1 Introduction

Consider the following abstract game:

There are two players, Max and Min, and the moves they make influence a specific shared value. Max is attempting to obtain the highest value possible while Min is attempting to obtain the lowest number possible.

Suppose the game is exactly two moves long - Max will make a move and then Min will make the move. Suppose each has two options for their move, call these A and B, and the options and results are diagrammed by the following tree:

```
   A   B
  /     \
A  B   A  B
/     /   /   /
2  10  4  8
```

Let us take a minute to make sure we understand what this tree means. Think of the root as the start of the game. Max can choose either move A (going left) or move B (going right) at which time Min makes a similar choice.

What should Max do?

Clearly Max sees that there is a 10 available so Max might choose move A but if Max does so then Min will choose move A and the game will have a value of 2. On the other hand if Max chooses move B then Min will choose move A and the game will have a value of 4.

It follows that Max should choose move B to obtain the highest possible value (given Min’s response) of 4.

2 Minimax Algorithm

2.1 Inspiration

Imagine a game with two players, Max and Min. We have a function which assigns a value to each position in the game. Max’s goal is to achieve the maximum possible value while Min’s goal is to achieve the minimum possible value.

The idea however is not to simply look one move ahead but to look many moves ahead.

When it is Max’s turn to move, Max does not know what Min will do (after
Max moves) but Max does know what possibilities exist for Min. Similarly Min knows what possibilities exist for Max.

The Minimax Algorithm is a simple algorithm which gives the best possible move, assuming it is Max’s turn. If there are multiple branches with the same value then Max may choose either branch.

Let us revisit the opening example:

Assume the leaf values are the values of the function two moves ahead. We saw in the opening that Max ought to choose move $B$ to achieve a score of 4. We could represent this by putting an 4 in the root:

In addition we could look at the middle layer and consider those are Min’s moves. On the left (after Max chooses move $A$) clearly Min would choose move $A$, achieving a value of 2, and on the right (after Max chooses move $B$) Min would also choose move $A$, achieving a value of 4. We could fill those values in too:
We now make the simple observation that we can fill the tree in from the bottom up using the following algorithm:

- If we are filling in Min’s row we should use the smaller (the minimum) of the child values.
- If we are filling in Max’s row we should use the larger (the maximum) of the child values.

So now onto the algorithm in pseudocode.

### 2.2 Algorithm

Let’s assume we have a tree consisting of a set of nodes. The leaf nodes are assumed to represent the ends of the game and so for a leaf node \( n \) the value \( n.value \) is pre-assigned. All other nodes have \( n.value \) undefined to start with.

The function \( \text{minimax} \) is called with the first argument being the root of the tree and the second argument being 0. Recursive calls will have \( \text{depth} \) being even when it is Max’s turn and odd when it is Min’s turn.

```plaintext
function minimax (n, depth):
    if n.value is not undefined:
        return (n.value)
    else:
        if depth is even:
            return ( max ( minimax (nc , depth+1) for all children nc of n) )
        else:
            return ( min ( minimax (nc , depth+1) for all children nc of n) )
        end if
    end if
end function
```

### 2.3 Time Complexity

The algorithm makes no assumptions about how many children each node has, the depth of the tree, or whether the tree is complete. Suppose however that
the depth of the tree is \( d \) and the tree is complete with \( b \geq 2 \) children per node (this is the branch factor). The function itself runs at constant time for a given node so we need to count the number of nodes in the tree.

Assuming the constant time for one node is 1 then this yields a simple geometric sum for the time complexity:

\[
1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} = \left( \frac{1}{b-1} \right) (b^{d+1} - 1) \leq b^{d+1} - 1
\]

This yields a time complexity of \( O(b^d) \).

### 2.4 Practicalities

In real-world applications it is generally not realistic to construct a tree which goes all the way to the end of the game due to the sheer number of possible moves available to each player and the number of moves the game lasts. In such cases we proceed as follows:

1. Decide how many moves we can realistically look ahead.
2. Expand the tree to that depth and to each leaf node (which will correspond to a game arrangement) we assign a value.
3. Apply the Minimax Algorithm to that tree.

When it comes to assigning values to leaf nodes we need some value which depends on the game arrangement.

**Definition 2.4.1.** The function which assigns a value to a particular game arrangement is often called the *heuristic function*.

**Example 2.1.** In chess, assuming it is white's move, a simple heuristic function might assign a value to each white piece and to each black piece and then subtract the black total from the white total.

**Example 2.2.** In poker we might develop some heuristic function which accounts for how close a player is to various hands.

Once everything is set up we can then proceed.

**Example 2.3.** Suppose we are playing a very simple game in which each player has a number of tokens. When it is each player's turn to move there are up to three possible moves:

- **A** results in the player gaining two tokens.
- **B** results in the player gaining one token.
- **C** results in the other player gaining one token.

However and importantly for any given position not all moves are always
possible.

In addition due to processing constraints we may only analyze future moves from positions in which there are fewer than 5 tokens in total.

The state of the game is given as a pair \((M,m)\) where \(M\) is the number of tokens that Max has while \(m\) is the number of tokens that Min has.

Currently the state of the game is \((1,0)\) and it is Max’s turn to move. We diagram all possible moves that we can and find them to be as follows. Note that leaf nodes correspond to states where there are 5 or more total tokens since we cannot analyze future moves at that point.

We wish to know which move Max should make.

The first thing we do is create a reasonable heuristic function. Since Max wants to end up with more tokens than Min, a reasonable heuristic for a position \((M,m)\) is \(M - m\).

Before proceeding further observe that Max might be tempted to do move \(A\) resulting in a value of \(3 - 0 = 3\) but this is naive because Min will then do move \(A\) too, resulting in \(3 - 2 = 1\), and we can’t look further than that.

Instead we assign the heuristic value to all the leaf nodes first. We’ll add these values to the image above:
We then use the Minimax Algorithm to fill in the values from the bottom up:

We now see that Max’s best move is move $B$. Min will then choose move $A$ and then Min will choose move $A$ for a final heuristic value of 2.