Date: 16th March,2017

Artificial Intelligence Fundamentals

## Chess Masters

## Project of an application to play chess with the Min-Max algorithm.

****

**IVAN MATYAZH**

**FILIP MATRACKI**

**FARDIN MOHAMMED**

**Table of Contents**

1. Description of the problem.
2. Descriptions of preferred solutions.
3. Analysis of the solutions
4. Existing solutions.
5. Bibliography.

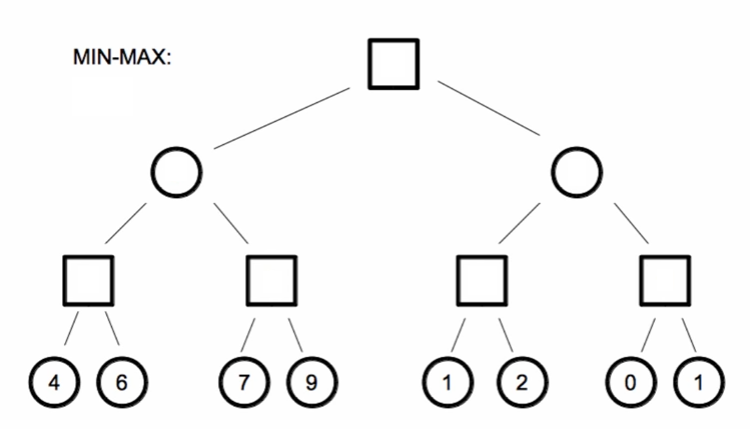
Description of the problem

Implementation of an application to play chess with Min Max algorithm. It is a WPF application written in C# using Microsoft Visual Studio as the development environment. It is aimed to make humans play against an AI which determines the score in game after certain number of moves according to the programmed evaluation function and tries to make the best move possible. Later, it can be extended for AI vs AI. After analyzing existing the solutions the problem tends to be difficult but research of people has made the choices for solutions had been narrowed and focused on choosing two simple solutions. Min Max Algorithm and Negamax Algorithm (simpler version derived from Min Max) described below in detail. After some research of our own we concluded that Min Max might not be shorter but simpler to implement while there were some cases of failing with Negamax.

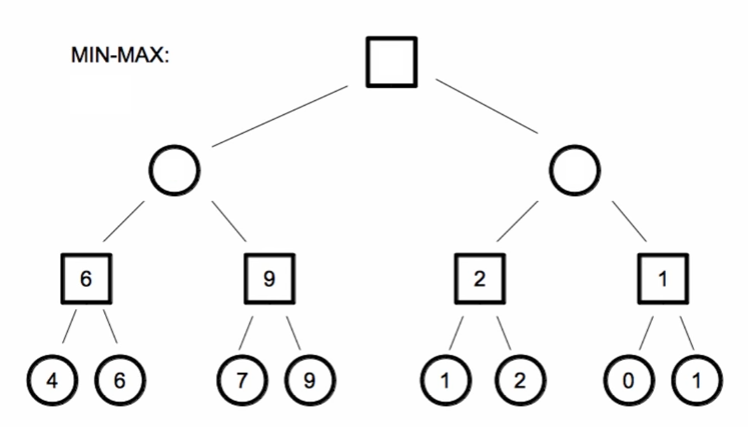
Descriptions of preferred solutions

* **Min-Max Algorithm.**

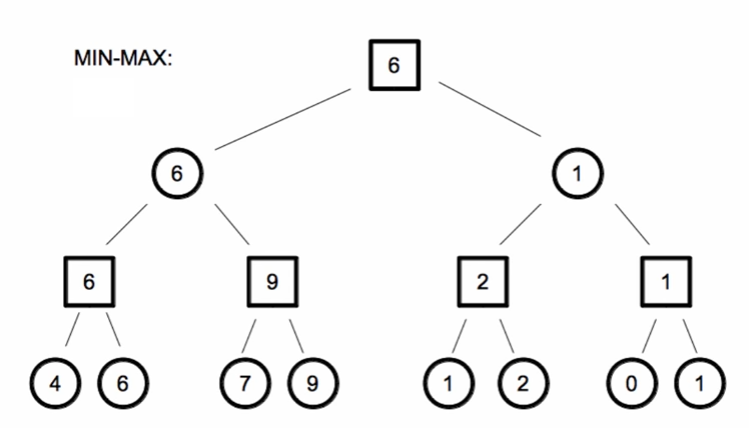
Min Max (also known as “Minimax”) is an algorithm used to determine the score in a zero-sum game after a certain number of moves, with best play according to an evaluation function. A zero-sum game can be defined as a game where one side’s gain is equalized by another side’s loss. For example, in chess when the evaluation function returns plus three for white, then the evaluation is minus three for black. The two values when summed up give zero, hence the name “zero-sum”. One side can be described as the “maximizing” side and another as the “minimizing” one. If one is playing white than it is the “maximizing” side and vice-versa. This can be easily visualized in the form of a tree shown below.



As one can see in the figure above, this is an example of a min max tree of height 3. Let’s assume nodes shown as squares represent white’s moves, and nodes shown as circles represent black’s moves. The children of the root node (level 0) are all the possible moves that the maximizing side can make (white). The value inside the node represents the evaluation of the position, the higher the value the greater white’s advantage is. Let’s also assume that after every move one can make 2 legal moves for the sake of the example. The min max algorithm is usually implemented recursively, evaluating the leaves of the tree first (level 3) and moving up from there. So once all of white’s moves have been evaluated, white chooses the maximum of the child nodes because it wants the highest possible evaluation to win the game. Now one can enter the values of each node at depth-level 2. The results are as follows:

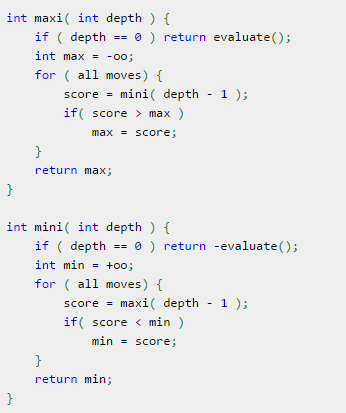


At this moment black wishes to choose the position with the smallest evaluation for white, because black is the minimizing player, and then in the same way as before white maximizes the values in the child nodes. Finally one obtains the filled in tree:



Now we have the information that if both sides were to make the best possible moves, the evaluation is plus 6 for white. So we decide to make the move represented by the left child of the root. If multiple child nodes have the same value as the maximum, we can choose one of them randomly. Once we choose the left child node for the move we make, it becomes the new root of the tree and we can run the Min Max algorithm again. This process is repeated until the game is over.

The pseudo code for the algorithm is shown in the figure below, please note that the *makeMove() and unmakeMove()* functions are missing before and after the recursive call respectively.



* **Negamax Algorithm.**

Negamax is a simplified and easier way of implementing Min Max. Instead of using two subroutines for the maximizing and maximizing player, one can apply the zero-sum property of chess to pass the negated score due to the following mathematical relation:

max(a, b) == -min(-a, -b)

Applying this property we obtain the following pseudo code for the *negaMax()* function:

int negaMax( int depth ) {

if ( depth == 0 ) return evaluate();

int max = -oo;

for ( all moves) {

score = -negaMax( depth - 1 );

if( score > max )

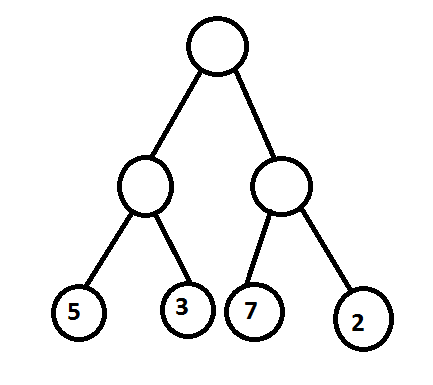
max = score;

}

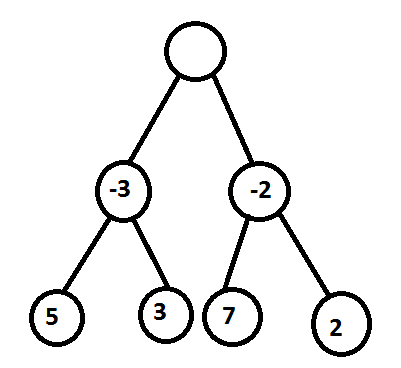
return max;

}

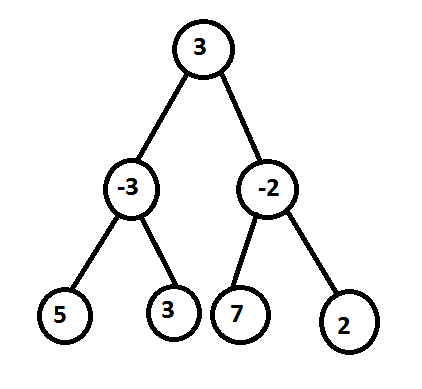
As before, the *makeMove() and unmakeMove()* functions are omitted before and after the recursive call for clarity.

Let’s create a simple tree to visualize this version of the algorithm as well. Consider the following tree of height 3 with the following evaluations of the leaf nodes:

Now at level 1 of the tree, we find the maximum of the child node and negate it. Which gives us the following tree:



We repeat this process one final time for the root node, to obtain the full tree:



One can verify that the evaluation of the root node will be the same after applying the original version of Min Max described previously. However there are possible issues that arise when using megamax. How to call Negamax? If Negamax only gives an optimal score then how to choose the move to be made?

An analysis of solutions

**Downsides and disadvantages of the Min Max algorithm**

Both Negamax and Min Max contain some serious downsides in their basic state. The first downside is that they are both brute-force algorithms, in other words it is necessary to search for every legal move at each depth, so the amount of moves that are checked at each level increases exponentially. There are some solutions that optimize the algorithm, such as ***alpha-beta pruning*** which seek to decrease the amount of nodes that are evaluated by the Minimax algorithm in the search tree.

Another known problem is what is known as the *horizon effect.* Because we use Min Max at a given depth, we know that we will wake make the best possible move if we only consider that depth. However computers can only search at a constant depth *n.* Min Max/Minimax and Negamax might give us the best move for that depth, but it fails to consider moves at level *n + 1.* Therefore we might be play a move that leads us to a position where our queen can be taken by the opponents pawn, which causes us to lose the game. This is due to the fact that the computer did not consider moves at depth *n + 1,* so it had no idea that our queen would be lost*.* What’s even more interesting, is that increasing the depth will not fix this problem, because there will always be the *n + 1* level that the computer did not consider. The solution is to use what is known as *Quiescence Search* which does not stop evaluating a position until it is “quiet”, meaning that no tactical winning moves can be made (such as taking a queen).

The Chess Programming wiki offers us this solution:

*One calls negaMax with another root negaMax which makes the call to the negaMax proper with the default search depth. In the body of the loop of this root negaMax, in the loop which generates all the root moves – there one holds a variable as you call negaMax on the movement of each piece – and that is where you find the particular move attached to the score – in the line where you find*score > max*, right after you keep track of it by adding*max = score*– in the root negamax, that is where you pick out your move – which is what the root negaMax will return (instead of a score).*

Some existing solutions

* An implementation of the Min Max Algorithm in JAVA.

<https://github.com/Arwid/chess/blob/master/source/chess/algorithms/MiniMax.java>

* An implementation of an application to play chess using Min Max Algorithm using alpa-beta pruning in Python.

<https://github.com/lamesjim/Chess-AI>

* An implementation of Negamax algorithm with alpha-beta pruning.

<https://github.com/schainan/211-Chess/blob/master/src/edu/cmu/cs211/chess/unittested/TestedAlphaBetaFixedDepth.java>

* An Implementation of the Evaluation function (for the best move evaluation) in JAVA.

<https://github.com/schainan/211-Chess/blob/master/src/edu/cmu/cs211/chess/unittested/TestedEvaluator.java>

Bibliography:

Chess Programming Wiki:

<https://chessprogramming.wikispaces.com/>

Negamax on CPW:

<https://chessprogramming.wikispaces.com/Negamax>

Negamax on Wikipedia:

<https://chessprogramming.wikispaces.com/Negamax>

..with the gif animation of the Negamax algorithm

<https://en.wikipedia.org/wiki/Negamax#/media/File:Plain_Negamax.gif>

Quiescence Search on CPW:

<https://chessprogramming.wikispaces.com/Quiescence+Search>