MiniMax implementation with look-ahead and α/β pruning in Java

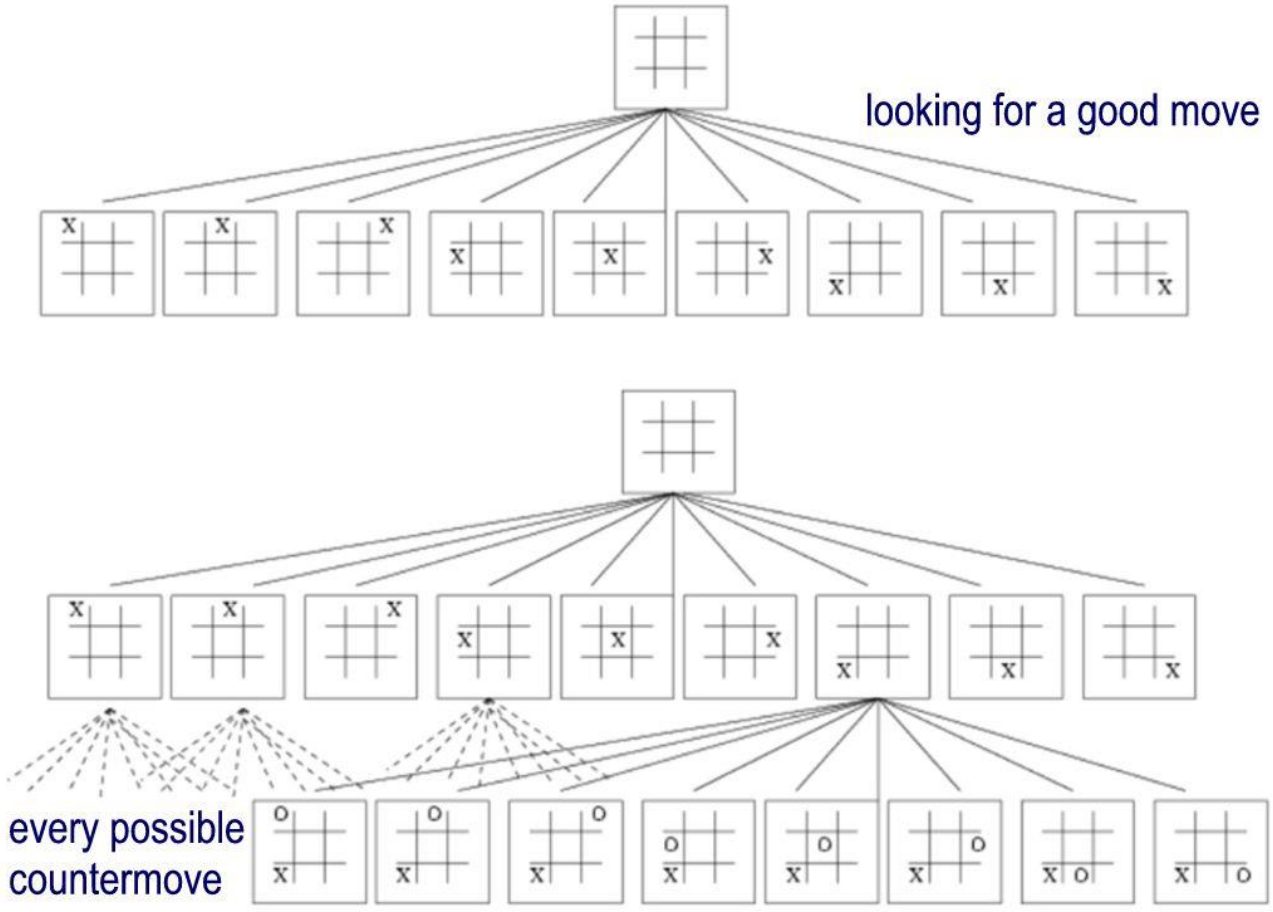
Two words on the theory

MiniMax is a decision rule used in decision theory, games theory, statistics, economics and philosophy for minimizing the possible loss while maximizing the potential gain.

- The MiniMax theorem was first published in 1928 by John von Neumann and states: for every two person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that:

1. Given player 2's strategy, the best payoff possible for player 1 is V.
2. Given player 1's strategy, the best payoff possible for player 2 is –V.

Equivalently, Player 1's strategy guarantees him a payoff of V regardless of Player 2's strategy, and similarly Player 2 can guarantee himself a payoff of −V. The name MiniMax arises because each player minimizes the maximum payoff possible for the other—since the game is zero-sum, he also minimizes his own maximum loss (i.e. maximize his minimum payoff).
Games are an important test bench for heuristic algorithm.
The MiniMax algorithm is applied in two players games, such as tic-tac-toe, chess and so on.
All these games have at least one thing in common, they are logic games.
Each player knows everything about the possible moves of the adversary.
Assume that we want to play tic-tac-toe, and we are looking for the best move to play.
For each possible our move we evaluate every possible countermove.



This is a very simple game but even if we play chess would attempt to do the same: unfortunately, if we are not a chess champion, after two or three moves ahead, the tree of variants becomes so great that not only is no longer possible to represent it on paper, but we are no longer able to move forward.

One thing is certain: if a computer (instead of a human) is able to expand the whole tree, it will certainly see all the advantageous moves and all the disadvantageous both.

If a software says to the computer how to make next move from any position, and when reached the end of each variant is able to assign a score, a computer can theoretically look for the best strategy for any zero-sum game or related problem.

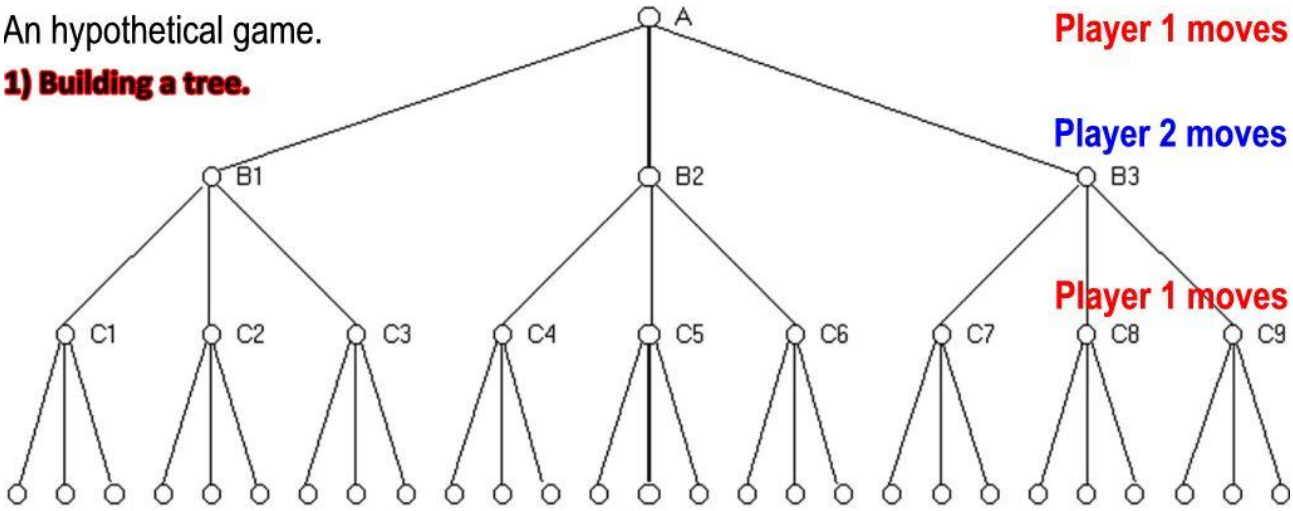
We have said that the computer needed:

1. ability to list all possible moves (**building a tree**);
2. ability to assign a rating to any position reached (**positional evaluation**).

Although these two ability are different from problem to problem, for the rest of the algorithm MiniMax will always be able to find the best move to play for each player.

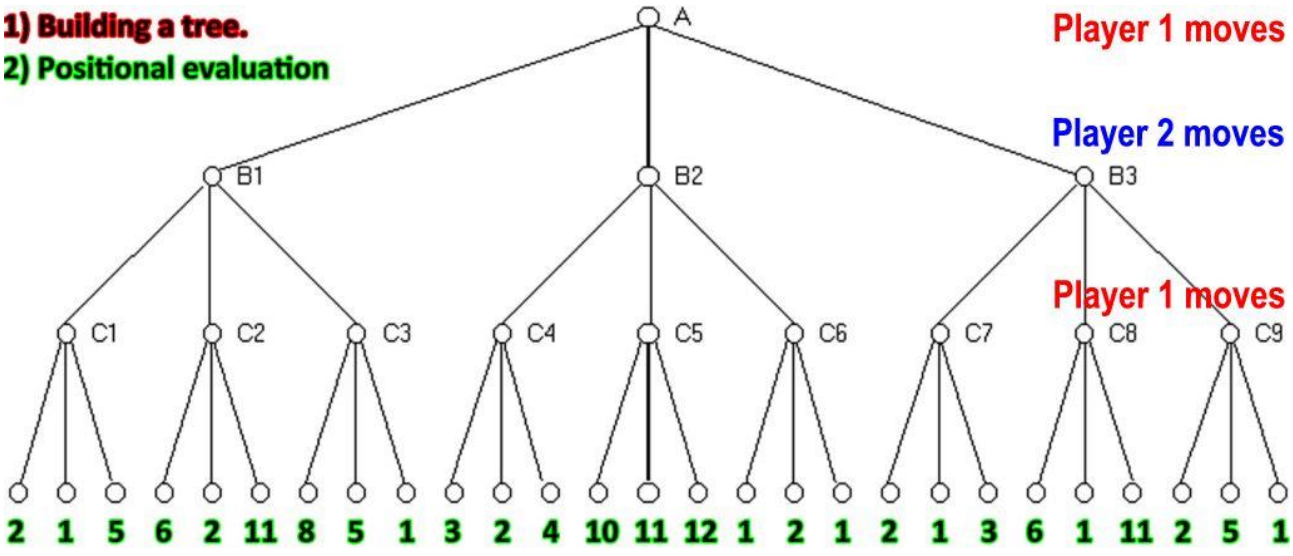
To better understand how we imagine to play a hypothetical game.

The first stage, starting from an initial position, the computer has to build a tree with all the possible variants.



In the construction of the tree, the computer knows of course when player 1 moves or player 2 does.

The second phase, for each position reached (terminal node or leaf of the tree), the computer must assign a score.



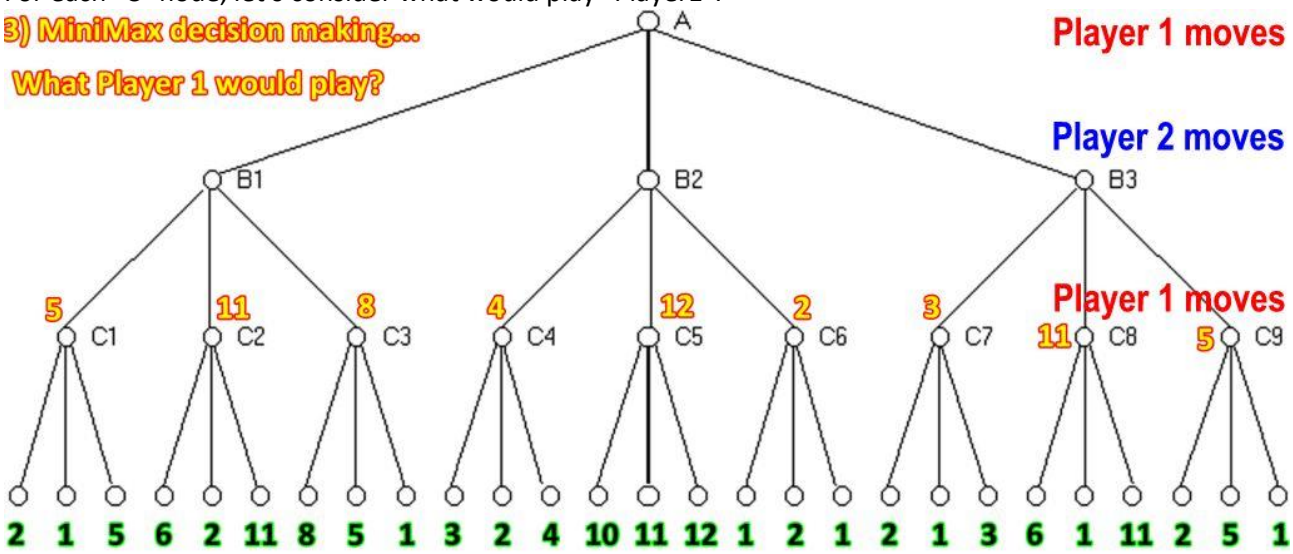
In the above example, assuming normally alternating turns, if the computer has the choice at level A (computer is Player 1), its opponent will have the choice at B, and the computer will again have the choice at C.

Due to the nature of these games (to which you apply the MiniMax theorem), if the goal of the Player 1 is to maximize the score, the goal of Player 2 is to minimize it. For this reason, player 1 is called Max and player 2 is called Min.

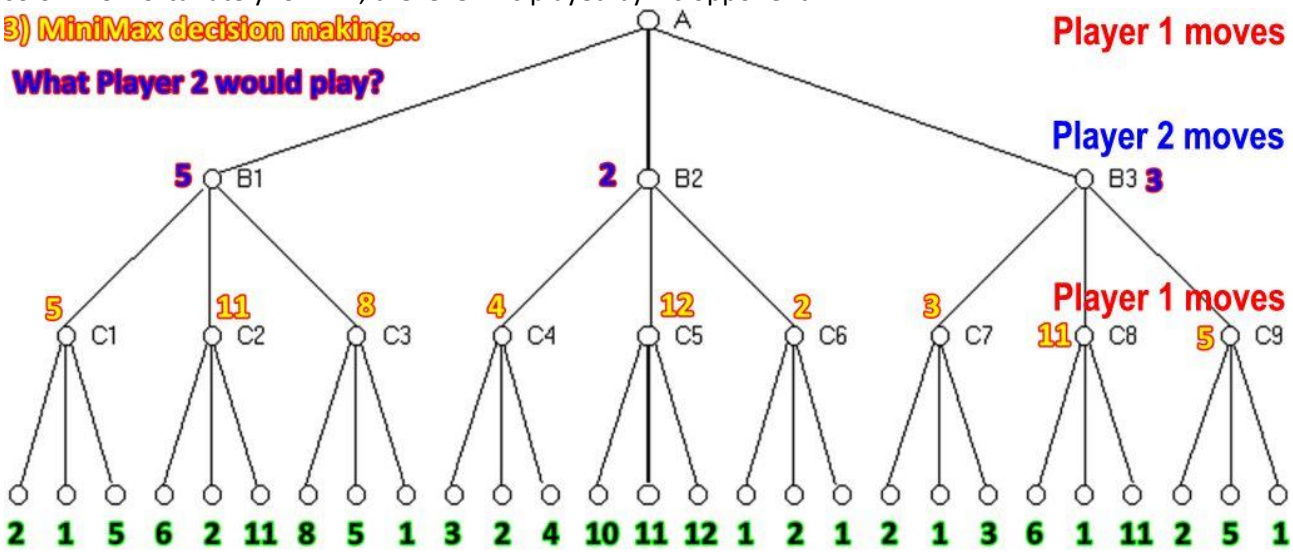
Obviously, player 1 would like to get the maximum possible score (12), but playing the move that brings in B2 would be a serious mistake because once it's up to player 2, he will move correctly in C6, since its goal is always to get the minimum.

So, how do we reason?

For each "C" node, let's consider what would play "Player1".



Player 1 will make the move that gives him score 5 from C1, C2 by 11, 8 by C3, C4 4, 12, from C5, C6 2 and so on... Unfortunately for him, the level B is played by his opponent ...



Player 2 will make the move that gives him score 5 from B1 (he moves in C1), 2 from B2 (moving in C6, certainly not in C5), 3 from B3 (moving C7).

At this point, Player 1 knows that the best move he can make is reasonably **B1** and the best result that can be achieved, in fair play, is 5.

You can check by scrolling the tree that, all other moves lead to a worse result.

- **The MiniMax algorithm is therefore simply explained**:

1. Develop the whole tree of the variants;
2. Assign a score (V) to each terminal position is reached;
3. Consolidate the result backward from each terminal position, taking into account that each node in which Player 1 plays, he will choose the move that maximizes the result, unlike Player 2, which chooses the move that leads to a worse result (for "Player1"), minimizing the score.

Note: assuming that a V ≥ 10 corresponds to a victory for Player 1, V ≤ -10 is a victory for Player 2 and you can judge equality in case of -10 < V < 10 (obviously a party may have a slight advantage but is not sufficient to determine victory), in games played by humans, sometimes we play B2 (thinking to C5), then the opponent makes a mistake (not playing correctly C6) and we get victory . ***You notice that the algorithm is pessimistic, that is, it always treats to play with opponent who is never wrong****. This allows you to lose a few victory due to human error, but also to avoid losses by underestimating the strength of the opponent. algorithm is therefore a "pessimistic" but identifies the best move " if the game is played rightly ".*

Practical use of MiniMax and look-ahead limitation

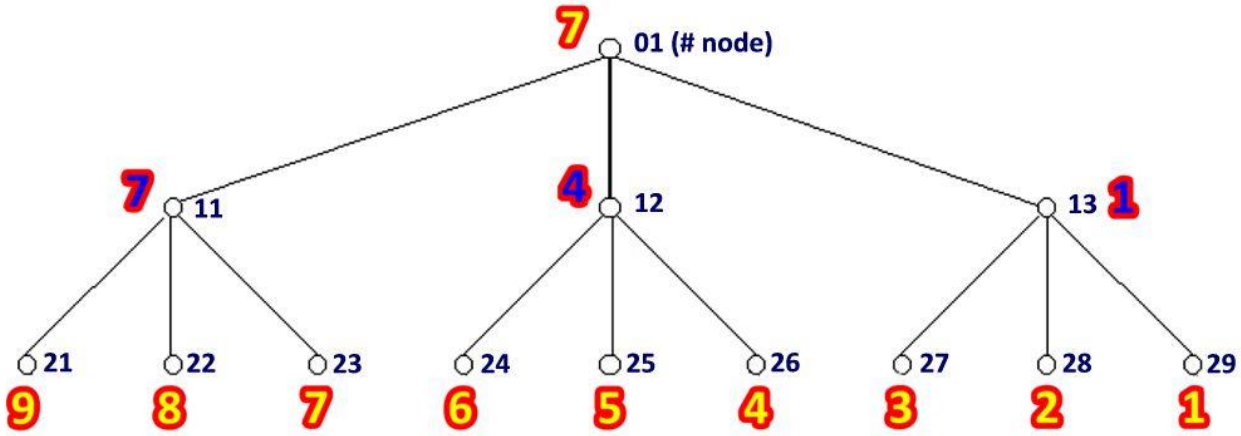
The hypothetical example game counts very few nodes.
Tic-Tac-Toe, which is the simplest "real application" of MiniMax, counts 9! = 362,880 possible outcomes.
The reality is much more complicated for games like chess, for example, in which the number of possible moves is practically infinite: simply, it isn't possible to develop the entire tree.
We should be content to examine small portions of the tree at a time.
The algorithm is adapted so that it arrived at a certain depth of the tree, the nodes are in any case terminated (that is proceed with a heuristic evaluation of the result, as if the game was finished).
Here is a pseudo code for Minimax with look ahead.

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, depth - 1, FALSE)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, depth - 1, TRUE)
            bestValue := min(bestValue, val)
        return bestValue
```

You notice how the algorithm is surprisingly simple.
You can call minimax from the current node, giving a depth limit (look-ahead) ... maximizingPlayer is true if computer has to maximize (it is the turn of player 1) or false otherwise.
A Minimax algorithm should always implement a look-ahead if we do not know in advance what can be deep analysis before finishing the game in all variants. For this reason, for me this is the standard Minimax.

α/β pruning

α/β pruning is an improvement over the MiniMax algorithm.
The problem with MiniMax (even if we use the look-ahead) is that the number of game states it has to examine is exponential in the number of moves. While it is impossible to eliminate the exponent completely, we are able to cut it in half. It is possible to compute the correct MiniMax decision without looking at every node in the tree.
Borrowing the idea of **pruning a tree**, or eliminating possibilities from consideration without having to examine them, the algorithm allows us to discard large parts of the tree from consideration. When applied to a standard MiniMax tree, it returns the same move as MiniMax would, but prunes away branches that cannot possibly influence the final decision.
To better understand how the α/β pruning works, we compare it with a simple MiniMax.



In the figure above, we see a Minimax tree.
The algorithm starts from node 1 to node 11, then passes to the nodes 21,22 and 23 where it got the first scores.
Scores are minimized (at the node 11 moves min player) so **the score at 11 is 7**.
Then node 1 goes to node 12 where due to the same proceeding **the score at node 12 is 4**.
The node 13 is assigned in the same way, too.
Since node 1 maximizes (moves the max player), the best score is 7, then the best move is certainly that corresponding to the node 11.

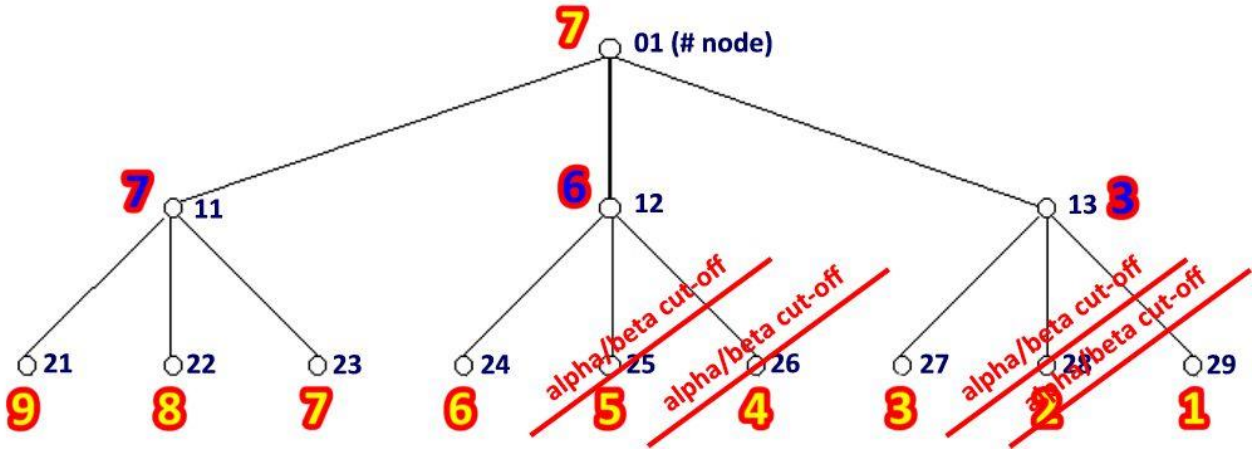
Now think in terms of α/β pruning. Initially, alpha and beta are two coefficients set to $\alpha = -\infty$ and $\beta = +\infty$.
When Max moves, α is the maximum achievable (better result for player Max), when Min moves, β is the minimum achievable (better result for player Min).

If during the analysis occurs $\alpha >= \beta$, certainly the subtree is irrelevant for the choice.
We try to understand it by following the computer analysis of the figure with α/β pruning
The algorithm starts from 1 to 11, then from 11 to 21. At node 21, the first result is 9 so $\beta = 9$.
At node 22, 8 is less than 9 so $\beta = 8$.
At node 23, 7 is less than 8 so $\beta = 7$.
The recursive algorithm returns to node 1. The best result up to that point is 7 so $\alpha = 7$.
Than the program goes to analyze the node 12 (entering with $\alpha = 7$).
The node 12 gets $\beta = 6$ through node 24.

As the computer arrives there with α = 7, that is greater than β = 6, the analysis of the sub shaft 12 is useless. There is no sense in fact, after finding a move (11) which provides a good result, play another (12), which offers the opponent a position that benefits itself. Consequently, the analysis of the node 12 ends immediately because all the branches of that level are pruned.

The program then continues from node 13, which finds a beta = 3 from the node 27.

For the same reason, when they have already a move that provides α = 7, it makes no sense to play a move that gives to the opponent a chance to get a better result for him (β = 3).



At the end, both algorithms finds the best move (11) however, MiniMax must visit all 13 nodes, while MiniMax with α/β pruning visit only 9 nodes.

Only nodes 1→11→{21,22,23},12→24,13→27 are visited by the algorithm MiniMax with α/β pruning.

The example does not emphasize enough the power of the α/β pruning, but in a program with very large trees (*programs that play chess*), this method can eliminate from the analysis billions of subtrees.

Very Important Note: the gains in terms of time are obtained if the nodes to be examined are ranked by the possibility of success (descending) so if you get high scores in the first subtrees, all others will be pruned, while the worst case is when the best move is the last examined.

The code in Java

zeroSumGame interface

We begin by defining an **interface that represents the common elements of any zero-sum game**.

```
public interface zeroSumGame {
    int score();
    boolean isGameOver();
    boolean isMaxPlayerTurn();
    zeroSumGame[] newInstances();
    void play(int index);
}
```

the method **int score()** returns the score from the status of the game: obviously, let +x is the maximum score returned (in favor of player Max),then -x is the best score in favor of the second player.
the method **isGameOver()** returns **true** if the game finished (because of the victory of one contender, or in case of inability to do next move). As long as there are other moves to be played returns false.
the method **isMaxPlayerTurn()** returns **true** if it's the turn of player who needs to maximize the score, false otherwise.
Now pay attention: the method **zeroSumGame [] newInstances()** creates an array of scenarios (instances of the game) that differ from the current one due to the fact that the move was played. In the above example, if the current node is 1, the array contains 11,12 and 13 (11,12 and 13 are not moves, are nodes). Another example: if at a certain point in a game of chess (current position, current node) I only have two moves to play, one with bishop and one with the knight, the array doesn't contain two instances of a class *"Move"*, but two new positions (two new nodes) that are generated in both scenarios.
The reason why I don't create a method that returns directly all possible moves is that the definition of a class "*Move*" is left fully managed by the class that extends **zeroSumGame**.
MiniMax algorithm does not need to know what are the moves but only the positions that they generate.
the **void play()** method updates the current node with the best move found by MiniMax. Its argument is the index to the best move found by MiniMax.



MiniMax implementation

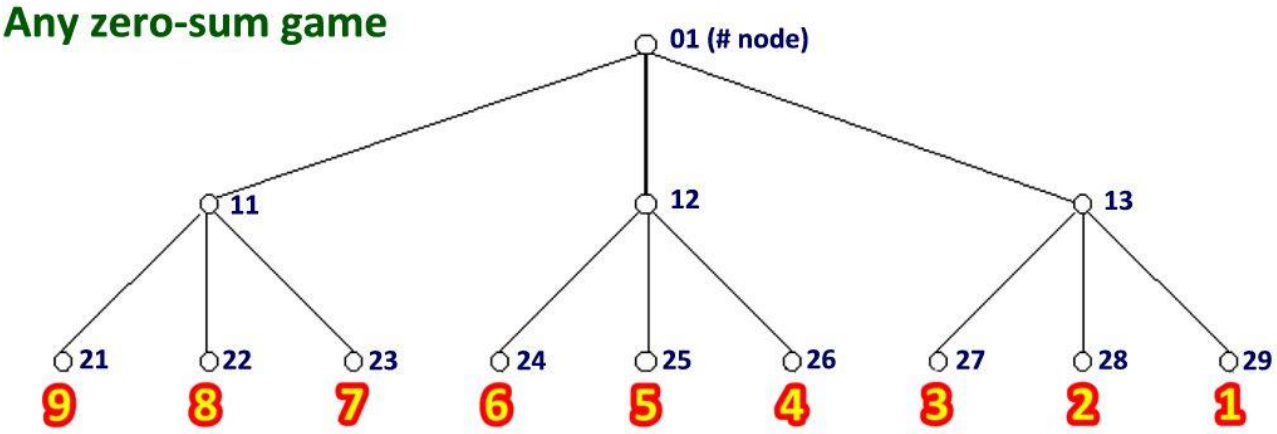
This class is the core of all this article. The constructor accepts any class that implements the methods declared in the **zeroSumGame** interface. Due to the educational approach of the article, the class has methods dedicated to the simple MiniMax and to the MiniMax optimized with α/β pruning, both.

```
class MiniMax {
    static int visited = 0; // only for debug, remove in your application
    static int visited_alphabeta = 0;
    static final int maxLookAhead = 10;
    static final int startAlpha = Integer.MIN_VALUE+1;
    static final int startBeta = Integer.MAX_VALUE-1;
    zeroSumGame game;
    int BestMove = -1;
    MiniMax(zeroSumGame Game) { game = Game; }
    int Start() {
        int v = minimax(game,0);
        game.play(BestMove);
        return BestMove;
    }
    int StartAlphaBeta() {
        int v = minimax(game,startAlpha,startBeta,0);
        game.play(BestMove);
        return BestMove;
    }
    int minimax(zeroSumGame node,int deep) { // Basic MiniMax
        System.out.println("Step = " + ++visited + ". Node = " + node);
        if(node.isGameOver()) return node.score();
        if(deep == maxLookAhead) return node.score();
        int index = -1; // index used to store the best move
        zeroSumGame c[] = node.newInstances();
        int k = c.length;
        // c[0]...c[k-1] are all the new nodes from current one
        if(node.isMaxPlayerTurn()) { // moves the player that maximizes
            int max = Integer.MIN_VALUE;
            for(int i=0;i<k;++i) { // for each new node in c[i]
                int v = minimax(c[i],deep+1); // recursively calls minimax
                if(v > max) { // maximizing the result
                    max = v; // save the best score (the max found so far)
                    index = i; // stores the index on the best move found so far
                }
            }
            BestMove = index; // save the best move found
            return max; // return the best score (for Player Max)
        } else { // This code is the same as before but minimizes for Player Min
            int min = Integer.MAX_VALUE;
            for(int i=0;i<k;++i) {
                int v = minimax(c[i],deep+1);
                if(v < min) { // minimizing the result
                    min = v;
                    index = i;
                }
            }
            BestMove = index;
            return min;
        }
    }
```

```
    // MiniMax with alpha/beta pruning optimization
    int minimax(zeroSumGame node,int alpha,int beta,int deep) {
        System.out.println("Step = " + ++visited_alphabeta + ". Node = " + node);
        if(node.isGameOver()) return node.score();
        if(deep == maxLookAhead) return node.score();
        int index = -1; // index used to store the best move
        zeroSumGame c[] = node.newInstances(); // c[0]... = all new nodes
        if(node.isMaxPlayerTurn()) {
            for(int i=0;i<c.length;++i) { // for each new node...
                int v = minimax(c[i],alpha,beta,deep+1);
                // alpha = max(alpha,v);
                if(alpha < v) {
                    alpha = v;
                    index = i;
                    System.out.println("(node " + ((aGame)node).thisNode +
                            ") alpha = " + alpha);
                }
                if(alpha >= beta) break; // alpha/beta pruning
            }
            BestMove = index;
            return alpha;
        } else {
            for(int i=0;i<c.length;++i) {
                int v = minimax(c[i],alpha,beta,deep+1);
                // beta = min(beta,v);
                if(beta > v) {
                    beta = v;
                    index = i;
                    System.out.println("(node " + ((aGame)node).thisNode +
                            ") beta = " + beta);
                }
                if(alpha>=beta) break; // alpha/beta pruning if alpha>=beta
            }
            BestMove = index;
            return beta;
        }
    }
}
```

Any zero-sum game

For didactic purposes we'll use the "game" defined by the tree of the variants previously seen. This is not strictly a game, but has all the elements of a zero-sum game: a tree variants, two players, a score, different positions. It's very simple so that you can launch the application in debug mode and see what happens stepping into, statement after statement. Any other game would have a much larger tree.



Here the Java code

```
class aGame implements zeroSumGame {
    public int thisNode;
    private boolean player;
    private int[] list;
    static final int f1[] = { 11,12,13 }, f11[] = { 21,22,23 },
                     f12[] = { 24,25,26 },f13[] = { 27,28,29 };
    aGame(boolean Player) { // default costructor
        thisNode = 1; // initial status
        player = Player; // player who moves first (Max in this case)
    }
    aGame(aGame game,int m) { // costructor used in newInstances()
        player = !game.player; // the move goes to the contender
        thisNode = m; // set the node to the right value
    }
    @Override
    public int score() { // return the score for any final position
        int p[] = {9,8,7,6,5,4,3,2,1 };
        if(thisNode>20) return p[thisNode-21]; else return 0;
    }
    @Override
    public boolean isGameOver() { // Game over if a node from 21 to 29 is reached
        return  thisNode>20 && thisNode<30;
    }
    @Override
    public boolean isMaxPlayerTurn() { return player; }
    @Override
    public zeroSumGame[] newInstances() {
        int k = getList(); // get (privately) a list of k moves
        zeroSumGame c[] = new zeroSumGame[k]; // generates a k sized array
        for(int i=0;i<k;++i) c[i] = new aGame(this,list[i]);
        return c;
    }
    @Override
```

```
public void play(int index) {
        thisNode = list[index]; // the new node is the one indexed in the list
        player=!player; // the moves goes to the opponent
        System.out.println("My move is: " + list[index]); // debug
    }

    @Override
    public String toString() {
        return "Node: " + thisNode;
    }
    private int getList() {
        switch(thisNode) {
            case 1:
                list = f1;
                break;
            case 11:
                list = f11;
                break;
            case 12:
                list = f12;
                break;
            case 13:
                list = f13;
                break;
        }
        return list.length;
    }
}
```

The test (main) application

The example simply initializes two games (*at node 1 with the initial move to the player who maximizes*) and call the basic algorithm and the optimized one with the α/β pruning, both, printing the best move found.

```
public class Application {
    static public void main(String args[]) throws CloneNotSupportedException {
        aGame game1 = new aGame(true);
        aGame game2 = new aGame(true);
        MiniMax m1 = new MiniMax(game1);
        MiniMax m2 = new MiniMax(game2);
        int simpleMiniMax = m1.Start();
        System.out.println(simpleMiniMax);
        int alphabeta = m2.StartAlphaBeta();
        System.out.println(alphabeta);
    }
}
```

Compiling and running the program you get the following output …

```
Step = 1. Node = Node: 1
Step = 2. Node = Node: 11
Step = 3. Node = Node: 21
Step = 4. Node = Node: 22
Step = 5. Node = Node: 23
Step = 6. Node = Node: 12
Step = 7. Node = Node: 24
Step = 8. Node = Node: 25
Step = 9. Node = Node: 26
Step = 10. Node = Node: 13
Step = 11. Node = Node: 27
Step = 12. Node = Node: 28
Step = 13. Node = Node: 29
My move is: 11
0
Step = 1. Node = Node: 1
Step = 2. Node = Node: 11
Step = 3. Node = Node: 21
(node 11) beta = 9
Step = 4. Node = Node: 22
(node 11) beta = 8
Step = 5. Node = Node: 23
(node 11) beta = 7
(node 1) alpha = 7
Step = 6. Node = Node: 12
Step = 7. Node = Node: 24
(node 12) beta = 6
Step = 8. Node = Node: 13
Step = 9. Node = Node: 27
(node 13) beta = 3
My move is: 11
0
```

As you can see the α/β pruning requires a smaller number of nodes to visit.
Obviously this is only a basic implementation.
Adding functionality to the class you can extend its use in almost all zero-sum games.

Paolo Ferraresi

