Introduction to Artificial Intelligence

LIAW, RUNG-TZUO

Department of Computer Science and Information Engineering

Fu Jen Catholic University, Taiwan

Chapter 5
Adversarial Search and Games

Outline

- **Games**
- **Optimal decisions in games**
- **Alpha-beta pruning**
- **Monte-Carlo Tree Search**

Outline

- **Games**
- **Optimal decisions in games**
- **Alpha-beta pruning**
- **Monte-Carlo Tree Search**

What Are and Why Study Games?

- **Games are a form of multi-agent environment**
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments
  - Competitive multi-agent environments give rise to adversarial search a.k.a. games
- **Why study games?**
  - **Fun!!!**
  - Interesting subject of study because they are hard
    - Chess: $b \approx 35$, $d \approx 40 \rightarrow$ search tree has $35^{100} \approx 10^{154}$ nodes
    - Games penalize **inefficiency** severely
  - Easy to represent and agents are restricted to small number of actions

Games vs. Search

- **Search – no adversary**
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP techniques can find optimal solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities
- **Games – adversary**
  - Solution is strategy (strategy specifies move for every possible opponent reply)
  - Time limits force an approximate solution
  - Evaluation function: evaluate “goodness” of game position
  - Examples: chess, checkers, go, …

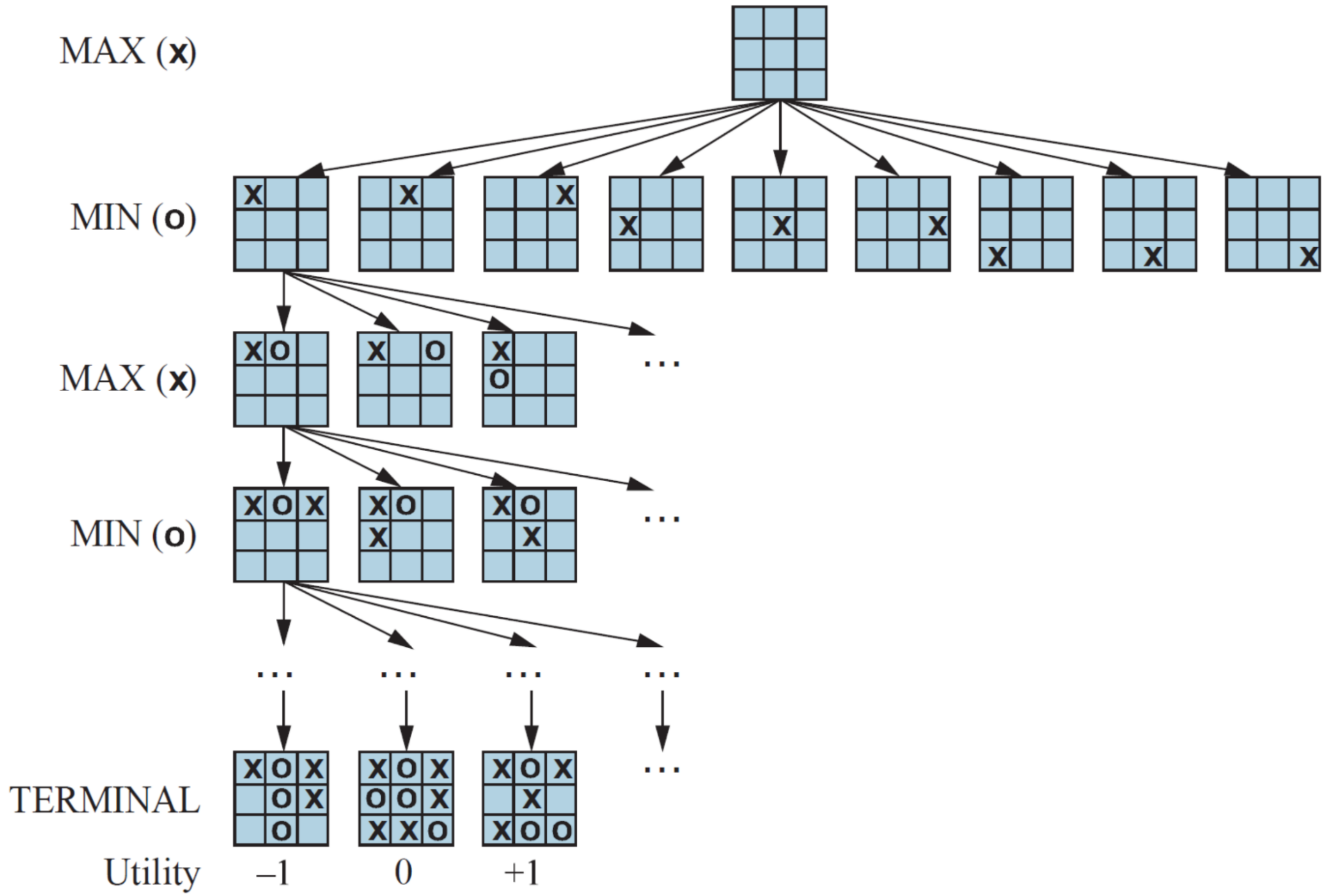
Outline

- **Games**
- **Optimal decisions in games**
- **Alpha-beta pruning**
- **Monte-Carlo Tree Search**

Game setup

- **Two players: MAX and MIN**
  - MAX moves first and they take turns until the game is over. Winner gets award, looser gets penalty.
- **Games as search:**
  - Initial state: e.g. board configuration of chess
  - Successor function: List of (move, state) pairs specifying legal moves
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states e.g. win(+1), loose(-1), draw(0) in tic-tac-toe
- **MAX uses search tree to determine next move**

Tie-Tac-Toe



A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Strategy

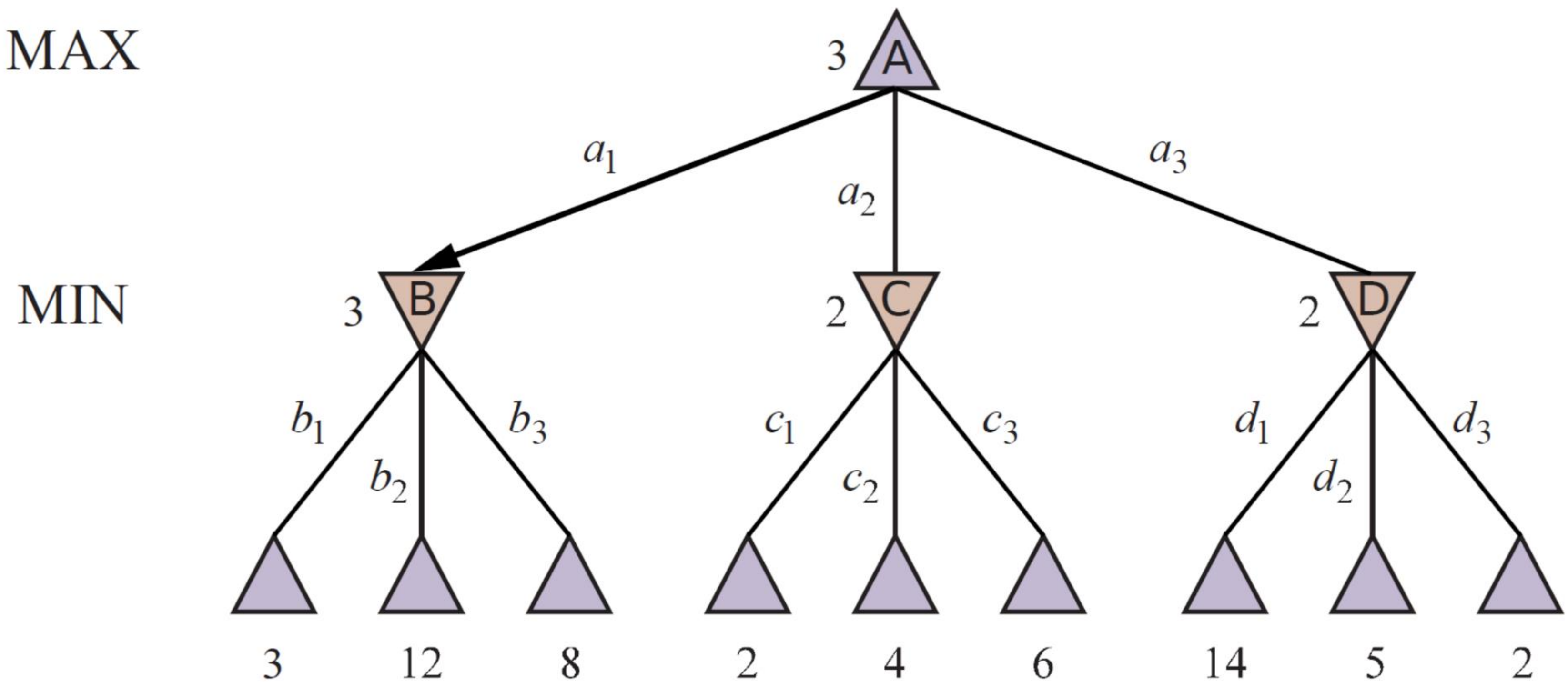
- **MAX must find a contingent strategy**
  - Move in the initial state
  - Move in the states resulting from every possible response by MIN
  - Move in the states resulting from every possible response by MIN to those moves
  - –...

Optimal Strategy

- **Optimal strategy**
  - To find the contingent strategy for MAX assuming an infallible MIN opponent
    - Assume: Both players play **optimally**!!
    - Given a choice
      - MAX will prefer to move to a state with maximum utility value
      - MIN will prefer a state with minimum value
  - Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

$$MinimaxValue(n) = \begin{cases} Utility(n) & n \text{ is a terminal state} \\ \max_{s \in successor(n)} MinimaxValue(s) & n \text{ is a max node} \\ \min_{s \in successor(n)} MinimaxValue(s) & n \text{ is a min node} \end{cases}$$

Two-Ply Game Tree



A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value.

Minimax Algorithm

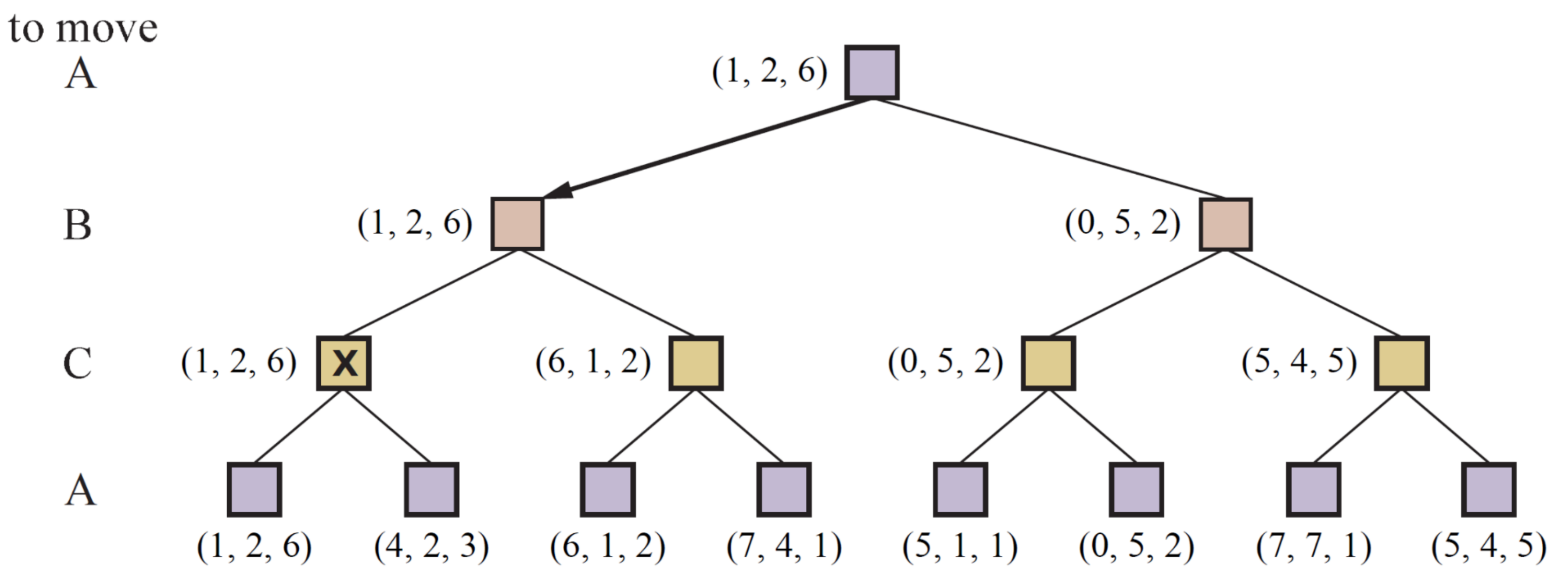
```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Minimax Algorithm

- **Minimax algorithm**
  - Uses simple recursive computation
  - Performs a complete depth-first exploration of game tree
- **Complexity**
  - Time: $O(b^m)$
  - Space: $O(bm)$

Q: What if more than 2 players?

- **Multiplayer games**
  - Allow more than two players
  - Extend single minimax values to vectors (why was there only one value for 2 players?)



The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

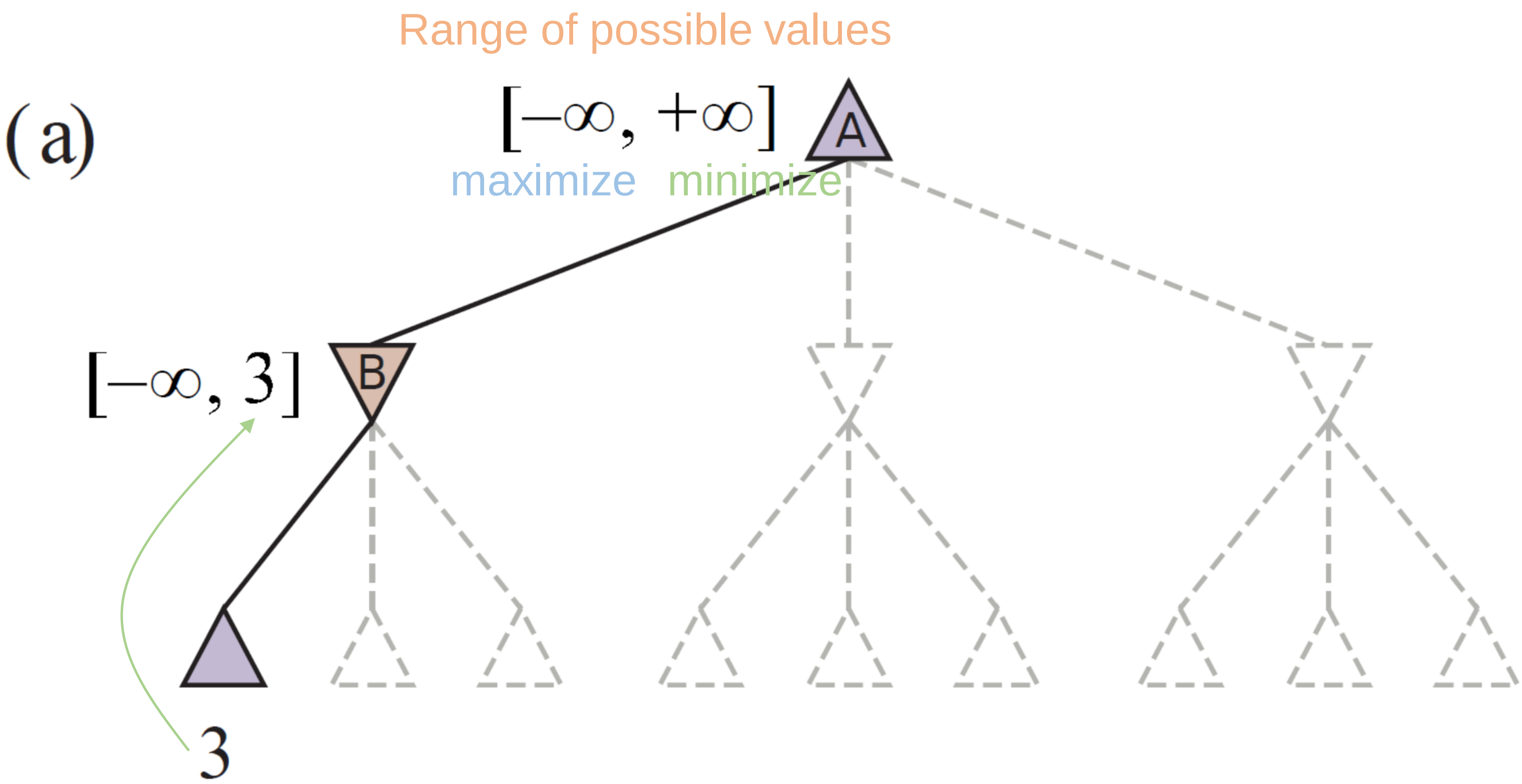
Discussion

- **Any problem with minimax search?**
  - Number of game states is exponential to the number of moves
  - Solution: Do not examine every node
    → Alpha-beta pruning
    - Remove branches that do not influence final decision

Outline

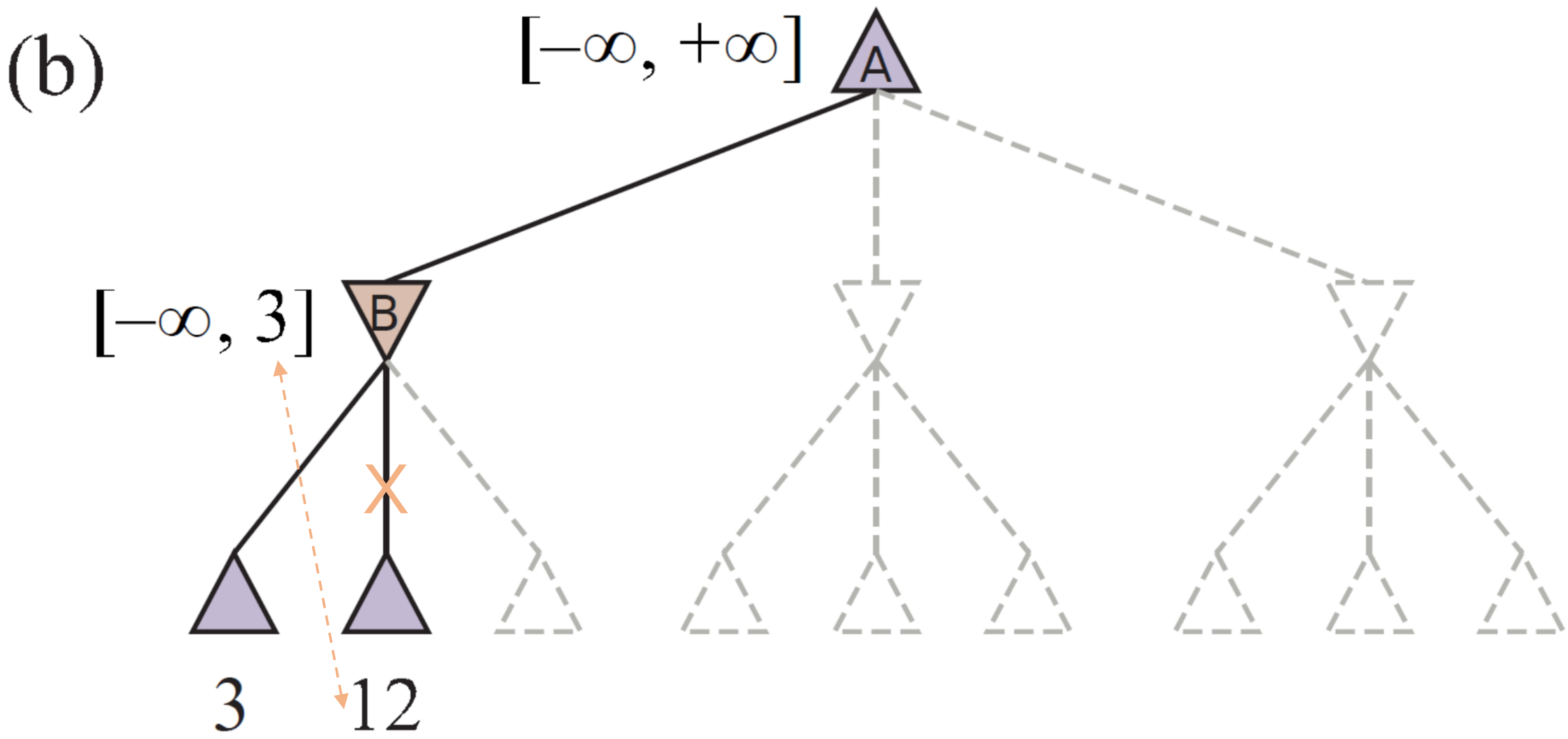
- Games
- Optimal decisions in games
- **Alpha-beta pruning**
- Monte-Carlo Tree Search

Alpha-Beta Example (1)



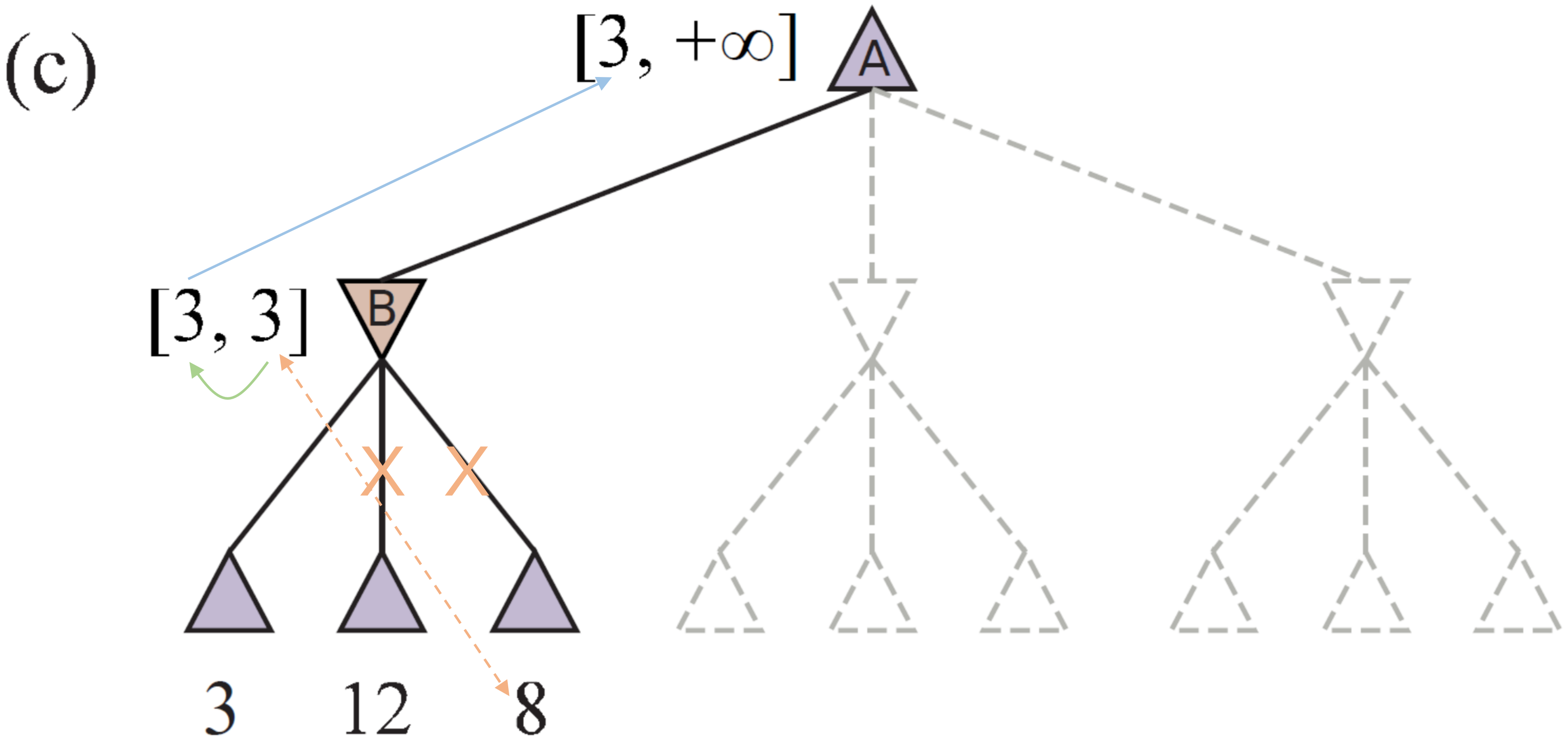
Do DFS until first leaf

Alpha-Beta Example (2)



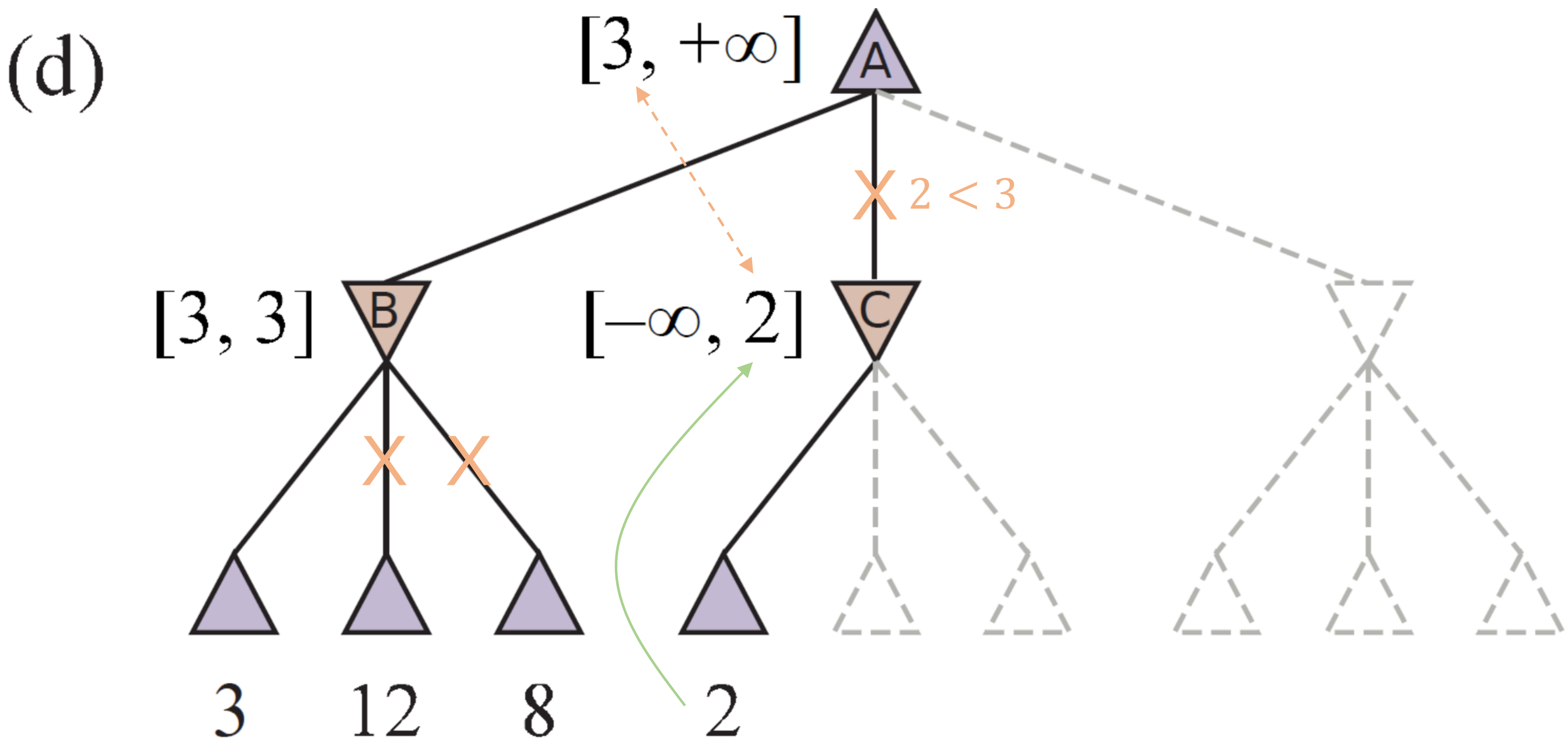
Pruning: MIN is to **minimize** the utility of MAX

Alpha-Beta Example (3)



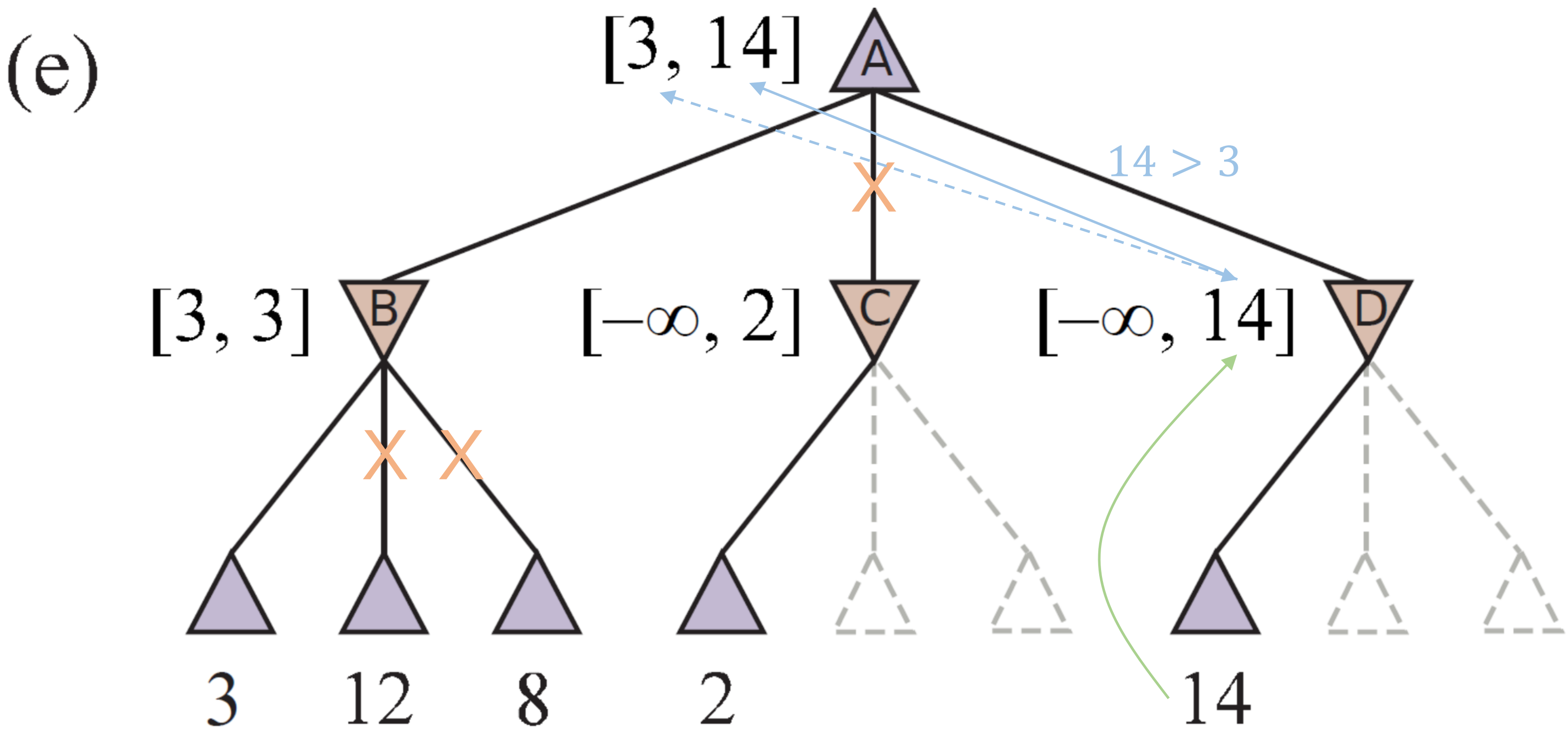
Pruning. MAX is to maximize the utility of MAX

Alpha-Beta Example (4)



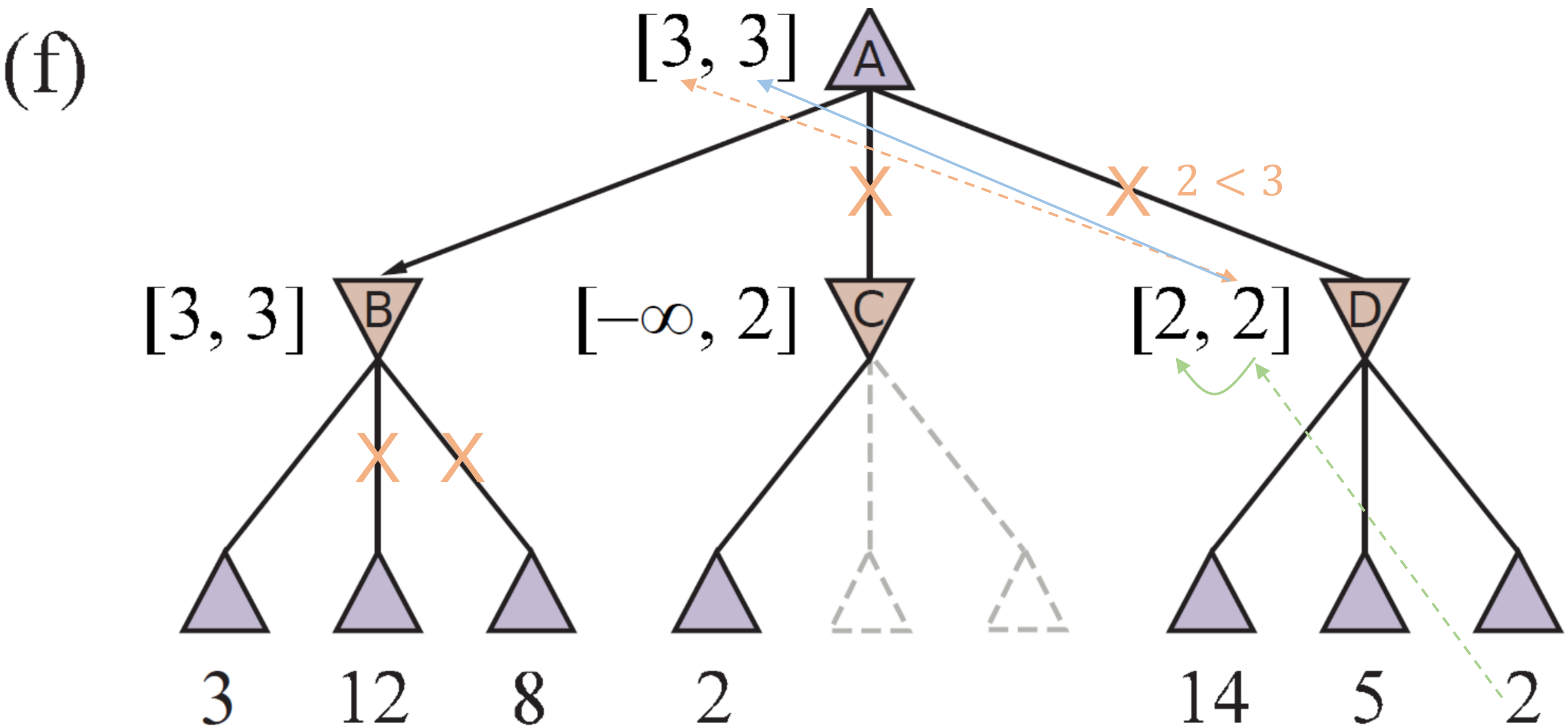
C is a MIN node with a value of at most 2 → worse than B (< 3) → pruned.

Alpha-Beta Example (5)



D is a MIN node with a value of at most 14 → better than B (> 3) → explore next.

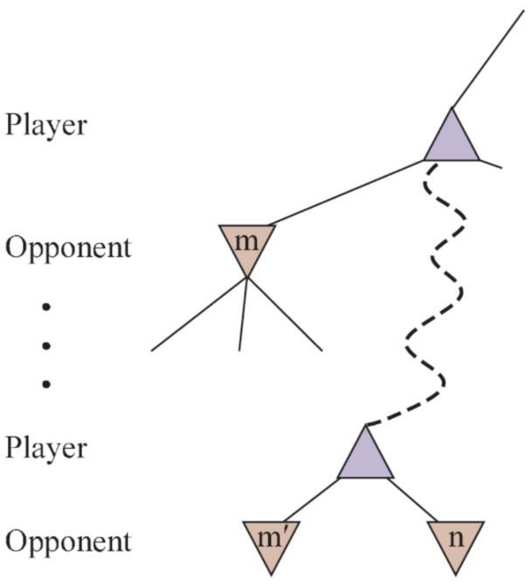
Alpha-Beta Example (6)



D is a MIN node with a value of at most 2 → worse than B (< 3) → pruned.

Alpha-Beta Pruning

- **Alpha-beta pruning**
  - Pruning: To eliminate large parts of the tree
  - Previous example:
    $Minimax - Value(root)$
    $= \max(\min(3,12,8), \min(2, x, y), \min(14,5,2))$
    $= \max(3, z, 2)$
    $= 3 \ \ (z \leq 2)$
  - If player has a better choice **m** at
    - Either parent node of n
    - Or any choice point further up
- n will never be reached in actual play
  - When enough is known about n, it can be pruned



Alpha-Beta Pruning: Evaluation

- **Behavior**
  - Pruning does not affect final results
  - Entire subtrees can be pruned
  - Good move ordering improves effectiveness of pruning
- **Complexity**
  - Time: $O(b^{m/2})$ for perfect ordering
    - Branching factor of $\sqrt{b}$, instead of *b*
    - Alpha-beta pruning can look twice as far as minimax in the same amount of time
  - Memory: Use transposition table for repeated states
    - Transpositions: different permutation of move sequence that ends up in the same position
    - Closed list in Graph-Search in a transposition table

Outline

- **Games**
- **Optimal decisions in games**
- **Alpha-beta pruning**
- **Monte-Carlo Tree Search**

Monte Carlo Tree Search (MCTS)

- A tree search algorithm based on Monte Carlo simulation
  - Heuristic search algorithm for making decisions
- Widely used for searching actions of games, including
  - Board games: Go, Chess, MTG, …
  - Platform games: Super Mario Bros, Ms. Pac-Man, …
  - , …
- A great success on Go
  - First been introduced in 2006
  - Compete low-dan amateur in 2012
  - Compete high-dan professional in 2016: AlphaGo

Principle of Operations

- Explore the most promising move in the search tree according to the results from random sampling
- Each round of MCTS consists of four basic steps:
  - Selection
  - Expansion
  - Simulation
  - Backpropagation

Selection

- Start from root node R which represents the current game state.
- Select a child node as an action to take and get to a new state.
  - Until a leaf node L is reached.
  - Based on the upper confidence bound 1 applied to trees (UCT):

$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$

    - v_i: a utility value for node i which should be maximized
    - w_i: the number of times the action from node i leads to a win
    - n_i: the number of tries from node i
    - N: the total number of tries
- Each node will be fully explored (all actions are tried) before exploring its child.

Expansion

- If L is **not** a **decisive** game state (i.e., win/lose/draw), expand one of its child node C.
  - A child is a possible action from the game state defined by L.

Simulation

- Simulate the game by performing random steps from C until a decisive state is reached.
  - A.k.a rollout or playout

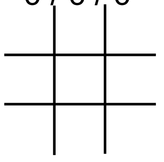
Backpropagation

- Propagate the result of playout from the deepest child node C to the shallowest root node R.

Example: MCTS for Tic-Tac-Toe

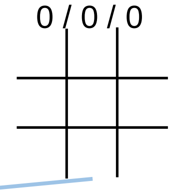
a) Initial state: **Select** root as the first leaf (no child)

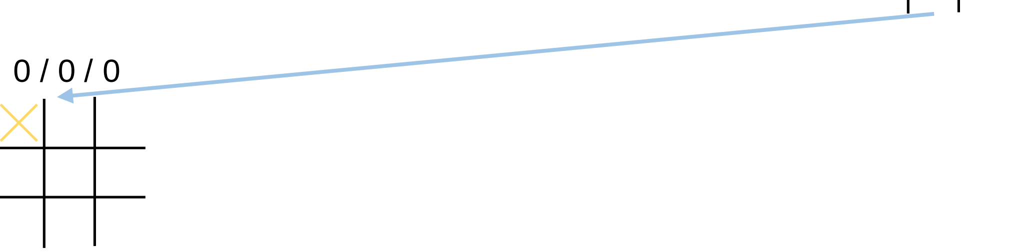
0 / 0 / 0



Example: MCTS for Tic-Tac-Toe

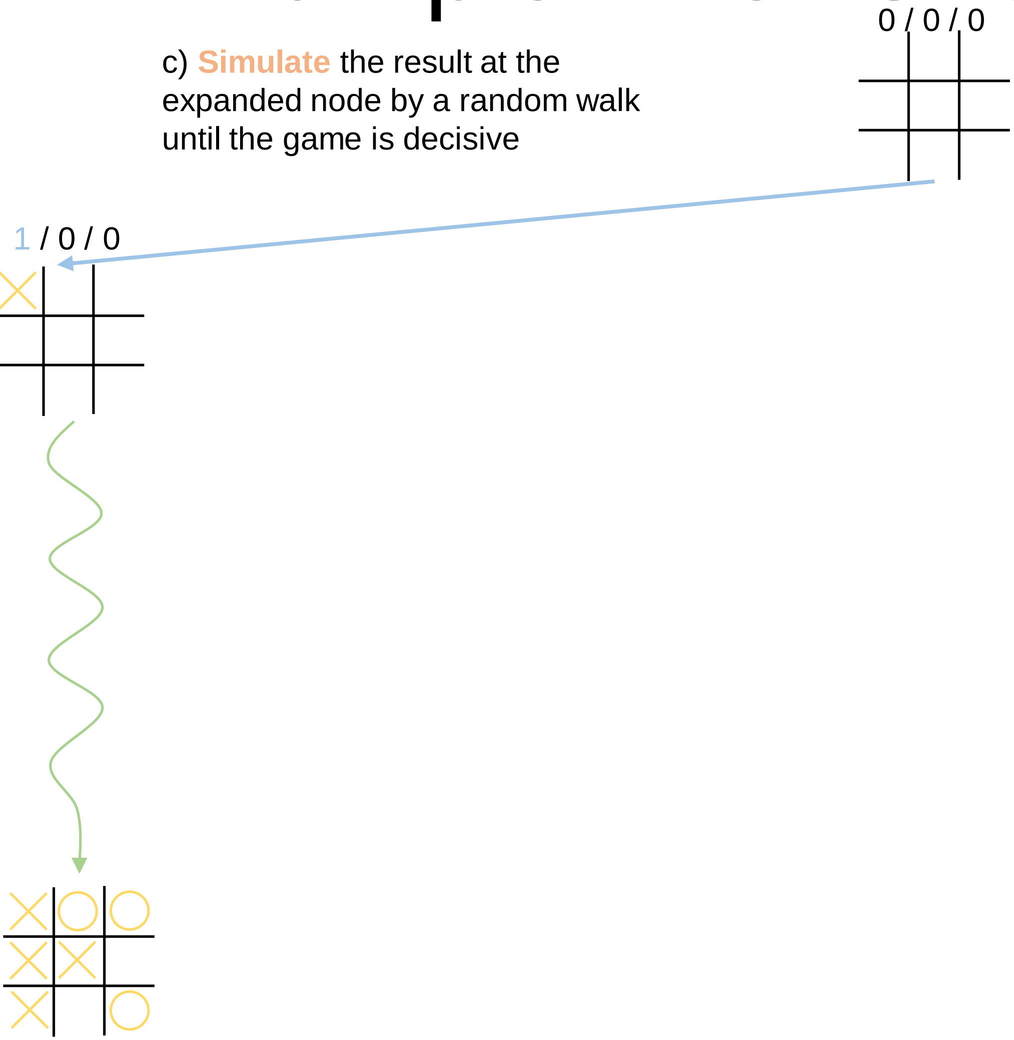
b) **Expand** a random child (upper left) of the selected node (root).





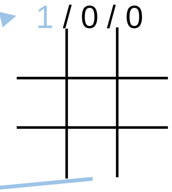
Example: MCTS for Tic-Tac-Toe

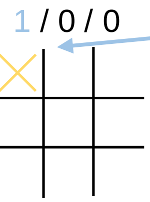
c) **Simulate** the result at the expanded node by a random walk until the game is decisive



Example: MCTS for Tic-Tac-Toe

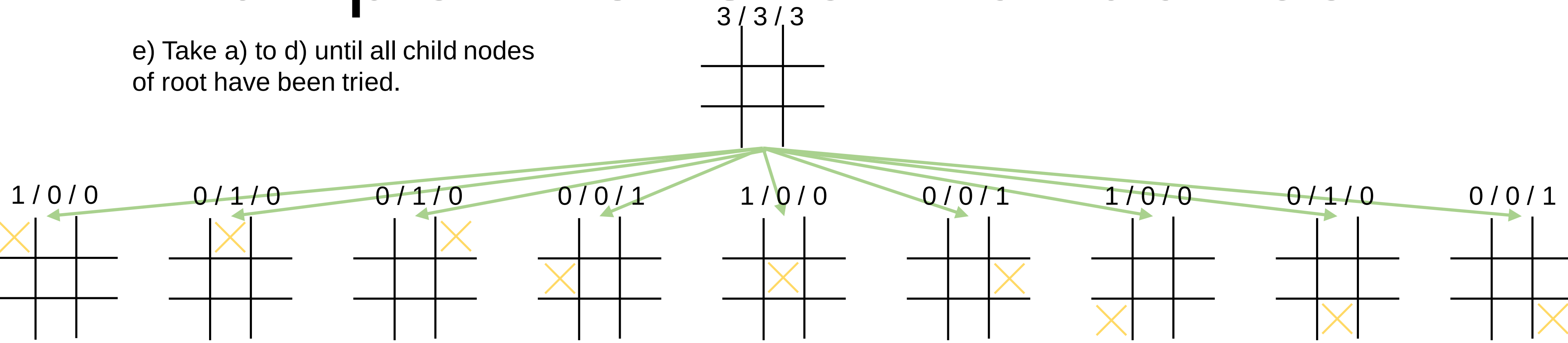
d) **Backpropagate** the result till the root.





Example: MCTS for Tic-Tac-Toe

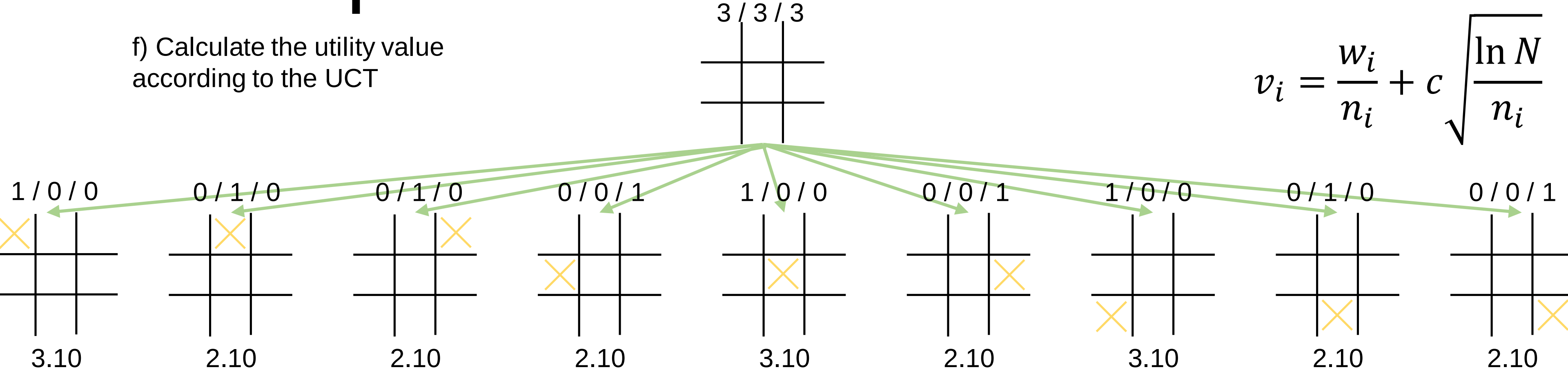
e) Take a) to d) until all child nodes of root have been tried.



Example: MCTS for Tic-Tac-Toe

f) Calculate the utility value
according to the UCT

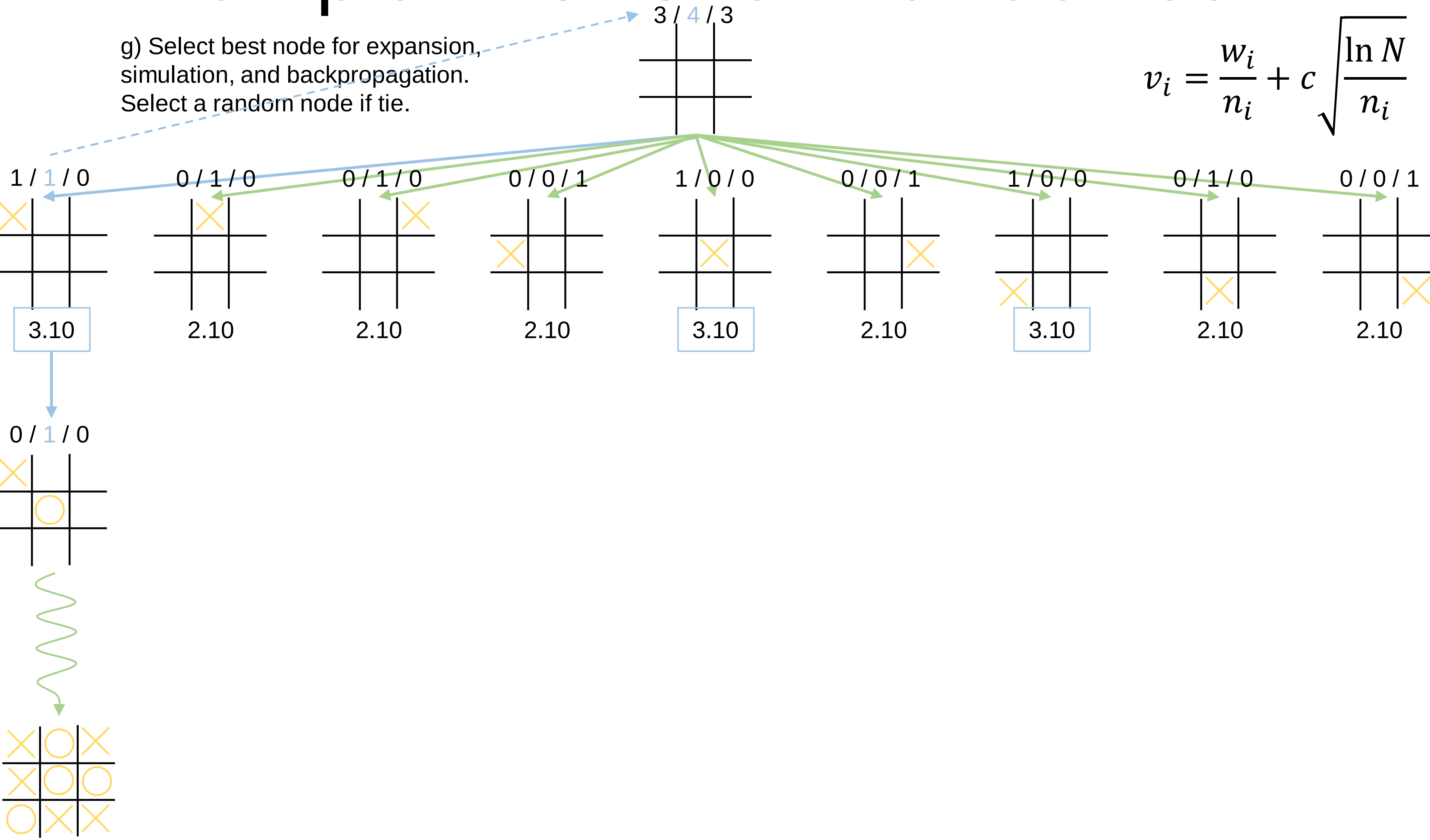
$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$



Example: MCTS for Tic-Tac-Toe

g) Select best node for expansion, simulation, and backpropagation. Select a random node if tie.

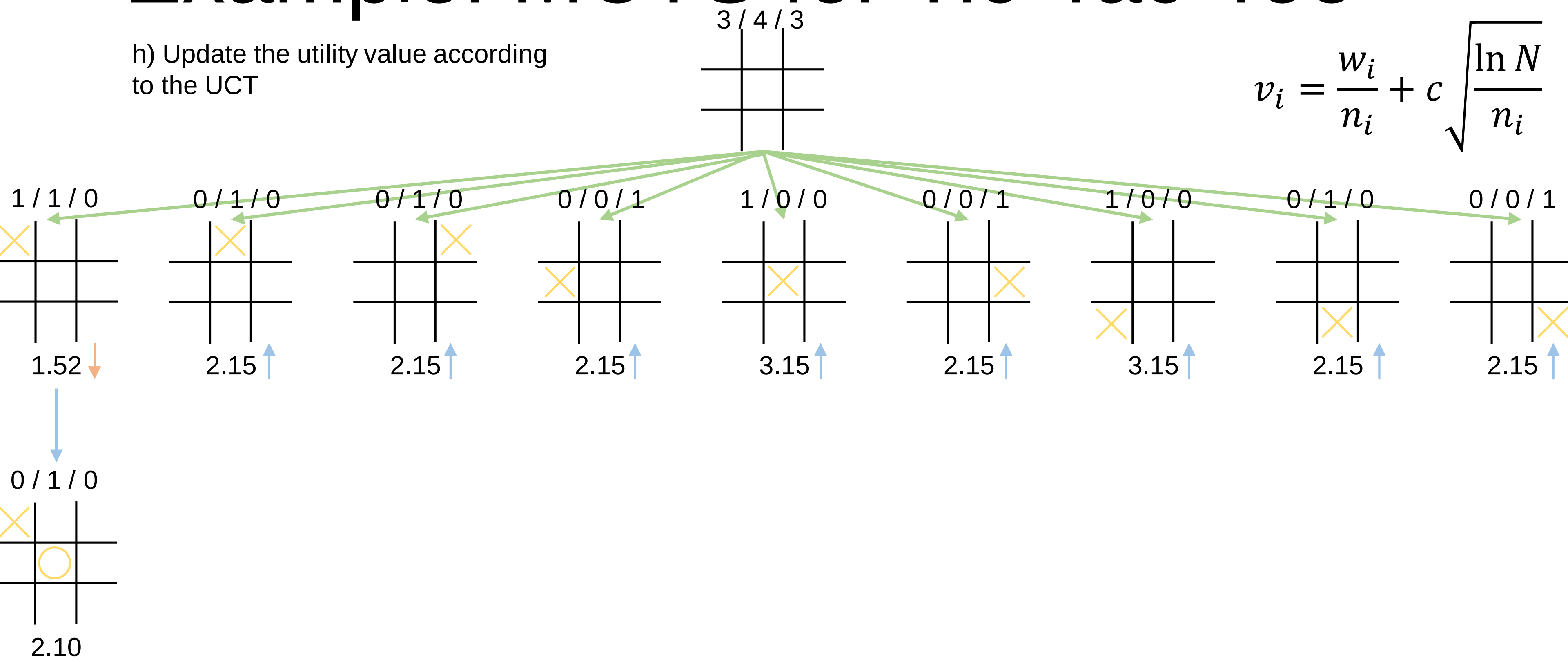
$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$



Example: MCTS for Tic-Tac-Toe

h) Update the utility value according to the UCT

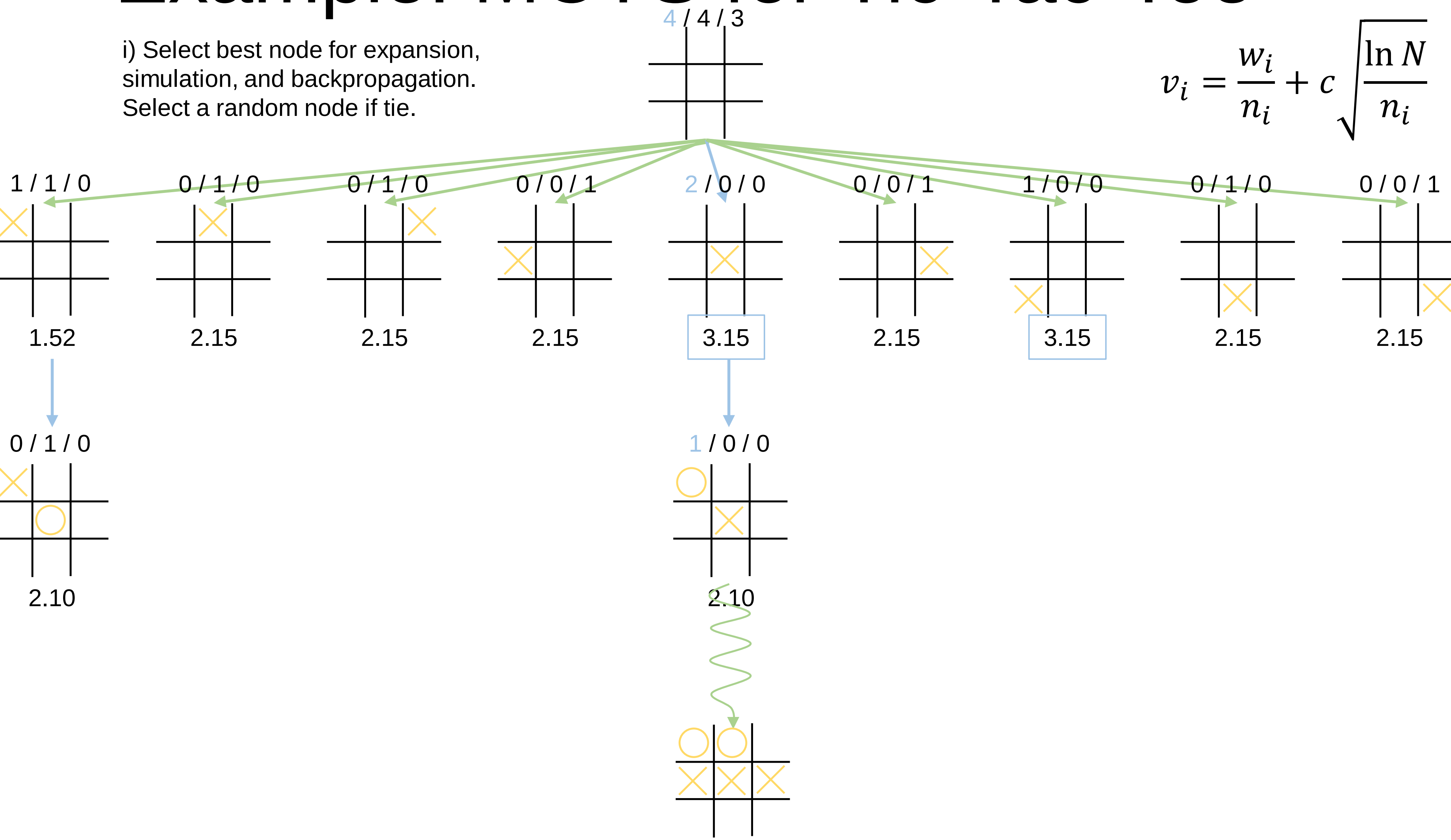
$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$



Example: MCTS for Tic-Tac-Toe

i) Select best node for expansion, simulation, and backpropagation. Select a random node if tie.

$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$



Example: MCTS for Tic-Tac-Toe

j) Update the utility value according to the UCT…

$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$