CS 235: Artificial Intelligence

Adversarial Search
Game Playing

Dr. Moumita Roy
CSE Dept., IIITG

Single-agent/Multi-agent environment

- In the previous lectures, we have formulate the different tasks as a search problem where the various search algorithms are utilized to solve the task

- In such case, we have considered the single agent only where one agent/machine starts from an initial position and change the states to reach goal state

- Now onward, we have consider the scenario of multiple agents

- In particular, the situation of two agents/players those are competing each other (i.e. gain of one player is the loss of other players)

Adversarial Search

- Such problem can be formulated as a different kind of search problem

- Competitive environments, in which the agent's goals are in conflict, require adversarial search problems (called as games)

Game playing

- Humans are always interested about the game
- Game challenges our ability of think
- Simple game: Tic-Tac-Toe, 8-puzzel
- Complex game: Chess

Game playing

- Two players
- The players alternate
- Focus on the game of perfect information (no chance factor involved/deterministic)
- Zero-sum game
- Game problem can be formulated as a search problem (where the initial position of the board is start state)

Zero-sum Game

- Fully competitive game; payoff/utility of one player is exactly opposite of the other player; example, in chess, either one player wins then the other losses or both draw

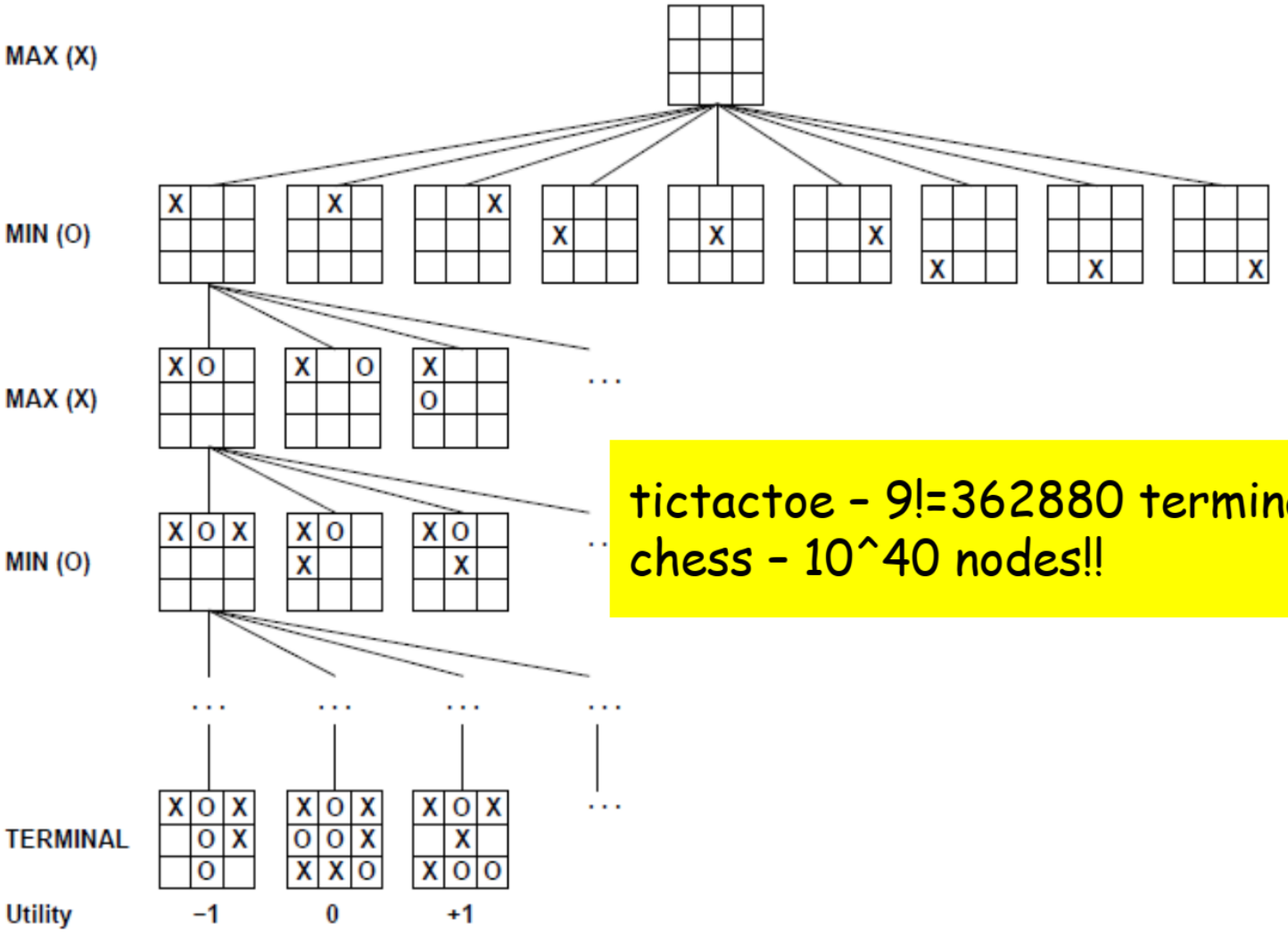
- Why are they interesting?
  - ➢ Most games we play are zero-sum: chess, tic-tac-toe, …
  - ➢ (win, lose), (lose, win), (draw, draw) ➢ (1, -1), (-1, 1), (0, 0)

- Why are they technically interesting?
  - ➢ Relation between the rewards of P1 and P2
  - ➢ P1 maximizes his reward
  - ➢ P2 maximizes his reward = minimizes reward of P1

Formalization of Such Search Problem

- S_0 - initial state specifies initial setup of the game
  Example: In tic-tac-toe, start position is empty 3/3 board with no x/0 on it
- Player(s): Defines which player has the move in a state
  Example: MAX (x), MIN (0)
- Actions(s): Returns the set of legal moves in a state
  Example: put x/0 based which turn is it
- Results(s, a): Transition model, that defines the outcome of a move in a state
  Example: modified board after applying any action
- Terminal-Test(s): A terminal test, which is true if the game is over and false otherwise. States where the game has ended are called terminal states
  Example: Terminal state may be (1) all three x in the same column/row/diagonal or (2) all three 0 in the same column/row/diagonal or (3) all squares are filled up in the board
- Utility(p , s): A utility/objective/payoff function that determines the final numeric value for a game that ends in the terminal state s for player p
  Example: if MAX wins utility of MAX +1; if MAX losses the game utility of MAX -1; if MAX draws with MIN the utility of MAX is 0

Game tree (2-player, deterministic, turns)



tictactoe – 9!=362880 terminal nodes
chess – 10^40 nodes!!

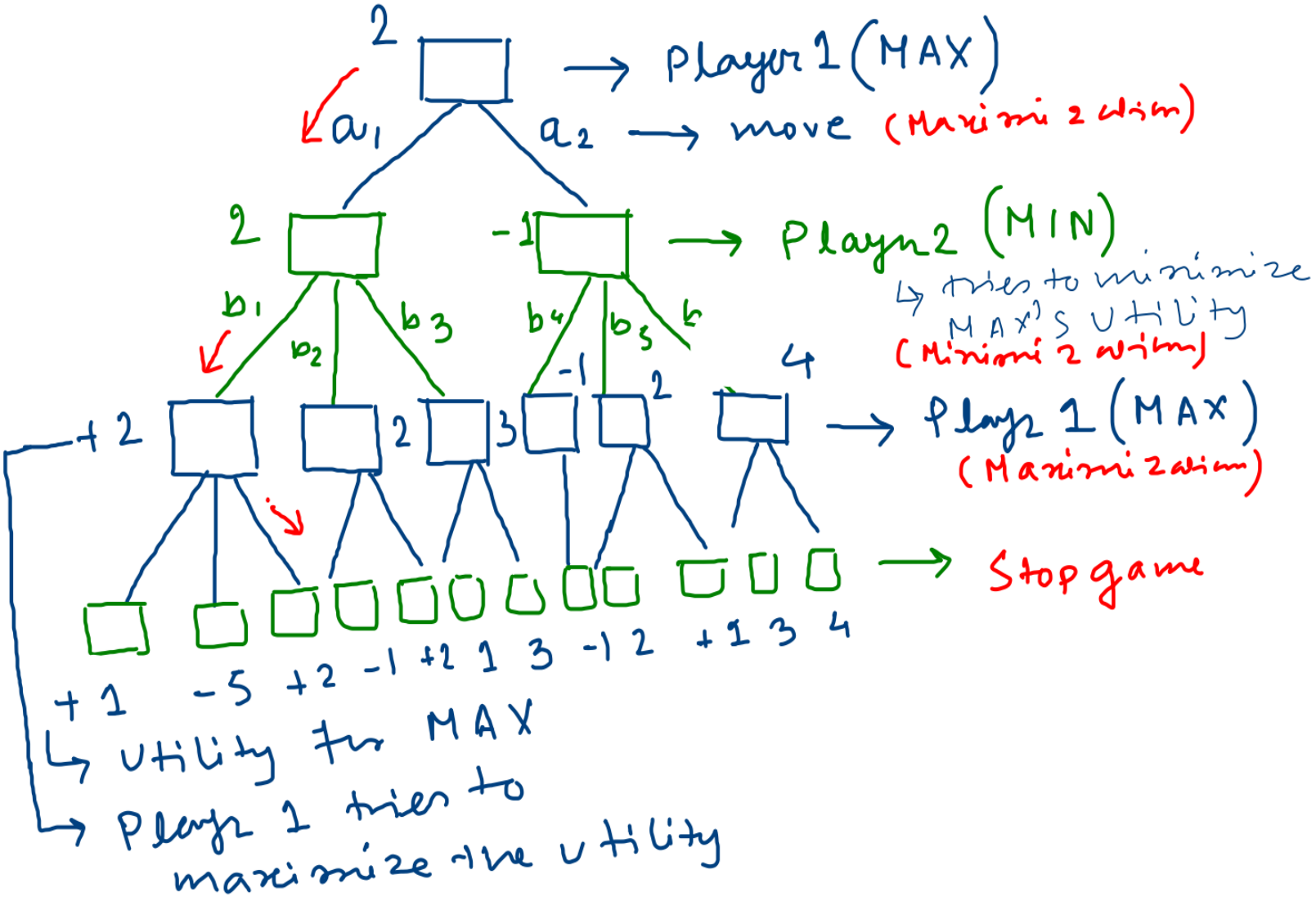
Basic strategy in Games

- Grow a search tree (Game tree)
- Only one player can make move at each turn
- We can assign utility for each terminal state
- When we are able to form the search tree, now we need to find the best move for both the players
- For this, the simple approach is to propagate the utility value backward from terminal state to obtain the some value (like utility) for the intermediate state to root node
- Assume, opponent (player 2) always make the move that is worst for us
- We (player 1) always pick the best move

In case of single agent environment, we can plan the entire path for the player (since only one player moves)

In case of two-agent environment, player 1 can make a move and after that it needs to wait for the player 2 to make the next move; then only player 1 can plan the next move; so the entire move can't plan beforehand by considering only player 1

MINIMAX Algorithm



Game Tree

- By a given description of game, we can grow a game tree
- Successor node represent position where different players must move (MAX node/MIN node)
- Game tree could be infinite
- The ply of a node is the number of moves needed to reach that node (i.e. arcs from the root of the tree). The ply of a game tree is maximum ply of its node (like depth of a tree).

Brute force approach

- Firstly, we discuss the exhaustive search approach, here we examine each and every node to calculate back-up utility (MINIMAX value).
- Suitable, only for small game
- However, we need to find out some heuristic based algorithm.

An optimal procedure: The Min-Max algorithm

Designed to find the optimal strategy for MAX and find best move:

1. Generate the whole game tree, down to the leaves.

2. Apply utility (payoff) function to each leaf.

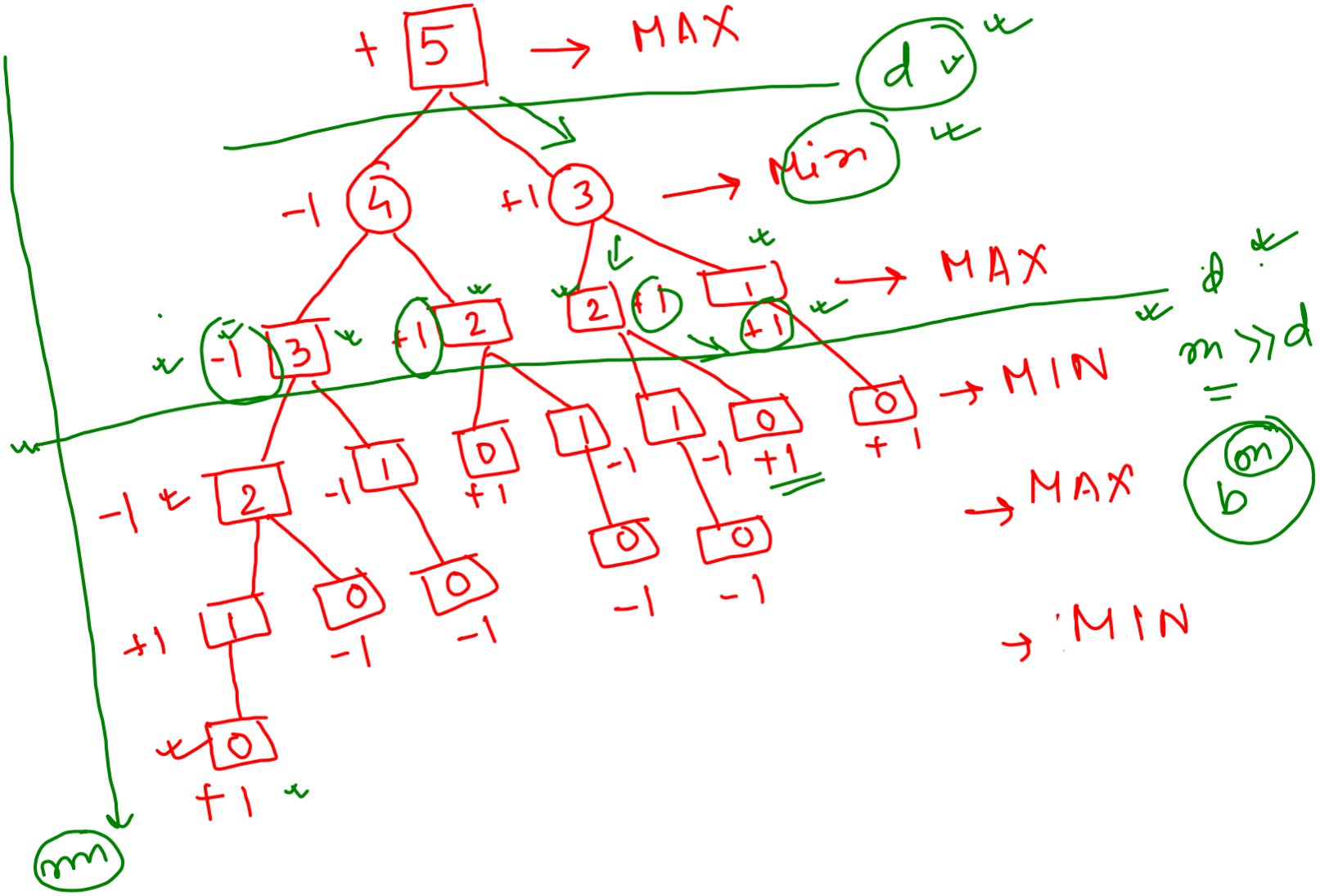
3. Back-up values from leaves through branch nodes:
   a Max node computes the Max of its child values
   a Min node computes the Min of its child values

4. At root: choose the move leading to the child of highest value.

Simple Game:5-stone Nim

- Played with two players and piles of 5-stones
- Each player removes 1/2 stones from the pile
- If it is the turn of player 1 and there are no stones left then player 1 lose

Demonstration



MINIMAX Search

- MAX (MIN) player selects the move that leads to the successor node with the highest (lowest) score
- The minimax score has been computed starting from the leaves of the tree and backing up their scores to their predecessor in accordance with the minimax strategy
- It explores each node in the tree
- If we could draw the entire search tree, we would know what to do, but the search trees get big fast

Properties of MINIMAX Algorithm

- In MINIMAX algorithm, nodes are examined (calculation of utility) in depth first manner

- The number of nodes to be examined for calculation of MINIMAX values:

$$O(b^m)$$

b: branching factor

m: maximum depth of the game tree

Heuristic MINIMAX Search

- MIN wants MAX to lose (and vice versa)
- No plan exists that guarantees MAX's success regardless of which actions MIN executes (the same is true for MIN)
- The state space is enormous: only a tiny fraction of this space can be explored within the time limit
- Alternatively, we can search to some fixed cutoff (h)
- We estimate the merits of a position at a terminal state by some heuristic static evaluation function when the final outcome has not yet been determined
- Thereafter, we can back-propagate these values using MINIMAX strategy to find out the better move from root
- We can compute these static evaluation functions at any level. However, if we go deeper and calculate the heuristic value and backed values; it will be provide us the better estimation

Choosing an Action: Basic Idea

1) Using the current state as the initial state, build the game tree uniformly to the maximal depth h (called horizon) feasible within the time limit
2) Evaluate the states of the leaf nodes
3) Back up the results from the leaves to the root and pick the best action (assuming the worst from MIN)

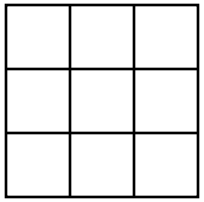
→ Minimax algorithm

Evaluation Function

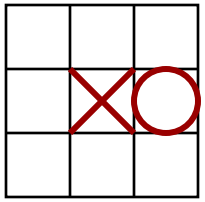
- Function e: state s $\rightarrow$ number e(s)
- e(s) is a heuristic that estimates how favorable s is for MAX
- e(s) $>$ 0 means that s is favorable to MAX (the larger the better)
- e(s) $<$ 0 means that s is favorable to MIN
- e(s) $=$ 0 means that s is neutral

Example: Tic-tac-Toe

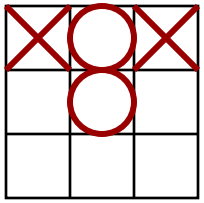
e(s) = number of rows, columns, and diagonals open for MAX – number of rows, columns, and diagonals open for MIN



8–8 = 0

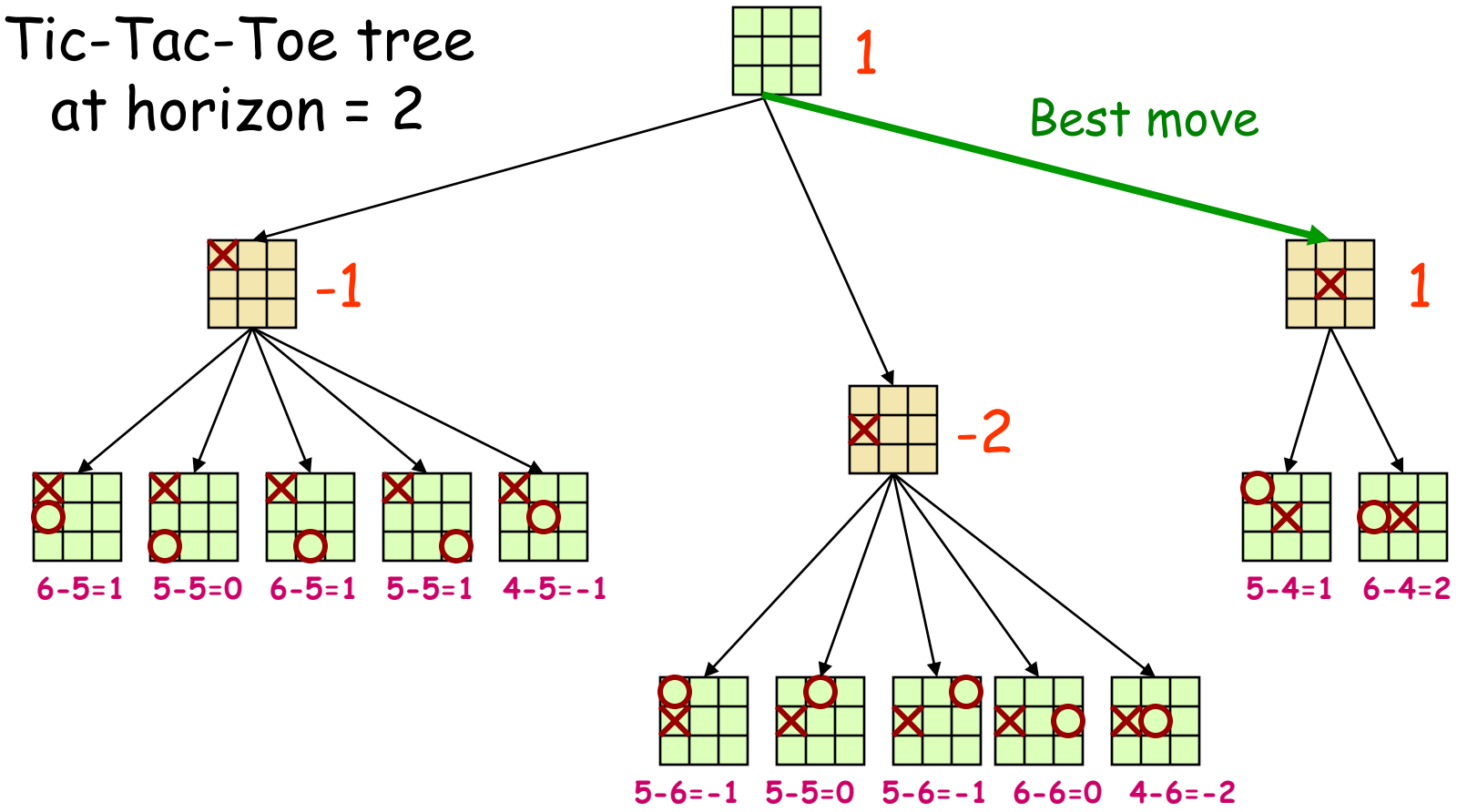


6–4 = 2

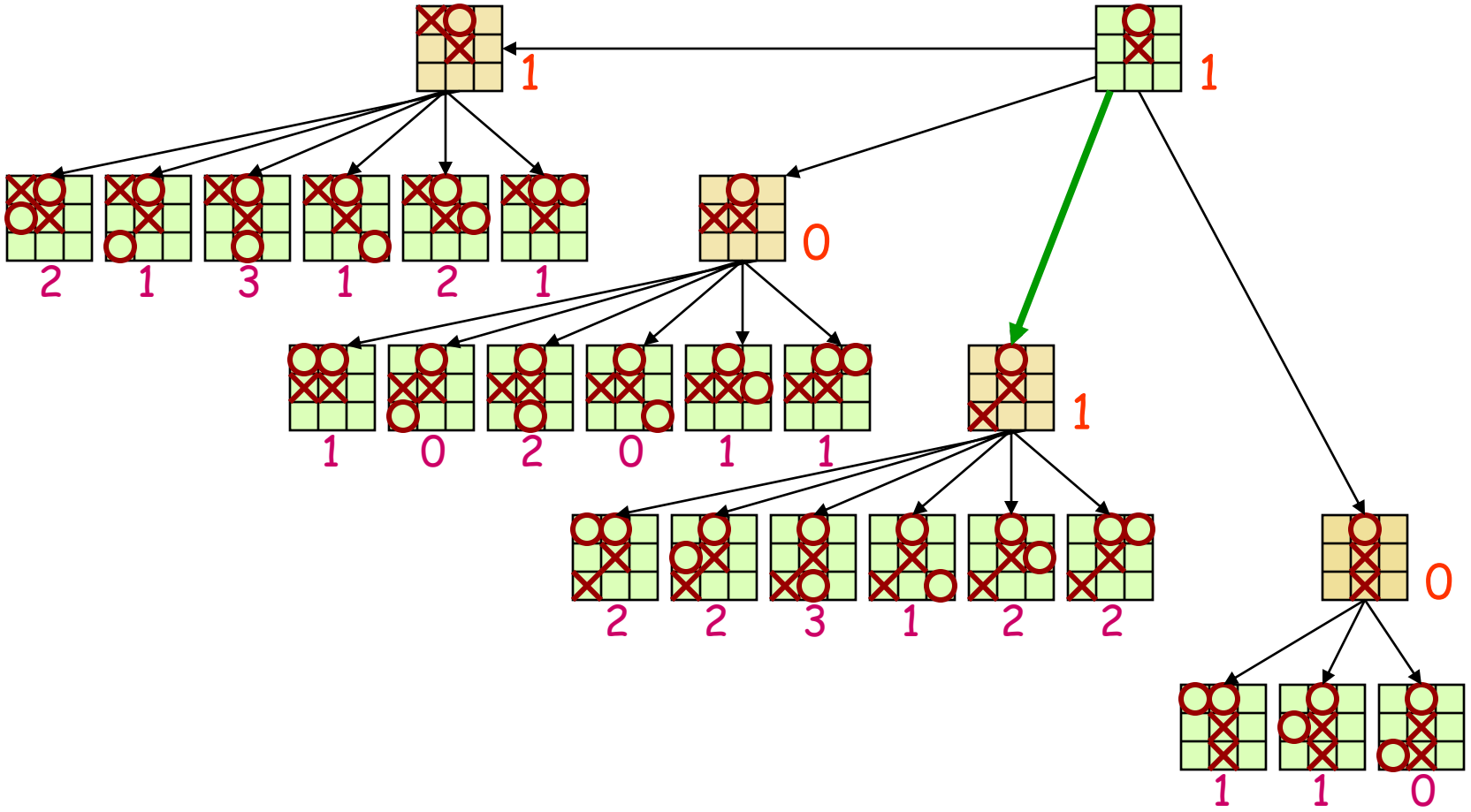


3–3 = 0

Backing up Values



Continuation



Heuristic Minimax Algorithm

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth h
2. Compute the evaluation function at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Properties of Heuristic based MINIMAX Algorithm

- In MINIMAX algorithm, nodes are examined (calculation of utility) in depth first manner
- In case of heuristic based strategy, we have placed cutoff (on depth/ply h)

- The number of nodes to be examined for calculation of MINIMAX values:

$$O(b^h)$$

  b: branching factor

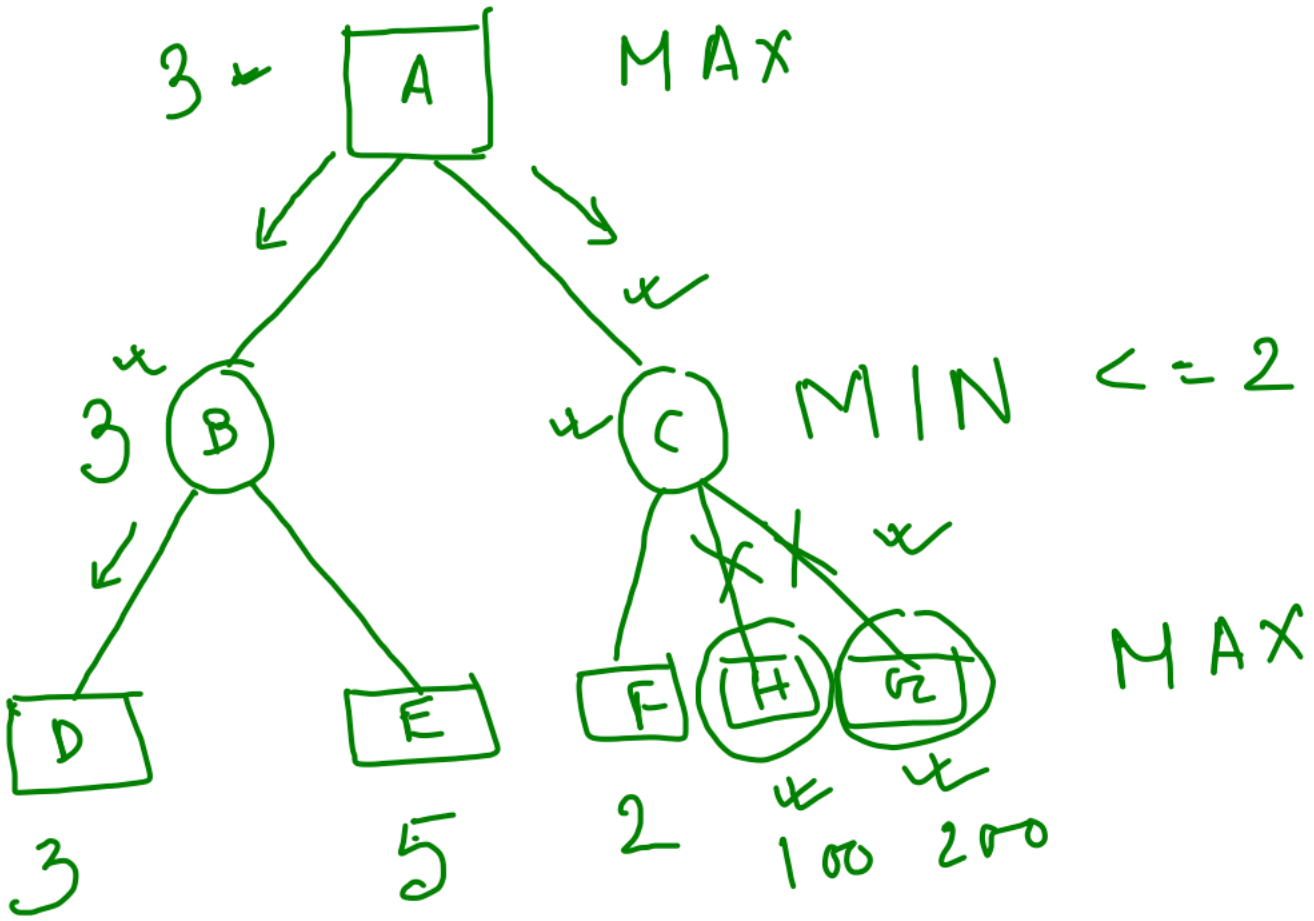
  h: cutoff depth

- Here, we are able to reduce the number of nodes to be examined, but it is heuristic based strategy

What is the problem in MINIMAX Algorithm?

- The number of game states it has to examine is exponential in the depth of the tree
- Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half
- The trick is that it is possible to compute the correct MINIMAX decision (not heuristic based/ using utility in terminal states/no cutoff on depth) without looking at every node in the game tree
- That is, we can borrow the idea of pruning
- The particular technique, we examine is called alpha-beta pruning

Demonstration (with example)



Alpha Beta Search

- We cut off the search when we cannot do better than the best so far
- To implement alpha beta pruning, the alpha and beta values are associated with each node
- alpha value is meaning full for MAX node and beta value for MIN node

Alpha value

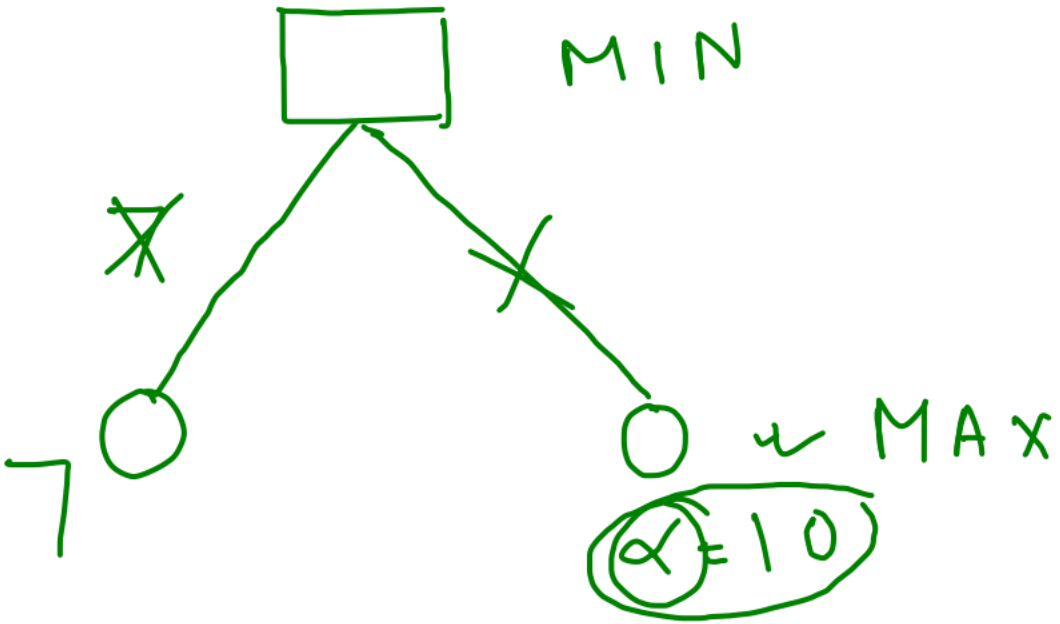
- At Max node, we will store an alpha value
- The alpha value at MAX node is lower bound on the exact MINIMAX score

- The true value might be greater than or equal to alpha

- If MIN can choose a move with score less than alpha; then the MIN will never choose to let MAX go to a node where the score will be alpha or more

Demonstration

Alpha value

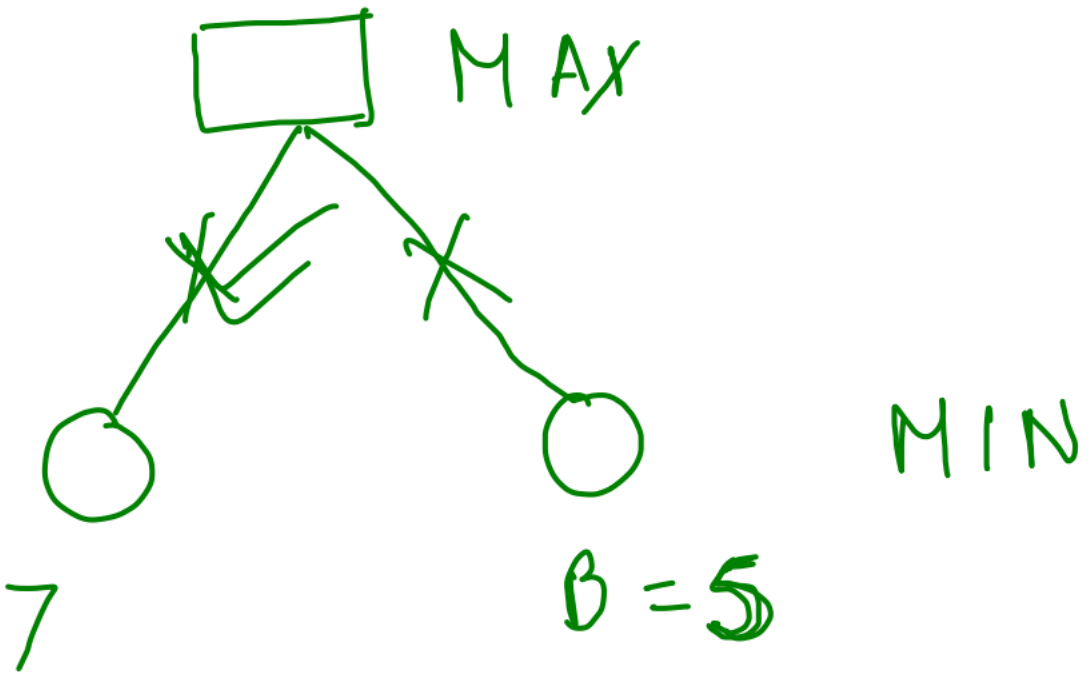


Beta value

- At MIN node, we will store an beta value
- The beta value at MIN node is upper bound on the exact MINIMAX score
- The true value might be less than or equal to beta
- If MAX can choose a move with score higher than beta; then the MAX will never choose to let MIN go to a node where the score will be beta or less

Demonstration

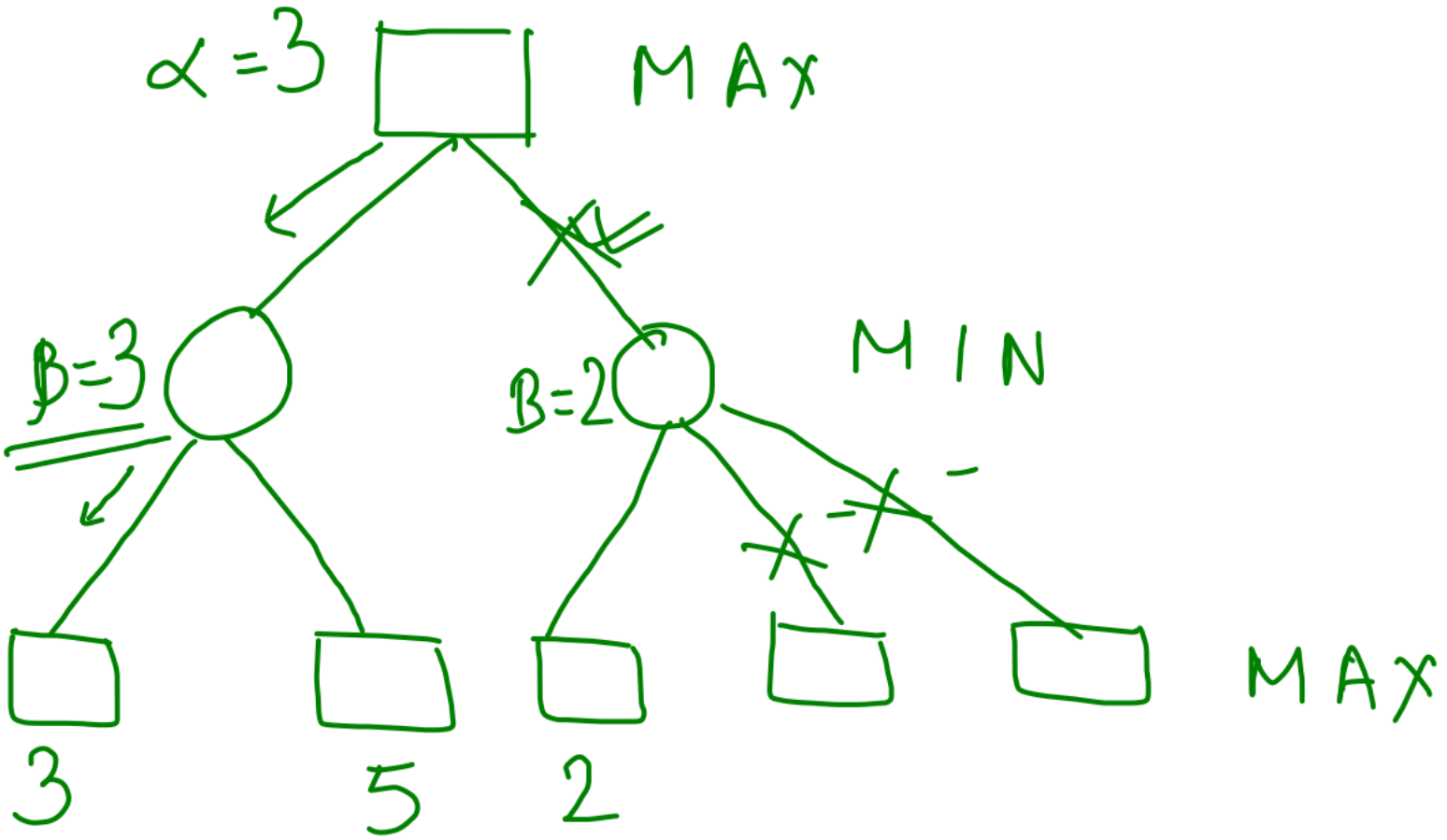
Beta value



Alpha-Beta in Action

- Where we can cut off the search using Alpha/Beta?
- The meaning of alpha value at MAX node: if MAX plays judiciously at this point, it can guarantee the score at least alpha
- The meaning of beta value at MIN node: if MIN plays judiciously at this point, it can guarantee the score no more than beta

Demonstration

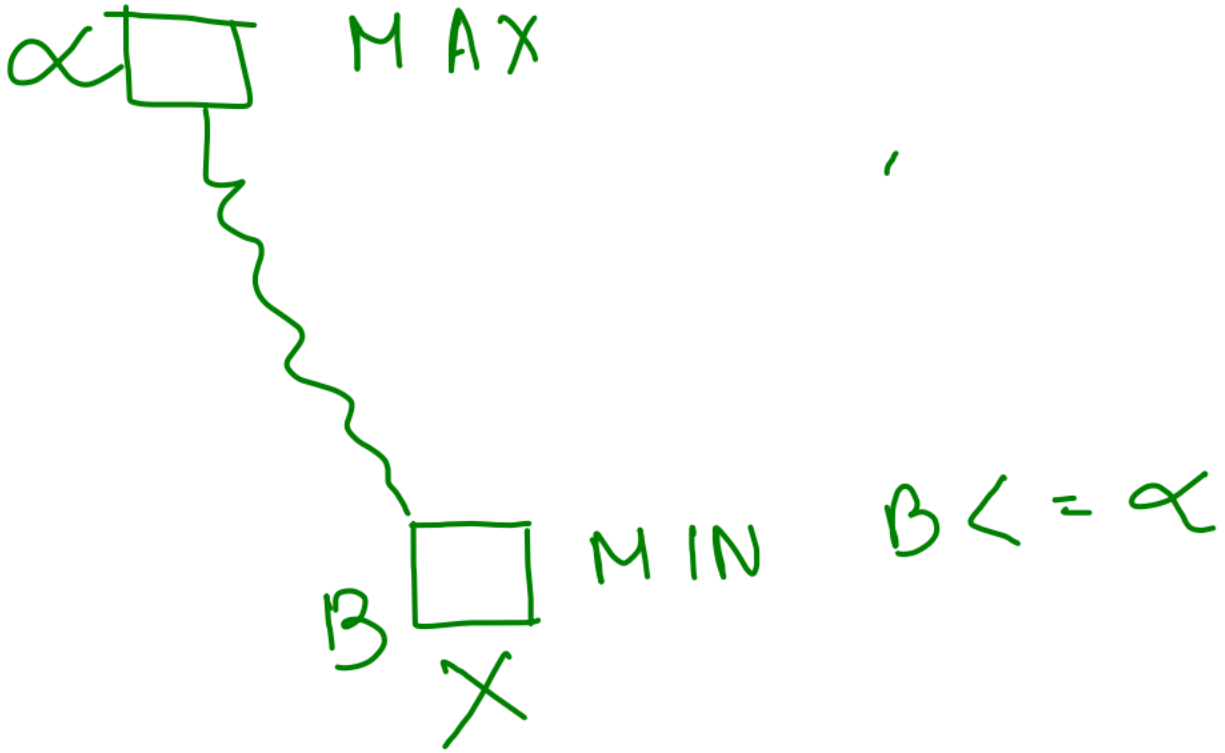


Alpha-Beta Pruning Rules

- There are two types of pruning: alpha cutoff and beta cutoff
- Search can be discontinued.
- Alpha Pruning: Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX ancestors.
- Beta Pruning: Search can be stopped below any MAX node having a alpha value greater than or equal to the beta value of any of its MIN ancestors.

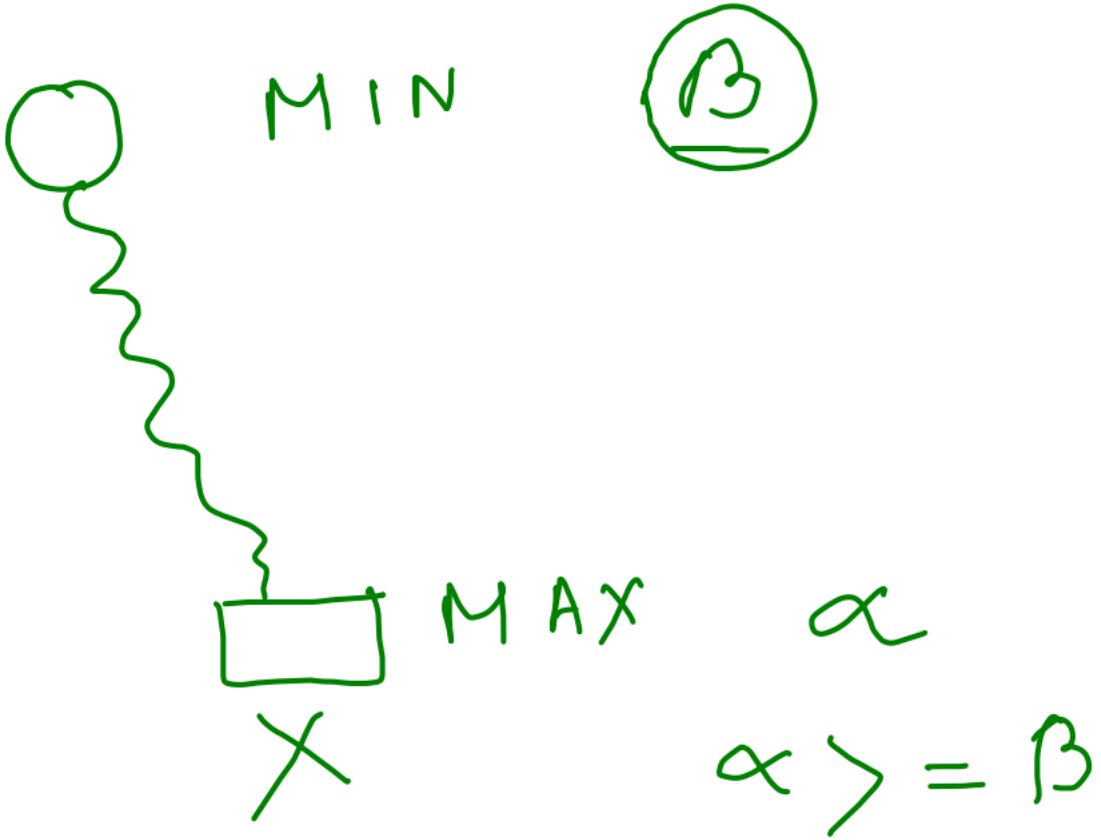
Demonstration

Alpha pruning



Demonstration

Beta cutoff



Computation of alpha, beta values

- Alpha-beta calculations are very similar to the MINIMAX, but the pruning rules are cut down the search

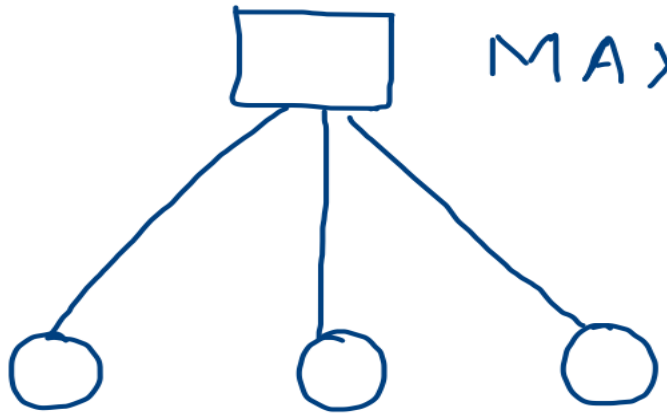
- Final backed-up values at a node:
  - May be exact MINIMAX value
  - May be the approximation where the search is cutoff
    I. Alpha value is less than or equal to the exact MINIMAX value at MAX node
    II. Beta value is greater than or equal to the exact MINIMAX value at MIN node
    III. As already mentioned, there is no need to calculate the exact MINIMAX values for all nodes to find the best strategy for MAX

Calculations of alpha value at MAX node

- After we calculate the final backed up value of the first child, we set alpha for this node with this value
- When we have the final backed up value of the second child, we update alpha for this node with this value if the new value is larger
- If we have the final child with final backed up value or beta cutoff
  - The alpha value becomes the final backed up value

Demonstration

Alpha Calculation



MAX $\alpha = -\infty / -\infty / 3$

1st 2nd 3rd

-1 3 (2) — no update

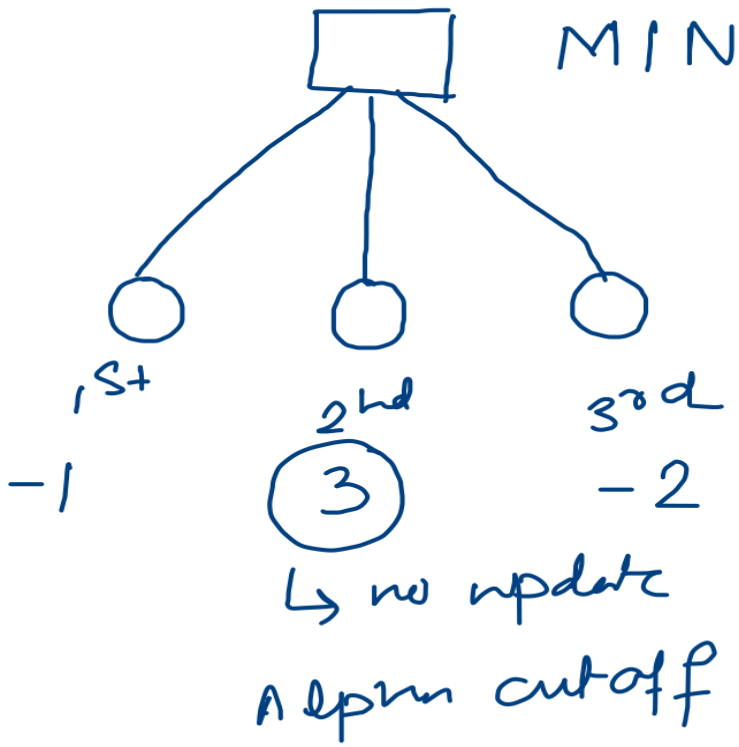
Beta cutoff ??

Calculations of beta value at MIN node

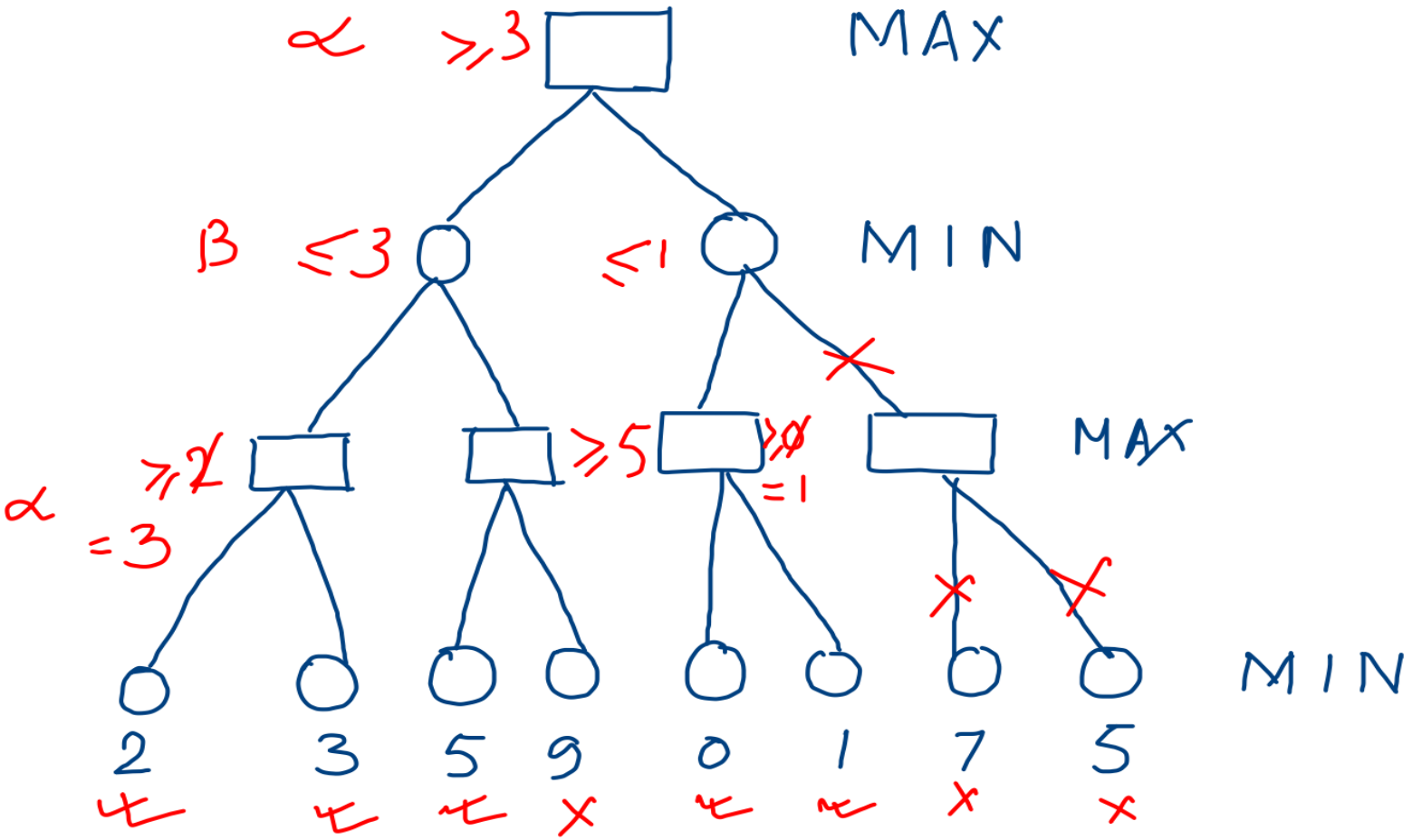
- After we calculate the final backed up value of the first child, we set beta for this node with this value
- When we have the final backed up value of the second child, we update beta for this node with this value if the new value is smaller
- If we have the final child with final backed up value or alpha cutoff
  - The beta value becomes the final backed up value

Demonstration

Beta value



Alpha beta search with example

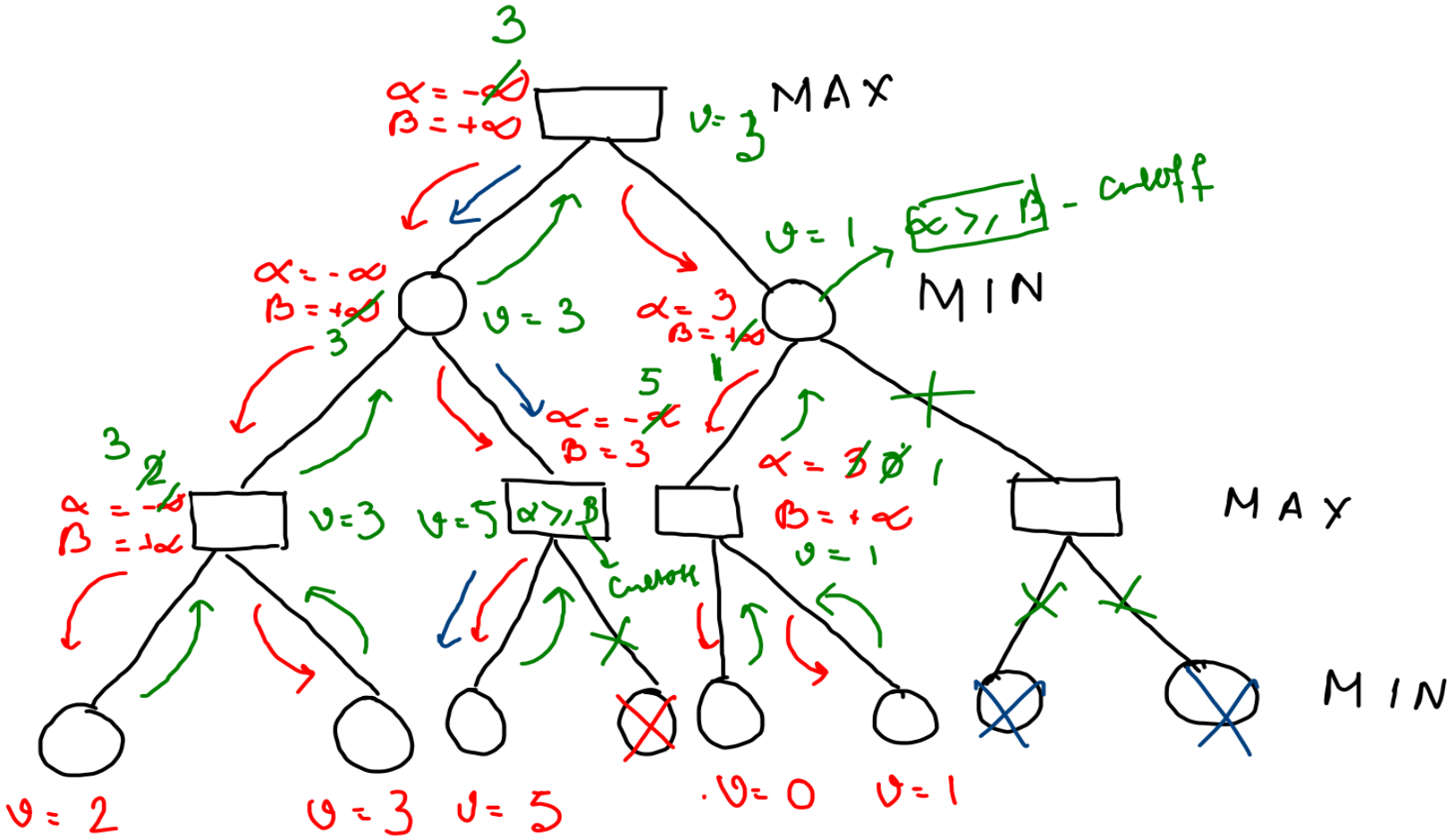


Important points of Alpha-beta pruning

- The Max player will only update the value of alpha
- The Min player will only update the value of beta
- While backtracking the tree, the final backed-up values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes

Complete Demonstration of Alpha beta pruning

(Re-check the correctness of best moves for MAX (blue color). Is it correct?)



Properties of Alpha-beta pruning (Move Ordering)

- Pruning does not affect final result. This means that it gets the exact same result as does full MINIMAX algorithm
- The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning
- It can be of two types:
  - Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$ (maximum depth of the tree)
  - Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. Complexity in ideal ordering is $O(b^{m/2})$
- Some improvement: Node ordering instead of generating the tree from left to right, we reorder the tree based on some static evaluation function