Minimax Alg.

A.

- evaluate(board) ⟶ given a gamestate, must create a heuristic/score to judge "how good" that state is.

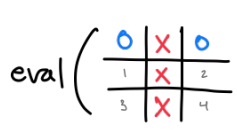
Tic-Tac-Toe evaluation

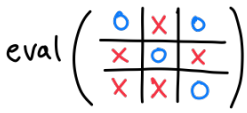
= maximizing team

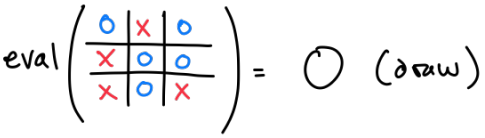
= minimizing team

- We will use these rules to score game states:
  - ⟶ a DRAW is worth 0 points
  - ⟶ a WIN is worth ...
    - * 1 · (available squares before placing token) for the maximizing player
    - * -1 · (available squares before placing token) for the minimizing player

examples

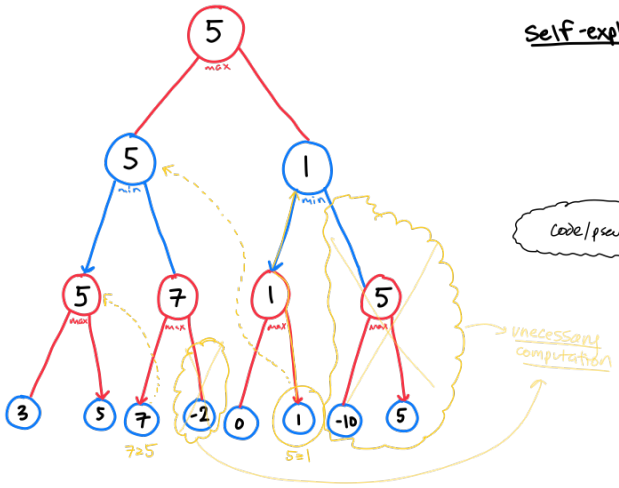






B.

minimax v1 := minimax(board, depth, MaximizingPlayer)



self-explanatory

depth: # of game states "in advance" minimax will look

a boolean which is 'true' if the current player is the maximizer and 'false' otherwise

code/pseudocode implementation:

```
int minimax(board, depth, maximizing)
  if depth == 0 or game is over (board is full or player got 3 in a row)
      return eval(board); // this is the score of this board, which will be passed up the game tree.
  → if maximizing
      • loop through children game states (recurse on minimax(new_board, depth-1, maximizing=false)
      • return max(children)
  → else
      • same as maximizing, except return min(children)
```

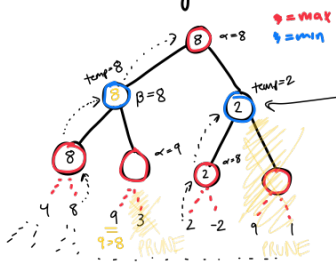
~ on larger game trees, there will be a lot of wasteful computation, slowing down the AI's decision-making (see above). Can we do better?

~ Yes, by "pruning" useless game tree paths!

minimax v2 (optimized w/ Alpha-Beta Pruning)

~ introduces 2 parameters (alpha and beta), with alpha representing the "maximization" of that tree node's subtree (beta obviously being the minimization of the subtree rooted @ the current node)

- Alpha and beta are essentially guaranteed safe bets (for max and min players respectively): if the current value of $\alpha = 8$, we shouldn't explore any subtrees where $\beta \leq 8$, because assuming the minimizing plays optimally, they would never choose to go down it.
  (in other words, when $\alpha \geq \beta$)



after finding a 2 here, we know that the nodes value can only decrease (since blue will minimize). However, it's parent (which maximizes) already has a guarentee of 8. 8 will always trump ≤2; therefore, there is no purpose of searching the remaining subtree.

ELIMINATED DEPTH VARIABLE
(can simply check if game is over)

code/pseudocode implementation:

```
int minimax(board, α, β, maximizing)
  if game is over (board is full or player got 8 in a row)
      return eval(board); // this is the score of this board, which will be passed up the game tree.
  if maximizing
    • loop through children game states (recurse on minimax(new_board, α, β, maximizing=False)
    • α = max(α, max(children))
    • break if α ≥ β
    • return max(children)
  else
    • β = min(α, min(children))
    • break if α ≥ β
    • same as maximizing, except return min(children)
```