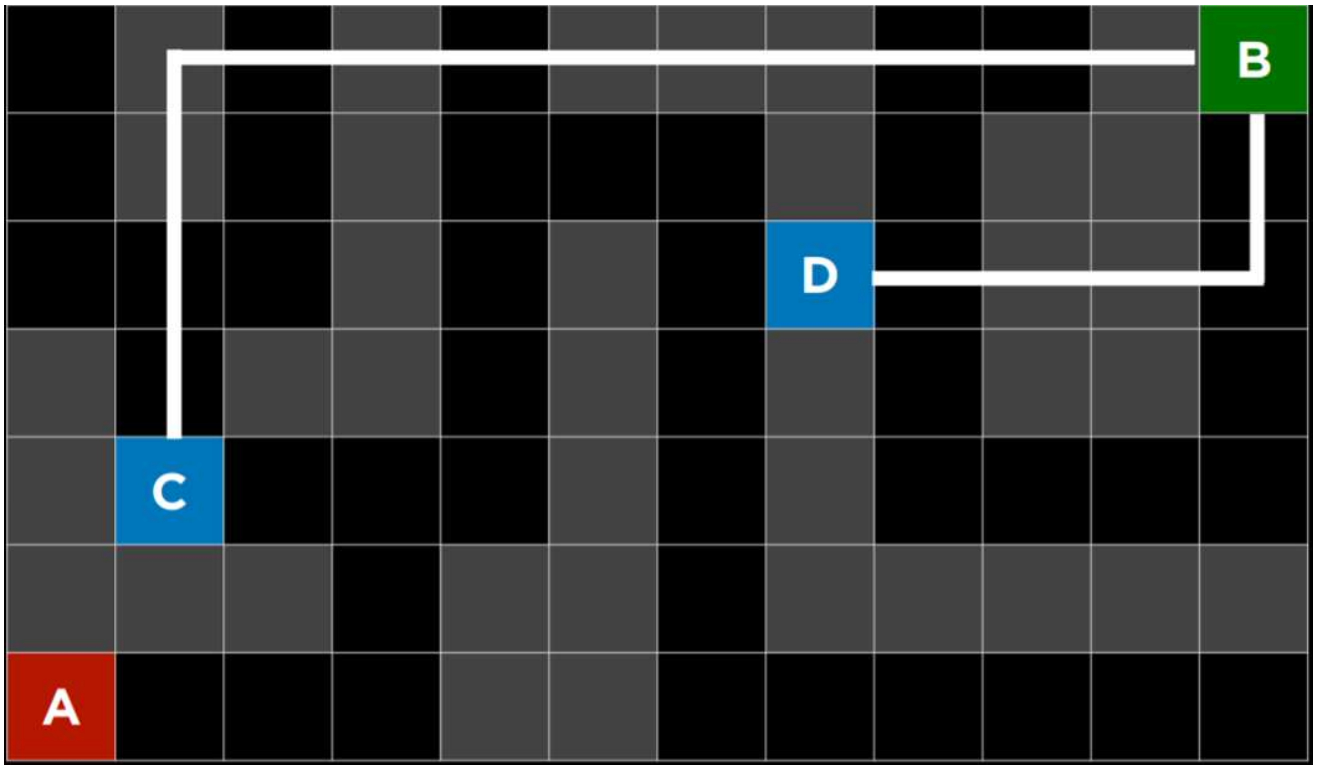
Greedy best-first search expands the node that is the closest to the goal, as determined by a **heuristic function** *h(n)*. As its name suggests, the function estimates how close to the goal the next node is, but it can be mistaken. The efficiency of the *greedy best-first* algorithm depends on how good the heuristic function is. For example, in a maze, an algorithm can use a heuristic function that relies on the **Manhattan distance** between the possible nodes and the end of the maze. The *Manhattan distance* ignores walls and counts how many steps up, down, or to the sides it would take to get from one location to the goal location. This is an easy estimation that can be derived based on the (x, y) coordinates of the current location and the goal location.



Manhattan Distance

However, it is important to emphasize that, as with any heuristic, it can go wrong and lead the algorithm down a slower path than it would have gone otherwise. It is possible that an *uninformed* search algorithm will provide a better solution faster, but it is less likely to do so than an *informed* algorithm.

A* Search

A development of the *greedy best-first* algorithm, *A* search* considers not only *h(n)*, the estimated cost from the current location to the goal, but also *g(n)*, the cost that was accrued until the current location. By combining both these values, the algorithm has a more accurate way of determining the cost of the solution and optimizing its choices on the go. The algorithm keeps track of (*cost of path until now* + *estimated cost to the goal*), and once it exceeds the estimated cost of some previous option, the algorithm will ditch the current path and go back to the previous option, thus preventing itself from going down a long, inefficient path that *h(n)* erroneously marked as best.

Yet again, since this algorithm, too, relies on a heuristic, it is as good as the heuristic that it employs. It is possible that in some situations it will be less efficient than *greedy best-first* search or even the *uninformed* algorithms. For *A* search* to be optimal, the heuristic function, *h(n)*, should be:

1. *Admissible*, or never *overestimating* the true cost, and
2. *Consistent*, which means that the estimated path cost to the goal of a new node in addition to the cost of transitioning to it from the previous node is greater or equal to the estimated path cost to the goal of the previous node. To put it in an equation form, *h(n)* is consistent if for every node *n* and successor node *n'* with step cost *c*, $h(n) \leq h(n') + c$.

Adversarial Search

Whereas, previously, we have discussed algorithms that need to find an answer to a question, in **adversarial search** the algorithm faces an opponent that tries to achieve the opposite goal. Often, AI that uses adversarial search is encountered in games, such as tic tac toe.

Minimax

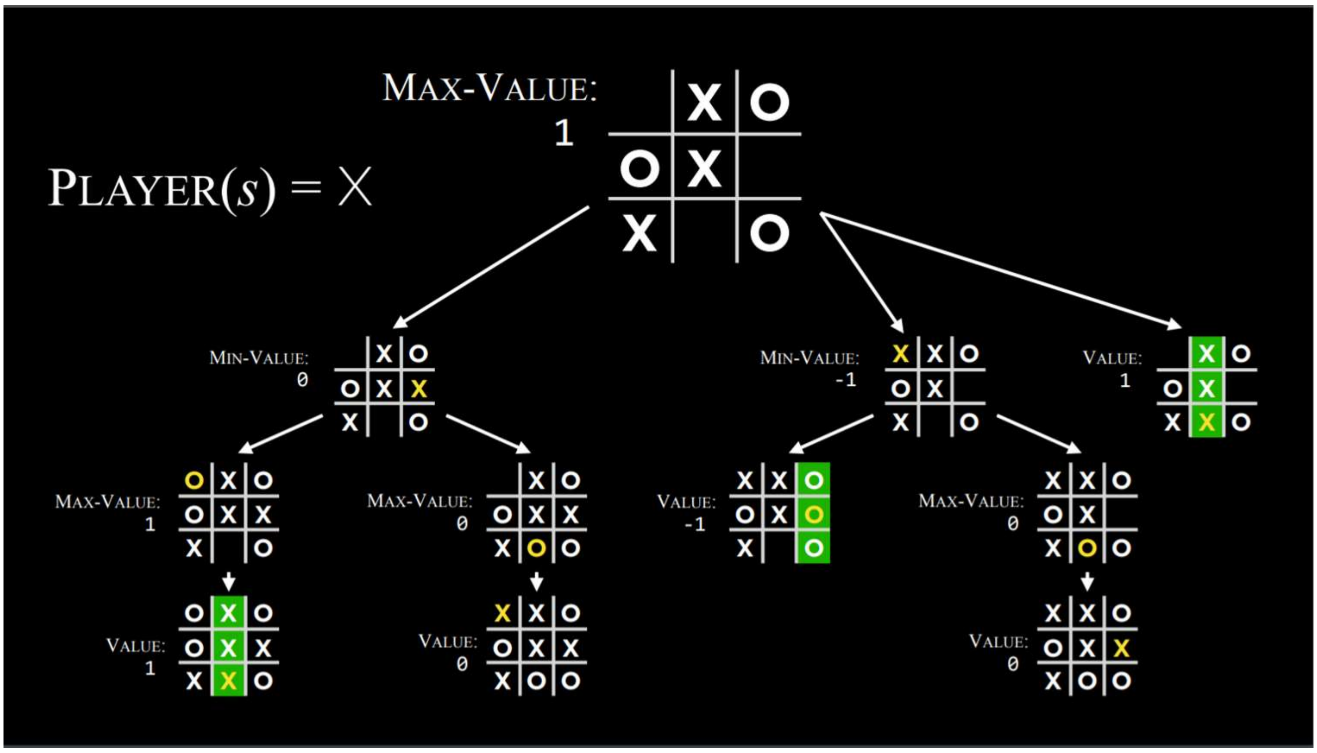
A type of algorithm in adversarial search, **Minimax** represents winning conditions as (-1) for one side and (+1) for the other side. Further actions will be driven by these conditions, with the minimizing side trying to get the lowest score, and the maximizer trying to get the highest score.

Representing a Tic-Tac-Toe AI:

- S_0: Initial state (in our case, an empty 3X3 board)
- *Players(s)*: a function that, given a state *s*, returns which player's turn it is (X or O).
- *Actions(s)*: a function that, given a state *s*, return all the legal moves in this state (what spots are free on the board).
- *Result(s, a)*: a function that, given a state *s* and action *a*, returns a new state. This is the board that resulted from performing the action *a* on state *s* (making a move in the game).
- *Terminal(s)*: a function that, given a state *s*, checks whether this is the last step in the game, i.e. if someone won or there is a tie. Returns *True* if the game has ended, *False* otherwise.
- *Utility(s)*: a function that, given a terminal state *s*, returns the utility value of the state: -1, 0, or 1.

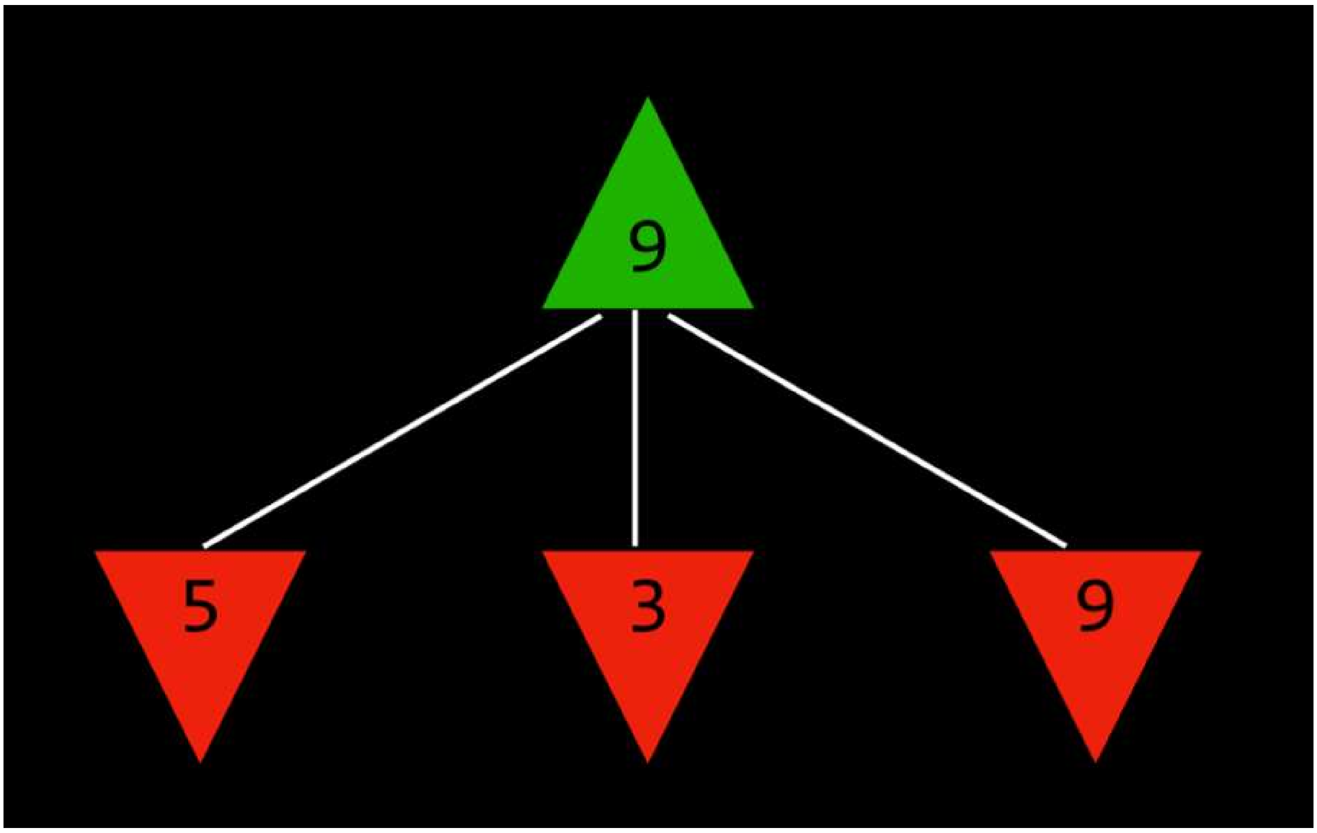
How the algorithm works:

Recursively, the algorithm simulates all possible games that can take place beginning at the current state and until a terminal state is reached. Each terminal state is valued as either (-1), 0, or (+1).



Minimax Algorithm in Tic Tac Toe

Knowing based on the state whose turn it is, the algorithm can know whether the current player, when playing optimally, will pick the action that leads to a state with a lower or a higher value. This way, alternating between minimizing and maximizing, the algorithm creates values for the state that would result from each possible action. To give a more concrete example, we can imagine that the maximizing player asks at every turn: “if I take this action, a new state will result. If the minimizing player plays optimally, what action can that player take to bring to the lowest value?” However, to answer this question, the maximizing player has to ask: “To know what the minimizing player will do, I need to simulate the same process in the minimizer’s mind: the minimizing player will try to ask: ‘if I take this action, what action can the maximizing player take to bring to the highest value?’” This is a recursive process, and it could be hard to wrap your head around it; looking at the pseudo code below can help. Eventually, through this recursive reasoning process, the maximizing player generates values for each state that could result from all the possible actions at the current state. After having these values, the maximizing player chooses the highest one.



The Maximizer Considers the Possible Values of Future States.

To put it in pseudocode, the Minimax algorithm works the following way:

- Given a state *s*
  - The maximizing player picks action *a* in *Actions(s)* that produces the highest value of *Min-Value(Result(s, a))*.
  - The minimizing player picks action *a* in *Actions(s)* that produces the lowest value of *Max-Value(Result(s, a))*.
- Function *Max-Value(state)*
  - *v* = -∞
  - if *Terminal(state)*:

     return *Utility(state)*
  - for *action* in *Actions(state)*:

     v = Max(v, Min-Value(Result(state, action)))

    return *v*
- Function *Min-Value(state)*:
  - *v* = ∞
  - if *Terminal(state)*:

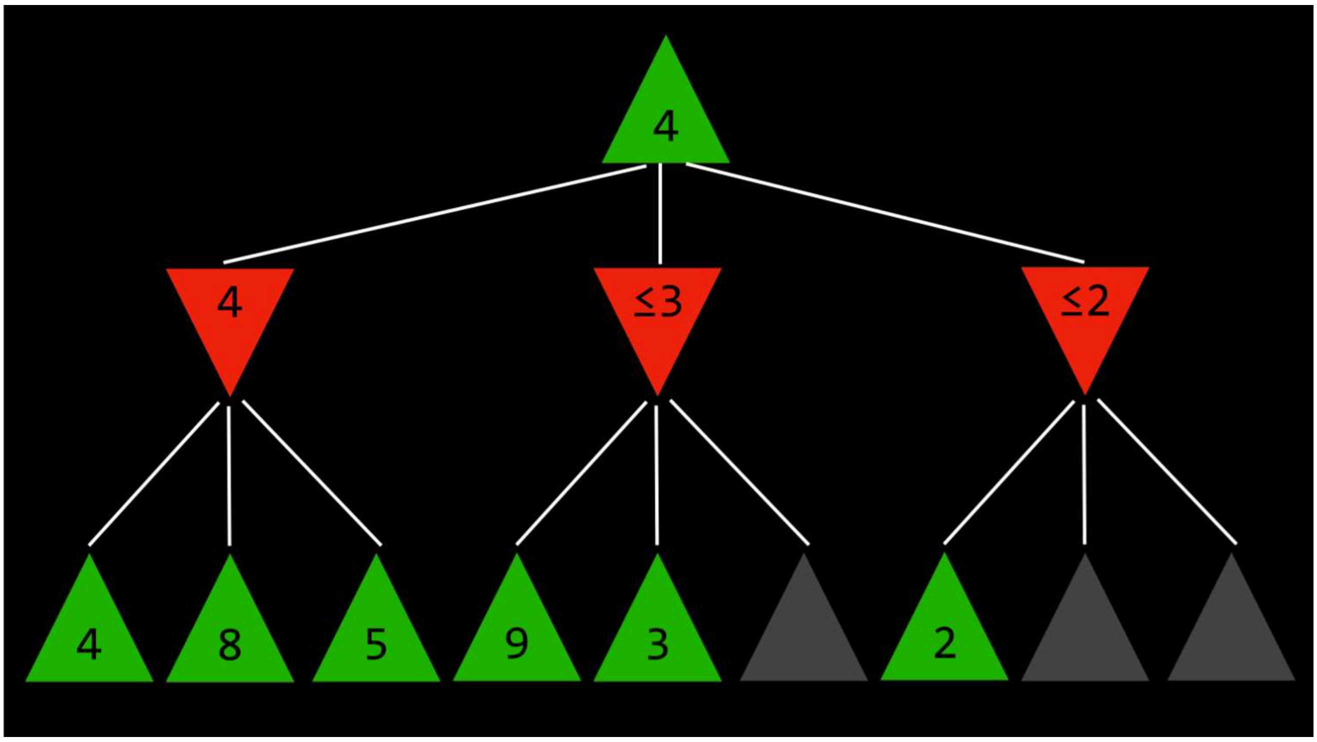
     return *Utility(state)*
  - for *action* in *Actions(state)*:

```
    v = Min(v, Max-Value(Result(state, action)))
  return v
```

Alpha-Beta Pruning

A way to optimize *Minimax*, **Alpha-Beta Pruning** skips some of the recursive computations that are decidedly unfavorable. After establishing the value of one action, if there is initial evidence that the following action can bring the opponent to get to a better score than the already established action, there is no need to further investigate this action because it will decidedly be less favorable than the previously established one.

This is most easily shown with an example: a maximizing player knows that, at the next step, the minimizing player will try to achieve the lowest score. Suppose the maximizing player has three possible actions, and the first one is valued at 4. Then the player starts generating the value for the next action. To do this, the player generates the values of the minimizer's actions if the current player makes this action, knowing that the minimizer will choose the lowest one. However, before finishing the computation for all the possible actions of the minimizer, the player sees that one of the options has a value of three. This means that there is no reason to keep on exploring the other possible actions for the minimizing player. The value of the not-yet-valued action doesn't matter, be it 10 or (-10). If the value is 10, the minimizer will choose the lowest option, 3, which is already worse than the preestablished 4. If the not-yet-valued action would turn out to be (-10), the minimizer will this option, (-10), which is even more unfavorable to the maximizer. Therefore, computing additional possible actions for the minimizer at this point is irrelevant to the maximizer, because the maximizing player already has an unequivocally better choice whose value is 4.



Depth-Limited Minimax

There is a total of 255,168 possible Tic Tac Toe games, and 10^{29000} possible games in Chess. The minimax algorithm, as presented so far, requires generating all hypothetical games from a certain point to the terminal condition. While computing all the Tic-Tac-Toe games doesn't pose a challenge for a modern computer, doing so with chess is currently impossible.

Depth-limited Minimax considers only a pre-defined number of moves before it stops, without ever getting to a terminal state. However, this doesn't allow for getting a precise value for each action, since the end of the hypothetical games has not been reached. To deal with this problem, *Depth-limited Minimax* relies on an **evaluation function** that estimates the expected utility of the game from a given state, or, in other words, assigns values to states. For example, in a chess game, a utility function would take as input a current configuration of the board, try to assess its expected utility (based on what pieces each player has and their locations on the board), and then return a positive or a negative value that represents how favorable the board is for one player versus the other. These values can be used to decide on the right action, and the better the evaluation function, the better the Minimax algorithm that relies on it.