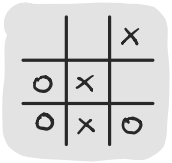
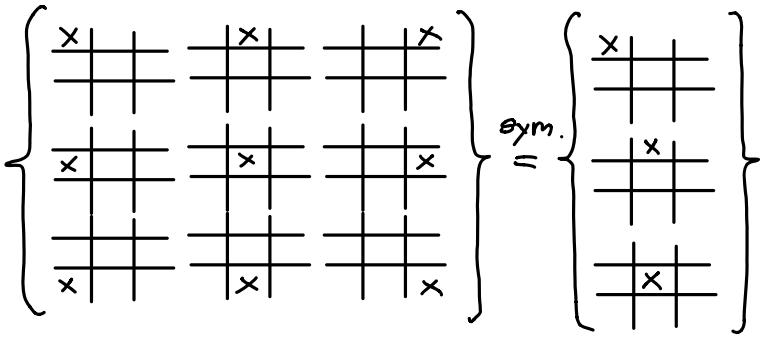
Tic-Tac-Toe



- 3 x 3 grid with Xs and Os
- Whoever scores row, column or a diagonal wins
- Upper limit on # unique gameboards $9! = 362,880$

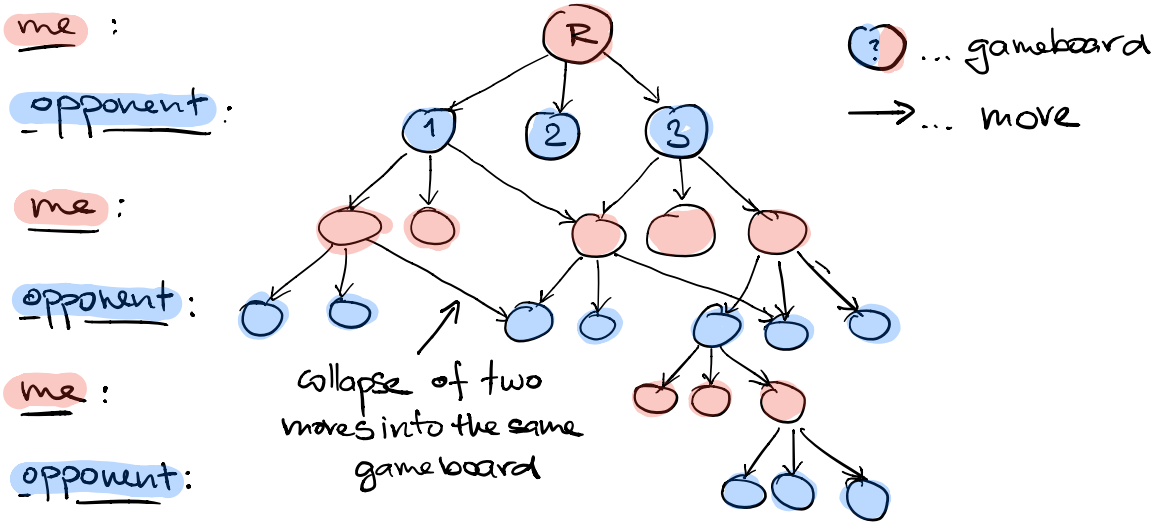
Symmetry

- Game rules are symmetric (i.e. the outcome doesn't change) with respect to 90° rotation and mirroring.
- 8-way symmetry significantly reduces the branching factor and # unique gameboards to 768.



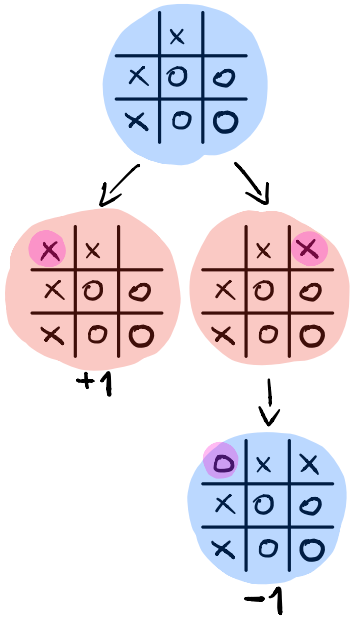
Game Graph

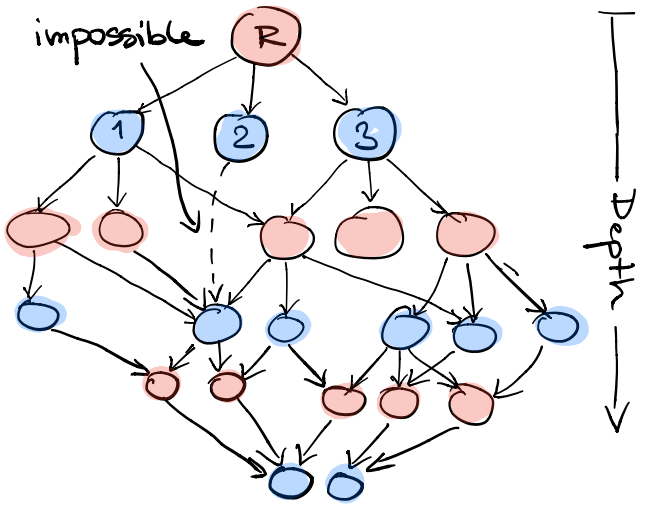
- Directed acyclic graph of possible game evolutions

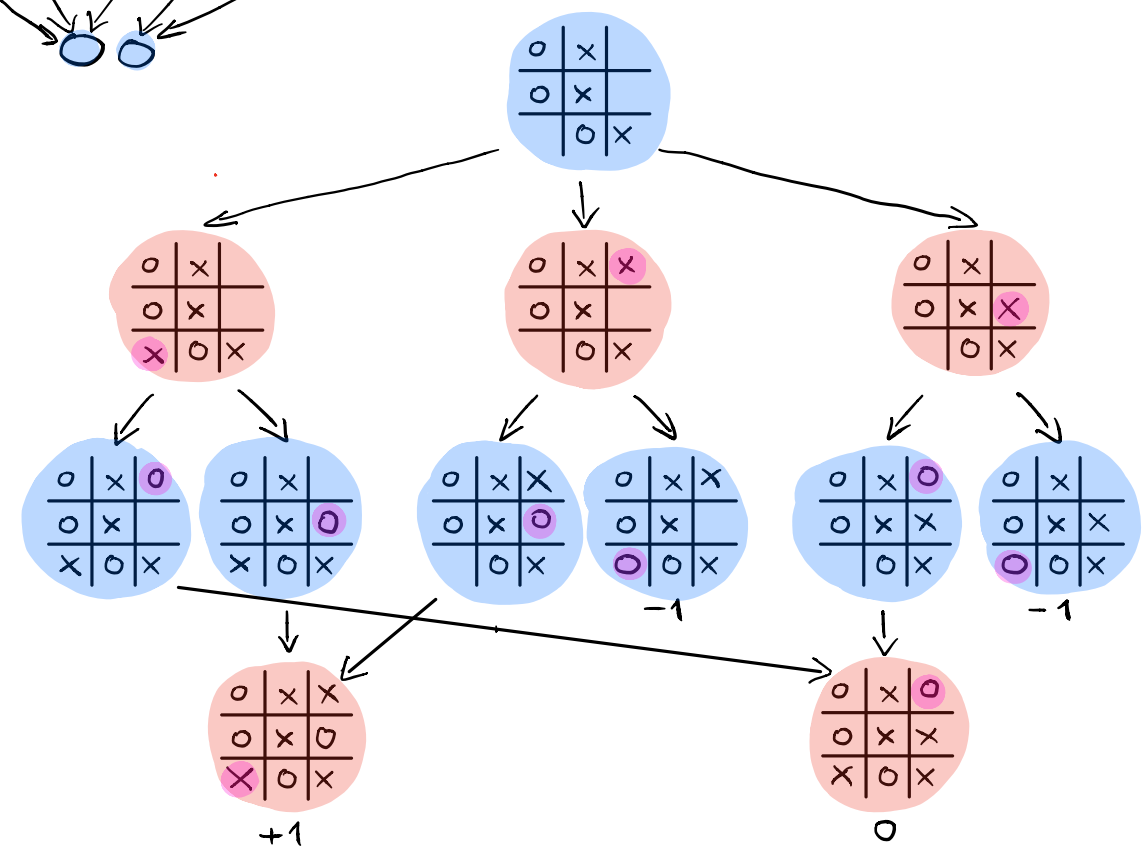


Example Playout Graphs

- Game states start to collapse towards the end as some of the states have multiple ways of reaching there (sequence of moves)
- Collapsing can't happen across multiple levels in the graph because the number of symbols have to be the same for any collapsed state (unlike the Rubik's cube)



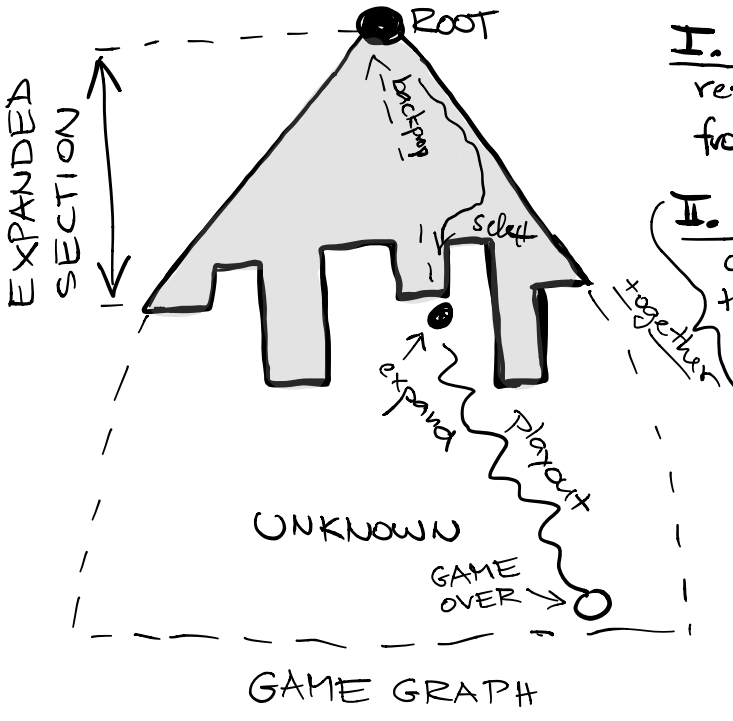




Minimax Algorithm

To-Do :)

Monte Carlo Tree Search Algorithm



I. select(...)
return the most promising node from the expanded part (not fully expanded)

II. expand(...)
choose a random untried node to expand the expanded section of the tree

III. playout(...)
do a playout from the random untried move

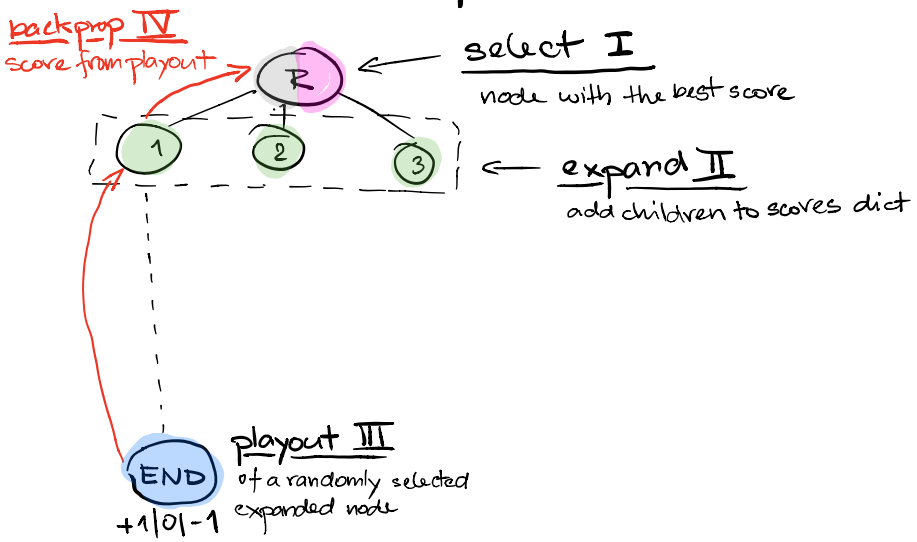
IV. backprop(...)
propagate the score all the way from the random untried node to the root

Implementation

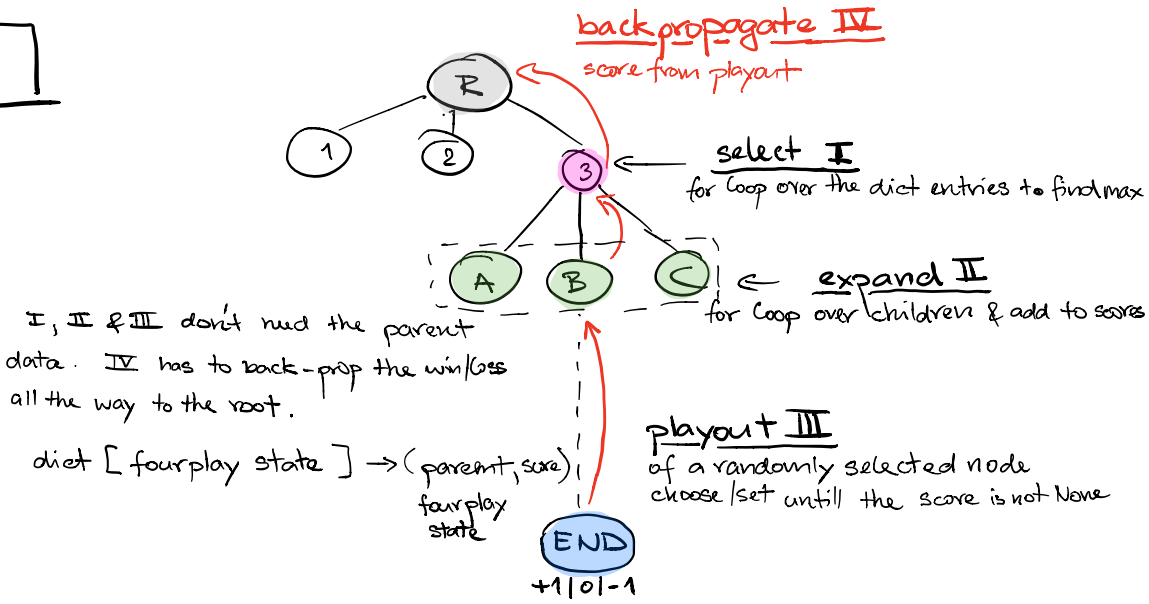
- Gameboard objects that represent a game situation have to be hashable
- Symmetry can be solved by a subclass of dictionary that checks for presence of symmetrically equivalent variants before inserting or when updating an entry.
- Alternatively symmetry can be handled at the level of available moves. The set of moves can be restricted to never allow a move that would leave the family of canonical unique gameboards.

Monte Carlo Tree Search Steps

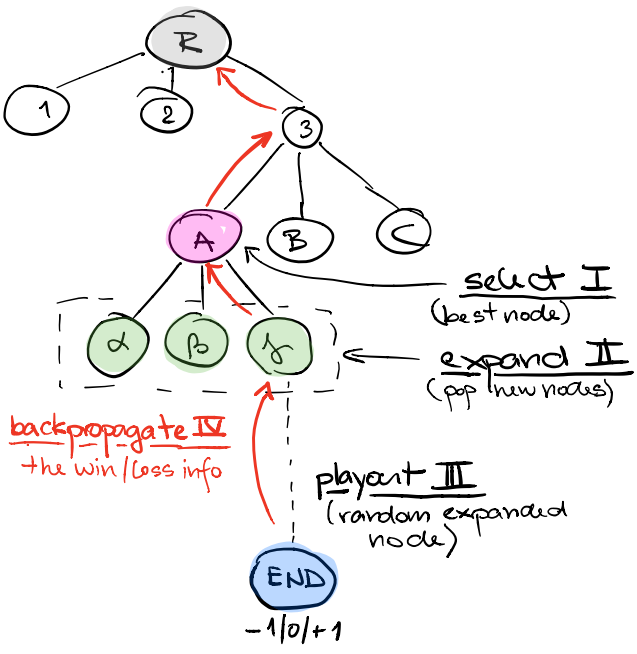
#0



#1



#2



Value Function Temporal Difference Iteration

Create a competitive arms race between two learning agents. The function values can be stored in a dict like structure thanks to a relatively small number of unique gameboards.

Value function update rule for each of the competing agents:

$$V_o^t \leftarrow V_o^t + \alpha\left(V_o^{t+1} - V_o^t\right)$$
$$V_x^t \leftarrow V_x^t + \alpha\left(V_x^{t+1} - V_x^t\right)$$

$$V_?^{END} = \begin{cases} +1 & \text{win} \\ 0 & \text{tie} \\ -1 & \text{loss} \end{cases}$$

$V_o^t = V_o(S_t)$ ... Value fcn. of O player

$V_x^t = V_x(S_t)$ .. Value fcn. of X player

When both sides learn they exploit each other's weaknesses caused by different random initializations of their own policies. Learning will eventually converge to the optimal minimax agent on both sides.

$$V_o(S_o) = \max_{S_o} V_o^0 \max_{S_1} V_x^1 \max_{S_2} V_o^2 \ldots \max_{S_N} V_?^N$$

$$\begin{cases} +1 & \text{win} \\ 0 & \text{tie (final value)} \\ -1 & \text{loss} \end{cases}$$

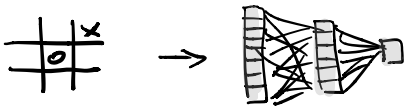
Previous values receive the update delayed.

Initially this value is 0 but over time and episodes it learn the actual value of the state.

Alpha Zero Algorithm

Deep Neural Network

- Predict outcome and probability of each move



- Neural network weights are learned solely from self-play

Monte Carlo Tree Search

- Series of simulated games starting from $\vec{s}_{root}$ to a leaf
- Simulations proceed by taking action a from state $\vec{s}$ either with low visit count, high move probability or high value. (averaged over leaf states of simulations that selected a from $\vec{s}$) as estimated by the current NN f_θ
- Search returns vector $\vec{\pi}$ representing a probability distribution over moves either proportionally or greedily wrt visit count at the leafs.

Reinforcement Learning

- Games are played as MCTS $a_t \sim \vec{\pi}_t$. At the end of the game terminal position $\vec{s}_T$ is stored together with the score +1|0|+1
- NN parameters θ are updated to minimize the error between predicted outcome v_t and the game outcome z.

$$(\vec{p}, v) = f_\theta(\vec{s}) \qquad \text{Loss} = (z - v)^2 - \vec{\pi} \cdot \log \vec{p} + c\|\theta\|^2$$

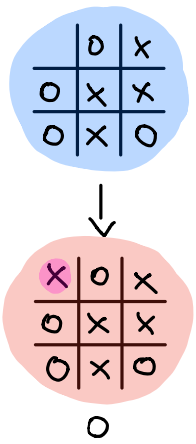
(z: game outcome, v: predicted outcome, $\vec{\pi}$: MCTS search, $\vec{p}$: NN output)

- Updated parameters are used in subsequent self-play games. (Original algorithm evaluated players and selected the best for self-play)

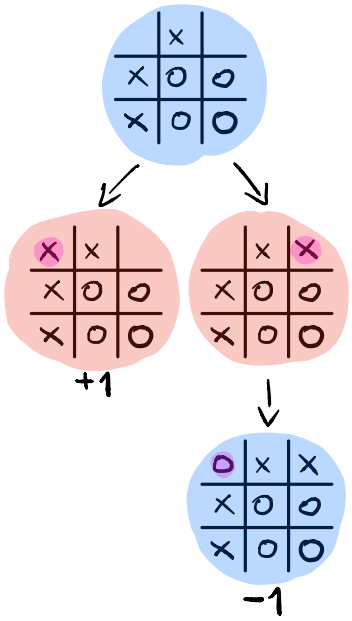
5000 GPUs (1st gen) to generate self play games
64 GPUs (2nd gen) to train the NN

Test Scenarios

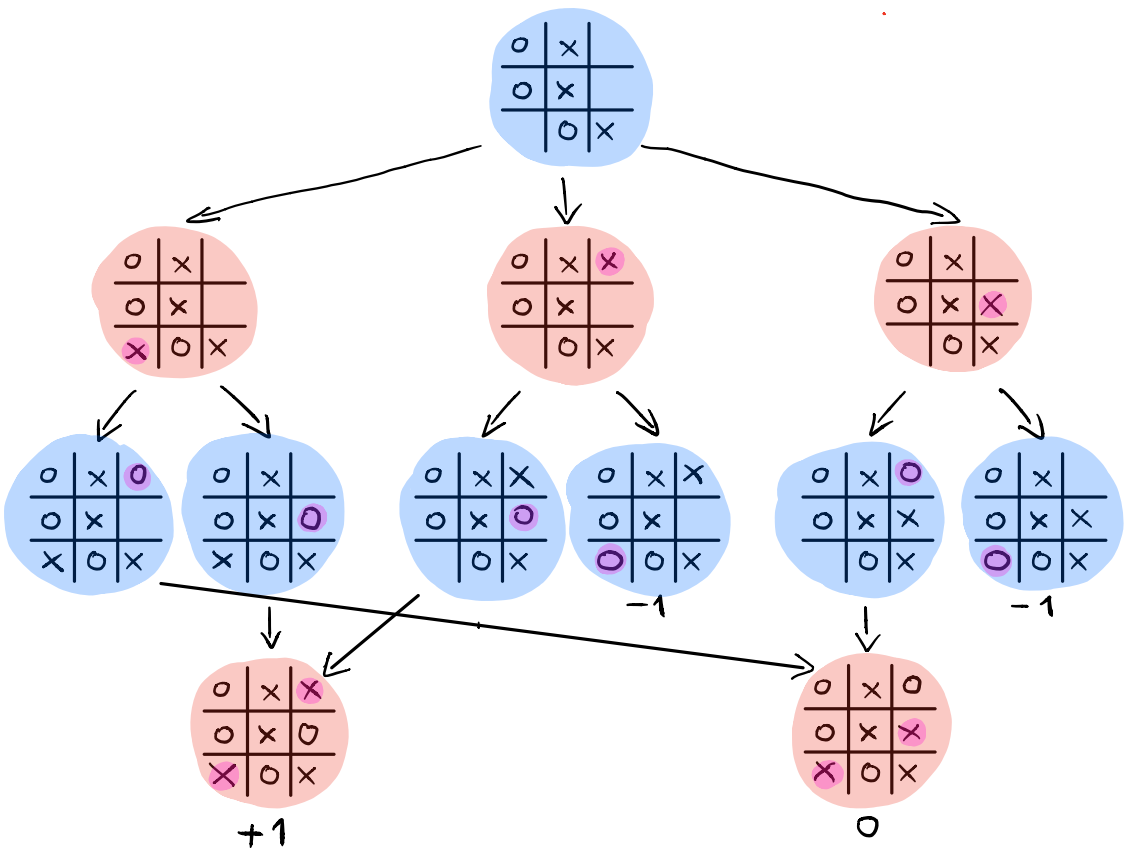
1) Finish



2) Easy win



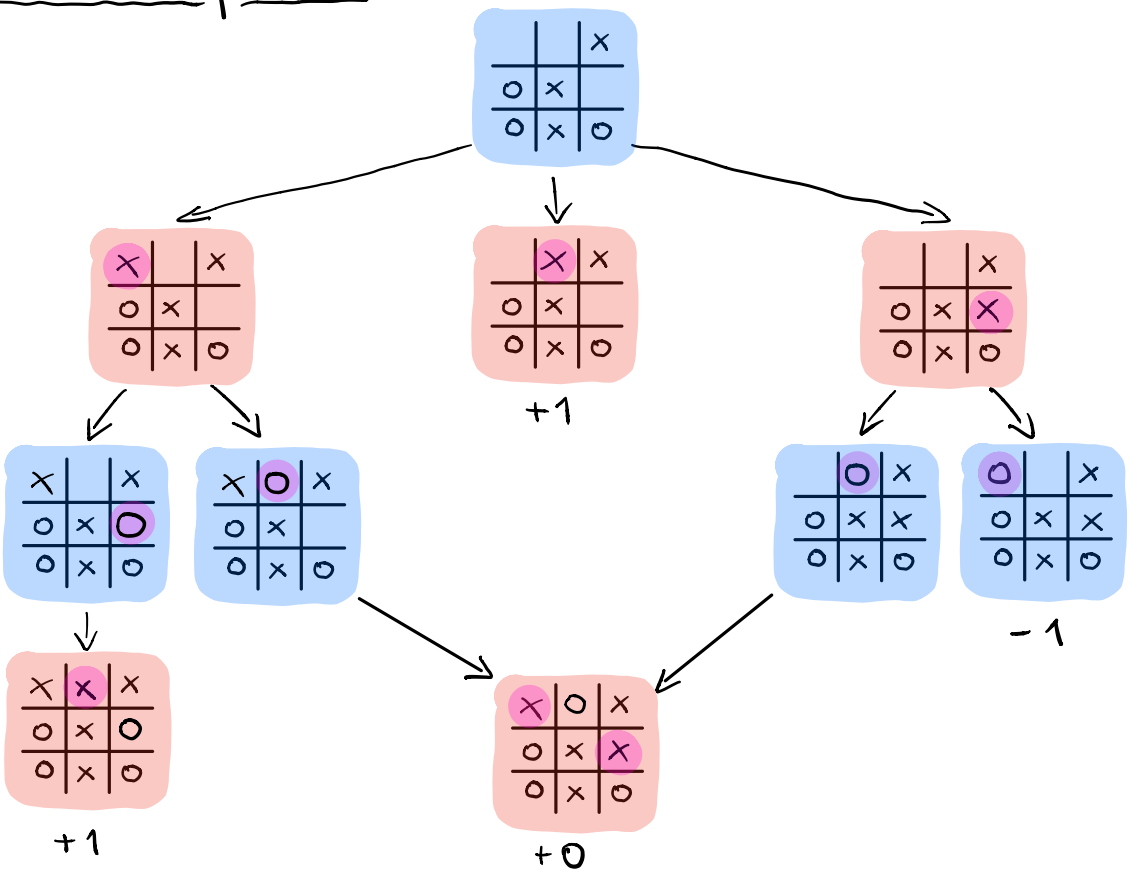
3) Don't screw up



4) Don't Mess Up #1

... large...

5) Don't Mess Up #2



6) Don't F**k Up!