CMSC 351: Minimax

Justin Wyss-Gallifent

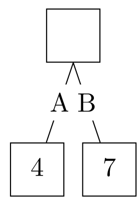
November 29, 2023

1 Introduction

Consider the following abstraction of a game:

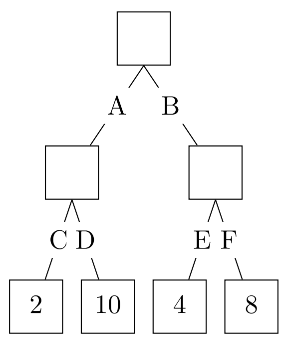
Max and Min are playing a game in which the entire state of the game can be represented by a single game value. Max's objective is to obtain the largest game value possible while Min's objective is to obtain the smallest game value possible.

Suppose it is Max's move and there are two moves available, A and B. Each leads to a specific game value:



Clearl Max would choose move B, resulting in a game value of 7.

Suppose on the other hand it is Max's move but Max doesn't know the game values which result. Instead, after Max's move it is Min's move. However suppose Max can see two moves ahead as follows:



What should Max do?

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

Now let's un-abstract it a bit.

Imagine a game with two players, Max and Min. We have a function which assigns a value to each position in the game.

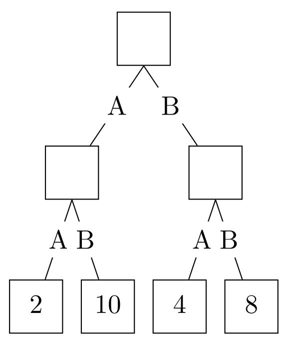
Definition 2.1.1. A function which assigns a value to each position in the game is called a *heuristic function*. We'll denote this by h.

Max's goal is to achieve the maximum possible value while Min's goal is to achieve the minimum possible value.

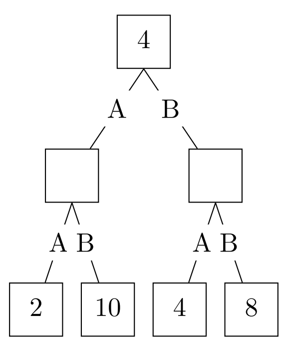
The underlying idea is that Max may look some number of moves ahead (maybe 1, maybe 100), calculate the corresponding heuristic values, and will make a choice of current move based upon that.

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.

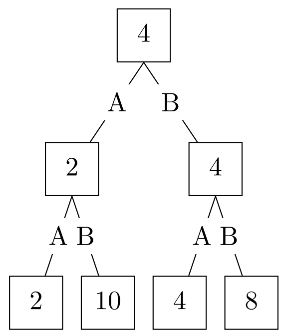
Let us revisit the opening example:



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 eventually achieve $h = 4$. 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 to eventually achieve $h = 2$, and on the right (after Max chooses move B) Min would also choose move A to eventually achieve $h = 4$. 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.

Note 2.1.1. It is important to note that the leaves of the tree are the only nodes which we fill with the values of the heuristic function based on the states of the game.

When we are filling in the nodes above these are minimax values, not heuristic function 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 `minimax` is called with the first argument being the root of the tree and the second argument being 0. Recursive calls will have *depth* being even when it is Max's turn and odd when it is Min's turn.

```
function minimax(node nd,depth):
  if nd.value is not undefined:
    return(nd.value)
  else:
    if depth is even:
```

```
      return( max(minimax(ndc,depth+1) for all children ndc of nd) )
    else:
      return( min(minimax(ndc,depth+1) for all children ndc of nd) )
    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:

$$1 + b + b^2 + ... + b^d = \frac{b^{d+1} - 1}{b - 1} = \left(\frac{1}{b-1}\right)\left(b^{d+1} - 1\right) \leq b^{d+1} - 1$$

This yields a time complexity of $\mathcal{O}(b^d)$.

3 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.

4 Examples

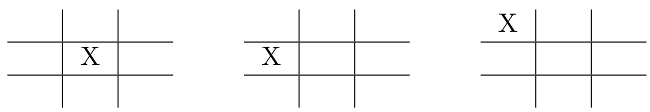
Example 4.1. Consider tic-tac-toe where Max is playing X and goes first while Min is playing O and goes second.

While this is a simple game to play, drawing a tree diagram is not trivial. Assuming Max starts, there are 9 moves available, followed by 8 for Min, and so on. Of course we can reduce this by symmetries since technically speaking Max has only three opening moves - Center, Edge, Corner.

One reasonable heuristic h would be the number of open winning lines (OWLs) for Max minus the number of OWLs for Min. An OWL is a line which contains at least one of the player's marks and none of the opponent's.

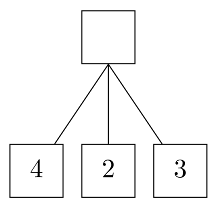
At the start of the game we have $h = 0 - 0 = 0$. and it is Max's move.

Let's look just one move ahead. If we look at Max's three opening moves (up to symmetry) we can calculate the heuristic function h:

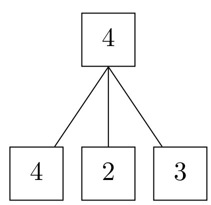


h=4−0=4 h=2−0=2 h=3−0=3

Let's put the values of the heuristic function after one move into the leaves of a tree:

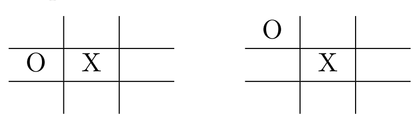


If we then fill it in according to minimax:



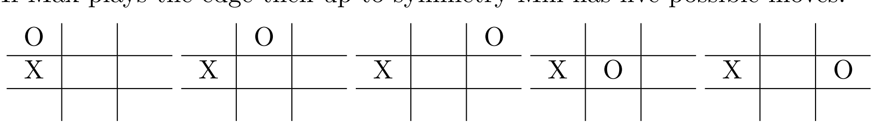
We see that it is in Max's best interest to play the middle.

Let's now look at two moves ahead. If Max plays the center then up to symmetry Min has two possible moves:



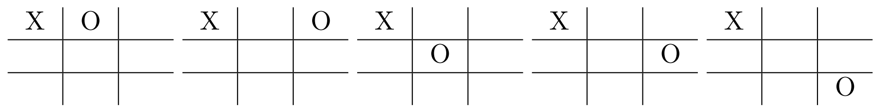
h=3−1=2 h=3−2=1

If Max plays the edge then up to symmetry Min has five possible moves:



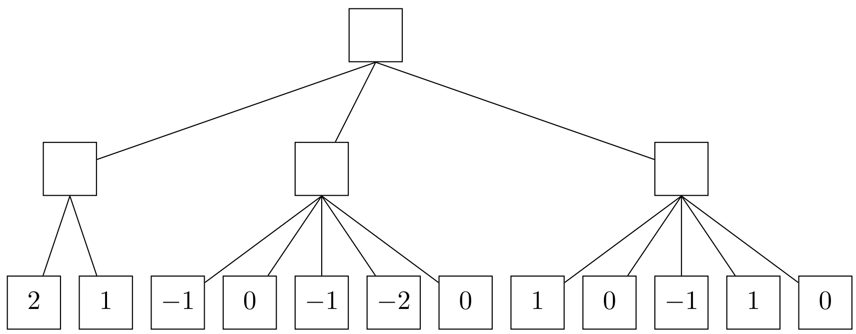
h=1−2=−1 h=2−2=0 h=2−3=−1 h=1−3=−2 h=1−1=0

If Max plays the corner then up to symmetry Min has five possible moves:

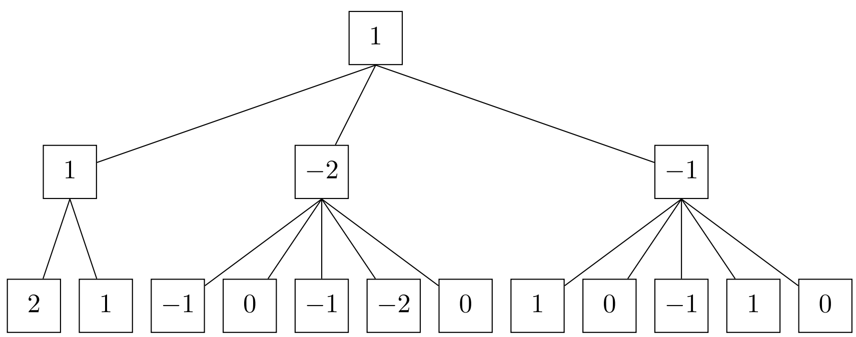


h=2−1=1 h=2−2=0 h=2−3=−1 h=3−2=1 h=2−2=0

Let's put the values of the heuristic function after two moves into the leaves of a tree:



If we then fill it in according to minimax:



We see that it is still in Max's best interest to play the middle.