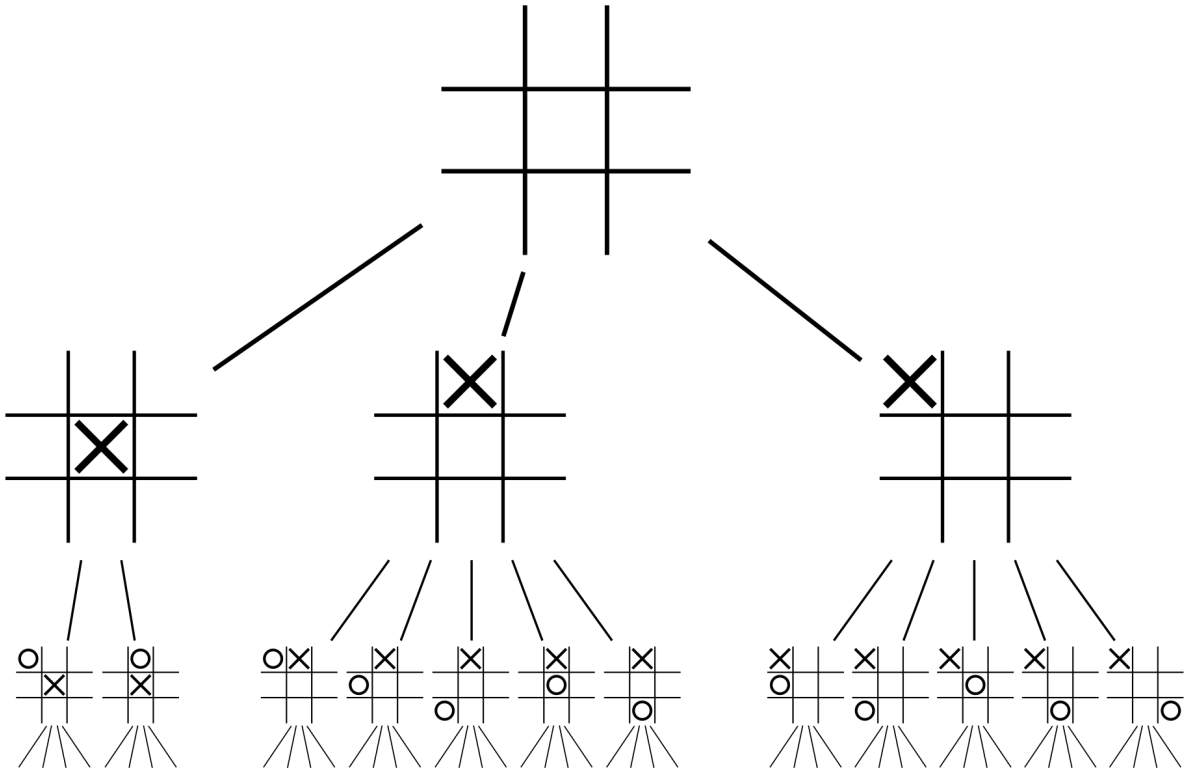
Minimax Algorithm

The minimax algorithm tries to find the best move for the current player by searching in the game state tree.

Game Trees

This is what a state tree looks like for tic-tac-toe:



Each players' moves lead to child states in the tree of all possible game sequences.
The size of this game tree grows exponentially in the depth (number of turns taken).
In practice, we must limit our search of the game tree to a certain depth.

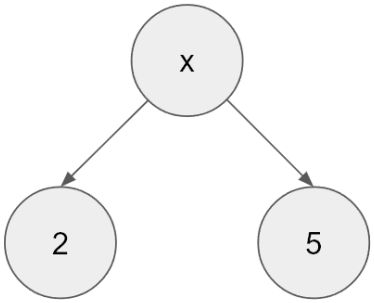
The Idea of Minimax

In minimax, we assume there is an evaluation function that assigns scores to states in the game tree.
For example, a winning move should lead to a state with a high score and a losing move should lead to a state with a low score.

Now we can search through our game tree to find a good move.
Good moves lead to states with a high score and don't allow the opponent to play a move that forces us to states with low scores.
The core assumption in minimax is that the opponent will always play their best move (which is the worst move for us).

Since we always evaluate states from our perspective, the opponent is trying to minimize the score of our state.
On the other hand, we will always pick the state with the highest score (maximize).
This is where the name of the algorithm comes from.

For example, in depth 1:
If our current state x has two possible successors:
one with a score of 2,
and one with a score of 5,
then we would pick the move that leads to the first state while the opponent would obviously pick the other:



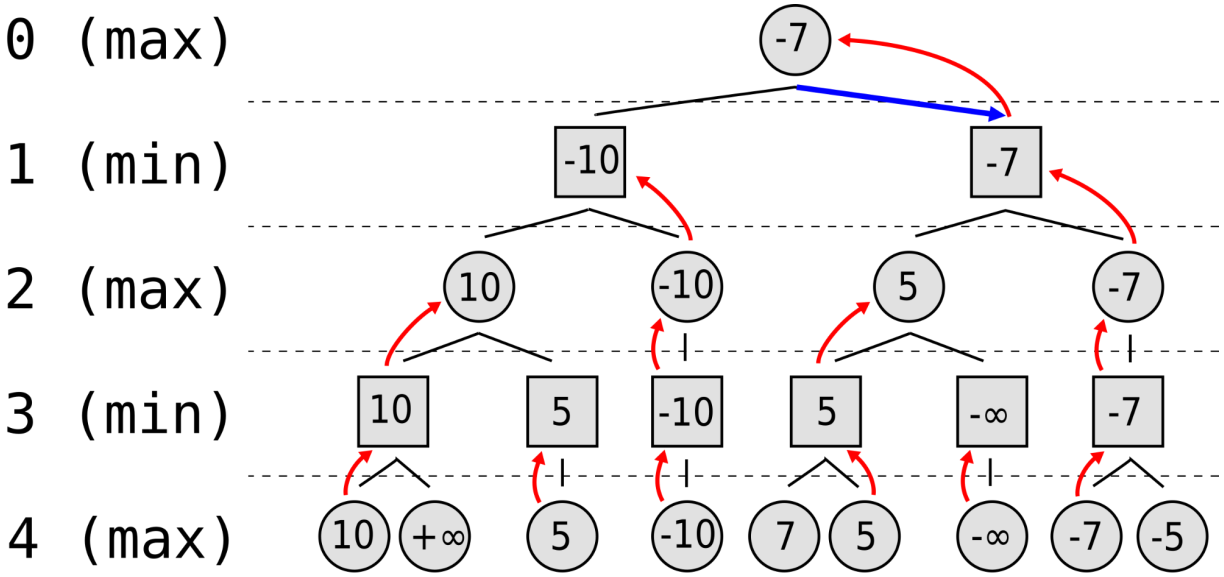
Using this information, we can assign a score or value to state x as well.
If it is our turn, then 5, otherwise 2.
We can do this for any number of successor states by taking the maximum or minimum of the child state values.

Now the minimax algorithm is as easy as:

1. Calculating the scores of the leaf nodes (the final states) using our evaluation function
2. Propagating these scores back up the tree by alternatively using the maximum or minimum of child values, depending on who's turn it is
3. At the root node (our current state) we choose the move that leads to the direct successor with the highest value

Example:

The current player will pick the second move.
In this case, winning and losing are scored as positive and negative infinities respectively.



Pseudocode:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value

(* Initial call *)
minimax(origin, depth, TRUE)
```

Alpha-Beta Pruning

Alpha-Beta pruning is a technique to prune states from the game tree so we don't have to search through them.
The trick is to only prune states that can’t be better or worse (depending on who’s turn it is) than the current evaluation.

Pseudocode:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α,
β, FALSE))
            if value ≥ β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α,
β, TRUE))
            if value ≤ α then
                break (* α cutoff *)
            β := min(β, value)
        return value

(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```