Chess Program thought process analysis

The Huo Chess example

Spyridon Kakos, Athens, Greece

Goal

The goal of this paper is to analyze the thought process of a chess program, so as to facilitate understanding among chess programming enthusiasts and to bolster potential improvements in that process. To that end, I will use the open-source Huo Chess program I have created. A more detailed analysis of the thinking mechanism is due to follow in a new article.

About Huo Chess

Huo Chess is the smallest open-source chess program that is created by me, Spiros Kakos, with education in mind from the first line of code. I started creating it simply because I did not know how to create a chess program. So I started experimenting and writing code on my own. The goal was to create the smallest playing chess program that could be used for people (me included) to understand the basics of chess programming.

The current version of Huo Chess exists in multiple programming languages. Namely, it can be found in:

- C# (.NET Framework)
- C# (.NET Core)
- Quick Basic (QB64)
- Java
- OBSOLETE - C++ (only older versions)
- OBSOLETE - Visual Basic (only older versions)

One can find the latest code for the Huo Chess in GitHub at the following repositories:

- Huo Chess C# edition (.NET Framework)
- Huo Chess C# edition (.NET Core)
- Huo Chess Quick Basic edition (QB64)
- Huo Chess Java edition

Please make a note to revisit these repositories often since the program is updated constantly with fixes or improvements.

Why Huo Chess?

As mentioned already, Huo Chess was made with education in mind from the very beginning. The code is as simple as possible and fully loaded with comments to make it understandable by anyone reading it.

This is more important than one might think. Programmers have the very bad habit of not using comments at all or of obscuring the code to such an extent that it is unreadable even by themselves years later.

Imagine reading the following code…

```
IF QMINIMAX% = 1 THEN
    IF EVAL% > QSCORE%(QPLY% - 1, 1) THEN
        QENGINE%(QPLY%, 0, 0) = QENGINE%(QPLY%, 1, 0)
    END IF
    IF QFIRSTPLY% = 1 THEN
        IF QENGINE%(1, 1, 0) > 1 THEN
            IF QENGINE%(QPLY% - 1, 1, 0) > 1 THEN
                IF QSCORE%(QPLY% - 1, 1) < QSCORE%(1, 1) THEN
                    QENGINE%(QPLY% - 1, 0, 0) = QENGINE%(QPLY% - 1, 1, 0)
                    QENGINE%(QPLY%, 0, 0) = QENGINE%(QPLY%, 1, 0)
```

```
                END IF
            END IF
        END IF
    END IF
END IF
```

Seems confusing right?

Now take a look at this…

```
FOR counter4 = 1 TO (NodeLevel_4_count - 1)

    IF (NodesAnalysis4(counter4, 2) <> parentNodeAnalyzed) THEN
        parentNodeAnalyzed = NodesAnalysis4(counter4, 2)
        'PRINT "counter4 = " + STR$(counter4)
        IF NodesAnalysis4(counter4, 2) = 0 THEN NodesAnalysis4(counter4, 2) = 1
        NodesAnalysis3(NodesAnalysis4(counter4, 2), 1) = NodesAnalysis4(counter4, 1)
        bestNode4 = counter4
        'bestNodes4(parentNodeAnalyzed) = counter4
    END IF

    ' v0.991: Original: >=
    ' v0.991: This should depend on the colour of the computer!!!
    ' v0.991: Tried to fix the problem in MinMax. Node1 elements for the SAME parent of Node2 must be filled accordingly.
```

```
    '         We do not need to take into account the Node2 elements which are empty, thus having a score of 0 but no
    '         assigned move! (this is why the Best Variant text is empty in the first moves)
    IF playerColor$ = "w" THEN
        IF (NodesAnalysis4(counter4, 1) <= NodesAnalysis3(NodesAnalysis4(counter4, 2), 1)) THEN
            IF NodesAnalysis4(counter4, 2) = 0 THEN NodesAnalysis4(counter4, 2) = 1
            NodesAnalysis3(NodesAnalysis4(counter4, 2), 1) = NodesAnalysis4(counter4, 1)
            bestNode4 = counter4
            'bestNodes4(parentNodeAnalyzed) = counter4
        END IF
    ELSEIF (playerColor$ = "b") THEN
        IF (NodesAnalysis4(counter4, 1) >= NodesAnalysis3(NodesAnalysis4(counter4, 2), 1)) THEN
            IF NodesAnalysis4(counter4, 2) = 0 THEN NodesAnalysis4(counter4, 2) = 1
            NodesAnalysis3(NodesAnalysis4(counter4, 2), 1) = NodesAnalysis4(counter4, 1)
            bestNode4 = counter4
            'bestNodes4(parentNodeAnalyzed) = counter4
        END IF
    END IF

NEXT counter4
```

Much better, right?

Note that having comments in the code does not make one understand the source code automatically or without effort. You still have to dig in the algorithm and spend time and effort to understand. But you will not be hindered by the lack of comments in the code or by the use of obscure variable names which you know what they are about.

Another good thing is that the variable names in Huo Chess are self-explanatory. No single-letter weird variable names! This helps you even more understand what is happening, even though it does cost in terms of size.

Last but certainly not least, Huo Chess comes with tons of online tutorials on how it works that you can consult to boost your understanding. Look out for them in Harmonia Philosophica portal or in the dedicated chess programming portal Chess-Programming.com.

Important Note: For the needs of this article, we will use the QBasic code for the examples used. However, remember that Huo Chess is available also in C# and Java and you can just go to the source code of these editions and look for the same code in these languages. The variables and the function names are the same in all editions.

Chess program overview

At first, when designing a chess program (or a program of any kind), one must sketch the outline of the program algorithm.

Without unnecessary adieu, this is something like the schema below.

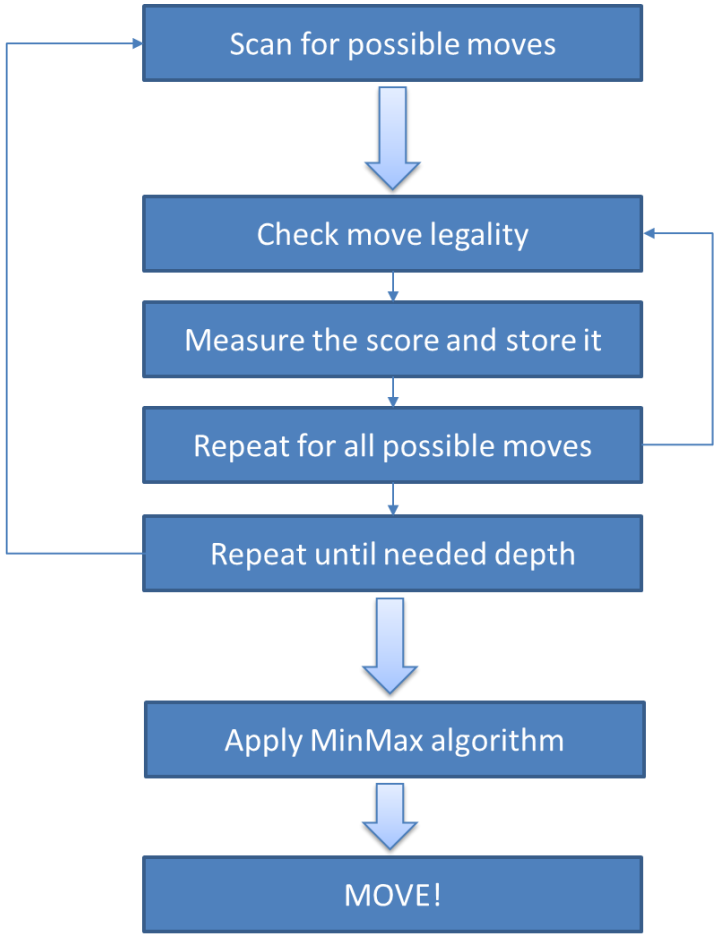


Figure 1: Chess program overall structure (simple version)

The chess program essentially performs an iterative process to find out what the best move is…

1. Scan the chessboard: Find all pieces of the computer and all possible moves.
2. For each move found, check its legality and, if legal, measure the score after the move and store it.
3. Apply the same process for all moves.
4. Repeat the two previous steps for each move depth until reaching the desire thinking depth.
5. After all moves in the desired depth are found, scored and stored, apply the MiniMax algorithm to find the best move.

What are the basic components of this thinking process?

Regardless of how complicated people are trying to portray that process, in essence ONE the major step in the process. And that is the part of the code which determines whether a potential move is legal or not!

Scanning…

Before we go into the checking of the legality of the move, we need a small step before that. That is to scan the chessboard, locate the pieces of the computer and then 'generate' all possible moves. The code does that by using two nested 'for' loops: The outer one scans the chessboard and, for every piece of the computer found, the inner loop investigates all possible target squares.

Check the computerMove function of the Huo Chess code to see how this is done.

```
'Scan the chessboard...
FOR i = 1 TO 8
    FOR j = 1 TO 8

        'If you find a piece of the computer...
        IF ((MID$(chessboard$(i, j), 1, 1) = "w" AND playerColor$ = "b") OR (MID$(chessboard$(i, j), 1, 1) = "b" AND playerColor$
= "w")) THEN
```

```
        'Scan all possible destination squares...
        FOR ii = 1 TO 8
              FOR jj = 1 TO 8

                  startingColumn = i
                  startingRank = j
                  finishingColumn = ii
                  finishingRank = jj

                  MovingPiece$ = chessboard$(i, j)
                  ProsorinoKommati$ = chessboard$(ii, jj)

              …

              NEXT jj
        NEXT ii

        NEXT j
NEXT i
```

Note how the code is written totally against any good practices (e.g. all those string comparisons are terrible for performance), yet the goal is to use the program for educational purposes as already emphasized.

But how do we check the legality of a move?

Check the legality of the move

Let me explain how important the checking of the legality of the move is.

Imagine that you have a ready function that is fed with a move and then returns as a result the conclusion on whether this move is legal or not based on a specific chessboard position. Can you imagine what you would do after that point?

Easy, isn't it?

You would 'just' have to filter all the possible moves analyzed and find out which one is the best! Surely this is easier said than done. But again, we are talking conceptually here. Writing actual code that does the work is always more difficult than laying down the overall plan.

So, this is the first thing we need to do: Find how a move can be checked for legality.

Please check the ElegxosNomimotitas function of Huo Chess on how this check is performed ('Nomimotita' means legality in Greek, while 'Elegxos' means 'checking'). This function is fed with a chessboard position and a move (i.e. a starting column, a starting rank and a finishing column and a finishing rank) and it gives back as a result a 0 or 1 according to the legality of the move in question.

For each piece there are different rules that are used, e.g. the knight can only move in specific ways, the bishop can only move in diagonals and so on. There are also additional rules applicable; for example, one cannot move to a square if there is another piece (of the same or different colour) between the starting and the finishing square.

```
' ---------------------------------- BISHOP ----------------------------------


IF (MovingPieceEN$ = "wbishop" OR MovingPieceEN$ = "bbishop") THEN
```

```
    'Check correctness of move (Bishop only moves in diagonals)

    IF ((startRank = finishRank) OR (startColumn = finishColumn)) THEN NomimotitaEN = 0
    IF (ABS(startColumn - finishColumn) <> ABS(startRank - finishRank)) THEN NomimotitaEN = 0
```

I will not go into the details of the code. I am sure there are much more efficient ways of checking the legality of the move than the Huo Chess code, however, please note that the code of Huo Chess is written with human comprehension as one of its main goals. Not performance.

Make the move… Score the move…

Each legal move is performed and then the score of the position after the move is measured and stored.

```
'If move is legal, then do the move and present it in the chessbooard
IF Nomimotita = 1 THEN

    '-------------- MinMax --------------
    'Important note: All dimensions in the tables are +1 compared to the Huo Chess in C#! (tables start from 1 here)

     NodesAnalysis0(NodeLevel_0_count, 3) = startingColumn
     NodesAnalysis0(NodeLevel_0_count, 4) = finishingColumn
     NodesAnalysis0(NodeLevel_0_count, 5) = startingRank
     NodesAnalysis0(NodeLevel_0_count, 6) = finishingRank
```

```
'Do the move
chessboard$(finishingColumn, finishingRank) = chessboard$(startingColumn, startingRank)
chessboard$(startingColumn, startingRank) = ""

'Count the score of the move
CALL countScore(chessboard$())

IF debugMode = 1 THEN
    PRINT ""
    PRINT "NodeLevel_0_count = " + STR$(NodeLevel_0_count)
    PRINT "NodeLevel_1_count = " + STR$(NodeLevel_1_count)
END IF

'-------------- MinMax --------------
'Store scores
NodesAnalysis0(NodeLevel_0_count, 1) = positionScore
' Store parents
NodesAnalysis0(NodeLevel_0_count, 2) = 0
'Store the move
'NodesMoves0$(NodeLevel_0_count)=STR$(startingColumn)+STR$(startingRank)+" -> "+STR$(finishingColumn)+STR$(finishingRank)
```

Remember that the way we count the score of a position is of immense importance. Yet, in this short article the details of the countScore function (no need to say what this function does right? That is the good thing of using good names in variables and functions…) will not be addressed.

Important Note: The move and its score are stored in nodes, i.e. in special arrays. These nodes will then be used for the MiniMax algorithm.

Going deeper…

The process described above (scanning for pieces and possible moves and measuring their score) is a process that covers the initial chessboard position. If we are to think deeper, we need to perform the same exactly process for the position resulting after the first move, then for the position resulting after this move and so on.

To that end, at each level of thinking the code checks to see if the desired thinking depth is reached. If not, then the function that analyzes the moves for the next thinking level is called. The relative functions have names that can easily distinguish them.

```
IF Move < thinkingDepth THEN
    Move = Move + 1
    IF playerColor$ = "b" THEN whichColorPlays$ = "Black" '14/3/2021
    IF playerColor$ = "w" THEN whichColorPlays$ = "White" '14/3/2021
    CALL HumanMove1(chessboard$())
END IF
```

So after the computerMove function we call the humanMove1 function. It is called like that because it analyzes all possible moves for the first move of the human opponents – his answers to the moves of the computer.

After humanMove1 we have… computerMove2, humanMove3 and computerMove4, thus making the program able to search and analyze moves for a depth of 5 half-moves (the first moves analyzed with the computerMove are considered ‘move-zero’).

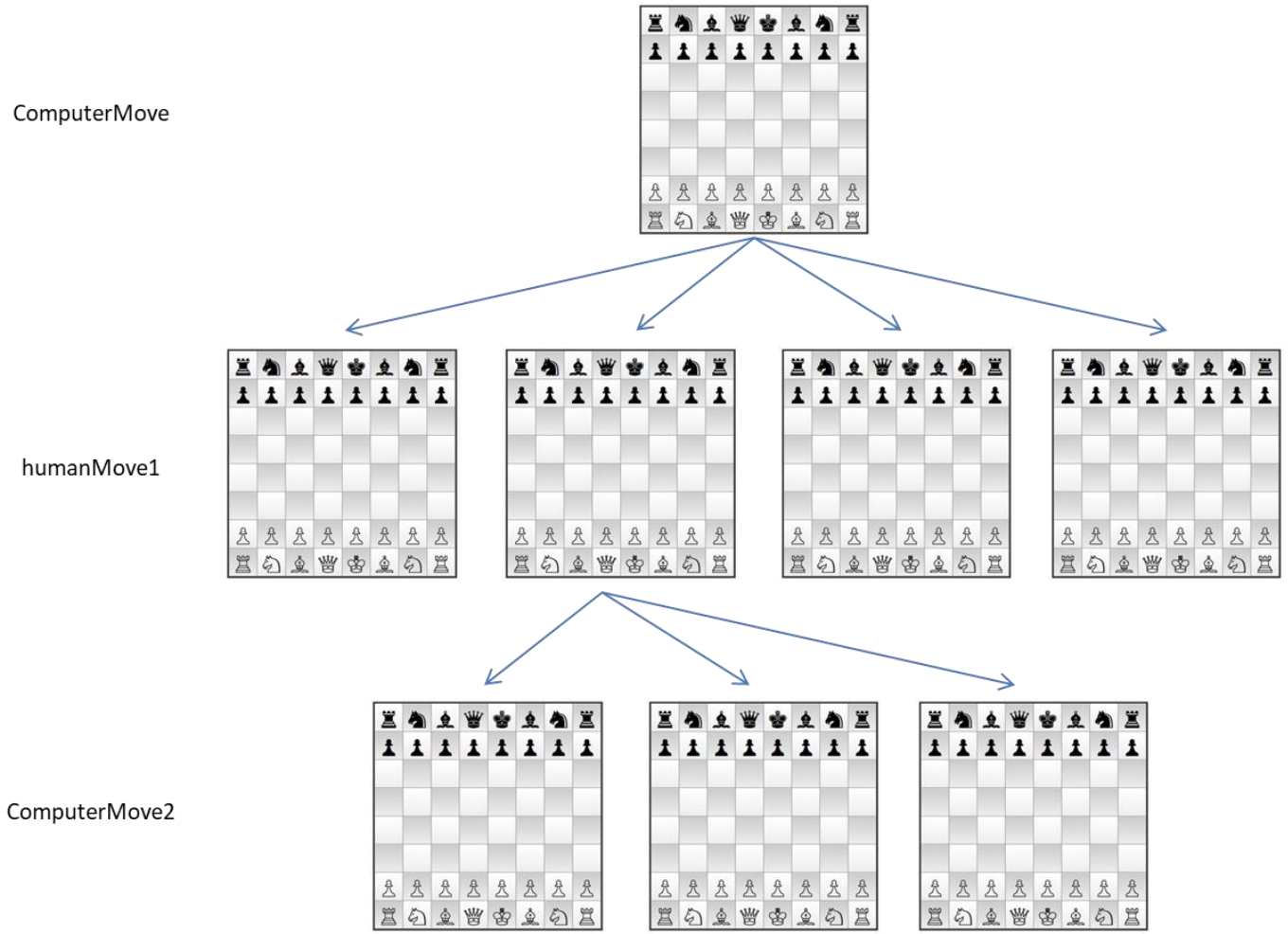


Figure 2: Thinking levels of Huo Chess

Important Note: This article provides a high-level overview of the way the chess program is working. It does not explain all the details in the source code. As you will notice for example, the code calling humanMove1 also changes the colour of the player being analyzed, because at the humanMove1 the code analyzes the moves not of the computer but of the human opponent who plays with the other colour.

I found all the legal moves! Now what?!

Well, after we have found all the possible legal moves in a position, we need to evaluate them. The simplest evaluation is the most dummy one: Just measure the score of the position after each move and then do the move with the best score!

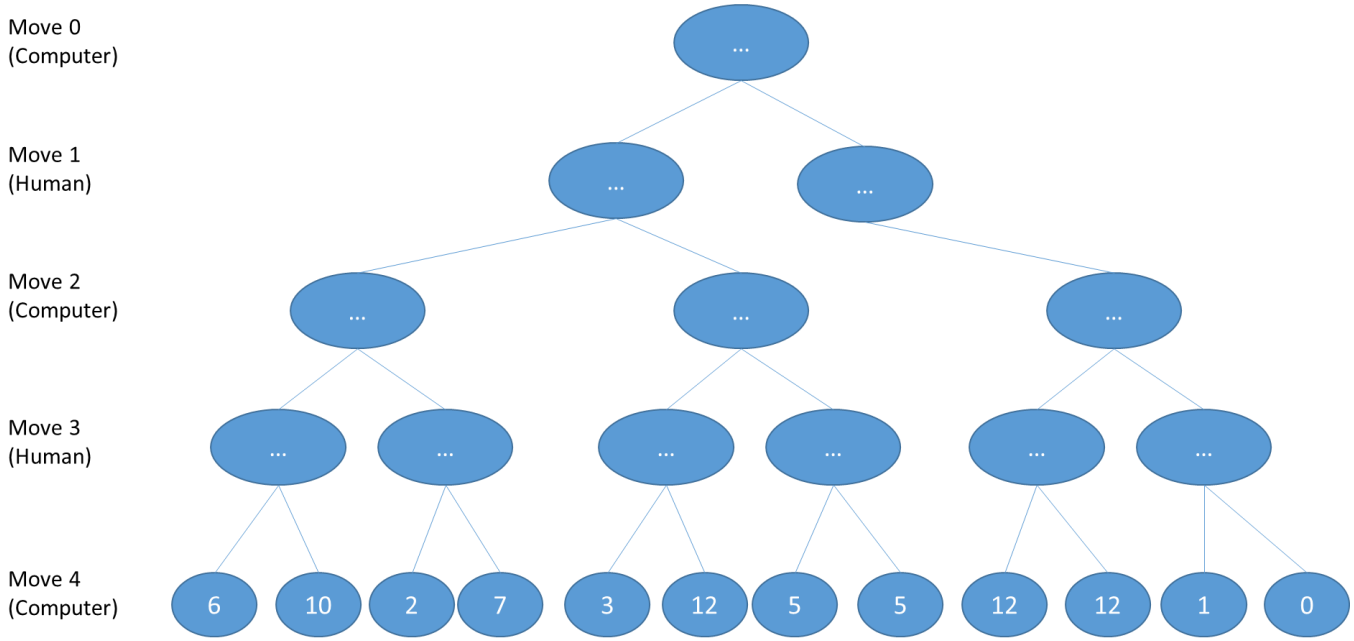
But we will do something more elaborate than that. We will apply the MiniMax algorithm.

MiniMax is a decision rule used in artificial intelligence, decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. When dealing with gains, it is referred to as "maximin" – to maximize the minimum gain (source).

In simple words, the algorithm tries to find the MAXIMUM (best) move for the computer but while at the same time considering the best moves of the human opponent (i.e. the human opponent moves that result in the MINIMUM gains for the computer).

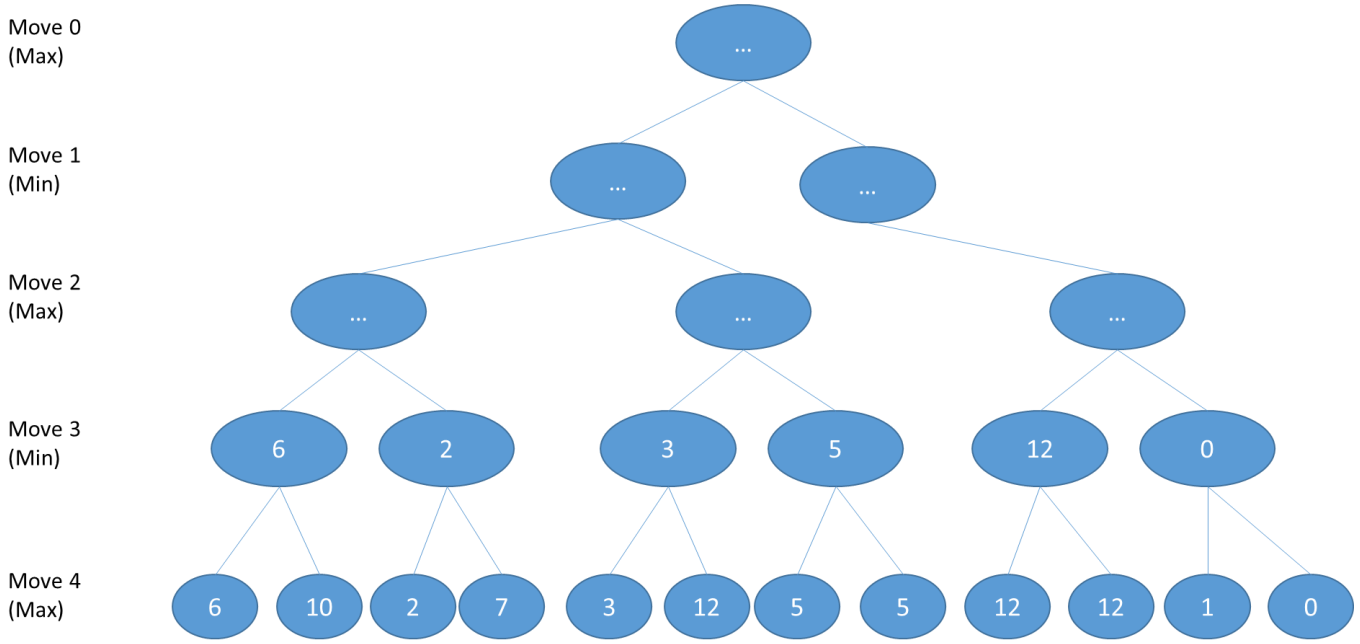
Let us see how this works…

First, the program has explored all the possible moves and countermoves and has built the moves tree. At the end of the analysis (where the analysis depth is reached) the score has been counted.



Starting from the 'deeper' level of thinking (i.e. depth 4), the algorithm calls for propagating the values in that level back to level 3 by choosing the minimum number (score) for each leaf of the tree.

Why is that? Simply because when it comes to the human opponent, he/she would prefer to choose the path which results in the minimum (least) gains for the computer (or else the maximum gains for the human).

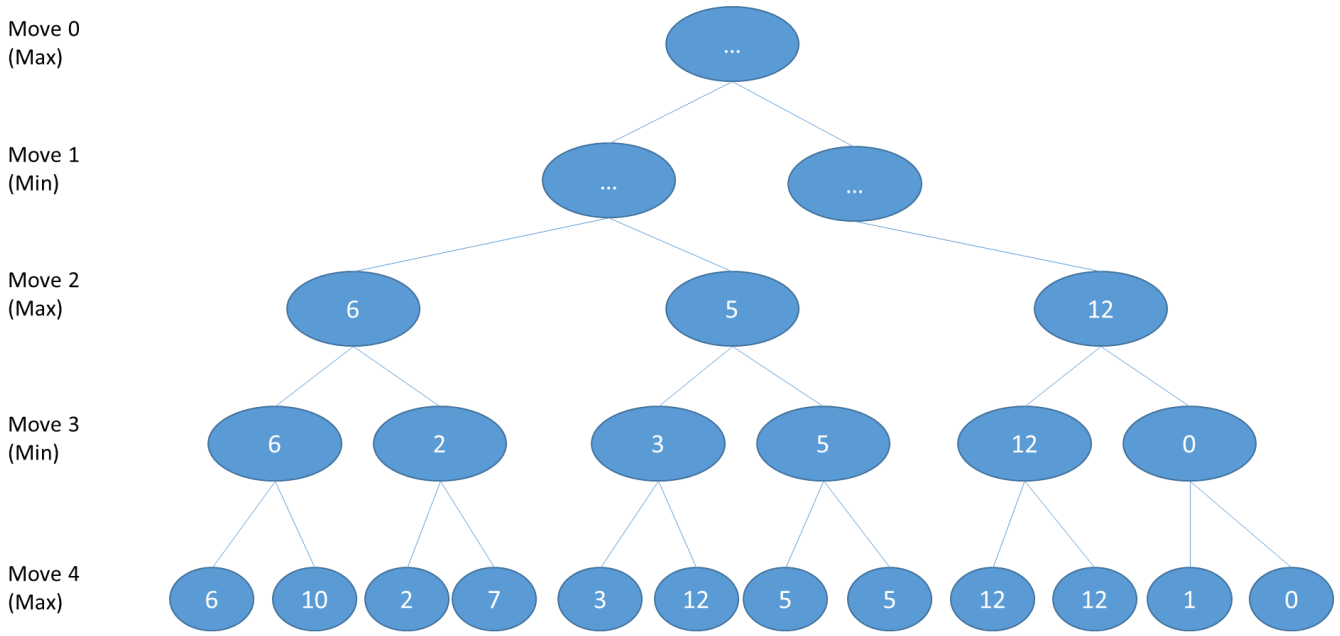


Then in the next level the values of that level are propagated back to the upper level by doing the opposite: selecting the maximum value of each leaf.

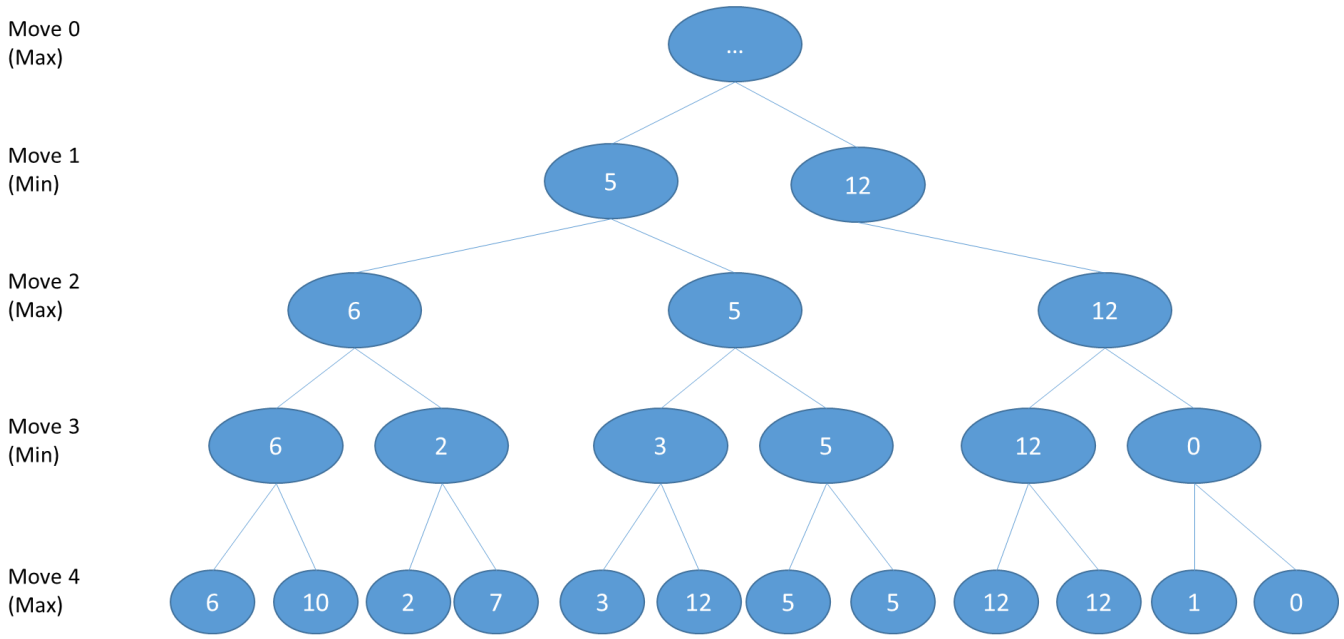
Why is that? Again, simply because when it comes to the computer, it has to choose the path with the maximum (most) gains, or in other words the minimum gains for the human opponent.

This process is repeated until we reach the ‘root’ of the tree.

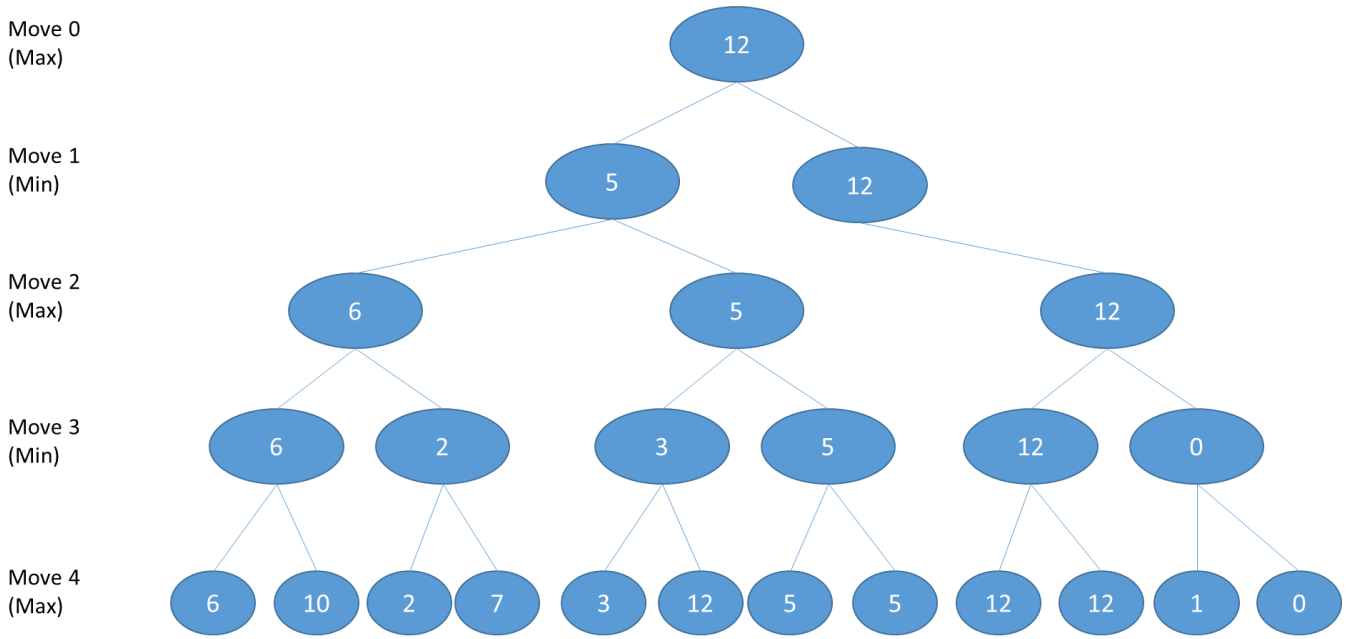
At Move 2 level we choose for the maximum…



At Move 1 level we seek the minimum again…



And finally, at Move 0 (the root) we seek the maximum…



The move that corresponds to the node selection at the root level is the move the computer will make!

Where can I see more details

Please refer to the “APPENDIX - The code” to see more details on the source code of Huo Chess and how it works.

Epilogue

This is a short article trying to show the basic steps of a chess program thinking process. Read this to get the high-level picture of a chess program, but do not stop here. There are many more things needed to make a working chess program. This article has just started to scratch the surface. The next article will be more detailed in many aspects and will cover the scoring function, the way the computer trims the thinking tree for performance reasons and other advanced subjects needed for a modern chess program.

Keep coding!

Keep experimenting!

APPENDIX - The code

The main sections of the code implementing all the above, along with some comments is presented in this section. Note that this code is based on the Huo Chess v0.970 version. Some things have changed since then, however the main lines of code and the main logic remains the same.

If human plays first => EnterMove
(else ComputerMove is called directly from Main_Console())

For the move entered by the human opponent…

- Check legality of the move
- Check for mate
- Check if there is check active
- Store move's coordinates
- Store the value of the piece human moves
- Store the coordinates of where that piece moved [Human_last_move_target_rank/column]

ComputerMove [Move_Analyzed = 0]

```
#region InitializeNodes       //Initialize all nodes
#region StoreInitialPosition  //Store initial position
#region OpeningBookCheck      //OPENING BOOK CHECK
#region DangerousSquares      //CHECK FOR DANGEROUS SQUARES

// Initialize variables (POTENTIALLY NOT NEEDED!)

For each possible move
{
       // Check if the move is stupid
       #region CheckStupidMove

              If Move < 5 and you move the knight to the edges => Stupid move etc…
              + Store moving piece's value
```

```
#endregion CheckStupidMove

If not stupid & Destination square not dangerous => ...
if ((ThisIsStupidMove.CompareTo("N") == 0) && (Skakiera_Dangerous_Squares[(m_FinishingColumnNumber - 1), (m_FinishingRank -
1)] == 0))...

Call CheckMove to check validity of the move
CheckMove(Skakiera_Thinking, m_StartingRank, m_StartingColumnNumber, m_FinishingRank, m_FinishingColumnNumber, MovingPiece);

<Call CheckMove(Skakiera_Thinking)> to:
    ➢ Check legality of the move
    ➢ Check for mate (This is not done! Must add it!)
    ➢ Check if there is check active
    ➢ ~~Store move's coordinates~~ (REMOVED! This is done in ComputerMove!)
    ➢ ~~Store moving piece's value~~ (This is not done! This is done in ComputerMove!)

// If move analyzed in the first: Store move to ***_HY variables (so as to keep the initial move somewhere)
if (((m_OrthotitaKinisis == true) && (m_NomimotitaKinisis == true)) && (Move_Analyzed == 0)) => ...

if ((m_OrthotitaKinisis == true) && (m_NomimotitaKinisis == true))

// Do the move
ProsorinoKommati = Skakiera_Thinking[(m_FinishingColumnNumber - 1), (m_FinishingRank - 1)];
Skakiera_Thinking[(m_StartingColumnNumber - 1), (m_StartingRank - 1)] = "";
Skakiera_Thinking[(m_FinishingColumnNumber - 1), (m_FinishingRank - 1)] = MovingPiece;

Check the score after the computer move

NodeLevel_0_count++;
Temp_Score_Move_0 = CountScore(Skakiera_Thinking, humanDangerParameter);

// Store the best move score at this level
if ((m_PlayerColor.CompareTo("Black") == 0) && (Temp_Score_Move_0 > bestScoreLevel0))
{
       bestScoreLevel0 = Temp_Score_Move_0;
}
```

Document...

- ValueOfKommati
- ValueOfTargetPiece

⇨ Check possibility to eat back

```
// v0.970: Check if you can eat back the piece of the human which moved!
if ((m_FinishingColumnNumber == Human_last_move_target_column)
     && (m_FinishingRank == Human_last_move_target_row)
     && (ValueOfMovingPiece <= ValueOfHumanMovingPiece))
{
    Best_Move_StartingColumnNumber = m_StartingColumnNumber;
    Best_Move_StartingRank = m_StartingRank;
    Best_Move_FinishingColumnNumber = m_FinishingColumnNumber;
    Best_Move_FinishingRank = m_FinishingRank;

    possibility_to_eat_back = true;
}
```

⇨ Check possibility to eat

If thinking depth not reached, call next level of analysis

```
// v0.970: If you can eat back the piece of the human, then go for it and don't analyze!
if ((Move_Analyzed < Thinking_Depth) && (possibility_to_eat_back == false) && (possibility_to_eat == false))
{
    Move_Analyzed = Move_Analyzed + 1;

    for (i = 0; i <= 7; i++)
    {
        for (j = 0; j <= 7; j++)
        {
            Skakiera_Move_After[(i), (j)] = Skakiera_Thinking[(i), (j)];
        }
    }
```

```
            Who_Is_Analyzed = "Human";
            //First_Call_Human_Thought = true;

            // Check human move (to find the best possible answer of the human
            // to the move currently analyzed by the HY Thought process)
            if (Move_Analyzed == 1)
                Analyze_Move_1_HumanMove(Skakiera_Move_After);
            else if (Move_Analyzed == 3)
                Analyze_Move_3_HumanMove(Skakiera_Move_After);
            else if (Move_Analyzed == 5)
                Analyze_Move_5_HumanMove(Skakiera_Move_After);
        }

        // Undo the move
        Skakiera_Thinking[(m_StartingColumnNumber0 - 1), (m_StartingRank0 - 1)] = MovingPiece0;
        Skakiera_Thinking[(m_FinishingColumnNumber0 - 1), (m_FinishingRank0 - 1)] = ProsorinoKommati0;
}

...

}
```

Analyze_Move_1_HumanMove [Move_Analyzed = 1]

```
Check all possible moves
{
       if ((m_OrthotitaKinisis == true) && (m_NomimotitaKinisis == true)) then...

       // Do the move
       ProsorinoKommati = Skakiera_Human_Thinking_2[(m_FinishingColumnNumber - 1), (m_FinishingRank - 1)];
       Skakiera_Human_Thinking_2[(m_StartingColumnNumber - 1), (m_StartingRank - 1)] = "";
       Skakiera_Human_Thinking_2[(m_FinishingColumnNumber - 1), (m_FinishingRank - 1)] = MovingPiece;

           // Measure score AFTER the move
       NodeLevel_1_count++;
       Temp_Score_Move_1_human = CountScore(Skakiera_Human_Thinking_2, humanDangerParameter);

       // Store the best move at this level
       if ((m_PlayerColor.CompareTo("Black") == 0) && (Temp_Score_Move_1_human < bestScoreLevel1))
       {
          bestScoreLevel1 = Temp_Score_Move_1_human;
       }

       If thinking depth not reached, call next level of analysis

       if (Move_Analyzed < Thinking_Depth) && (Temp_Score_Move_1_human better than bestScoreLevel1)
       {
           Move_Analyzed = Move_Analyzed + 1;
           Who_Is_Analyzed = "HY";

           if (Move_Analyzed == 2)
               Analyze_Move_2_ComputerMove(Skakiera_Move_After);
           else if (Move_Analyzed == 4)
               Analyze_Move_4_ComputerMove(Skakiera_Move_After);
           else if (Move_Analyzed == 6)
               Analyze_Move_6_ComputerMove(Skakiera_Move_After);
       }

       // Undo the move
```

```
        Skakiera_Human_Thinking_2[(m_StartingColumnNumber1 - 1), (m_StartingRank1 - 1)] = MovingPiece1;
        Skakiera_Human_Thinking_2[(m_FinishingColumnNumber1 - 1), (m_FinishingRank1 - 1)] = ProsorinoKommati1;
}

Move_Analyzed = Move_Analyzed - 1;
Who_Is_Analyzed = "HY";
```

Analyze_Move_2_ComputerMove [Move_Analyzed = 2]

```
Check all possible moves
{
       if ((m_OrthotitaKinisis == true) && (m_NomimotitaKinisis == true))
       {
       // huo_sw1.WriteLine(string.Concat("Human move 1: Found a legal move!"));

       // Do the move
       ProsorinoKommati = Skakiera_Thinking_HY_2[(m_FinishingColumnNumber - 1), (m_FinishingRank - 1)];
       Skakiera_Thinking_HY_2[(m_StartingColumnNumber - 1), (m_StartingRank - 1)] = "";
       Skakiera_Thinking_HY_2[(m_FinishingColumnNumber - 1), (m_FinishingRank - 1)] = MovingPiece;

       // Check the score after the computer move.
       if (Move_Analyzed == 0)
           NodeLevel_0_count++;
           Temp_Score_Move_0 = CountScore(Skakiera_Thinking_HY_2, humanDangerParameter);
       if (Move_Analyzed == 2)
           NodeLevel_2_count++;
           Temp_Score_Move_2 = CountScore(Skakiera_Thinking_HY_2, humanDangerParameter);

       // Store the best score at this level
       if ((m_PlayerColor.CompareTo("Black") == 0) && (Temp_Score_Move_2 > bestScoreLevel2))
       {
              bestScoreLevel2 = Temp_Score_Move_2;
       }

       If thinking depth not reached, call next level of analysis

       if (Move_Analyzed < Thinking_Depth)
       {
           Move_Analyzed = Move_Analyzed + 1;
           Who_Is_Analyzed = "Human";
           First_Call_Human_Thought = true;    < WHAT IS THIS???

           // Check human move
           if (Move_Analyzed == 1)
```

```
                Analyze_Move_1_HumanMove(Skakiera_Move_After);
            else if (Move_Analyzed == 3)
                Analyze_Move_3_HumanMove(Skakiera_Move_After);
            else if (Move_Analyzed == 5)
                Analyze_Move_5_HumanMove(Skakiera_Move_After);
        }

        if (Move_Analyzed == Thinking_Depth)
        {
        // [MiniMax algorithm - skakos]
        // Record the node in the Nodes Analysis array (to use with MiniMax algorithm) skakos
        NodesAnalysis0[NodeLevel_0_count, 0] = Temp_Score_Move_0;
        NodesAnalysis1[NodeLevel_1_count, 0] = Temp_Score_Move_1_human;
        NodesAnalysis2[NodeLevel_2_count, 0] = Temp_Score_Move_2;
        NodesAnalysis3[NodeLevel_3_count, 0] = Temp_Score_Move_3_human;

        // Store the parents (number of the node of the upper level)
        NodesAnalysis0[NodeLevel_0_count, 1] = 0;
        NodesAnalysis1[NodeLevel_1_count, 1] = NodeLevel_0_count;
        NodesAnalysis2[NodeLevel_2_count, 1] = NodeLevel_1_count;
        NodesAnalysis3[NodeLevel_3_count, 1] = NodeLevel_2_count;
        }

        => Because the analysis ends only in Analyze_Move_2_ComputerMove functions, the ThinkigDepth must be an even number!

        // Undo the move
        Skakiera_Thinking_HY_2[(m_StartingColumnNumber2 - 1), (m_StartingRank2 - 1)] = MovingPiece2;
        Skakiera_Thinking_HY_2[(m_FinishingColumnNumber2 - 1), (m_FinishingRank2 - 1)] = ProsorinoKommati2;
}

Move_Analyzed = Move_Analyzed - 1;
Who_Is_Analyzed = "Human";
```

ComputerMove [Continued... - The End of Analysis]

Check for mate

```
// DO THE BEST MOVE FOUND: Use MiniMax algorithm

if (possibility_to_eat_back == false)
{
    // [MiniMax algorithm - skakos]
    // Find node 1 move with the best score via the MiniMax algorithm.
    int counter0, counter1, counter2;

    // ------------------------------------------------------
    // NodesAnalysis
    // ------------------------------------------------------
    // Nodes structure...
    // [ccc, xxx, 0]: Score of node No. ccc at level xxx
    // [ccc, xxx, 1]: Parent of node No. ccc at level xxx-1
    // ------------------------------------------------------

    int parentNodeAnalyzed = -999;

    //v0.980: Remove
    //parentNodeAnalyzed = -999;

    for (counter2 = 1; counter2 <= NodeLevel_2_count; counter2++)
    {
        if (Int32.Parse(NodesAnalysis2[counter2, 1].ToString()) != parentNodeAnalyzed)
        {
            //parentNodeAnalyzedchanged = true;
            parentNodeAnalyzed = Int32.Parse(NodesAnalysis2[counter2, 1].ToString());
            NodesAnalysis1[Int32.Parse(NodesAnalysis2[counter2, 1].ToString()), 0] = NodesAnalysis2[counter2, 0];
        }

        if (NodesAnalysis2[counter2, 0] >= NodesAnalysis1[Int32.Parse(NodesAnalysis2[counter2, 1].ToString()), 0])
            NodesAnalysis1[Int32.Parse(NodesAnalysis2[counter2, 1].ToString()), 0] = NodesAnalysis2[counter2, 0];
    }
```

```
// Now the Node1 level is filled with the score data

parentNodeAnalyzed = -999;

for (counter1 = 1; counter1 <= NodeLevel_1_count; counter1++)
{
    if (Int32.Parse(NodesAnalysis1[counter1, 1].ToString()) != parentNodeAnalyzed)
    {
        //parentNodeAnalyzedchanged = true;
        parentNodeAnalyzed = Int32.Parse(NodesAnalysis1[counter1, 1].ToString());
        NodesAnalysis0[Int32.Parse(NodesAnalysis1[counter1, 1].ToString()), 0] = NodesAnalysis1[counter1, 0];
    }

    if (NodesAnalysis1[counter1, 0] <= NodesAnalysis0[Int32.Parse(NodesAnalysis1[counter1, 1].ToString()), 0])
        NodesAnalysis0[Int32.Parse(NodesAnalysis1[counter1, 1].ToString()), 0] = NodesAnalysis1[counter1, 0];
}

// Choose the biggest score at the Node0 level
// Check example at http://en.wikipedia.org/wiki/Minimax#Example_2

// Initialize the score with the first score and move found
double temp_score = NodesAnalysis0[1, 0];
Best_Move_StartingColumnNumber = Int32.Parse(NodesAnalysis0[1, 2].ToString());
Best_Move_StartingRank = Int32.Parse(NodesAnalysis0[1, 4].ToString());
Best_Move_FinishingColumnNumber = Int32.Parse(NodesAnalysis0[1, 3].ToString());
Best_Move_FinishingRank = Int32.Parse(NodesAnalysis0[1, 5].ToString());

for (counter0 = 1; counter0 <= NodeLevel_0_count; counter0++)
{
    if (NodesAnalysis0[counter0, 0] > temp_score)
    {
        temp_score = NodesAnalysis0[counter0, 0];

        Best_Move_StartingColumnNumber = Int32.Parse(NodesAnalysis0[counter0, 2].ToString());
        Best_Move_StartingRank = Int32.Parse(NodesAnalysis0[counter0, 4].ToString());
        Best_Move_FinishingColumnNumber = Int32.Parse(NodesAnalysis0[counter0, 3].ToString());
```

```
            Best_Move_FinishingRank = Int32.Parse(NodesAnalysis0[counter0, 5].ToString());
        }
    }

}
```

If no move found => Resign