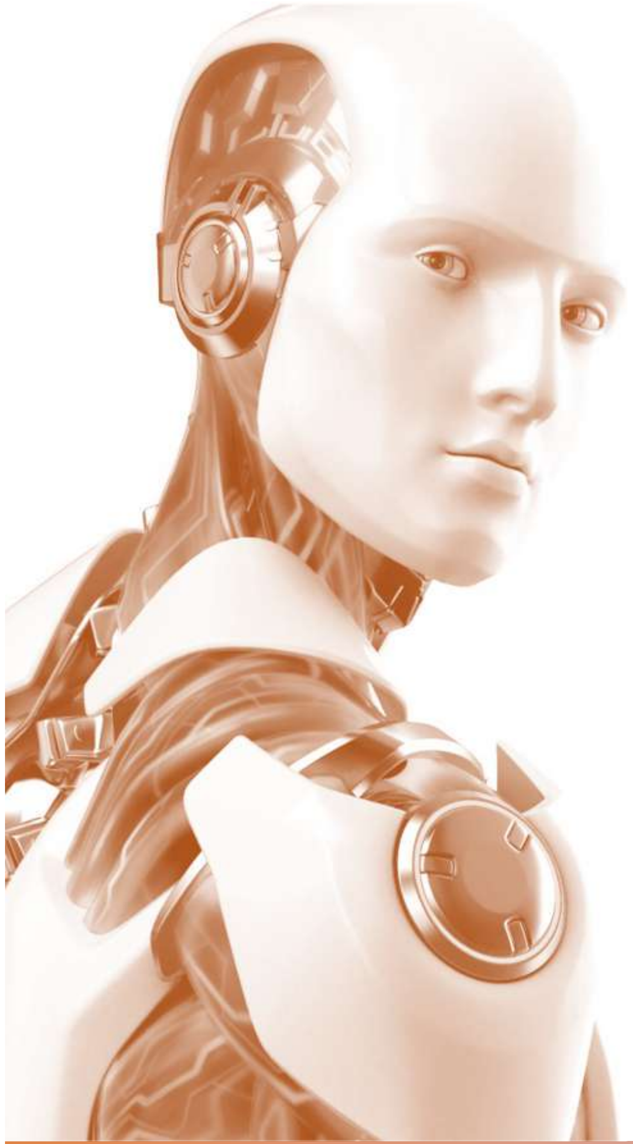
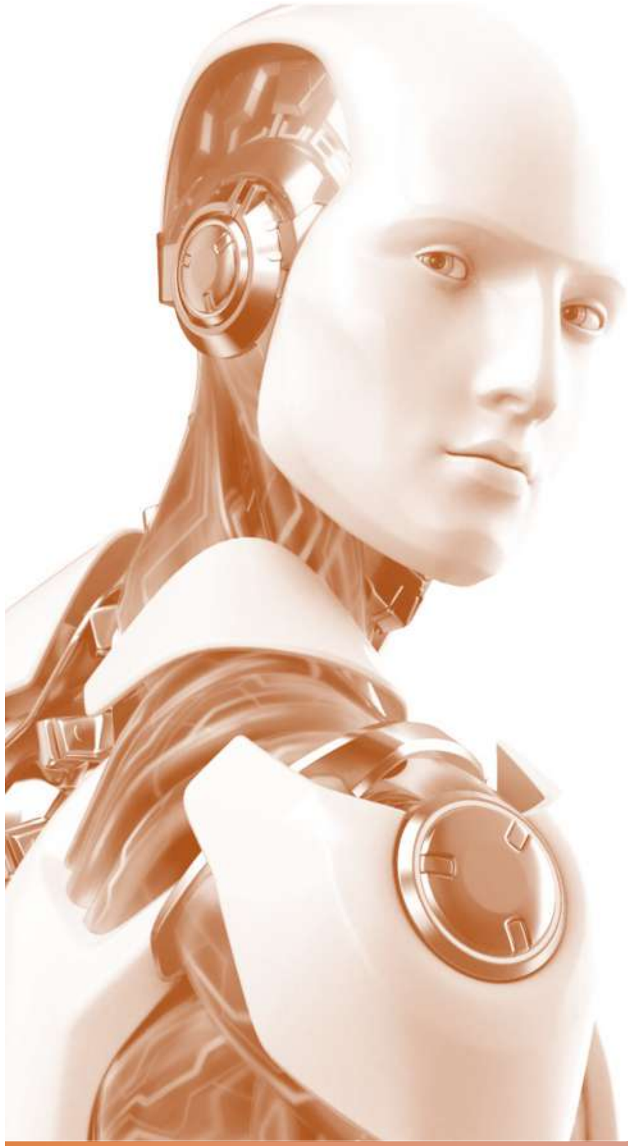


MC906/MO416 Introduction to AI

Lecture 6

UNICAMP



Introduction to AI

Lecture 6 - Search in Games

Profa. Dra. Esther Luna Colombini
esther@ic.unicamp.br

Prof. Dr. Alexandre Simões
alexandre.simoes@unesp.br

LaRoCS – Laboratory of Robotics and Cognitive Systems

Summary

- Games x Search
- History
- MINMAX
- Alpha-beta pruning
- MINMAX - more players
- Non-deterministic games

Game Playing

- Many different kinds of games
- Types:
  - Deterministic or stochastic
  - One, two, or more players
  - Perfect information (can you see the state?)

- We want algorithms for calculating a strategy (policy) which recommends a move in each state

Games vs Search

- Games are harder than search problems
- Opponent's Presence
  - introduces uncertainty
  - contingency problem: state is changed not only due to decisions made by the agent but by those of the opponent
- Search Tree Complexity
  - number of states to be tested is much higher
  - there is usually not enough time to calculate all the possible consequences of a move during a game
- Questions
  - How to find the supposed best move?
  - How to act when the decision time is limited?

AI Game History

- 1846: Charles Babbage seems to have contributed to the first serious discussion on the viability of computer chess and checkers
- 1956: John McCarthy conceived the idea of alpha-beta search, although he did not publish it
- 1961: Alpha-beta pruning was described by Hart and Edwards

AI Game History

- Chess:
  - Deep Blue beat Kasparov seeking up to 30 billion positions per move, routinely reaching a depth of 14. The assessment function had more than 8,000 characteristics.
  - Belle: first computer dedicated to the game of chess (1970s).
  - Hitech: generates 104 positions per move
    - most accurate assessment function to date
    - First to win a chess champion (1987)
    - Search about 175,000 positions / second and perform a “full-width depth-first alpha-beta search”
- Checkers:
  - Chinook: program that plays checkers. Became world champion in 1994
    - Previous systems used learning to discover the best assessment function
    - Employs alpha-beta cut + a Database with perfect solutions
- Othello (reversi): much better than humans. Chess variation with much smaller search space: 5 to 15 possible moves.
- Go: the most popular board game in Asia, requiring at least as much discipline from its professionals as chess. Since the board is 19 × 19 and bids are allowed in the entry of (almost) all blank squares, the branch factor starts at 361, a scary value for common alpha-beta search methods
  - AlphaGo

Game Types

- Perfect information games: accessible environment
- Imperfect information games: inaccessible environment

	Deterministic	Non-Deterministic
Perfect information	Go, othello, chess	Gammon, *monopoly*
Imperfect information	Naval battle	Bridge, poker

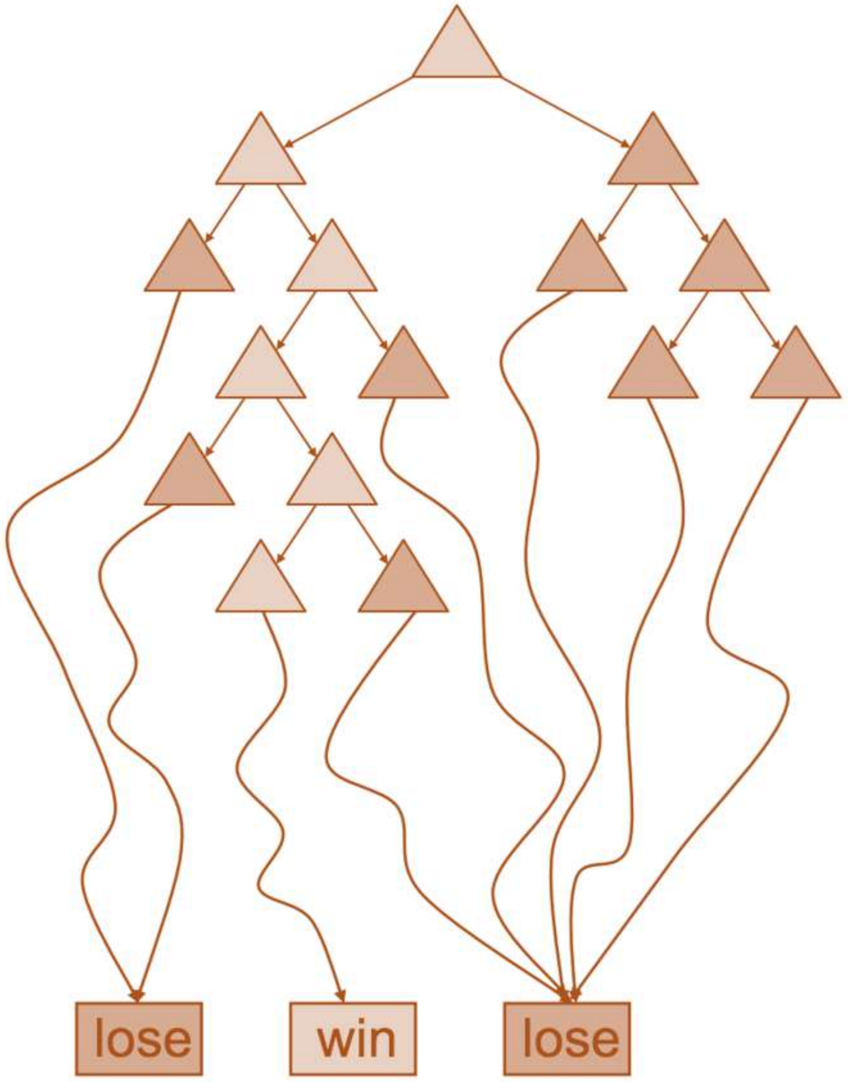
Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player/state)
  - Transition Function: S x A $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: S x P $\rightarrow$ R

- Solution for a player is a policy: S $\rightarrow$ A

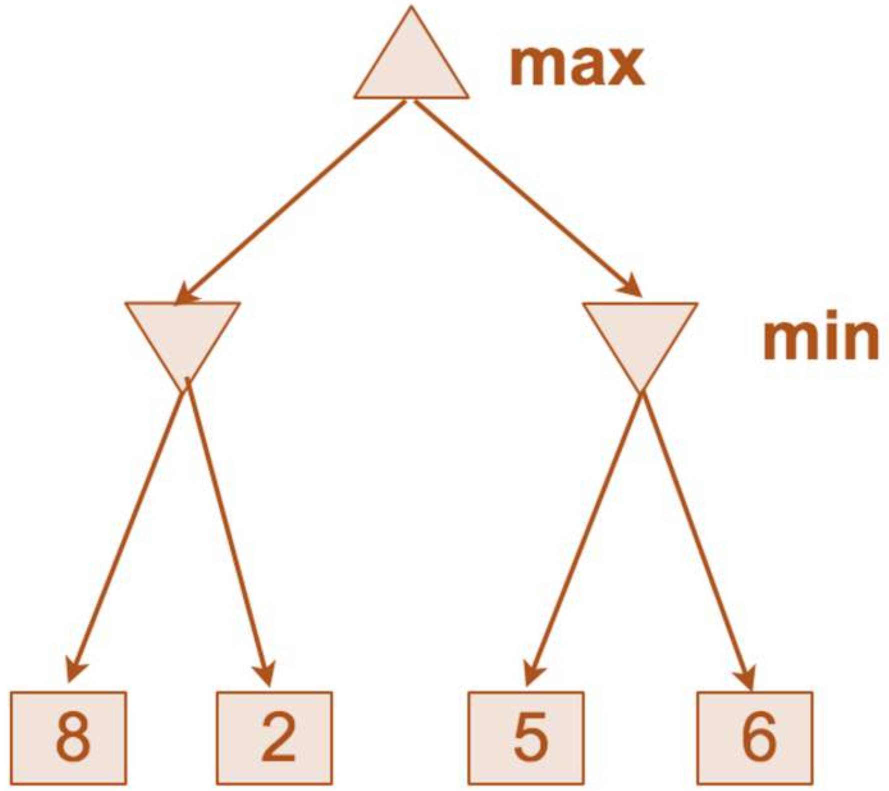
Deterministic Single Player

- Last lectures
  - Deterministic, single player, perfect information:
    - Know the rules, action effects, winning states
    - Freecell, 8-Puzzle, Rubik's cube … it's just search!
  - Slight reinterpretation:
    - Each node stores a value: the best outcome it can reach
    - This is the maximum outcome of its children (the max value)
    - Note that we don't have path sums as before (utilities at end)
  - After search, can pick move that leads to best node



Deterministic Two-Player

- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
  - One player maximizes result
  - The other minimizes result
- Minimax search



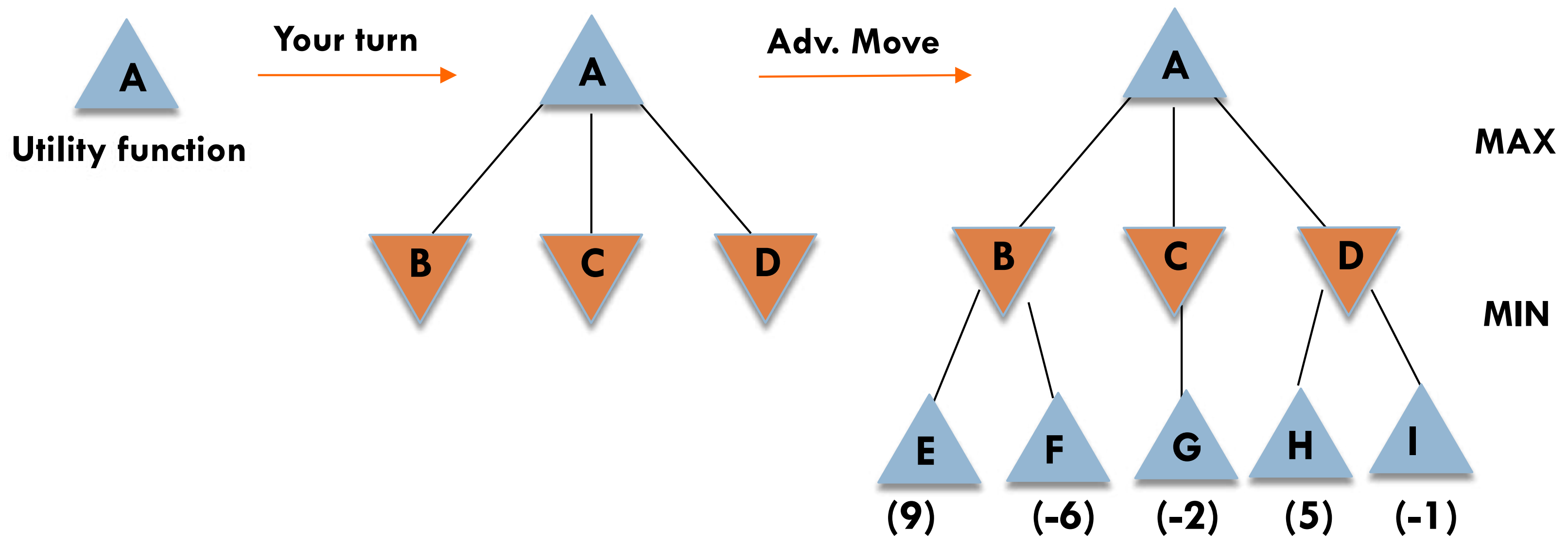
MINMAX Algorithm

- Proposes moves for deterministic games with perfect information
  - MAXIMIZE your play
  - MINIMIZE the opponent's play
  - Idea: choose the move to the state with the highest minmax value = best move against the best of the opponent
- MINMAX algorithm (characteristics):
  - You need a utility function;
  - Choose a step that maximizes your chance and decreases your opponent's chance;
  - Consider that the opponent is very intelligent → will choose the best option for him and, consequently, worse for you;
  - The opponent does not make mistakes due to distraction and is not affected psychologically (overestimates the opponent)

MINMAX Algorithm

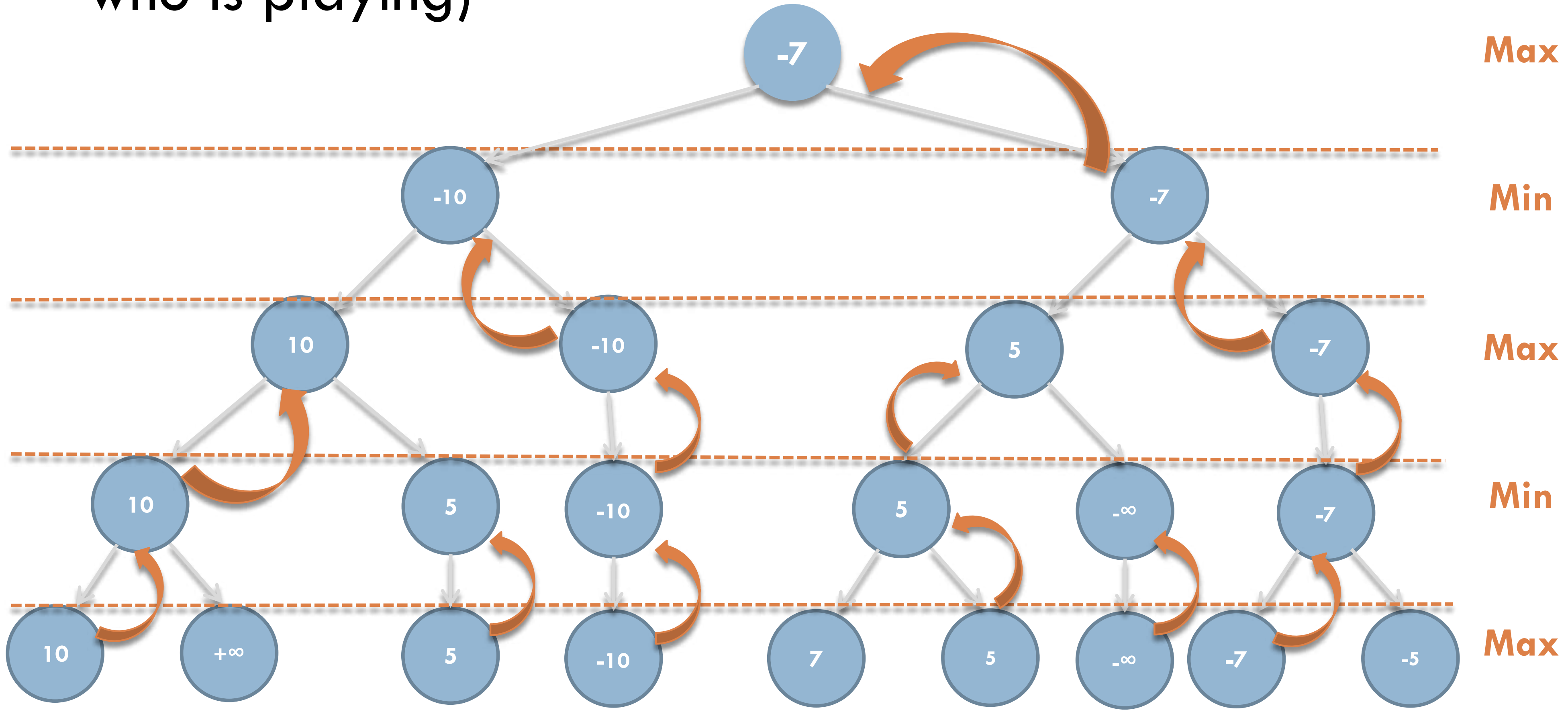
In your opponent's (MIN) play he will choose the worst situation for you. In your play (MAX) you choose the best.

Example:



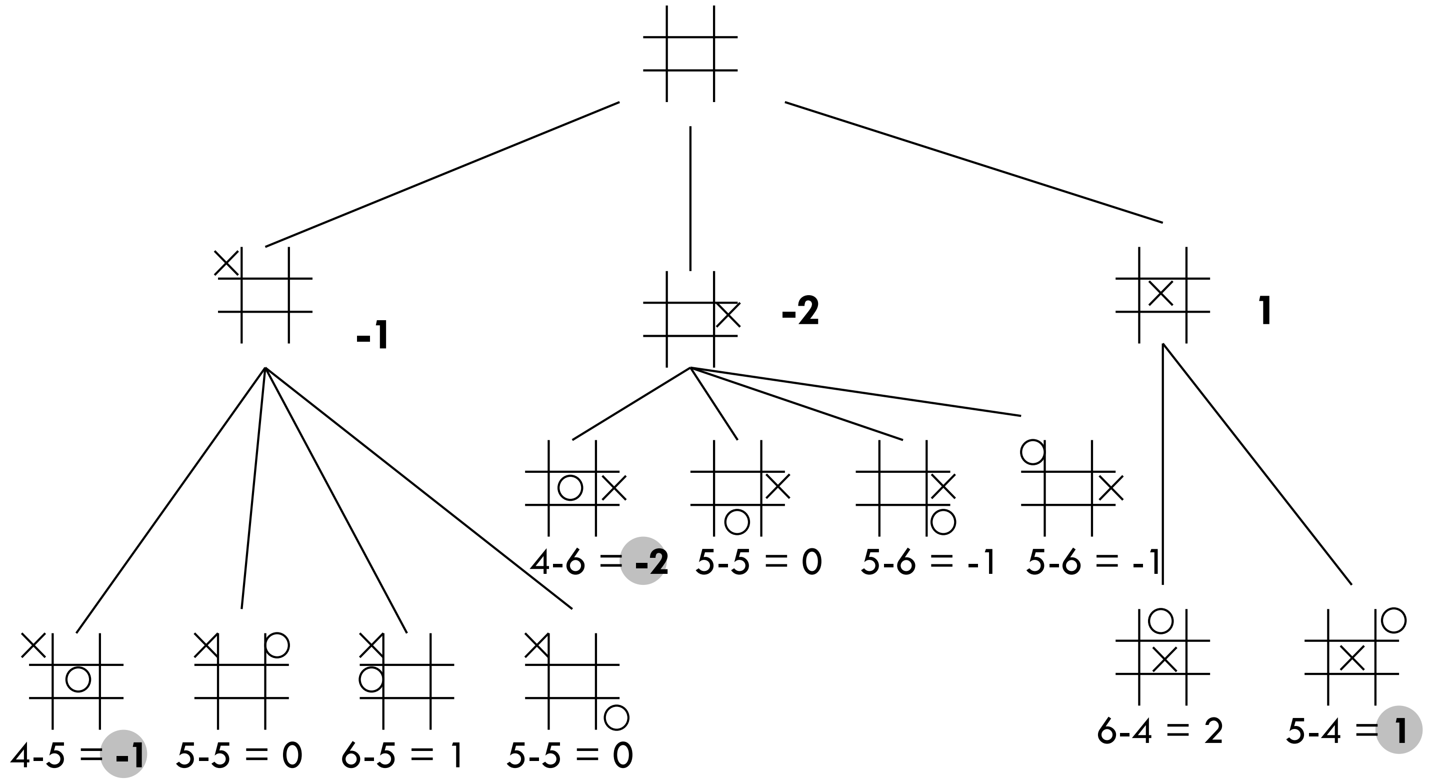
General outline of MINMAX

- Calculates the utility function of the last level of analysis and then "raises" the MIN or MAX Values (depends on who is playing)



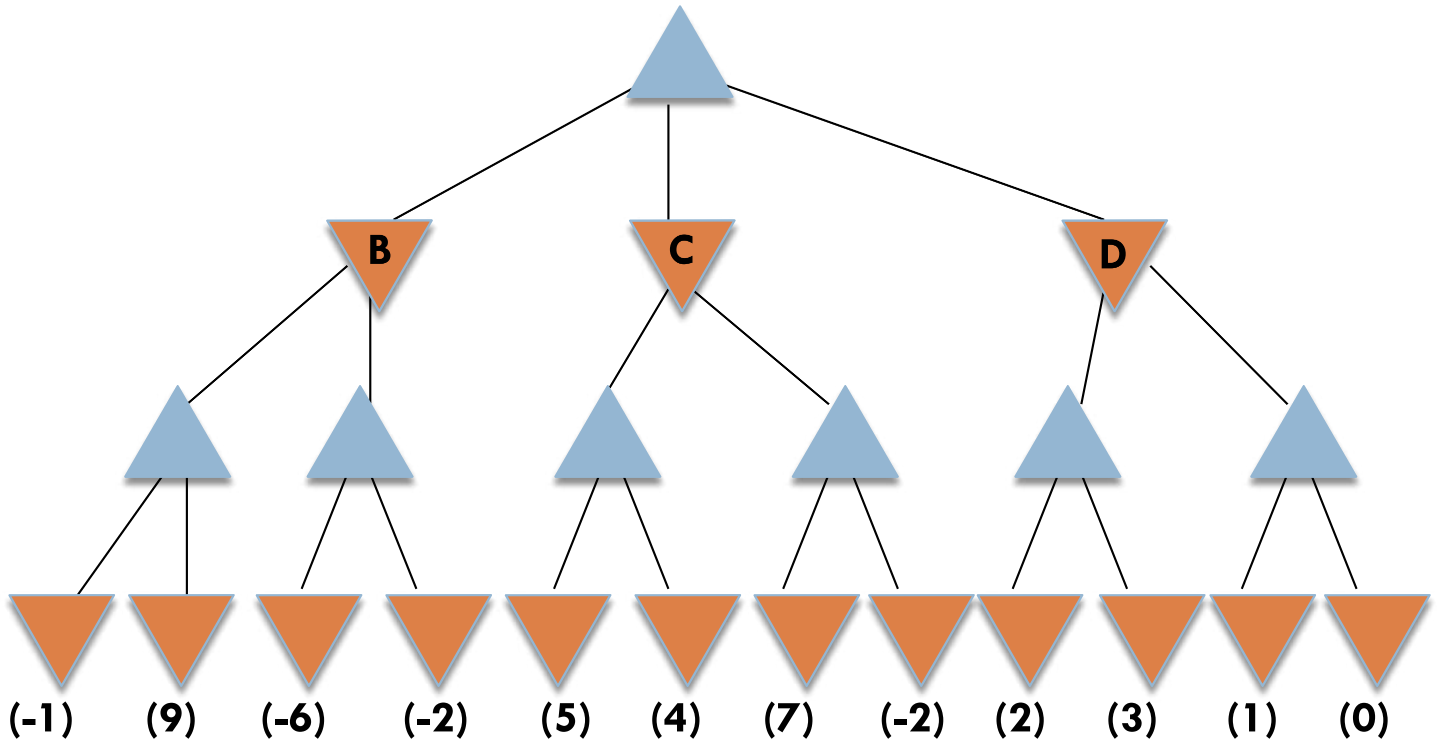
MINMAX Algorithm: Tic-Tac-Toe

- Utility function (F):
  - F = number of chances for X to win - number of chances for O



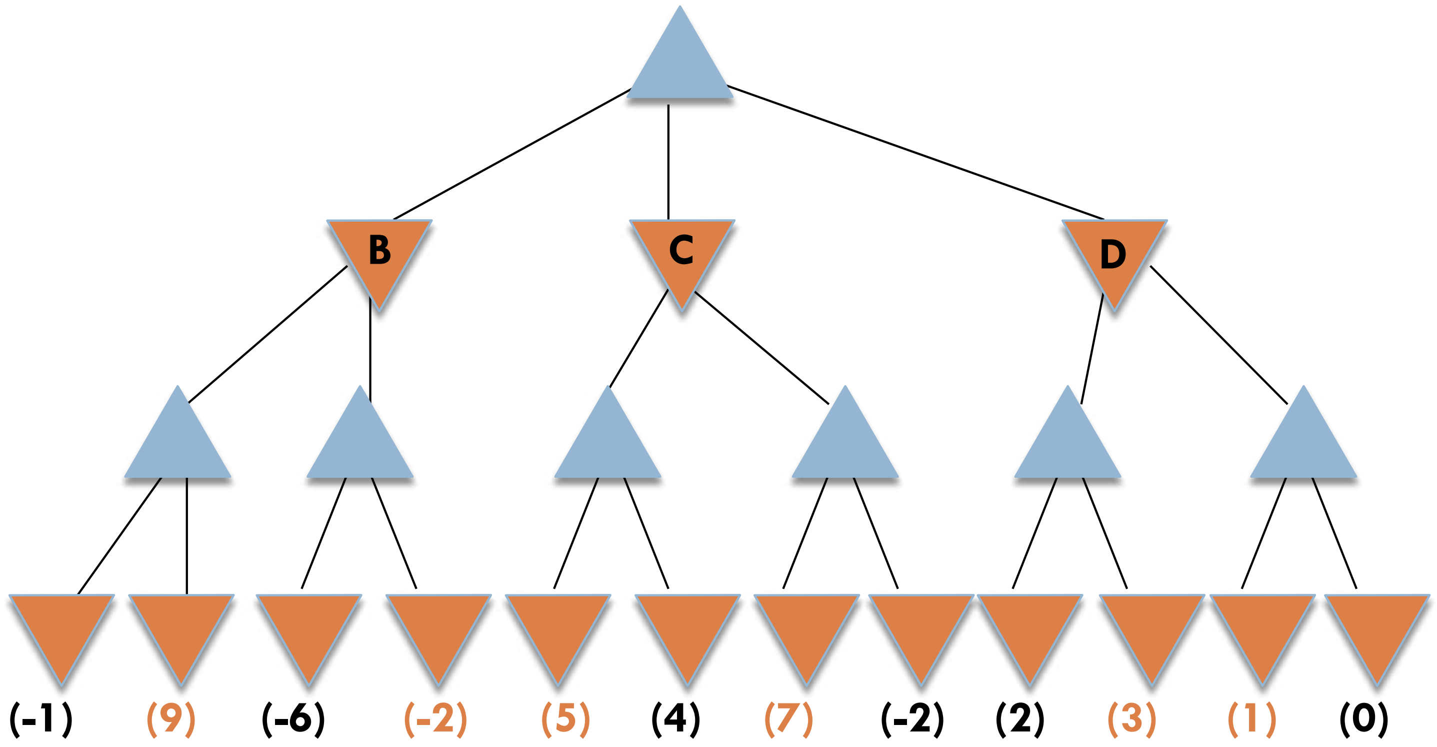
Exercise MINMAX

- What is the best move, B, C or D?



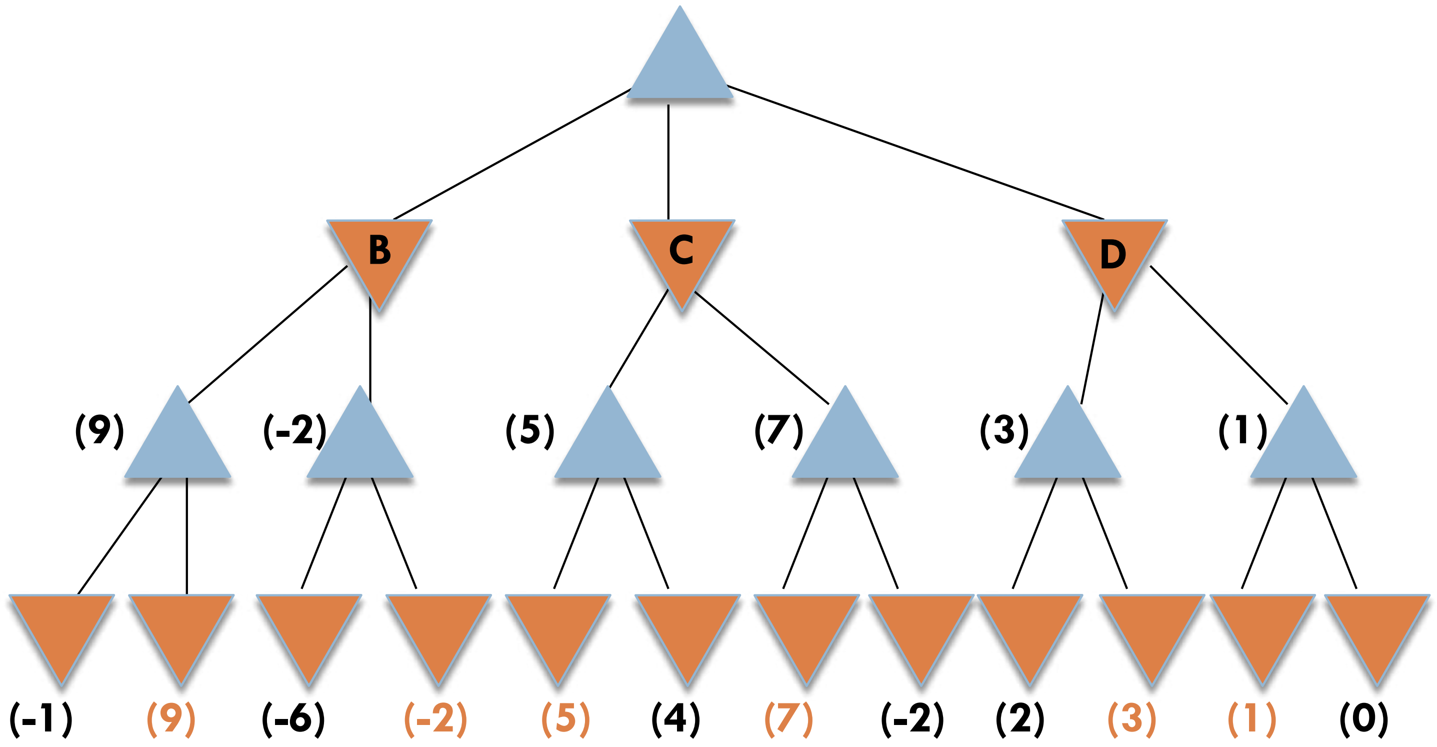
Exercise MINMAX

- Solution:



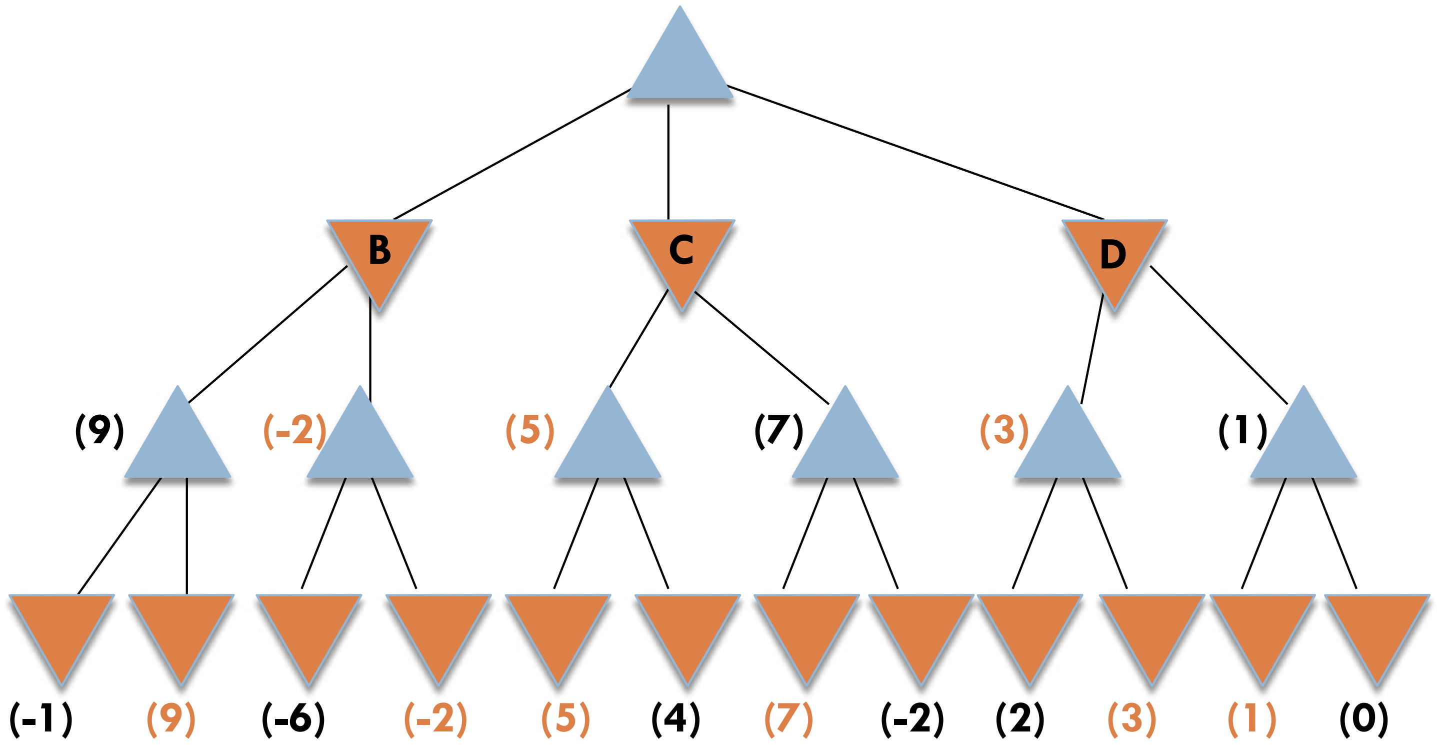
Exercise MINMAX

- Solution:



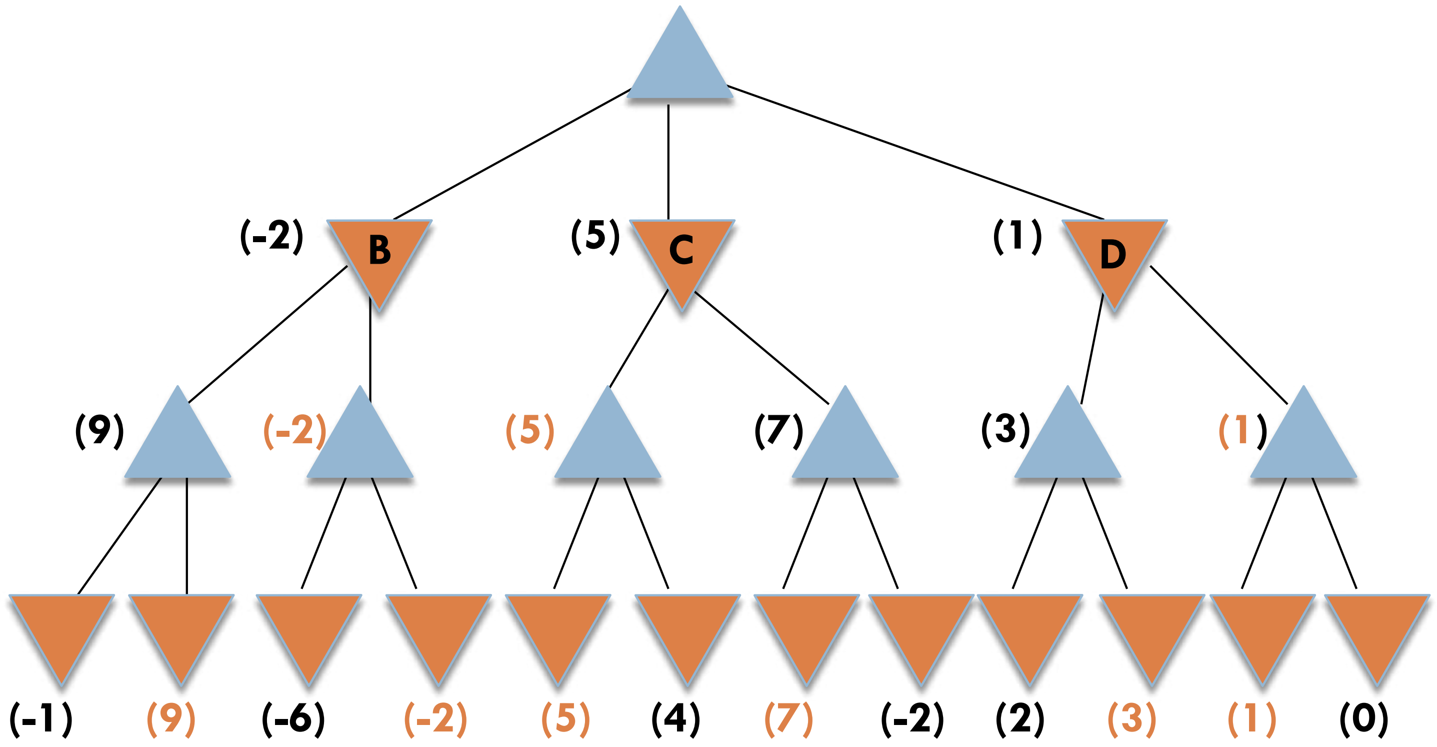
Exercise MINMAX

- Solution:



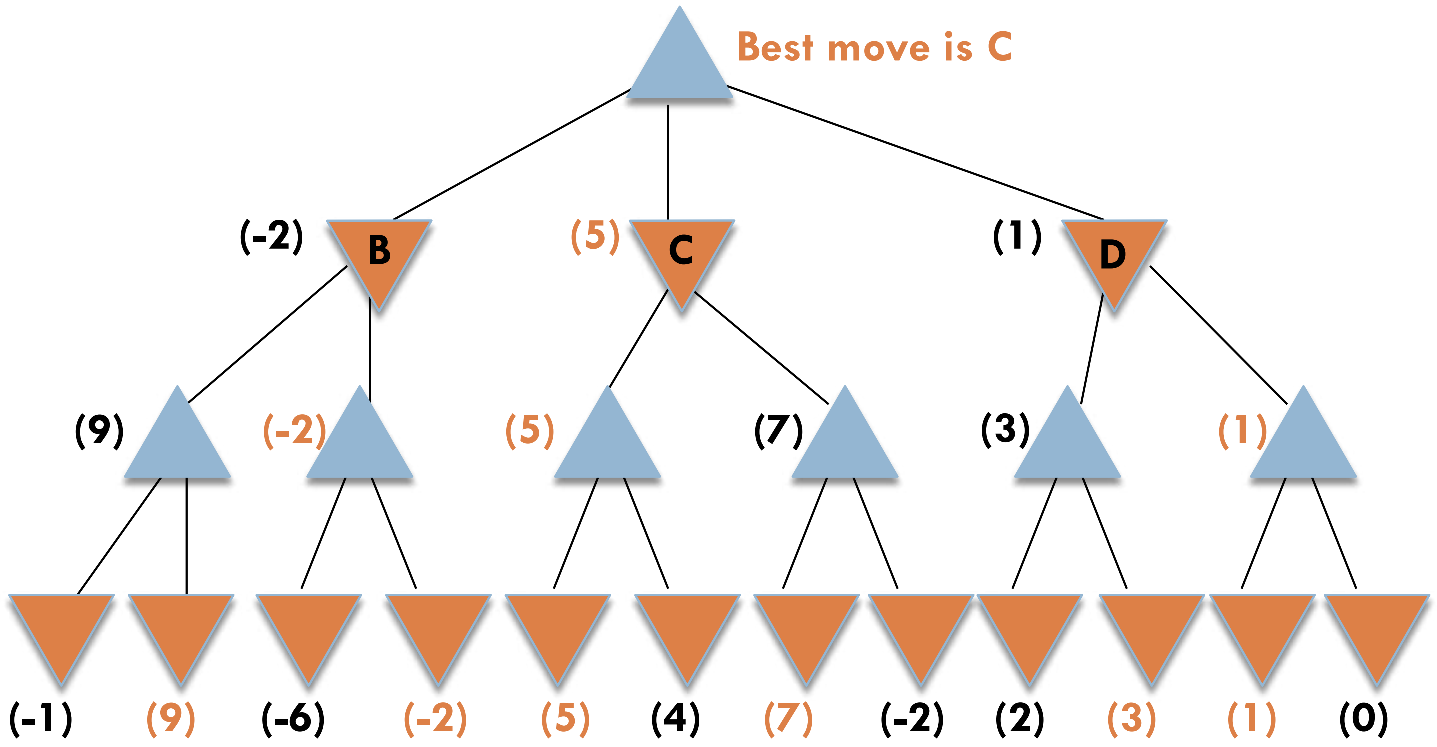
Exercise MINMAX

- Solution:



Exercise MINMAX

- Solution:



MINMAX: Algorithm

```
function DECISION-MINIMAX(state) returns an action
    return argmax$_{a \in Actions(s)}$ VALUE-MIN(RESULT (state, a))
```

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

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

MINMAX: Evaluation

- Complete?
  - Yes, if the tree is finite (chess has rules to ensure that)
- Optimal?
  - Yes, against an opponent
- Time complexity
  - $O(b^m)$
- Memory complexity
  - $O(b^m)$ – if you generate all successors at once
  - $O(bm)$ – depth-first exploration

Improving MINMAX

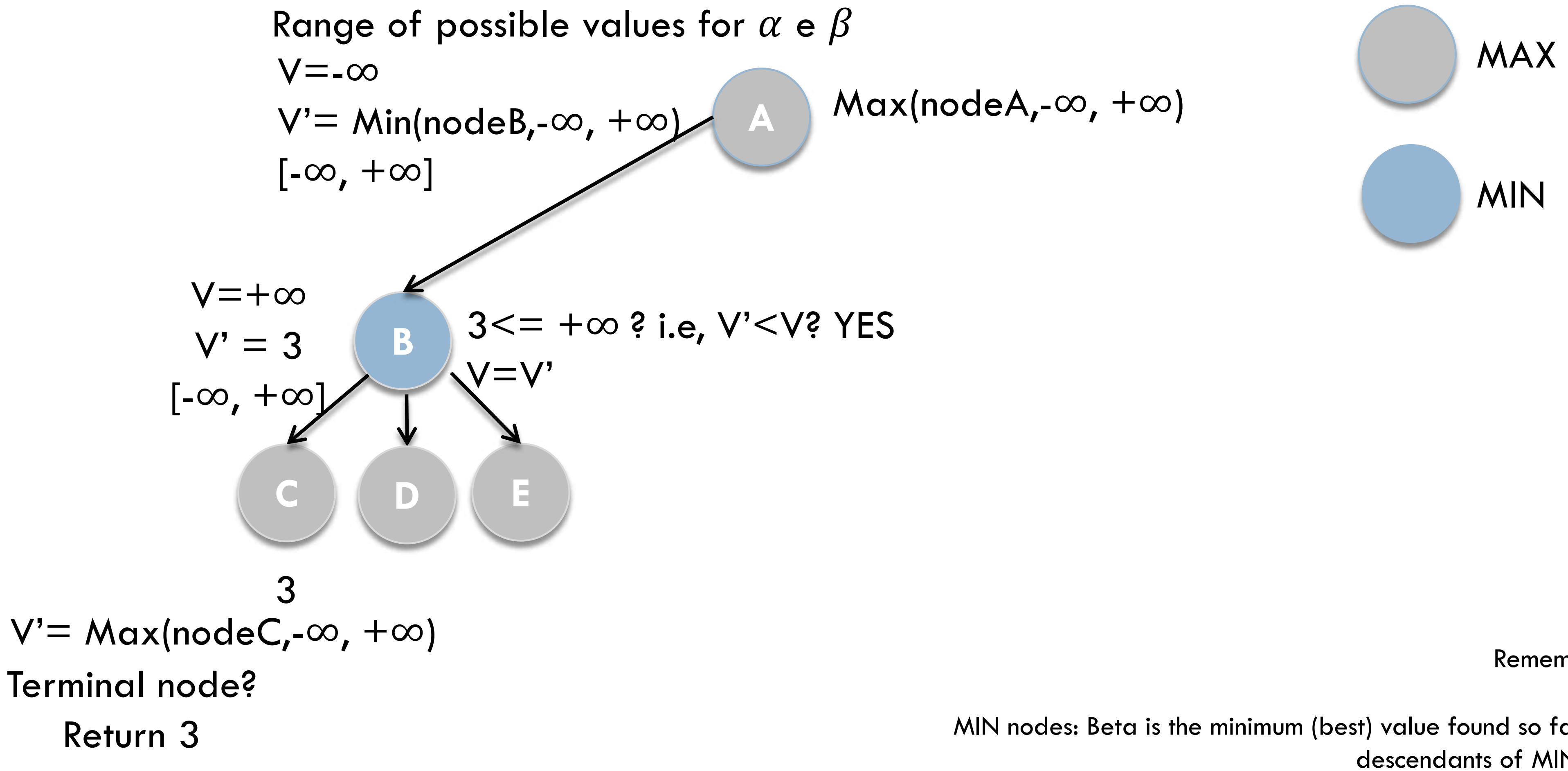
- Alpha-Beta pruning: in most cases, it is not possible to explore the entire search tree
  - Objective: decrease the search space
    - Alpha cut → Based on the minimum score that the maximizing player is assured α cannot decrease (cannot be less than an ancestor)
    - Beta cut → Based on the maximum score that the minimizing player is assured
      - β cannot increase (cannot be greater than an ancestor)
- We don't need to go through the entire tree to know which is the best option for our play
  - Idea: it's not worth getting worse, if you've found something better
  - Exact values don't matter

Alpha-Beta Cut

- Initially, the game tree is traversed in order of depth
- For each node other than a leaf, a value is stored, being:
  - Alpha: for MAX nodes
    - It is the maximum (best) value found so far in the descendants of the MAX nodes
  - Beta: for MIN nodes
    - It is the minimum (best) value found so far in the descendants of the min nodes
  - If:
    - If it is a MAX node:
      - If alpha_value(node) >= value_beta_ancestor;
    - Or If it is a MIN node:
      - If value_beta(node) <= value_alpha_ancestor;
    - THEN cut_search_down(node)

Alpha-Beta Cut

- Search the first deepest to the first leaf



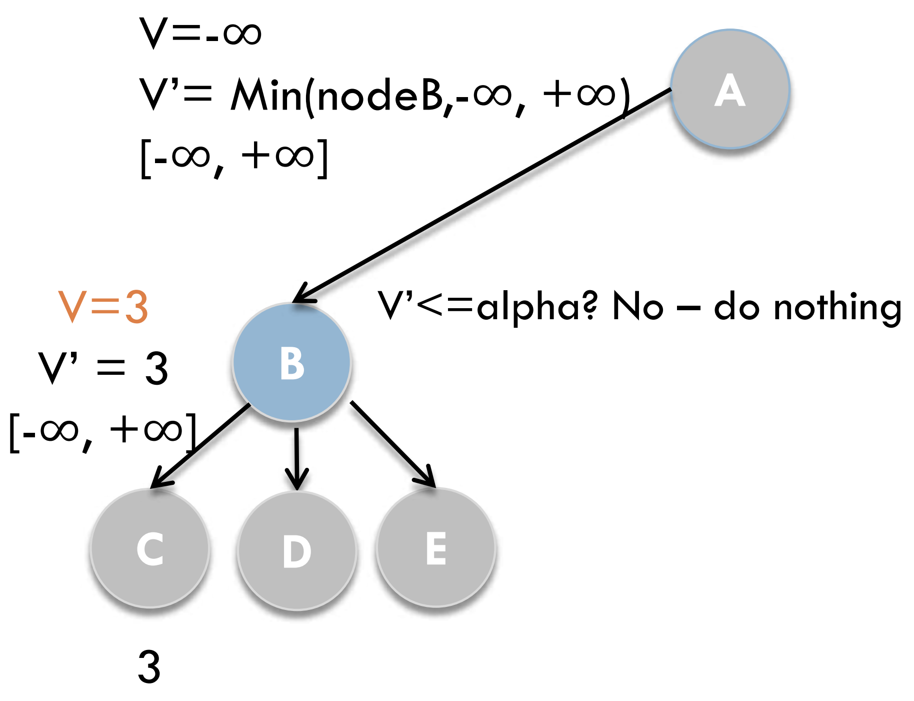
Remembering:

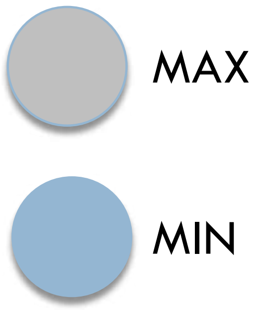
MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf





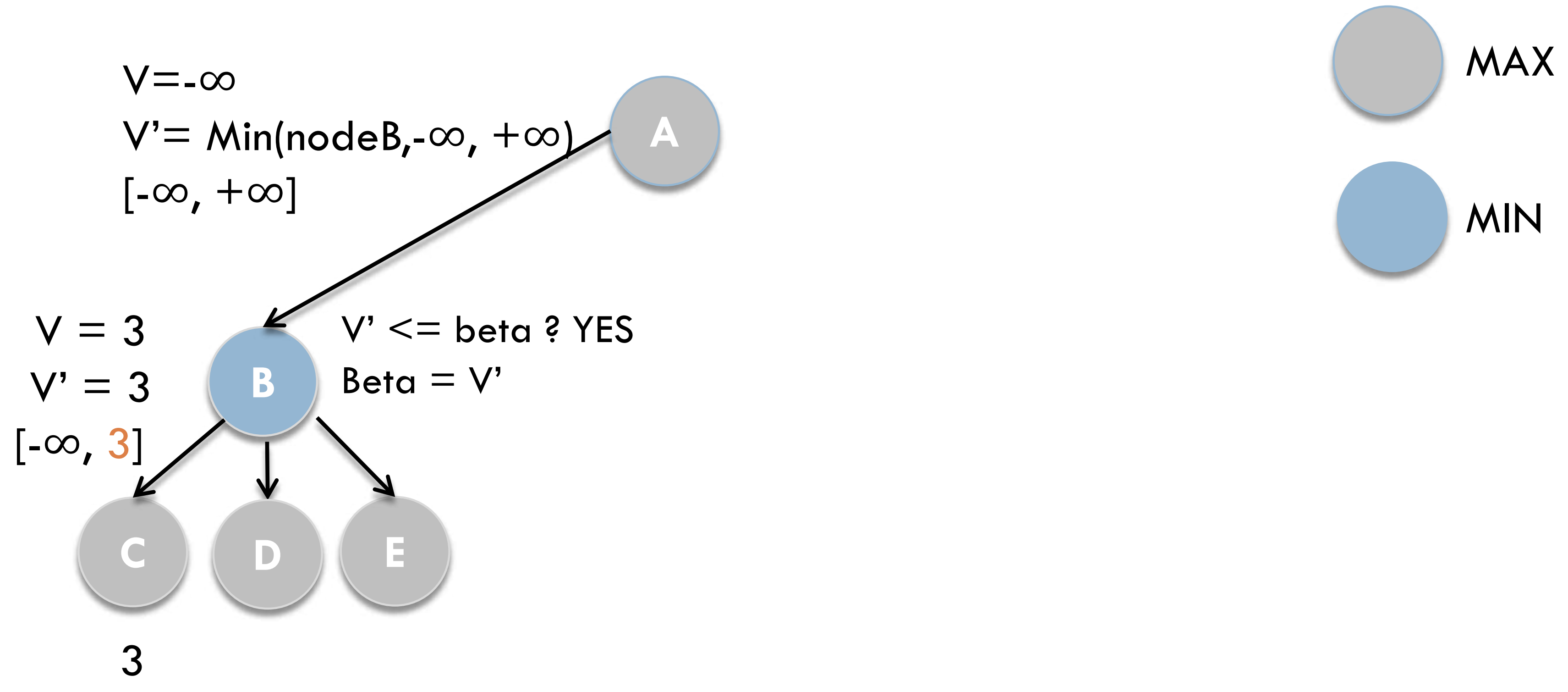
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



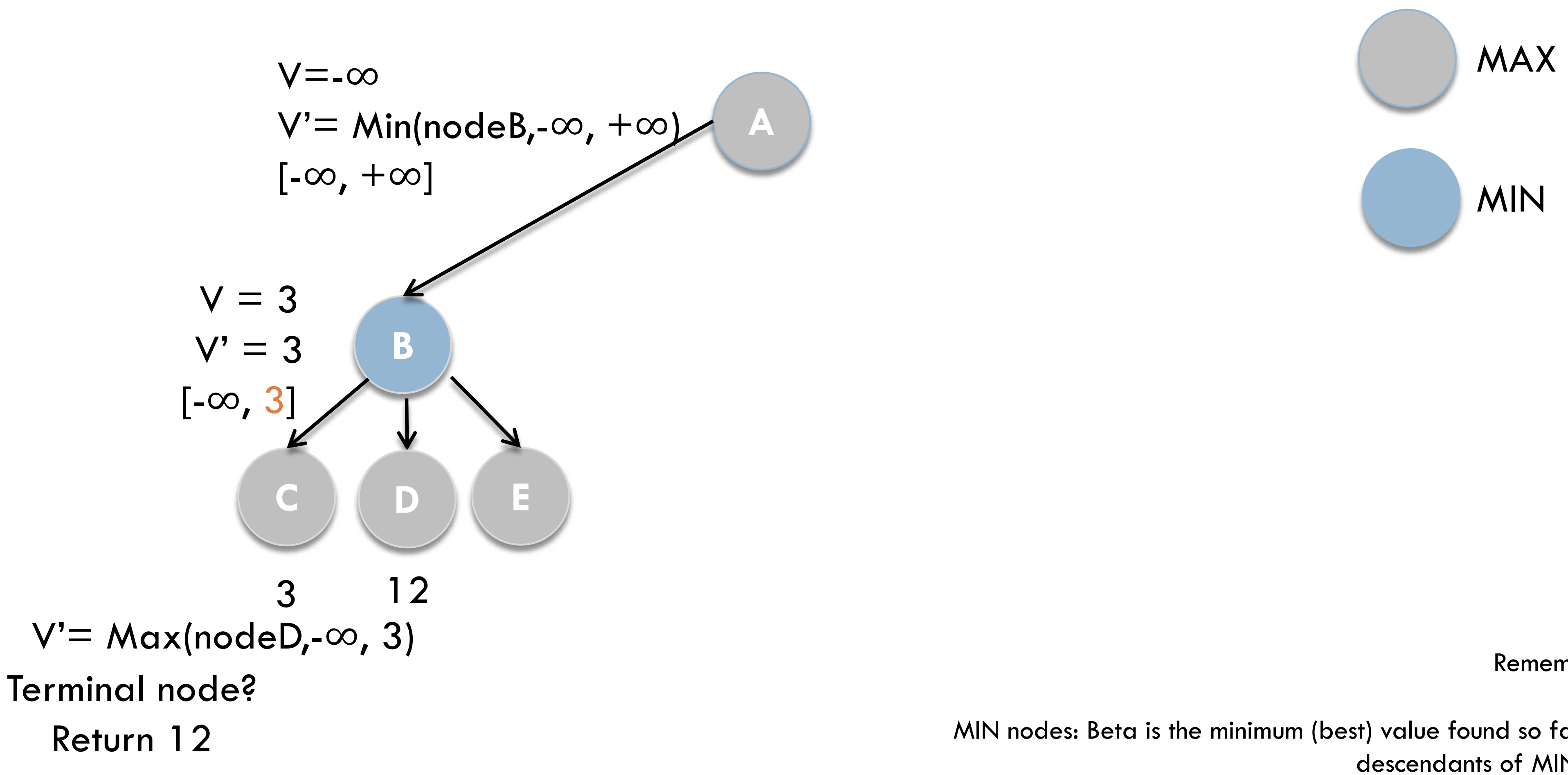
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



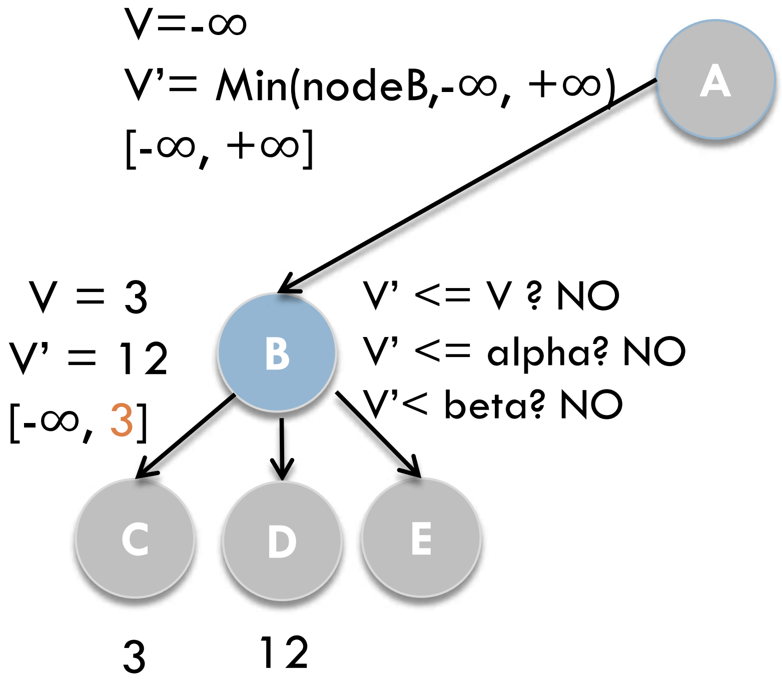
Remembering:

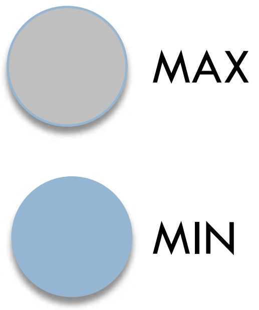
MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf





V'= Max(nodeD,-∞, 3)

Terminal node?

Return 12

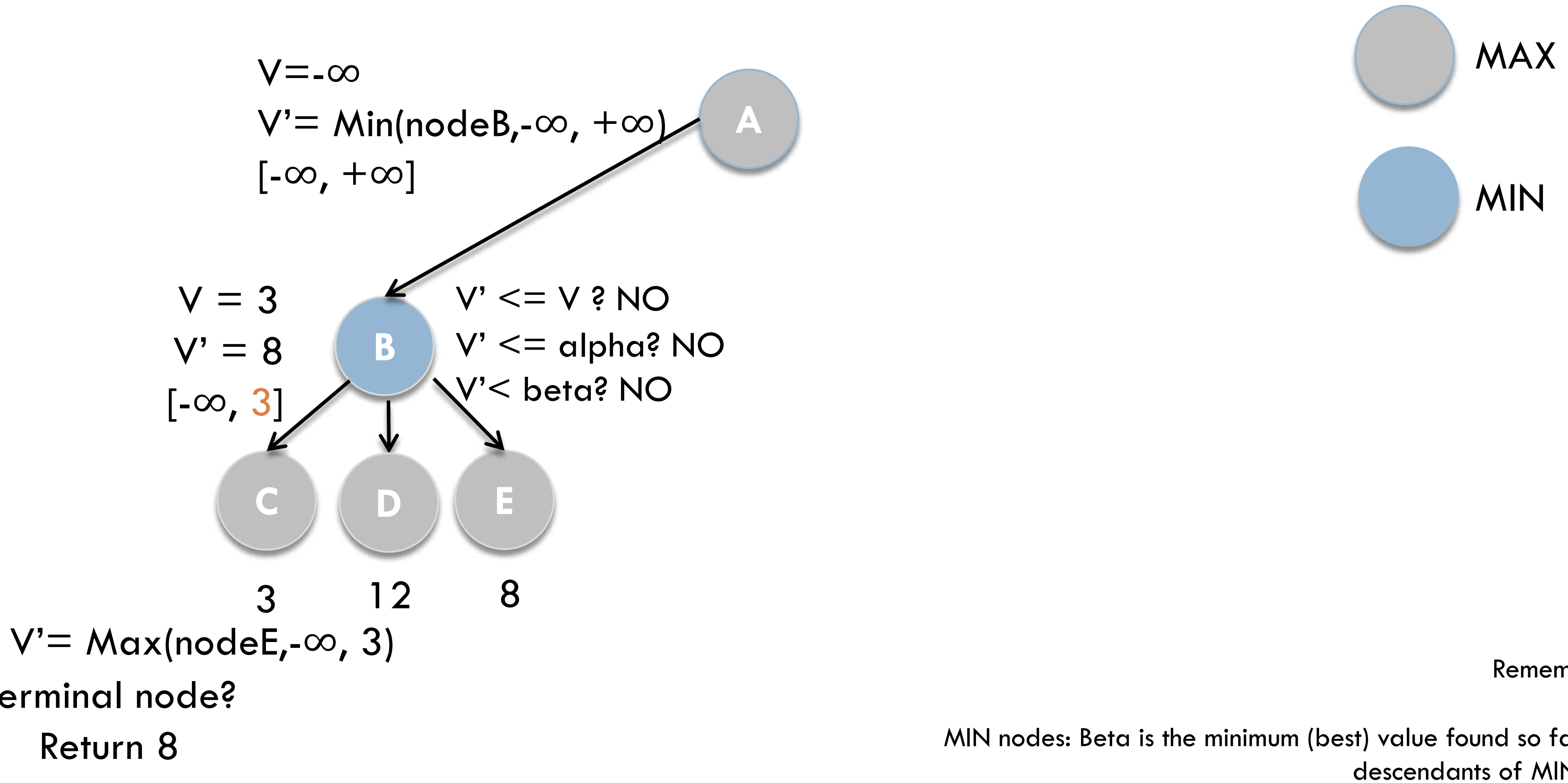
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



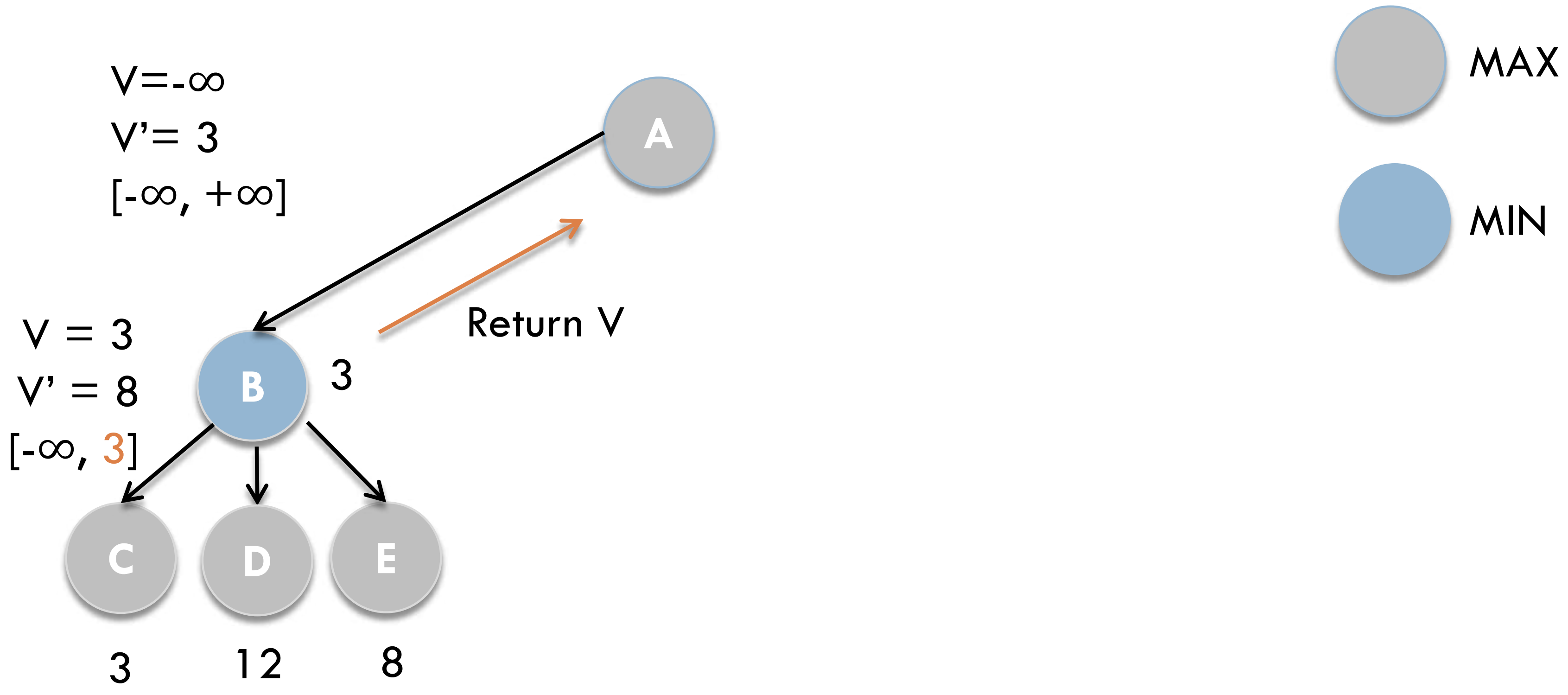
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



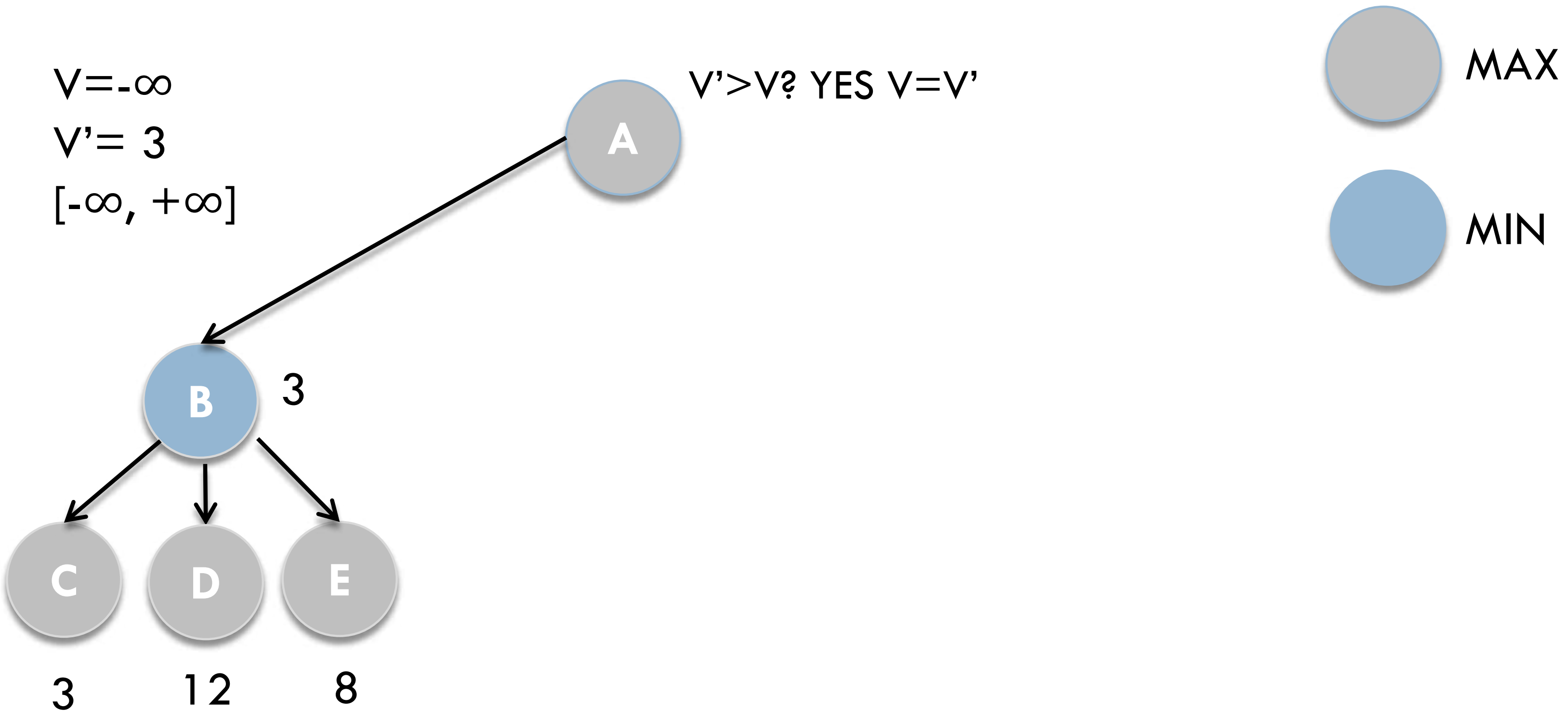
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



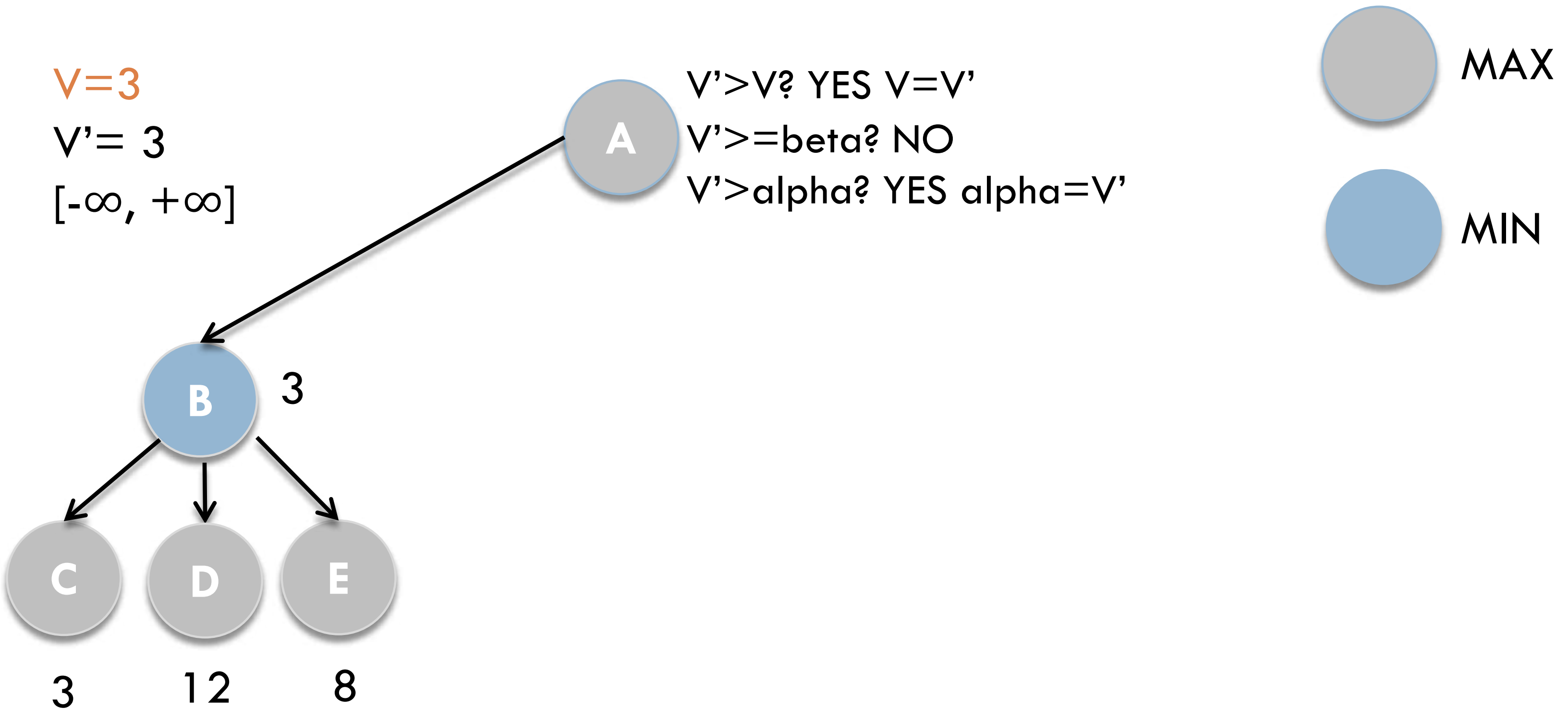
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



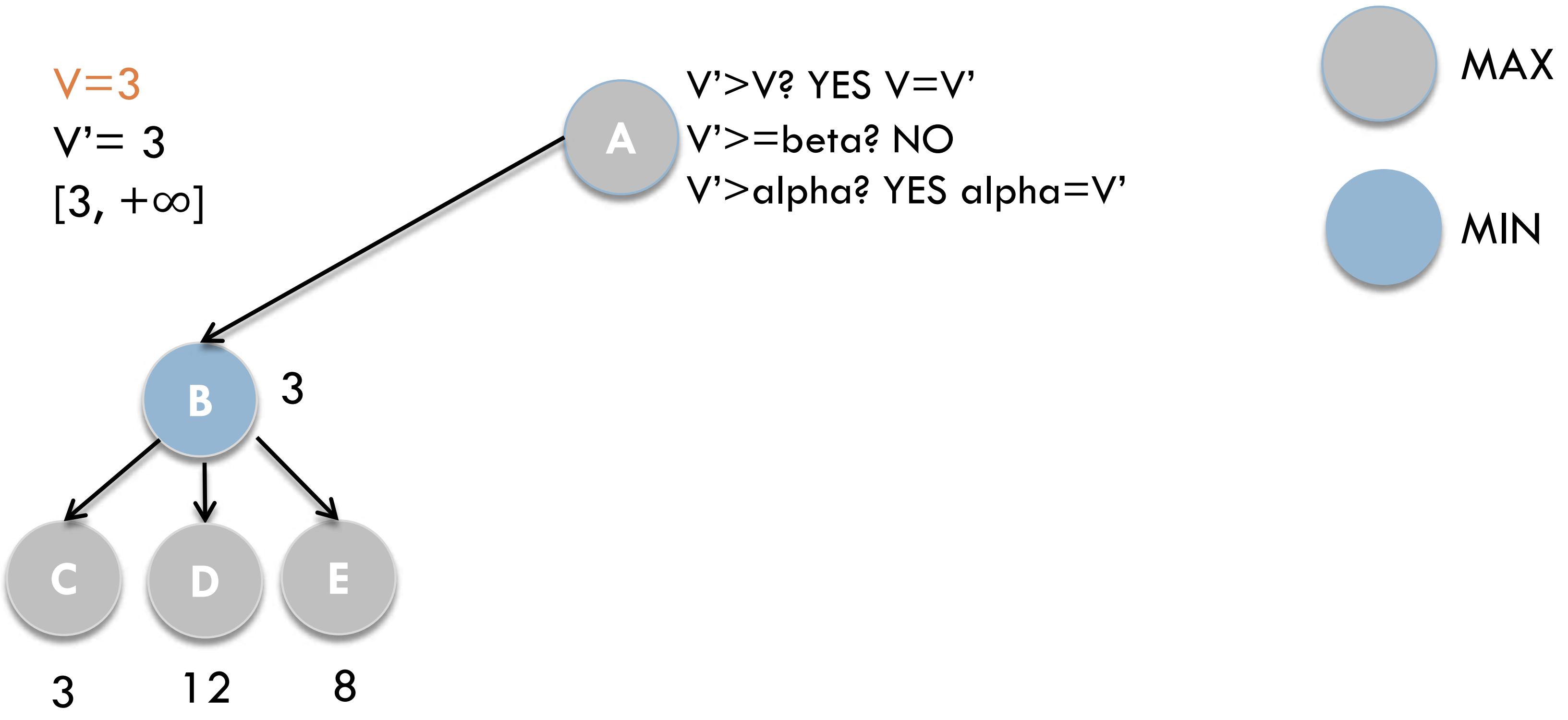
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



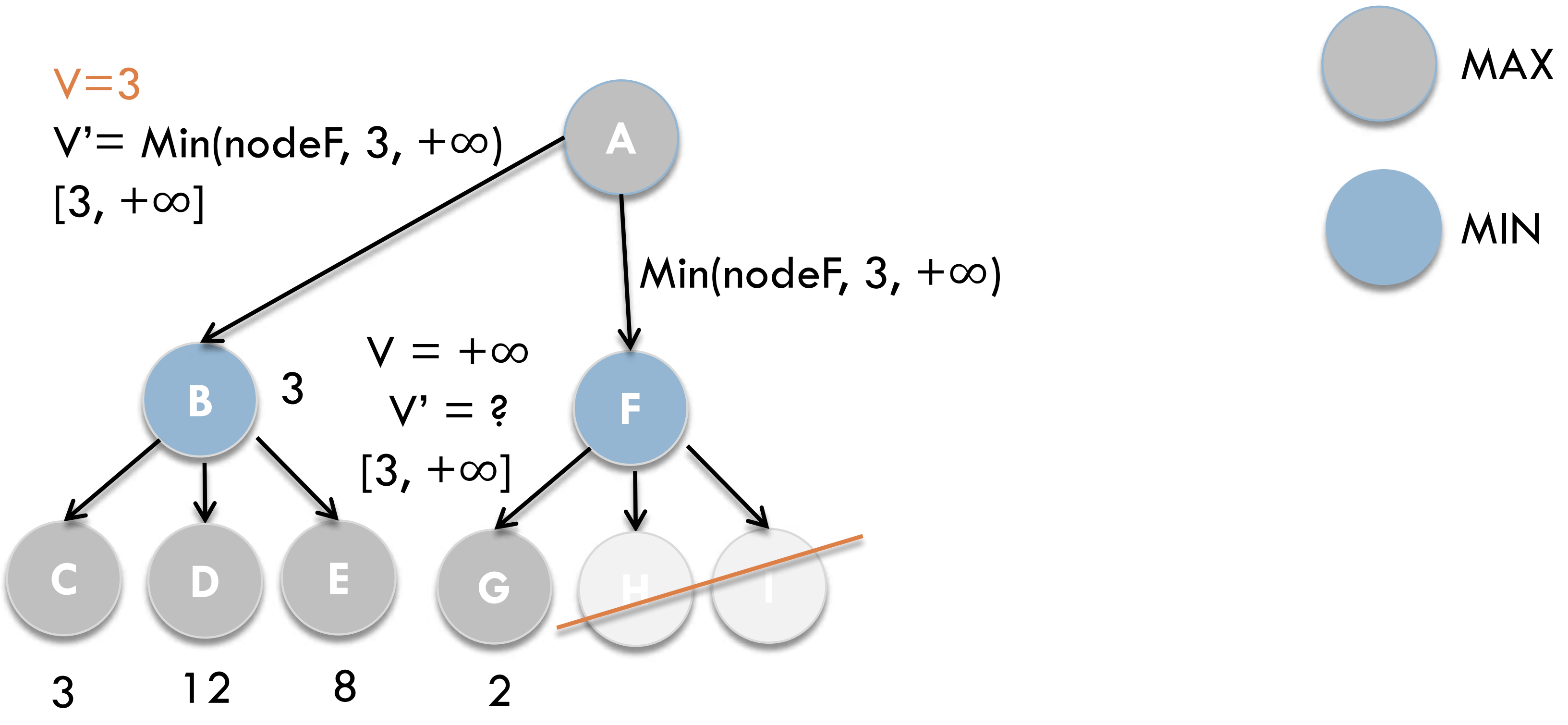
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



V'= Max(nodeG, 3, +∞)

Terminal node?

Return 2

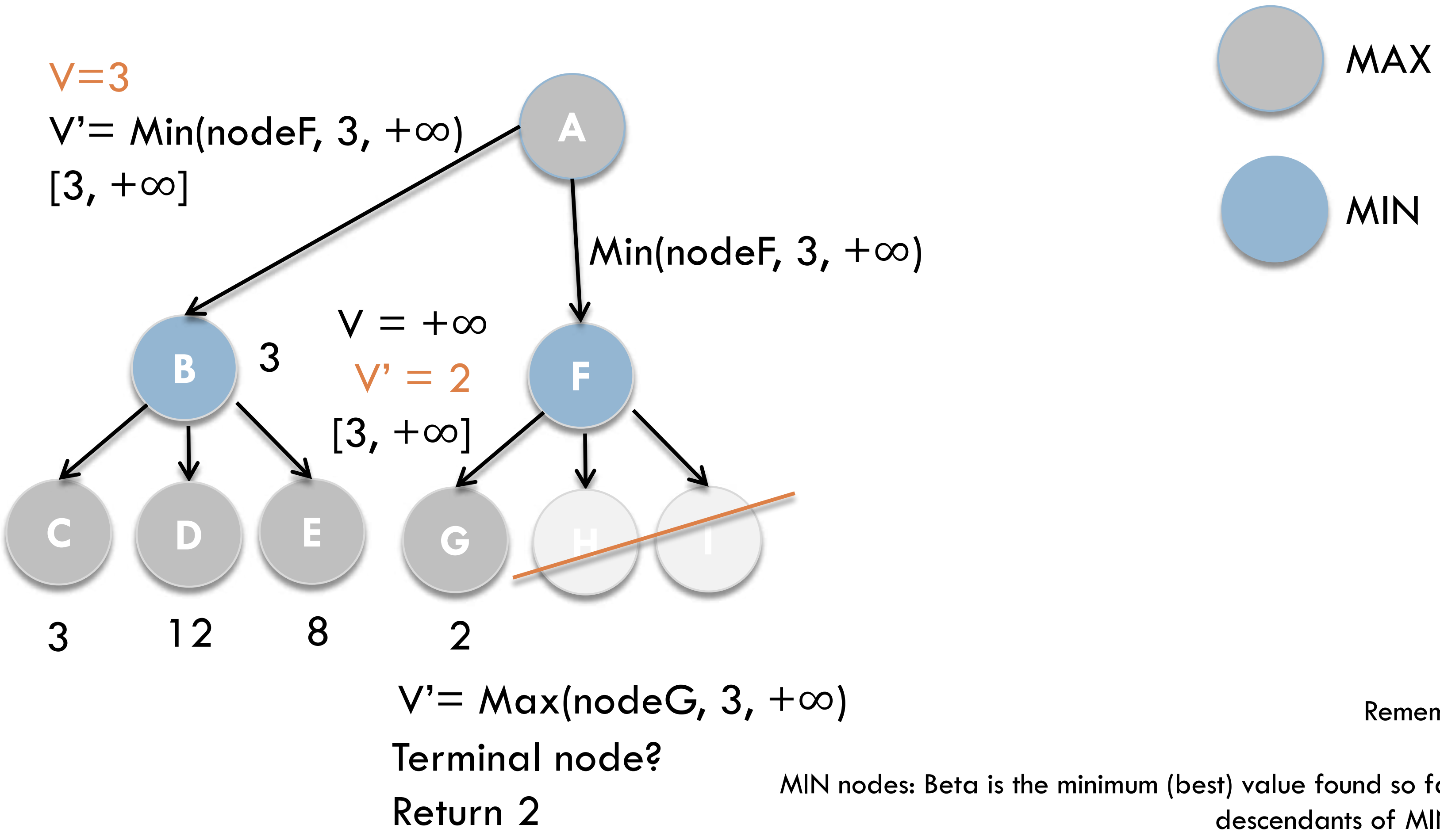
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



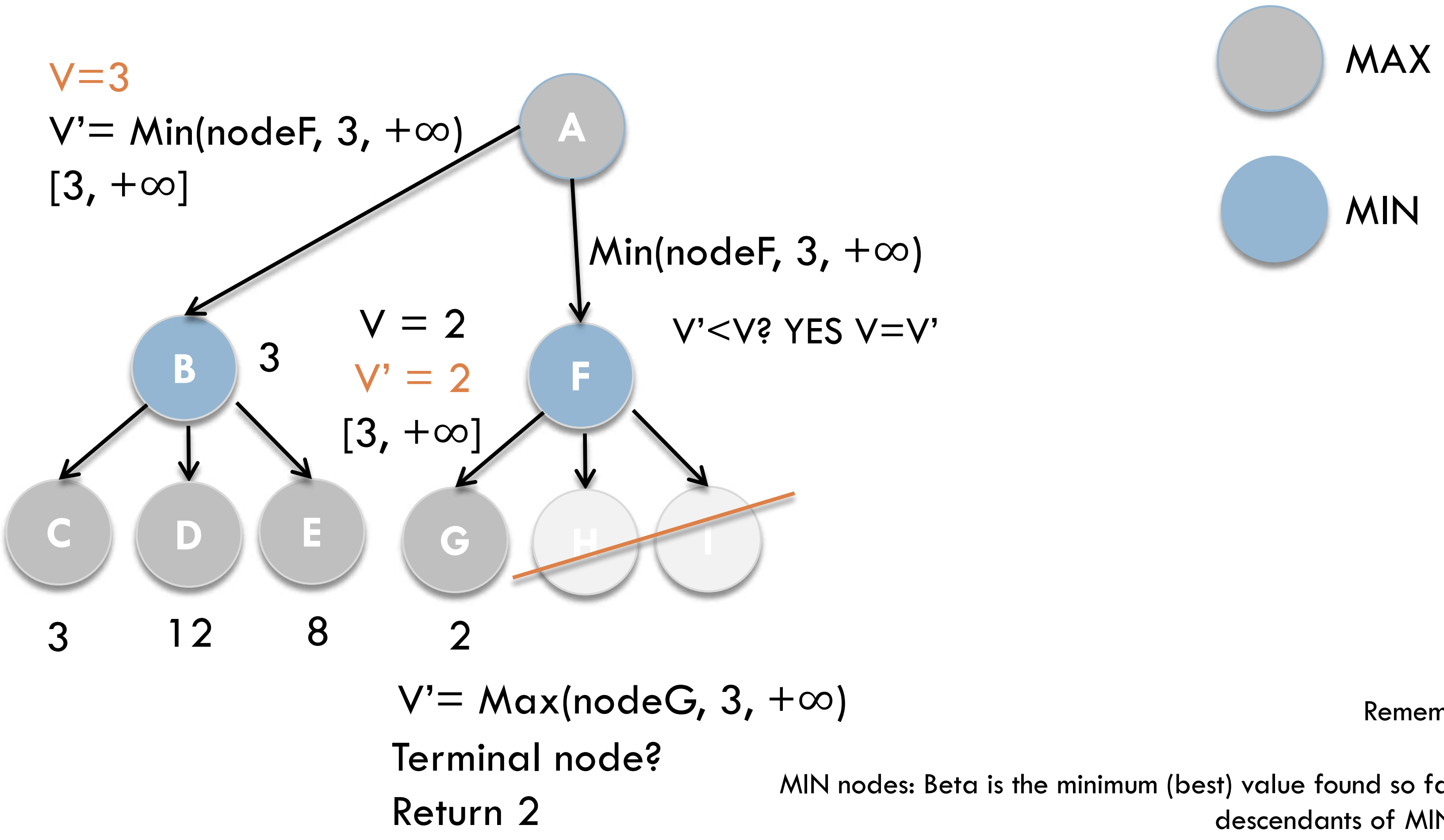
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



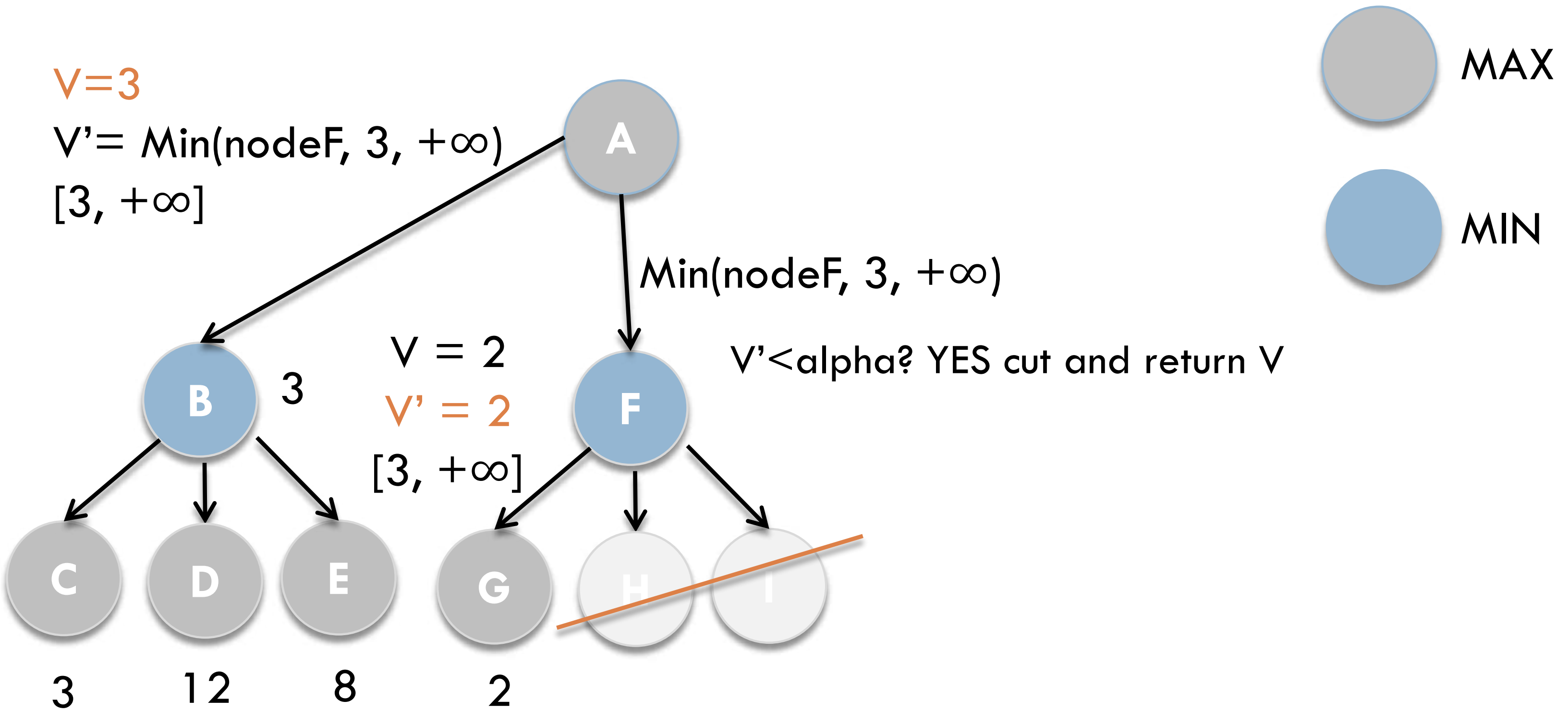
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



V'= Max(nodeG, 3, +∞)

Terminal node?

Return 2

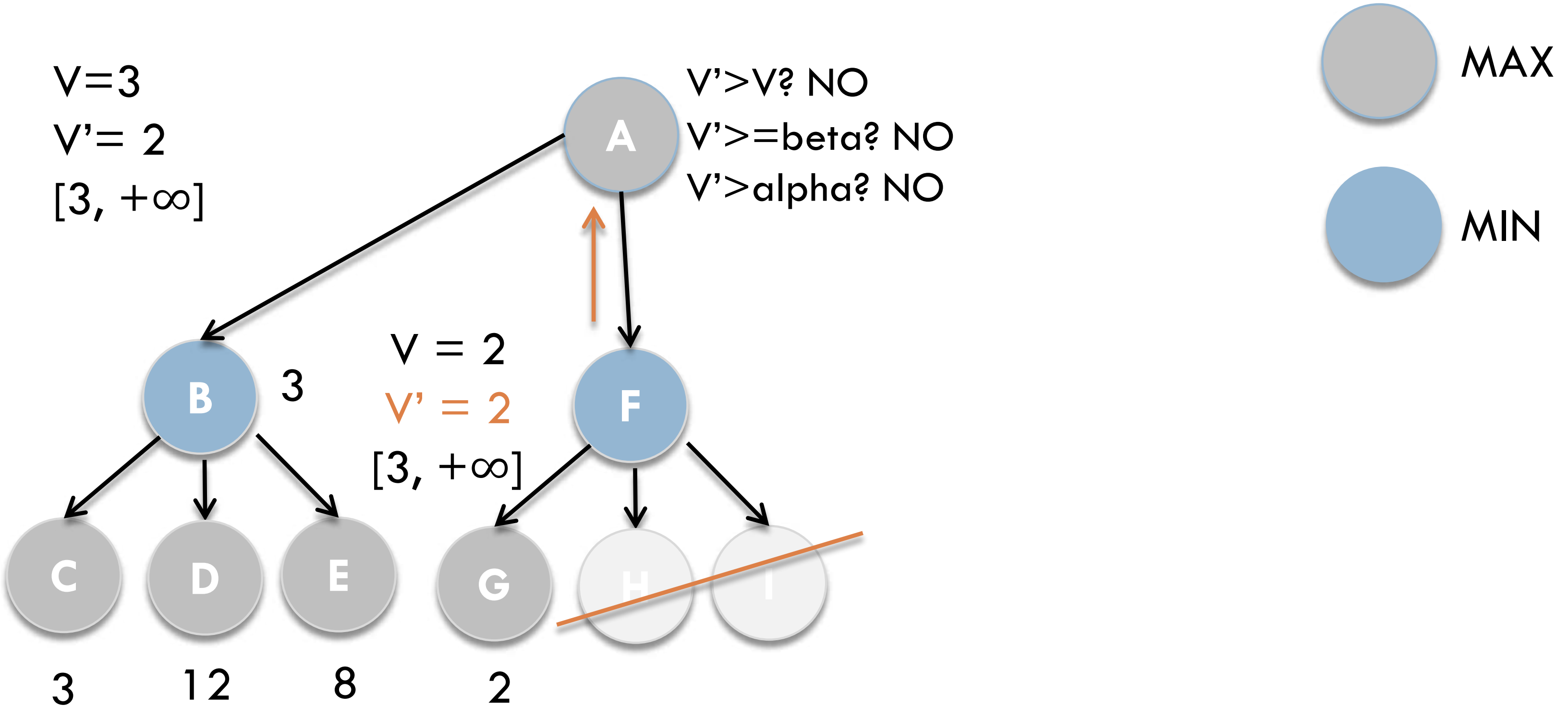
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



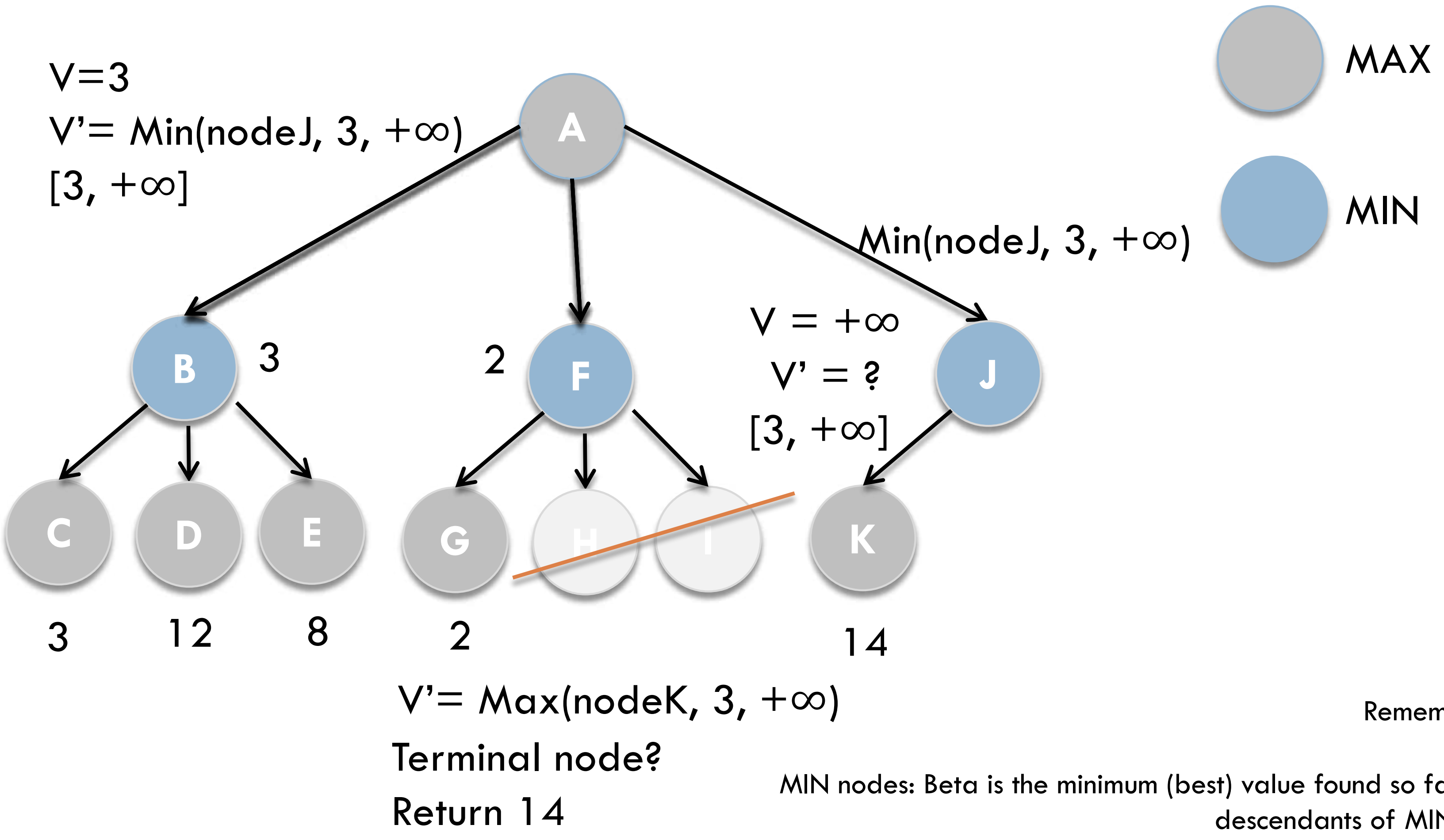
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



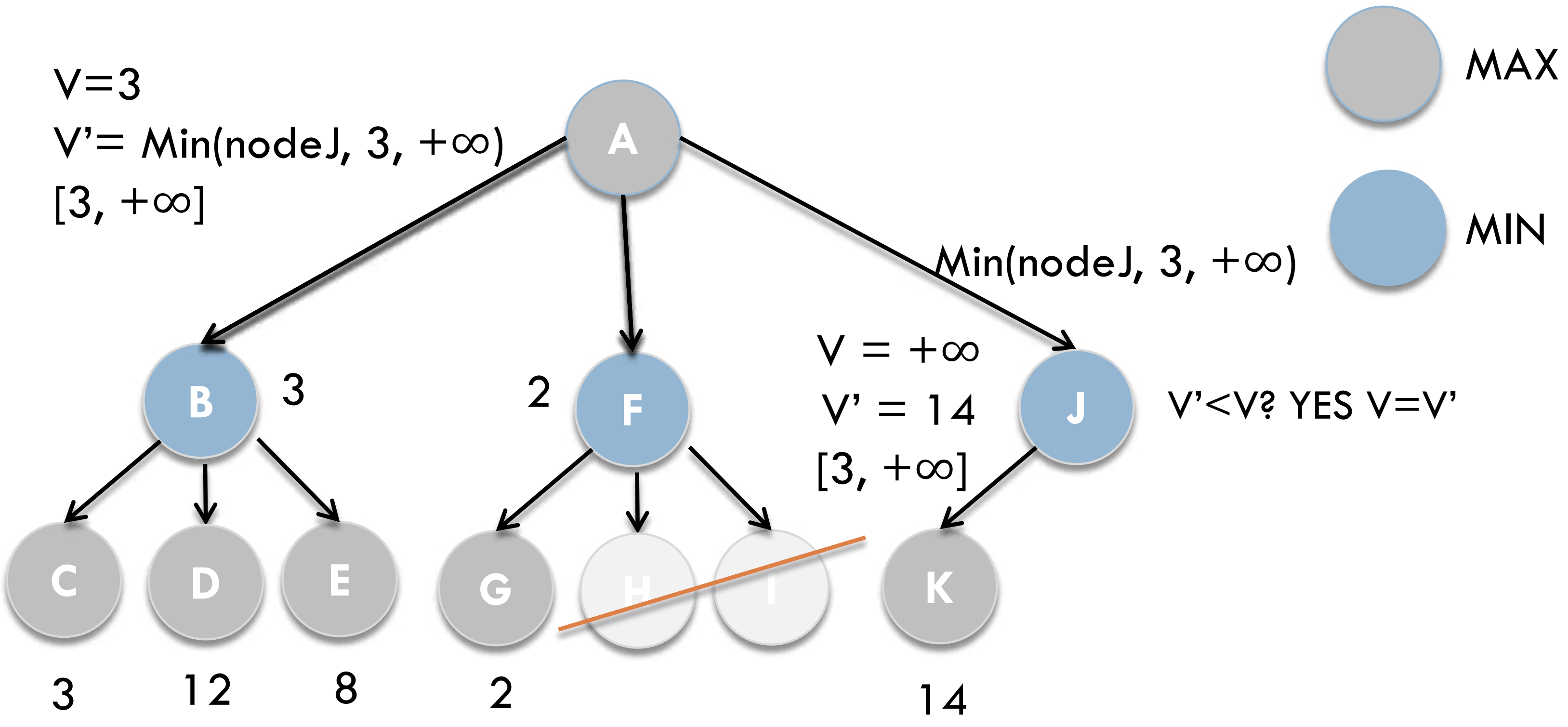
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



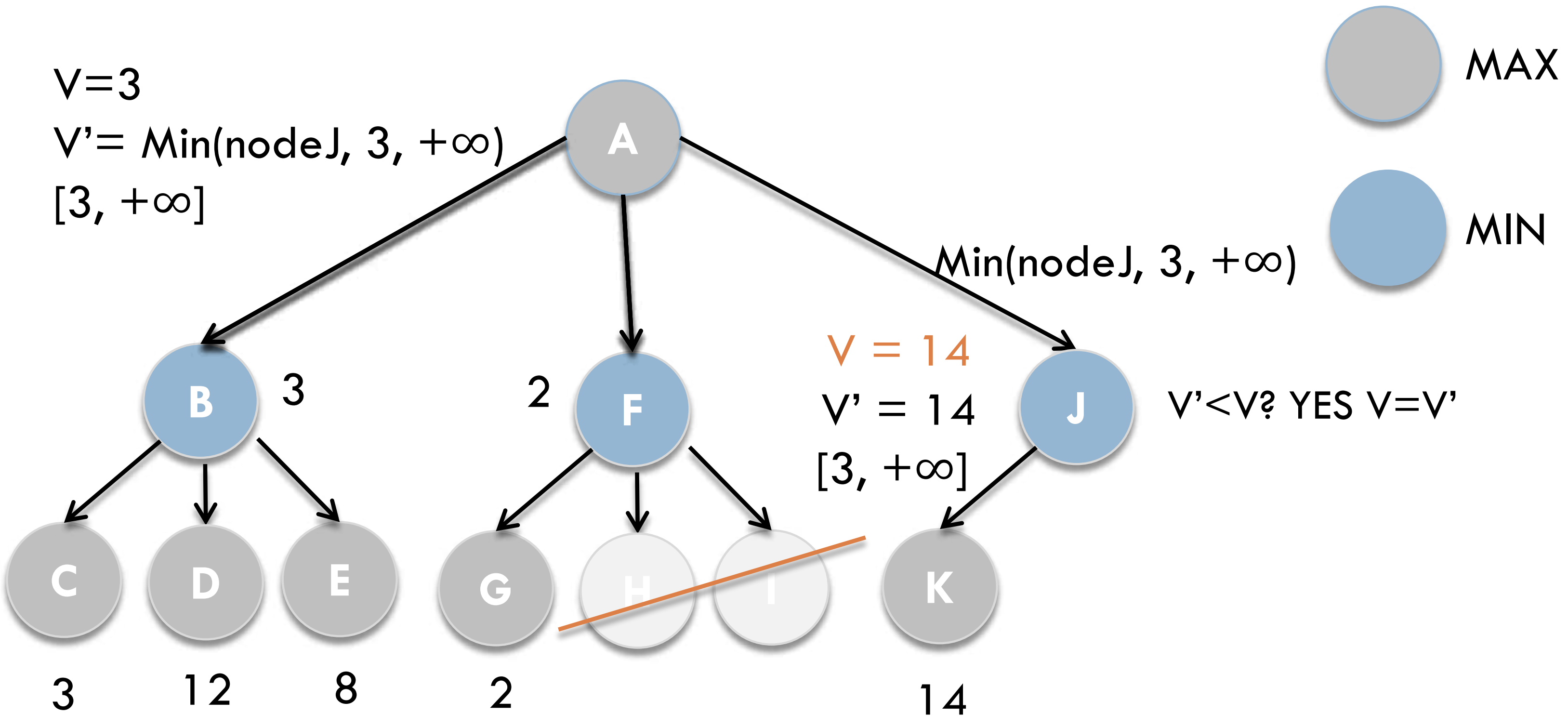
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



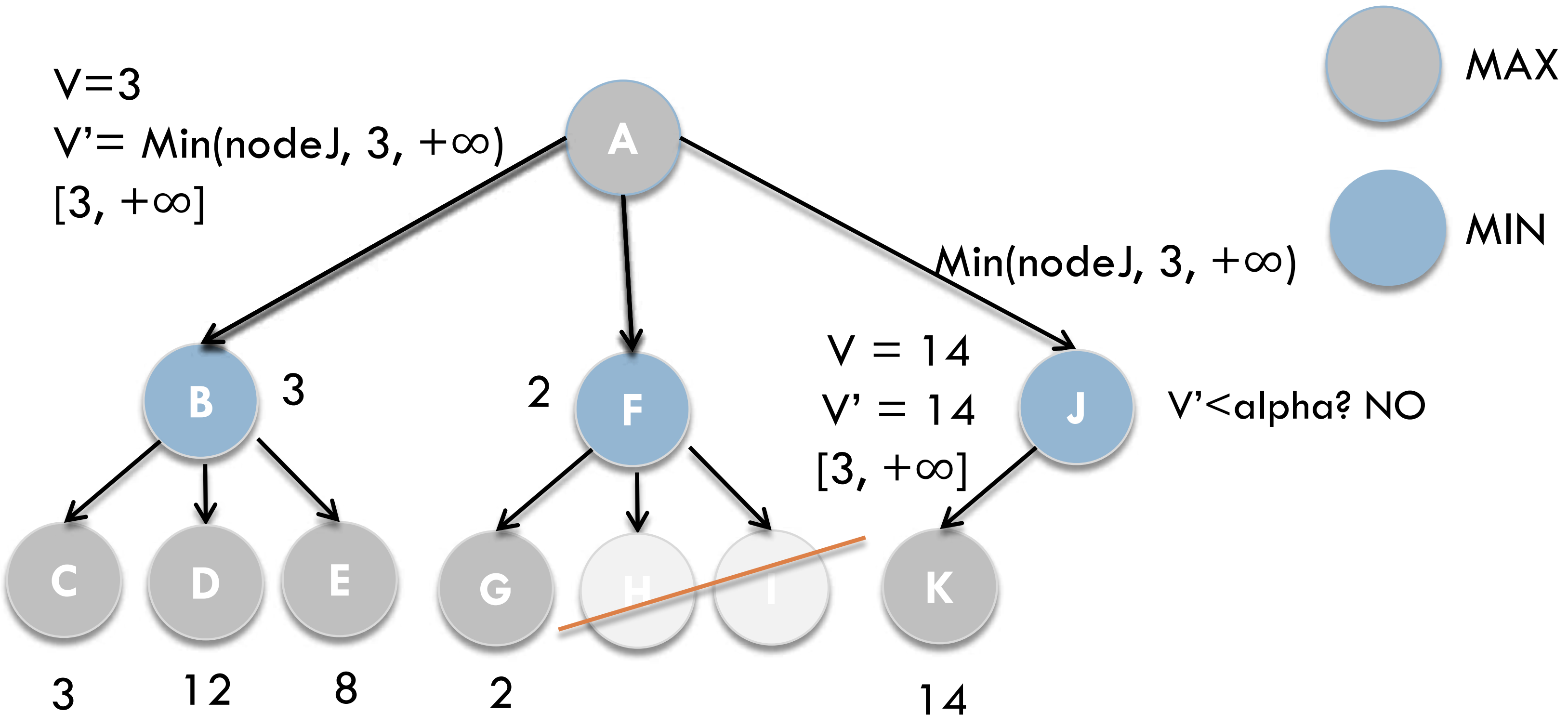
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



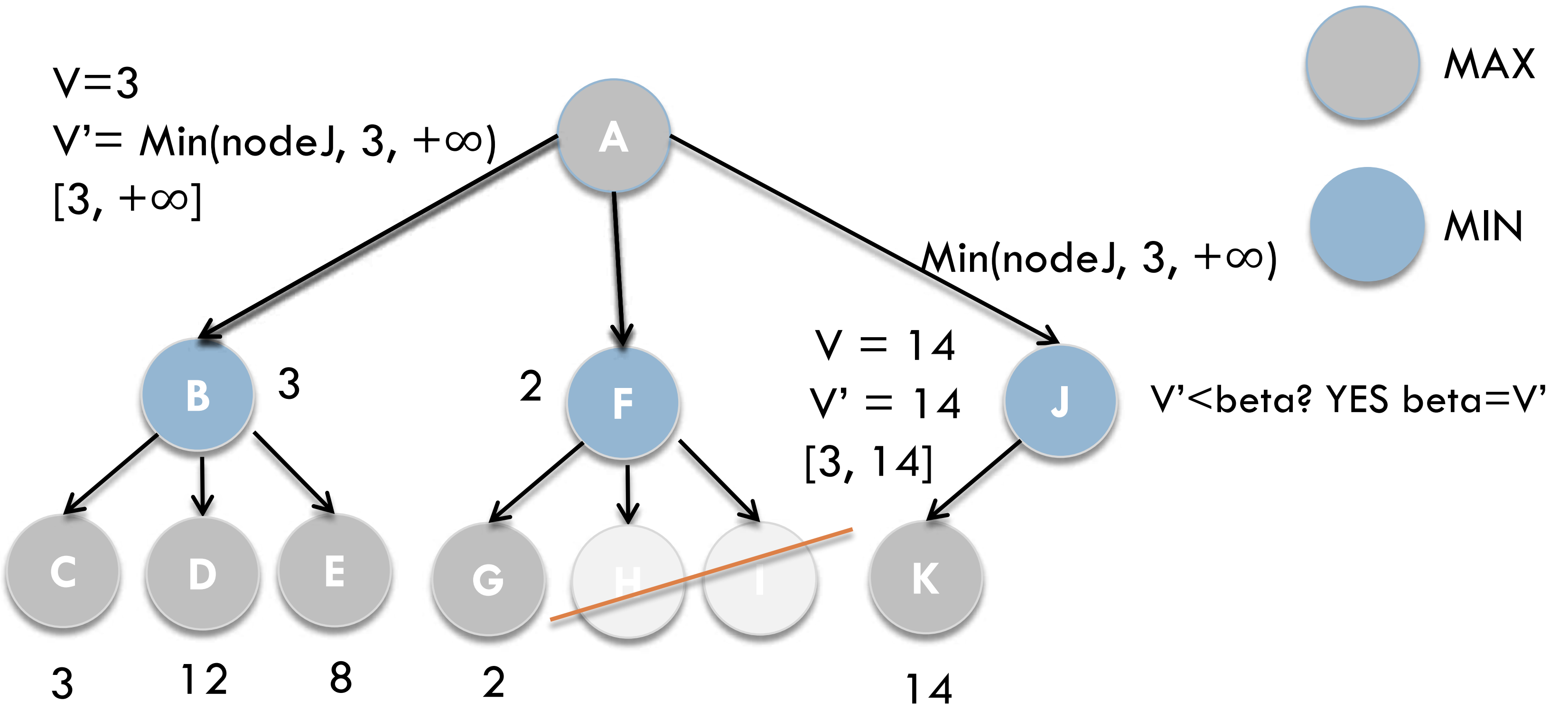
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



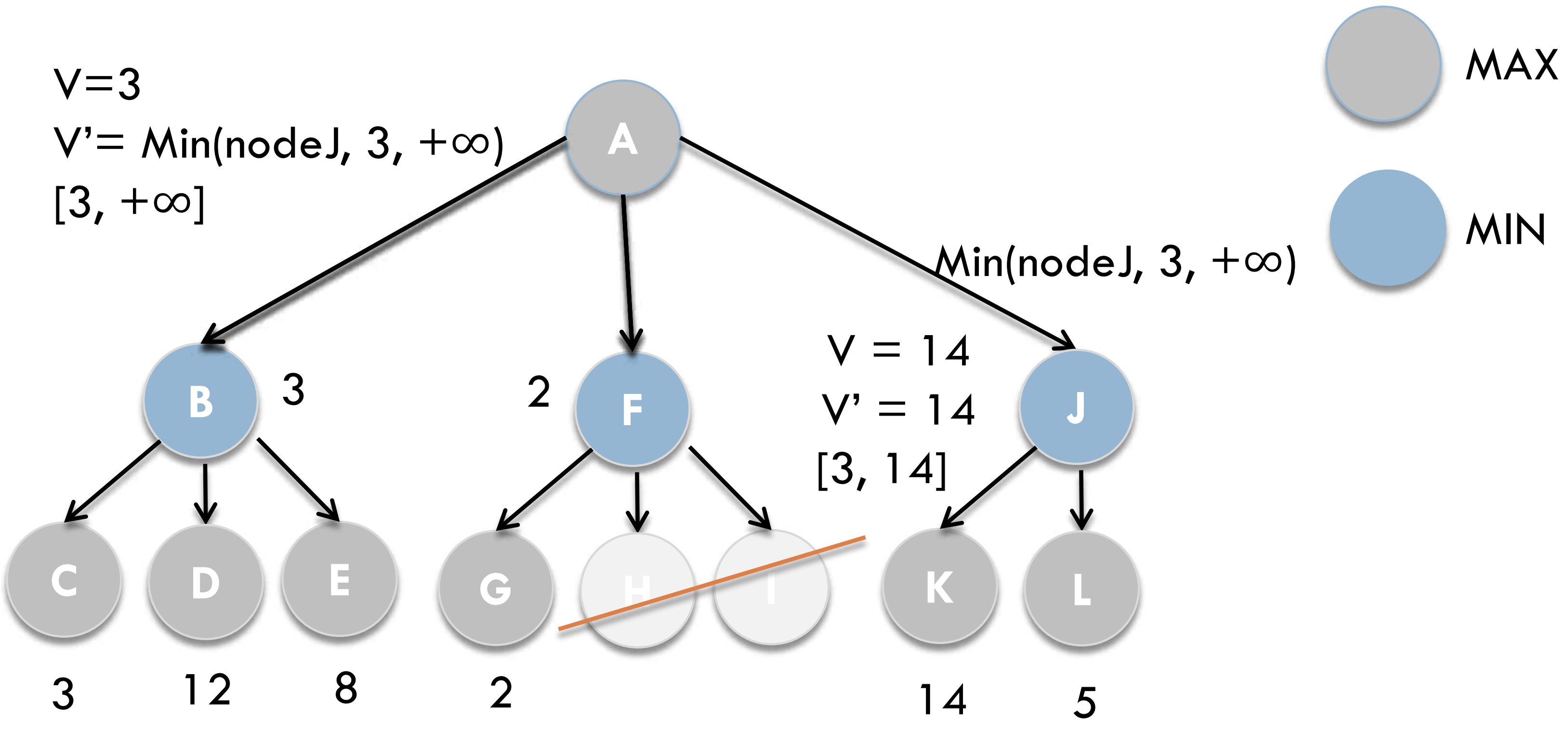
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



V'= Max(nodeL, 3, 14)
Terminal node?
Return 5

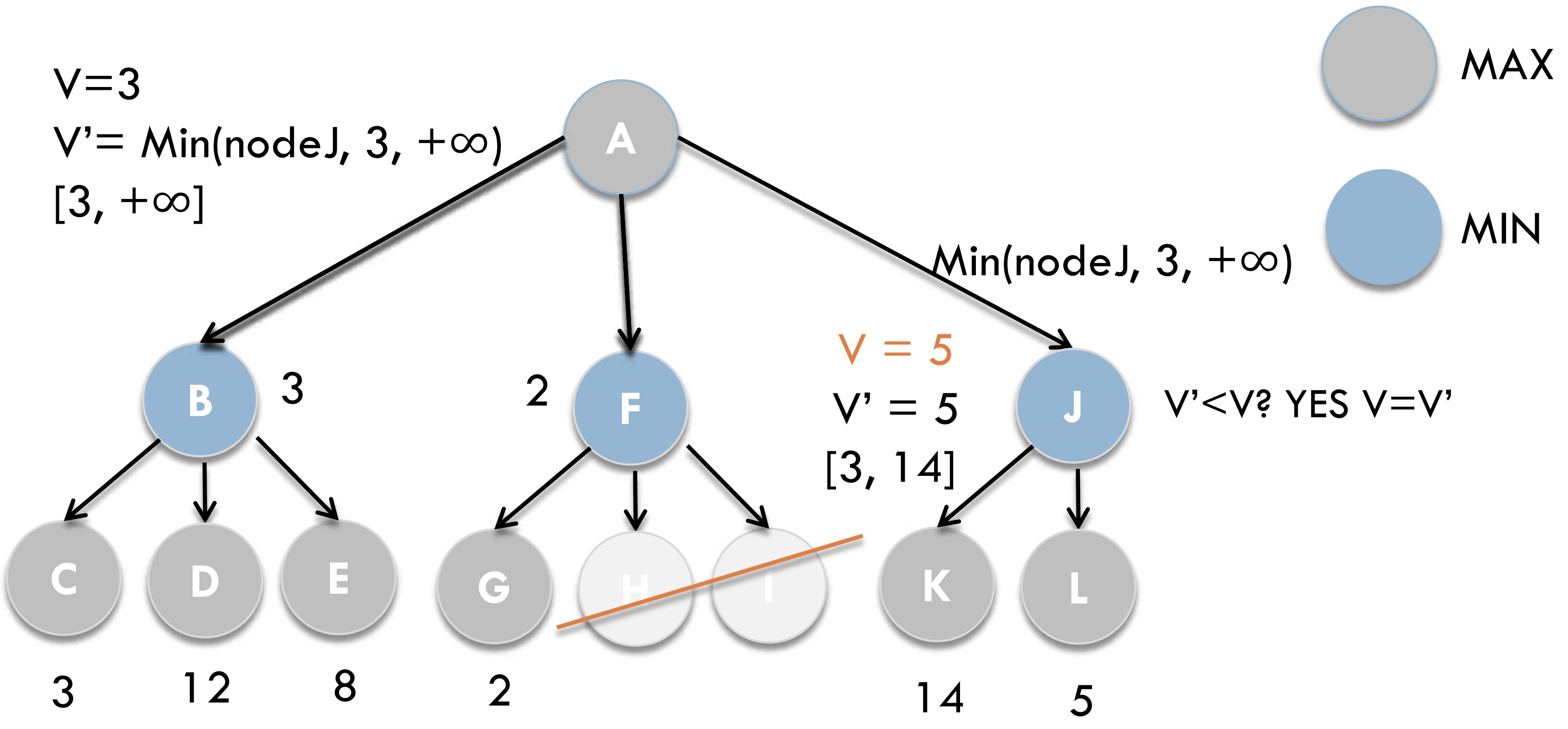
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



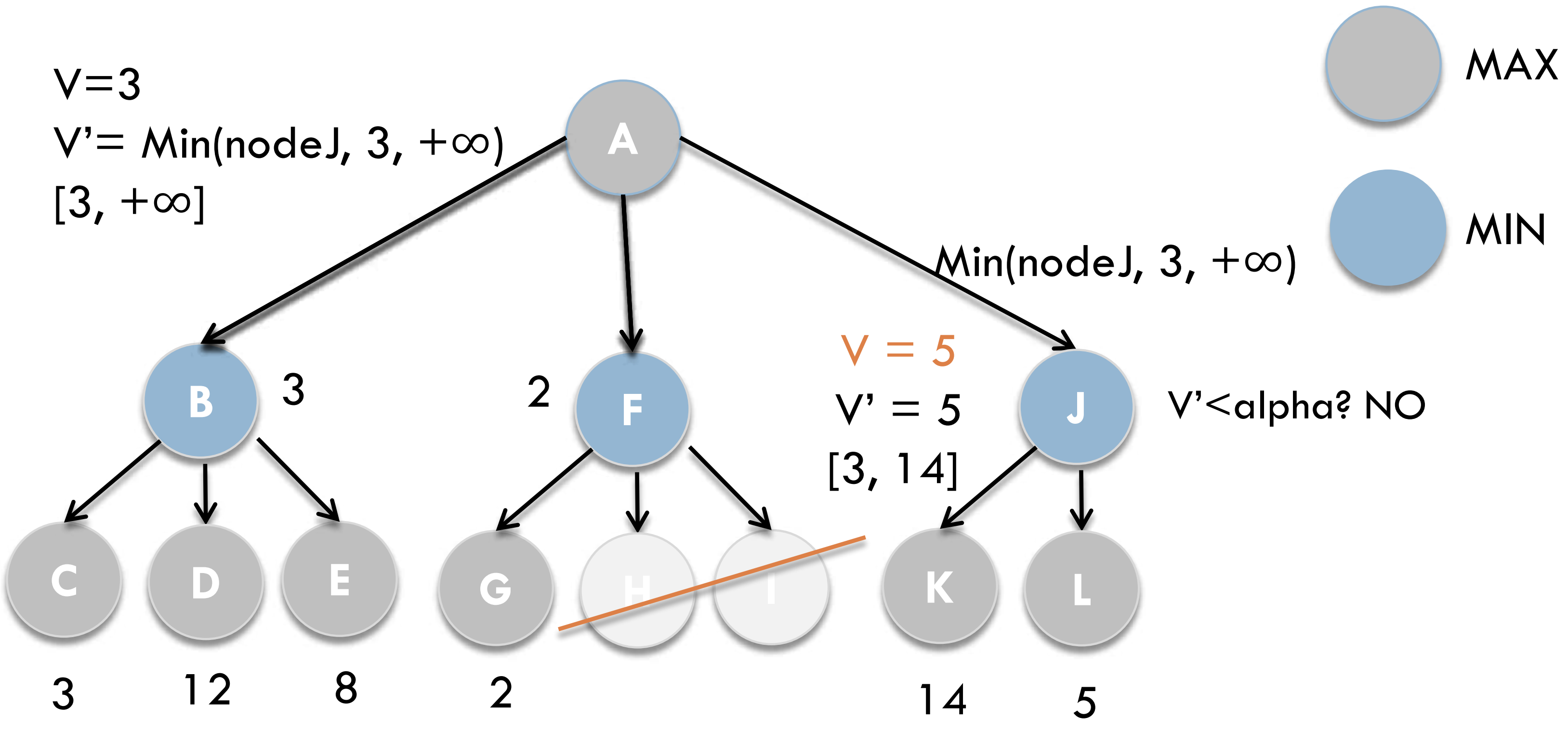
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



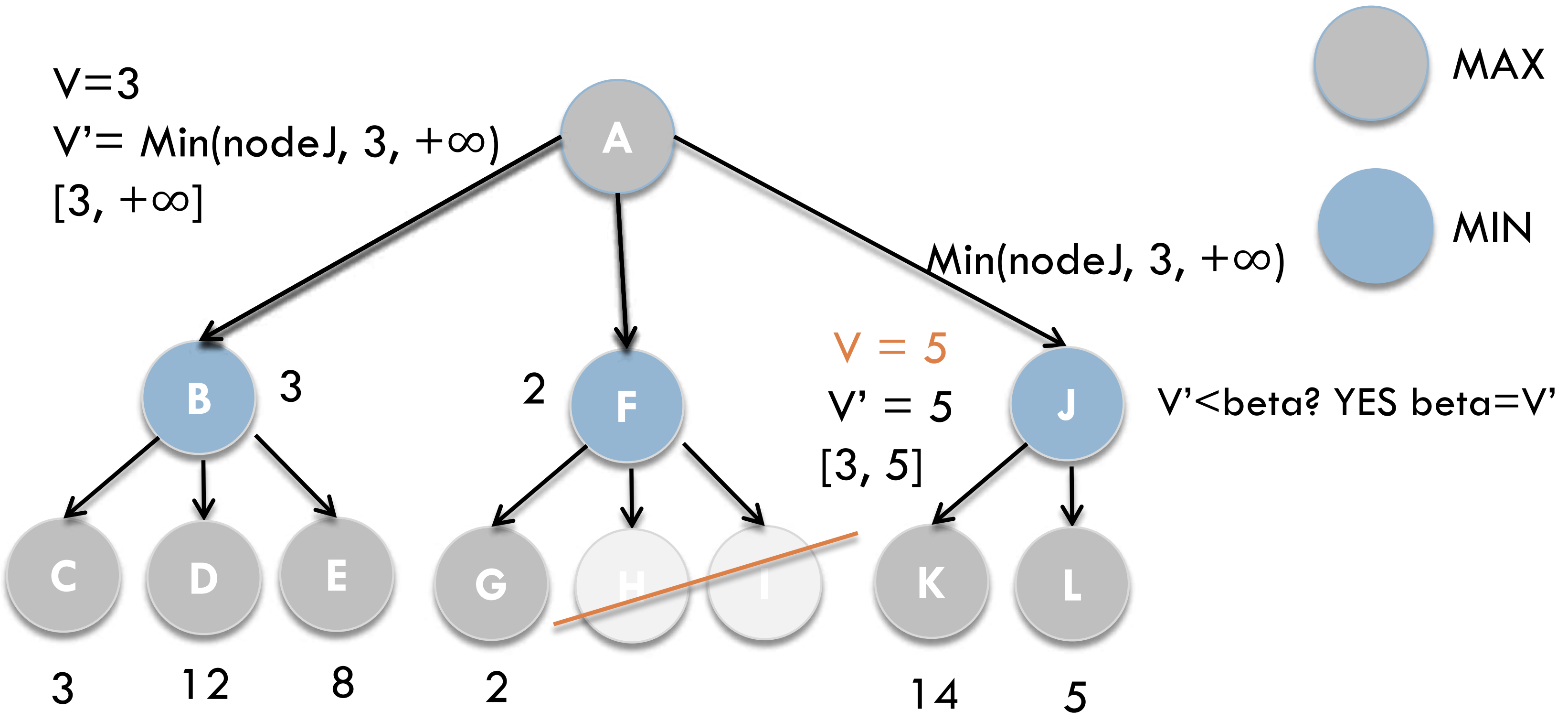
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



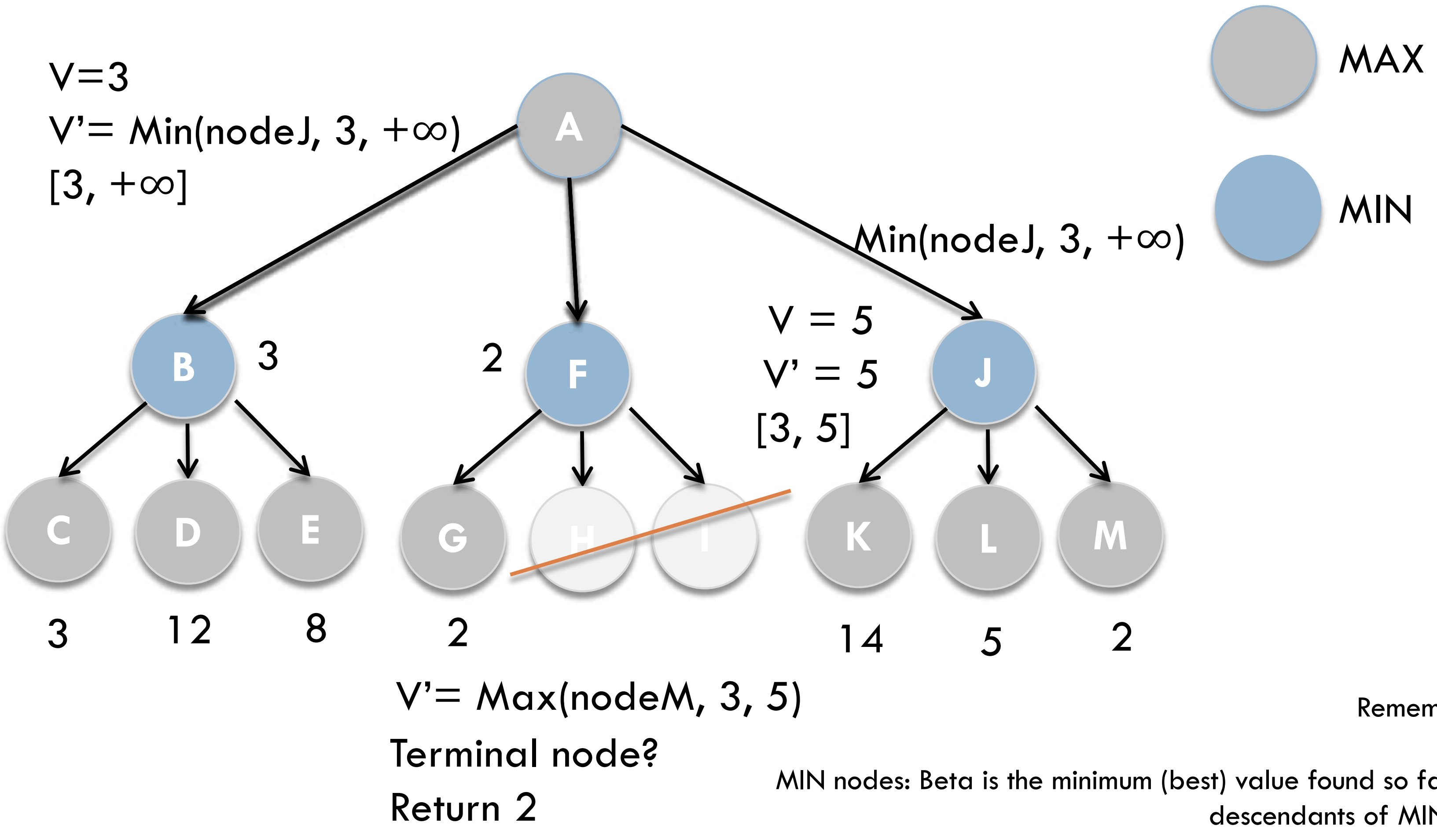
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



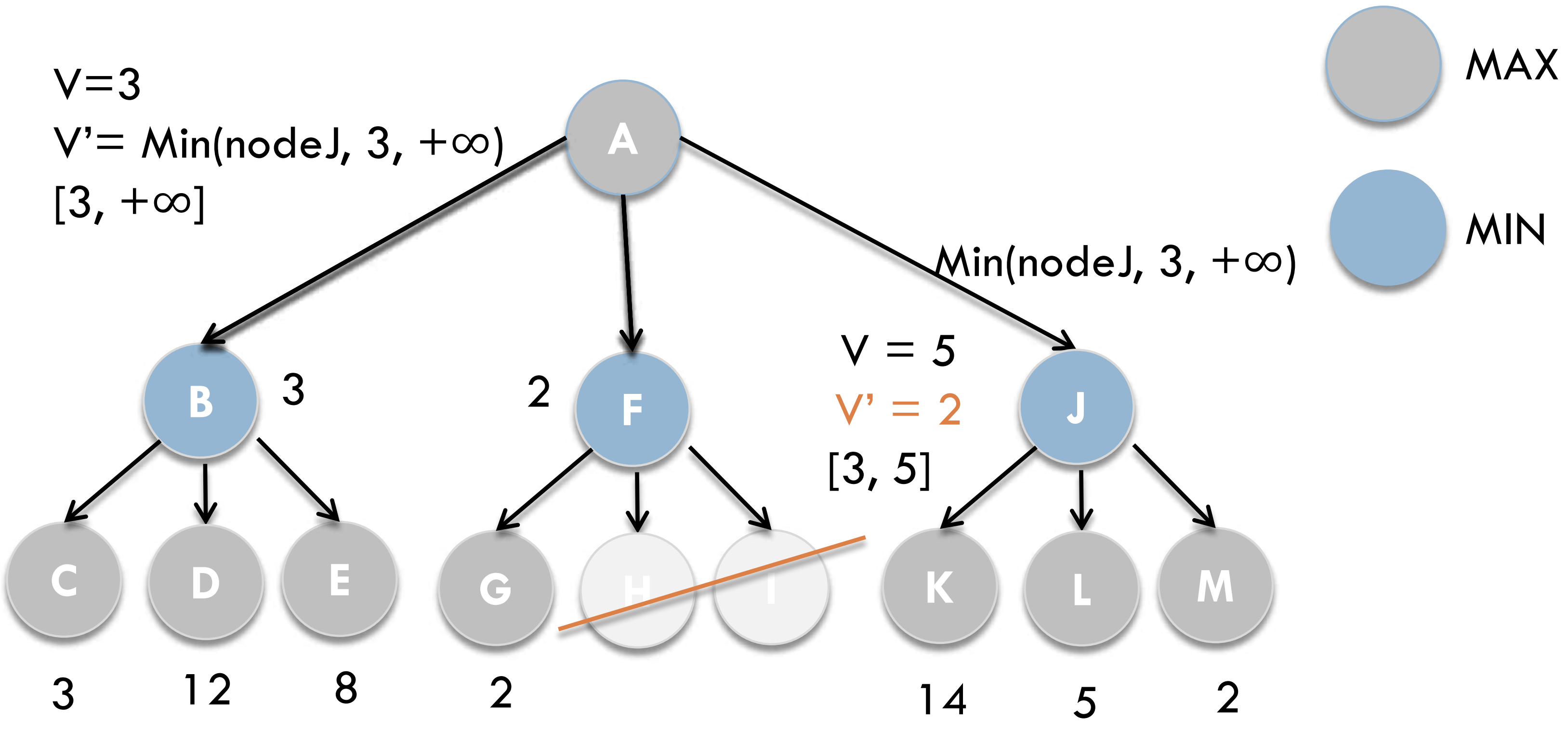
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



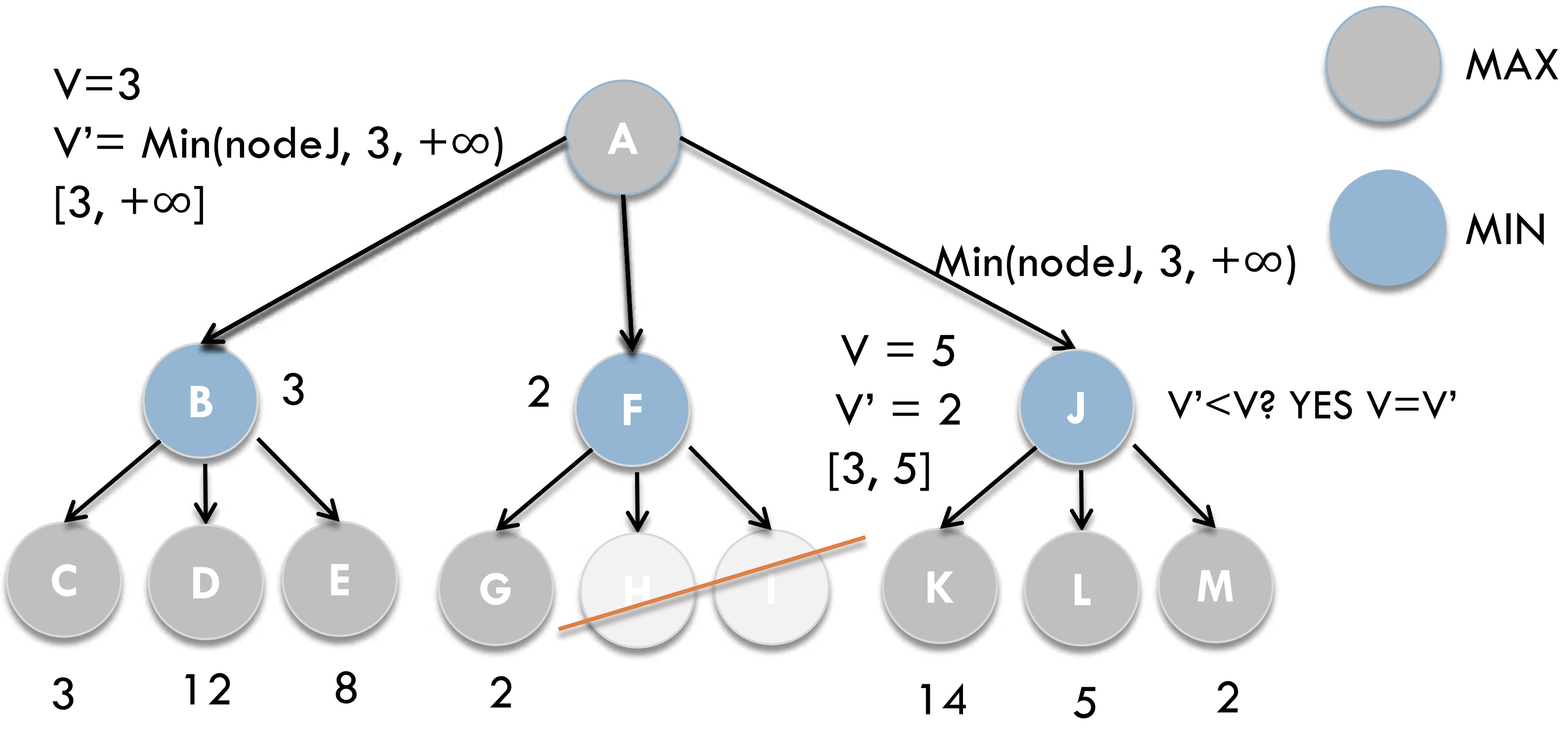
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



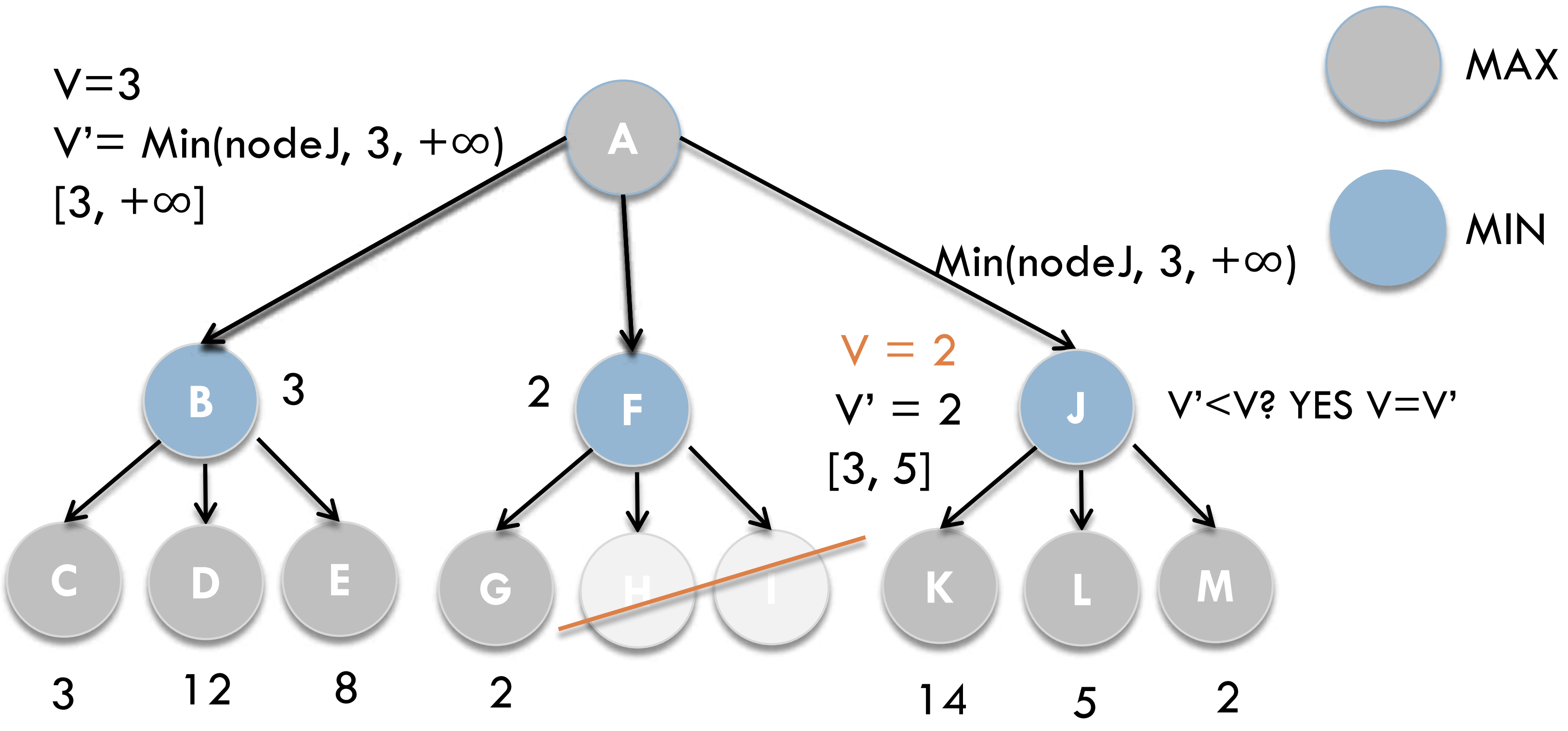
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



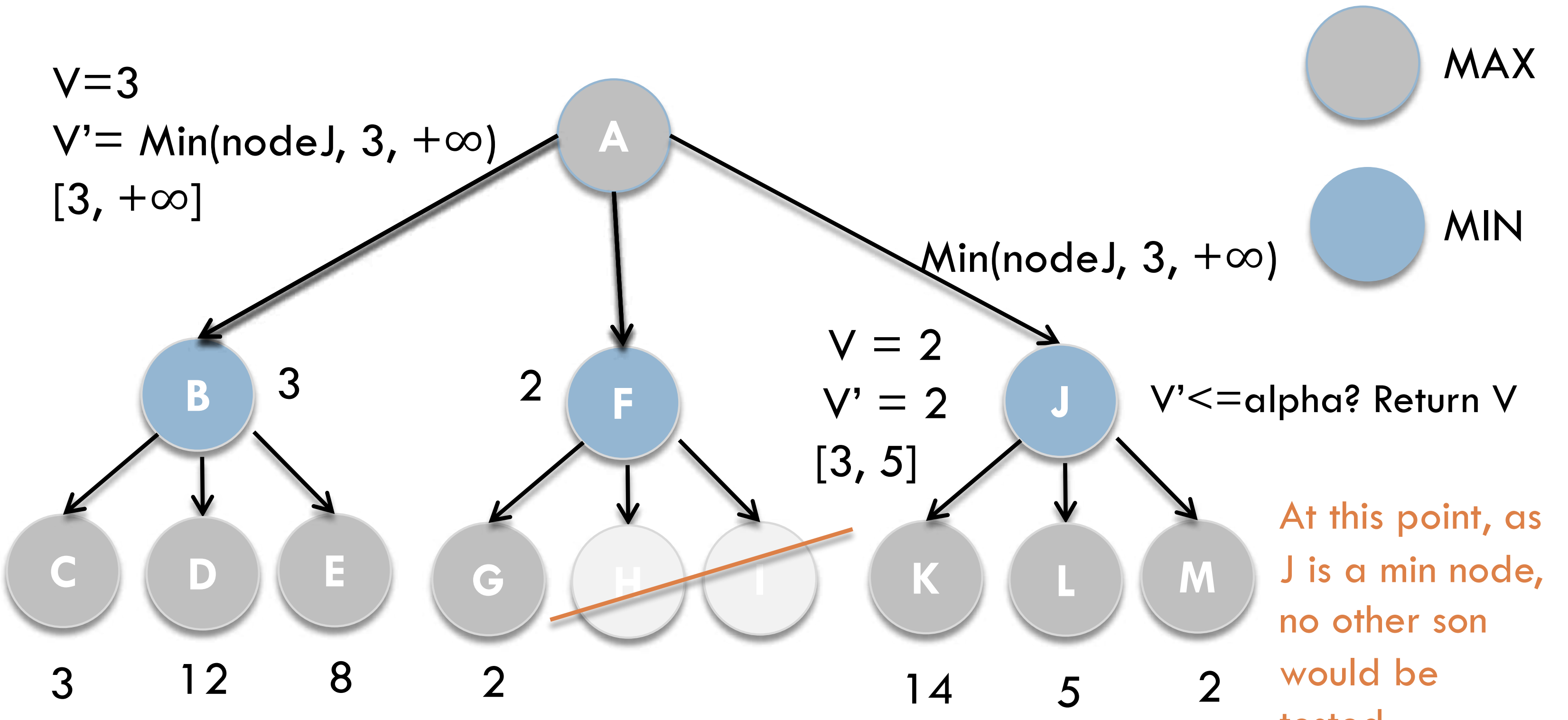
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



At this point, as J is a min node, no other son would be tested

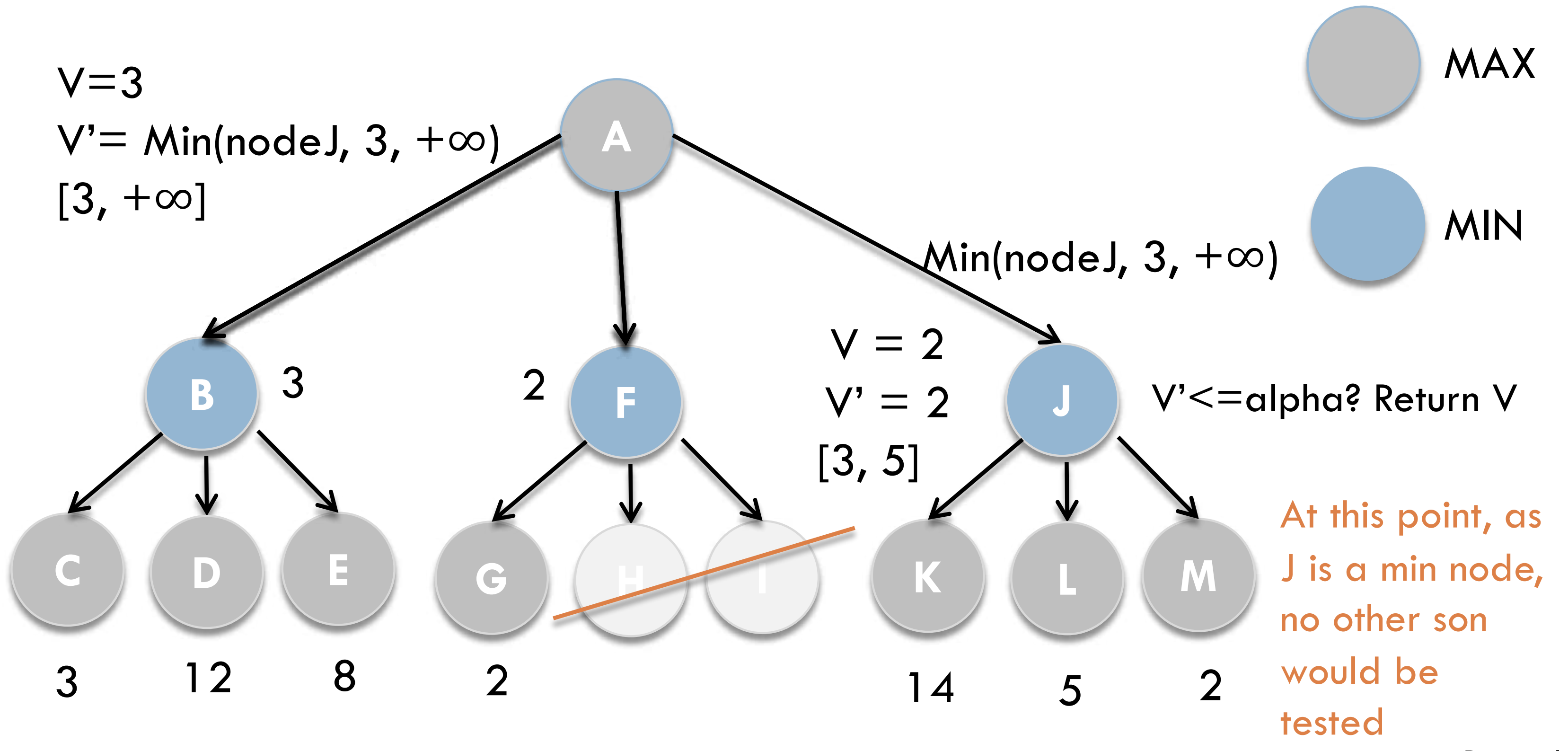
Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf



At this point, as J is a min node, no other son would be tested

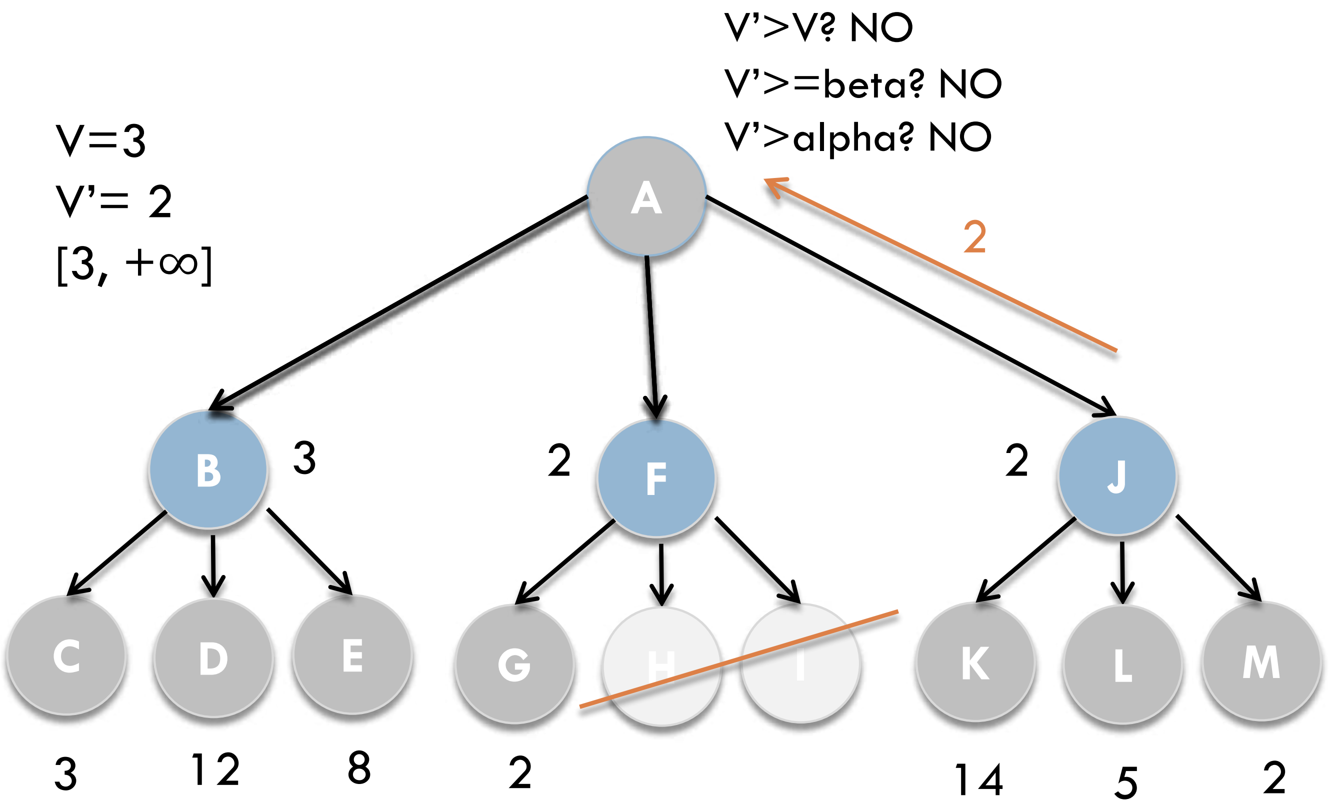
Remembering:

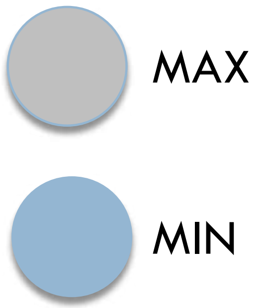
MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf





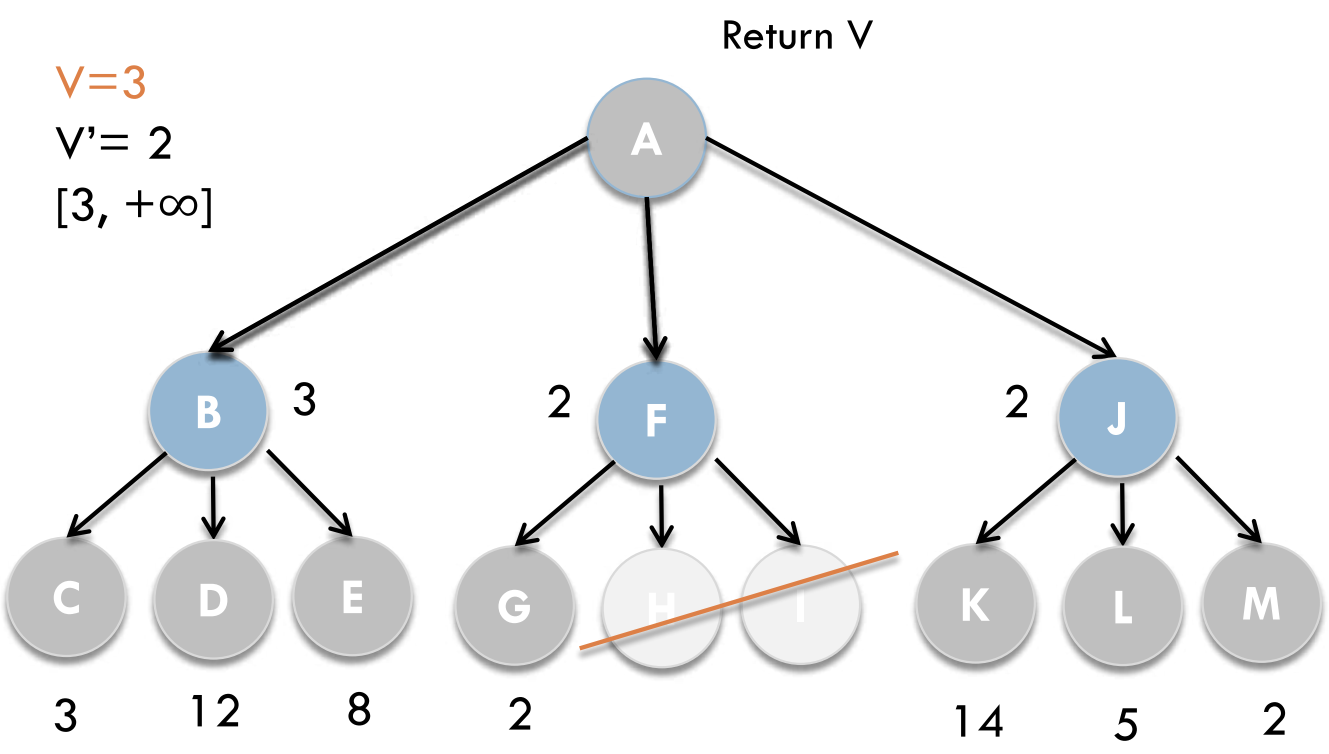
Remembering:

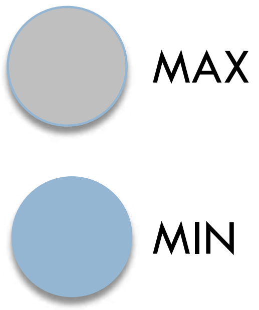
MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

Alpha-Beta Cut

- Search the first deepest to the first leaf





Remembering:

MIN nodes: Beta is the minimum (best) value found so far in the descendants of MIN nodes

MAX nodes: Alpha is the maximum (best) value found so far in the descendants of MAX nodes

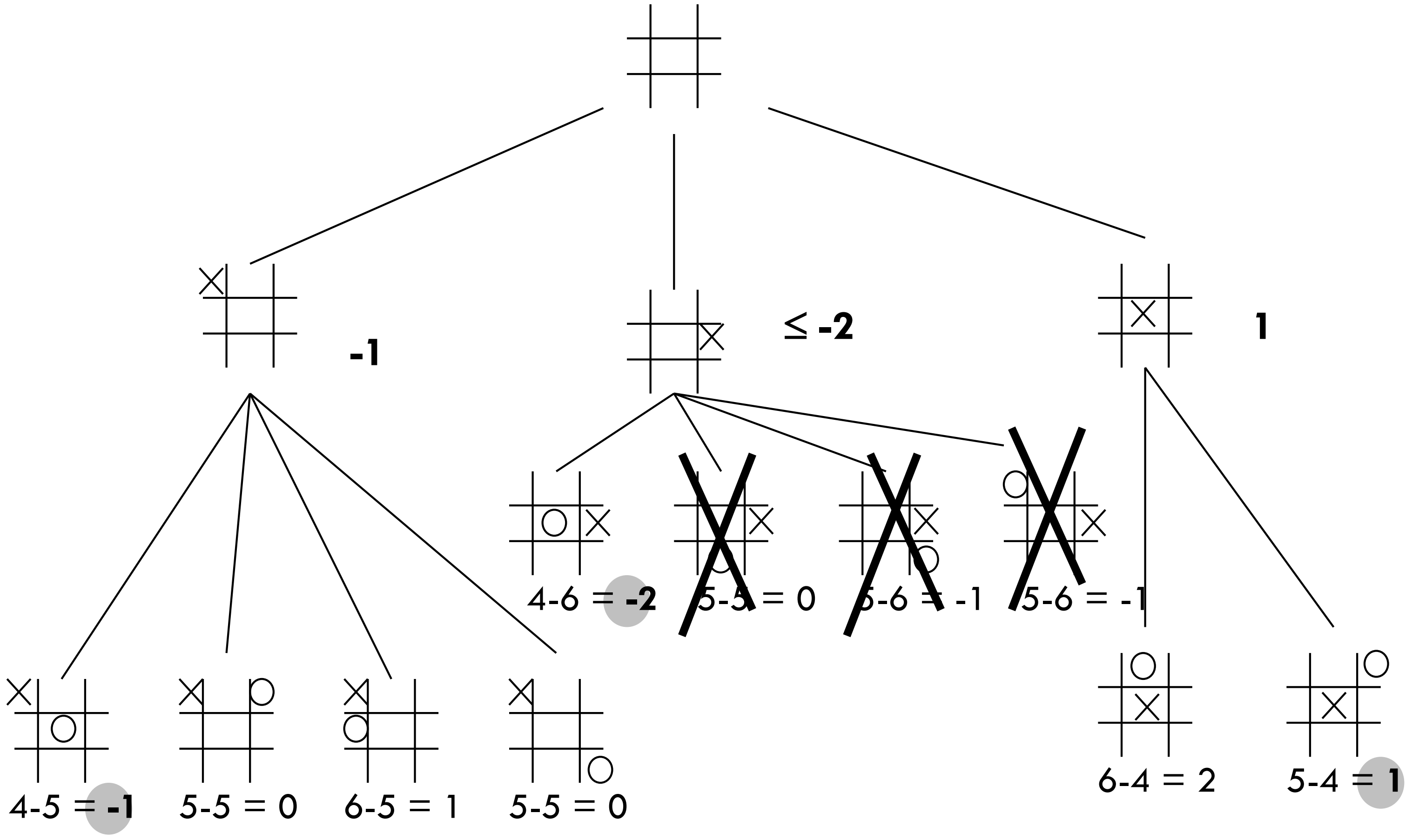
Alpha-Beta Cut

```
Max_Value(s,a,b) {
   if terminal(s), return U(s)
   v = -oo
   for c in next_states(s) {
      v' = Min_value(c,a,b)
      if v'> v, v=v’
      if v'>= b, return v
      if v'>a, a=v’
    }
    return v
}
```

```
Min_Value(s,a,b) {
   if terminal(s), return U(s)
    v = +oo
    for c in next_states(s) {
         v' = Max_value(c,a,b)
         if v'< v, v=v'
         if v'<= a, return v
         if v'<b, b=v’
    }
    return v
}
```

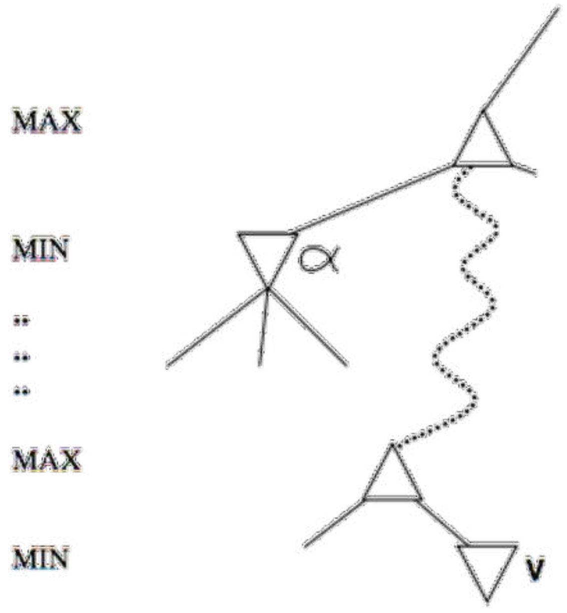
Alpha-Beta Cut

- **Alpha cut:** the best choice value (that is, the highest value) that we have found so far at any point of choice along the way for MAX.



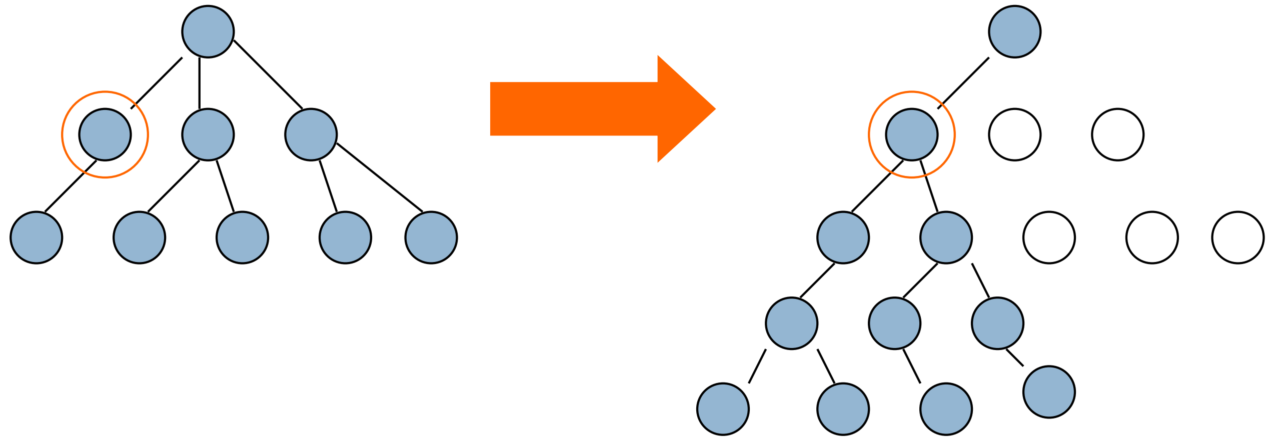
Alpha-Beta Cut

- α is the best value (for max) found so far along the way. If V is worse than α, max will avoid it $\Rightarrow$ cut off this branch.
- Similarly defined for β and min.
- With perfect ordering, $O(b^{m/2})$



MINMAX with Secondary Search

- We can further improve MinMax:
  - What's the big problem with MinMax?
    - Search depth (m) causes $O(b^m)$ complexity
  - How to solve the problem?
    - Do an analysis with a small depth …
    - … And after choosing the move, make an analysis with a greater depth only of the chosen branch (to check if the choice is really good) …

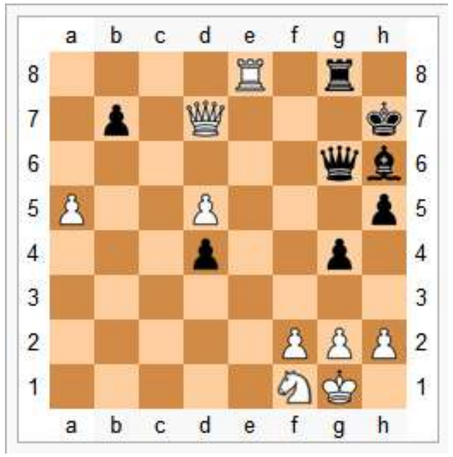


Playing chess with MINMAX

- Chess $\rightarrow$ b $\approx$ 35 (branch factor)
  - Depth of a solution: 50 for each player (100 for both players)
  - Order search space 35^{100}
- **Some systems that play chess:**
  - Deep Thought 2 (IBM + CMU researchers)
    - simple evaluation function and alpha-beta pruning
    - 10 processors capable of examining 1/2 billion positions per move: reach depths of 10 to 11 moves (depth 37 has already been reached with identified cases)

Playing chess with MINMAX

- Some systems that play chess:
- Deep Blue:
  - He won some matches for Kasparov in 1996 and 1997 and lost others
  - 32 nodes each with 16 “chess processors” - a total of 512 processors
  - 126 million nodes per second
- X3D Fritz (German software):
  - Drew with Kasparov at the end of 2003 in a
  - best of 4 games. Run on a computer
  - Intel Pentium 4 Xeon 2.8GHz



End of the second round between Kasparov and X3D Fritz with victory for X3D Fritz

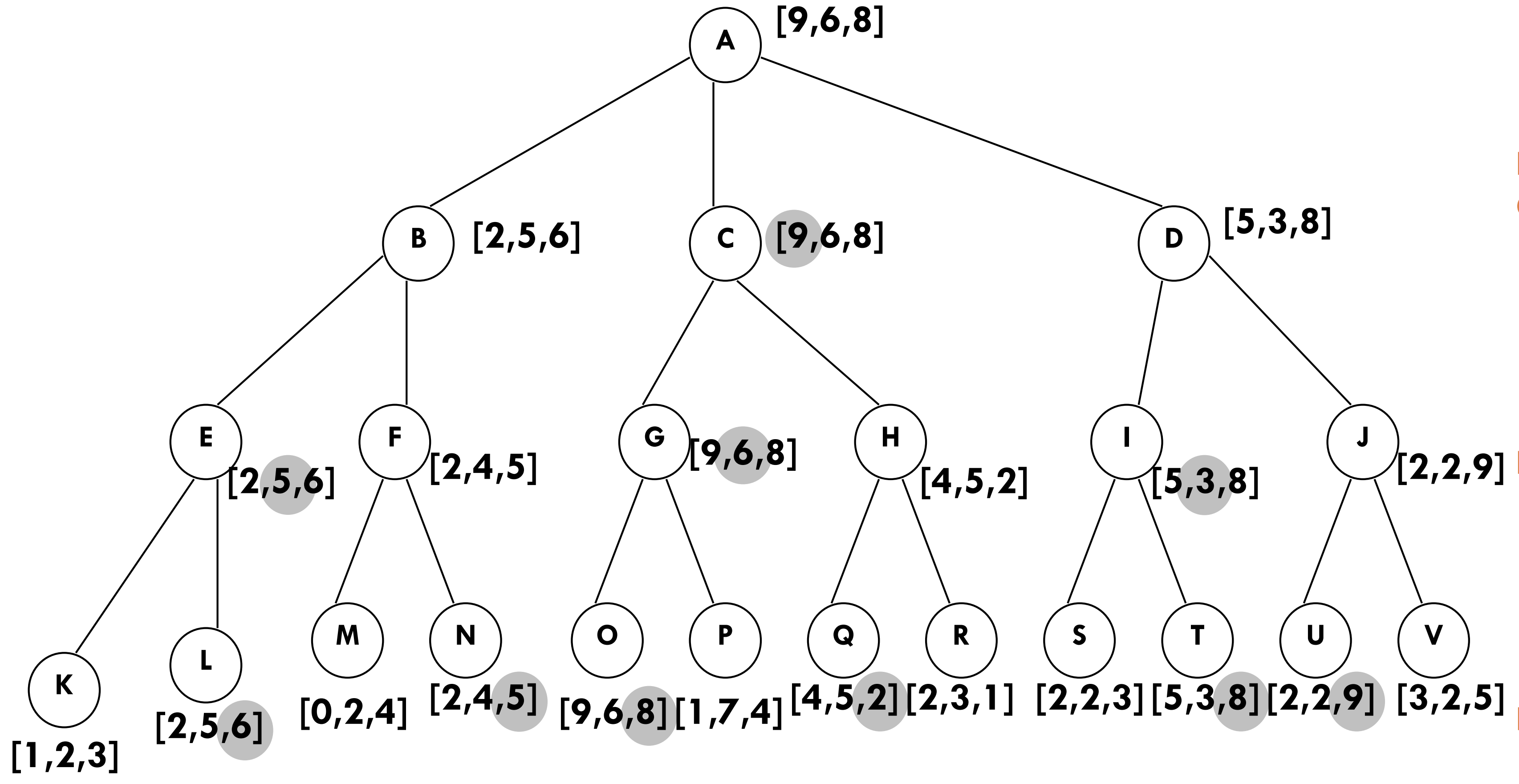
Playing chess

- A Utility Function for Chess
  - Assign a material value for each part:
    - Pawn = 1
    - Knight or bishop = 3
    - Tower = 5
    - Queen = 9
  - Player has a material value of pieces
- Utility function: F = value of your pieces - value of opponent pieces
  - The side with a sure advantage of a pawn's material value or more is likely to be the winner
  - We can also include: bishop closer to the end of the game is worth more, position of the piece in safety is worth 1 point, etc ...

Games with more than 2 players

- There are games with more than two players
- How to adapt MINMAX for these games?
  - Instead of a value, we will have a VECTOR of values. In zero-sum games with two players, the vector of two elements can be reduced to a single value because the values are always opposite.
  - One value for each player
  - Players A, B and C → [Va, Vb, Vc]
  - Each step in the tree will be a player's
  - The root remains the play of the computer
- The utility function should return the vector of values. Allows you to implement alliances or games between partners.
- It only works for games where all the values of all players are known.

MINMAX: 3 palyers



Player A
Computer

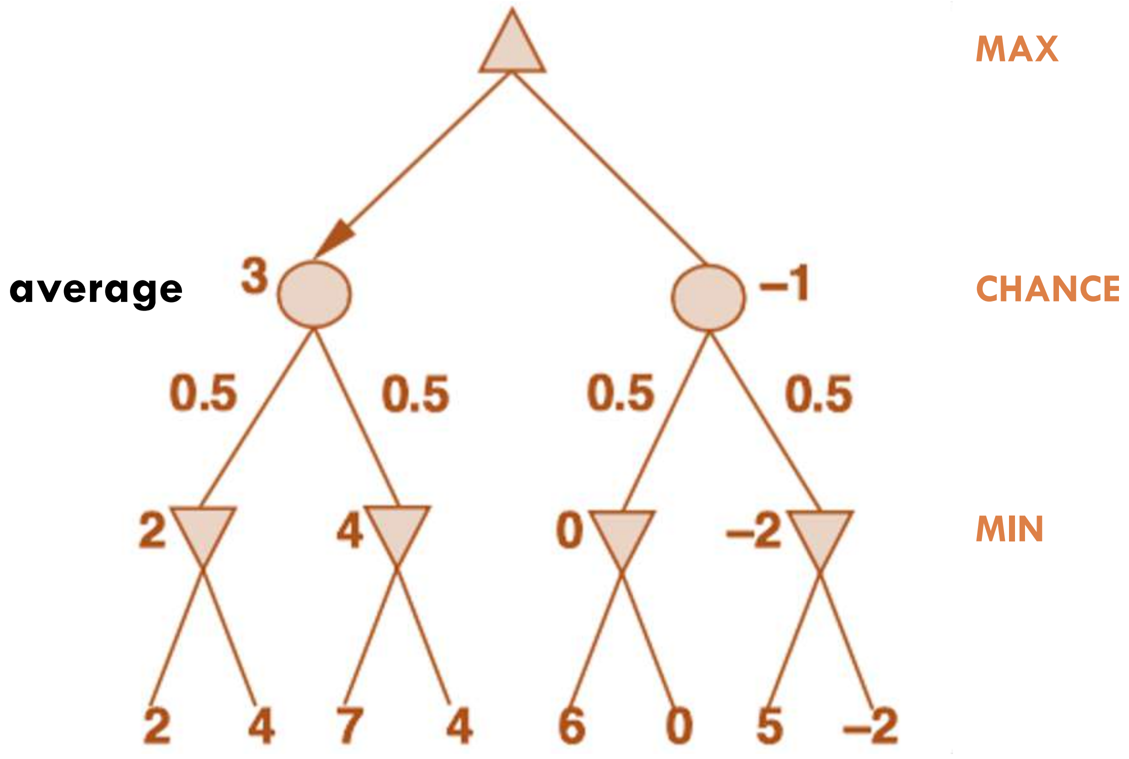
Player B

Player C

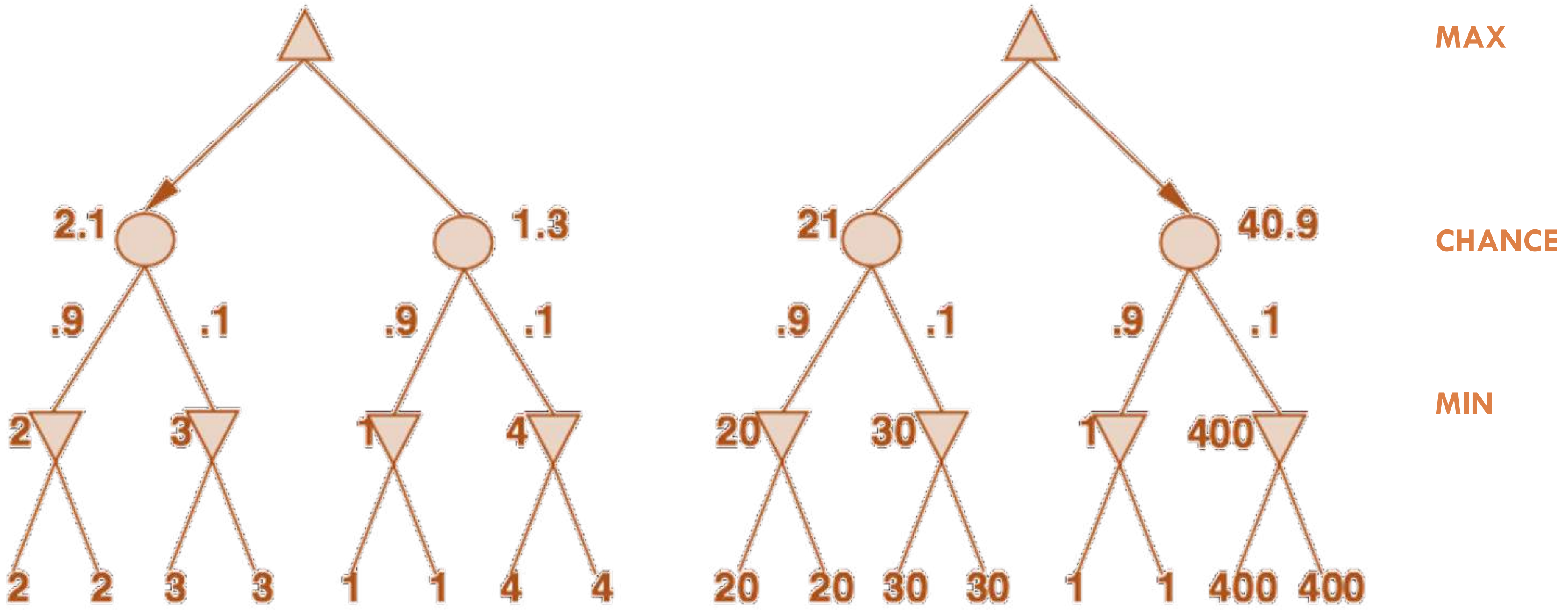
Non-deterministic games

- A third player appears: the random factor
- With the occurrence of unpredictable events (eg. data entry), there is no way to build a standard search tree
  - the search tree must include random nodes, in addition to the MAX and MIN nodes
  - ramifications on chance nodes denote possible launches and respective probabilities

Non-deterministic games

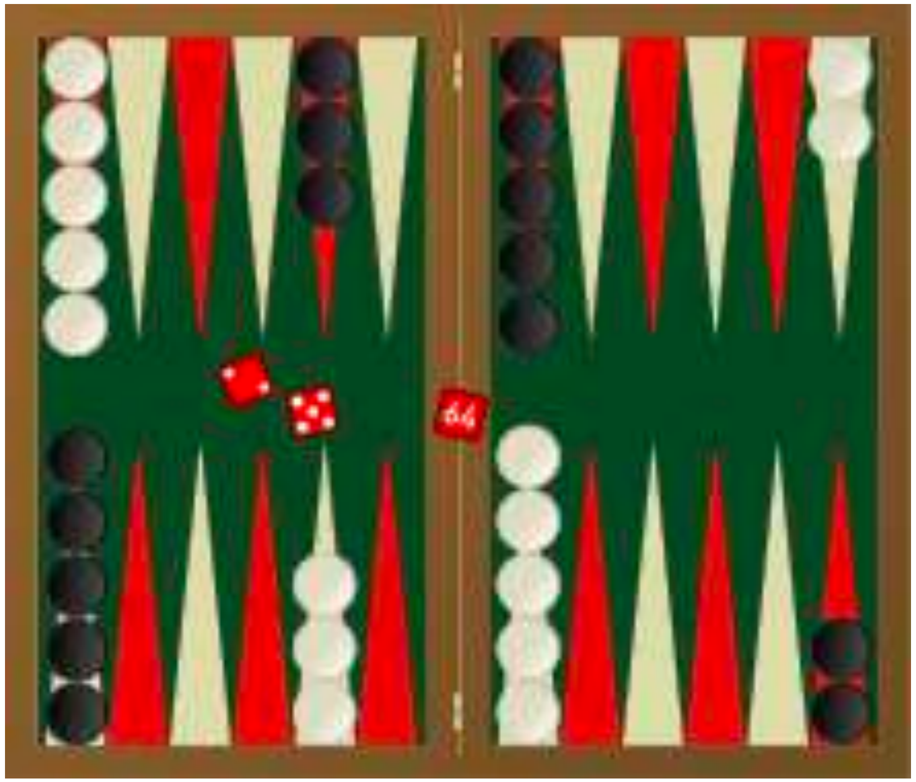


Non-deterministic games



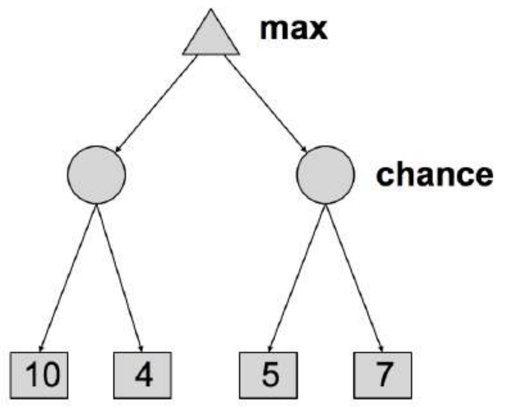
Stochastic Two-Player

- Dice rolls increase b: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - Depth 4 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given node shrinks
  - So value of lookahead is diminished
  - So limiting depth is less damaging
  - But pruning is less possible… !
- TDGammon uses depth-2 search + very good eval function + reinforcement learning: world champion level play



Expectimax Search Trees

- What if we don't know what the result of an action will be? E.g.
  - In solitaire, next card is unknown
  - In minesweeper, mine locations
  - In pacman, the ghosts act randomly
- Can do expectimax search
  - Chance nodes, like min nodes, except the outcome is uncertain
  - Calculate expected utilities
  - Max nodes as in minimax search
  - Chance nodes take average (expectation) of value of children



- Later, we'll learn how to formalize the underlying problem as a Markov Decision Process

Expectimax Search Trees

- Why should we average utilities? Why not minimax?
- Principle of maximum expected utility: an agent should chose the action which maximizes its expected utility, given its knowledge
  - General principle for decision making
  - Often taken as the definition of rationality
  - We'll see this idea over and over in this course!
- Let's decompress this definition…

Reminder: Probabilities

- A **random variable** represents an event which outcome is unknown
- A **probability distribution** is an assignment of weights to outcomes
- Example: traffic on freeway?
  - Random variable: T = whether there's traffic
  - Outcomes: T in {none, light, heavy}
  - Distribution: P(T=none) = 0.25, P(T=light) = 0.55, P(T=heavy) = 0.20
- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one
- As we get more evidence, probabilities may change:
  - P(T=heavy) = 0.20, P(T=heavy | Hour=8am) = 0.60
  - We'll talk about methods for reasoning and updating probabilities later

Reminder: Probabilities

- Objectivist / frequentist answer:
  - Averages over repeated experiments
  - E.g. empirically estimating P(rain) from historical observation
  - E.g. pacman's estimate of what the ghost will do, given what it has done in the past
  - Assertion about how future experiments will go (in the limit)
  - Makes one think of inherently random events, like rolling dice
- Subjectivist / Bayesian answer:
  - Degrees of belief about unobserved variables
  - E.g. an agent's belief that it's raining, given the temperature
  - E.g. pacman's belief that the ghost will turn left, given the state
  - Often learn probabilities from past experiences (more later)
  - New evidence updates beliefs (more later)

Uncertainty

- Not just for games of chance! !
  - I'm sick: will I sneeze this minute?
  - Email contains "FREE!": is it spam?
  - Tooth hurts: have cavity?
  - 60 min enough to get to the airport?
  - Robot rotated wheel three times, how far did it advance?
  - Safe to cross street? (Look both ways!)
- Sources of uncertainty in random variables:
  - Inherently random process (dice, etc) ! Insufficient or weak evidence ! Ignorance of underlying processes ! Unmodeled variables ! The world's just noisy – it doesn't behave according to plan

Uncertainty

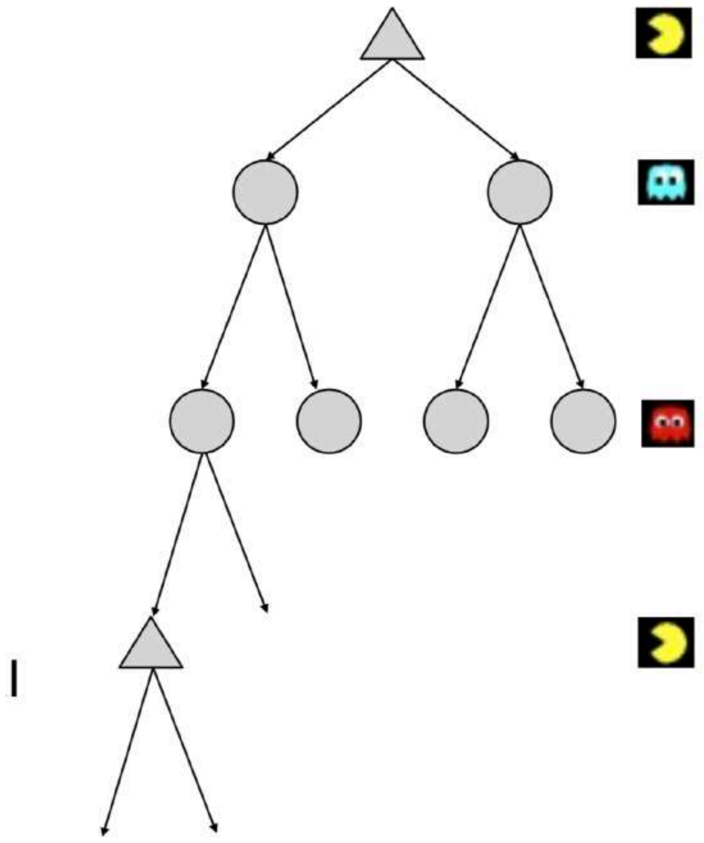
- We can define function f(X) of a random variable X
- The expected value of a function is its average value, weighted by the probability distribution over inputs
- Example: How long to get to the airport?
  - Length of driving time as a function of traffic:
    - L(none) = 20, L(light) = 30, L(heavy) = 60
  - What is my expected driving time?
    - Notation: EP(T)[L(T)]
    - Remember, P(T) = {none: 0.25, light: 0.5, heavy: 0.25}
    - E[L(T)] = L(none) * P(none) + L(light) * P(light) + L(heavy) * P(heavy)
    - E[L(T)] = (20 * 0.25) + (30 * 0.5) + (60 * 0.25) = 35

Uncertainty

- Utilities are functions from outcomes (states of the world) to real numbers that describe an agent's preferences
- Where do utilities come from?
  - In a game, may be simple (+1/-1)
  - Utilities summarize the agent's goals
  - Theorem: any set of preferences between outcomes can be summarized as a utility function (provided the preferences meet certain conditions)
- In general, we hard-wire utilities and let actions emerge (why don't we let agents decide their own utilities?)
- More on utilities soon…

Expectimax

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a node for every outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!
- For now, assume for any state we magically have a distribution to assign probabilities to opponent actions / environment outcomes

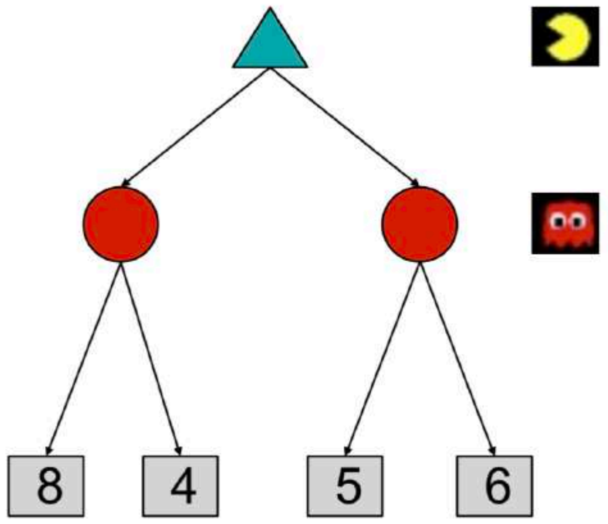


Expectimax

```
def value(s)
        if s is a max node return maxValue(s)
        if s is an exp node return expValue(s)
        if s is a terminal node return evaluation(s

def maxValue(s)
        values = [value(s') for s' in successors(s)]
        return max(values)

def expValue(s)
        values = [value(s') for s' in successors(s)]
        weights = [probability(s, s') for s' in
successors(s)]
        return expectation(values, weights)
```



Expectimax

- Notice that we've gotten away from thinking that the ghosts are trying to minimize pacman's score
  - Instead, they are now a part of the environment
  - Pacman has a belief (distribution) over how they will act
  - Quiz: Can we see minimax as a special case of expectimax?
  - Quiz: what would pacman's computation look like if we assumed that the ghosts were doing 1-play minimax and taking the result 80% of the time, otherwise moving randomly?

Extra-class activities

Lecture 6

- Activities
  - Reading:
    - RUSSELL, S. NORVIG, P. Inteligência Artificial. 3ª edição. Capítulos 5.
  - Recommended exercises:
    - 5.2, 5.3, 5.9 e 5.12.

References

Lecture 6

- RUSSELL, S. NORVIG, P. Inteligência Artificial. 3ª edição.
- COLOMBINI, E. C, TONIDANDEL, F. Notas de Aula de IA.