Chess Engine

with MiniMax and Alpha Beta Pruning

Nick Zhang

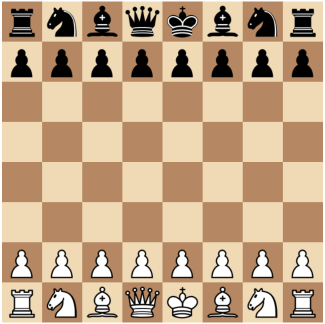
Wednesday 20th October, 2021

Table of Contents

Table of Contents

Introduction

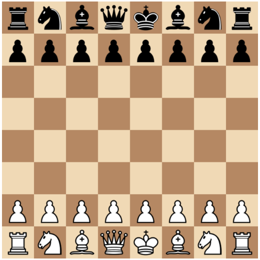
- Chess is a game played between two players.
- It is a strategy zero-sum game with perfect information.
- Objective of the game is to checkmate opponent's king



Main Ingredients

Preliminary Concepts

- Game Implementation



Main Ingredients

Preliminary Concepts

- Game Implementation
  - chess.js
  - chessboard.js

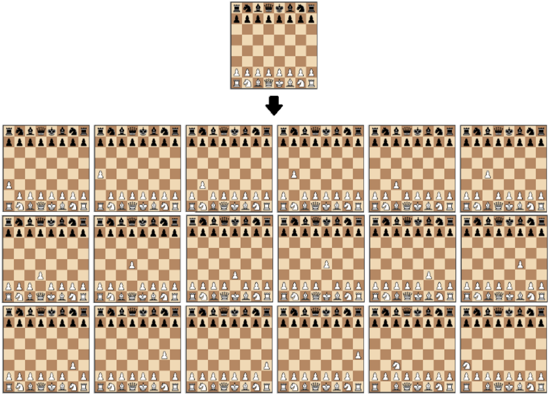


Figure: Move Generating Function

Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function

$f\Big($  $\Big)$

$= ?$

Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function
    - Piece Values

♙	1.00	♟	-1.00
♘	3.20	♞	-3.20
♗	3.30	♝	-3.30
♖	5.00	♜	-5.00
♕	9.00	♛	-9.00
♔	200.00	♚	-200.00

Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function
  - Piece Values
  - Piece-Square Tables

```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

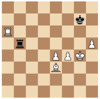
```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function
  - Piece Values
  - Piece-Square Tables

$f\Big($  $\Big) = +7.05$

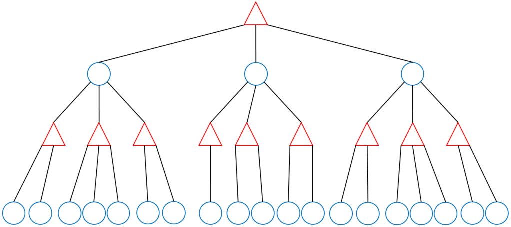
$f\Big($  $\Big) = +0.05$

$f\Big($  $\Big) = -9.55$

Main Ingredients

Preliminary Concepts

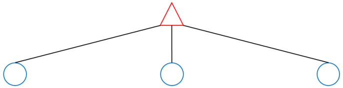
- Game Implementation
- Evaluation Function
- Tree Search



Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function
- Tree Search
  - Branching Factor

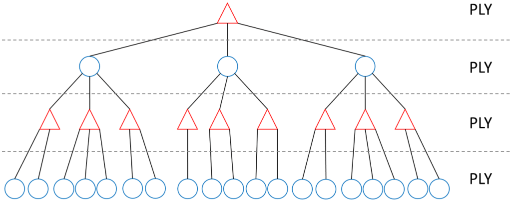


$$b = 3$$

Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function
- Tree Search
  - Branching Factor
  - Plies



Main Ingredients

Preliminary Concepts

- Game Implementation
- Evaluation Function
- Tree Search
  - Branching Factor
  - Plies
  - Depth

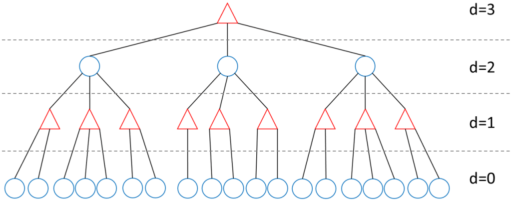


Table of Contents

Basic Idea

Task

Basic Idea

Task

Given a board position, win the game.

Basic Idea

Task

Given a board position, win the game.

Problem:

- Impossible (with current technology) to traverse the entire tree:

Basic Idea

Task

Given a board position, win the game.

Problem:

- Impossible (with current technology) to traverse the entire tree: *The Shannon Number*, with a branching factor of 35 and a game lenght of 80 plies, sets a conservative lower-bound of the game-tree complexity of chess to be around 10^{120}.
- In comparison, the number of atoms in our observable universe, is roughly estimated to be around 10^{80}.

Basic Idea

Task

Given a board position, make the best possible move.

Basic Idea

Task

Given a board position, make the best possible move.

By scaling back the task, it becomes feasible to attempt to solve it by brute forcing all the possible moves from the current position, up to a certain number of moves ahead (*look-ahead*).

Basic Idea

Task

Given a board position, make the best possible move.

By scaling back the task, it becomes feasible to attempt to solve it by brute forcing all the possible moves from the current position, up to a certain number of moves ahead (*look-ahead*).

- Although easier, the number of positions to be evaluated is still exponentially increasing. With an average branching factor of 30, a game-tree will have approximately 27,000 terminal nodes to evaluate after 3 plies ahead; 810,000 after 4; 24,300,000 after only 5 half-moves.

Basic Idea

Task

Given a board position, make the best possible move.

By scaling back the task, it becomes feasible to attempt to solve it by brute forcing all the possible moves from the current position, up to a certain number of moves ahead (*look-ahead*).

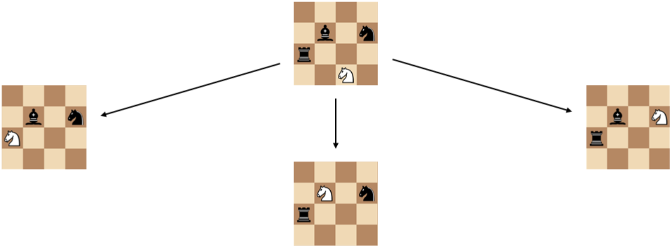
- Although easier, the number of positions to be evaluated is still exponentially increasing. With an average branching factor of 30, a game-tree will have approximately 27,000 terminal nodes to evaluate after 3 plies ahead; 810,000 after 4; 24,300,000 after only 5 half-moves.
- How do we traverse and evaluate the game-tree?

MiniMax Routine

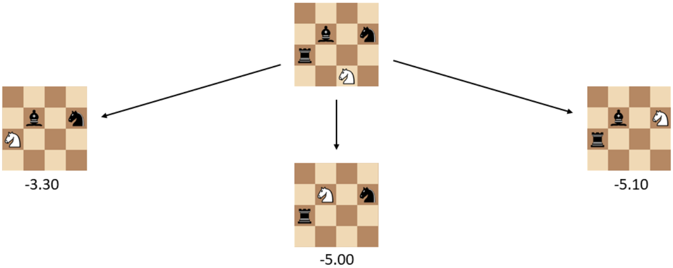
MiniMax Routine

-8.30

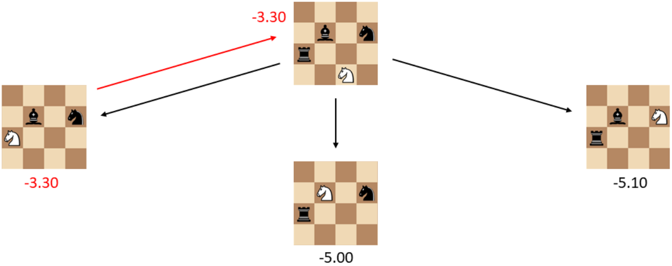
MiniMax Routine



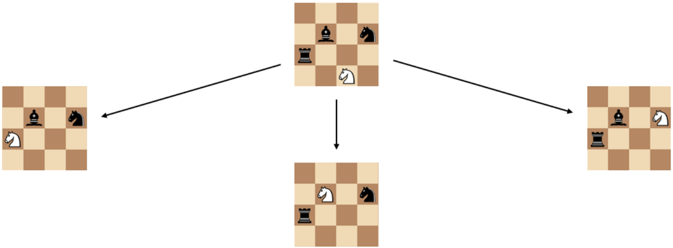
MiniMax Routine



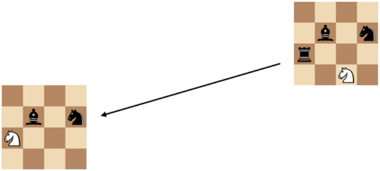
MiniMax Routine



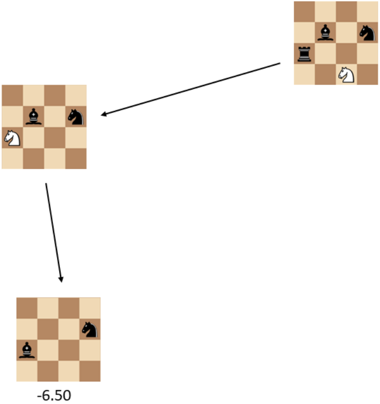
MiniMax Routine



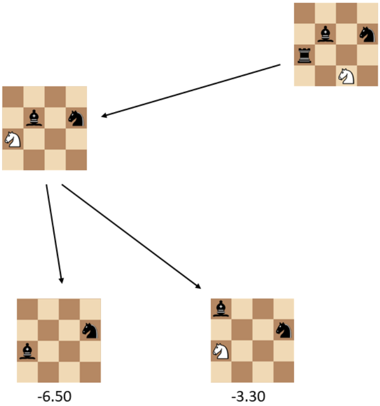
MiniMax Routine



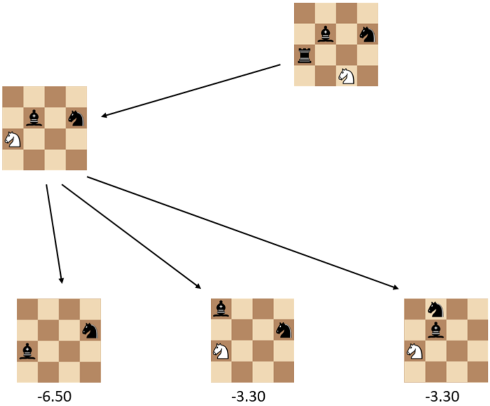
MiniMax Routine



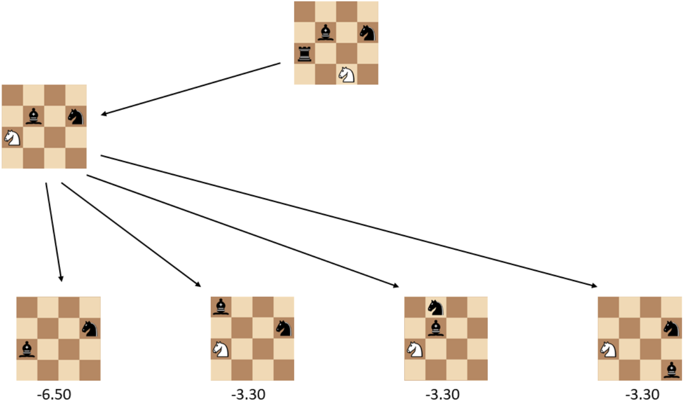
MiniMax Routine



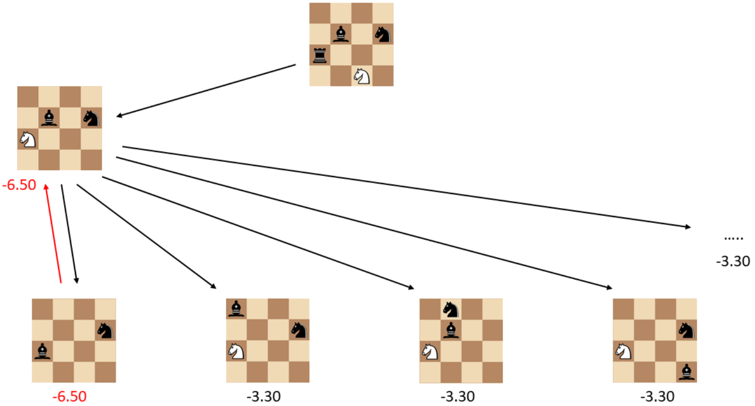
MiniMax Routine



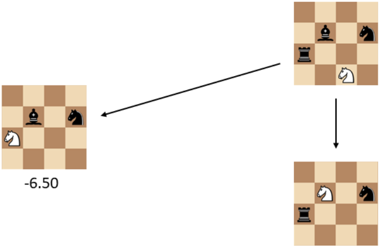
MiniMax Routine



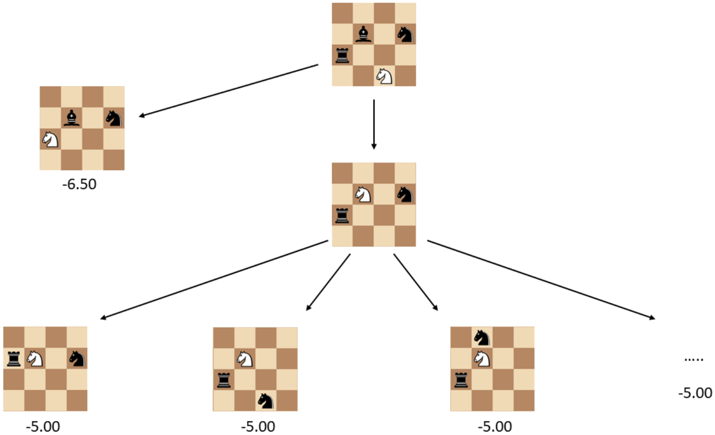
MiniMax Routine



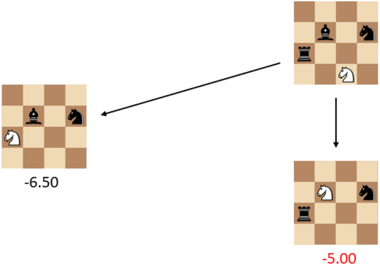
MiniMax Routine



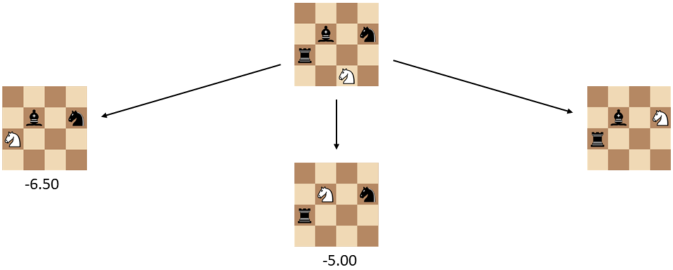
MiniMax Routine



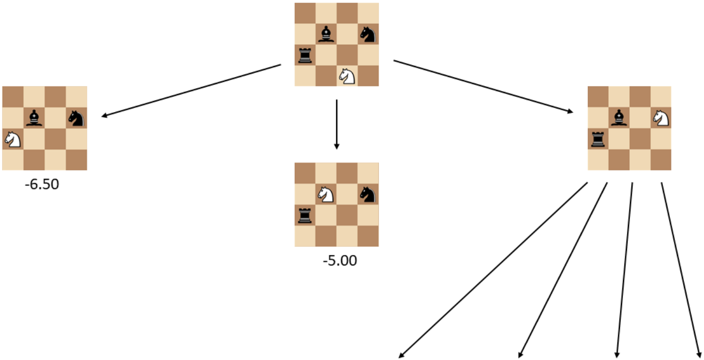
MiniMax Routine



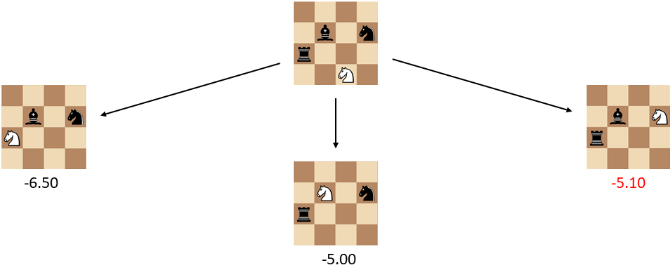
MiniMax Routine



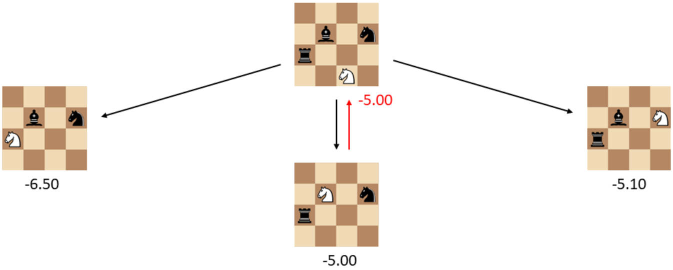
MiniMax Routine



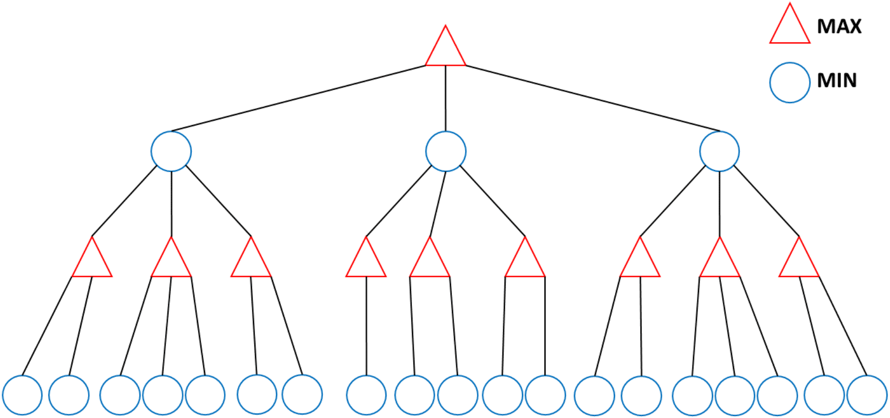
MiniMax Routine



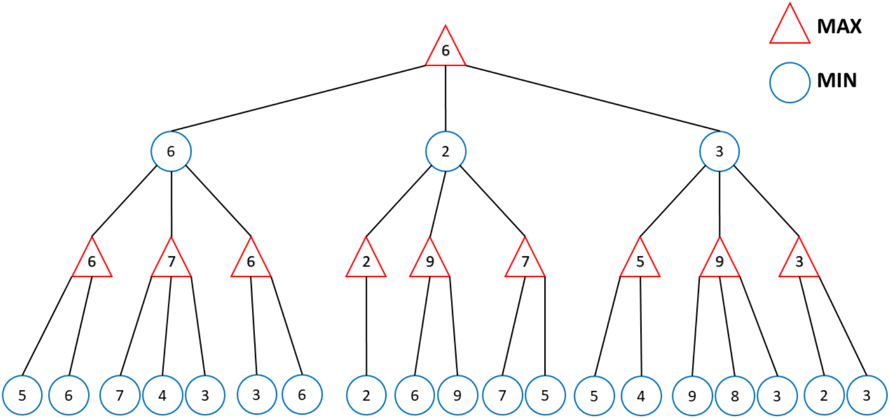
MiniMax Routine



MiniMax Routine



MiniMax Routine



Formal Definition

For a generalization to n-player game, the **minimax** value for player i is:

- The smallest value that the other players can force i to receive, without knowing i's moves;
- The largest value that the player can be sure to get when they know the actions of the other players

Formal Definition

For a generalization to n-player game, the **minimax** value for player *i* is:

- The smallest value that the other players can force *i* to receive, without knowing *i*'s moves;
- The largest value that the player can be sure to get when they know the actions of the other players

Definition (Minimax Value)

Let $v_i(A)$ be the value function for player *i* with set of actions A. The **minimax value** is defined as:

$$\underline{v_i} = \min_{s_{-i}} \max_{s_i} u_i(a_i, a_{-i})$$

Where:

- -*i* denotes all other players except *i*
- a_i, a_{-i} denote actions taken by $i, -i$

Pseudocode

Algorithm MiniMax

```
function MINIMAX(node, depth, isMaximizingPlayer)
    if terminal node or depth = 0 then
        return eval(node)
    if isMaximizingPlayer then
        value ← −∞
        for each child of node do
            value ← max(value, minimax(child, depth − 1, FALSE))
        return value
    else            #Minimizing Player
        value ← +∞
        for each child of node do
            value ← min(value, minimax(child, depth − 1, TRUE))
        return value
```

Complexity Analysis

Time Complexity

Asymptotic time complexity of $O(b^d)$

Space Complexity

Asymptotic space complexity of $O(b \times m)$

Where:

- b is the average *branching factor* (number of children at each node, number of legal moves possible at each board position);
- d is *depth* (number of plies, number of *look-ahead* moves) of the search tree.

Table of Contents

Main Ideas

Minimax's problem:

- In naive minimax, the amount of nodes to be visited grows exponentially with the depth of search.
- This is clearly very costly and requires an incredible amount of computational power to reach deeper depths.

Main Ideas

Minimax's problem:

- In naive minimax, the amount of nodes to be visited grows exponentially with the depth of search.
- This is clearly very costly and requires an incredible amount of computational power to reach deeper depths.
- We want to find a way to reduce the number of nodes to be computed.

Main Ideas

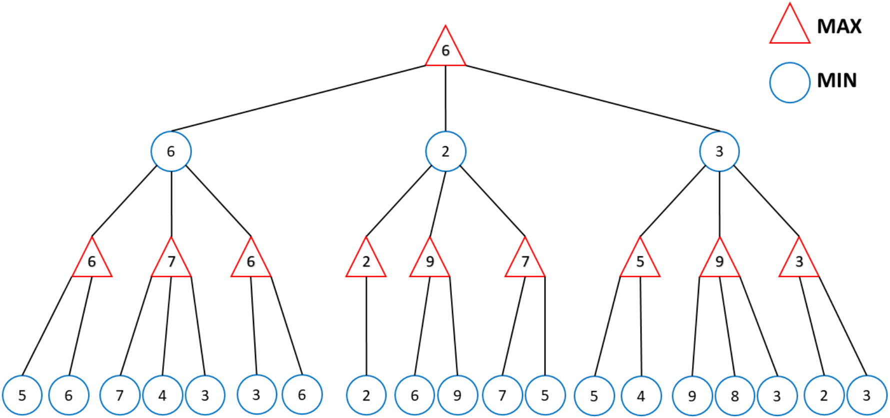
Minimax's problem:

- In naive minimax, the amount of nodes to be visited grows exponentially with the depth of search.
- This is clearly very costly and requires an incredible amount of computational power to reach deeper depths.
- We want to find a way to reduce the number of nodes to be computed.

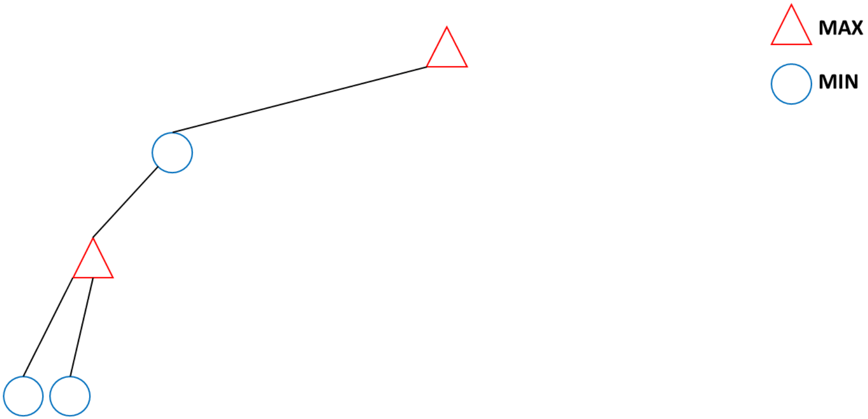
Main Intuition:

- Stop evaluating a move further when at least one possibility has been found that proves the move to be worse than a previously examined move.

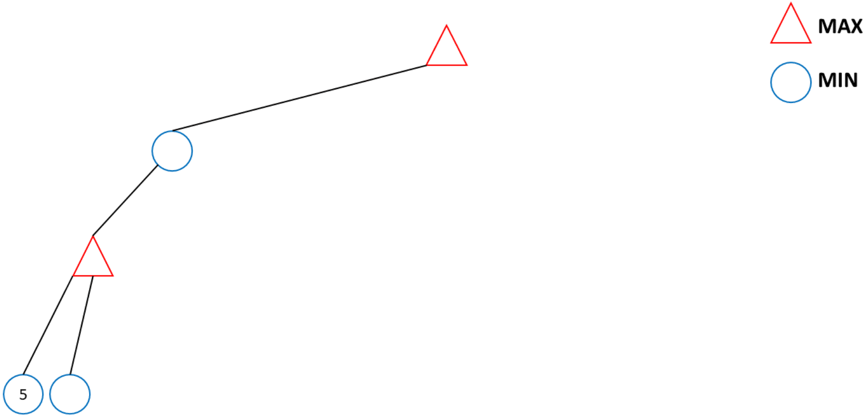
Graphical Example



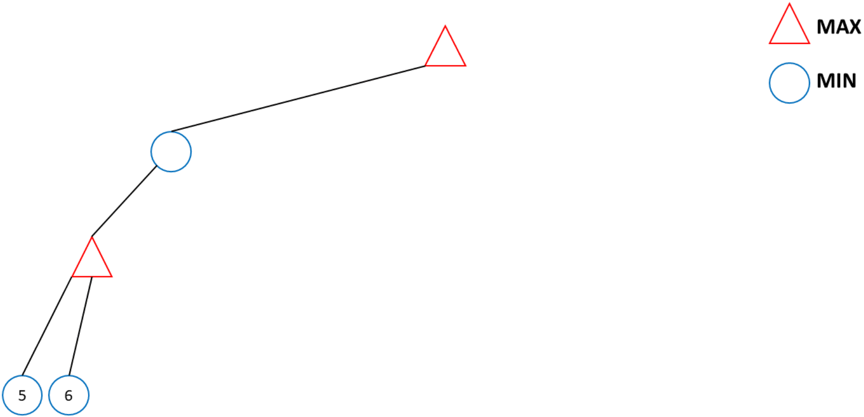
Graphical Example



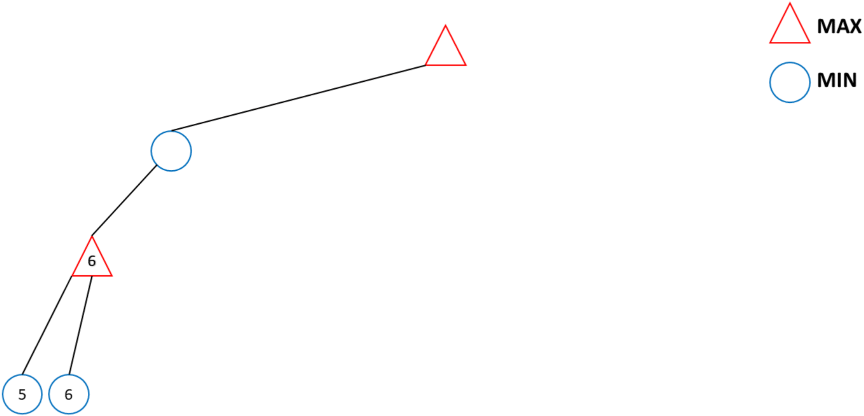
Graphical Example



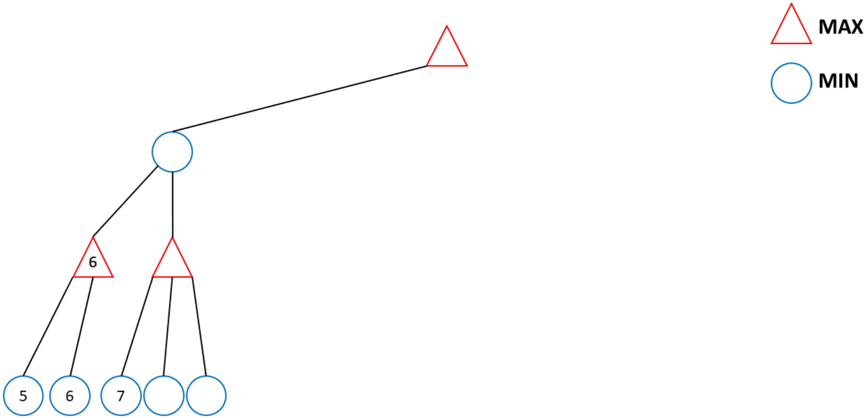
Graphical Example



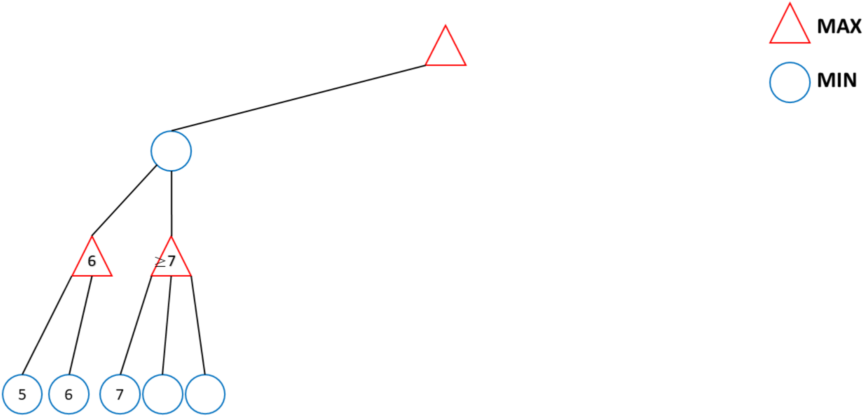
Graphical Example



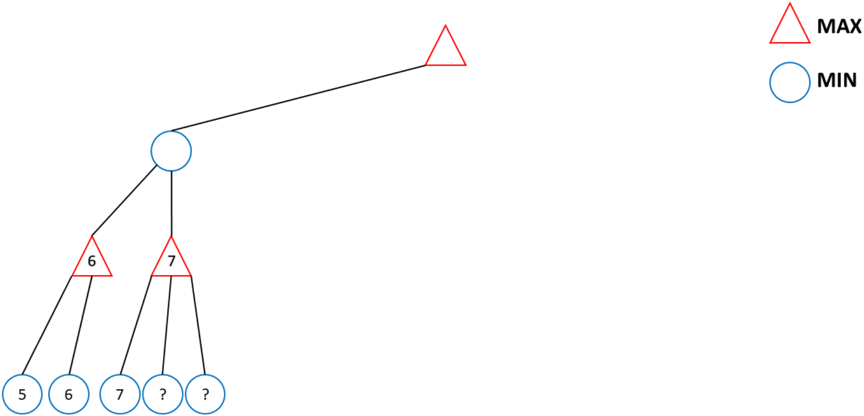
Graphical Example



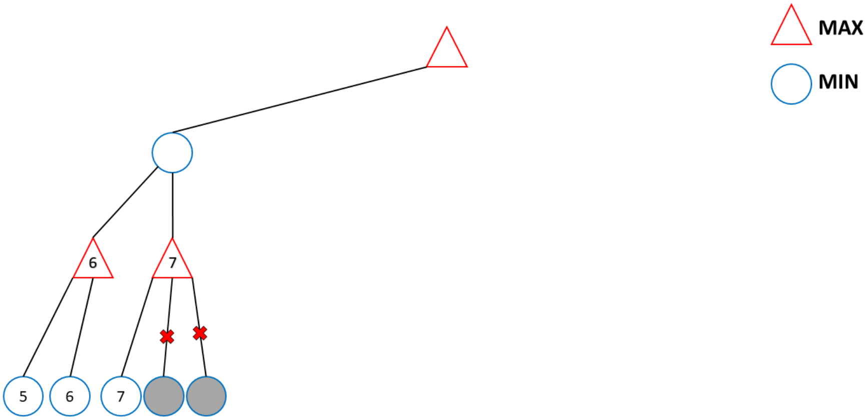
Graphical Example



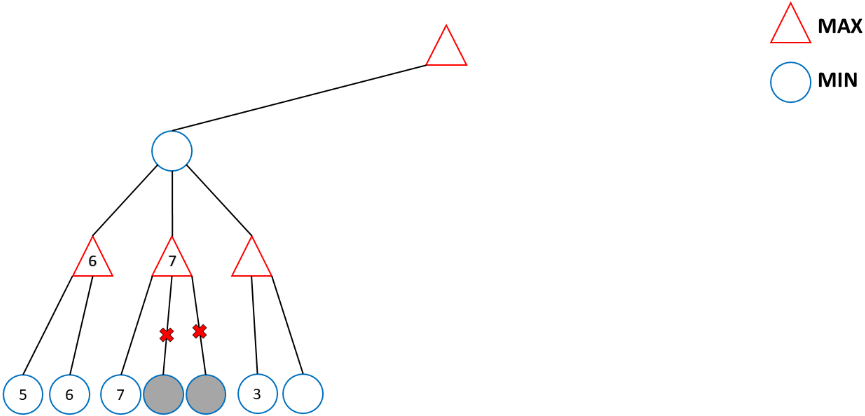
Graphical Example



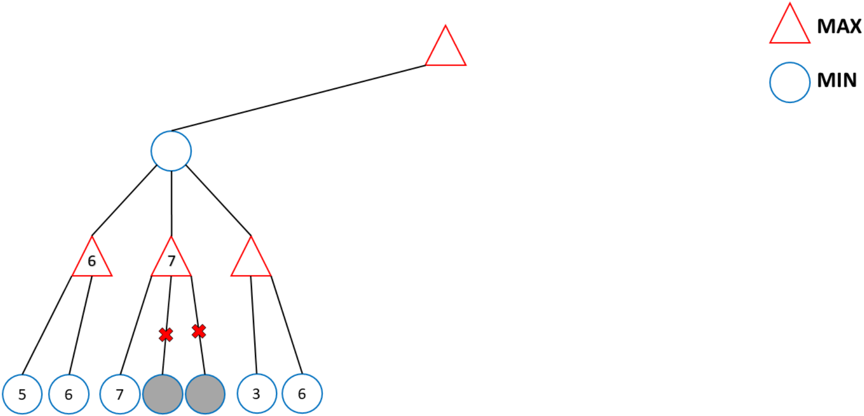
Graphical Example



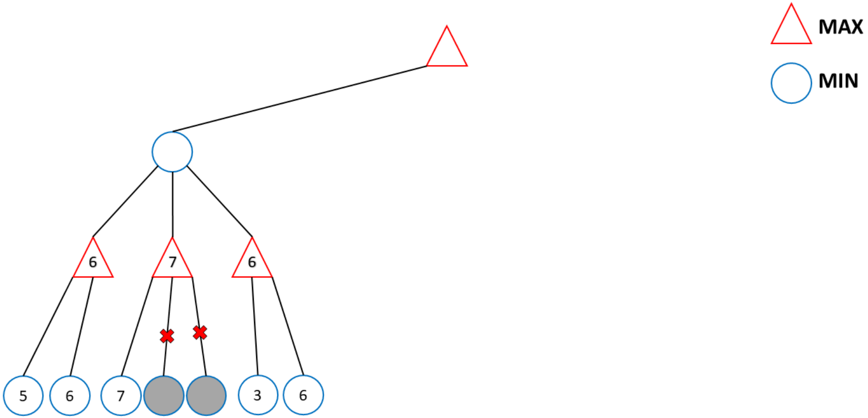
Graphical Example



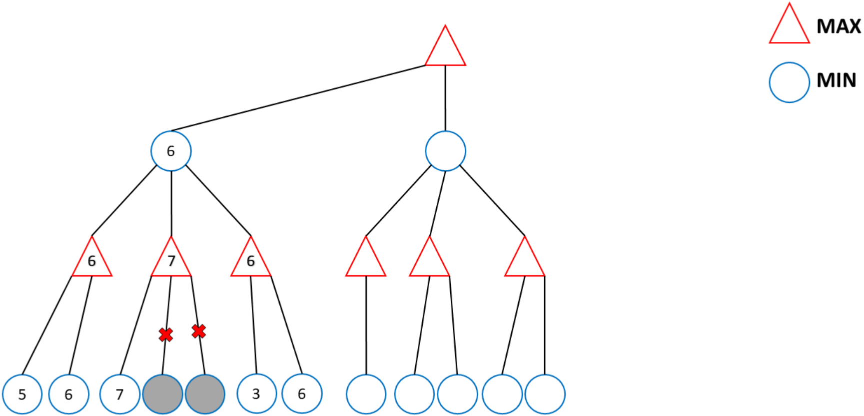
Graphical Example



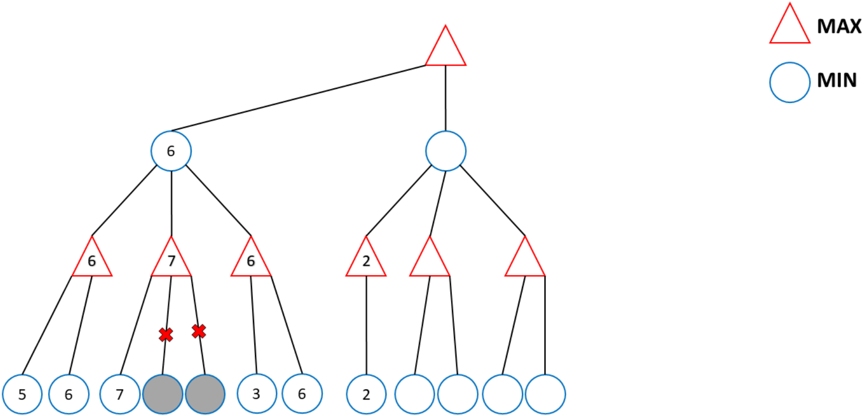
Graphical Example



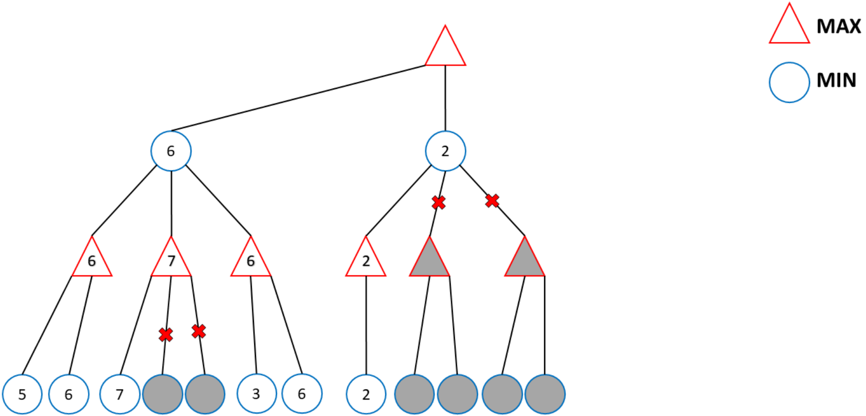
Graphical Example



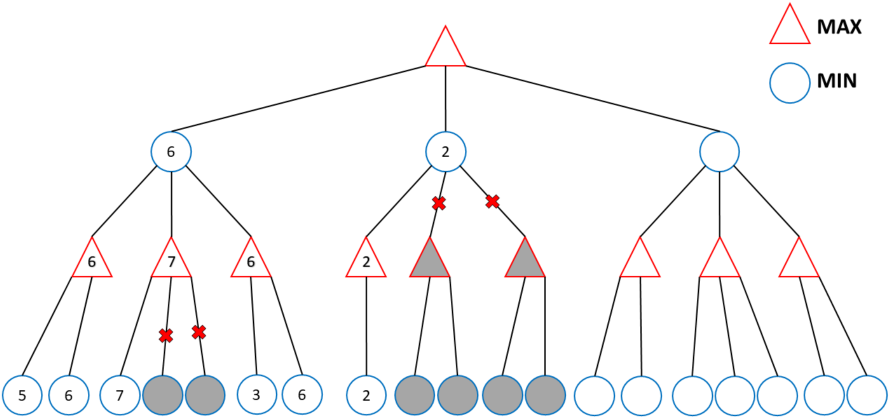
Graphical Example



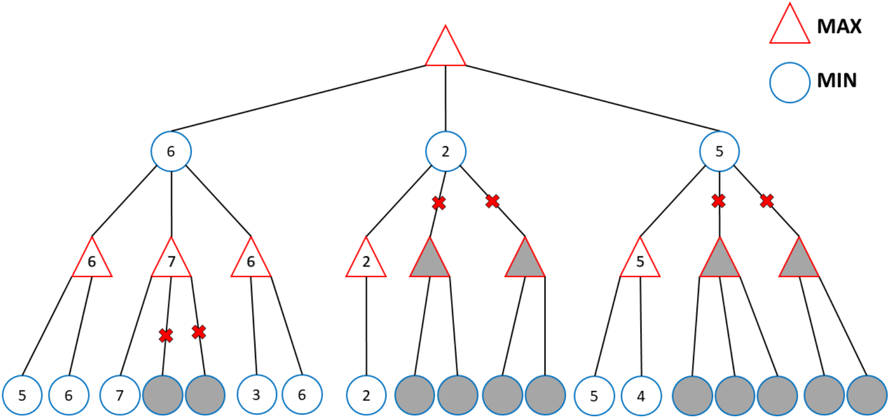
Graphical Example



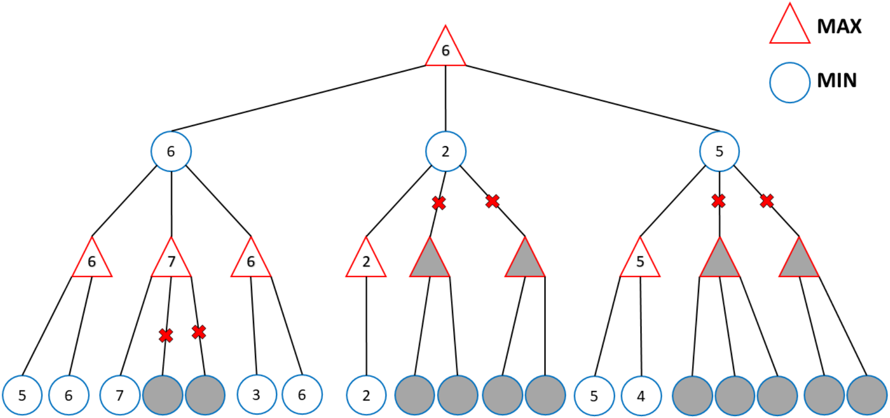
Graphical Example



Graphical Example



Graphical Example



AlphaBeta

- Minimax algorithm is a way of finding an optimal move in a two player game.
- Alpha-Beta is a way of finding the same MM optimal solution while pruning off subtrees of moves that cannot possibly influence the final decision.
- The name comes from the two bounds that are passed along during the tree search process, which restrict the set of possible solutions based on the portion of the search tree that has already been seen.

AlphaBeta

- Minimax algorithm is a way of finding an optimal move in a two player game.
- Alpha-Beta is a way of finding the same MM optimal solution while pruning off subtrees of moves that cannot possibly influence the final decision.
- The name comes from the two bounds that are passed along during the tree search process, which restrict the set of possible solutions based on the portion of the search tree that has already been seen.

More specifically:

- β (associated with MIN nodes), represents the minimum upper bound of possible solutions.
- α (associated with MAX nodes), represents the maximum lower bound of possible solutions.

AlphaBeta

In practice:

- The algorithm mantains two values, β and α, that are "*passed*" along the nodes as we search through them.

AlphaBeta

In practice:

- The algorithm mantains two values, β and α, that are "*passed*" along the nodes as we search through them.
- β represents the minimum score that the maximizing player is assured of;
- α represents the maximum score that the minimizing player is assured of.

AlphaBeta

In practice:

- The algorithm mantains two values, β and α, that are "*passed*" along the nodes as we search through them.
- β represents the minimum score that the maximizing player is assured of;
- α represents the maximum score that the minimizing player is assured of.
- β and α are initialized as $-\infty$ and $+\infty$, respectively (both players start with the worst possible score).

AlphaBeta

In practice:

- The algorithm mantains two values, β and α, that are "*passed*" along the nodes as we search through them.
- β represents the minimum score that the maximizing player is assured of;
- α represents the maximum score that the minimizing player is assured of.
- β and α are initialized as $-\infty$ and $+\infty$, respectively (both players start with the worst possible score).
- These two boundary values are then updated as we compute the heuristic values of nodes we traverse.

AlphaBeta

In practice:

- The algorithm mantains two values, β and α, that are "*passed*" along the nodes as we search through them.
- β represents the minimum score that the maximizing player is assured of;
- α represents the maximum score that the minimizing player is assured of.
- β and α are initialized as $-\infty$ and $+\infty$, respectively (both players start with the worst possible score).
- These two boundary values are then updated as we compute the heuristic values of nodes we traverse.
- Whenever $\beta \leq \alpha$, the maximizing player need not to consider further moves of the node, as under the assumption of optimal actions by both players, such moves will never be reached in actual play (we can *prune off* those branches of the search tree).

AlphaBeta

Algorithm MiniMax with AlphaBeta pruning

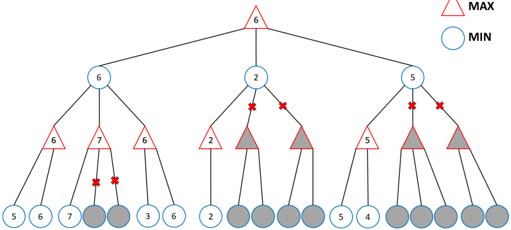
```
function ALPHABETA(node, depth, α, β, isMaximizingPlayer)
    if terminal node or depth = 0 then
        return eval(node)
    if isMaximizingPlayer then
        value ← −∞
        for each child of node do
            value ← max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α ← max(α, value)
            if β ≤ α then
                break
        return value
    else            #Minimizing Player
        value ← +∞
        for each child of node do
            value ← min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β ← min(β, value)
            if β ≤ α then
                break
        return value
```

Improvements over Naive Minimax

- Alphabeta allows to decrease the number of branches to be examined in the search tree; the search time can be limited to more 'promising' subtrees and a deeper search can be performed consuming the same time.

Improvements over Naive Minimax

- Alphabeta allows to decrease the number of branches to be examined in the search tree; the search time can be limited to more 'promising' subtrees and a deeper search can be performed consuming the same time.
- In the previous example, we were able to cut down the number of terminal nodes visited to 8 out of the 19 visited in naive minimax.



Complexity Analysis

- Given the unverifiability *a priori* of the number of nodes that can be eliminated, the exact improvement over naive minimax is hard to quantify.

Complexity Analysis

- Given the unverifiability *a priori* of the number of nodes that can be eliminated, the exact improvement over naive minimax is hard to quantify.
- Worst case (*pessimal* move ordering):
  - children of MAX nodes are searched from least to greatest; children of MIN nodes are searched from greatest to least.

Complexity Analysis

- Given the unverifiability *a priori* of the number of nodes that can be eliminated, the exact improvement over naive minimax is hard to quantify.
- Worst case (*pessimal* move ordering):
  - children of MAX nodes are searched from least to greatest; children of MIN nodes are searched from greatest to least.
  - All nodes of the search tree will be visited, giving $O(b^d)$ time complexity: same as naive Minimax

Complexity Analysis

- Given the unverifiability *a priori* of the number of nodes that can be eliminated, the exact improvement over naive minimax is hard to quantify.
- Worst case (*pessimal* move ordering):
  - children of MAX nodes are searched from least to greatest; children of MIN nodes are searched from greatest to least.
  - All nodes of the search tree will be visited, giving $O(b^d)$ time complexity: same as naive Minimax
- Best case (*optimal* move ordering):
  - children of MAX nodes are searched from greatest to least; children of MIN nodes are searched from least to greatest.

Complexity Analysis

- Given the unverifiability *a priori* of the number of nodes that can be eliminated, the exact improvement over naive minimax is hard to quantify.
- Worst case (*pessimal* move ordering):
  - children of MAX nodes are searched from least to greatest; children of MIN nodes are searched from greatest to least.
  - All nodes of the search tree will be visited, giving $O(b^d)$ time complexity: same as naive Minimax
- Best case (*optimal* move ordering):
  - children of MAX nodes are searched from greatest to least; children of MIN nodes are searched from least to greatest.
  - AlphaBeta's time complexity is $O(b^{d/2})$; doubles the effective depth we could reach in the same amount of time!

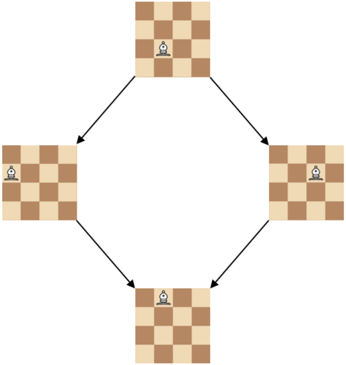
Table of Contents

Move Ordering

- As saw before, correct move ordering can be crucial for optimizing the routine.
- As we cannot know the best moves a priori, we can resort to some domain specific heuristics.
- The simplest type of ordering one can apply is to *rank* the moves according to some *importance* feature: pawn promotions, captures, checks, attacks...
- Further popular heuristics that have proven to be effective and not extremely costly:
  - History Heuristics
  - Killer Heuristics

Transposition Table

- In chess, as in many other cases, the same position can be reached in multiple different ways.
- This is highly inefficient as we end up evaluating the same position multiple times across our search routine.



Transposition Table

- To account for this, a Transposition Table can be implemented, allowing us to store visited positions as hashes, and returning the previously computed values directly from the hash map.
- Forsyth–Edwards Notation (FEN) strings implementation:
- Store fen strings as keys in a dictionary, with values being score and depth of current node.
- *rnbqkbnr/ppp1pppp/8/3p4/2PP4/8/PP2PPPP/RNBQKBNR b KQkq - 0 2*

Figure: Queen's Gambit Opening

Other Improvements

- The Horizon Effect and Quiescence Search
- Book Move Ordering
- End game behaviour and solved positions
- Improved Evaluation Function
- Iterative Deepening

Table of Contents

Implementation Details

- Language used: JavaScript
  - JS allows for interactions with the web page, and direct embedding in html documents that give graphical structure.
  - It handles both client and server side scripting, offering a seamless conjunction between back-end computations and front-end interface.
  - Similarly to python, JS is a very high-level OOP-language: lower abstraction languages would be preferred for the engine back-end to improve considerably the efficiency and performance of the AI.
  - Furthermore, it is versatile and widely used, offering a boundless amount of public libraries and packages.

Alphabeta Code

```
var minimax = function (game, depth, alpha, beta, isMaximisingPlayer) {
    positionCount++;
    if (depth === 0) {
        return -evaluateBoard(game.board());
    }


    var legalMoves = game.ugly_moves();
    moveSort(legalMoves);
    if (isMaximisingPlayer) {
        var bestMoveScore = -Infinity;
        for (var i = 0; i < legalMoves.length; i++) {
            game.ugly_move(legalMoves[i]);
            bestMoveScore = Math.max(bestMoveScore,
                minimax(game, depth - 1, alpha, beta, !isMaximisingPlayer));
            game.undo();
            alpha = Math.max(alpha, bestMoveScore);
            if (beta <= alpha) {
                break
            }
        }
        return bestMoveScore;
    }
    else {
        var bestMoveScore = Infinity;
        for (var i = 0; i < legalMoves.length; i++) {
            game.ugly_move(legalMoves[i]);
            bestMoveScore = Math.min(bestMoveScore,
                minimax(game, depth - 1, alpha, beta, !isMaximisingPlayer));
            game.undo();
            beta = Math.min(beta, bestMoveScore);
            if (beta <= alpha) {
                break
            }
        }
        return bestMoveScore;
    }
};
```

Transposition Table

```
var ttable = {}
var updateTtable = function(hash, score, depth) {
    if (!(hash in ttable)) {
        ttable[hash] = {}
    }
    ttable[hash].score = score
    ttable[hash].depth = depth
}
```

```
var minimax = function (game, depth, alpha, beta, isMaximisingPlayer) {
    positionCount++;
    if (depth === 0) {
        return -evaluateBoard(game.board());
    }
    // Transposition table, return previously computed score if
    // position already reached at same or higher depth than current
    let hash = game.fen()
    if (hash in ttable) {
        if (ttable[hash].depth >= depth) {
            hashedCount++
            return ttable[hash].score
        }
    }

    var legalMoves = game.ugly_moves();
    moveSort(legalMoves);
    if (isMaximisingPlayer) {
        var bestMoveScore = -Infinity;
        for (var i = 0; i < legalMoves.length; i++) {
            game.ugly_move(legalMoves[i]);
            bestMoveScore = Math.max(bestMoveScore,
                minimax(game, depth - 1, alpha, beta, !isMaximisingPlayer));
            game.undo();
            alpha = Math.max(alpha, bestMoveScore);
            if (beta <= alpha) {
                break
            }
        }
        updateTtable(hash, bestMoveScore, depth);
        return bestMoveScore;
    }
    else {
```

Move Ordering

```
// Move Ordering
var moveSort = function(movesList) {
    // Very simple move ordering logics.
    // flags are: 2 -> Capture, 16-> Promotion, 18 -> Promotion+Capture
    var importances = {
        18: 15,
        16: 10,
        2: 5
    }

    for (let move of movesList) {
        if (move.flags in importances) {
            move.importance = importances[move.flags]
        }
        else {
            move.importance = 0
        }
    }
    return movesList.sort((a,b) => b.importance - a.importance);
}
```

Evaluation Function

```
// Board Evaluation functions starts here
var evaluateBoard = function (board) {
    var totalEvaluation = 0;
    for (var i = 0; i < 8; i++) {
        for (var j = 0; j < 8; j++) {
            totalEvaluation = totalEvaluation + getPieceValue(board[i][j], i ,j);
        }
    }
    return totalEvaluation;
};
```

```
var weights = {
    'p': 100,
    'n': 320,
    'b': 330,
    'r': 500,
    'q': 900,
    'k':20000
}
```

```
var pst = {
    'p': [
        [ 0,   0,   0,   0,   0,   0,   0,   0],
        [50,  50,  50,  50,  50,  50,  50,  50],
        [10,  10,  20,  30,  30,  20,  10,  10],
        [ 5,   5,  10,  25,  25,  10,   5,   5],
        [ 0,   0,   0,  20,  20,   0,   0,   0],
        [ 5, - 5, -10,   0,   0, -10, - 5,   5],
        [ 5,  10,  10, -20, -20,  10,  10,   5],
        [ 0,   0,   0,   0,   0,   0,   0,   0]
    ],

    'n': [
        [-50, -40, -30, -30, -30, -30, -40, -50],
        [-40, -20,   0,   0,   0,   0, -20, -40],
        [-30,   0,  10,  15,  15,  10,   0, -30],
        [-30,   5,  15,  20,  20,  15,   5, -30],
        [-30,   0,  15,  20,  20,  15,   0, -30],
        [-30,   5,  10,  15,  15,  10,   5, -30],
        [-40, -20,   0,   5,   5,   0, -20, -40],
        [-50, -40, -30, -30, -30, -30, -40, -50]
    ],
```

```
var getPieceValue = function (piece, x, y) {
    if (piece === null) {
        return 0;
    }
    return piece.color === 'w' ?  weights[piece.type] + pst[piece.type][y][x]:
            -(weights[piece.type] + reverseArray(pst[piece.type])[y][x])
};
```

Conclusion

- nick.zhang@studbocconi.it
- Full source code available at:
  `https://github.com/FreeTheOtter/js-minimax-chessai`
- Also hosted online on Heroku and playable at:
  `nick-chess.herokuapp.com`