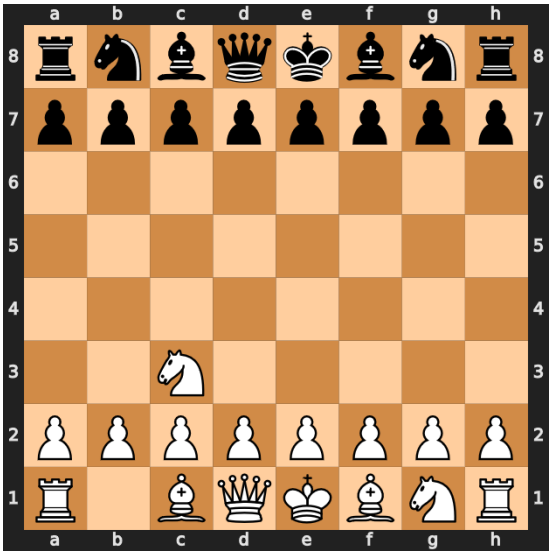


Fig. 1-3 minmax move 1

BLACK Turn

- **Move in UCI format:** c7c6
- **Nodes per depth:** {0: 20, 1: 440, 2: 9755, 3: 234656}
- **Time taken by Move:** 33.26433849334717

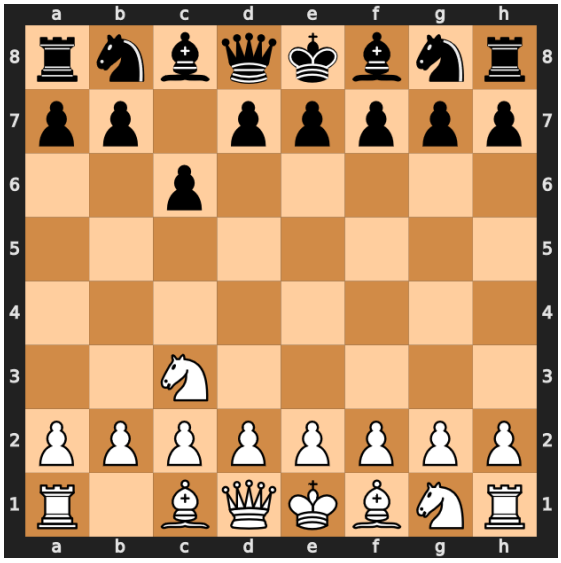


Fig. 1-4 minmax move 2

WHITE Turn

- **Move in UCI format:** a1b1
- **Nodes per depth:** {0: 22, 1: 462, 2: 11116, 3: 264316}
- **Time taken by Move:** 36.86292481422424

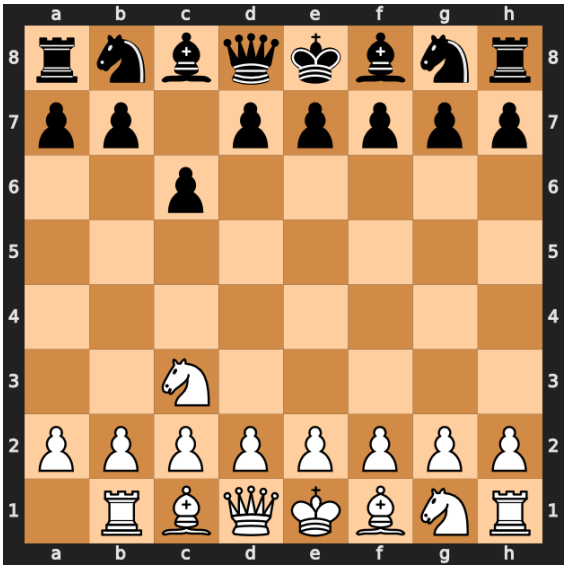


Fig. 1-5 minmax move 3

BLACK Turn

- **Move in UCI format:** g8h6
- **Nodes per depth:** {0: 21, 1: 441, 2: 10648, 3: 249005}
- **Time taken by Move:** 33.524622201919556

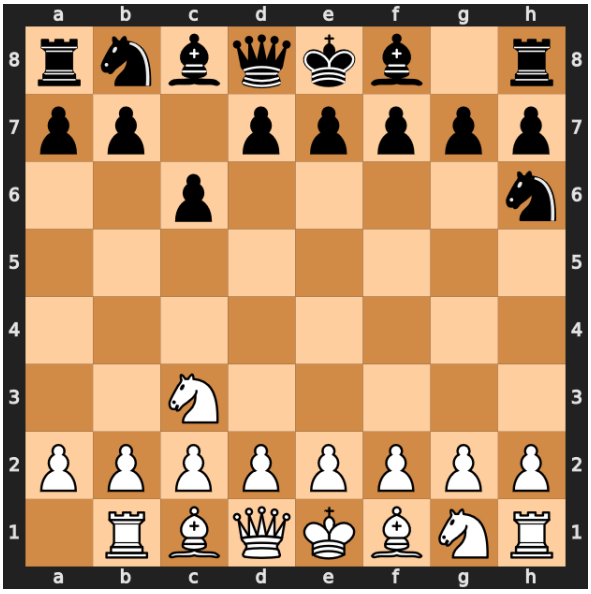


Fig. 1-6 minmax move 4

WHITE Turn

- **Move in UCI format:** c3e4
- **Nodes per depth:** {0: 21, 1: 441, 2: 10363, 3: 247970}
- **Time taken by Move:** 34.17363452911377

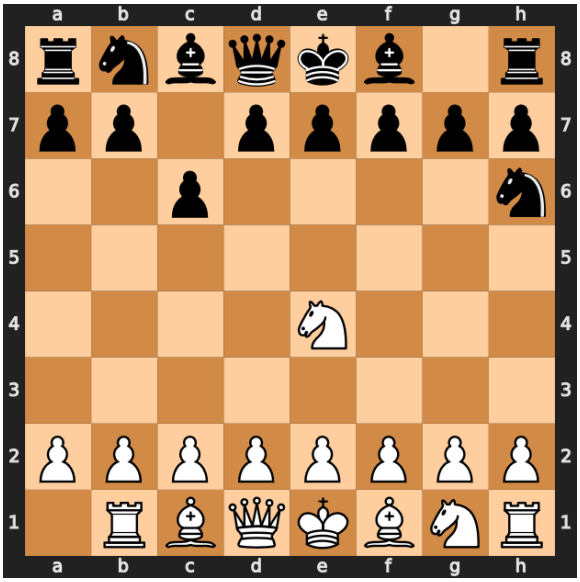


Fig. 1-7 minmax move 5

BLACK Turn

- **Move in UCI format:** d8c7
- **Nodes per depth:** {0: 21, 1: 501, 2: 11217, 3: 279541}
- **Time taken by Move:** 38.26315665245056

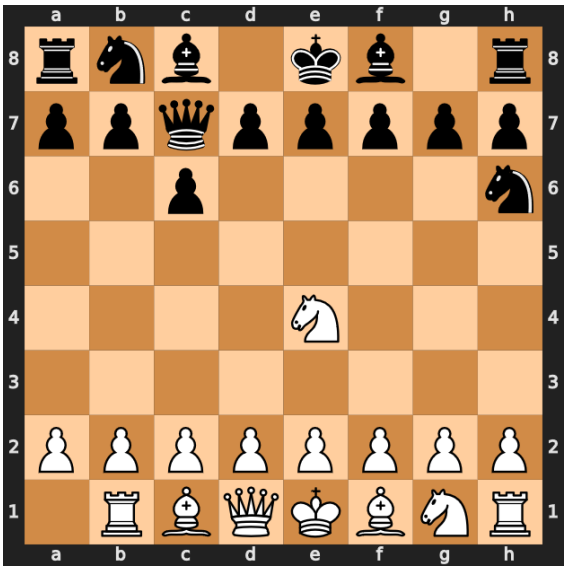


Fig. 1-8 minmax move 6

It is visible from above experiment following can be inferred:

- Average Branching Factor: 20
- Average time taken by algorithm to calculate move: 30 seconds

Time Complexity

Minmax uses Depth First Search (DFS) on Game Tree, hence the time complexity of minmax algorithm is **$O(b^m)$**, *where b is branching factor of the game-tree, and m is the maximum depth of the tree.*

Space Complexity

Space complexity of minmax algorithm is also similar to DFS which is **O(bm)**, *where b is branching factor of the game-tree, and m is the maximum depth of the tree.*

Completeness

Minmax algorithm is Complete. It will definitely find a solution, if exists, in the finite search tree.

Optimality

Minmax algorithm is optimal if both opponents are playing optimally.

Alpha-Beta Pruning

The minmax algorithm can be optimized by pruning few branches. Pruned branches are the ones that are not going to affect result. It will improve time-complexity. This version minmax is knows as minmax with alpha-beta pruning. It is also called as alpha-beta algorithm.

In Alpha-Beta Pruning, there are two values, Alpha and Beta. Below are the few points to consider about alpha and beta:

Some point about Alpha(α)

- Alpha is the highest value, that is found along the MAX path.
- Initial value of alpha is negative infinity because alpha value will keep on increasing or remain same with every move. If we choose some value other than negative infinity, then the scenario may occur in which all values of alpha may be less than chosen value. So, we have to choose lowest possible value, and that is negative infinity.
- Alpha is only updated by MAX player.

Some points about Beta(β)

- Beta is the lowest value, that is found along the MIN path.
- Initial value of beta is positive infinity because beta value will keep on decreasing or remain same with every move. If we choose some value other than positive infinity, then scenario may occur in which all values of beta may be more than chosen value. So, we have to choose maximum possible value, and that is positive infinity.
- Beta value is only updated by MIN player.

While backtracking only node value is passed to parent, not the alpha and beta value.

The Condition for alpha-beta pruning:

α >= β

Move Ordering

The effectiveness of alpha-beta algorithm is highly depending on order of traversal. It plays crucial role in Time and Space Complexity.

Worst Ordering

In some cases, no node or sub-tree is pruned out of Game Tree. In this case, best move occurs in right sub-tree of Game Tree. This will result in increased Time Complexity.

Ideal Ordering

In this case, maximum number of node and sub-tree is pruned. Best moves occur in left subtree. It will reduce the Time and Space Complexity.

Pseudo Code

```
function alpha_beta (node, depth, is_max_player, alpha, beta)
        if depth is 0 or node is a leaf node then ## This is the base case
                return static evaluation of node.
        if is_max_player then
            max_val = -infinity
           for each child of node do
                node_val = alpha_beta (child, depth -1, false, alpha, beta) ## This is the recursion
                max_val = max (max_val, node_val)
                undo the last move
                alpha = max (alpha, node_val)
                if beta <= alpha:
                        return max_val
            return max_val.
        else
             min_val = infinity
             for each child of node do
                node_val = alpha_beta (child, depth -1, false, alpha, beta) ## This is the recursion
                min_val = max (min_val, node_val) ## here we are backtracking
                undo the last move
                beta = min (beta, node_val)
                if beta <= alpha:
                        return max_val
              return min_val.
```

Python Implementation

Use this link for complete python implementation

```
def alpha_beta( board_instance, max_depth, current_depth, is_max_player, alpha, beta, nodes_per_depth ):

    # This if else code block is only used for analysis of algorithm, by counting number of nodes explored
    if max_depth-current_depth in nodes_per_depth:
        nodes_per_depth[max_depth-current_depth] += 1
    else:
        nodes_per_depth[max_depth-current_depth] = 1

    if current_depth == 0:
        leaf_node_score = static_eval(board_instance)
        return (leaf_node_score, nodes_per_depth)

    if is_max_player:

        # set absurdly high negative value such that none of the static evaluation result less than this value
        best_score = -100000

        for legal_move in board_instance.legal_moves:
            move =  chess.Move.from_uci(str(legal_move))

            # pusshing the current move to the board
            board_instance.push(move)

            #calculating node score, if the current node will be the leaf node, then score will be calculated by static evaluation;
            #score will be calculated by finding max value between node score and current best score.
            node_score, nodes_per_depth = alpha_beta(board_instance, max_depth, current_depth - 1, False, alpha, beta, nodes_per_depth)

            # calculating best score by finding max value between current best score and node score
            best_score = max(best_score, node_score)

            # undoing the last move, so as to explore new moves while backtracking
            board_instance.pop()

            # calculating alpha for current MAX node
            alpha = max(alpha, best_score)

            # beta cut off
            if beta <= alpha:
                return (best_score, nodes_per_depth)

        return (best_score, nodes_per_depth)
    else:

        # set absurdly high positive value such that none of the static evaluation result more than this value
        best_score = 100000

        for legal_move in board_instance.legal_moves:
            move =  chess.Move.from_uci(str(legal_move))

            # pushing the current move to the board
            board_instance.push(move)

            #calculating node score, if the current node will be the leaf node, then score will be calculated by static evaluation;
            #score will be calculated by finding min value between node score and current best score.
            node_score, nodes_per_depth = alpha_beta(board_instance, max_depth, current_depth - 1, True, alpha, beta, nodes_per_depth)

            # calculating best score by finding min value between current best score and node score
            best_score = min(best_score, node_score)

            # undoing the last move, so as to explore new moves while backtracking
            board_instance.pop()

            # calculating alpha for current MIN node
            beta = min(beta, best_score)

            # beta cut off
            if beta <= alpha:
                return (best_score, nodes_per_depth)

        return (best_score, nodes_per_depth)
```

Fig. 1-9 alpha-beta python implementation

Game Play Analysis

In this section, we will allow two alpha-beta algorithms to compete against each other. We will calculate the total time taken by each move and number of nodes explored per depth. We will only explore till depth = 3 because of system restriction. We will allow 3 move each for Black and White piece. In the complete python implementation we have used 10 moves per player.

WHITE Turn

- **Move in UCI format:** b1c3
- **Nodes per depth:** {0: 1, 1: 20, 2: 135, 3: 1076}
- **Time taken by Move:** 1.9858489036560059

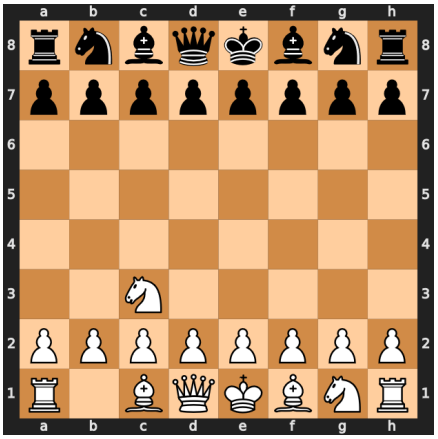


Fig. 1-10 alpha beta move 1

BLACK Turn

- **Move in UCI format:** c7c6
- **Nodes per depth:** {0: 1, 1: 22, 2: 43, 3: 609}
- **Time taken by Move:** 1.7624576091766357

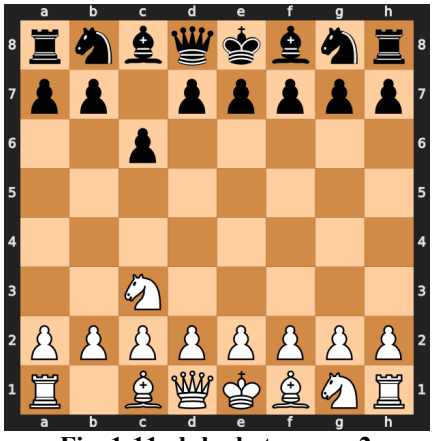


Fig. 1-11 alpha beta move 2

WHITE Turn

- **Move in UCI format:** a1b1
- **Nodes per depth:** {0: 1, 1: 21, 2: 214, 3: 2143}
- **Time taken by Move:** 7.133767604827881

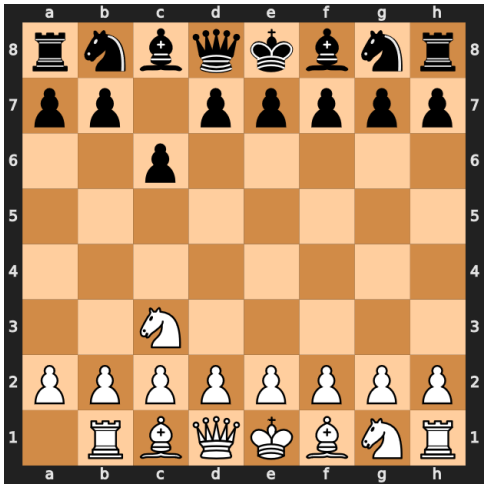


Fig. 1-12 alpha beta move 3

BLACK Turn

- **Move in UCI format:** g8h6
- **Nodes per depth:** {0: 1, 1: 21, 2: 41, 3: 692}
- **Time taken by Move:** 2.445237159729004

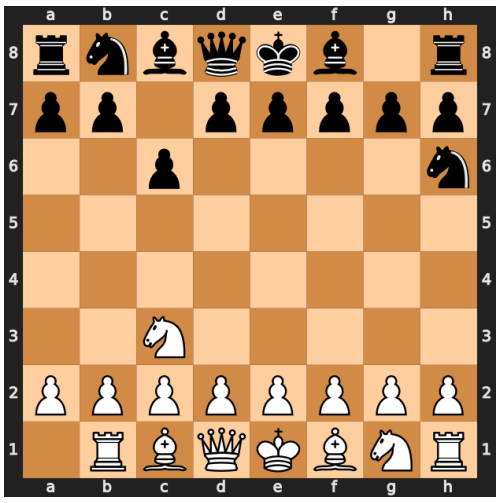


Fig. 1-13 alpha beta move 4

WHITE Turn

- **Move in UCI format:** c3e4
- **Nodes per depth:** {0: 1, 1: 21, 2: 103, 3: 1610}
- **Time taken by Move:** 5.03103232383728

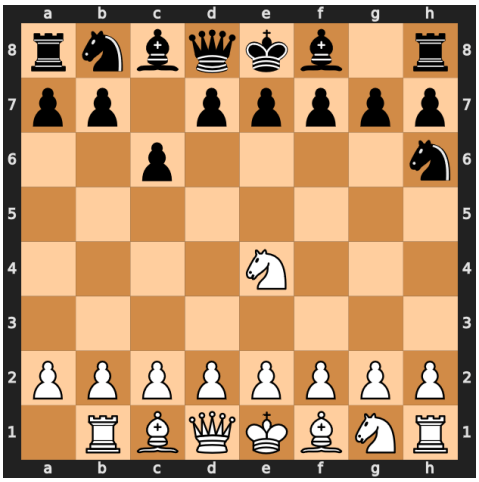


Fig. 1-14 alpha beta move 5

BLACK Turn

- **Move in UCI format:** d8c7
- **Nodes per depth:** {0: 1, 1: 24, 2: 25, 3: 601}
- **Time taken by Move:** 1.78654146194458

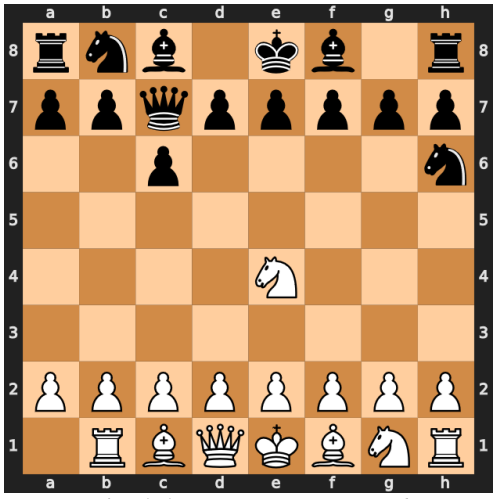


Fig. 1-15 alpha beta move 6

It is visible from above experiment following can be inferred:

- Average Branching Factor: 20
- Average time taken by algorithm to calculate move: 3 seconds

Time Complexity

Alpha Beta Algorithm, also uses Depth First Search (DFS) on Game Tree.

- **In case of Worst Ordering:** $O(b^m)$
- **In case of Ideal Ordering:** $O(b^{m/2})$

Space Complexity

- **In case of Worst Ordering:** O(bm)
- **In case of Ideal Ordering:** O(b(m/2))

Completeness

Alpha-Beta algorithm is Complete. It will definitely find a solution, if exists, in the finite search tree.

Optimality

Alpha-Beta algorithm is optimal if both opponents are playing optimally.

Quiescence Search

Quiescence Search is a modification on top of alpha-beta pruning with min-max. Quiescence means quiet. This search only evaluates only quiet moves. In other words, after a certain depth, it only uses capture moves to calculate the next moves. A quiescence search can avoid the horizon effect.

The horizon effect occurs when we only search to a certain depth, but it may happen that if we look one more level deep, then move may score fewer points. A quiescence search only uses capture moves to prevent this.

Quiescence search also reduces the branching factor at different levels, resulting in a fast algorithm.

Python Implementation

Use this link for complete python implementation

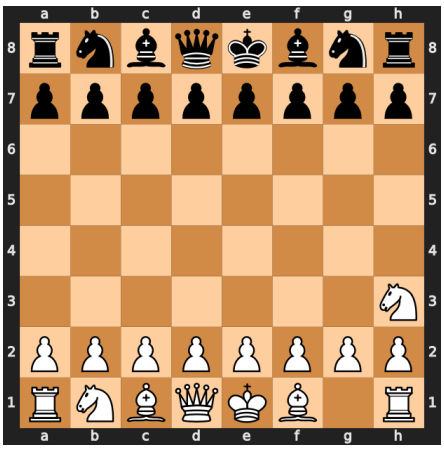
```
def quiescence_Search( board_instance, max_depth, current_depth, is_max_player, alpha, beta, nodes_per_depth ):

    # This if else code block is only used for analysis of algorithm, by counting number of nodes explored
    if max_depth-current_depth in nodes_per_depth:
        nodes_per_depth[max_depth-current_depth] += 1
    else:
        nodes_per_depth[max_depth-current_depth] = 1

    if current_depth == 0:
        leaf_node_score = static_eval(board_instance)
        return (leaf_node_score, nodes_per_depth)

    if max_depth-current_depth > 3:
        all_possible_capture_moves = [ move for move in board_instance.legal_moves if  is_favorable_move(board_instance, move) ]
    else:
        all_possible_capture_moves = board_instance.legal_moves


    if is_max_player:

        # set absurdly high negative value such that none of the static evaluation result less than this value
        best_score = -100000

        for legal_move in all_possible_capture_moves:
            move =  chess.Move.from_uci(str(legal_move))

            # pusshing the current move to the board
            board_instance.push(move)

            #calculating node score, if the current node will be the leaf node, then score will be calculated by static evaluation;
            #score will be calculated by finding max value between node score and current best score.
            node_score, nodes_per_depth = quiescence_Search(board_instance, max_depth, current_depth - 1, False, alpha, beta, nodes_per_d
epth)

            # calculating best score by finding max value between current best score and node score
            best_score = max(best_score, node_score)

            # undoing the last move, so as to explore new moves while backtracking
            board_instance.pop()

            # calculating alpha for current MAX node
            alpha = max(alpha, best_score)

            # beta cut off
            if beta <= alpha:
                return (best_score, nodes_per_depth)

        return (best_score, nodes_per_depth)
    else:

        # set absurdly high positive value such that none of the static evaluation result more than this value
        best_score = 100000

        for legal_move in all_possible_capture_moves:
            move =  chess.Move.from_uci(str(legal_move))

            # pushing the current move to the board
            board_instance.push(move)

            #calculating node score, if the current node will be the leaf node, then score will be calculated by static evaluation;
            #score will be calculated by finding min value between node score and current best score.
            node_score, nodes_per_depth = quiescence_Search(board_instance, max_depth, current_depth - 1, True, alpha, beta, nodes_per_de
pth)

            # calculating best score by finding min value between current best score and node score
            best_score = min(best_score, node_score)

            # undoing the last move, so as to explore new moves while backtracking
            board_instance.pop()

            # calculating alpha for current MIN node
            beta = min(beta, best_score)

            # beta cut off
            if beta <= alpha:
                return (best_score, nodes_per_depth)

        return (best_score, nodes_per_depth)
```

Game Play Analysis

In this section, we will allow two quiescence algorithms to compete against each other. We will calculate the total time taken by each move and number of nodes explored per depth. We will only explore till depth = 10. We will allow 3 move each for Black and White piece. In the complete python implementation we have used 10 moves per player.

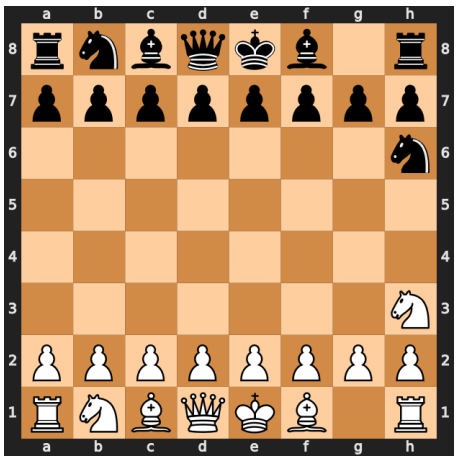
WHITE Turn

- **Move in UCI format:** g1h3
- **Nodes per depth:** {0: 1, 1: 20, 2: 20, 3: 445, 4: 638, 5: 201, 6: 8}
- **Time taken by Move:** 4.668341875076294



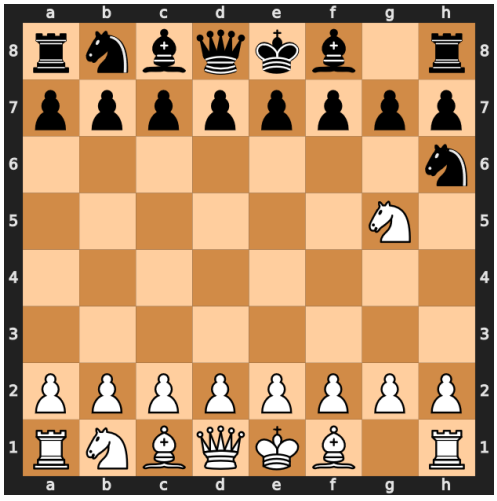
BLACK Turn

- **Move in UCI format**: g8h6
- **Nodes per depth:** {0: 1, 1: 20, 2: 20, 3: 449, 4: 485, 5: 40, 6: 4}
- **Time taken by Move:** 2.7566113471984863



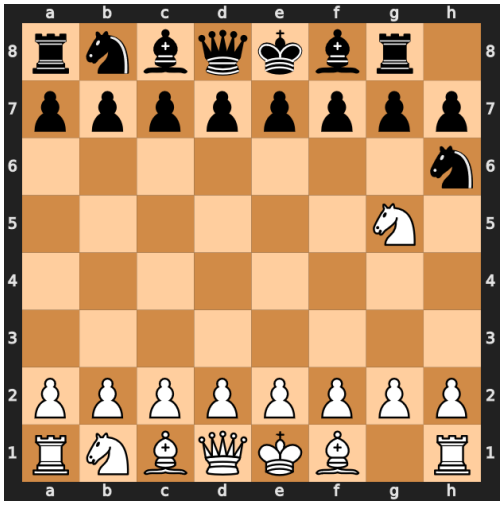
WHITE Turn

- **Move in UCI format:** h3g5
- **Nodes per depth:** {0: 1, 1: 20, 2: 20, 3: 426, 4: 1547, 5: 1166, 6: 48, 7: 3}
- **Time taken by Move:** 9.771346092224121



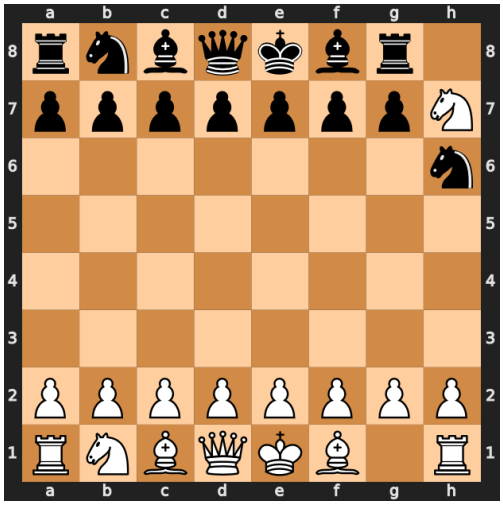
BLACK Turn

- **Move in UCI format**: h8g8
- **Nodes per depth:** {0: 1, 1: 25, 2: 29, 3: 690, 4: 923, 5: 257, 6: 21, 7: 2, 8: 2, 9: 1}
- **Time taken by Move:** 5.28132176399231



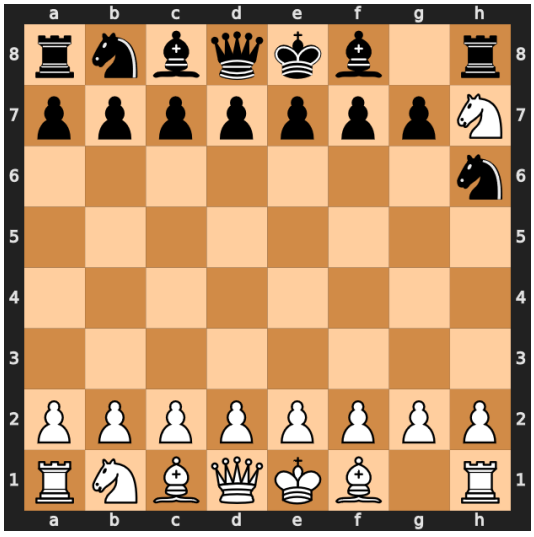
WHITE Turn

- **Move in UCI format:** g5h7
- **Nodes per depth:** {0: 1, 1: 18, 2: 23, 3: 409, 4: 1464, 5: 1151, 6: 93, 7: 2}
- **Time taken by Move:** 8.836541652679443



BLACK Turn

- **Move in UCI format:** g8h8
- **Nodes per depth:** {0: 1, 1: 22, 2: 22, 3: 528, 4: 528}
- **Time taken by Move:** 2.7431085109710693



Time Complexity

Quiescence Search Algorithm, also uses Depth First Search (DFS) on Game Tree.

- **In case of Worst Ordering:** $O(b^m)$
- **In case of Ideal Ordering:** $O(b^{m/2})$

Space Complexity

- **In case of Worst Ordering:** O(bm)
- **In case of Ideal Ordering:** O(b(m/2))

Completeness

Quiescence Search algorithm is Complete. It will definitely find a solution, if exists, in the finite search tree.

Optimality

Quiescence Search algorithm is optimal if both opponents are playing optimally.

CONCLUSION

We performed three experiment. We utilized three algorithms (Minmax, Alpha-Beta and Quiescence Search) to play against each other

Below are the few conclusions that can be drawn from 3 experiments:

- Nodes explored per move is reduced by 10 folds in Alpha Beta as compared with Minmax.
- Time Taken to find best move is also reduced by 10 folds in Alpha Beta as compared with Minmax.
- Quiescence search performs well in higher depth.
- Quiescence search is the fastest algorithm out of all three.
- Quiescence search evaluates less nodes.
- Quiescence search may result in repeated moves, if not handled properly.