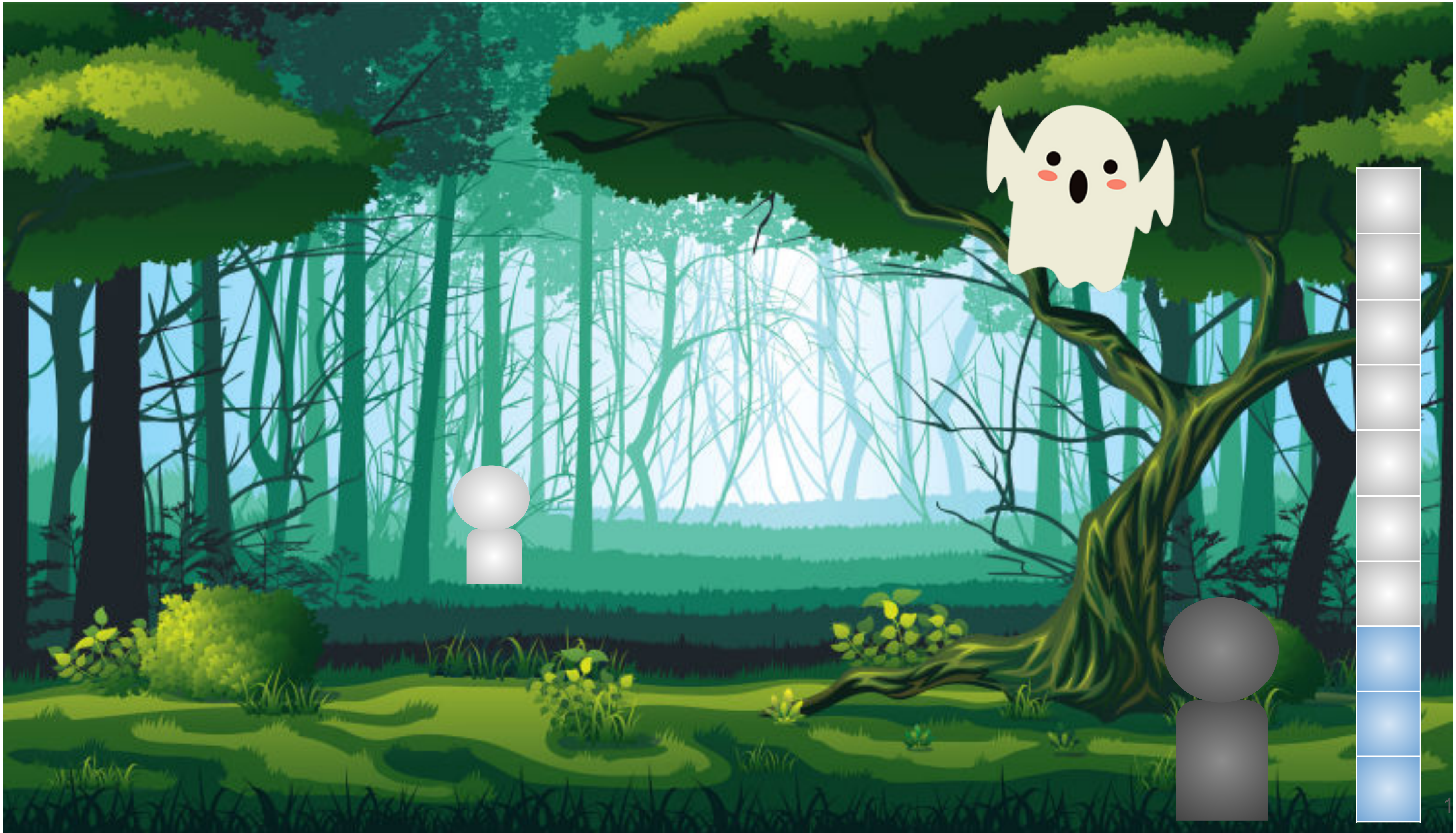


Source: https://www.52112.com/pic/454515.html,https://www.google.com/url?sa=i&url=https%3A%2F%2F90sheji.com%2Fpng%2F8275972.html&psig=AOvVaw0jk1PVJSxr60qZ6uL8jhU1&ust=1673595572038000&source=images&cd=vfe&ved=0CA0QjRxqFwoTCJCoo43EwfwCFQAAAAAdAAAAABAD

AI

Competitive Environments

Games vs. Search Problems

- Unpredictable opponent
  - Solution is a **strategy** specifying a move for every possible opponent reply
- Time limits

Search Topics

- Search problems (Ch. 3)
- Uninformed search (Ch. 3)
- Informed search (Ch. 3)
- Local search (Ch. 4)
- **Adversarial search (Ch. 6)**
- Constraint Satisfaction Problems (CSPs) (Ch. 5)

Adversarial Search

L.-Y. Wei

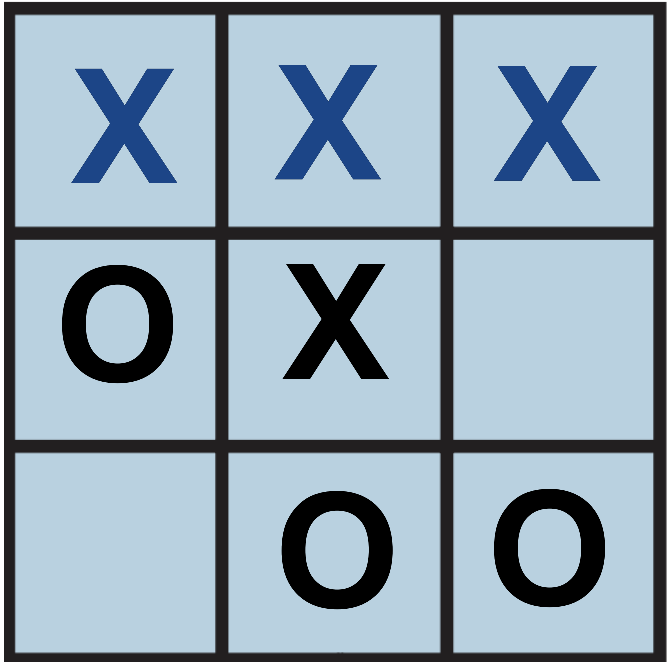
Spring 2024

Game Types

- The game most commonly studied are
  - Deterministic
  - Two-player
  - Turn-taking
  - Perfect information
    - Fully observable

  (Imperfect information, e.g., poker and bridge)
  - Zero-sum games 零和賽局
    - No “win-win” outcome
      (i.e., what is good for one player is just as bad for the other)

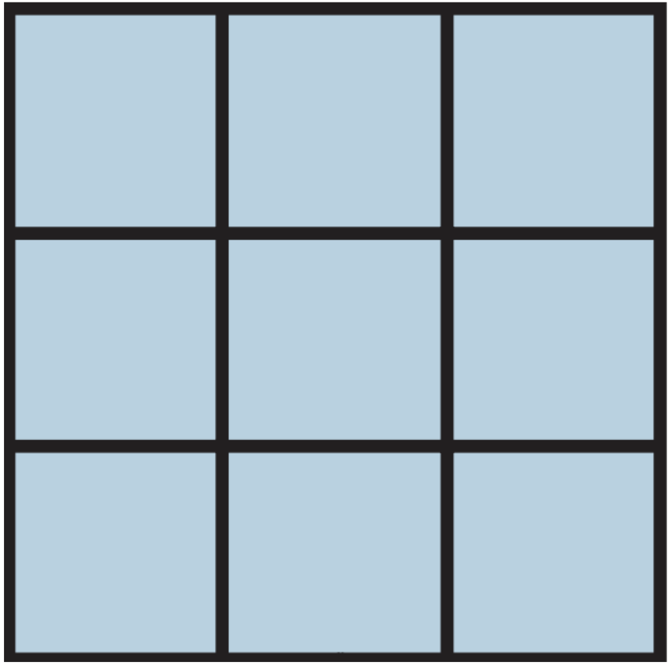
Example: Tic-Tac-Toe



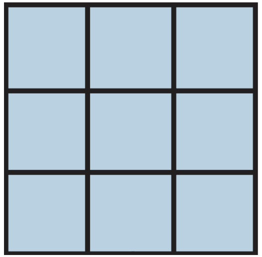
Two-Player Zero-Sum Games

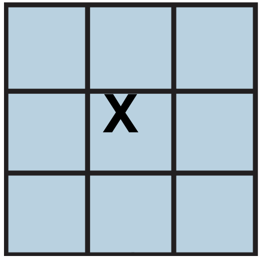
- Initial state, S_0
- TO-MOVE(*s*)
  - Returns which **player** to move in state *s*
- ACTIONS(*s*)
  - Returns the set of legal moves in state *s*
- RESULT(*s*,*a*), transition model
  - Defines the state resulting from taking action *a* in state *s*
- IS-TERMINAL(*s*), terminal test
  - Checks if state *s* is a terminal state
- **UTILITY**(*s*,*p*), utility function/objective function/
  - Defines the final numeric value to player *p* when the game ends in terminal state *s*

Example: Initial State



Example: TO-MOVE(s)

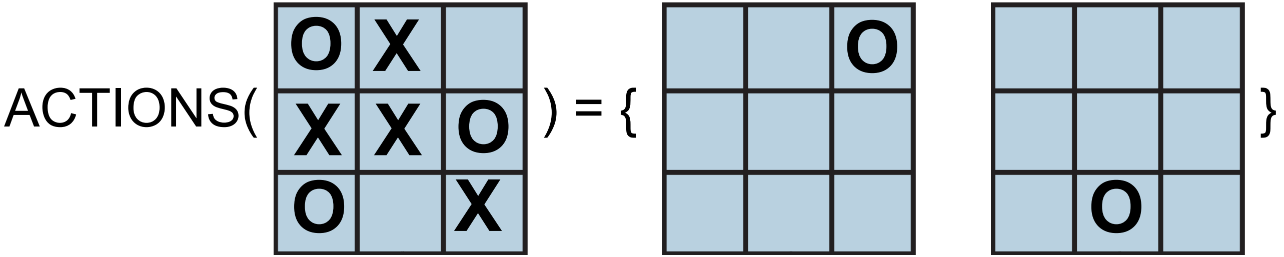
TO-MOVE() = X

TO-MOVE(

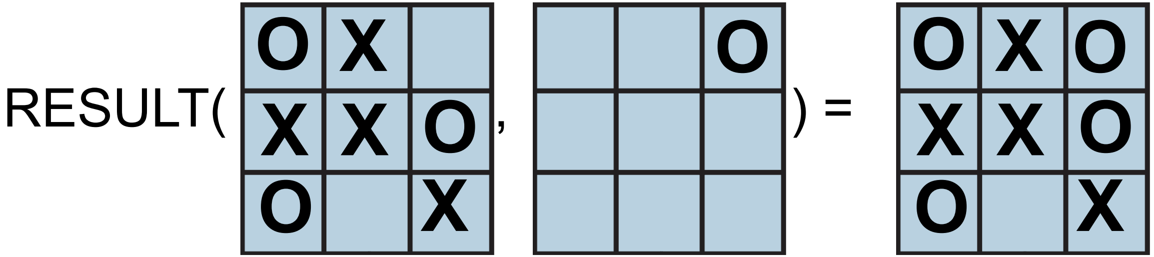
) = O

換誰？

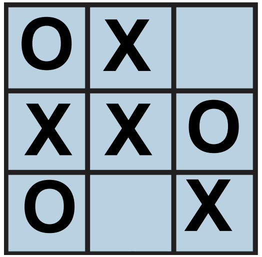
Example: ACTIONS(*s*)



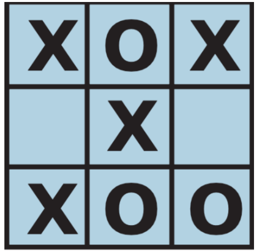
Example: RESULT(*s*,*a*)



Example: IS-TERMINAL(*s*)

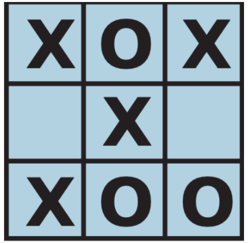
IS-TERMINAL(

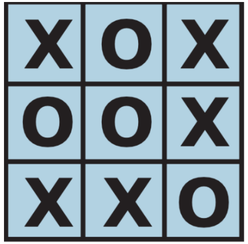
) = false

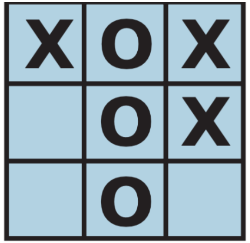
IS-TERMINAL(

) = true

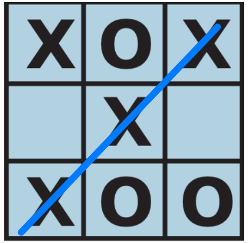
Example: UTILITY(*s*,*p*)

UTILITY( , X) =

UTILITY( , X) =

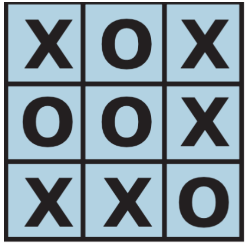
UTILITY( , X) =

Example: UTILITY(s,p)

UTILITY(

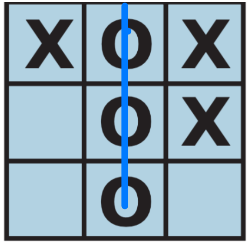
, X) = 1

X wins

UTILITY(

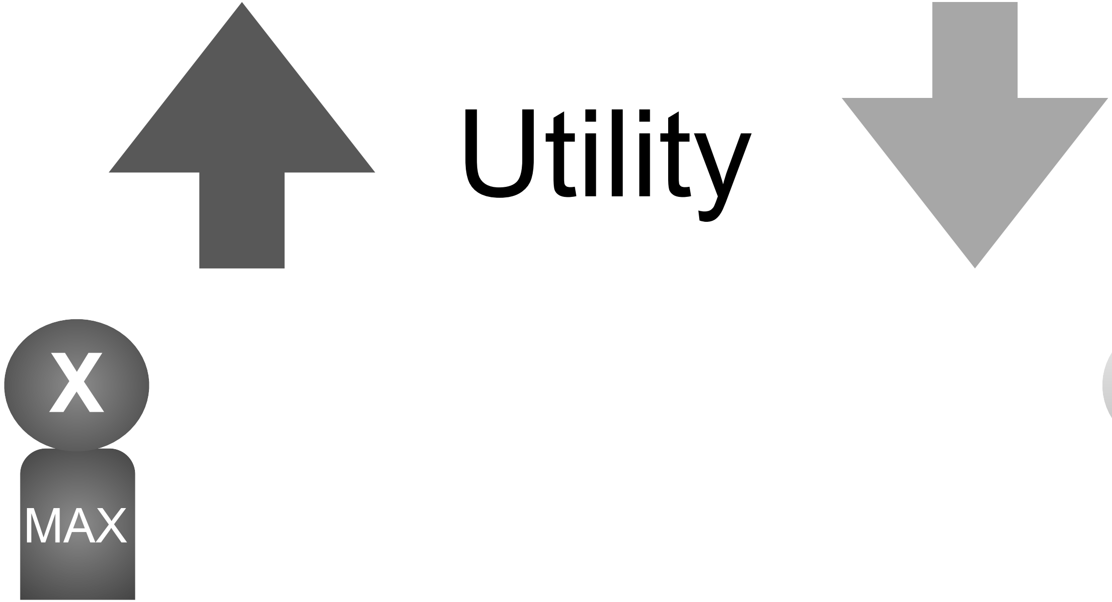
, X) = 0

draw

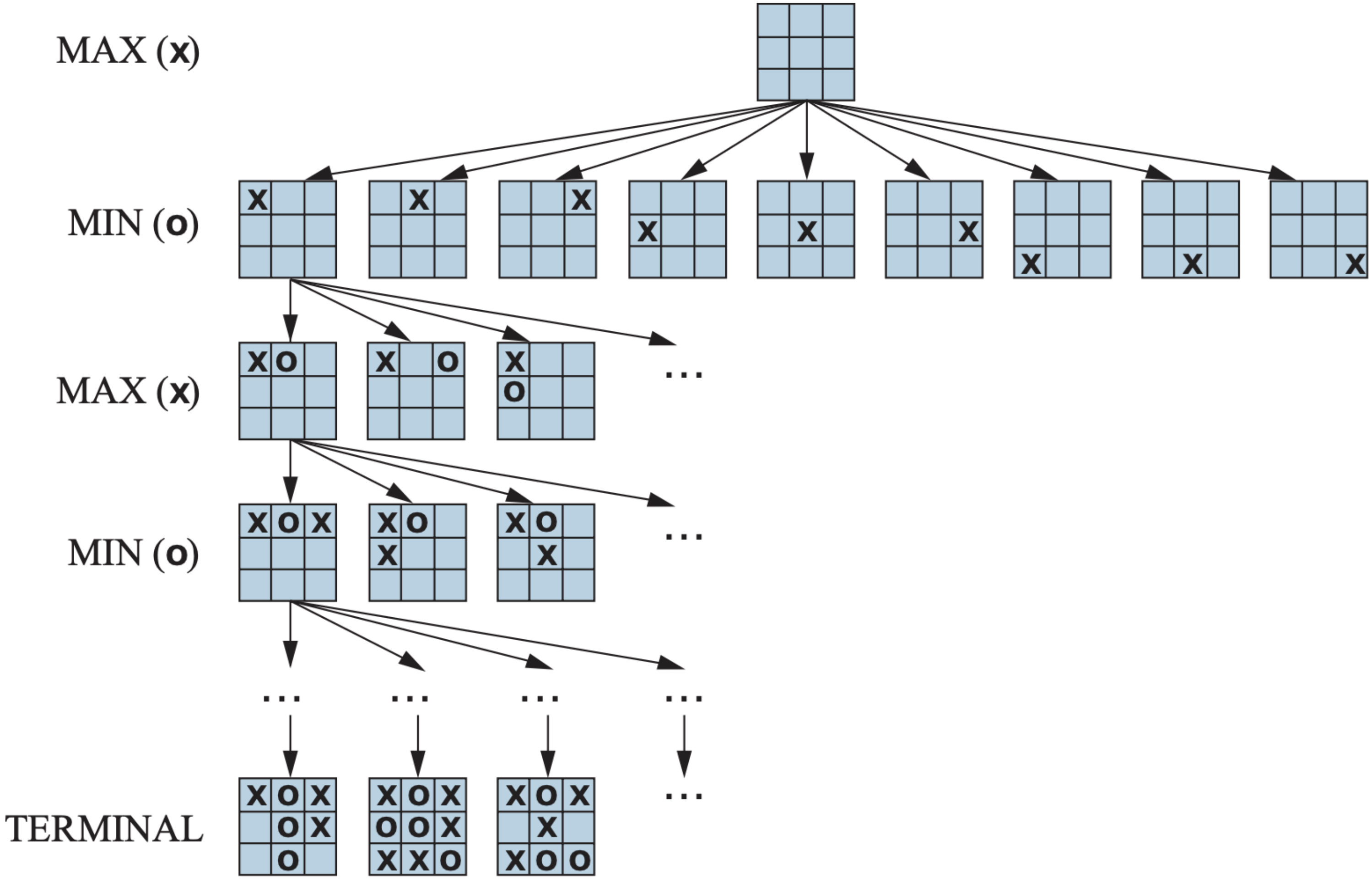
UTILITY(

, X) = -1

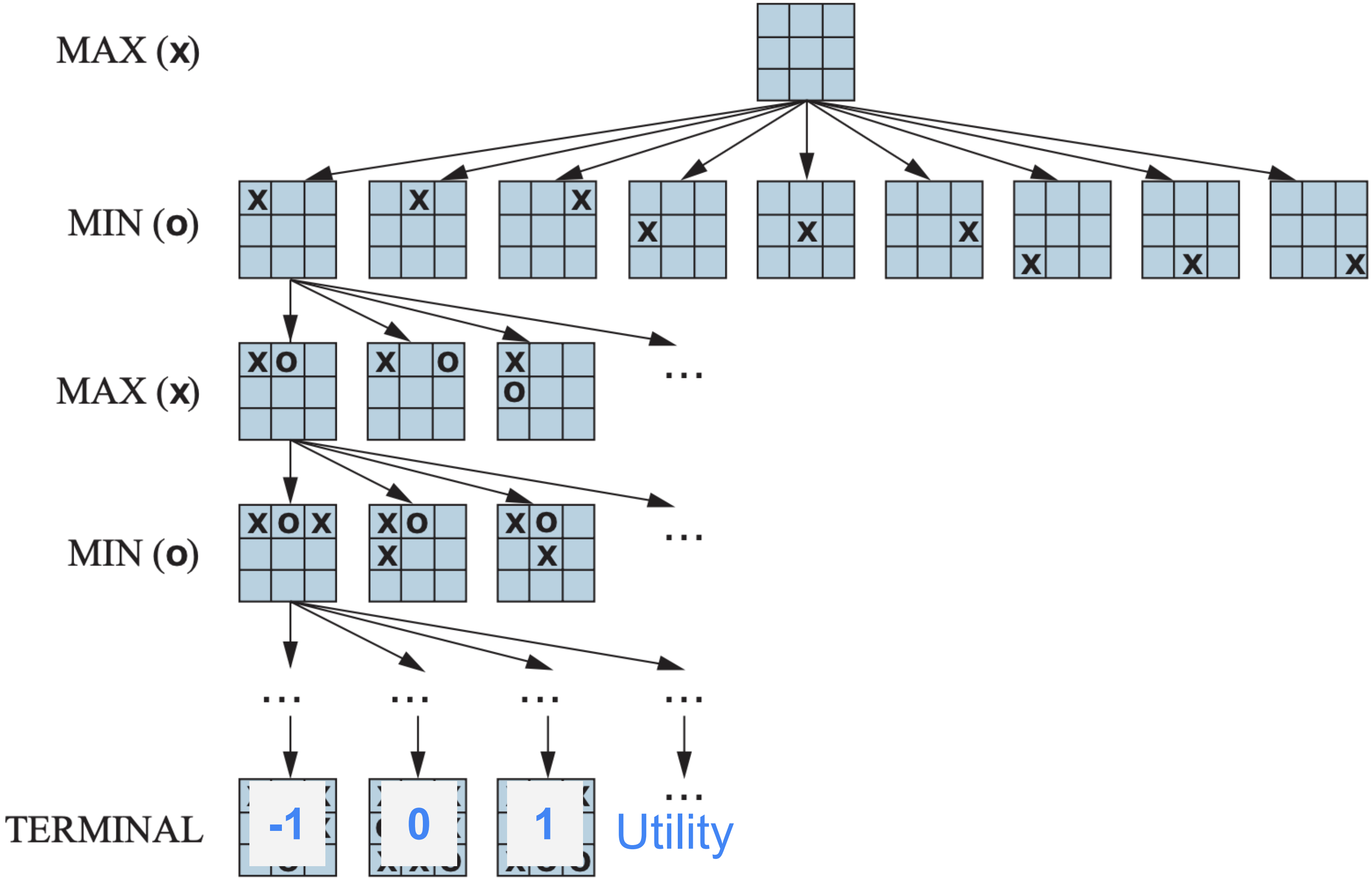
O wins


Utility
X
MAX
O
MIN

Example: Game Tree for Tic-Tac-Toe



Example: Game Tree for Tic-Tac-Toe



MAX *wants to find a sequence of actions leading to win.*

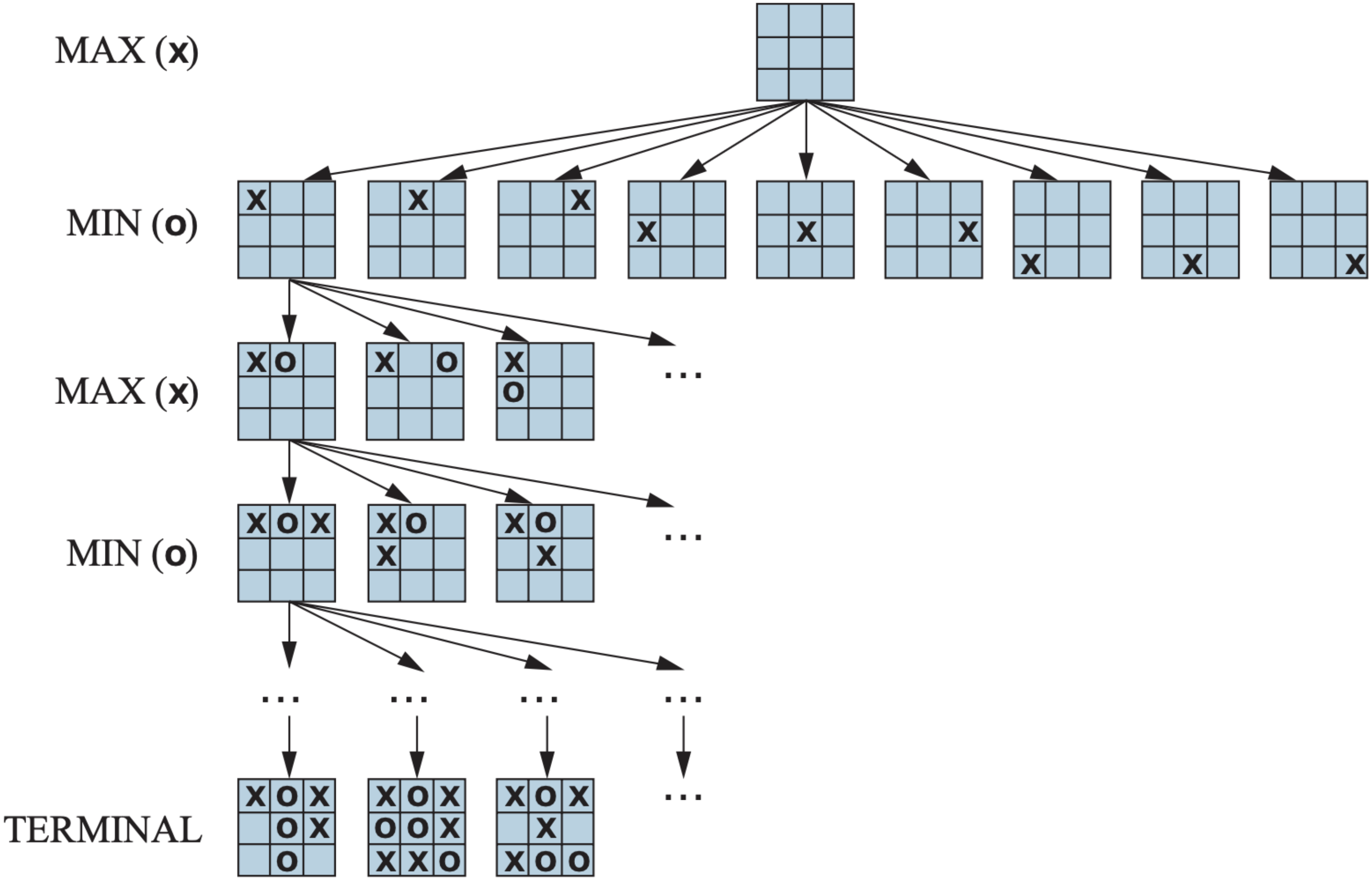
Optimal Decisions in Games?

Assume both players play optimal.

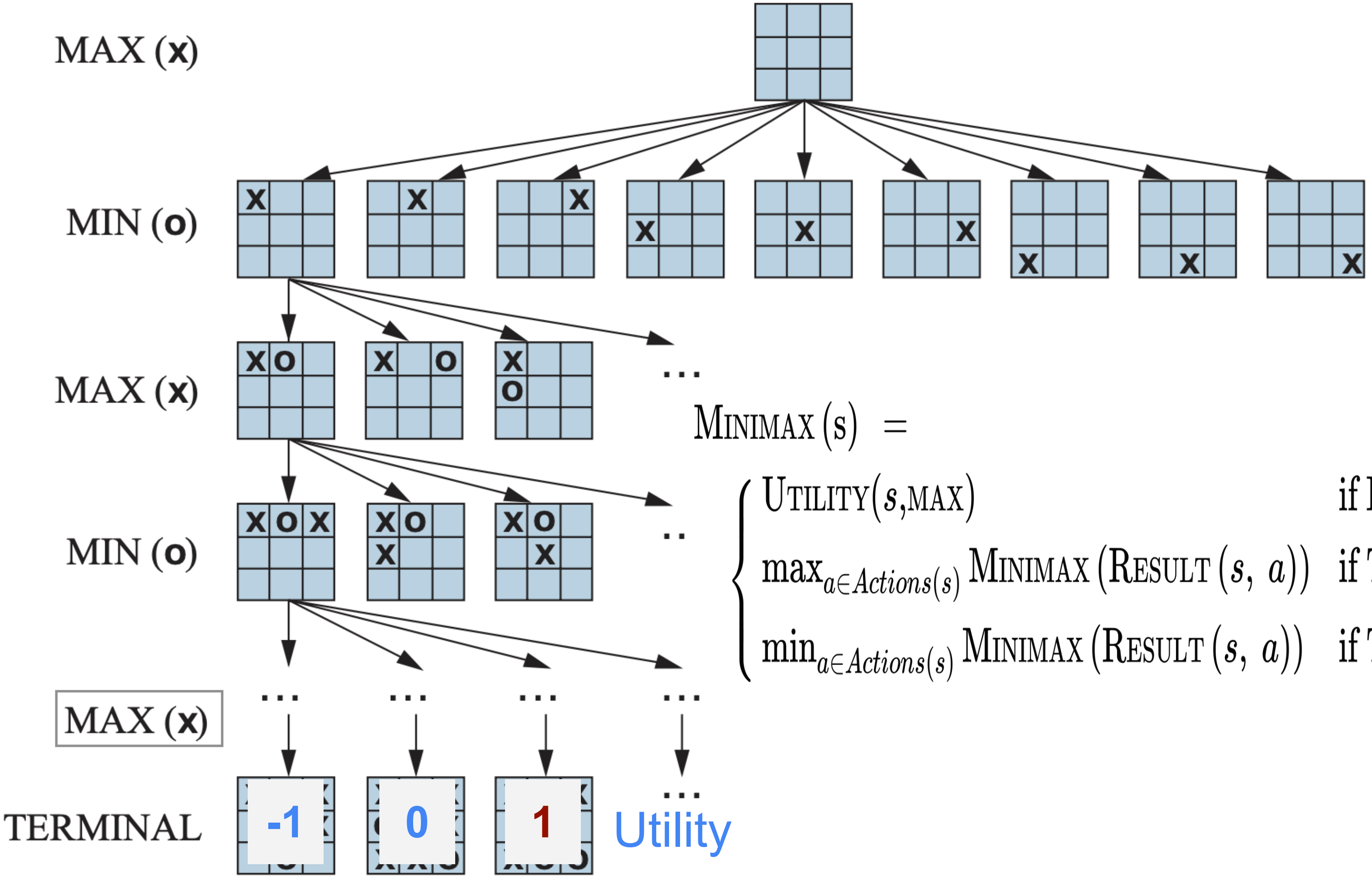
- When it is MAX's turn to move,
  MAX prefers to move to a state of maximum value

- MIN prefers a state of minimum value
  (i.e., minimum value for MAX and thus maximum value for MIN)

Example: Game Tree for Tic-Tac-Toe



Example: Game Tree for Tic-Tac-Toe



$$
\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}
$$

A General Algorithm for Perfect-Information Games

- **Minimax Search**
  - Given a game tree, the **minimax value** of each state in the tree is the utility for MAX of being in that state, i.e.,

$$\textsc{Minimax}(\mathrm{s}) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

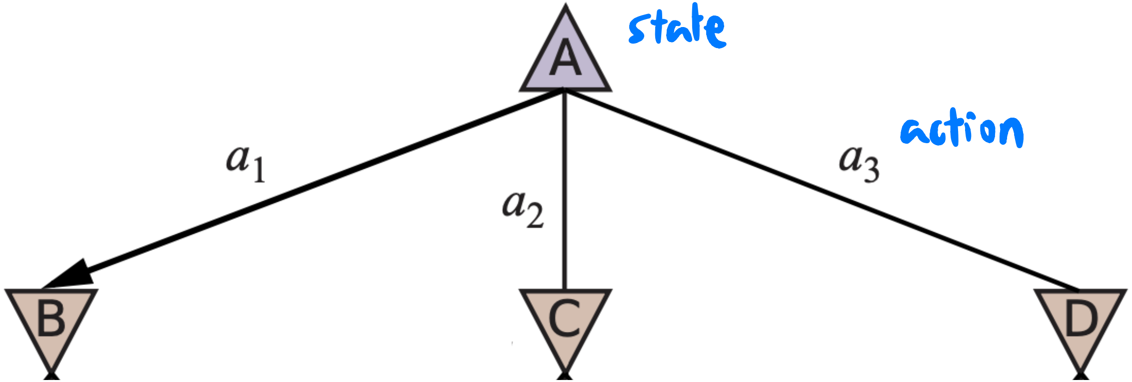
    - Assume both players play optimally from the state to the end of the game
  - Use the **depth-first** exploration of the game tree to derive the optimal strategy by working out the minimax value of each state in the tree

Example: Minimax Search for Two-Ply Game Trees

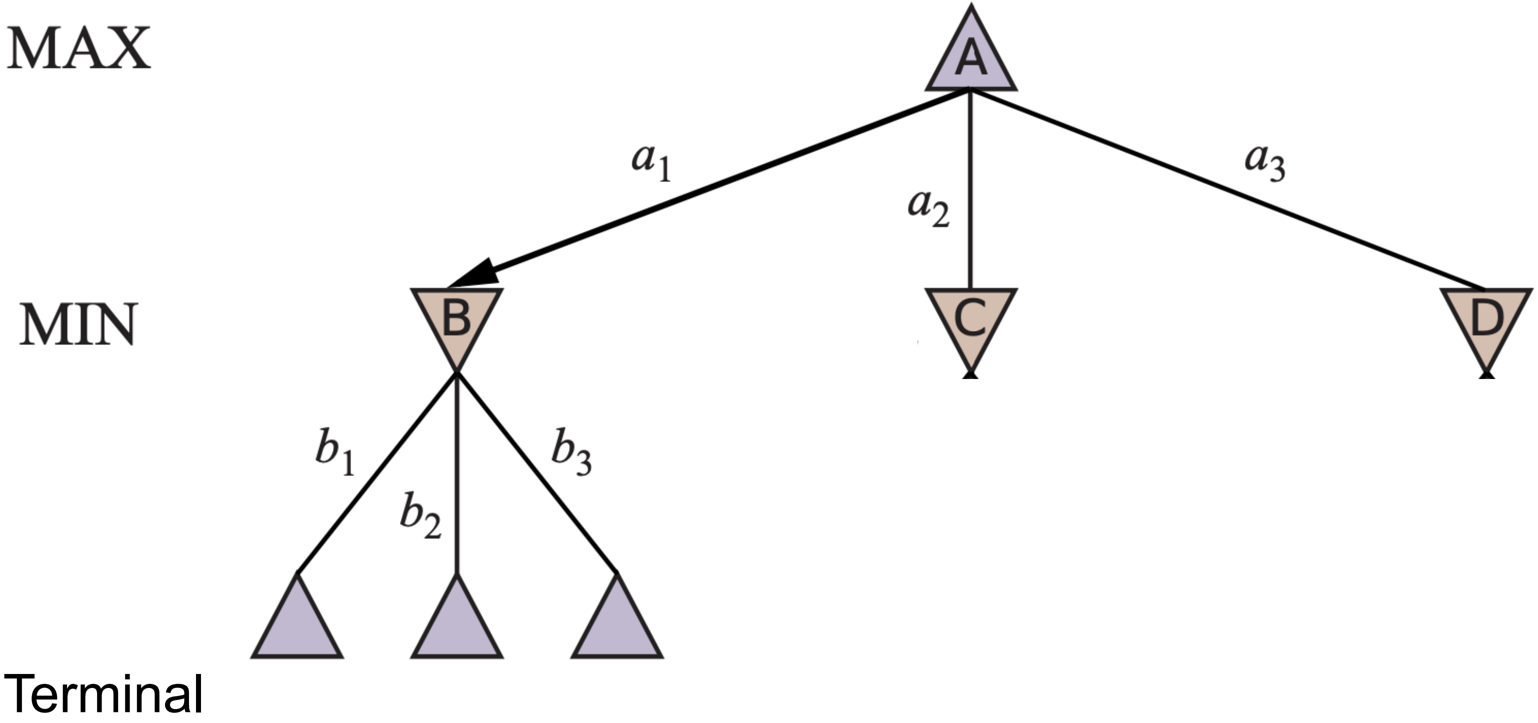
MAX

Example: Minimax Search for Two-Ply Game Trees

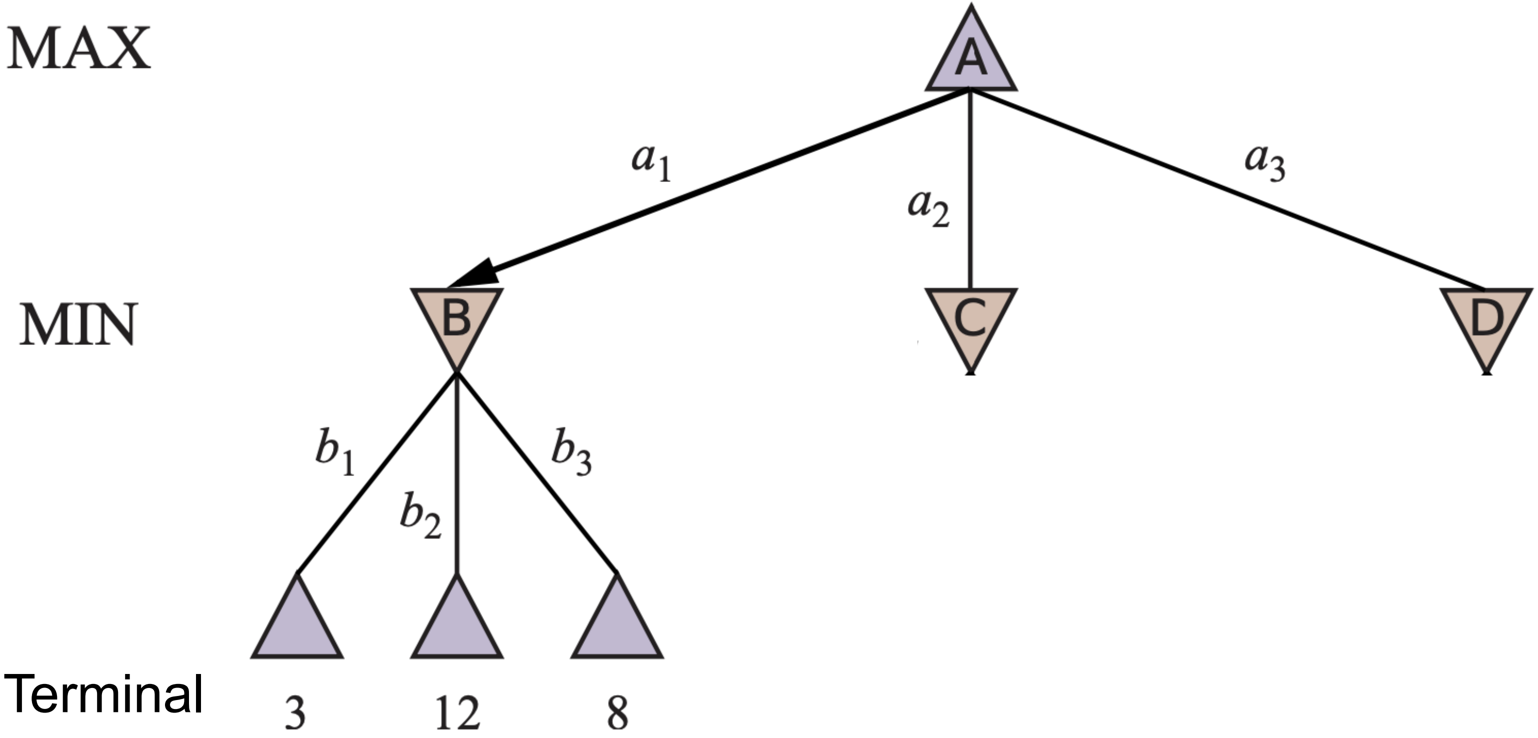
MAX



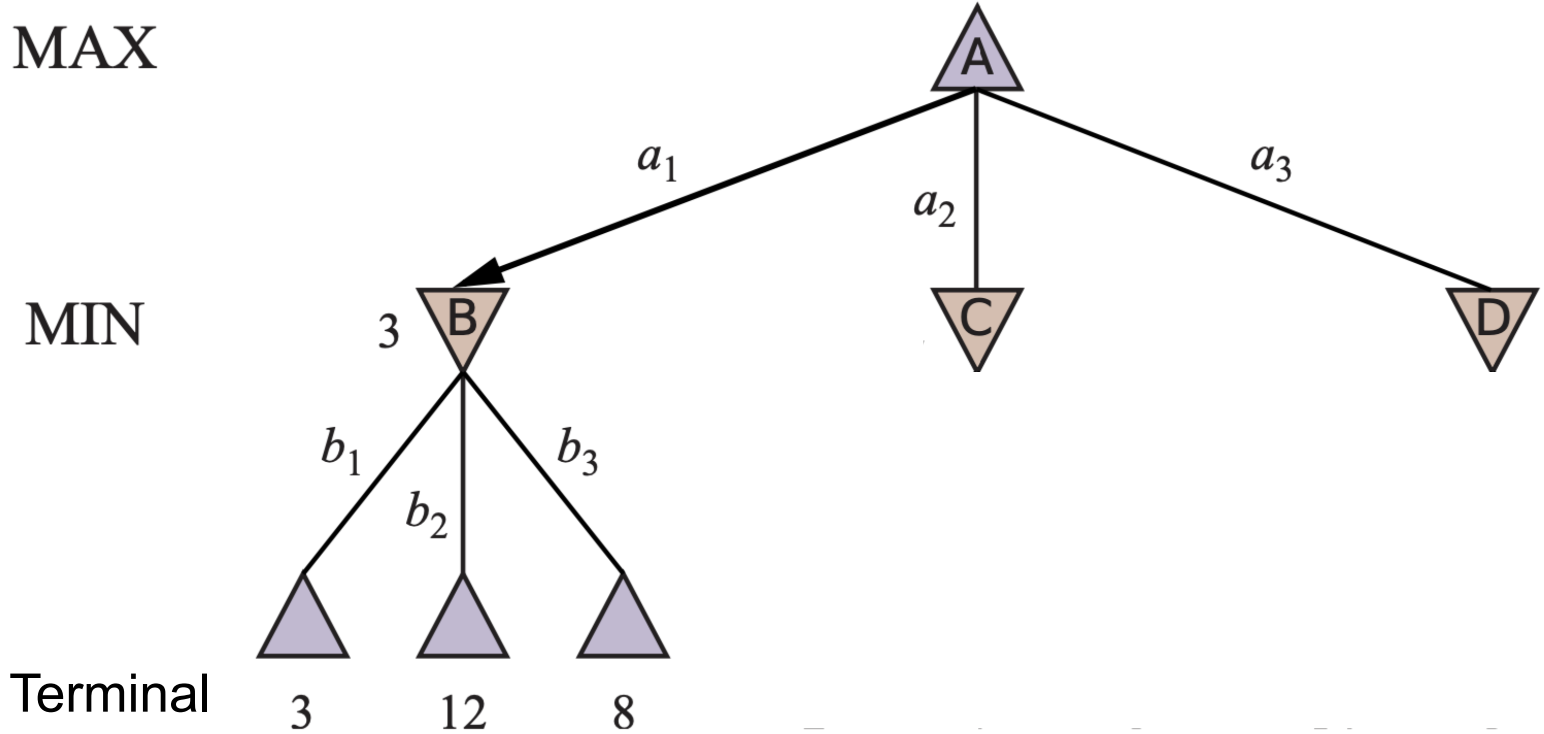
Example: Minimax Search for Two-Ply Game Trees



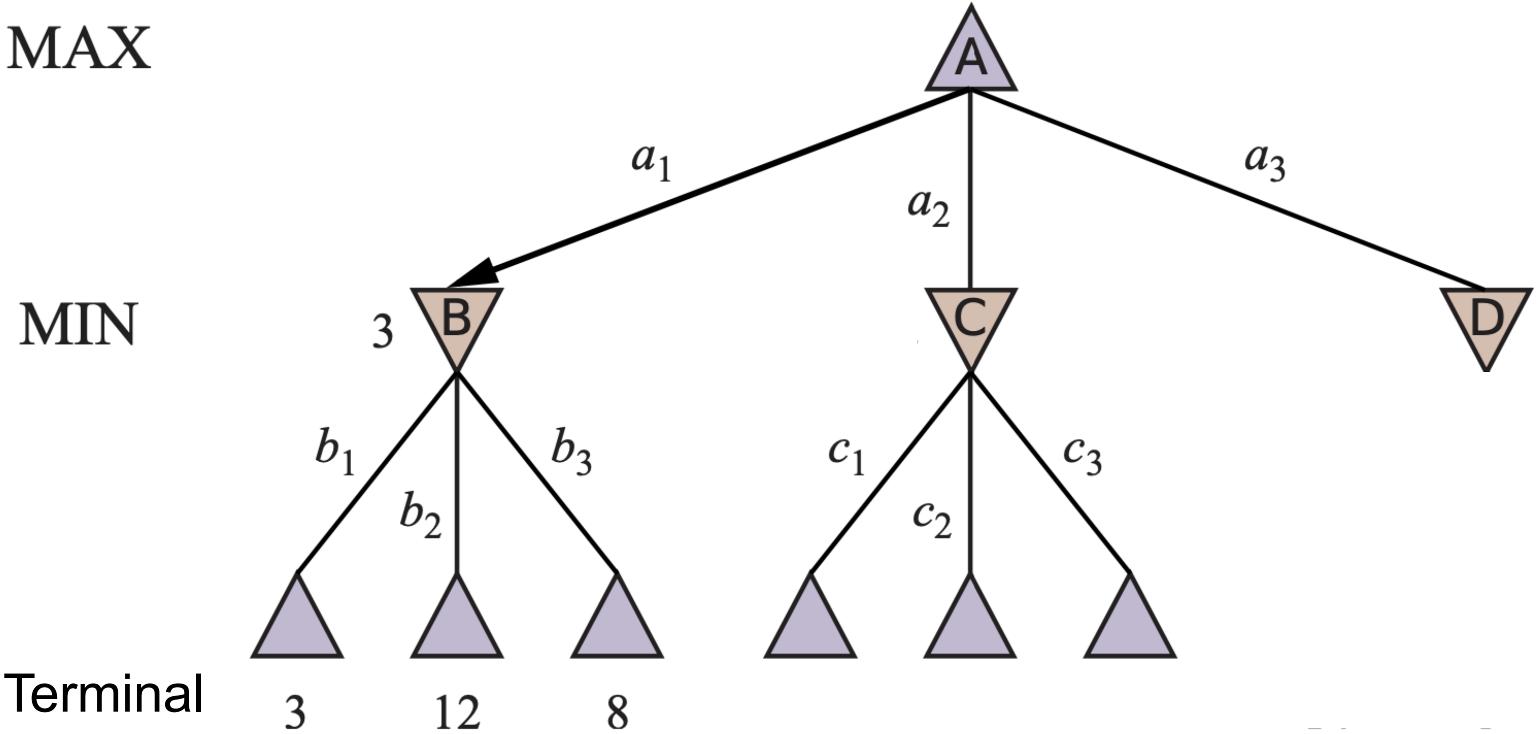
Example: Minimax Search for Two-Ply Game Trees



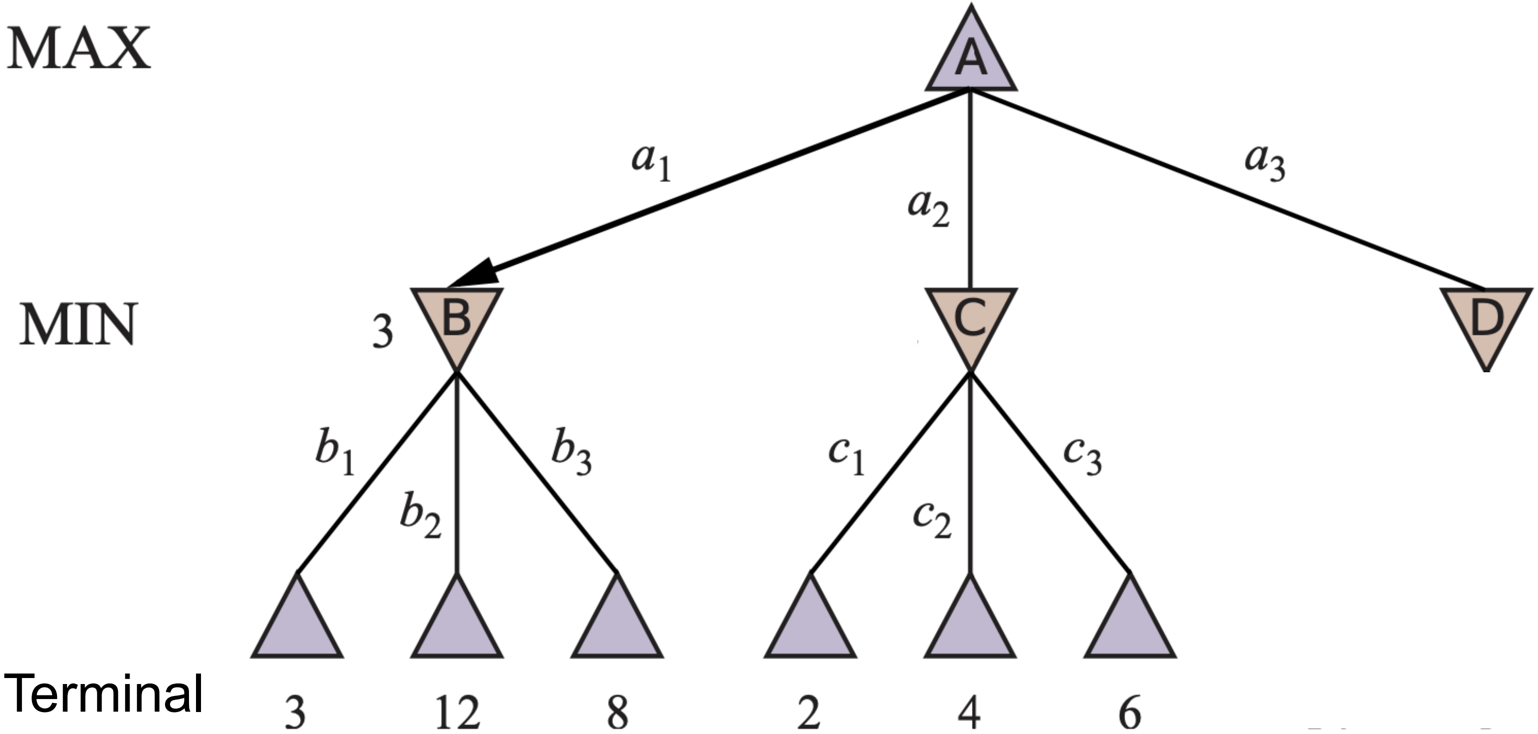
Example: Minimax Search for Two-Ply Game Trees



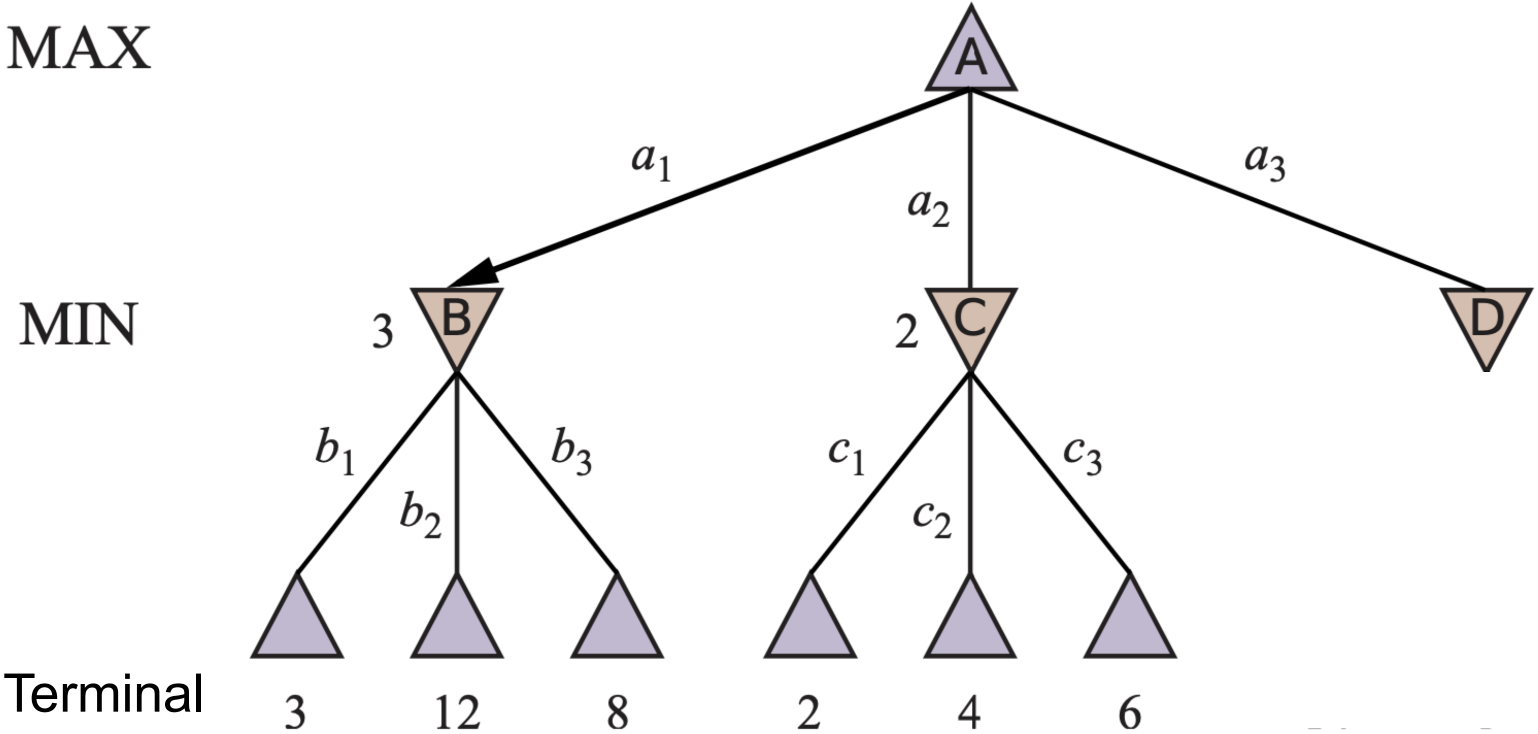
Example: Minimax Search for Two-Ply Game Trees



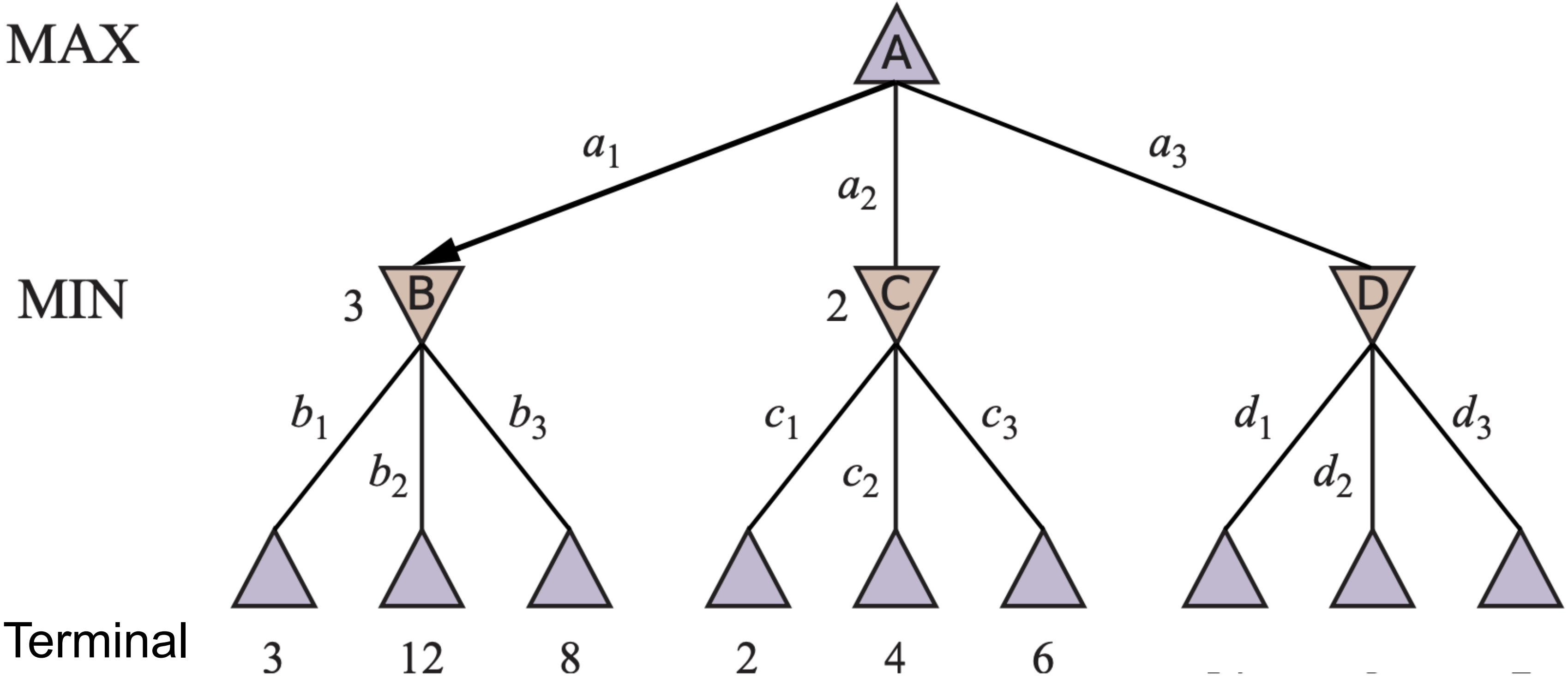
Example: Minimax Search for Two-Ply Game Trees



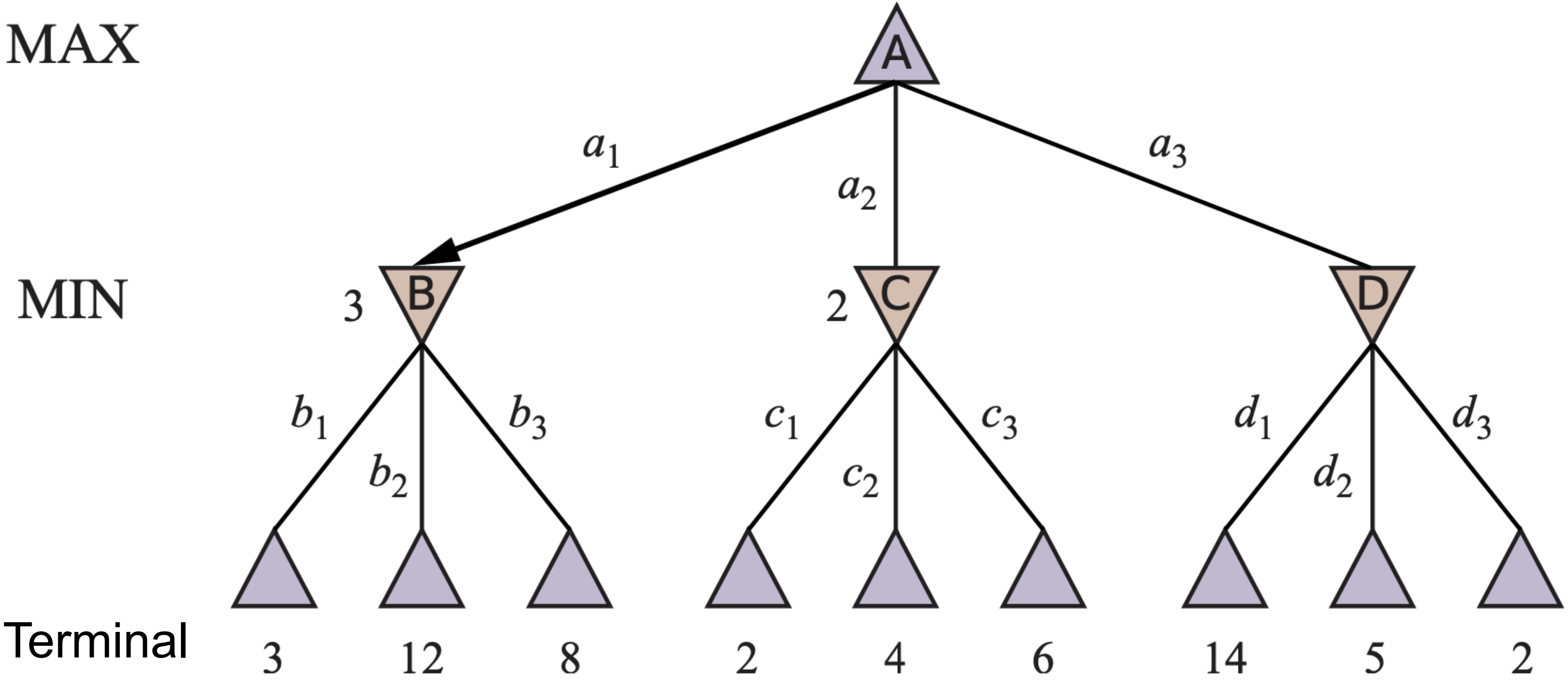
Example: Minimax Search for Two-Ply Game Trees



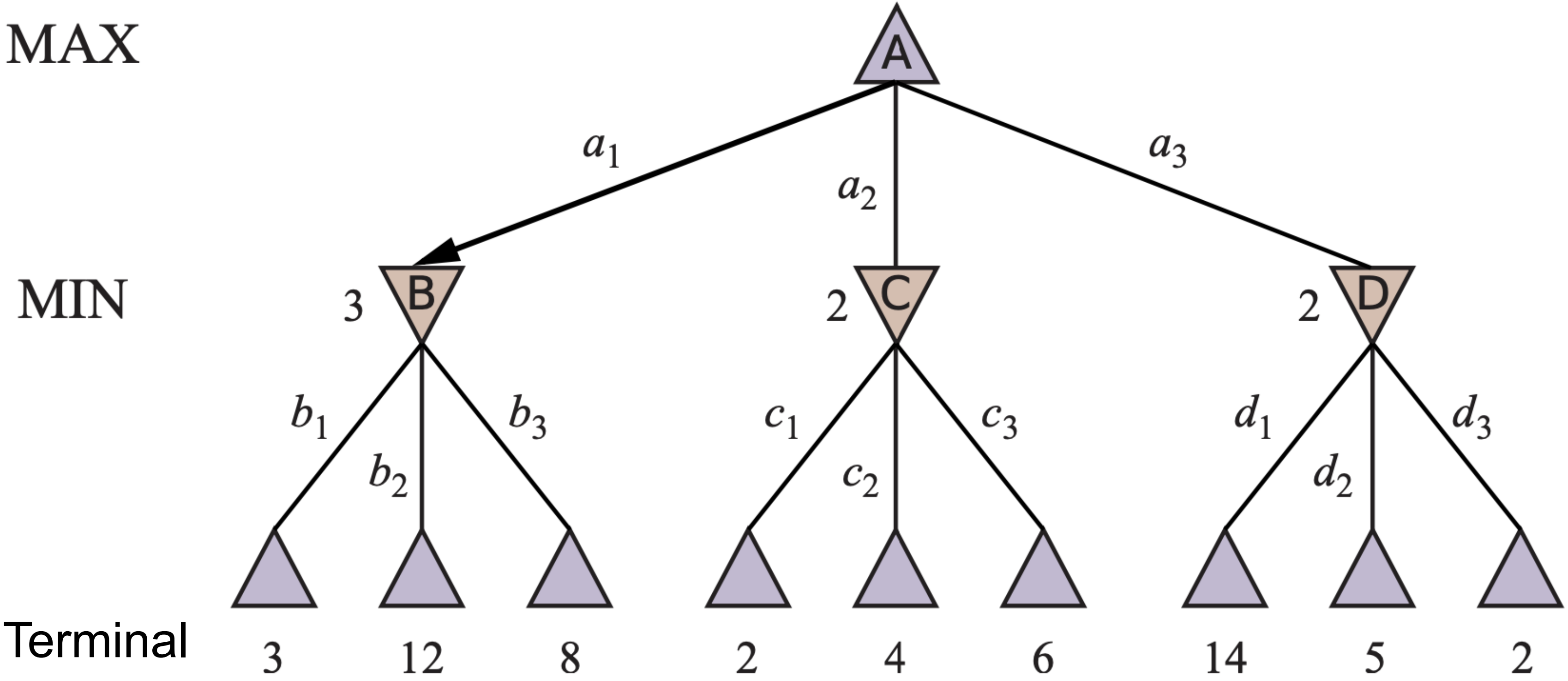
Example: Minimax Search for Two-Ply Game Trees



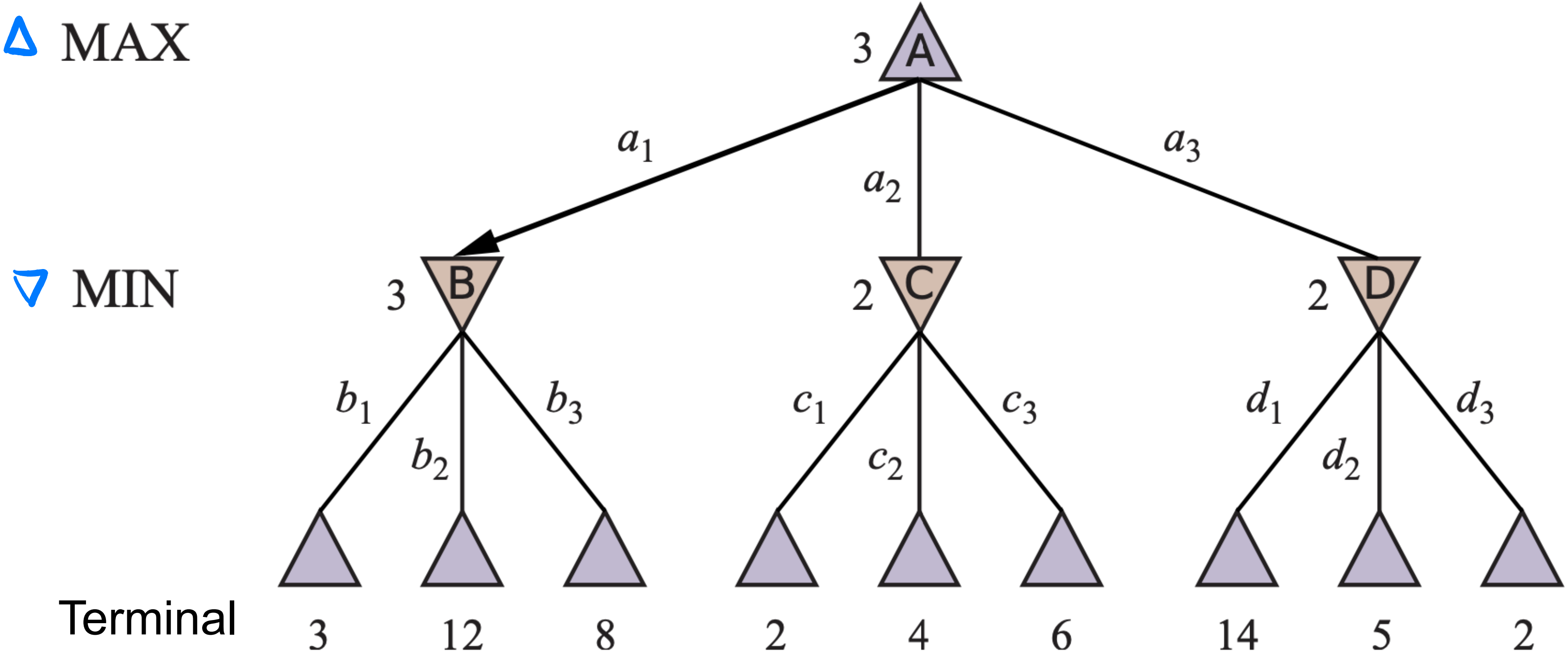
Example: Minimax Search for Two-Ply Game Trees



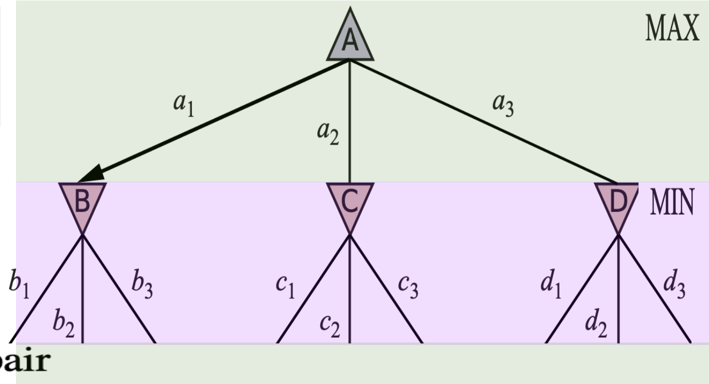
Example: Minimax Search for Two-Ply Game Trees



Example: Minimax Search for Two-Ply Game Trees



Minimax Search Algorithm




```
function MINIMAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v, move ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
      if v2 > v then
         v, move ← v2, a
   return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v, move ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

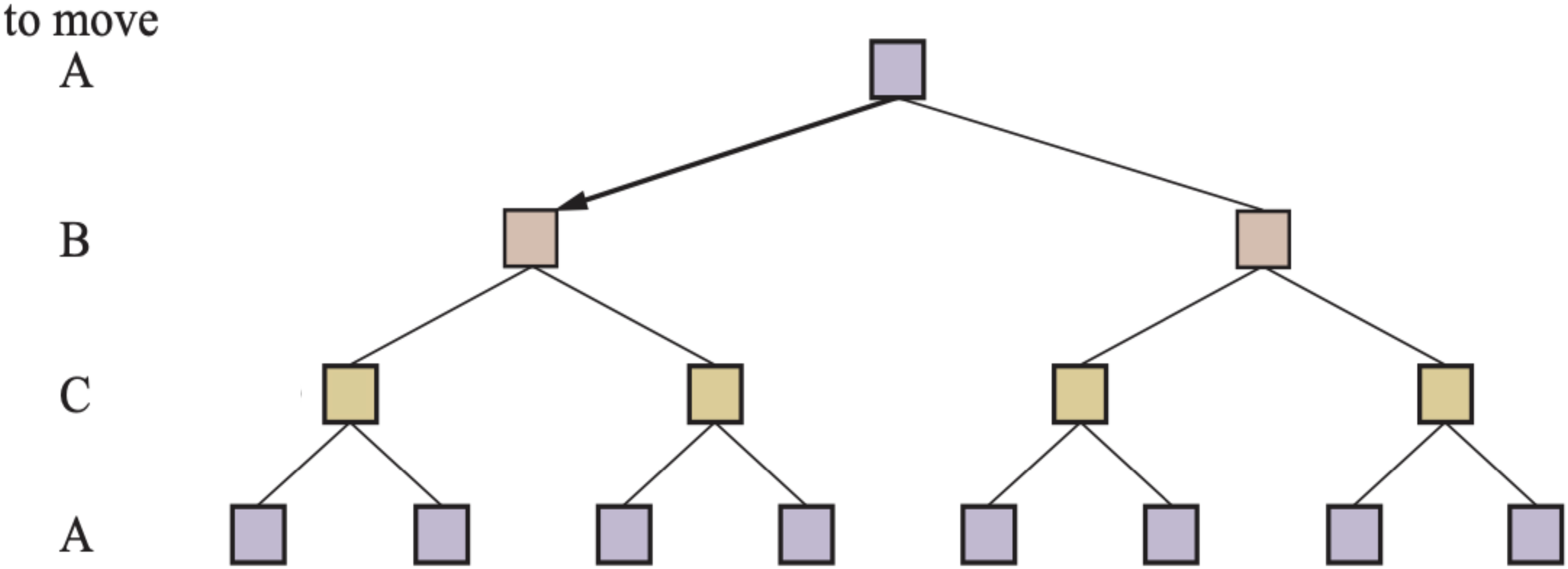
Properties of Minimax

Note. *b* is the branching factor and *m* is the depth limit

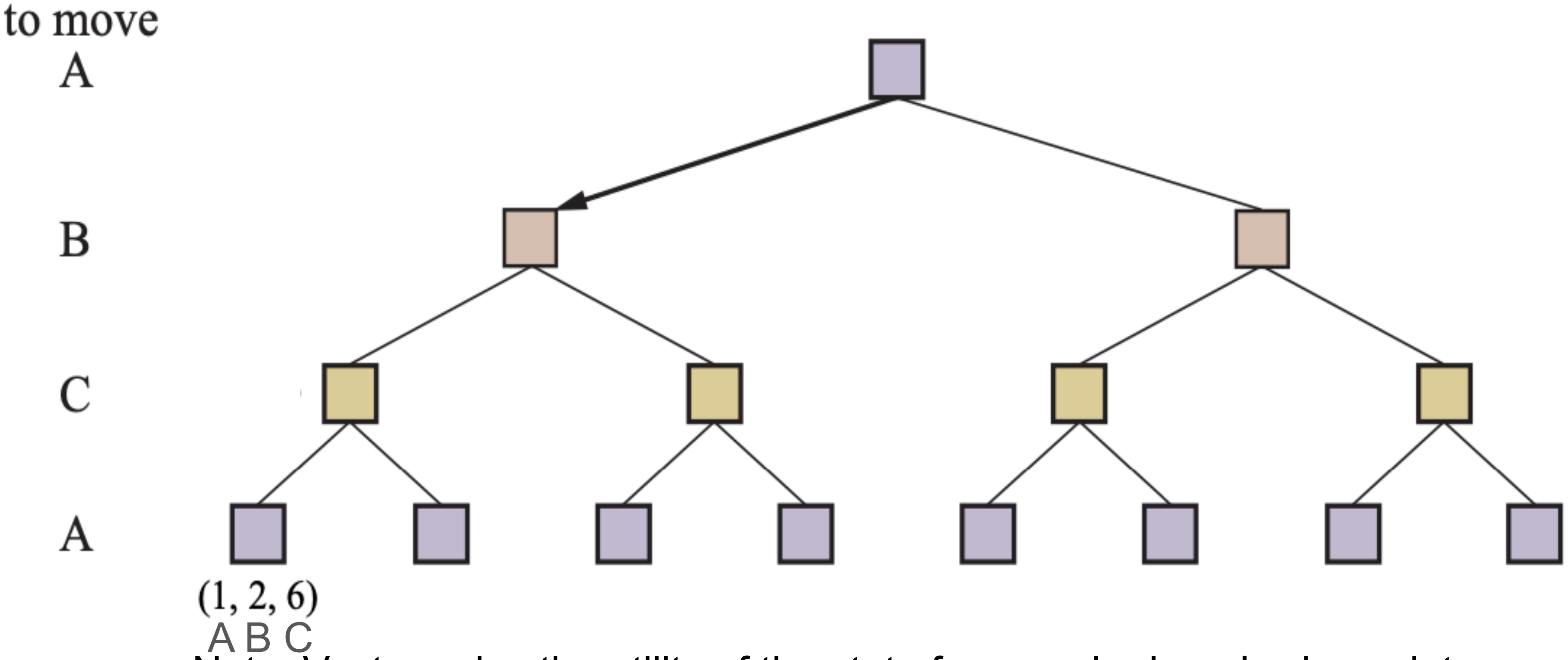
- Complete?
  - Yes, if the tree is finite
- Optimal cost?
  - Yes
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$ (depth-first exploration)

DFS

Example of Three Players

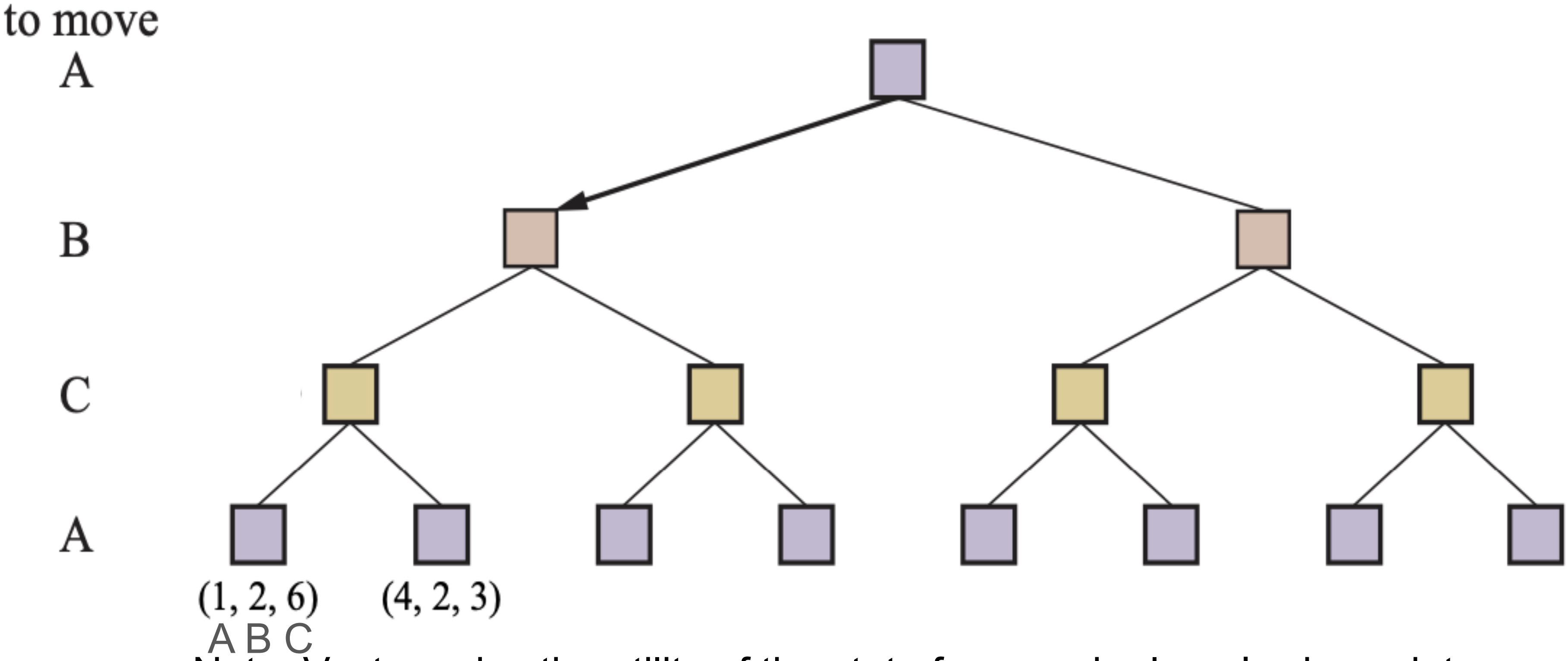


Example of Three Players



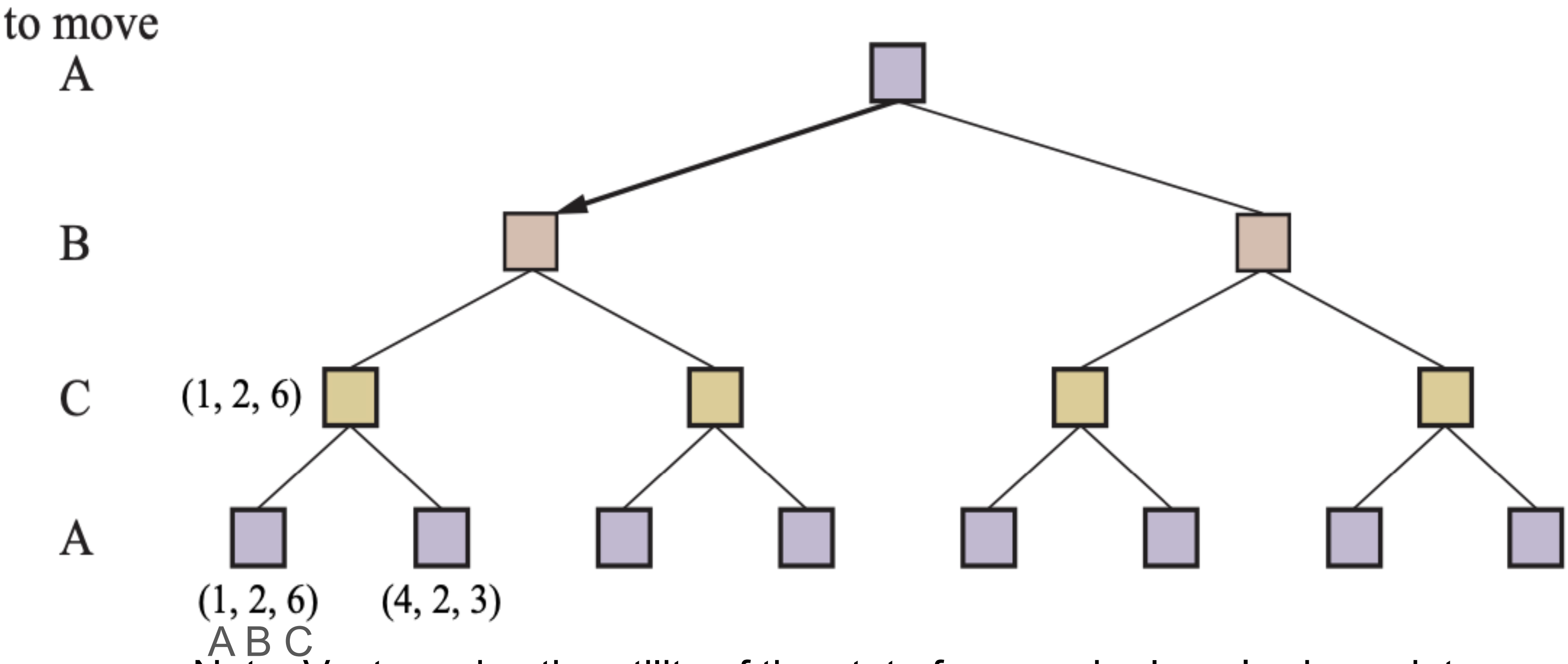
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



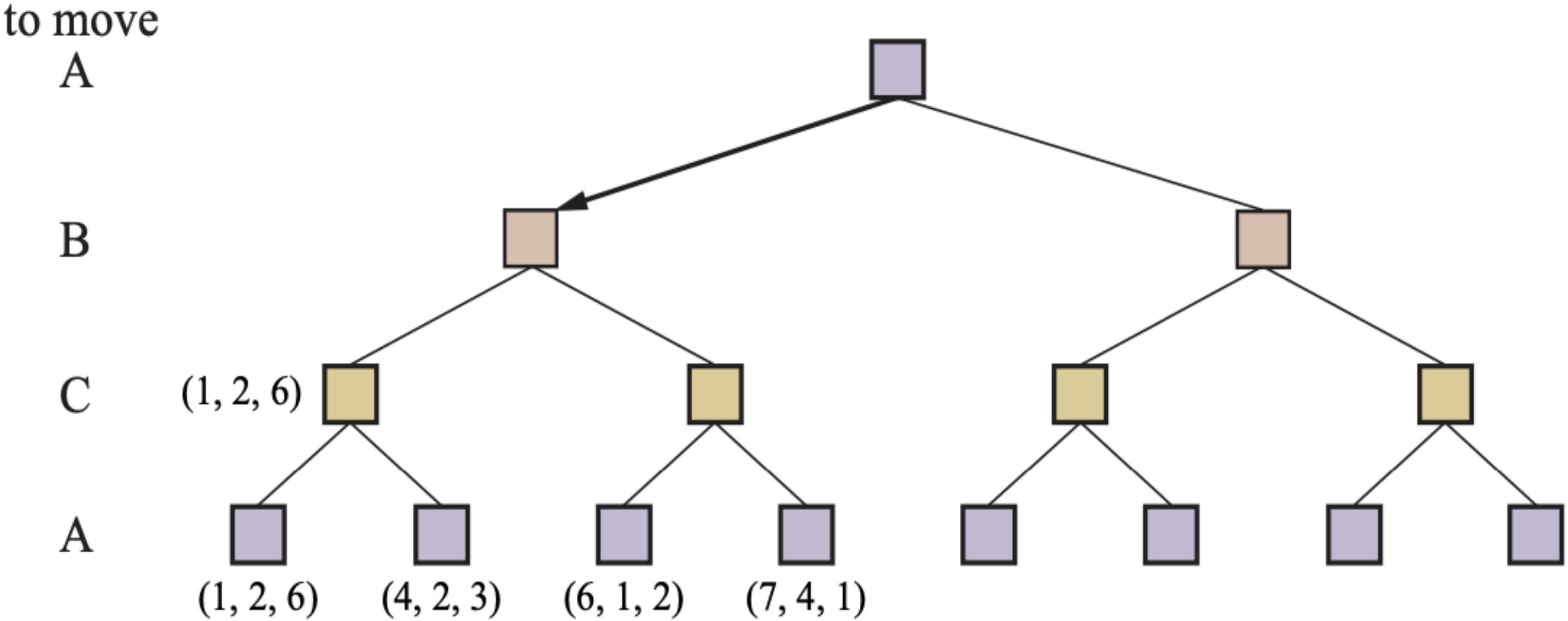
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



Note. Vectors give the utility of the state from each player’s viewpoint

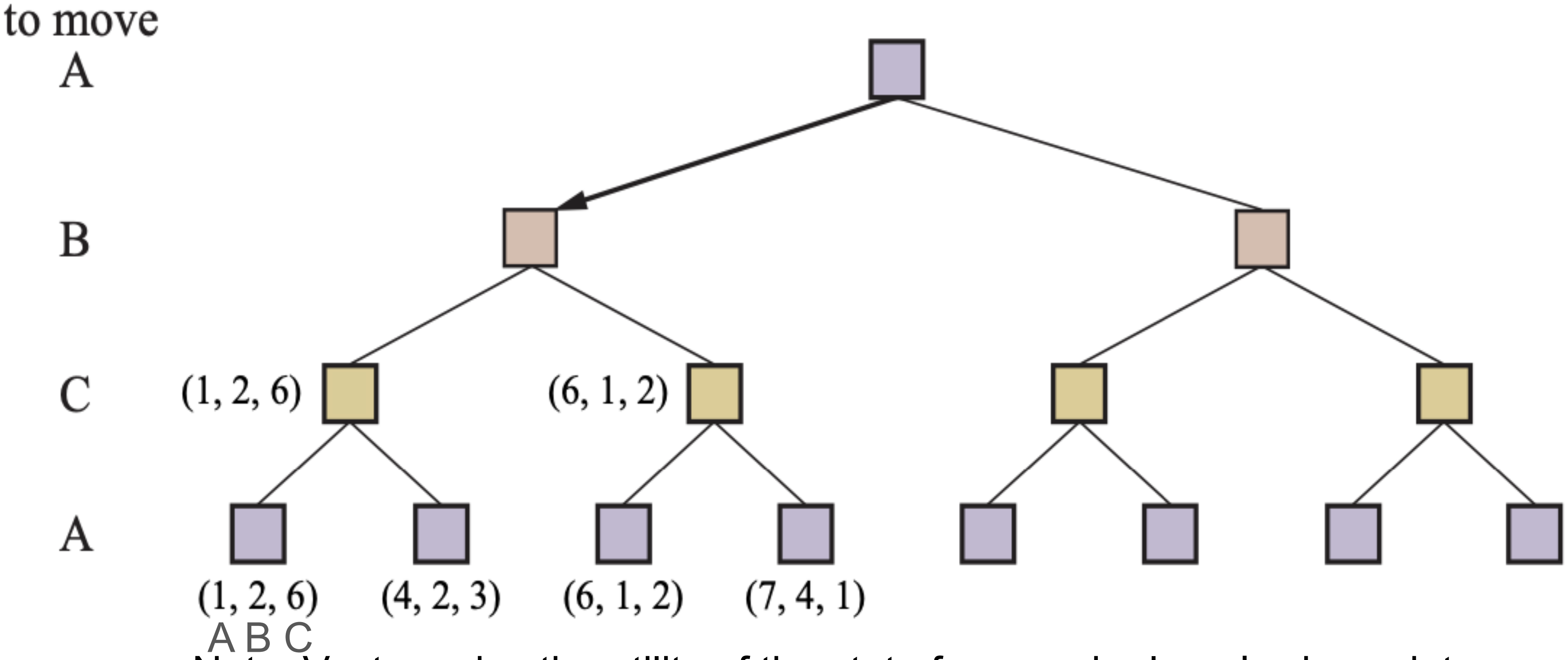
Example of Three Players



A B C

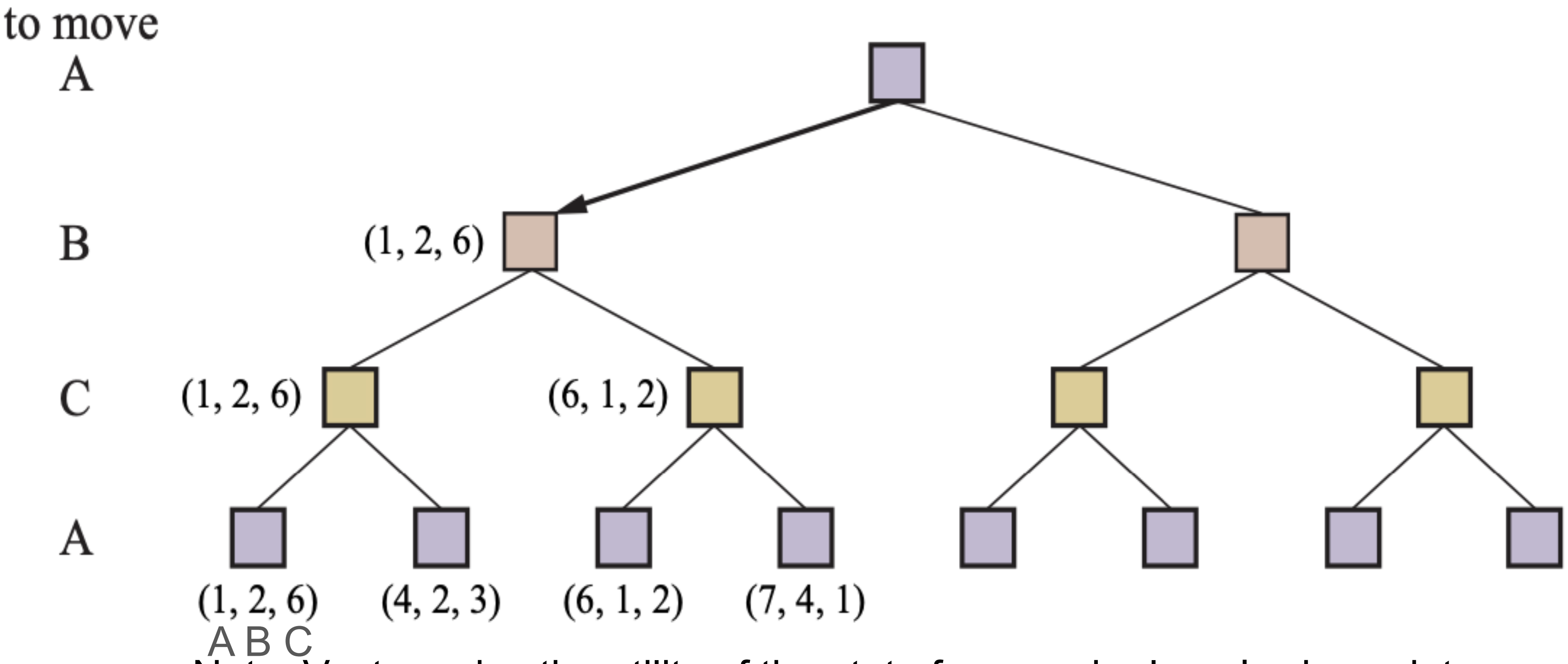
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



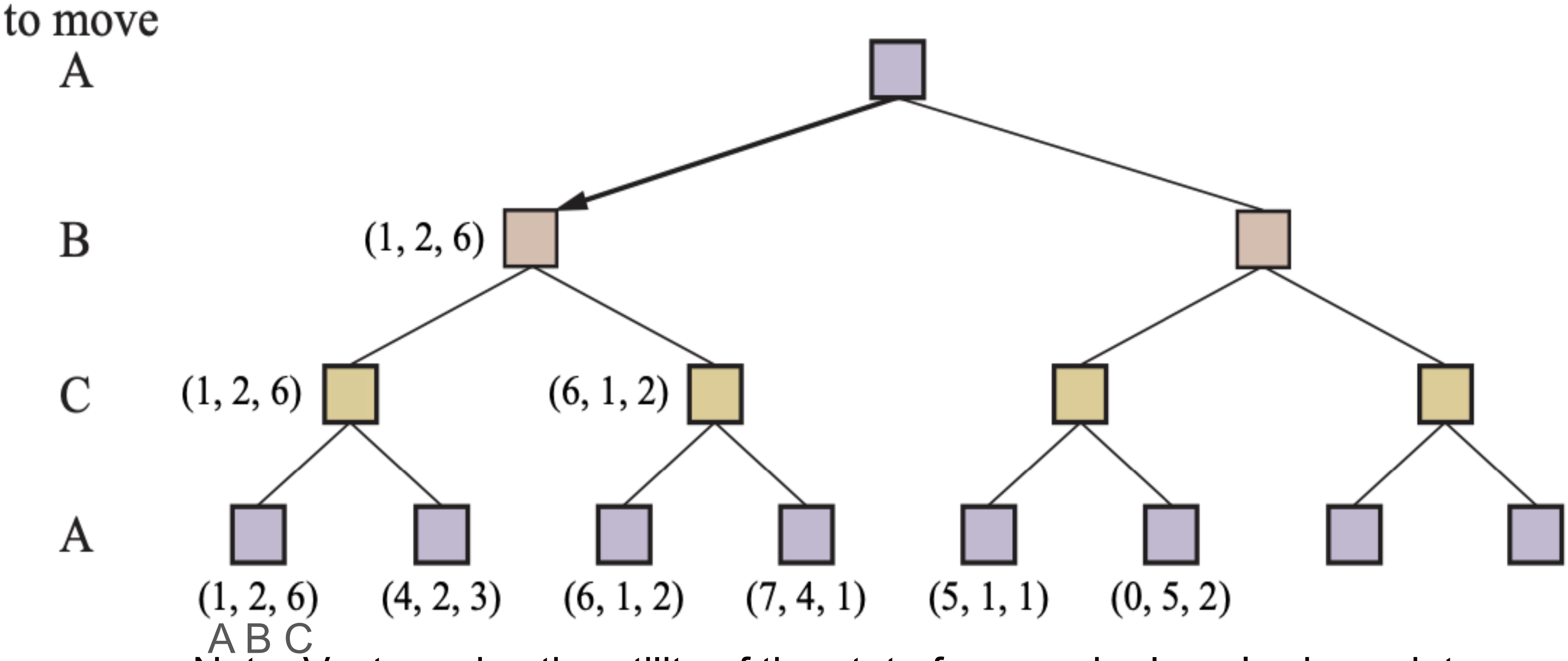
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



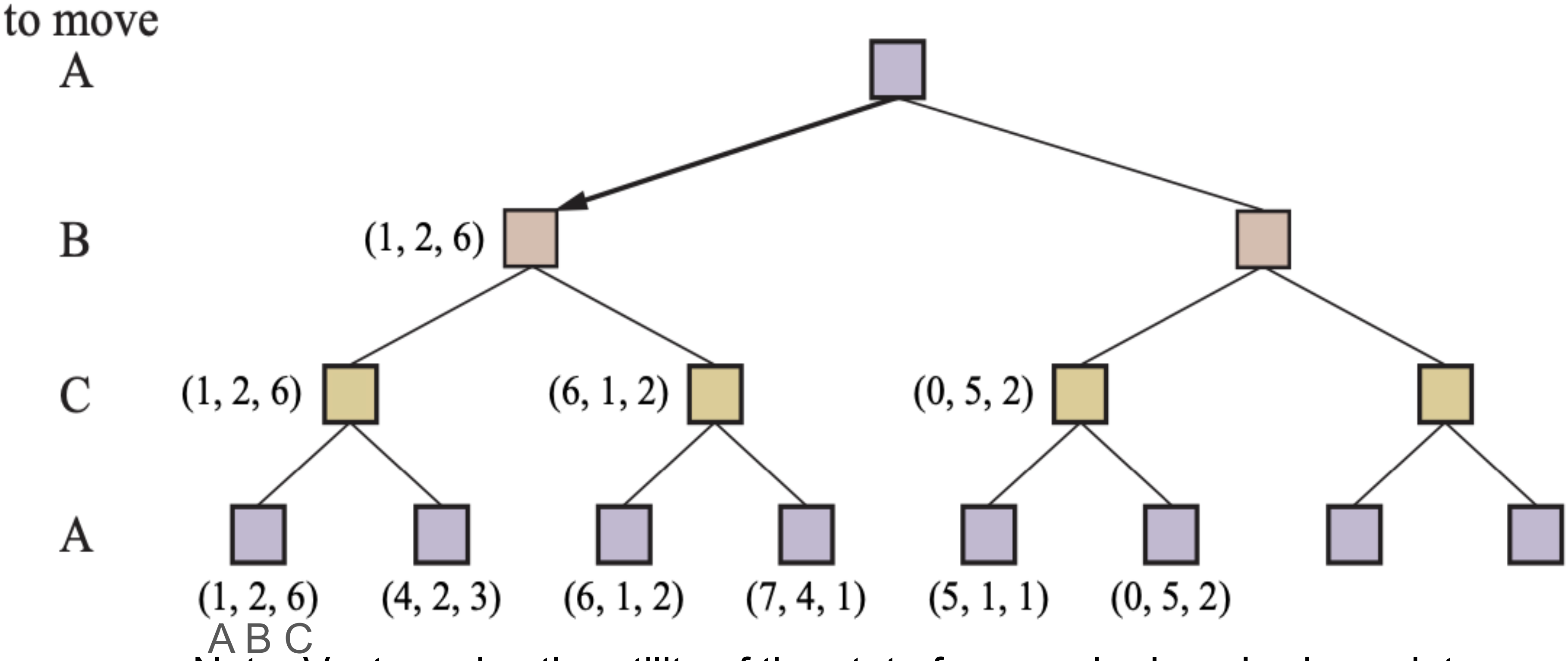
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



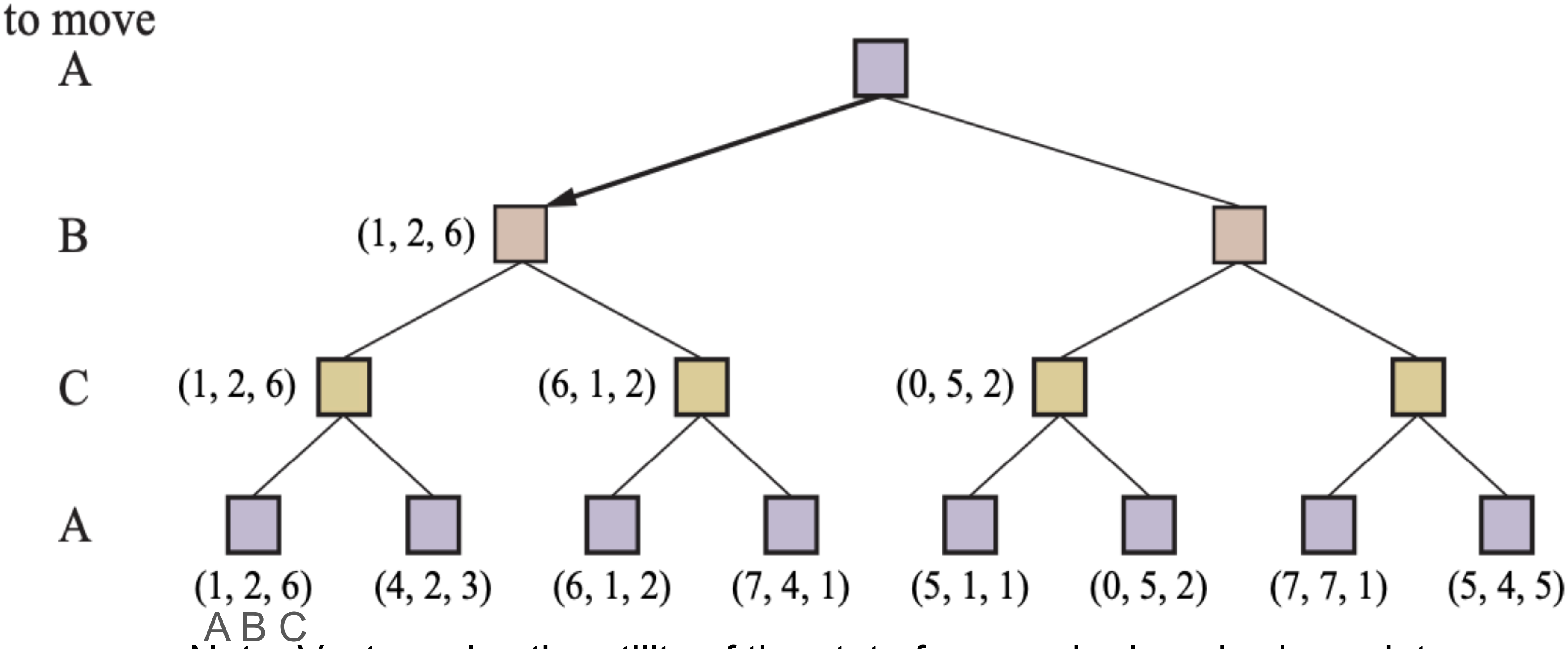
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



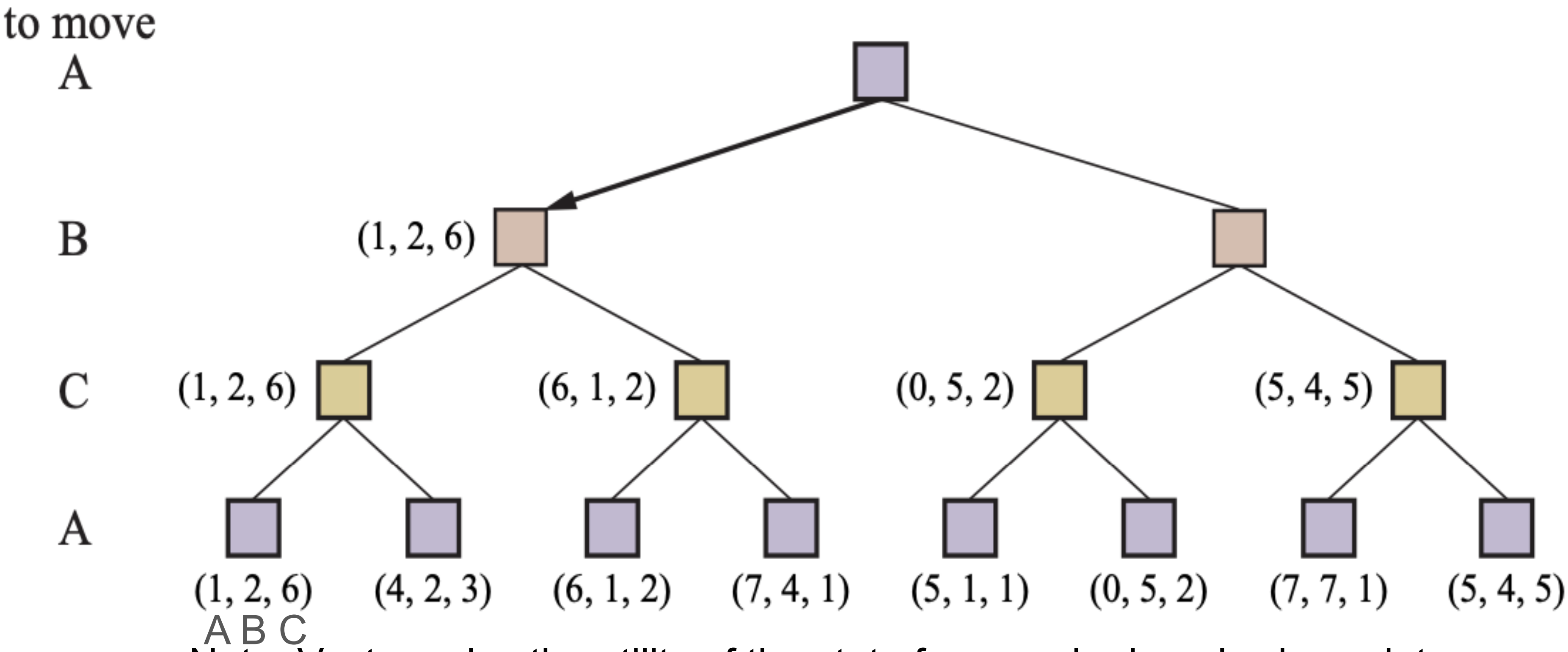
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



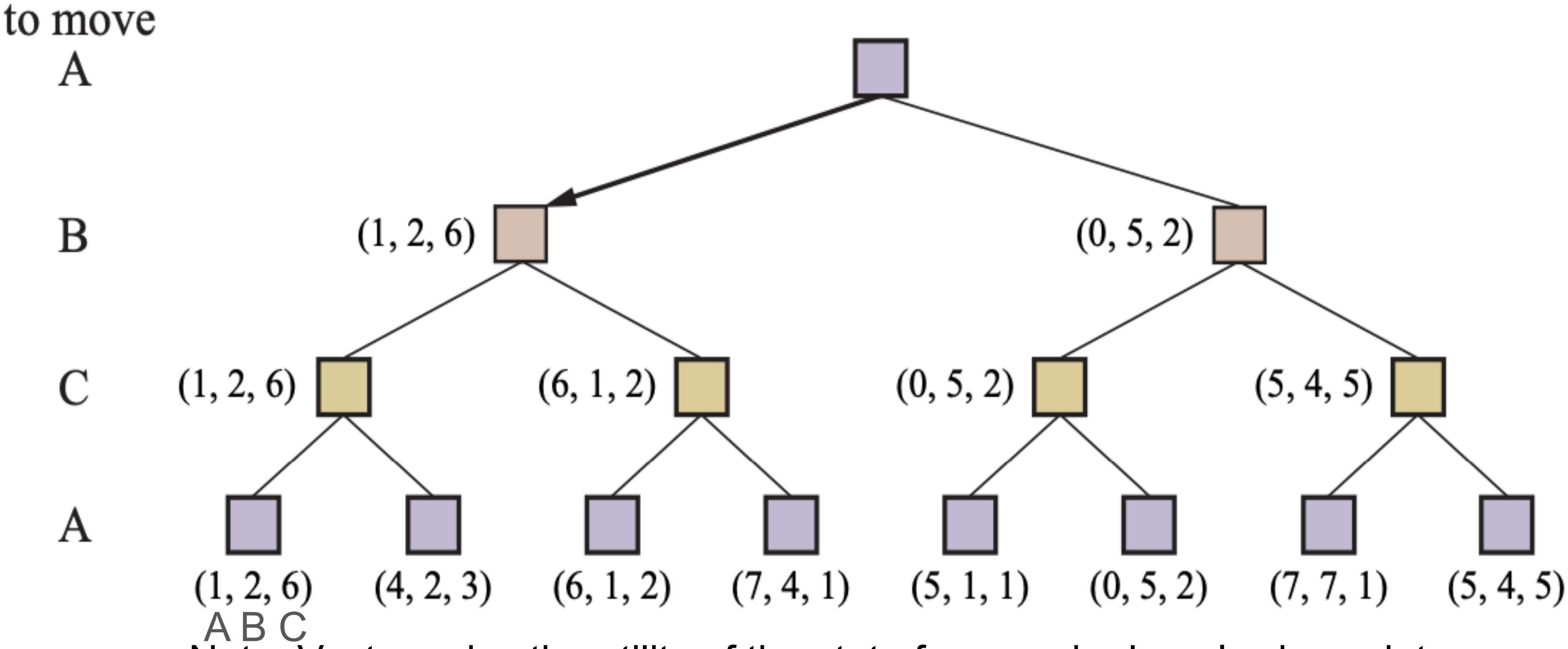
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



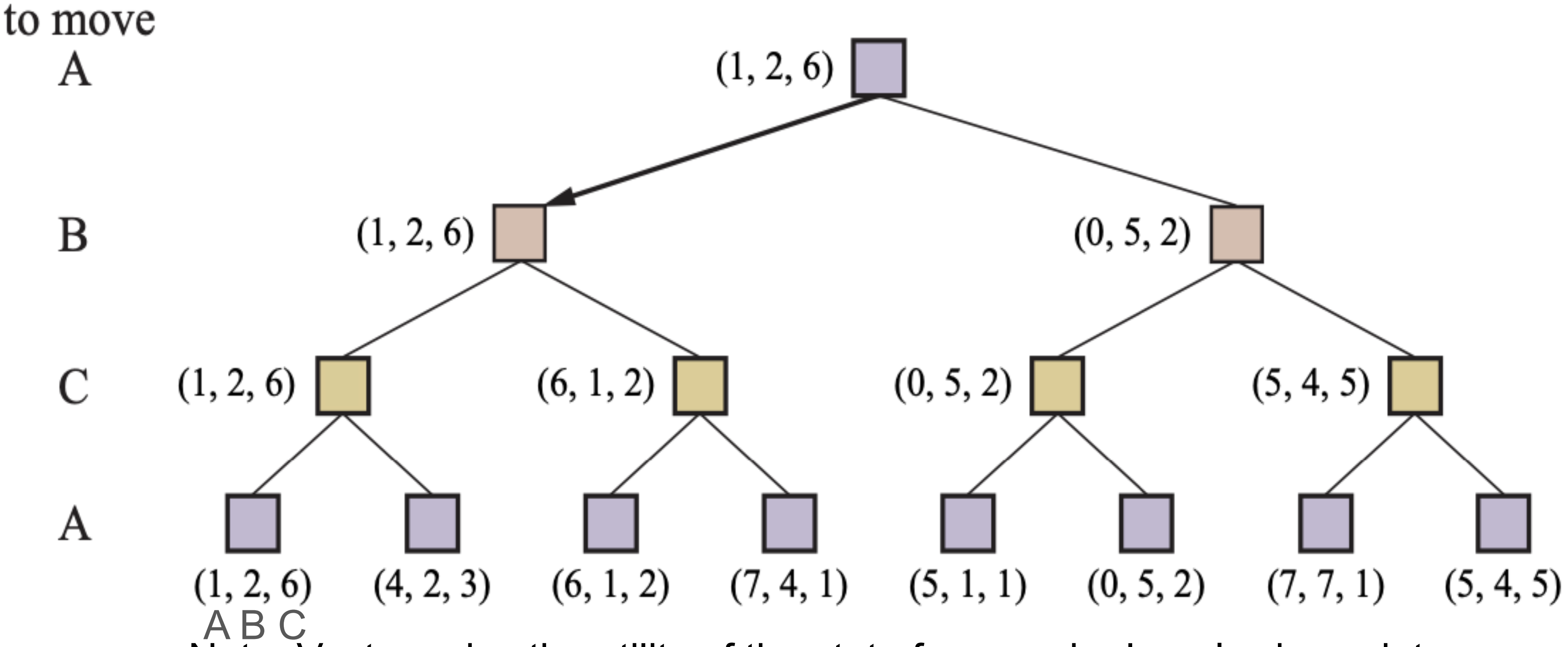
Note. Vectors give the utility of the state from each player's viewpoint

Example of Three Players



Note. Vectors give the utility of the state from each player's viewpoint

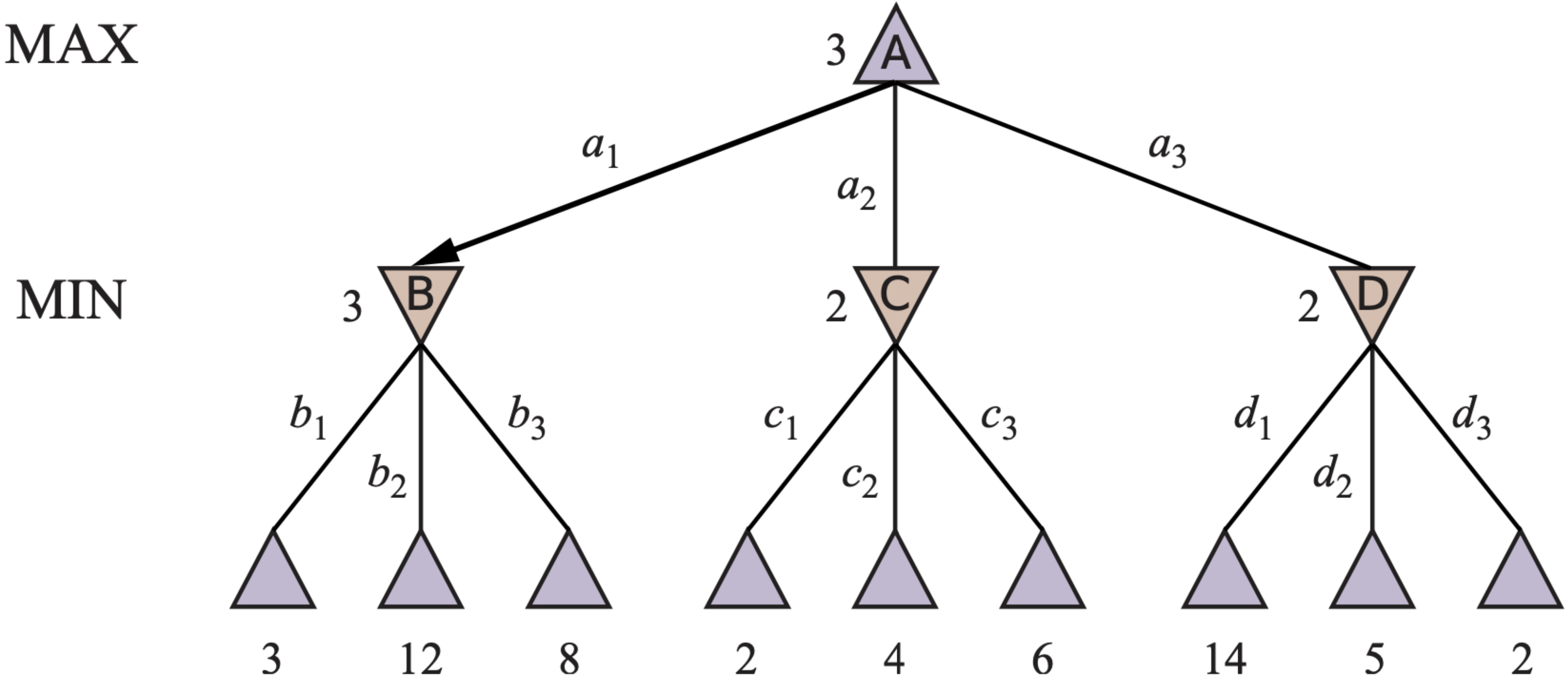
Example of Three Players



Note. Vectors give the utility of the state from each player's viewpoint

Performance?

Observation: Minimax Search



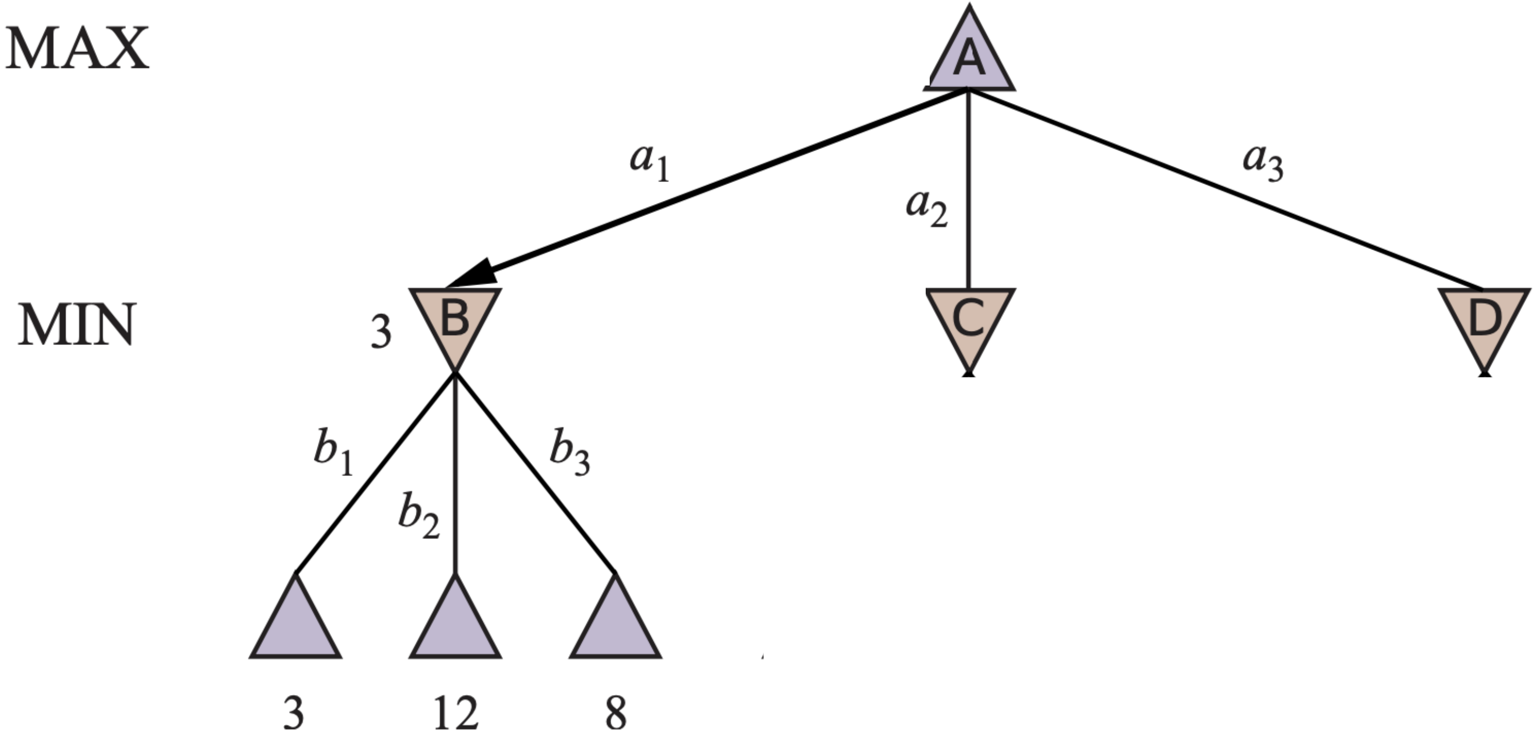
Expand All Nodes

Improvement ?

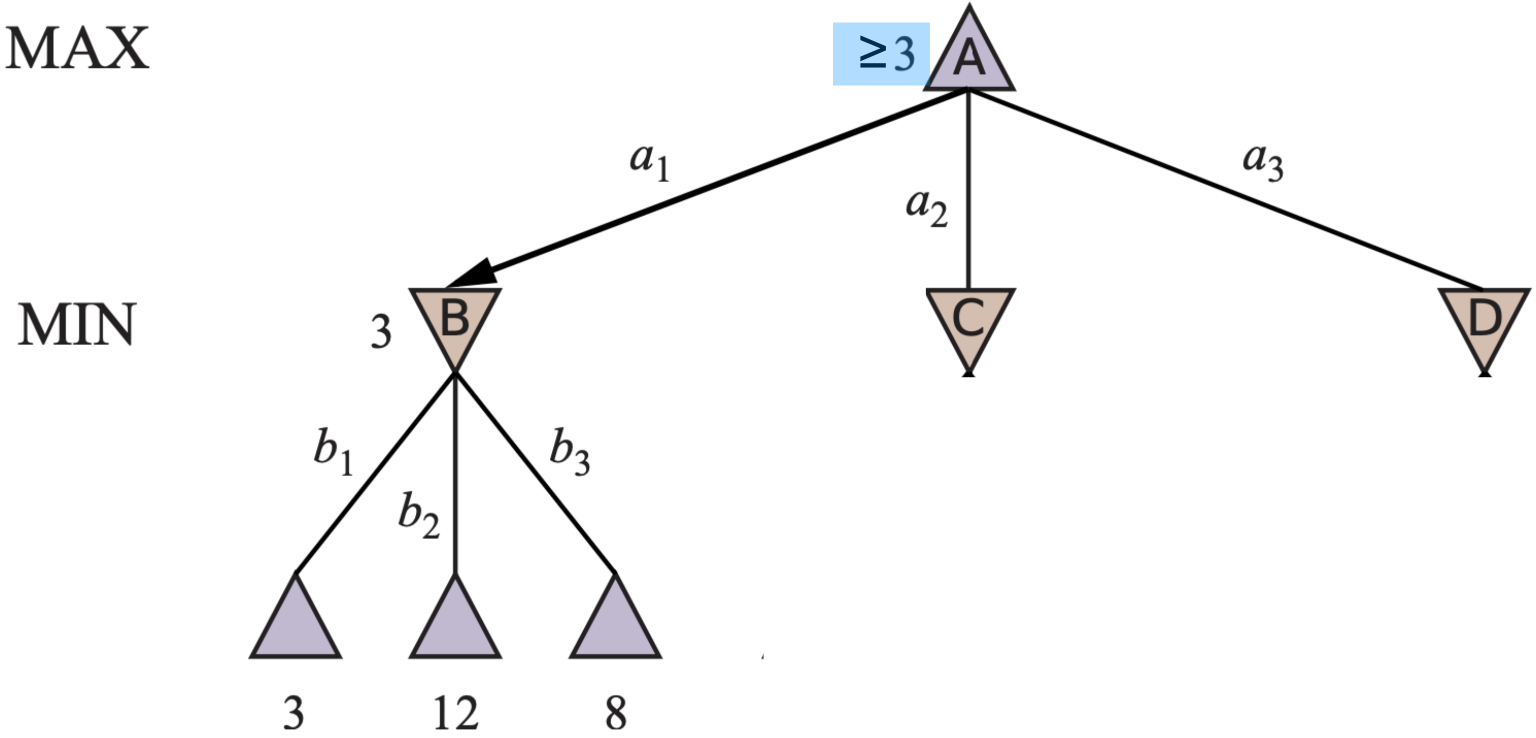
Expand ~~All~~ Nodes

Partial

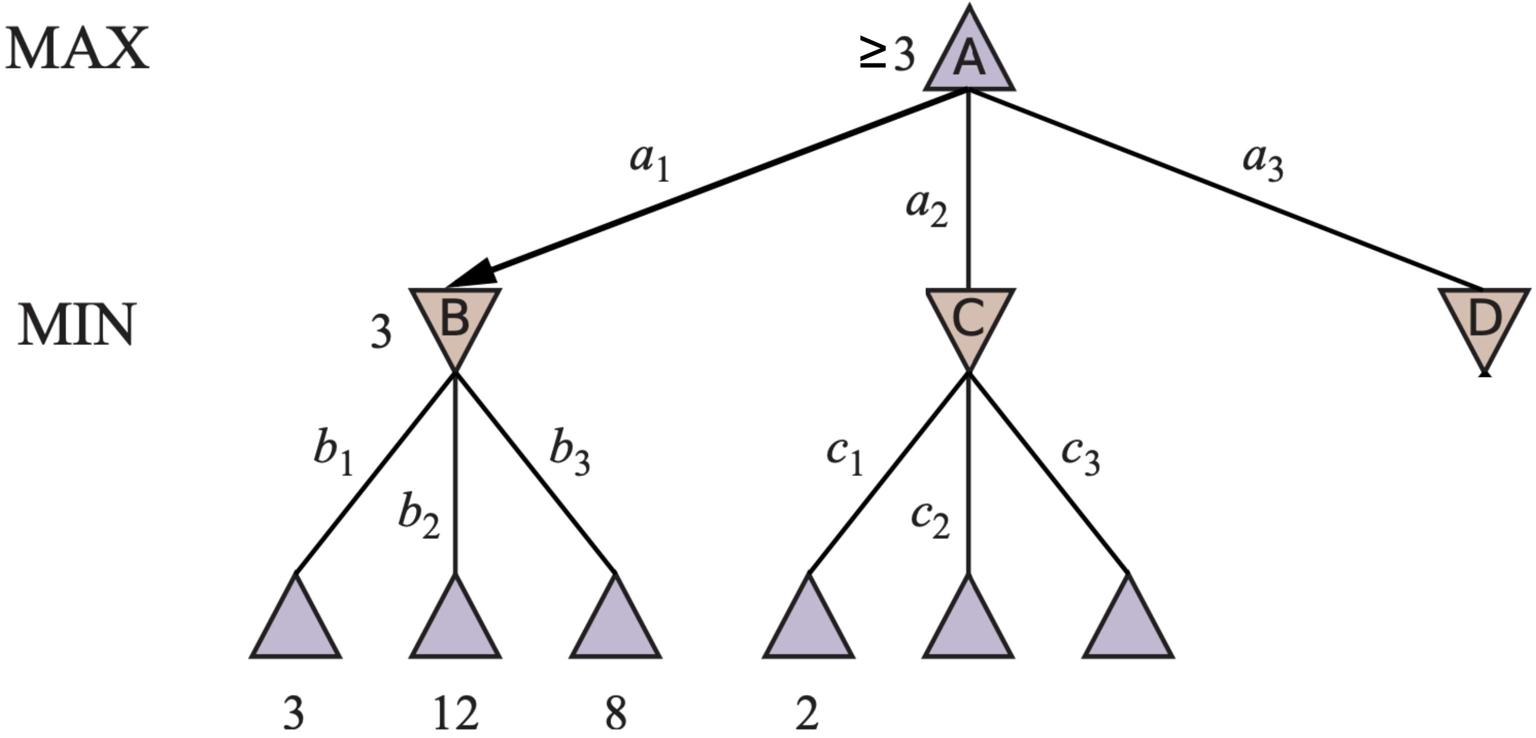
Pruning Technique?



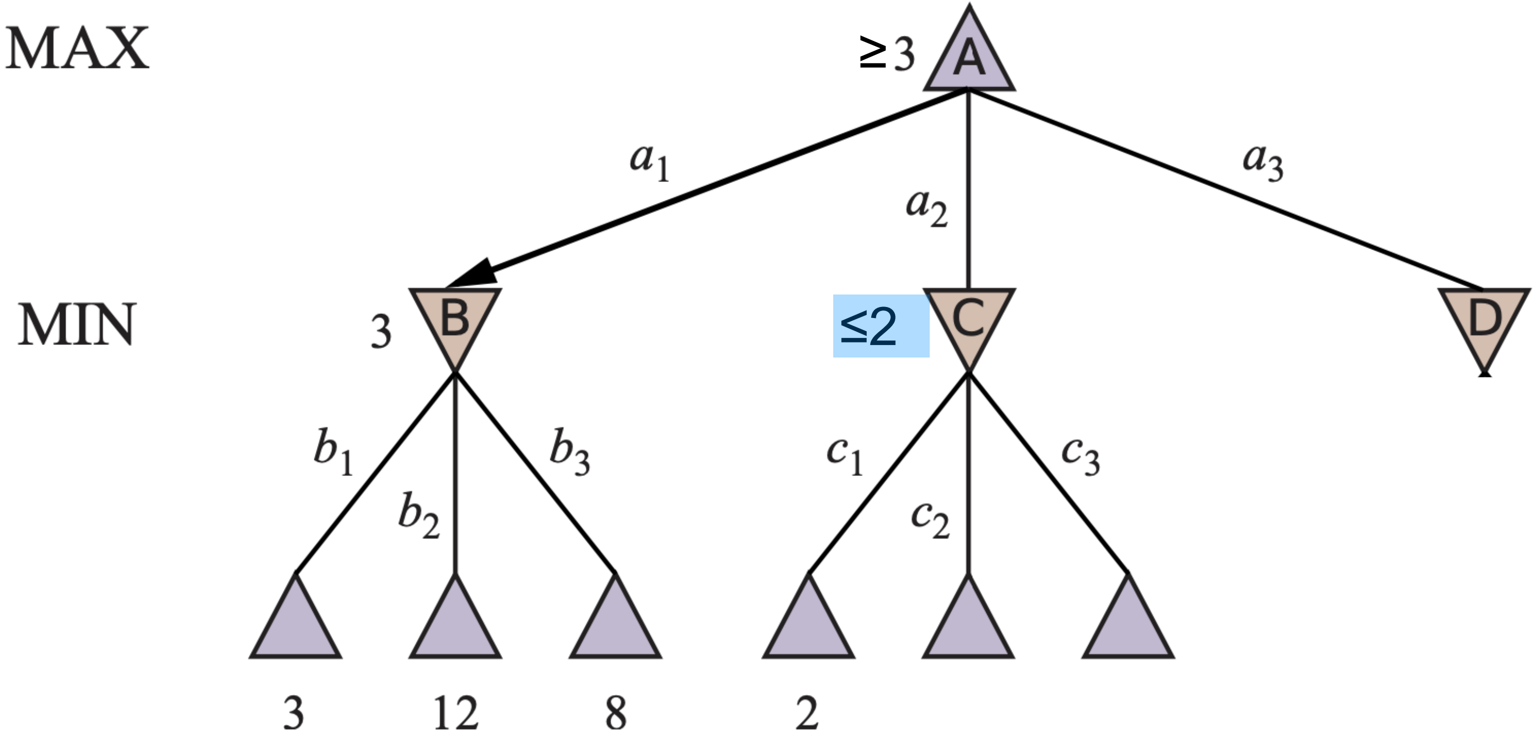
Pruning Technique?



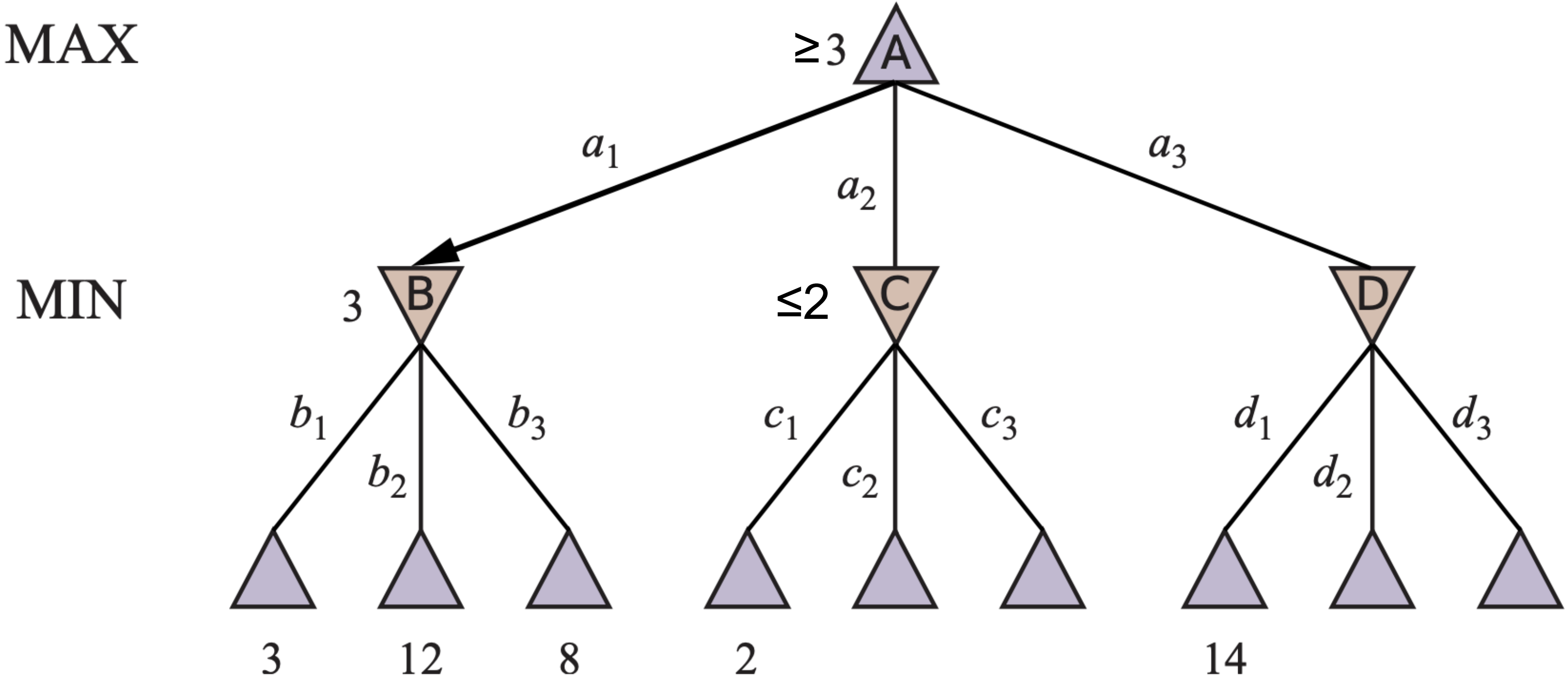
Pruning Technique?



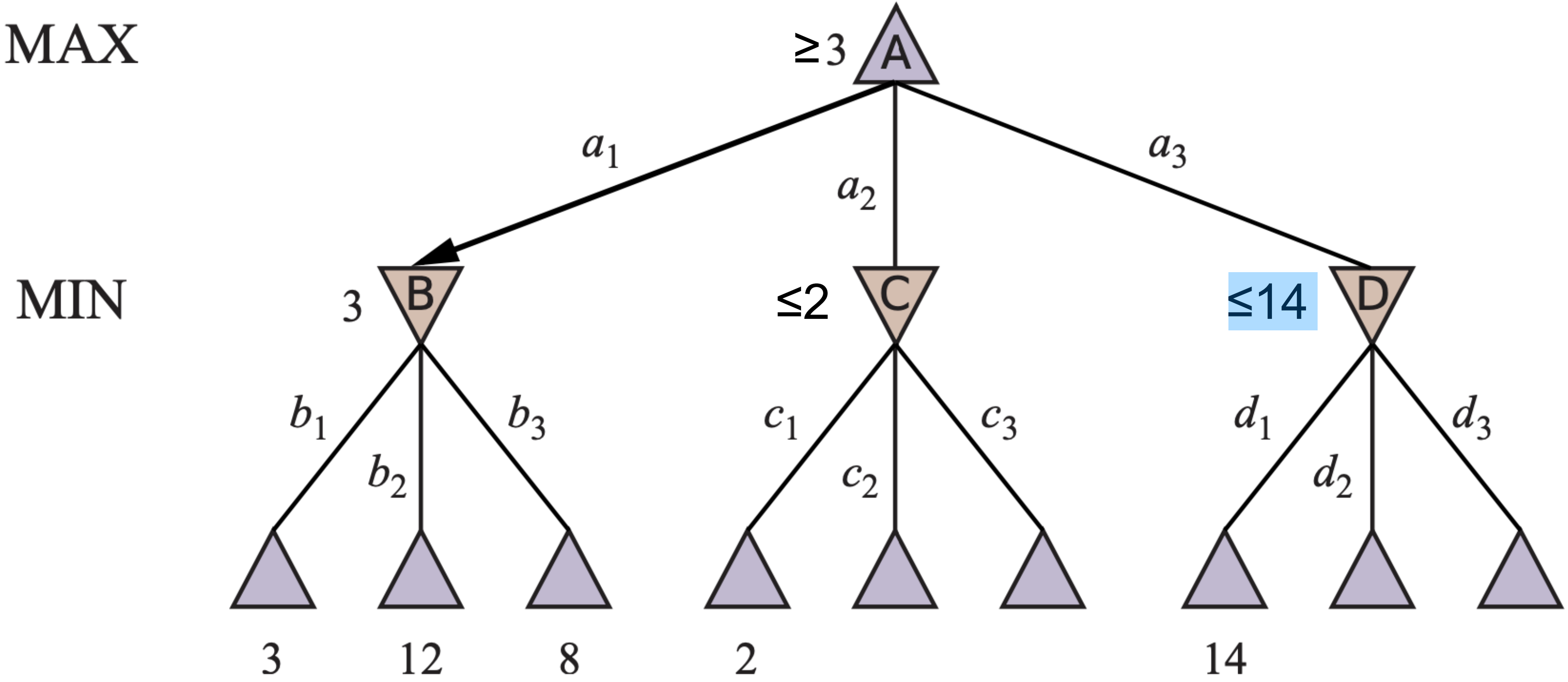
Pruning Technique?



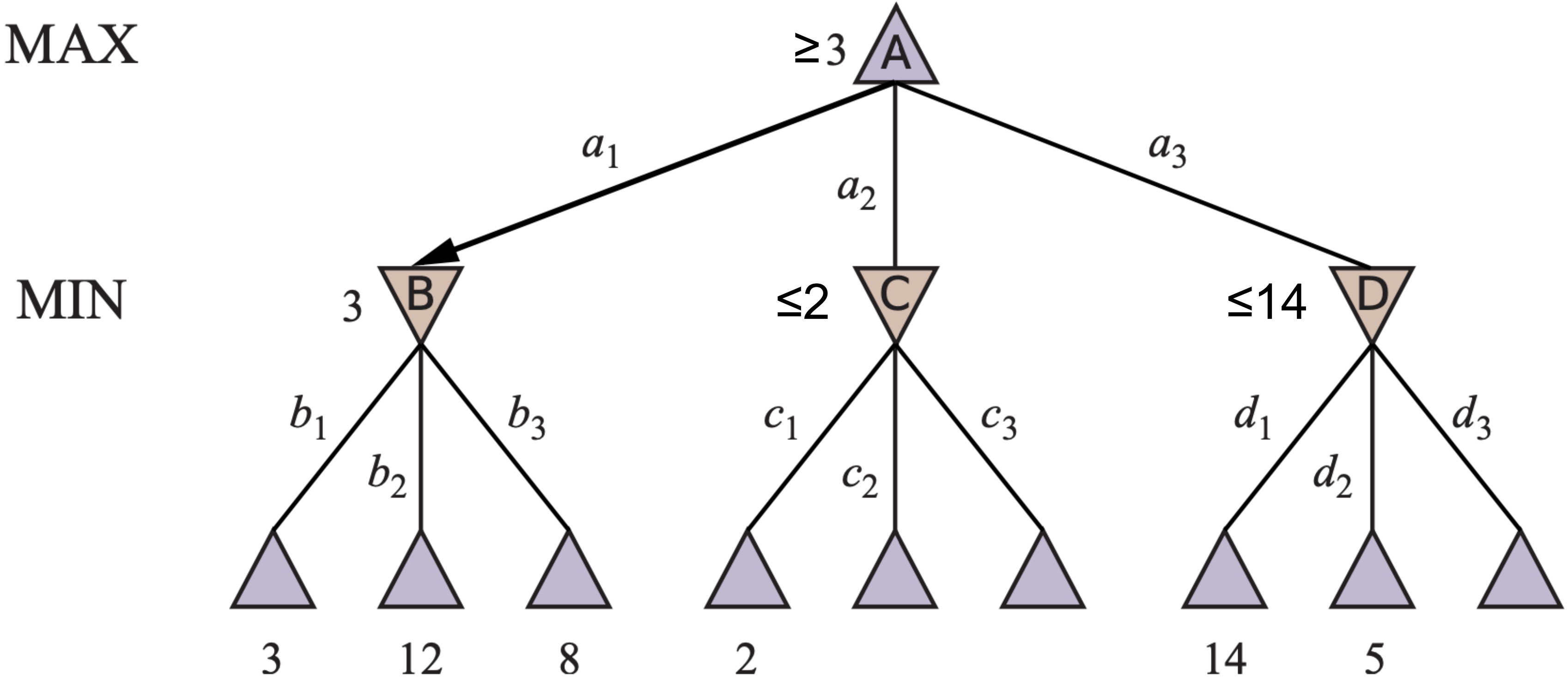
Pruning Technique?



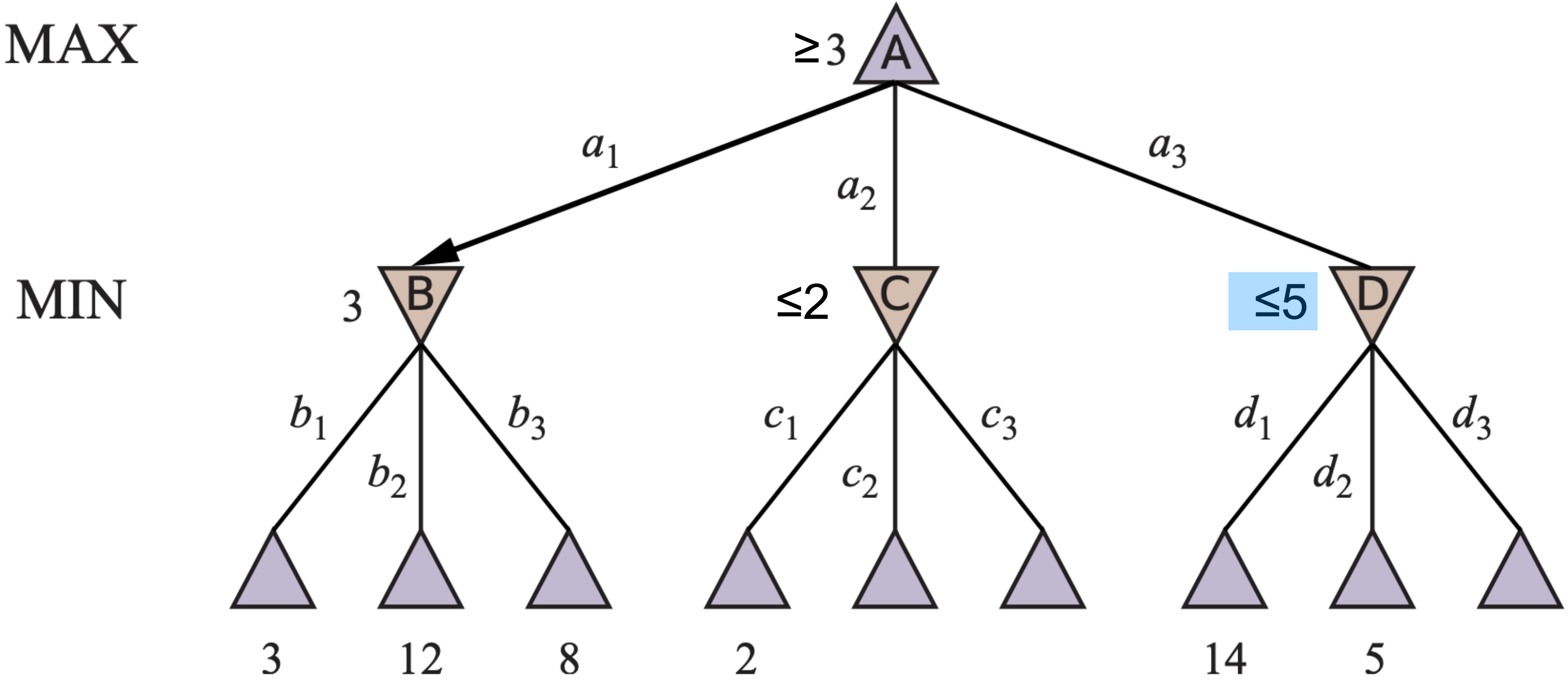
Pruning Technique?



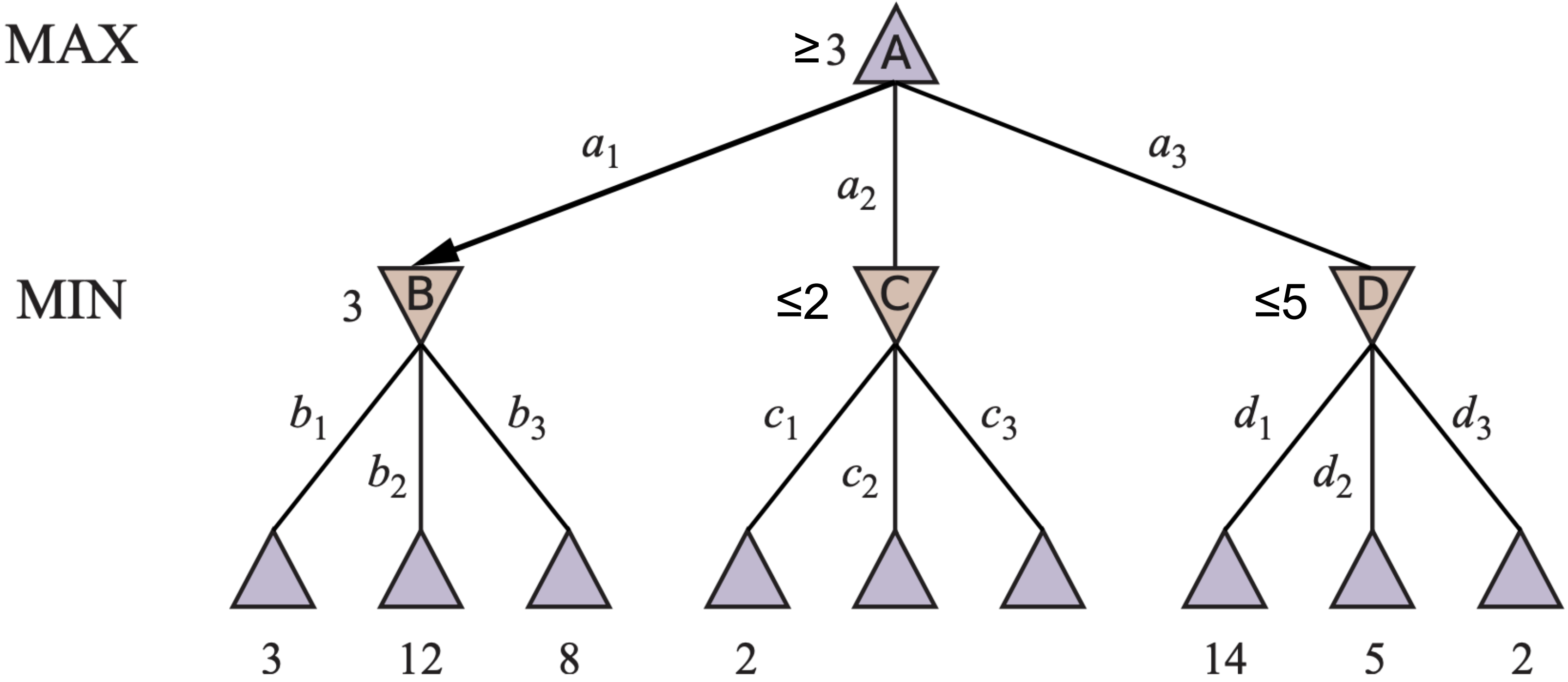
Pruning Technique?



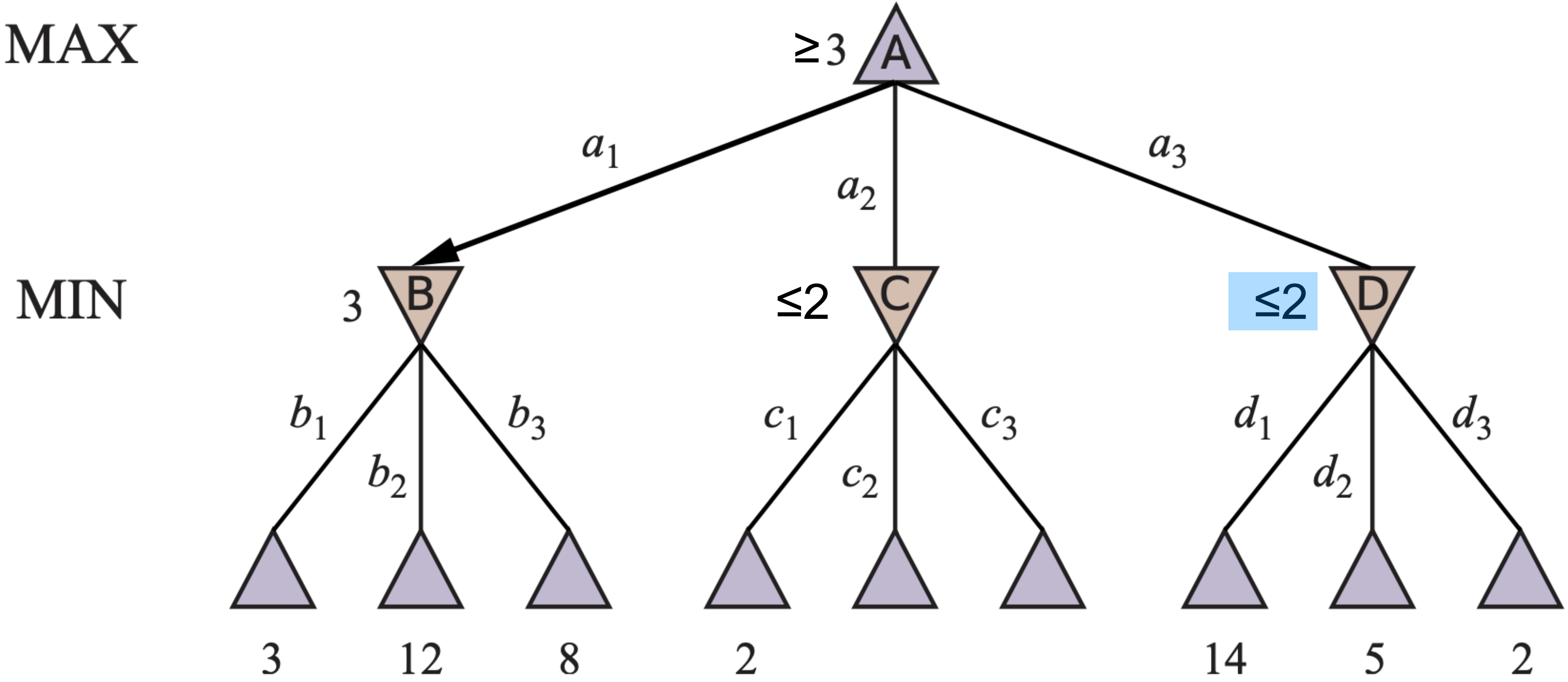
Pruning Technique?



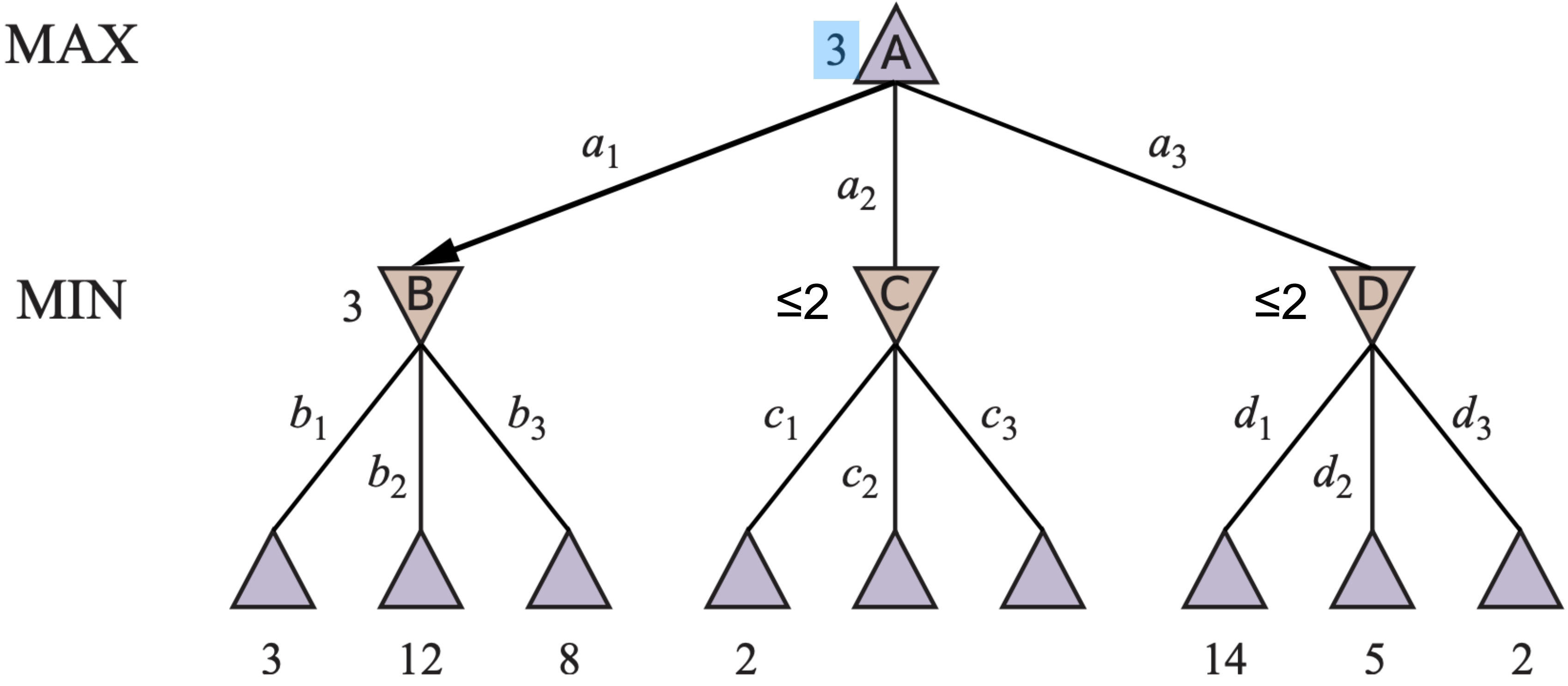
Pruning Technique?



Pruning Technique?



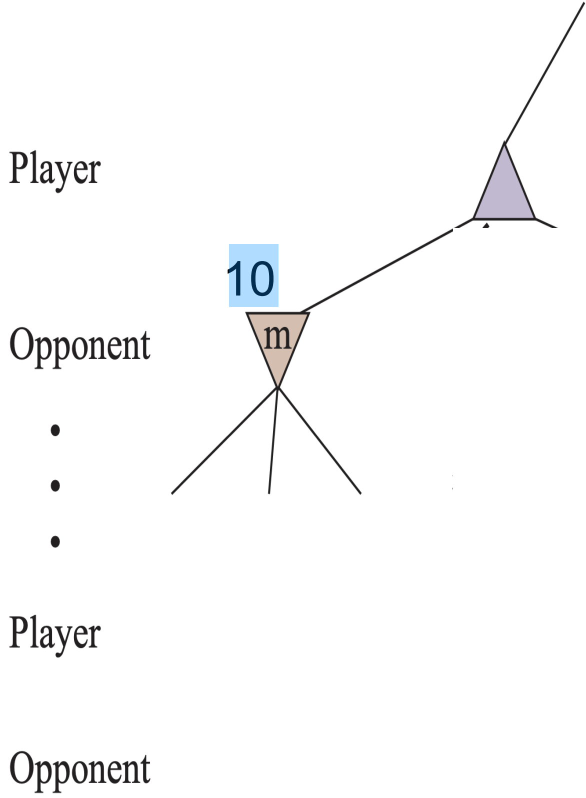
Pruning Technique?



Alpha-Beta Pruning

- α = the value of the best choice (i.e., highest-value) we have found so far at any choice point along the path for **MAX**
  (Think: α = “at least”)

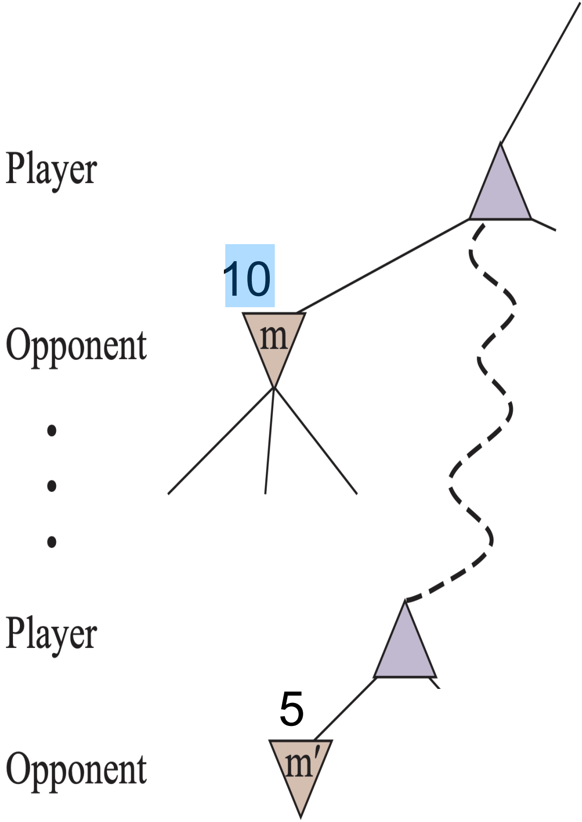
- β = the value of the best choice (i.e.,lowest-value) we have found so far at any choice point along the path for **MIN**
  (Think: β= “at most”)



Alpha-Beta Pruning

- α = the value of the best choice (i.e., highest-value) we have found so far at any choice point along the path for **MAX**
  (Think: α = “at least”)

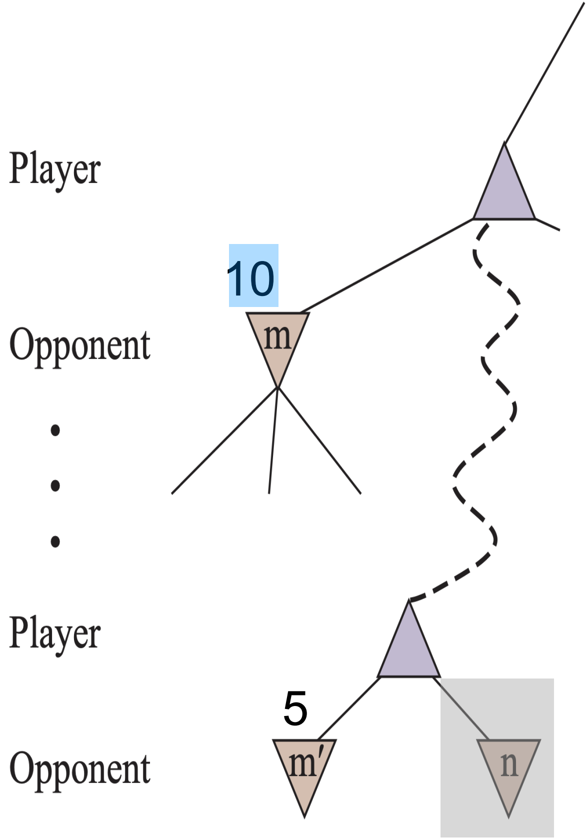
- β = the value of the best choice (i.e.,lowest-value) we have found so far at any choice point along the path for **MIN**
  (Think: β= “at most”)



Alpha-Beta Pruning

- α = the value of the best choice (i.e., highest-value) we have found so far at any choice point along the path for **MAX**
  (Think: α = "at least")

- β = the value of the best choice (i.e.,lowest-value) we have found so far at any choice point along the path for **MIN**
  (Think: β= "at most")

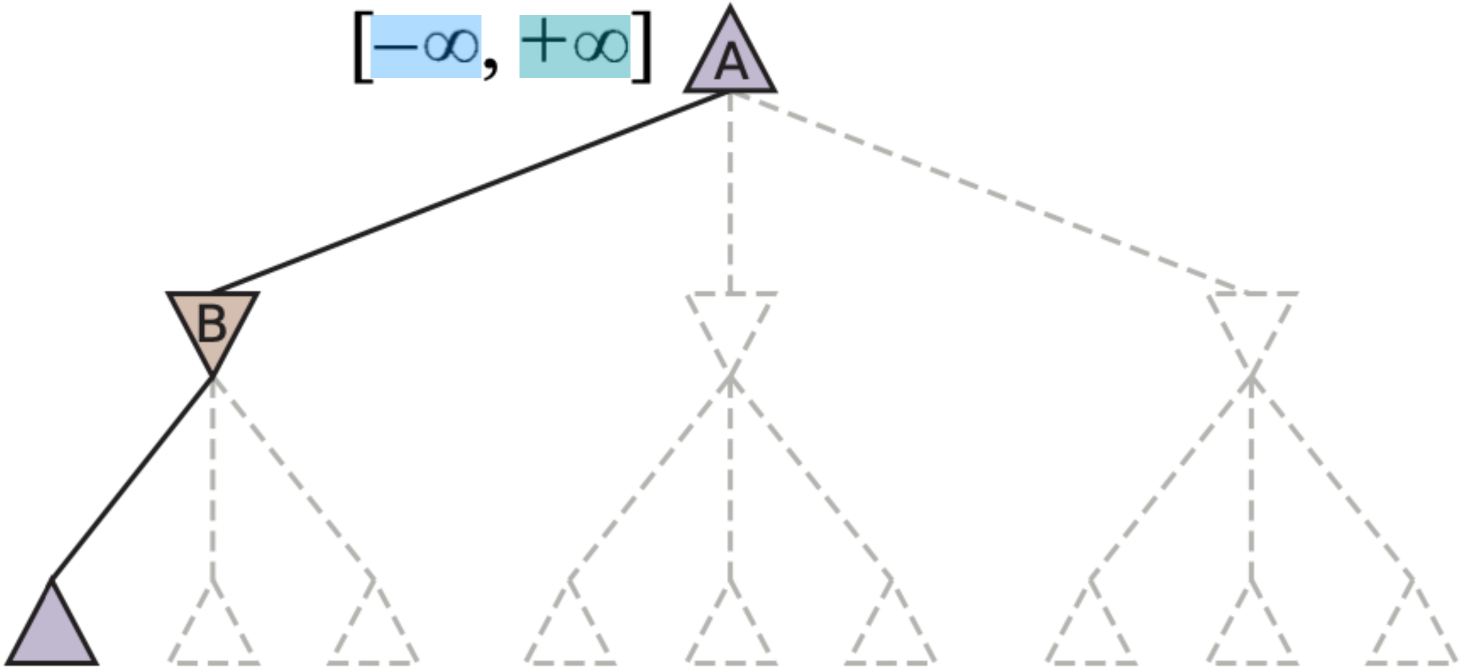


Example: Pruning

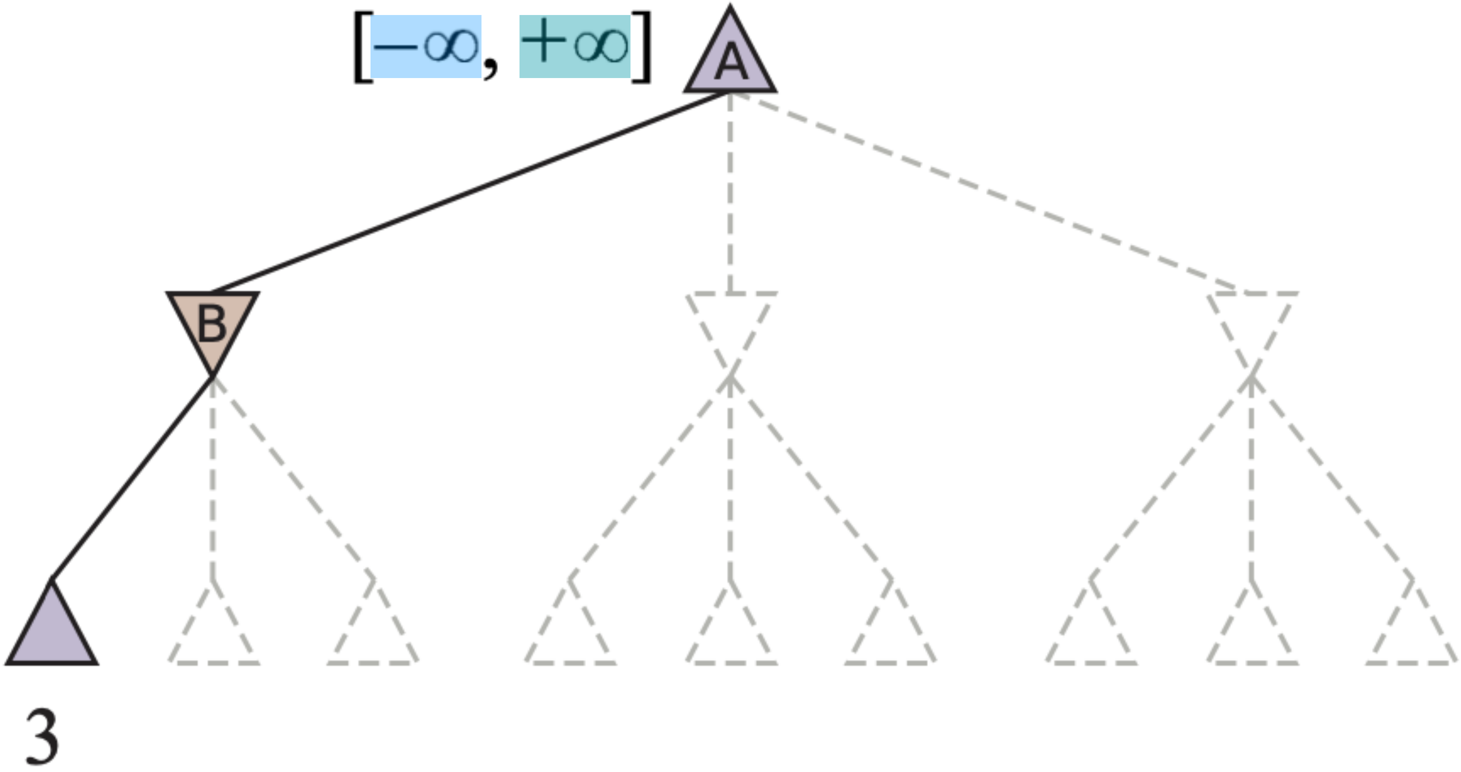
Example: Pruning

$[-\infty, +\infty]$

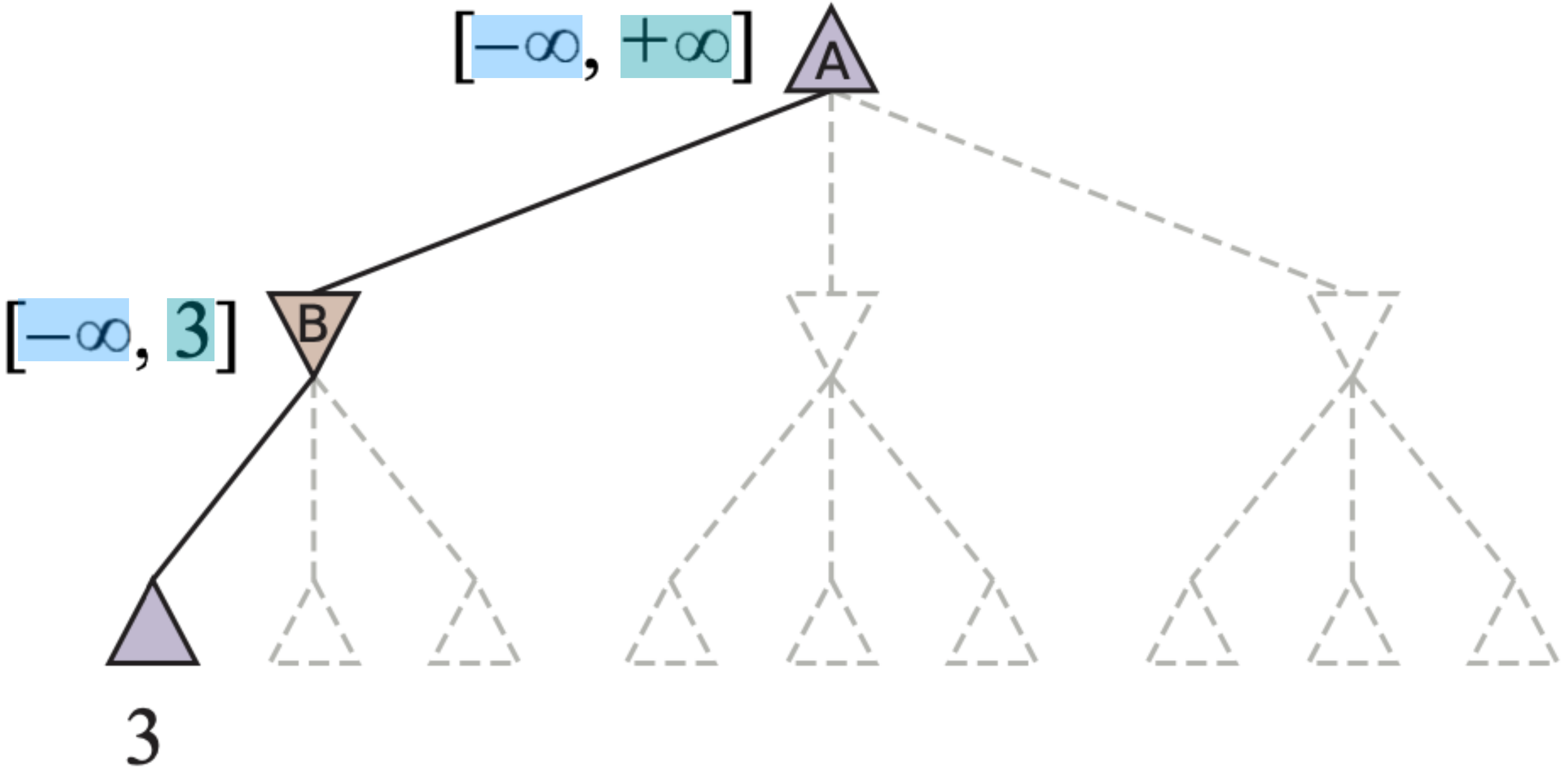
Example: Pruning



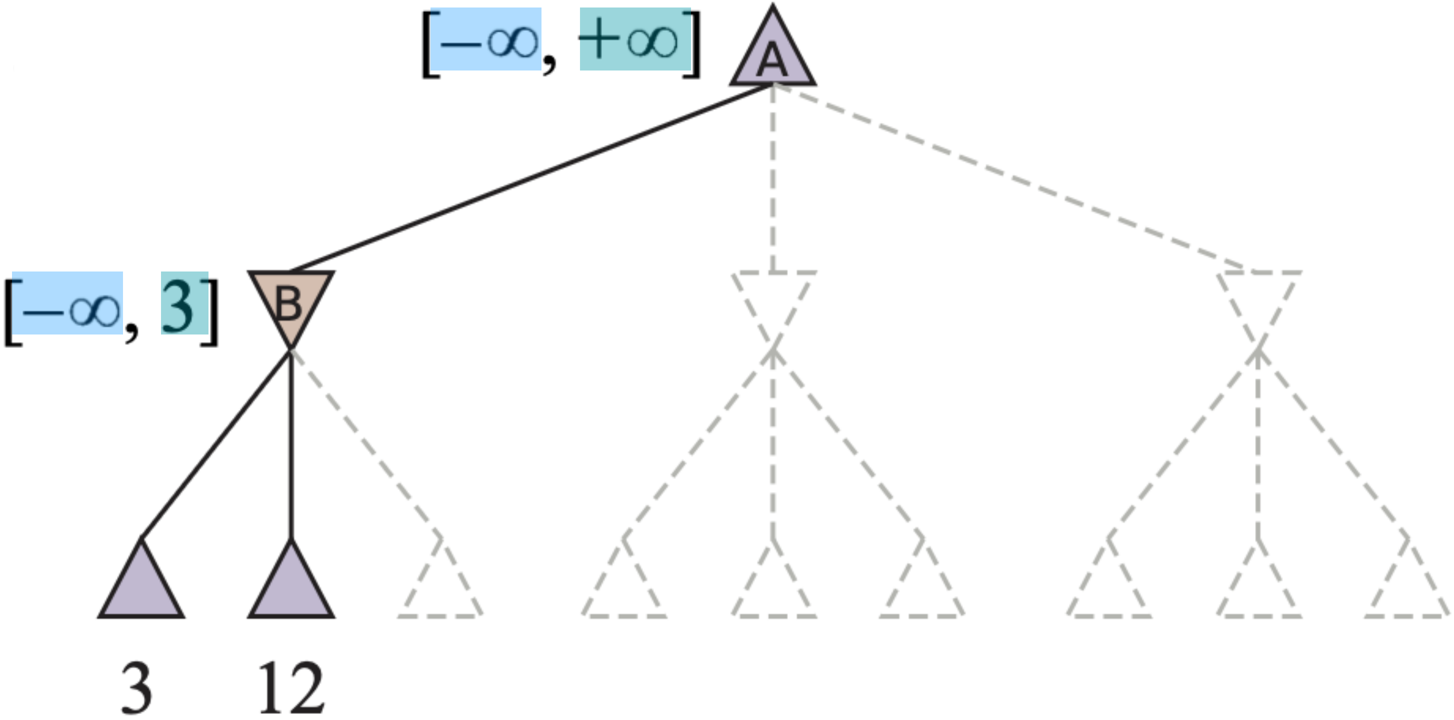
Example: Pruning



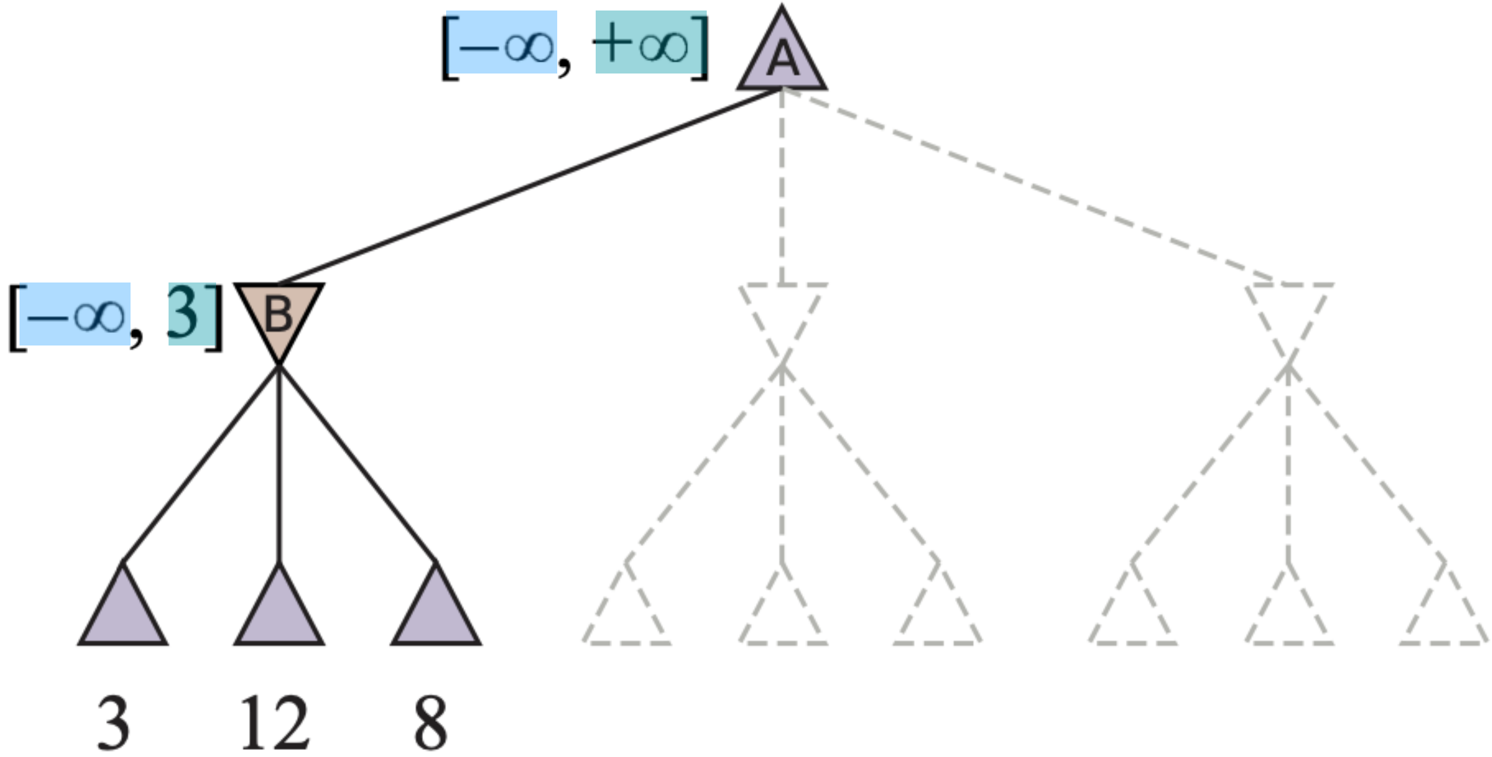
Example: Pruning



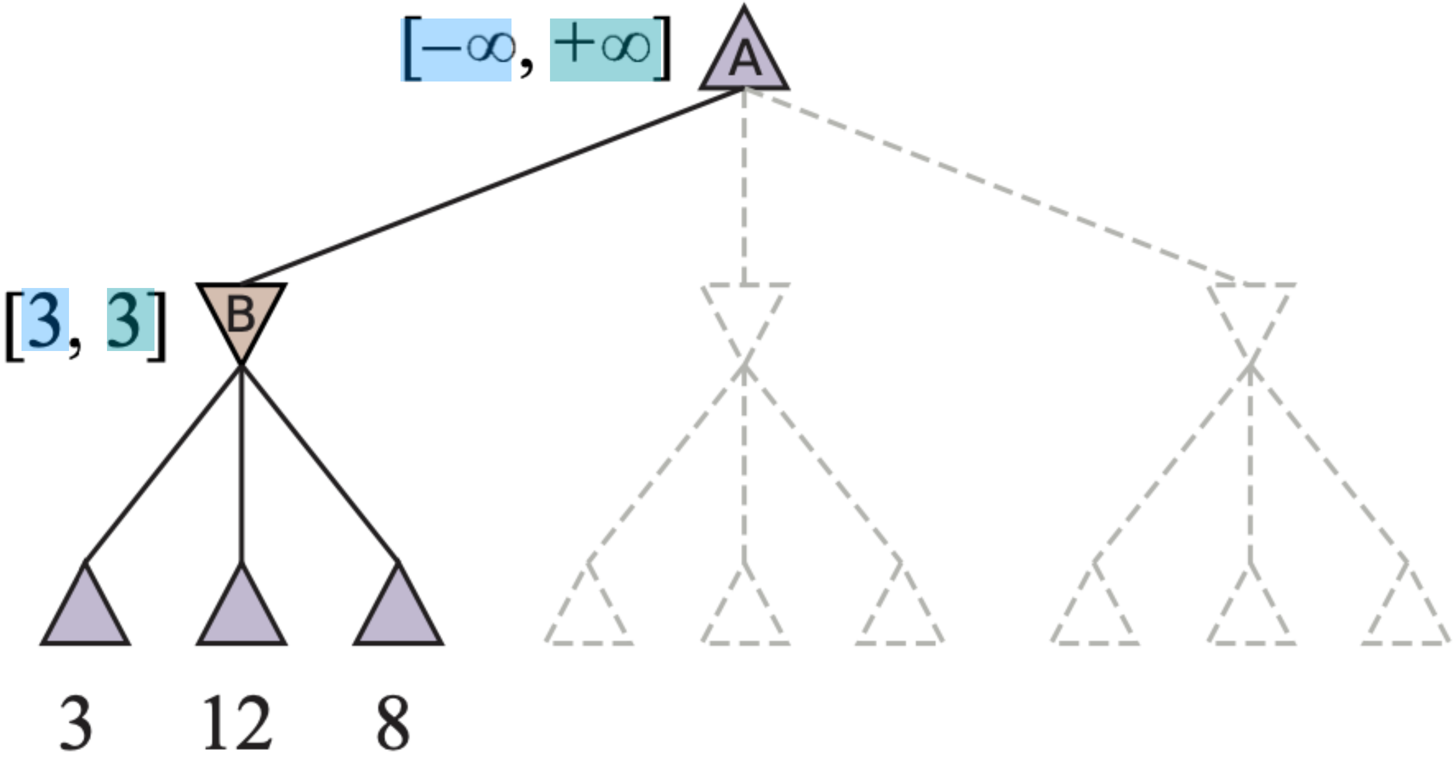
Example: Pruning



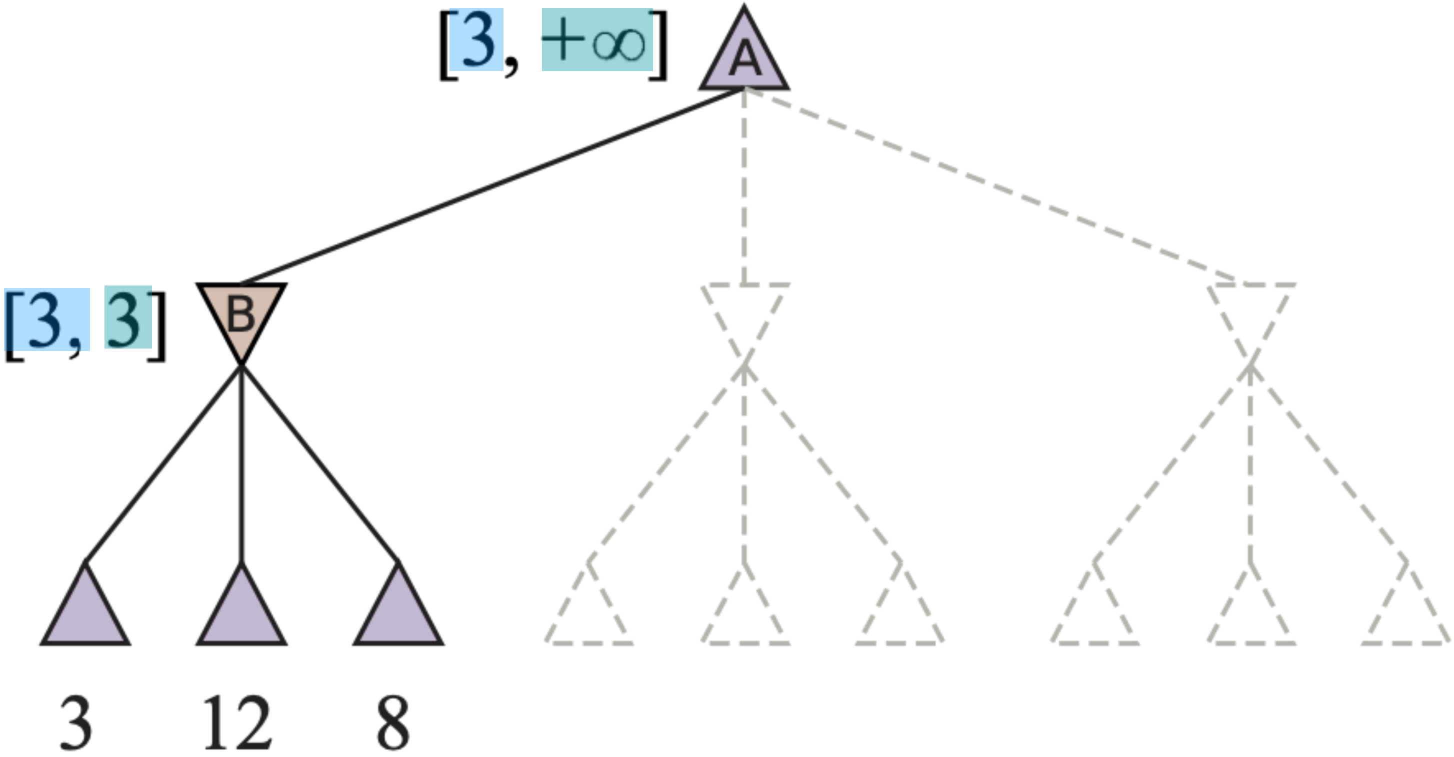
Example: Pruning



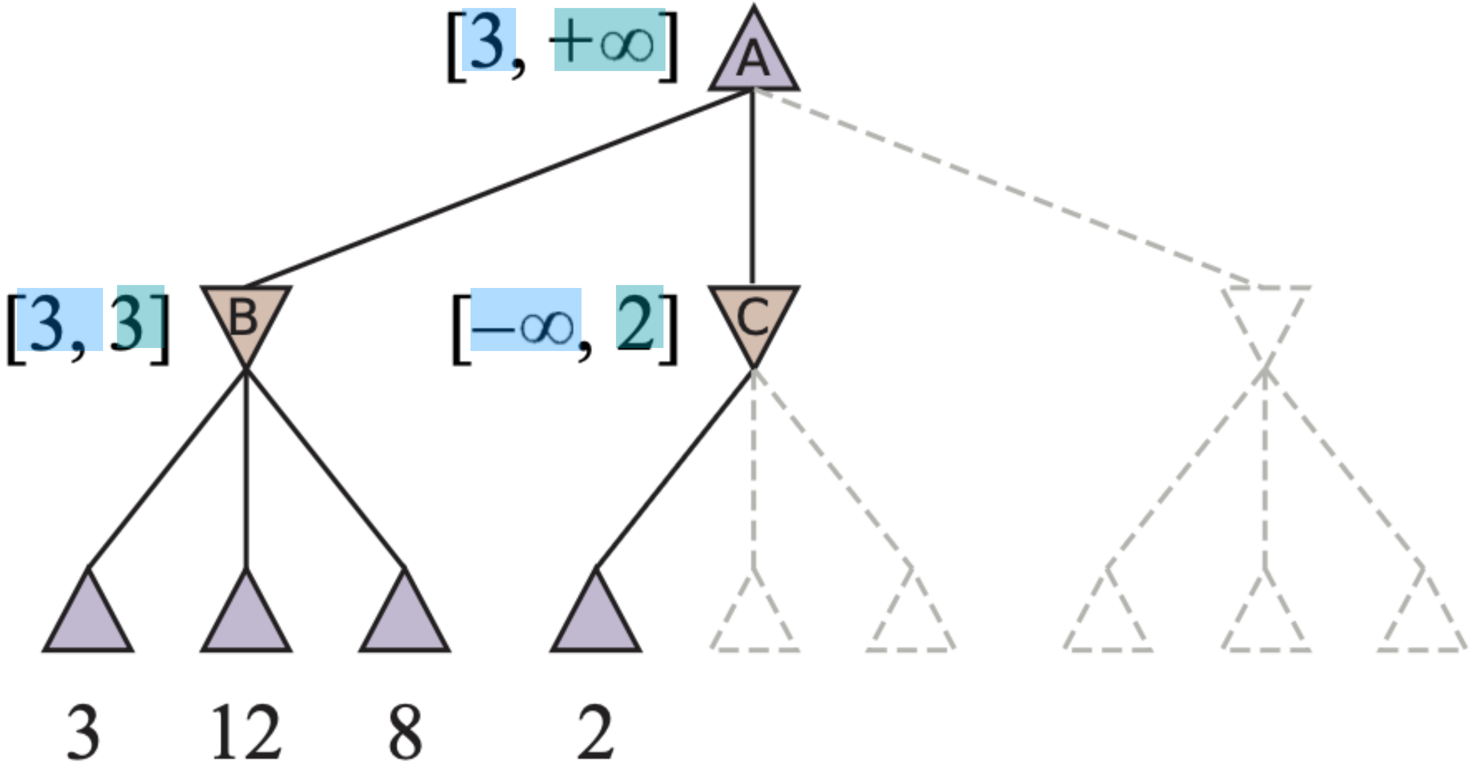
Example: Pruning



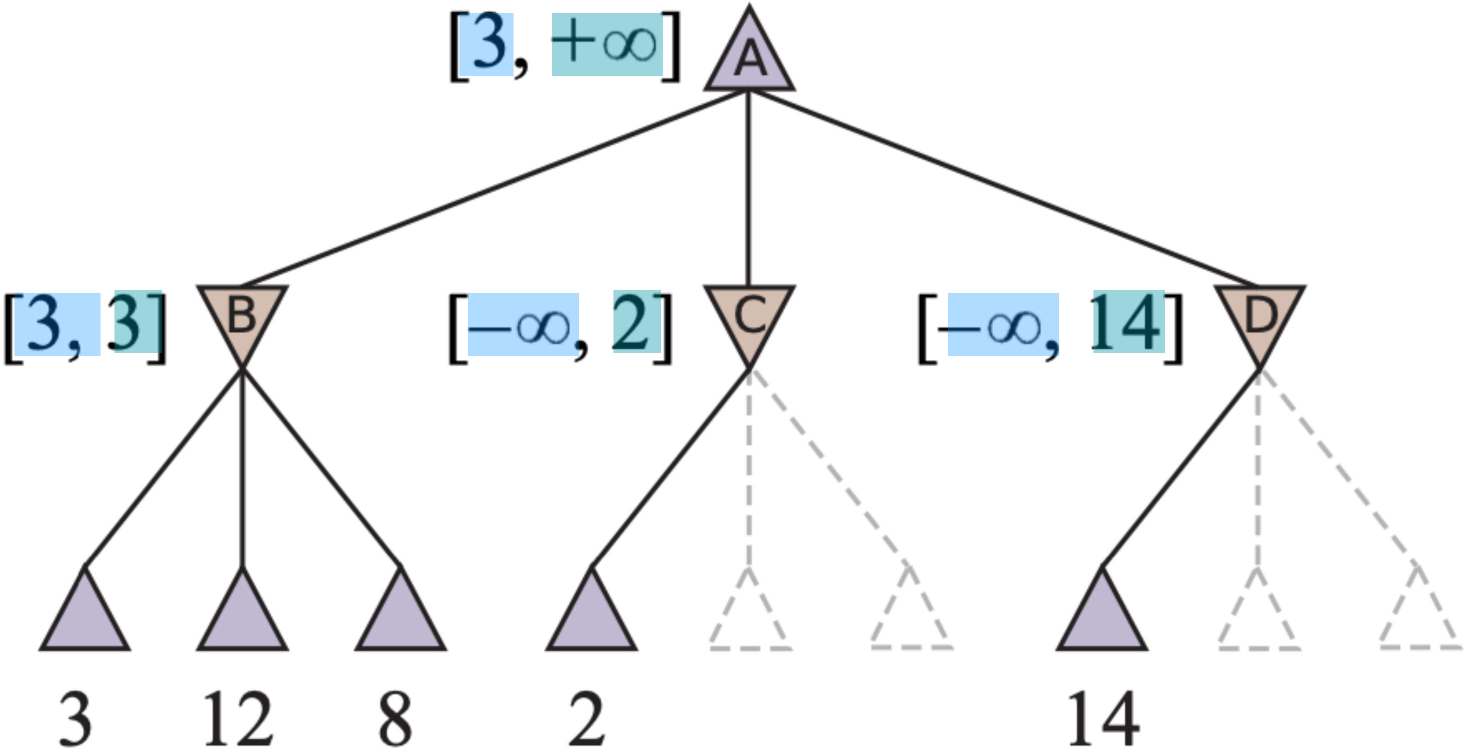
Example: Pruning



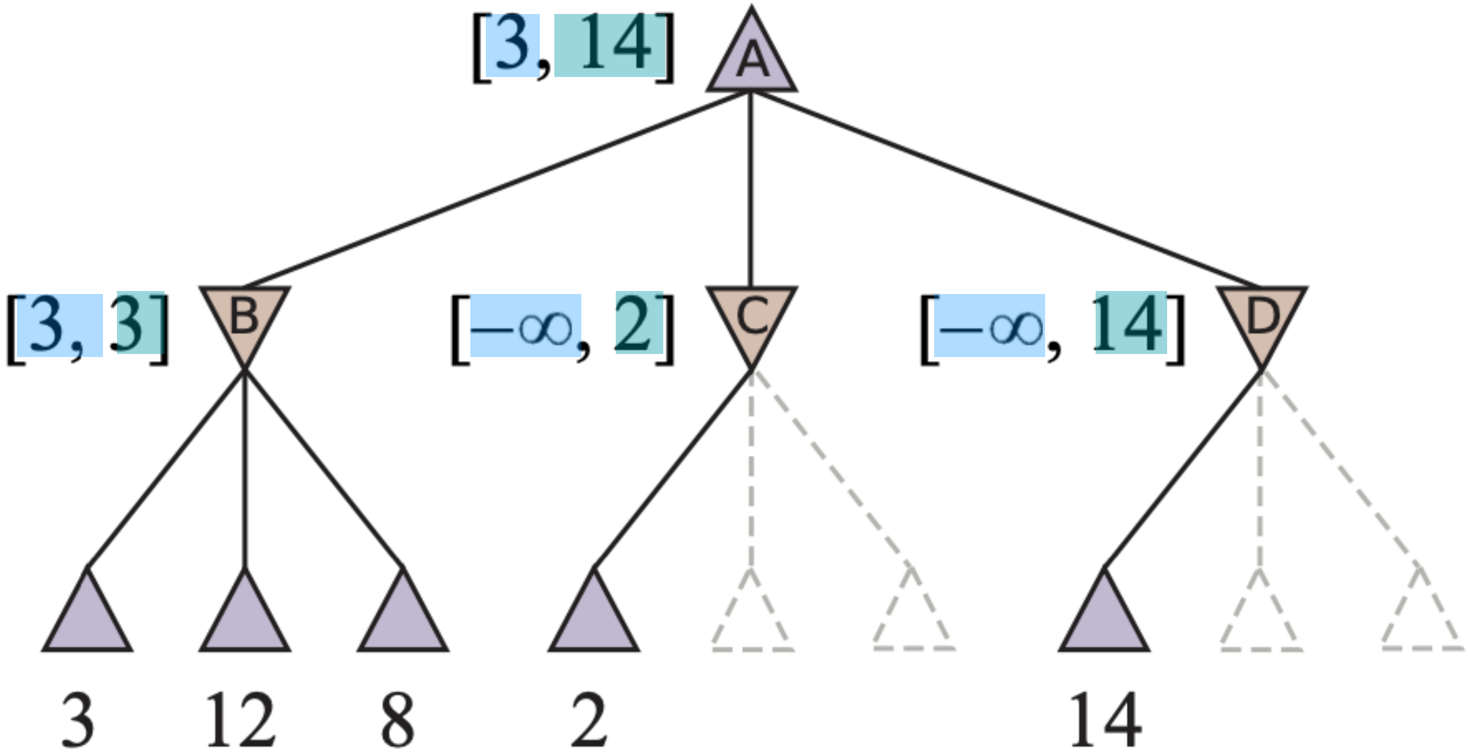
Example: Pruning



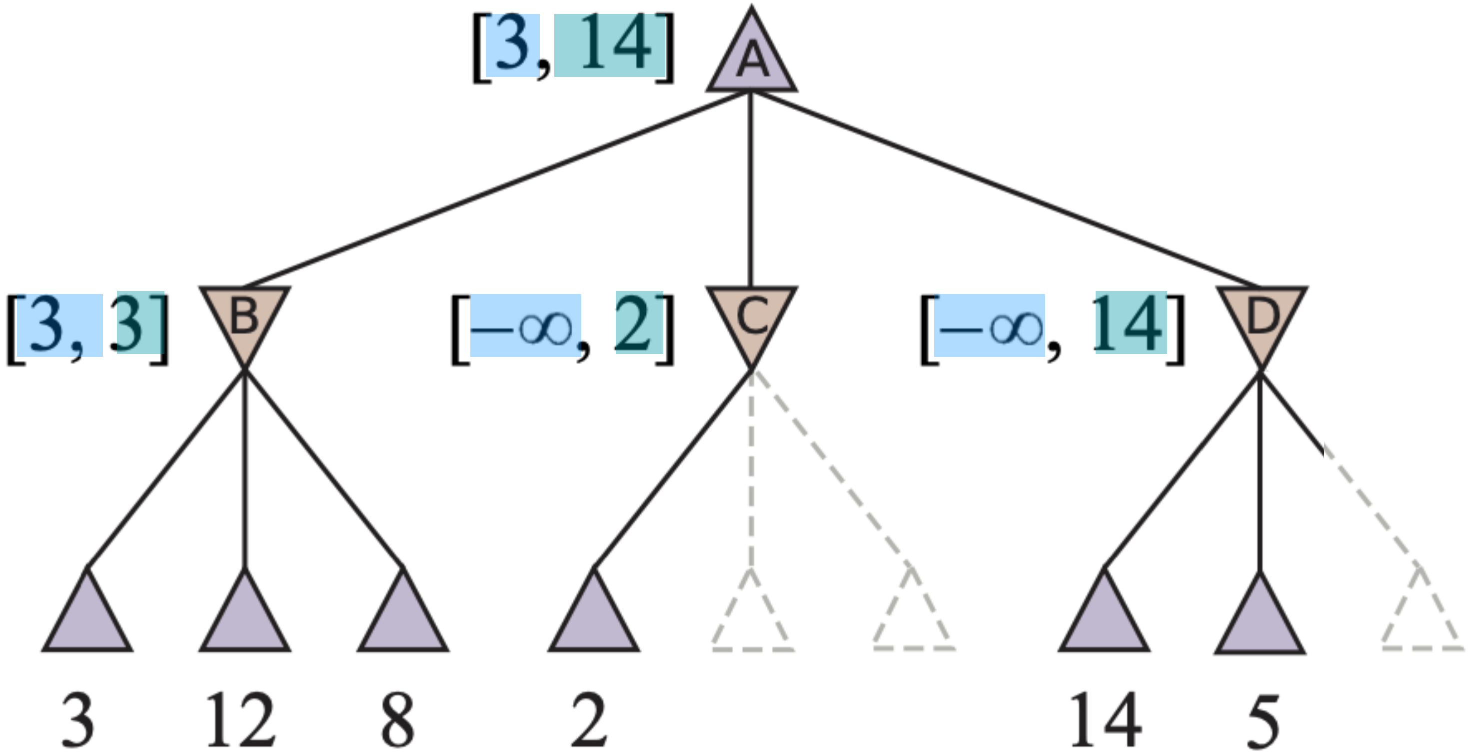
Example: Pruning



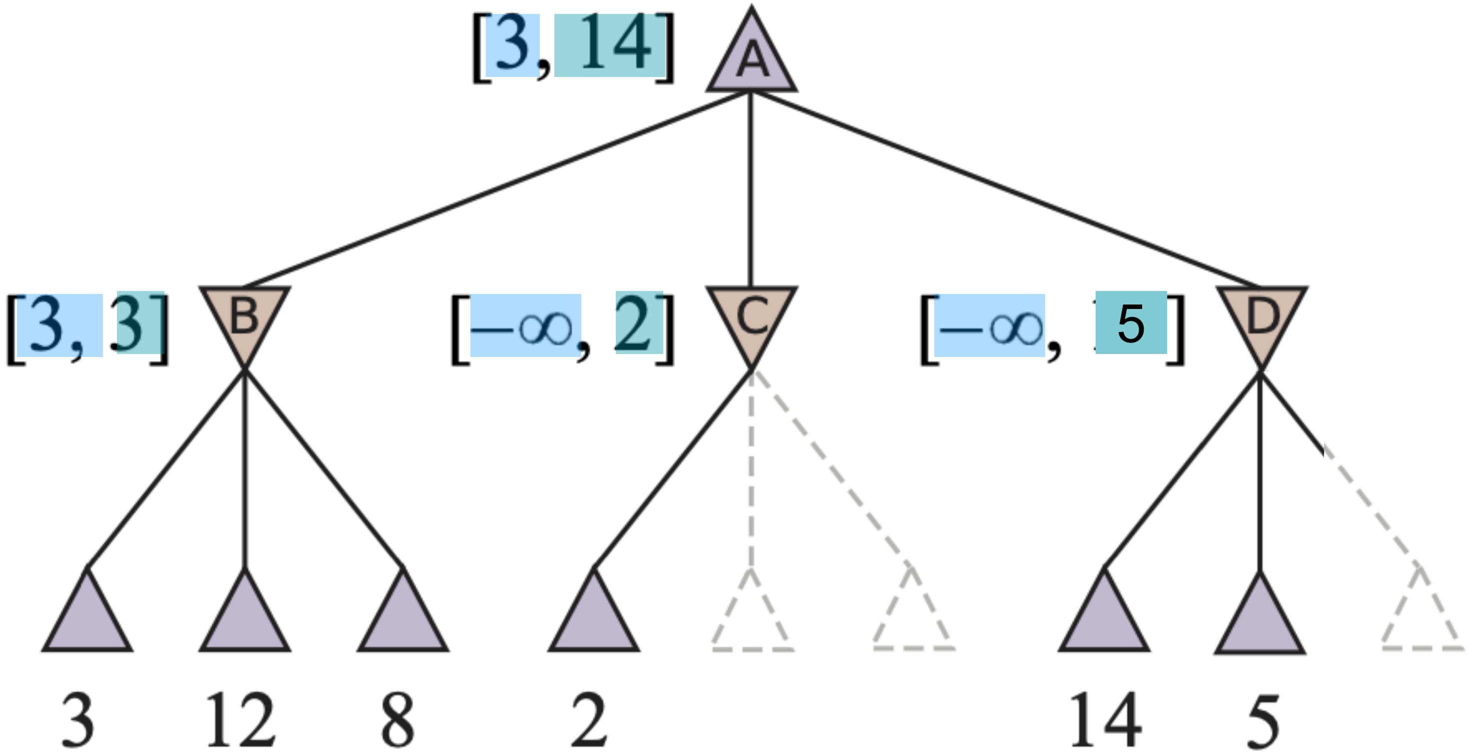
Example: Pruning



Example: Pruning

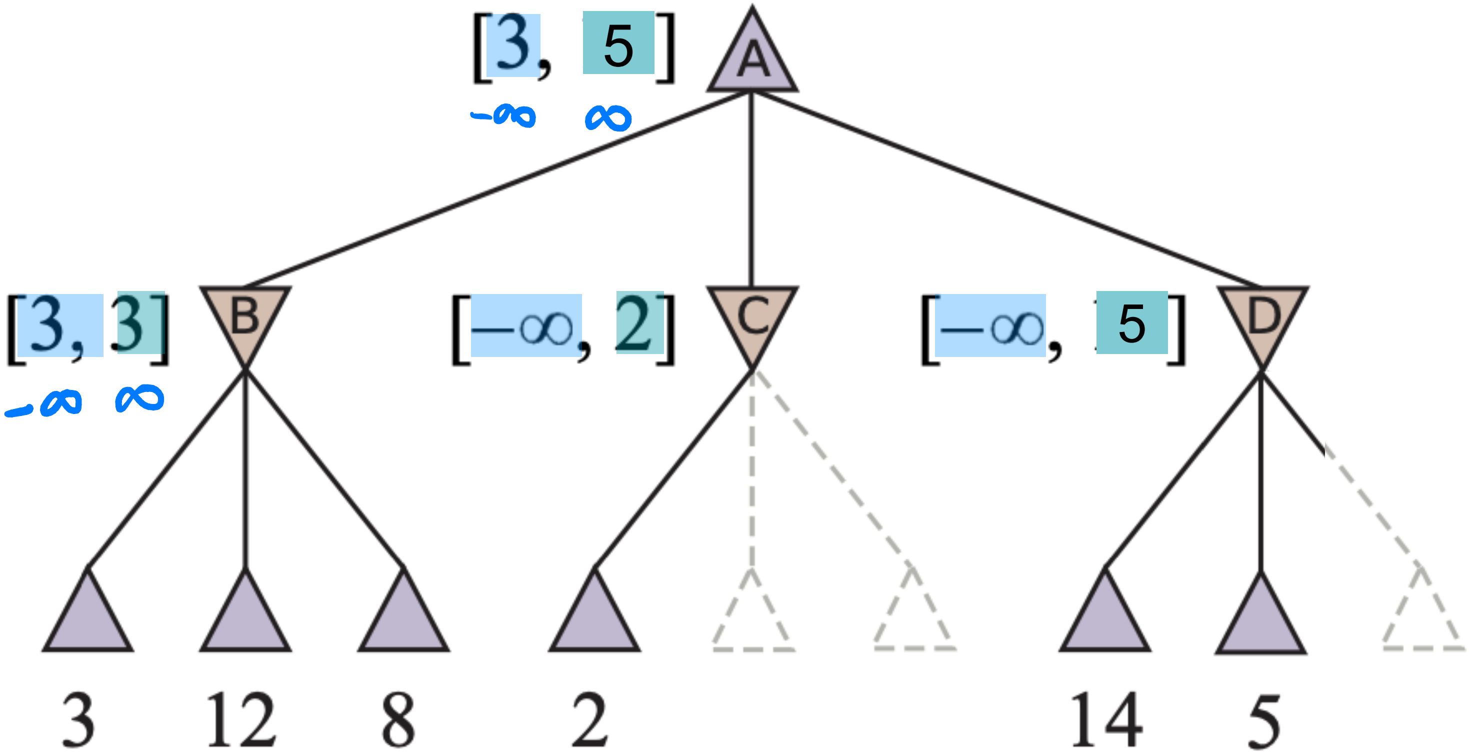


Example: Pruning

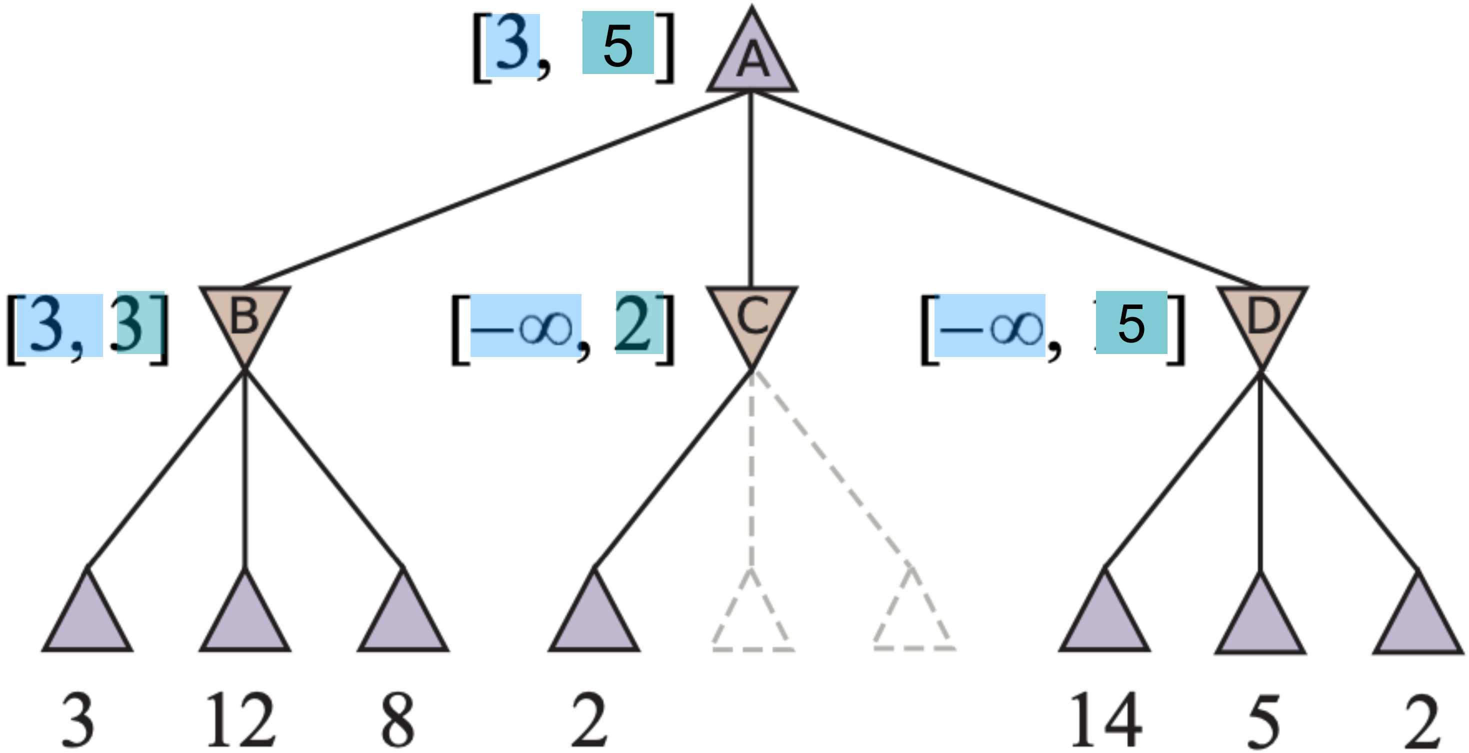


136

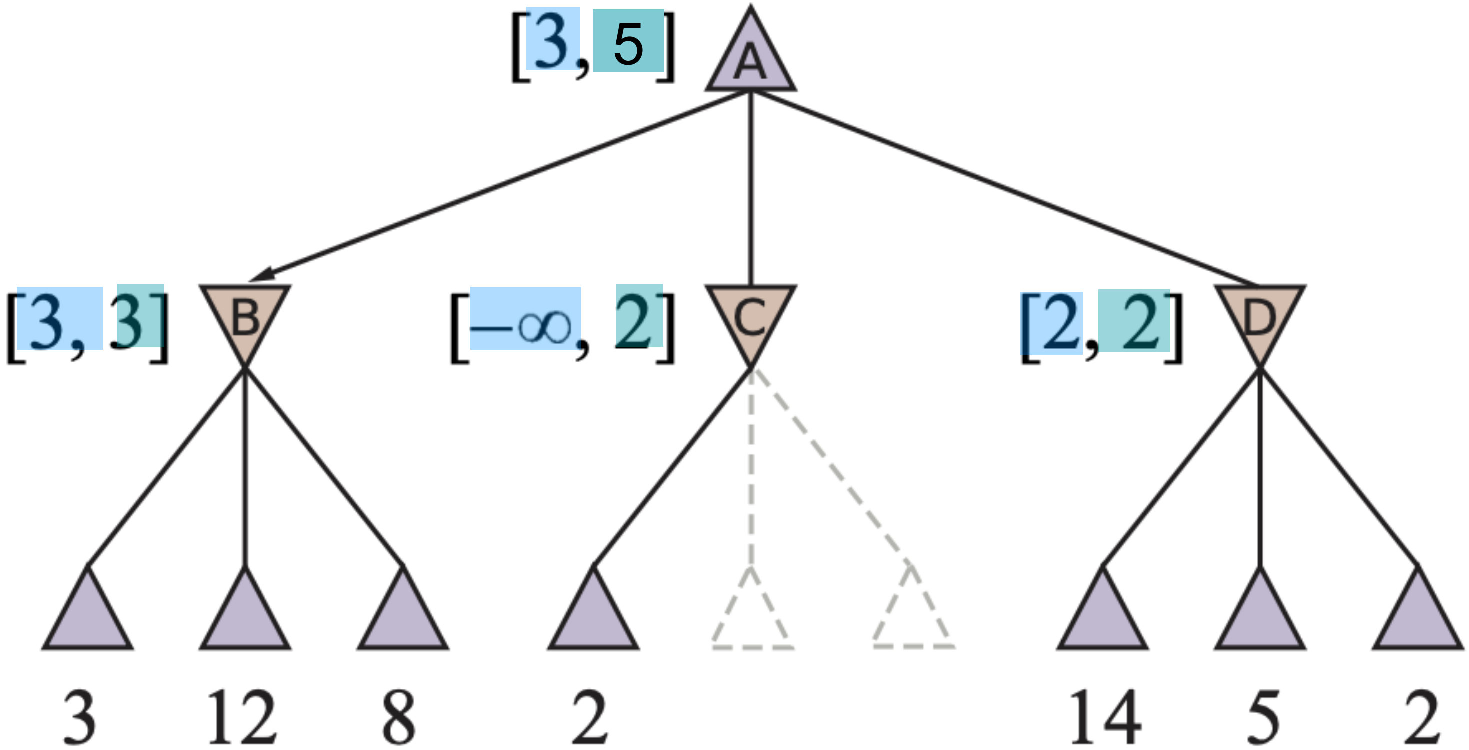
Example: Pruning



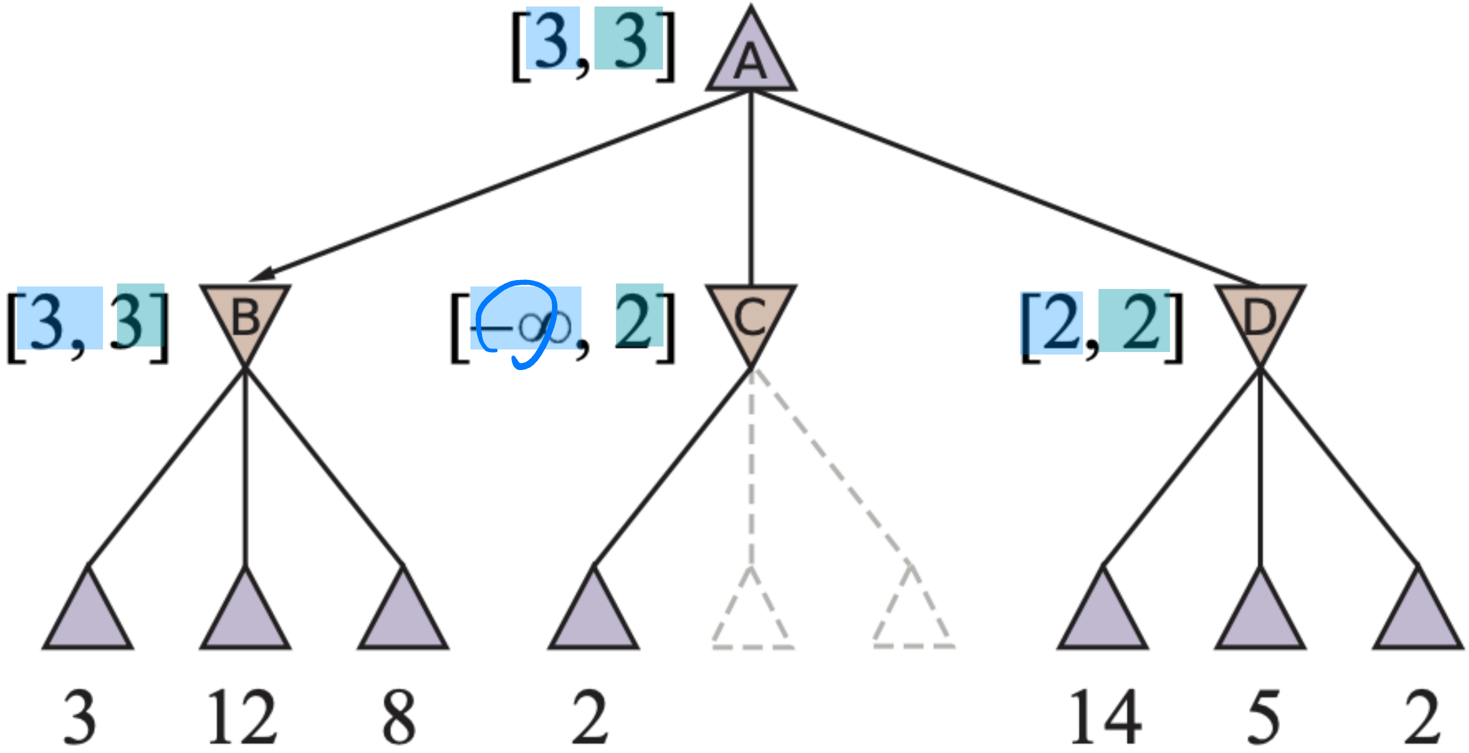
Example: Pruning



Example: Pruning



Example: Pruning



Alpha-Beta Search Algorithm

function ALPHA-BETA-SEARCH(*game*, *state*) **returns** an action
  player ← *game*.TO-MOVE(*state*)
  value, *move* ← MAX-VALUE(*game*, *state*, $-\infty, +\infty$)
  return *move*

Alpha-Beta Search Algorithm

function $\text{MAX-VALUE}(game, state, \alpha, \beta)$ **returns** a (*utility*, *move*) pair
  if *game*.$\text{IS-TERMINAL}(state)$ **then return** *game*.$\text{UTILITY}(state, player)$, *null*
  $v \leftarrow -\infty$
  for each a **in** *game*.$\text{ACTIONS}(state)$ **do**
    $v2, a2 \leftarrow \text{MIN-VALUE}(game, game.\text{RESULT}(state, a), \alpha, \beta)$
    if $v2 > v$ **then**
      $v, move \leftarrow v2, a$
      $\alpha \leftarrow \text{MAX}(\alpha, v)$
    if $v \geq \beta$ **then return** v, *move*
  return v, *move*

Alpha-Beta Search Algorithm

function MIN-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    $v2$, $a2 \leftarrow$ MAX-VALUE(*game*, *game*.RESULT(*state*, *a*), α, β)
    if $v2 < v$ **then**
      v, *move* $\leftarrow v2$, a
      $\beta \leftarrow \text{MIN}(\beta, v)$
    if $v \leq \alpha$ **then return** v, *move*
  return v, *move*

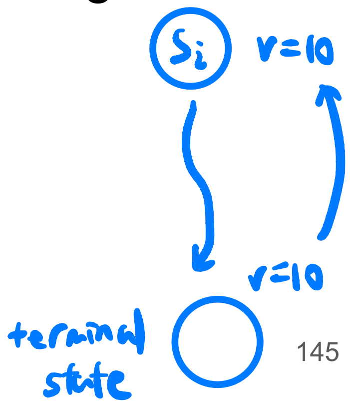
Weaknesses for Games

- Large branching factor
- Difficult to define a heuristic function

Monte Carlo Tree Search (MCTS)

- Idea: It estimates **the average utility** over a number of **simulations** of complete games starting from the state
  - A simulation (also called a playout or rollout) chooses moves first for one player, then for the other, repeating until a terminal position is reached
  - For games, “average utility” is the same as “win percentage”

```
Rollout(S_i):
    loop forever:
        if S_i is a terminal state
            return value(S_i)
        A_i = random(available actions(S_i))
        S_i = simulate(A_i, S_i)
```



Simulation Policy

- From what positions do we start the simulations?
- How many simulations do we allocate to each position?

Pure Monte Carlo Search

- Do *N* simulations starting from the current state of the game
- Track which of the possible moves from the current position has the highest win percentage
- Issue
  - Optimal play
    - Increase *N* → Computational resources ↑

Selection Policy

- Focus on the important parts of the game tree and balance two factors
  - **Exploration** of states that had few simulations
  - **Exploitation** of states that have done well in past simulations to get a **more accurate estimate** of their value

Monte Carlo Tree Search

Maintain a search tree and grow it on each iteration of the following four steps:

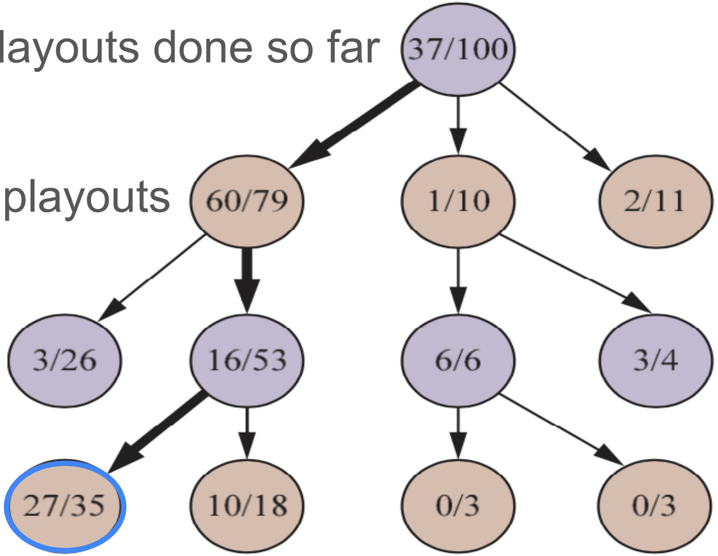
- Selection
- Expansion
- Simulation
- Back-propagation

Step 1: Selection

exploitation

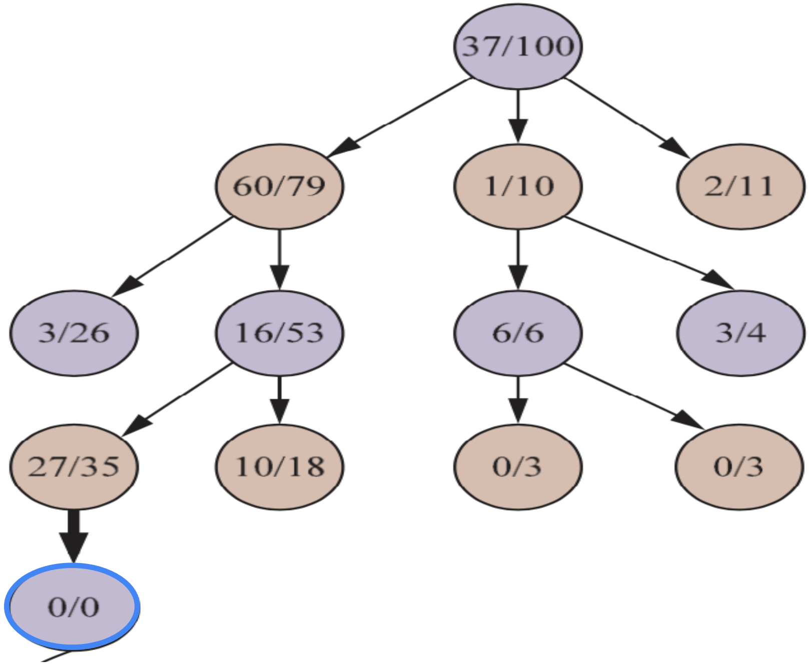
- Starting at the root node of the search tree, we choose a move (e.g., best win percentage), leading to a successor node, and repeat that process, moving down the tree to a leaf

  - White: won 37 out of 100 playouts done so far
  - Black: won 60/79 playouts



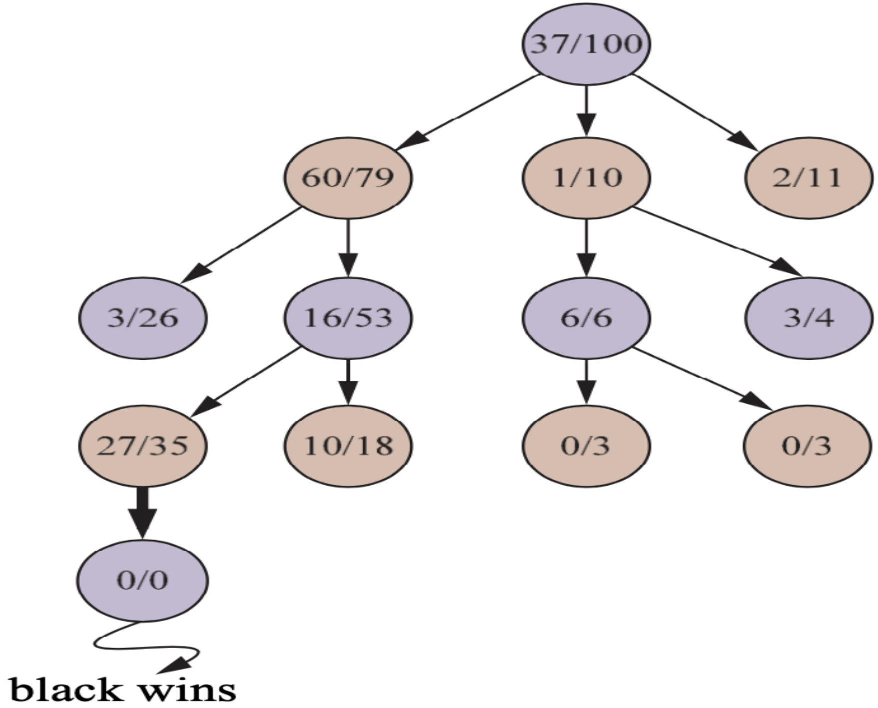
Step 2: Expansion

- We grow the search tree by generating a new child of the selected node



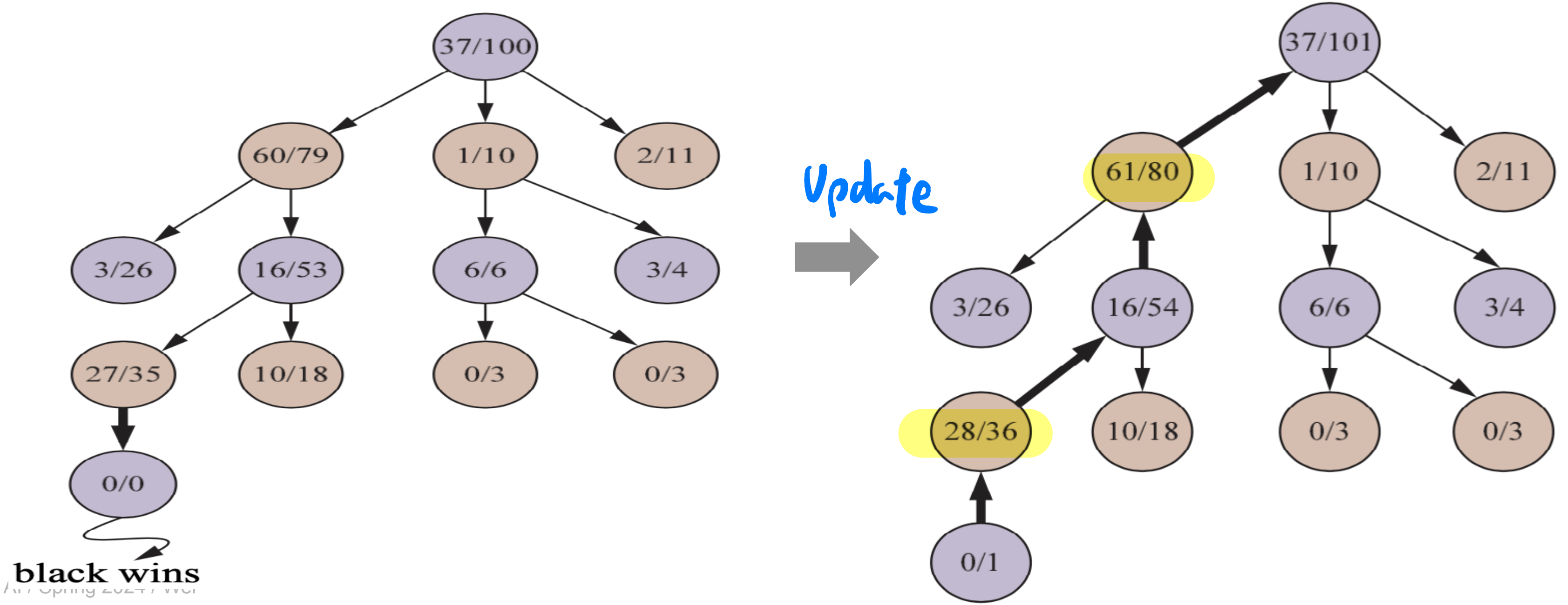
Step 3: Simulation

- We perform a playout from the newly generated child node



Step 4: Back-propagation

- We now use the result of the simulation to update all search tree nodes going up to the root



Upper Confidence Bounds Applied to Trees (UCT)

- The effective selection policy ranks each possible move based on an upper confidence bound formula UCB1 as below

average win rate

$$UCB1(n) = \underbrace{\frac{U(n)}{N(n)}}_{\text{Exploitation}} + C \times \underbrace{\sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}}_{\text{Exploration}}$$

how many times have the parent node been visited?

Exploitation　　Exploration

  - $U(n)$: the total utility of all playouts that went through node n
  - $N(n)$: the number of playouts through node n how many times has this node been tried?
  - Partent(n): the parent node of in the tree
  - C: a constant that balances exploitation and exploration

UCT MCTS Algorithm

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

- UCB1 formula ensures that the node with most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up

$\because$ exploration term $\to 0$ when $n \to \infty$

Course Topics

- Intelligent agents (AIMA Ch. 2)
- Search (AIMA Ch. 3, 4, 6, 5)
- Reasoning (AIMA Ch. 7-9, 12-15)
- Machine learning (AIMA Ch. 19-20)
- Deep learning (AIMA Ch. 22)
- Natural language processing (AIMA Ch. 24)
- Generative AI (Optional)