CSC 384
Introduction to Artificial Intelligence

Game Tree Search

Alice Gao and Randy Hickey

Winter 2023

Outline

1. Types of Games
2. Minimax Search
3. Alpha-Beta Pruning
4. Extensions to Alpha-Beta

TYPES OF GAMES

Learning Outcomes

By the end of this section, you should be able to

- Determine whether a game has a property.
- Explain how each property has implications on the complexity of analyzing a game.

Games

Games are the oldest, most well-studied domain in AI.

- Fun
- Easy to represent. Clear rules.
- State spaces can be very large
  - Search tree for chess has $\sim 10^{154}$ nodes.
- Decisions must be made in real-time.
- Easy to determine when a program is doing well.

Properties of Games

- Single-Player v.s. Multiple-Player
- Zero-Sum v.s. Non-Zero-Sum
- Deterministic v.s. Stochastic
- Perfect Information v.s. Imperfect Information

Single v.s. Multiple Player

Single Player

- Your opponent is the game...

Multiple Player

- Other players may be
  - adversarial, or
  - cooperative

Zero-Sum or not

Zero (Constant) – Sum

- Total payoff to all players is constant.
- You win what the other players lose.

Non–Zero (Constant) Sum

Deterministic v.s. Stochastic

Deterministic

- Change in state is fully controlled by the players.
- No random elements.

Stochastic

- Change in state is partially determined by chance.

Perfect v.s. Imperfect Information

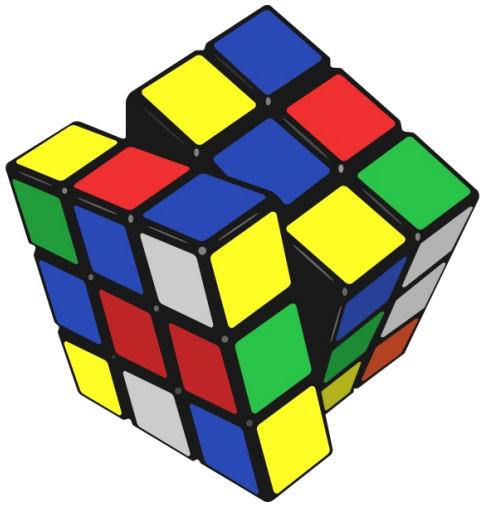
Perfect information

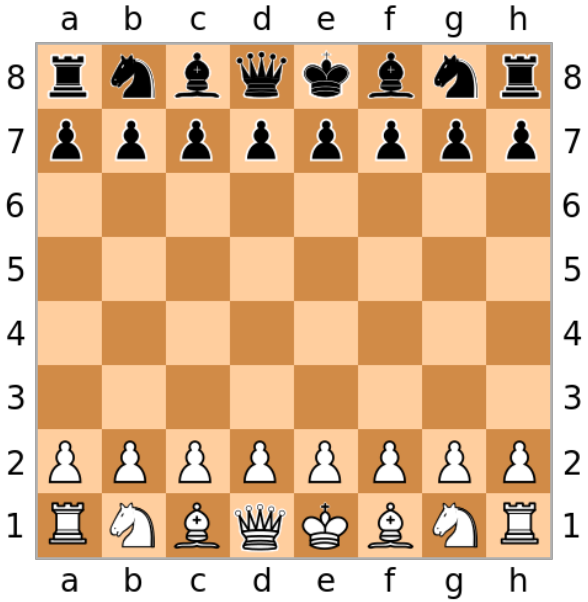
- State is fully observable.

Imperfect information

- Part of state is hidden.

Let's categorize some games





We will study one type of game

- Two-player
  - Must model the other player's goals and strategies.
- Zero-sum
  - One player's gain = the other player's loss
  - It suffices to model one player only.
- Deterministic
  - No need to model random elements. No probabilities.
- Perfect-Information
  - No need to model hidden information.

MINIMAX SEARCH

Learning Outcomes

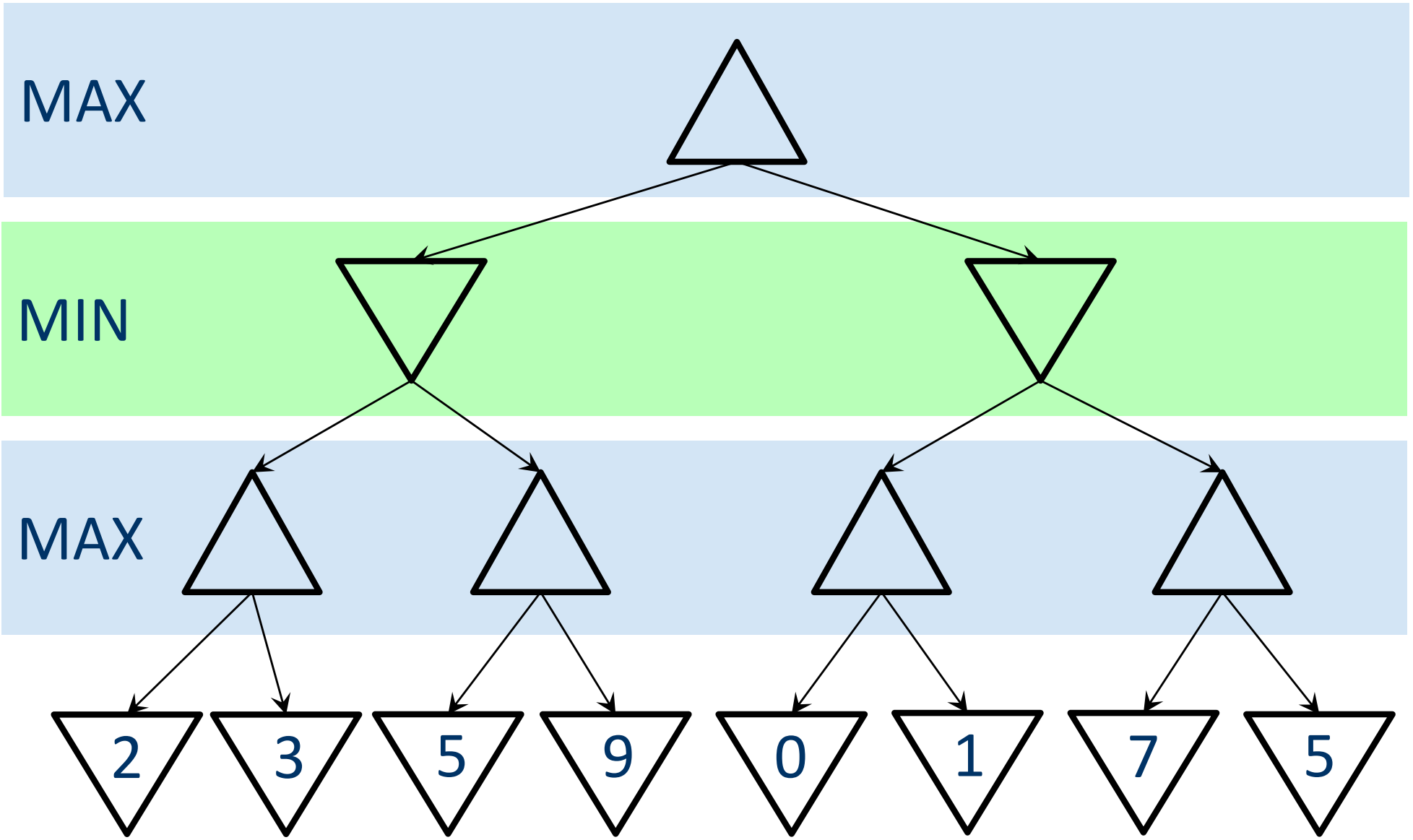
By the end of this section, you should be able to

- Explain the difference between a solution for a standard search and a strategy for an adversarial game.
- Explain the conditions under which the minimax strategy is optimal.
- Execute Minimax Search on a game tree and determine the minimax value of the root node.
- Compare and contrast Depth-First Search and Depth-First Minimax Search.

Two-Player Zero-Sum Game

- Two players: MAX and MIN
- Initial state s_0
- $player(s)$:
  - returns the player who moves in state s.
- $actions(s)$:
  - returns the legal moves in state s.
- $result(s, a)$:
  - returns the next state after taking action a in state s.
- $terminal(s)$:
  - returns True iff s is a terminal state.
- $utility(s)$:
  - returns MAX's payoff in terminal state s.

Game Tree Example



Player's Strategy

- In standard search, a solution is a sequence of moves leading to a goal state.
- In a game, the strategy (for MAX) specifies
  - a move for the initial state.
  - a move for all possible states arising from MIN's response.
  - all possible responses to all of MIN's responses to MAX's previous moves.
- In short, a strategy specifies what move to make at every contingency.

Minimax Strategy

An optimal strategy leads to outcomes

at least as good as any other strategy.

The minimax strategy is optimal

assuming that the opponent is playing optimally.

If the opponent isn't playing optimally, we can have a better strategy that exploits the opponent's weaknesses.

Minimax Value

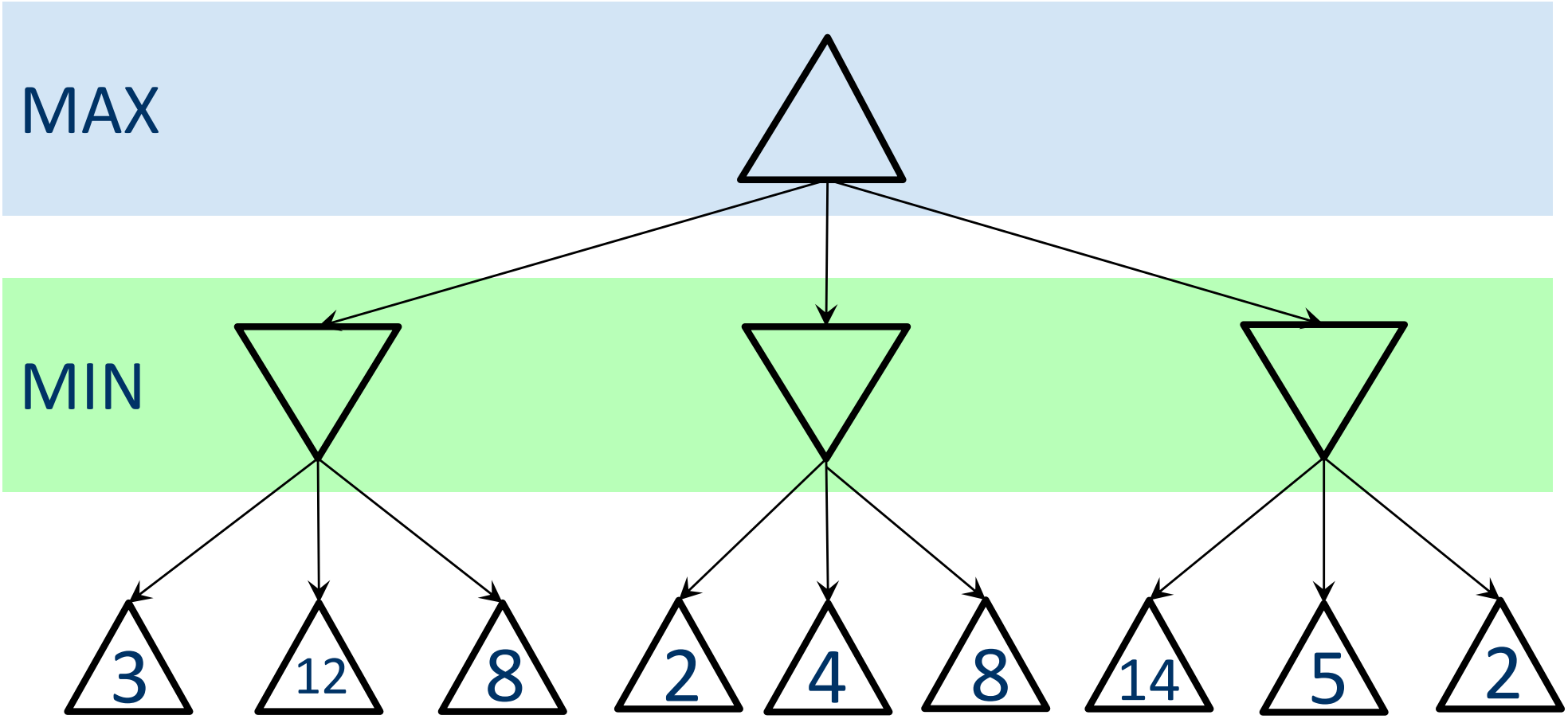
minimax-value(s) =

- utility(s), if s is a terminal state,
- max_{s' in succ(s)} minimax-value(s') if s is a MAX node.
- min_{s' in succ(s)} minimax-value(s') if s is a MIN node.

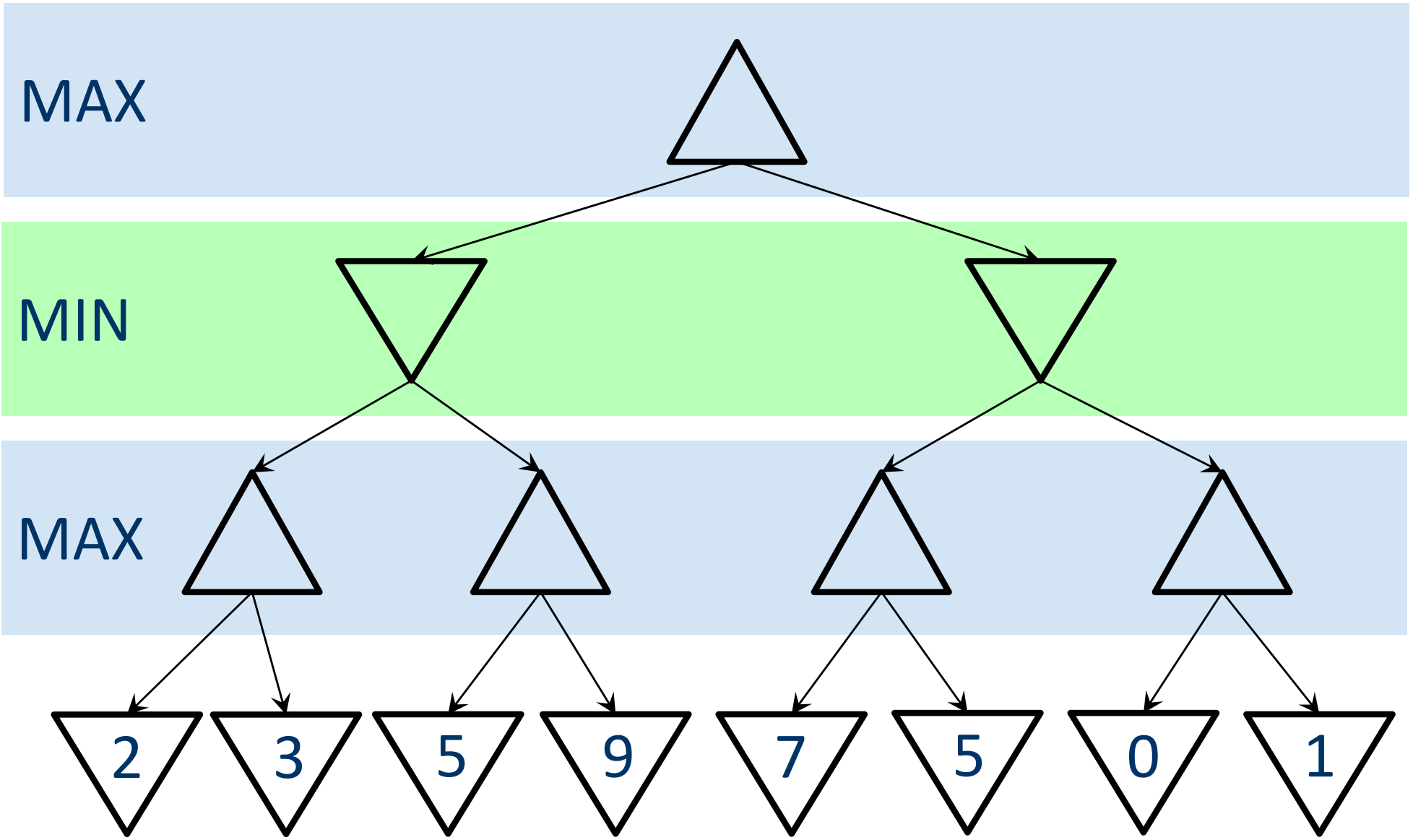
How can we determine the minimax value of the root?

- Start from the terminal states.
- Propagate the values up the tree.
  - As a MAX node, take max of its children's values.
  - At a MIN node, take min of its children's values.

Example 1: Compute Minimax Values



Example 2: Compute Minimax Values



DEPTH-FIRST MINIMAX

Why Depth-First Minimax?

- So far, computing the minimax value requires
  - Building the entire game tree
  - Backing up values
- Save space by using a depth-first version of minimax.
  - Avoids representing the exponentially sized game tree
  - Allows us to prune some states using alpha-beta algorithm.

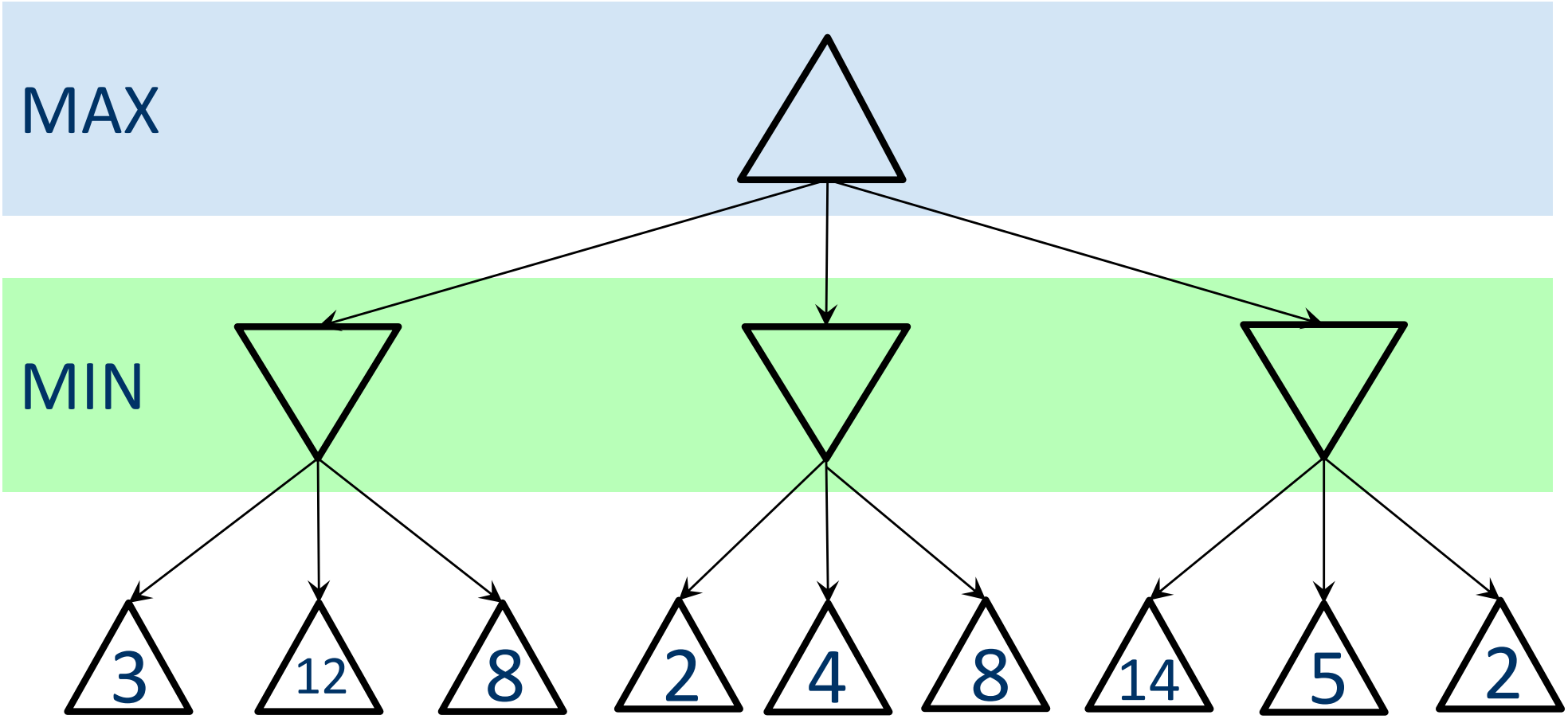
Depth-First Minimax Pseudocode

```
function MINIMAX-DECISION(state) returns an action
    return argmax_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Q1: Depth-First Minimax



Time and Space Complexity

- m is the max depth of the tree.
- b is the branching factor.

- Space complexity

$$O(bm)$$

- Time complexity is

$$O(b^m)$$

- Must traverse entire search tree to evaluate all moves.

Compare and Contrast

Depth-First Search

Depth-First Minimax

ALPHA-BETA PRUNING

Learning Outcomes

By the end of this section, you should be able to

- Explain the roles of alpha and beta in the alpha-beta pruning algorithm.
- Explain the reasoning behind pruning at a MAX/MIN node.
- Execute Alpha-Beta Pruning on a game tree, indicate all the pruned branches, and determine the minimax value of the root node.
- Explain the correctness and space complexity of alpha-beta pruning.

Problem with Minimax

- Must visit the entire search tree.
- # nodes visited is exponential in the depth of the tree.
- Cannot eliminate the exponent, but can cut it in half.
- Can compute the minimax value without looking at every node in the tree.
  - Prune branches that do not influence our computation.

Values in Alpha-Beta Pruning

- v = the current value for a node.

- α = value of best (highest-value) choice so far for MAX.
  - MAX guarantees that the minimax value $\geq \alpha$
  - α is a lower bound.

- β = value of best (lowest-value) choice so far for MIN.
  - MIN guarantees that the minimax value $\leq \beta$.
  - β is an upper bound.

- Whenever $\alpha \geq \beta$, pruning happens!
  - Whoever decides first determines the outcome.

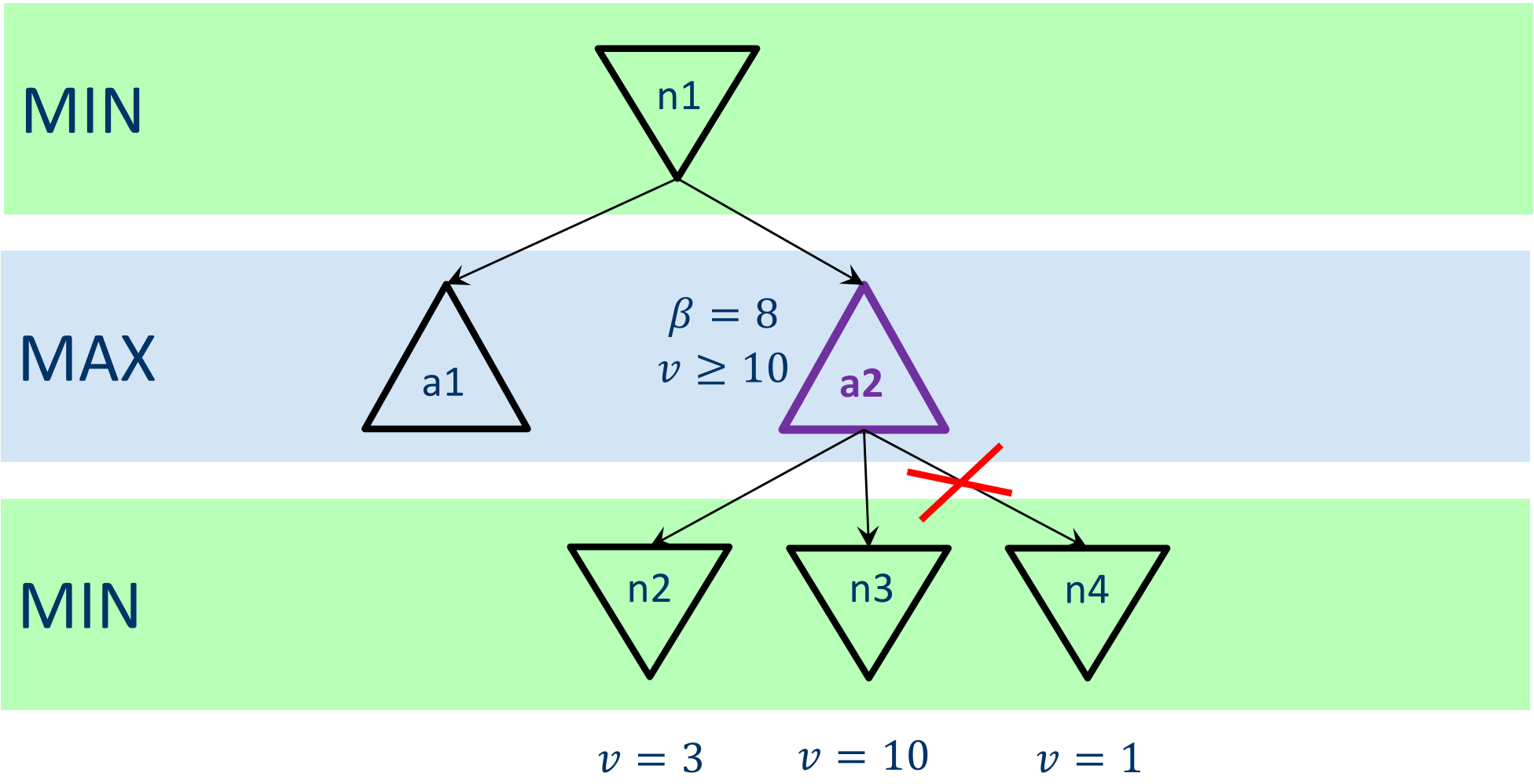
Pruning at a MAX node

- At a MAX node,
- If $v \geq \beta$, prune remaining children of current node.

Reasoning:

- MIN can guarantee a value of at most β.
- Current MAX node has value greater than β.
- Thus, the current node would NOT be reached.
  - MIN would not allow it!

Pruning at a MAX Node



At node a2, MIN can guarantee a value of ≤ 8, but MAX has a value of ≥ 10.
Thus, **a2 would NEVER be reached (MIN would NOT allow it).**

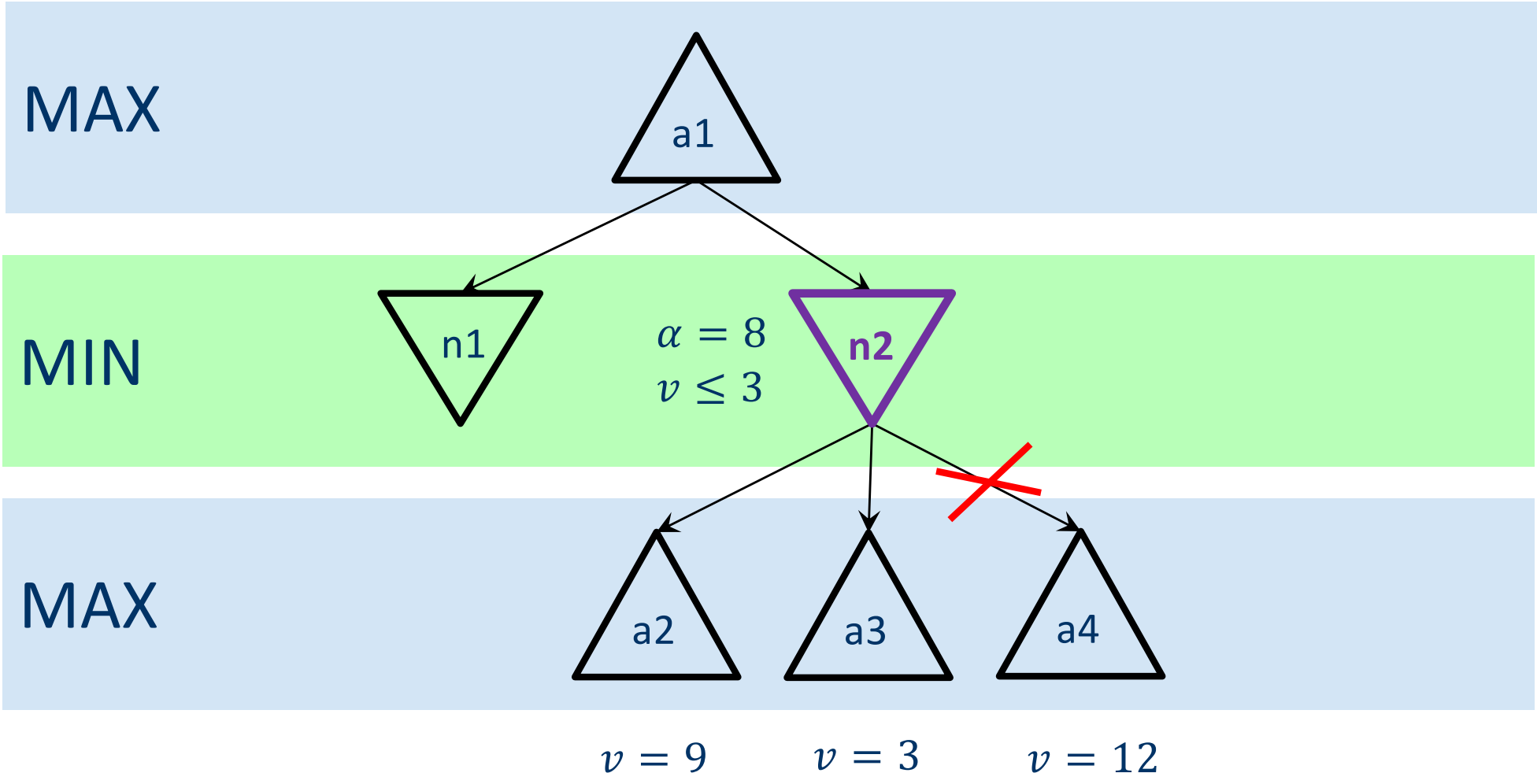
Pruning at a MIN Node

- At a MIN node,
- If $v \leq \alpha$, prune remaining children of current node.

Reasoning:

- MAX can guarantee a value of at least α.
- Current MIN node has value less than α.
- Thus, the current node would NOT be reached.
  - MAX would not allow it!

Pruning at a MIN Node



At node n2, MAX can guarantee a value of ≥ 8, but MIN has a value of ≤ 3.
Thus, **n2 would NEVER be reached (MAX would NOT allow it).**

Alpha-Beta Pruning Pseudocode

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state,−∞,+∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL(state) then return UTILITY(state)
    v ← −∞
    for each action in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, action), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL(state) then return UTILITY(state)
    v ← +∞
    for each action in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, action), α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Alpha-Beta Pruning Pseudocode

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state,−∞,+∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each action in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, action), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each action in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, action), α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Did you notice? We never return α and β to the parent node.

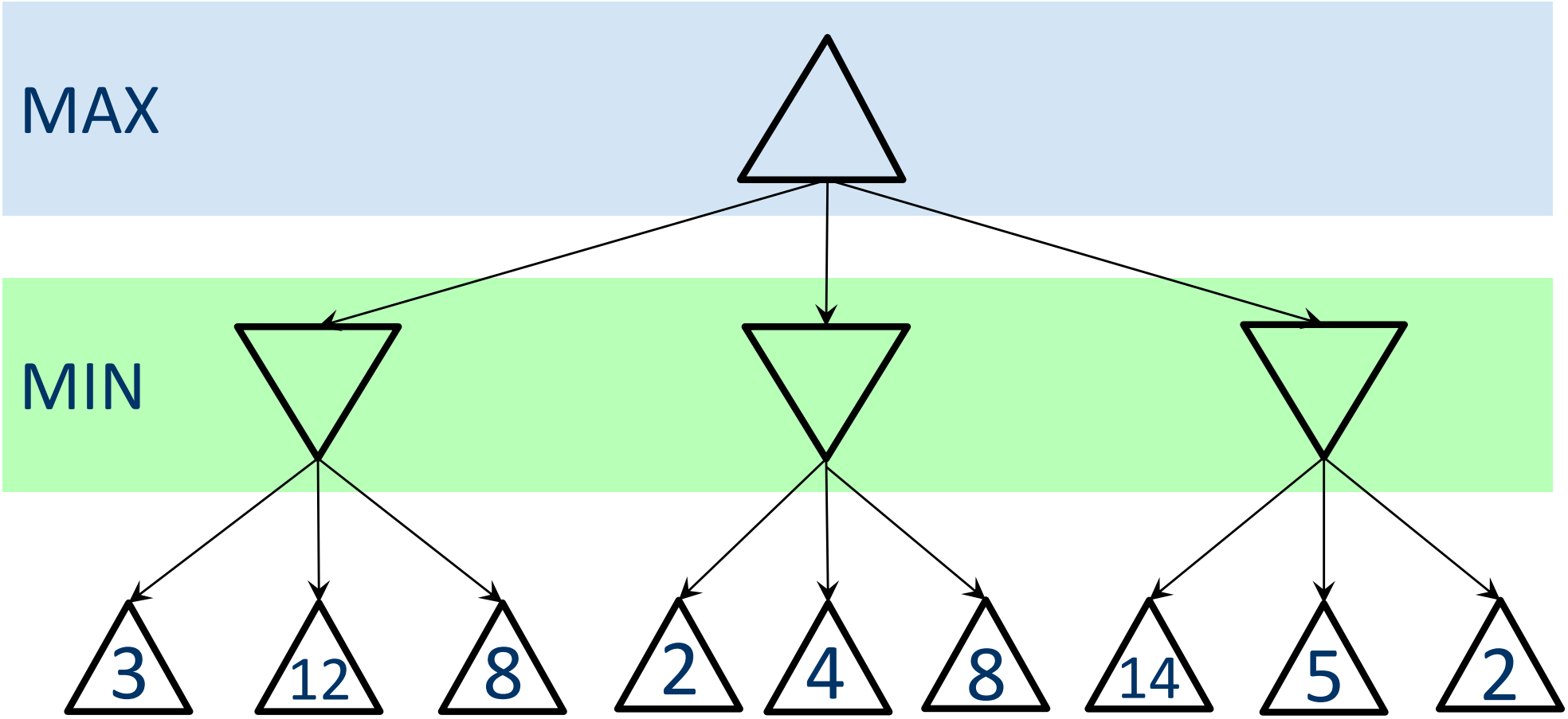
Pruning at a MAX node

MAX increases α.

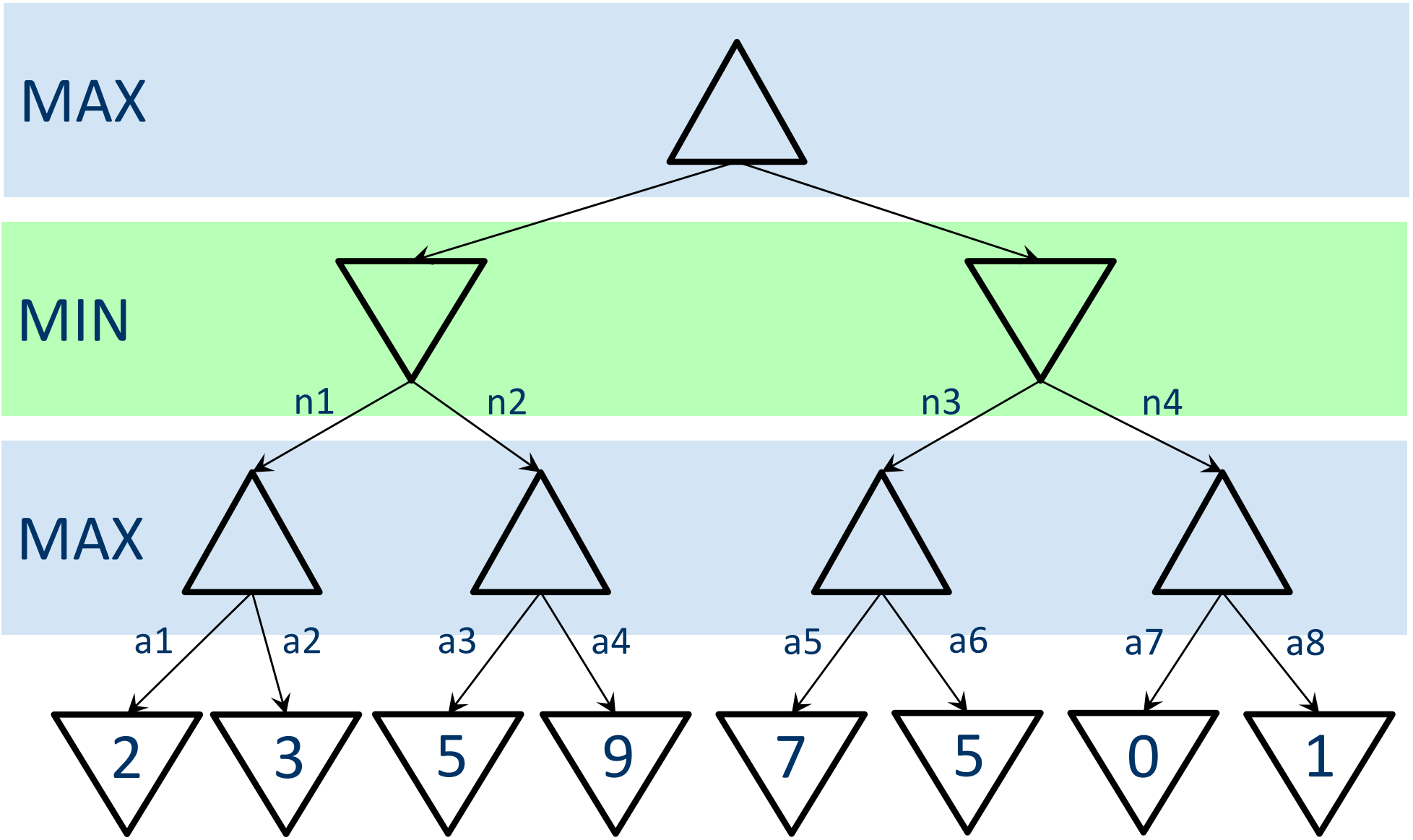
Pruning at a MIN node

MIN decreases β.

Example 1: Alpha-Beta Pruning



Example 2: Alpha-Beta Pruning



Alpha-Beta Pruning Properties

- Can pruning result in a different outcome than minimax search?
  - No. Pruning only eliminates states that we did not have to visit in the first place.

- How much can be pruned when searching?
  - Can reduce the branch factor from b to $\sqrt{b}$ w/ perfect pruning.
  - In theory, we can search **twice as deep**.

EXTENSIONS TO ALPHA-BETA PRUNING

Learning Outcomes

By the end of this section, you should be able to

- Explain how the move ordering affects the complexity of alpha-beta pruning.
- Given a game tree, change the node ordering to maximize/minimize pruning.
- Describe strategies to enhance the performance of alpha-beta pruning (move orderings, handling repeated states, and using a cut-off test and an evaluation function).
- Describe strategies to design an evaluation function with desirable properties.

Real-Time Decisions

- Alpha-beta dramatically improves over minimax.
- But it is still not good enough sometimes.
  - Need to search to terminal states for part of search space.
  - Need to make decisions quickly.
- Solutions:
  - Evaluation function + cutoff tests.
  - Move ordering.
  - Caching states.

Cut-off test + Evaluation function

Problem:

- α-β pruning must search to terminal states for part of the search space.
- Searching to terminal states is impractical!

Solution:

- Cutting off the search earlier, and
  - Terminal test -> cutoff test
  - Decides when to apply the evaluation function.
- Applying an evaluation function at cut-off.
  - Utility function -> evaluation function
  - Estimates the expected utility of the state.

Alpha-Beta Pruning with Evaluation Function

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state,−∞, +∞, 0)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β, depth) returns a utility value
    if CUTOFF-TEST(state, depth) then return EVAL(state)
    v ← −∞
    for each action in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, action), α, β, depth + 1))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β, depth) returns a utility value
    if CUTOFF-TEST(state, depth) then return EVAL(state)
    v ← +∞
    for each action in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, action), α, β, depth + 1))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

The Evaluation Function

- Returns an estimated utility of a state.
  - Just as the heuristic function estimates the distance to the goal.
- Program performance depends strongly on the quality of the evaluation function.

- For example, the evaluation function can
  - Return the actual utility for any terminal state, and
  - Return an estimated utility for any non-terminal state.

Desirable Properties of Evaluation Functions

- Ensures correct behaviour for terminal states.
- Estimated utilities of non-terminal states should strongly correlate with the actual chances of winning.
- Fast to compute!

Designing an Evaluation Function

- Use expert knowledge.
- Learn from experience.
- A weighted combination of features.
  - linear combination: $EVAL(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$
  - non-linear combination.

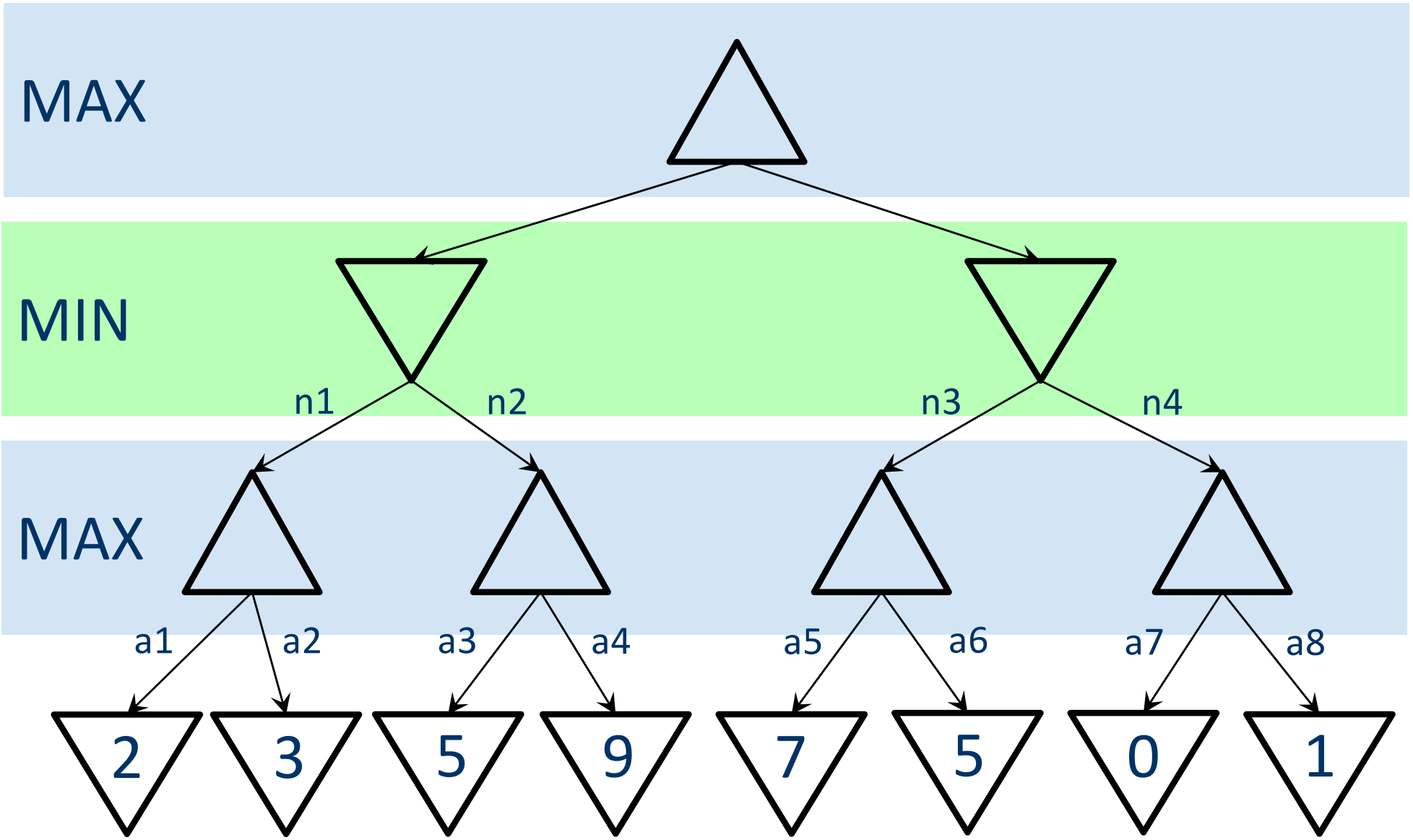
Features for estimating the state's utility

- Tic-tac-toe:
  - # of length 3 runs that are left open for each player
- Chess:
  - Each piece has a material value.
    - pawn (1), knight/bishop (3), rook (5), queen (9).
  - "Good pawn structure" or "King safety" may be worth ½ a pawn.
  - DeepBlue's evaluation function used thousands of hand-crafted features.
- Go:
  - AlphaGo's evaluation function used neural networks.

Move Ordering

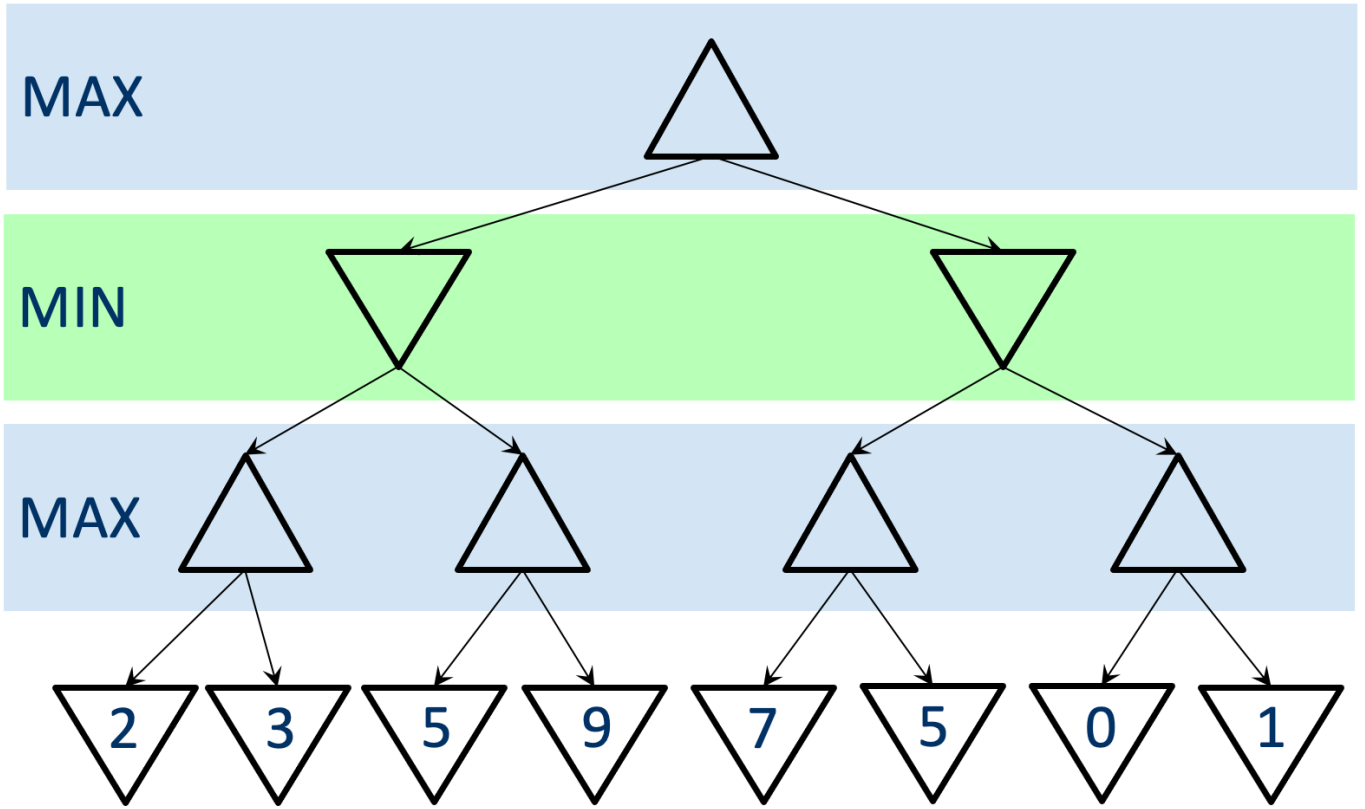
- The effectiveness of alpha-beta pruning is highly dependent on the order in which the nodes are visited.
- Ideally, we want to **visit the best child first**.
- With perfect ordering, time complexity of alpha-beta pruning becomes $O(b^{m/2})$ or $O(\sqrt{b}^{m})$.
  - The branching factor becomes $\sqrt{b}$ instead of b.

Example 2

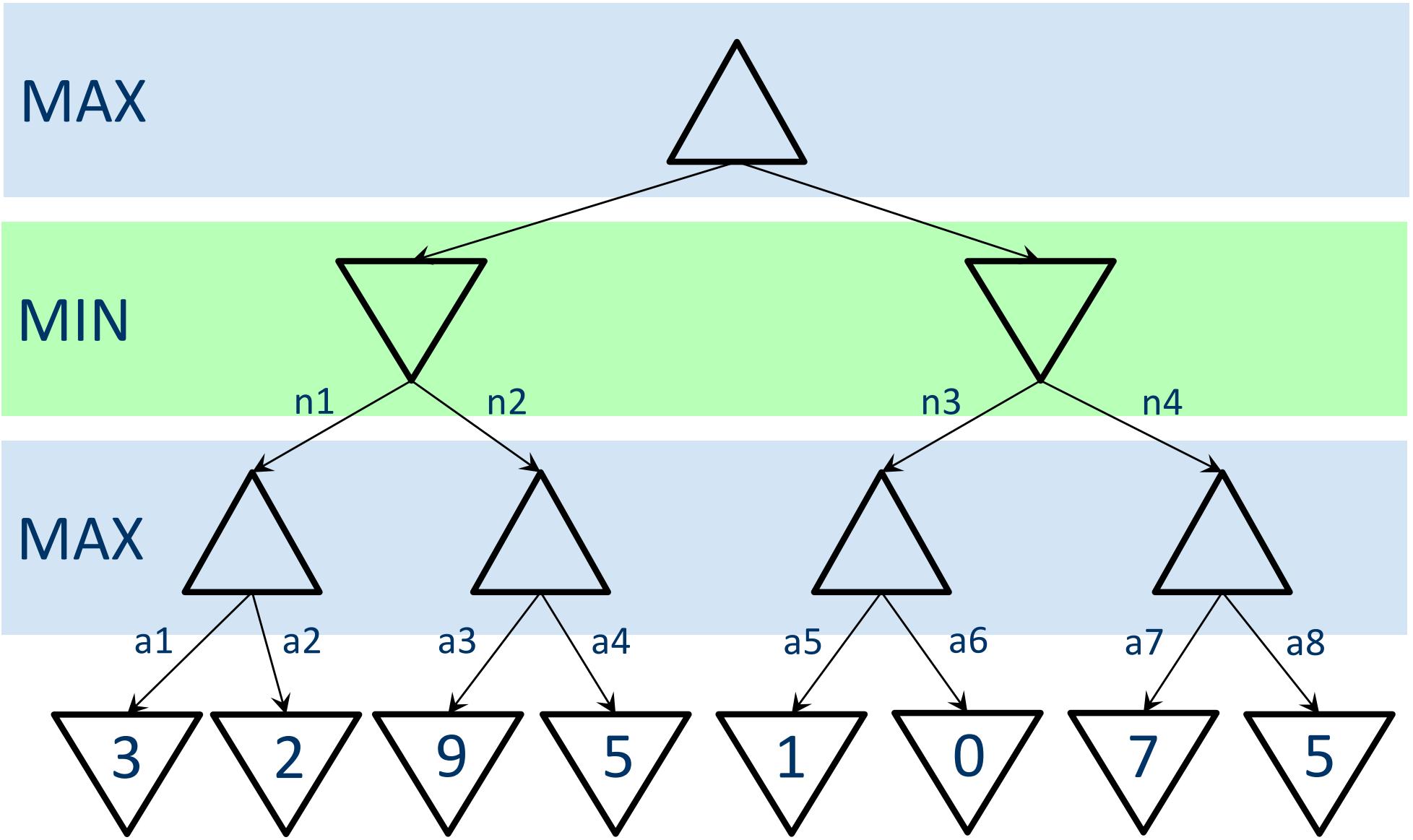


Example 2: Let's Change the Move Ordering

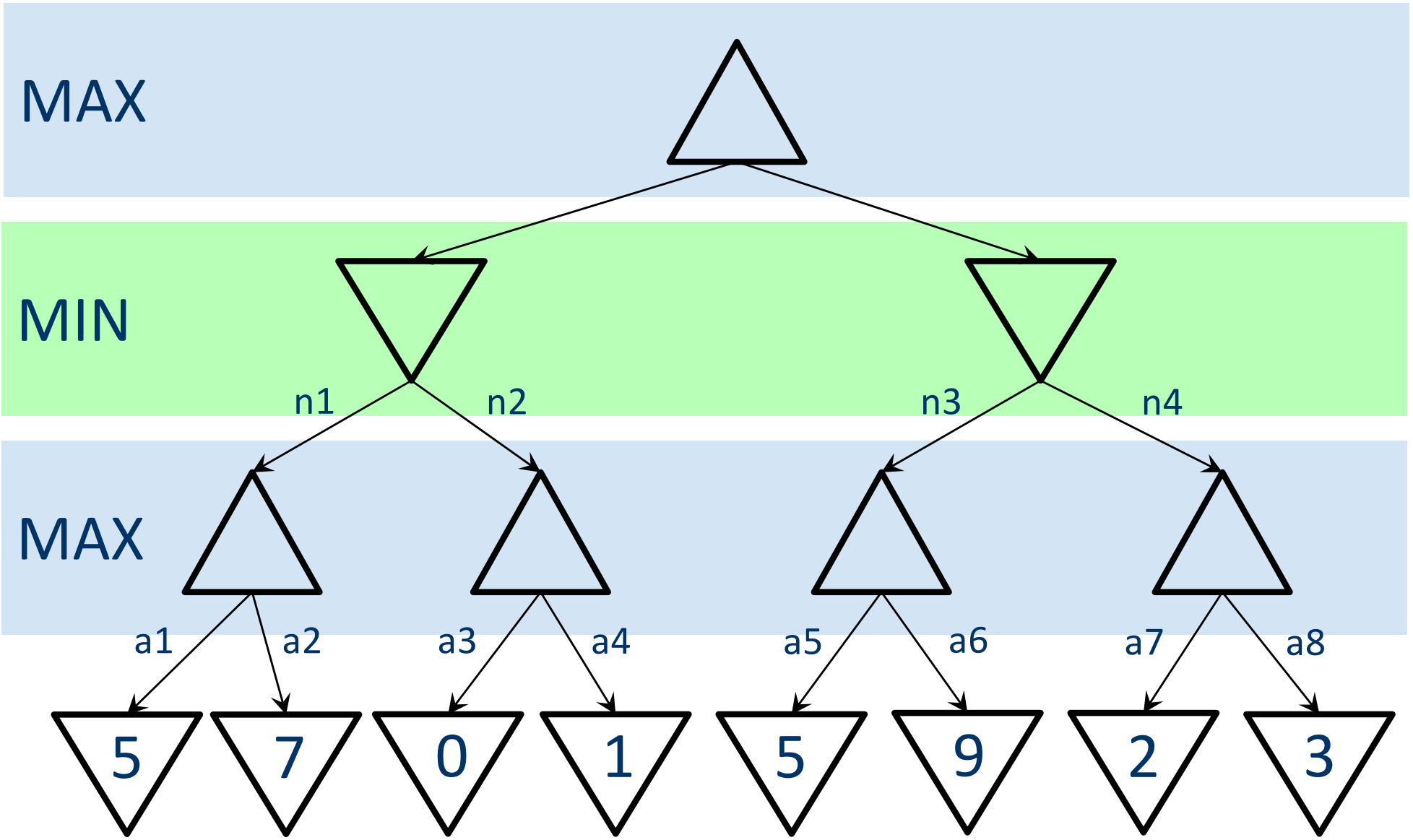
- Could you change move ordering to maximize pruning?
- Could you change move ordering to minimize pruning?



Revised Example 2a



Revised Example 2b



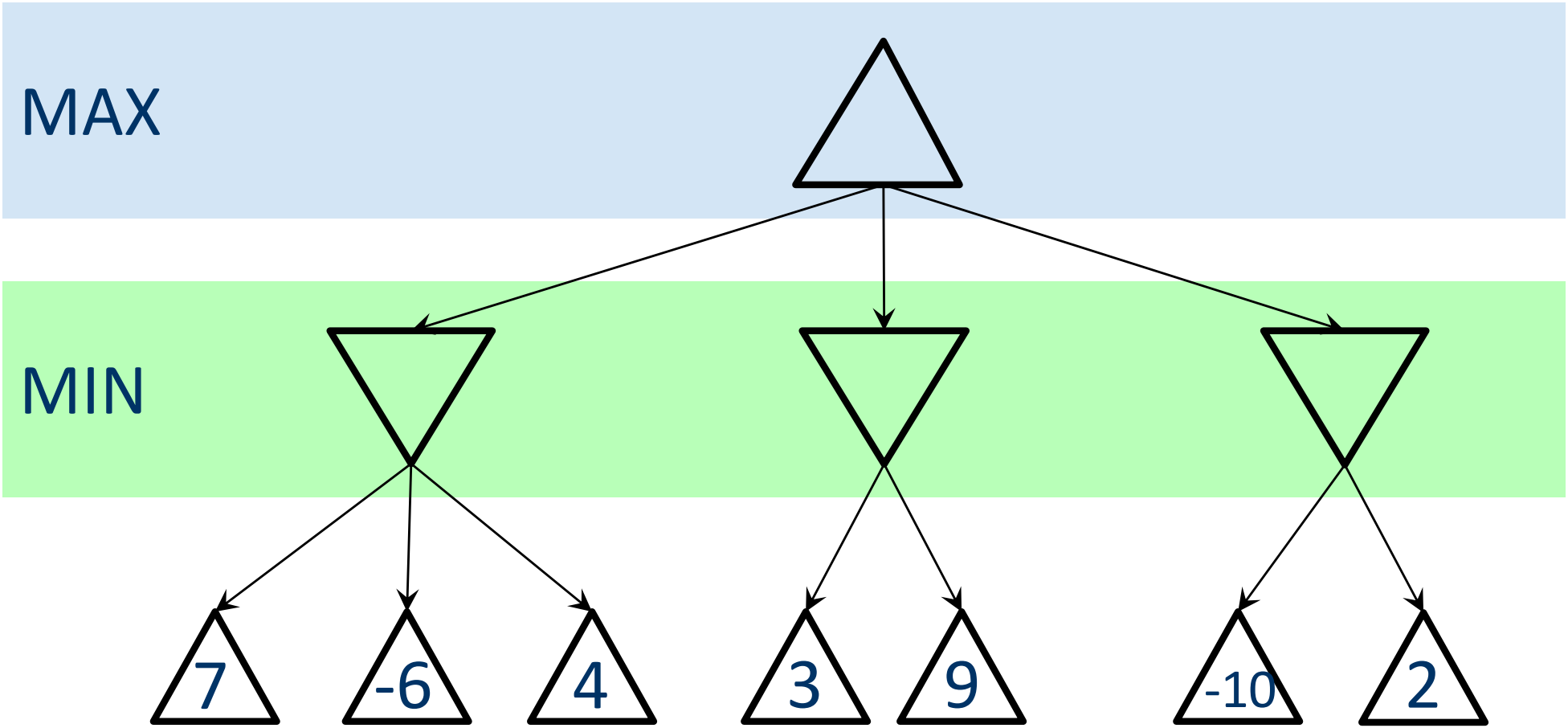
Dynamic Move Ordering

- A heuristic for chess:
  - captures first, then threats, and forward and backward moves
  - Gets you within a factor of 2 of $O(b^{m/2})$.
- Order successors using your evaluation function.
- Gain information from the current move by search.
  - Iterative deepening search
  - Search k moves deep and record the best path of moves.
  - Determine move ordering using the recorded paths.

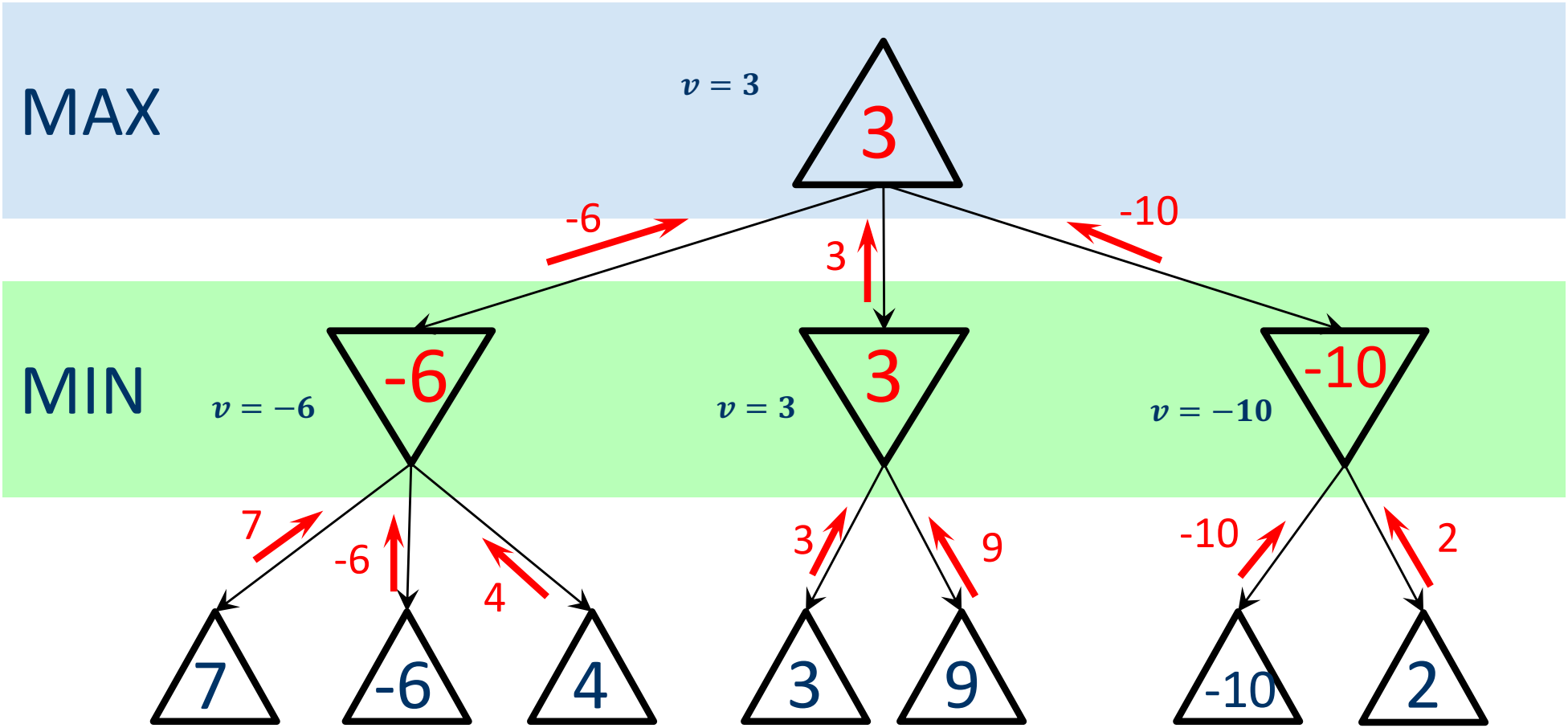
Caching States

- Permutations of moves may lead to the same position.
- Remembering repeated states can dramatically increase the maximum search depth.
  - E.g., double the search depth in chess.
- Store useful information of a state in a dictionary.
  - Like using the explored set to perform pruning.
  - Useful information: minimax value, alpha and beta values.
- May need to prune the table with limited memory.

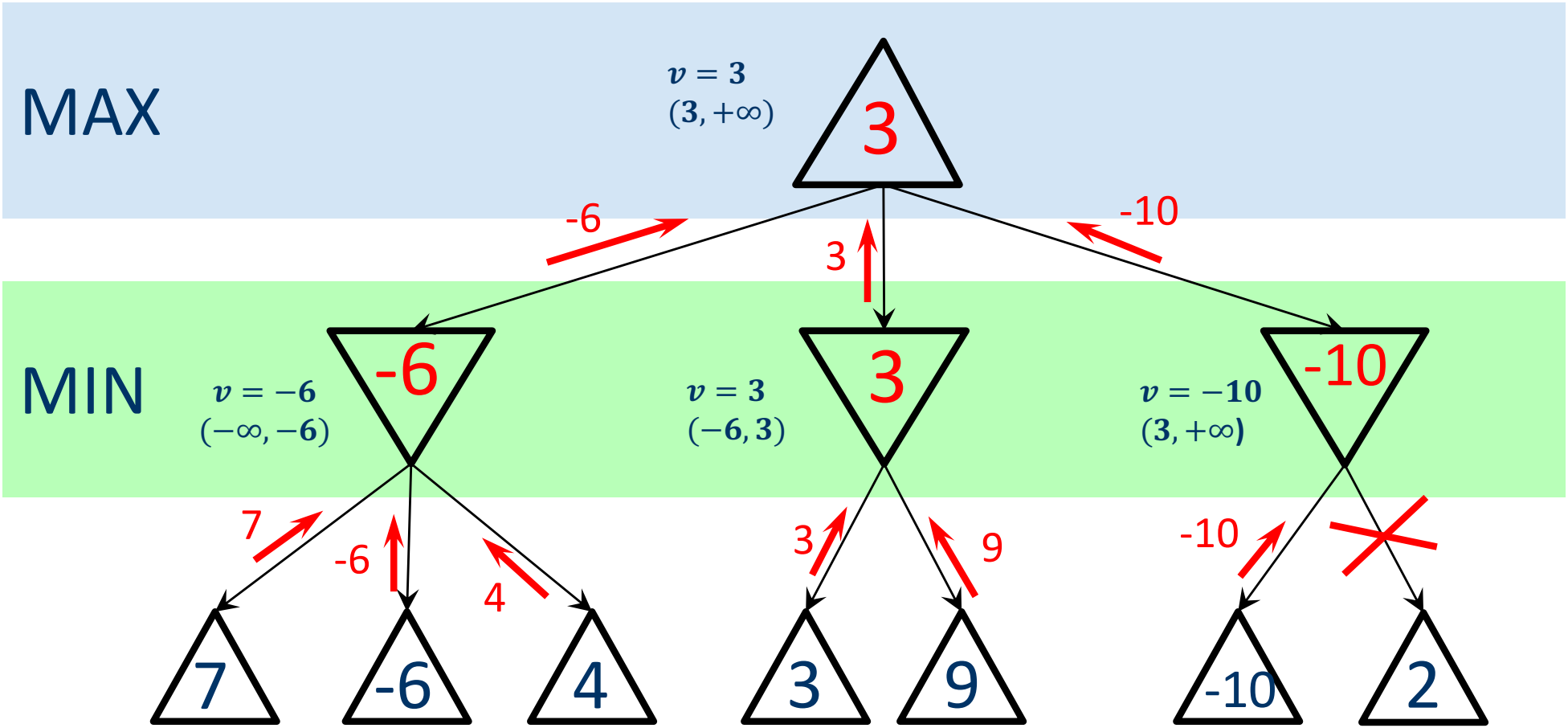
Extra Example 1



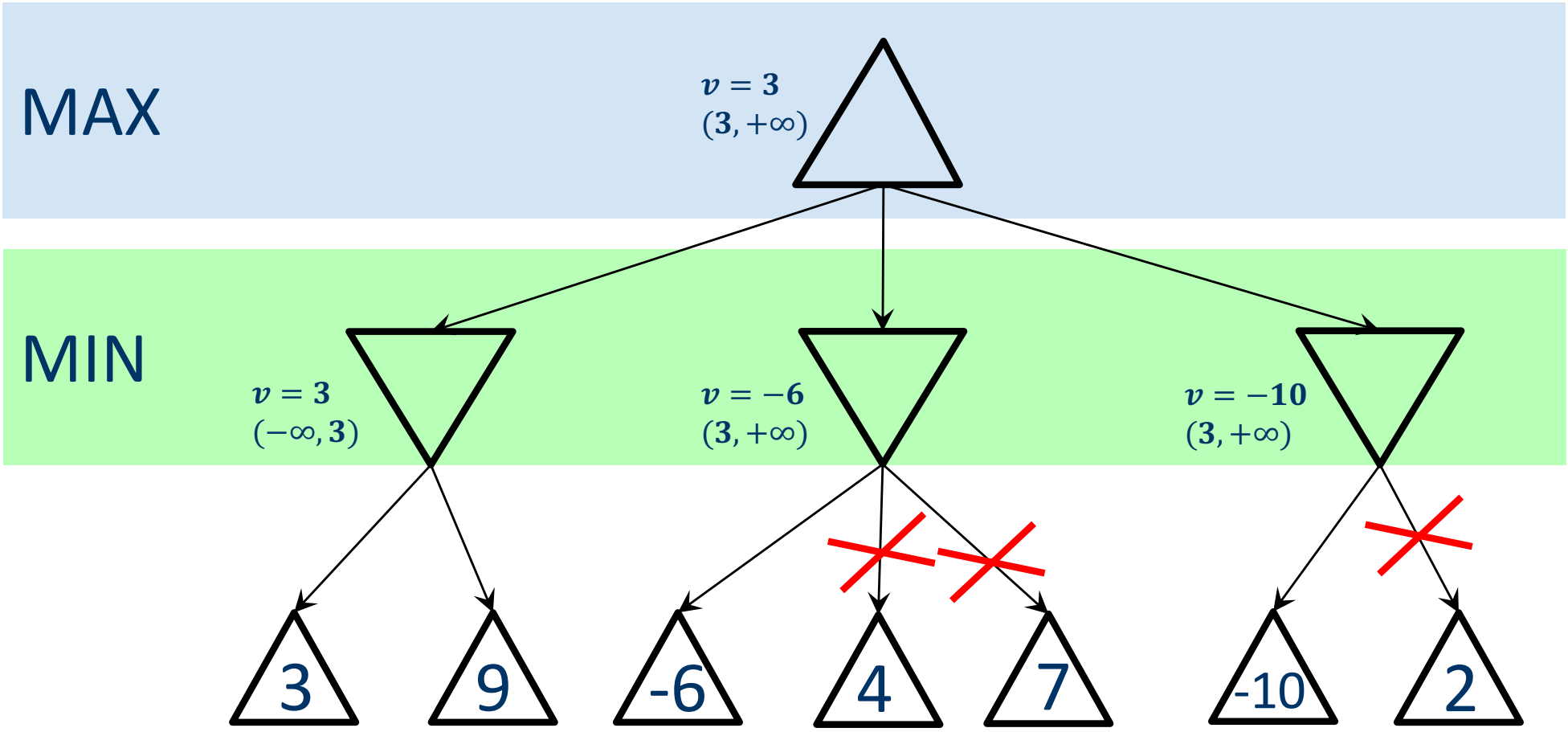
Extra 1: Depth-First Minimax



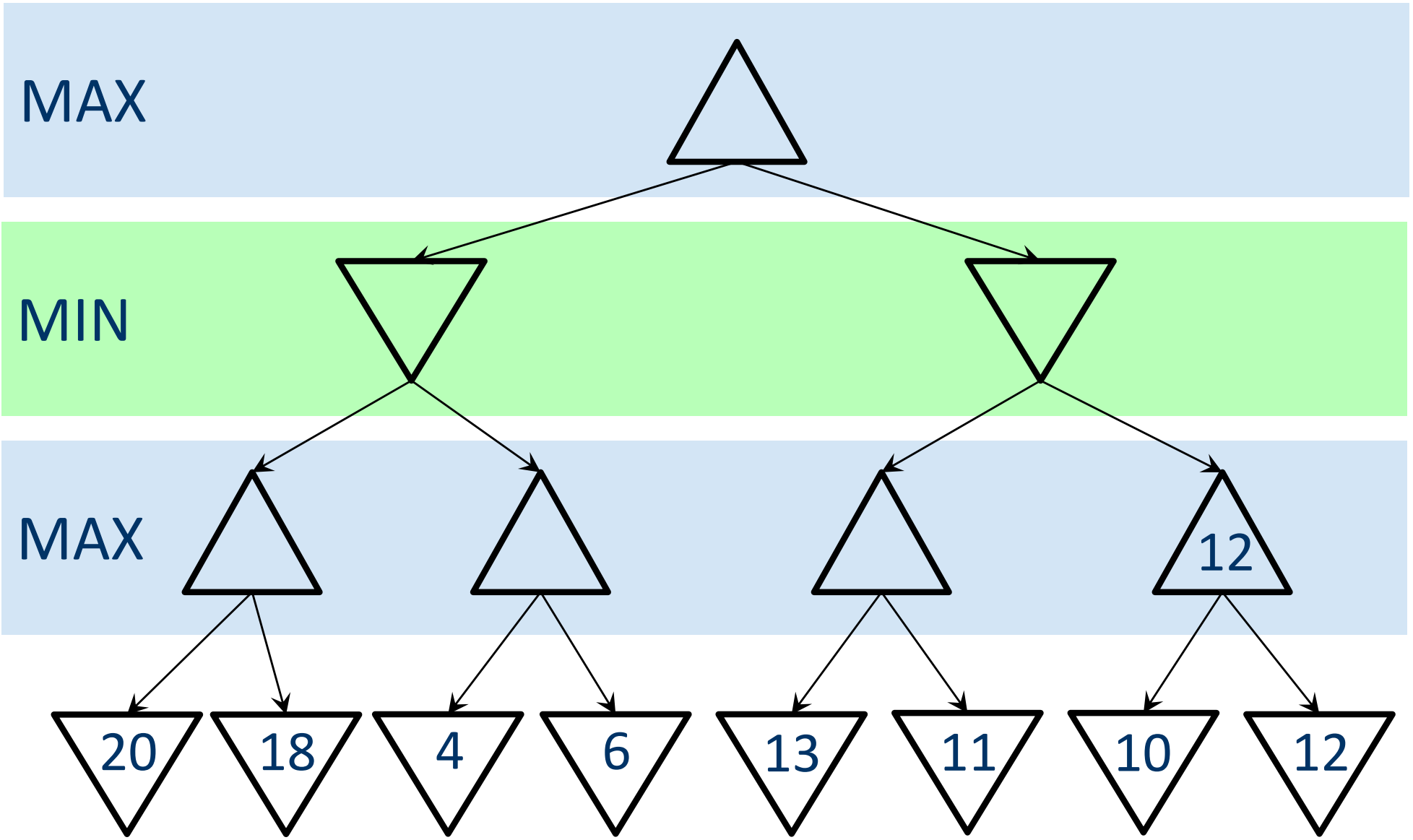
Extra 1: Alpha-Beta Pruning



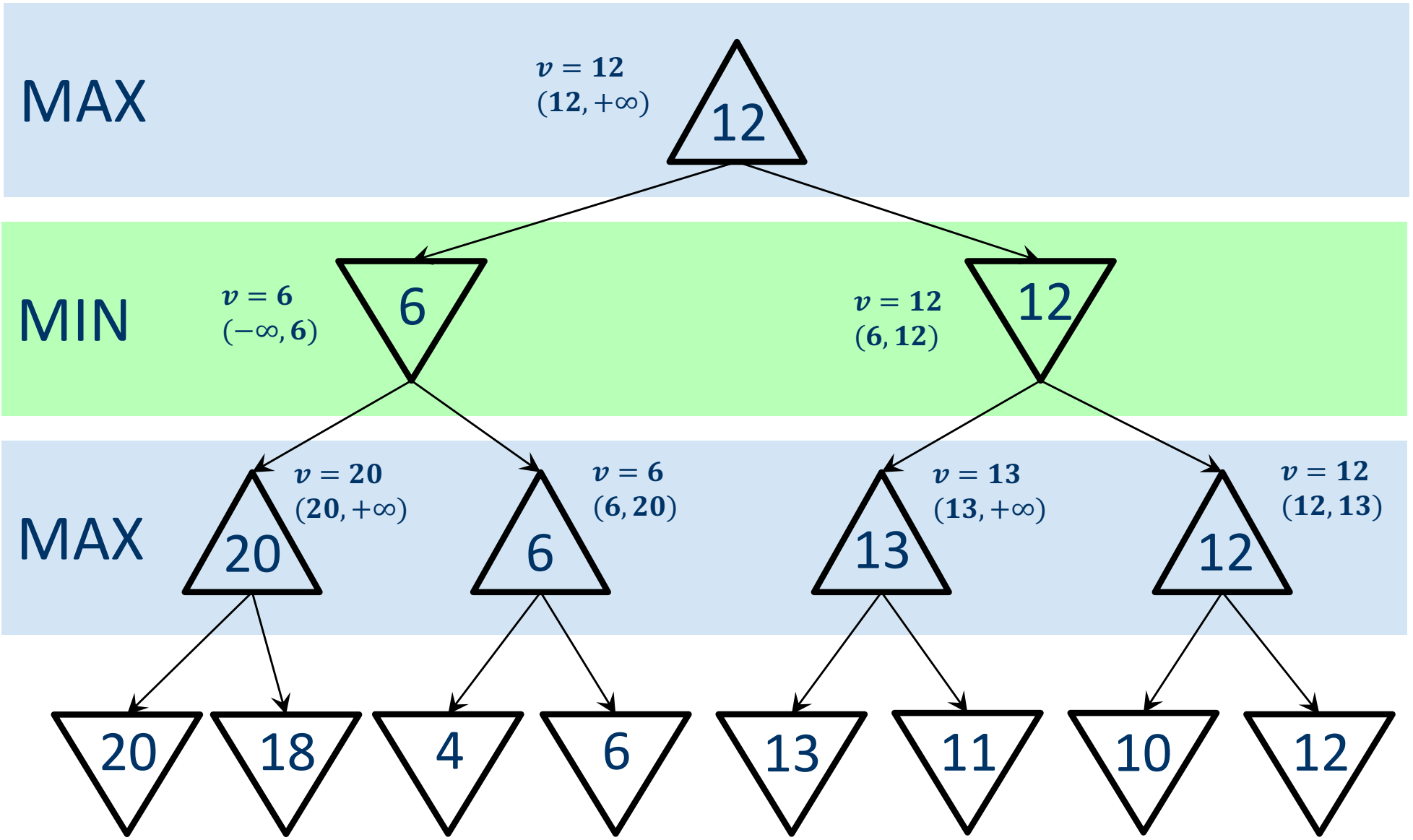
Extra 1: Move Ordering to Increase Pruning



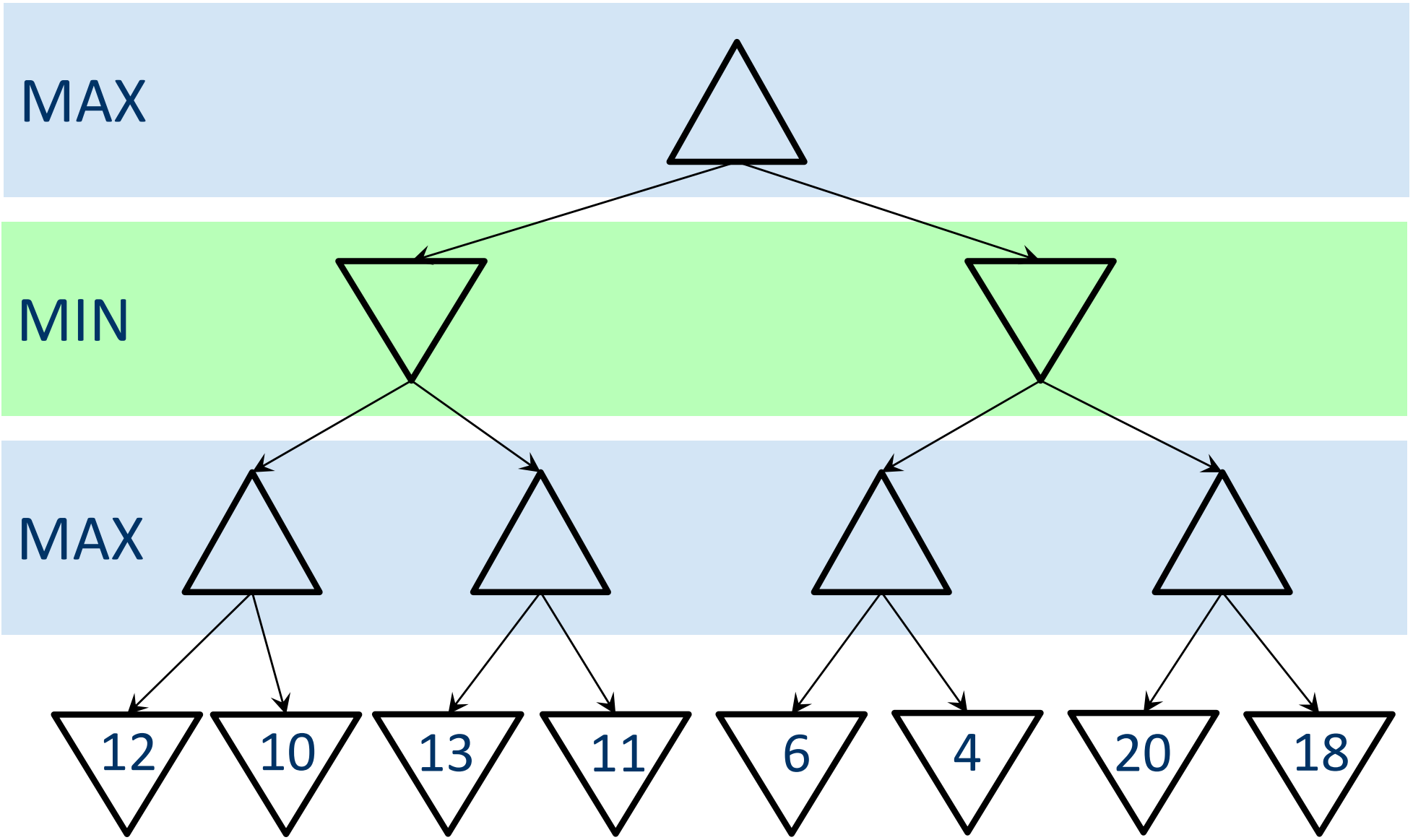
Extra Example 2



Extra Example 2



Revised Extra 2: Increase Pruning



Extra 2: Increase Pruning