# CS380
# Artificial Intelligence for Games

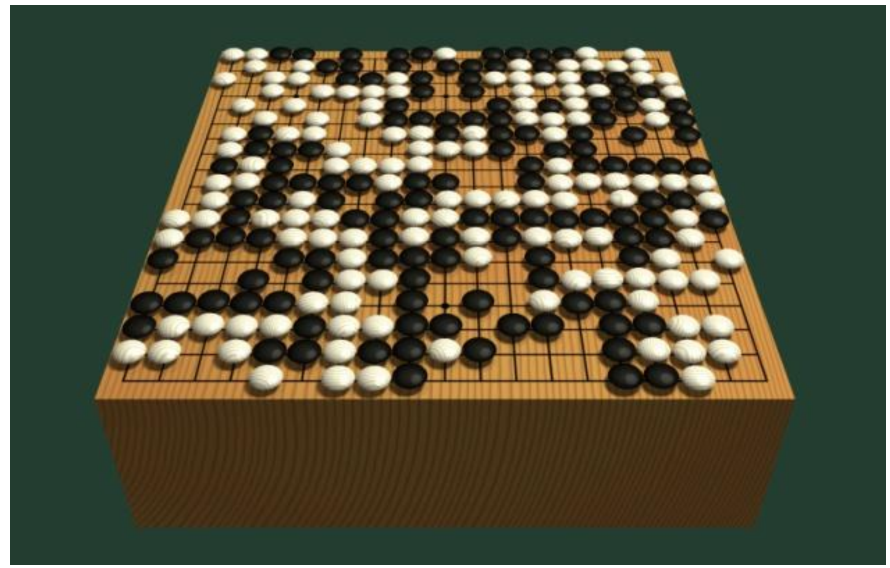
# Adversarial Search

# Game Playing State-of-the-Arts

- **Chess:** 1997: Deep Blue defeats human champion **Gary Kasparov** in a six-game match. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply

# Game Playing State-of-the-Arts

- **Go:** Human champions are now starting to be challenged by machines, though the best humans still beat the best machines. In Go, number of terminal states > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods



# Applications

- Two player
  - Player or Human, Computer or AI
- Players take turns when playing
  - Somebody goes first. Who? Flip a coin
- Zero-sum
  - if one player loses, the other player wins
- Perfect information
  - Each player is completely informed of previous moves
- Deterministic
  - No randomness, follows a strict pattern
- Have small number of possible actions
- Precise, formal rules

# Applications

| | **Perfect Information** | **Imperfect Information** |
|---|---|---|
| **Deterministic** | Tic-Tac-Toe, Go, Chess, Checkers, Othello, Kalah, NIM | Monopoly, Aeroplane Chess, Stratego |
| **Non-deterministic** | Backgammon | Poker, Scrabble |

# Challenge

- Challenging due to huge search space
  - Chess:
    - branching factor = 35,
    - number of moves per player = 50

- Search optimization
  - Alpha-Beta pruning

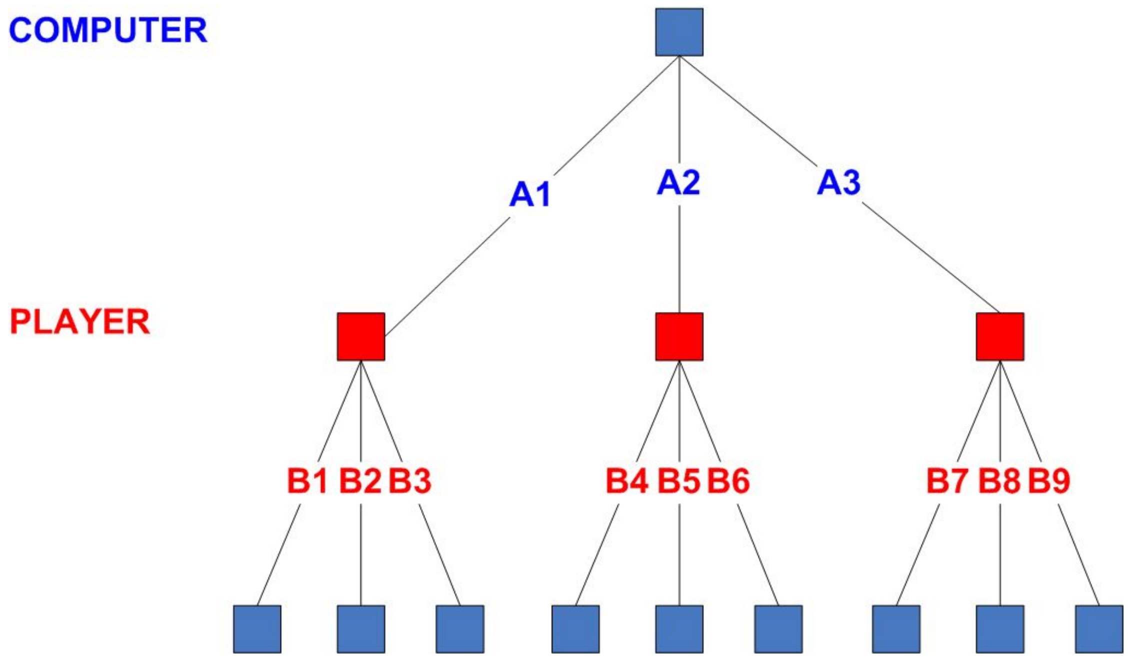
- If unlikely to find goal, time limit is introduced
  - so a terminal state is approximated

# Games as a Search Problem

- Game has
  - **initial state**
    - configuration + player to move
  - **successor function**
    - defines the set of possible moves/actions/states from a state
  - **terminal test**
    - game over or not
  - **utility function**
    - maps terminal states to numeric values
      - win : 1, 10, +INF, loss : -1, -10, -INF, draw : 0
- Players have to find **contingent strategies**
  - Devised for a specific situation where things could go wrong
  - An optimal strategy assumes an infallible opponent

# Game Tree

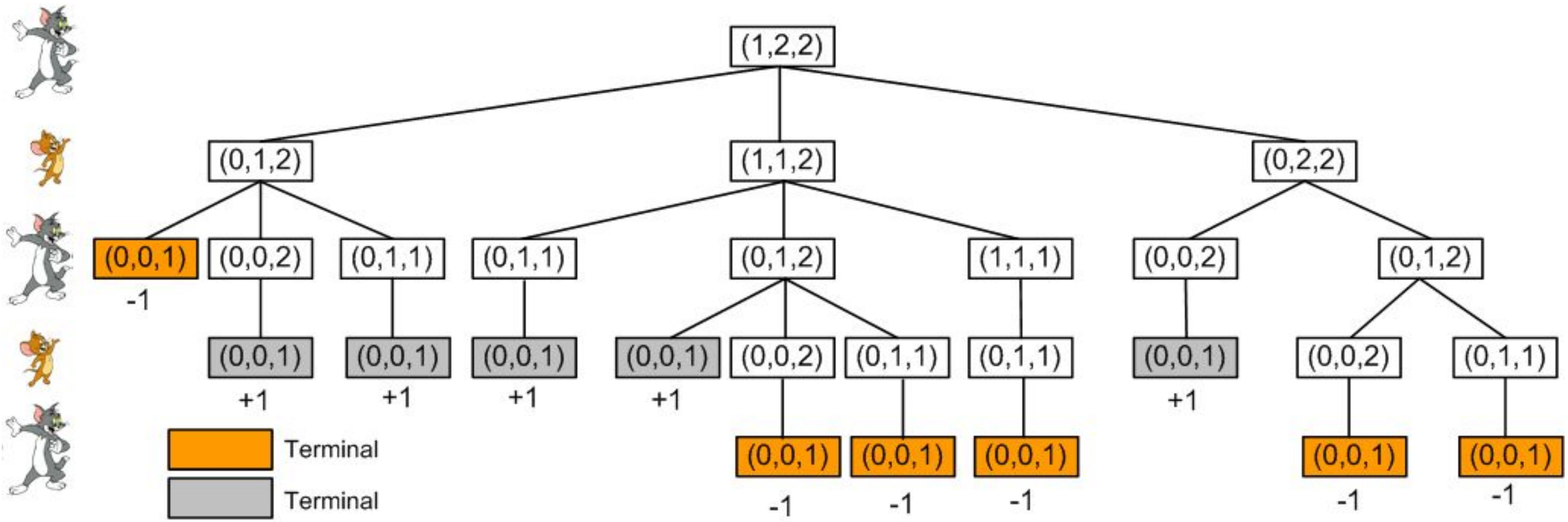
- Game tree = initial state + legal moves or actions
- Given tree is one move deep, consisting of two halfmoves, each of which is called a ply



# Game of NIM

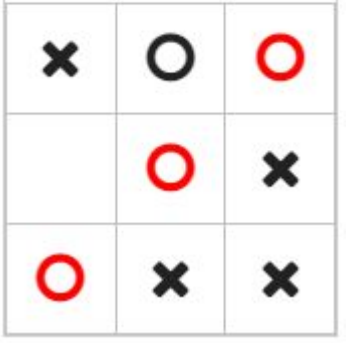
- Several piles of sticks are given. We represent the configuration of the piles by a **monotone sequence** of integers, such as (1,3,5) or (0,0,1)
- A player may remove, in one turn, any number of sticks from one pile.
  - Thus, (1,3,5) would become (1,1,3) if the player were to remove 4 sticks from the last pile. The player who takes the last stick loses.
- Let’s represent the game with initial state (1, 2, 2) as a game tree

# Example: Game of NIM

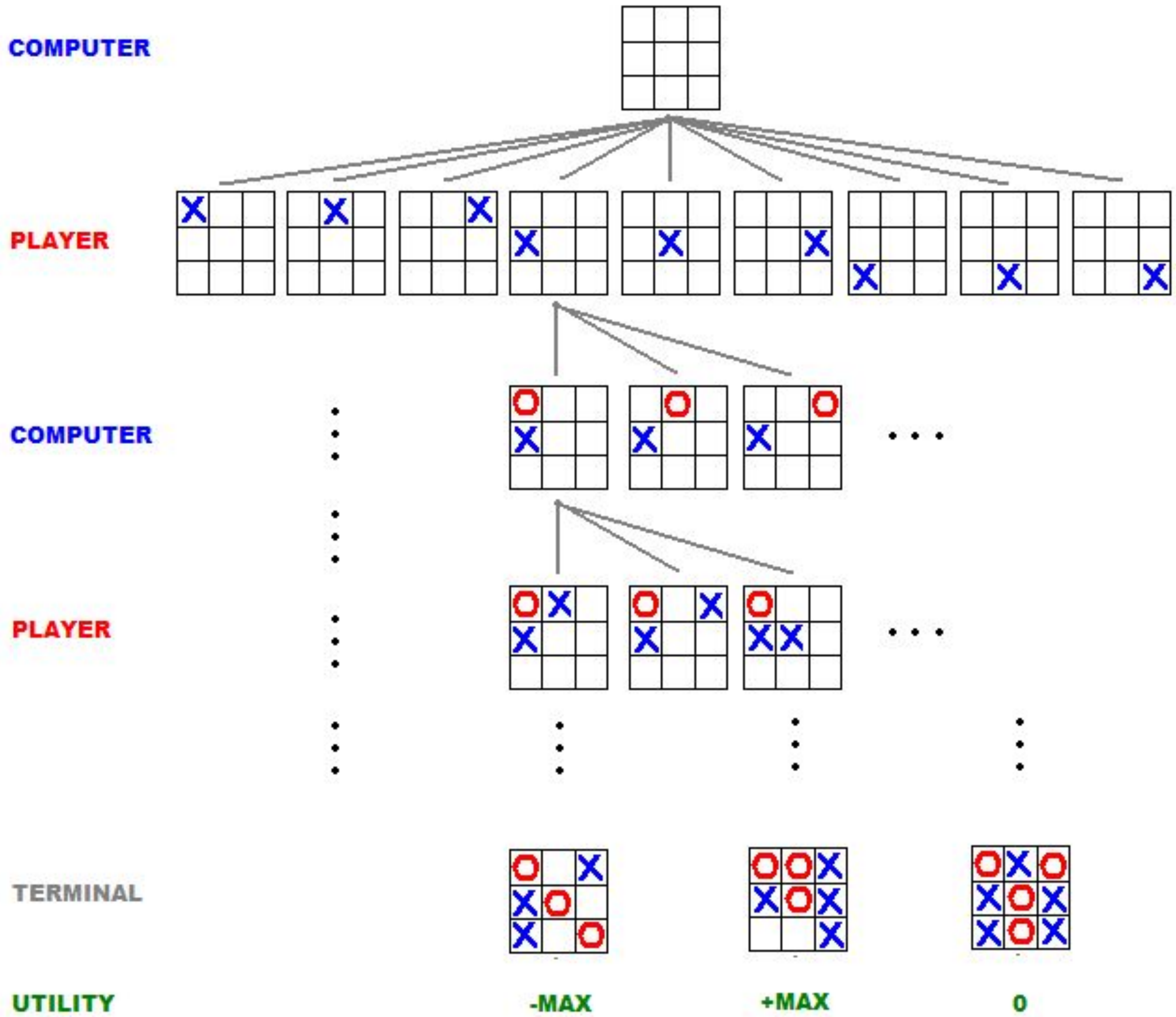


# Tic-Tac-Toe Game

- Tic-Tac-Toe is well-known pencil-and-paper game for two players, who take turns marking the spaces in a 3×3 grid.
- The player who succeeds in placing three of their marks in a diagonal, horizontal, or vertical row is the winner.



# Example: Tic-Tac-Toe



# How to Choose the Optimal Decision?

## Minimax Search

# How to Choose the Optimal Decision?

- Player has always a role in complicating the trajectory of Computer to reach a win
- Computer needs to consider every possible response by Player
- Computer must take in consideration that Player is always playing his best moves when he has turn

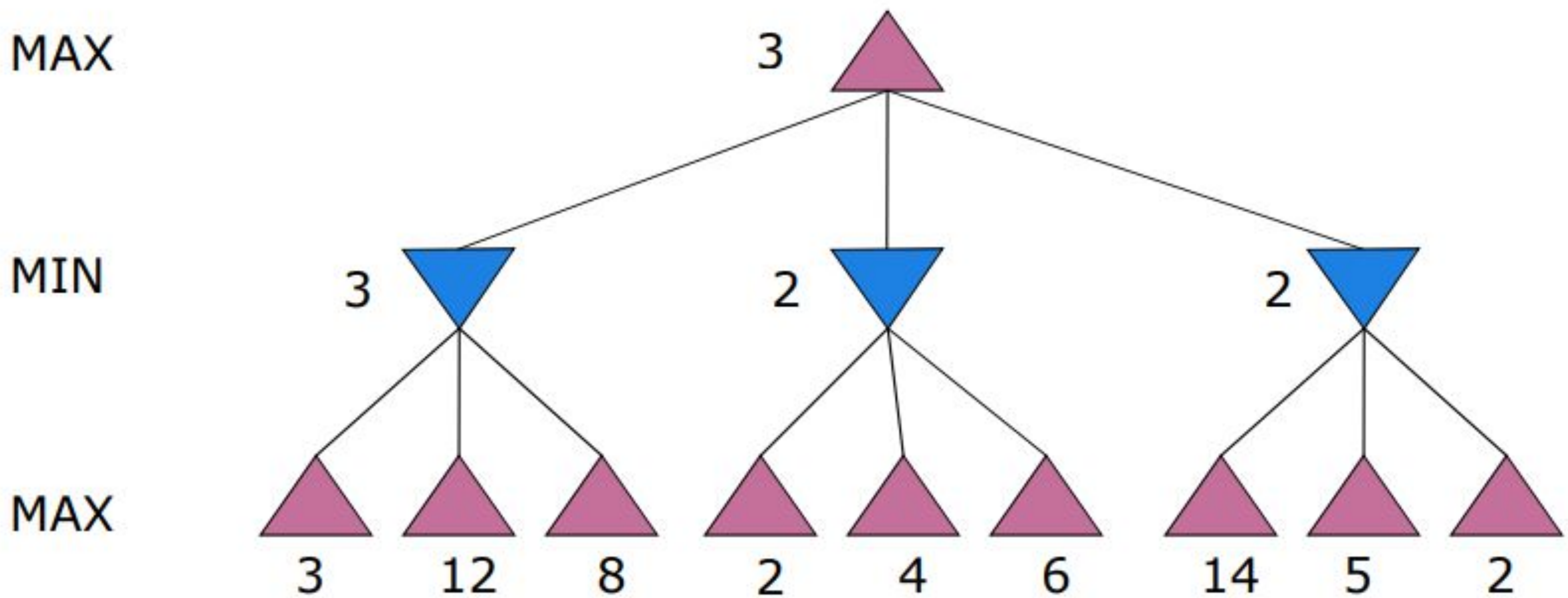
# Minimax Search

- Minimax search guarantees the perfect play for deterministic, perfect-information games
- Idea: choose move to position with highest minimax value = best achievable payoff against best play

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST(s)} \\ \max_{a \in Actions(s)} \text{MINIMAX(RESULT(s,a))} & \text{if PLAYER-TURN(s)} = \text{ MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX(RESULT(s,a))} & \text{if PLAYER-TURN(s)} = \text{ MIN} \end{cases}$$

# Minimax Search

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST(s)} \\ \max_{a \in Actions(s)} \text{MINIMAX(RESULT(s,a))} & \text{if PLAYER-TURN(s)} = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX(RESULT(s,a))} & \text{if PLAYER-TURN(s)} = \text{MIN} \end{cases}$$



# Minimax Search Algorithm

**function** MINIMAX-DECISION(*state*) **returns** *an action*

$v \leftarrow$ MAX-VALUE(*state*)
**return** the *action* in SUCCESSORS(*state*) with value $v$

---

**function** MAX-VALUE(*state*) **returns** *a utility value*

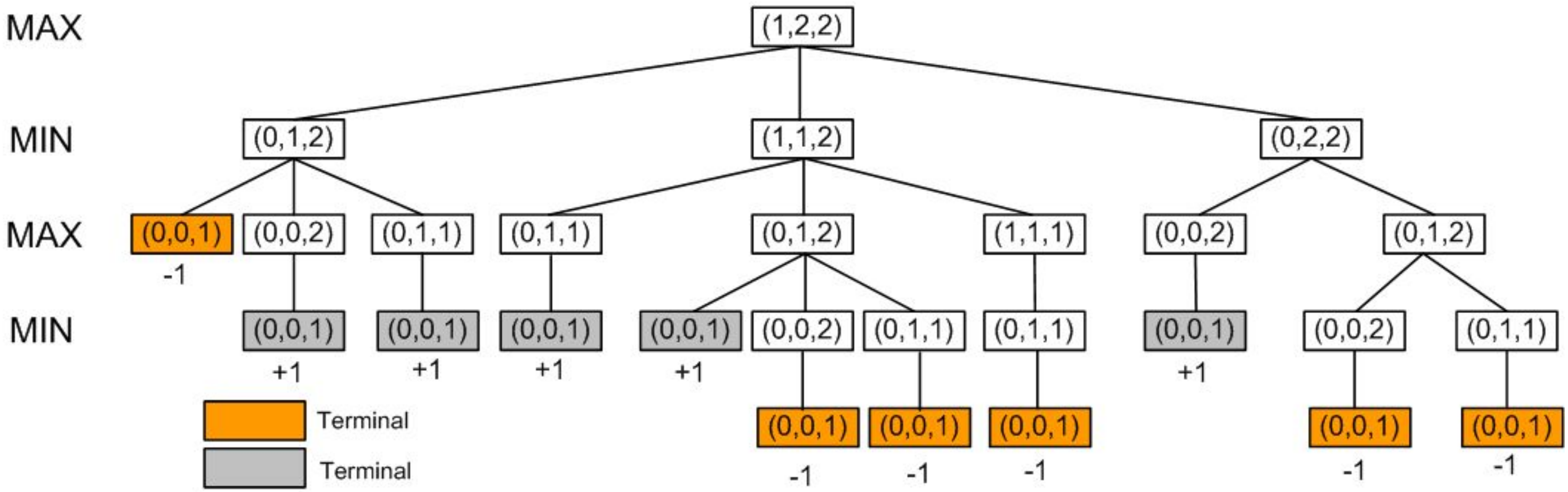
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow -\infty$
**for** $a, s$ in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX($v$, MIN-VALUE($s$))
**return** $v$

---

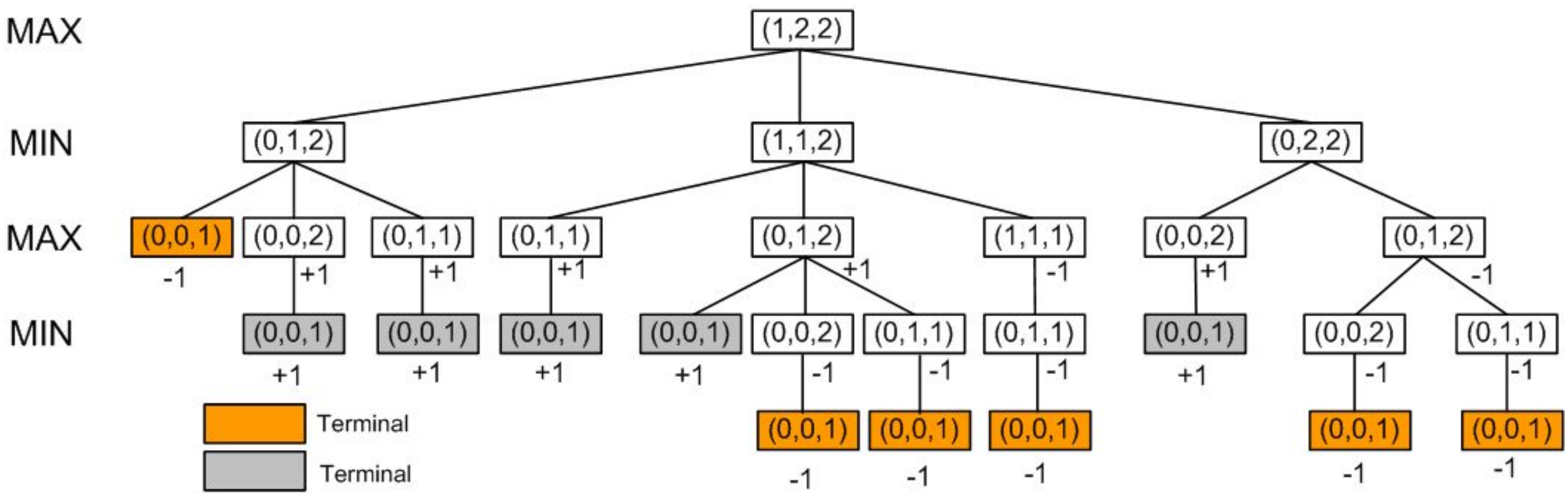
**function** MIN-VALUE(*state*) **returns** *a utility value*

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow \infty$
**for** $a, s$ in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN($v$, MAX-VALUE($s$))
**return** $v$

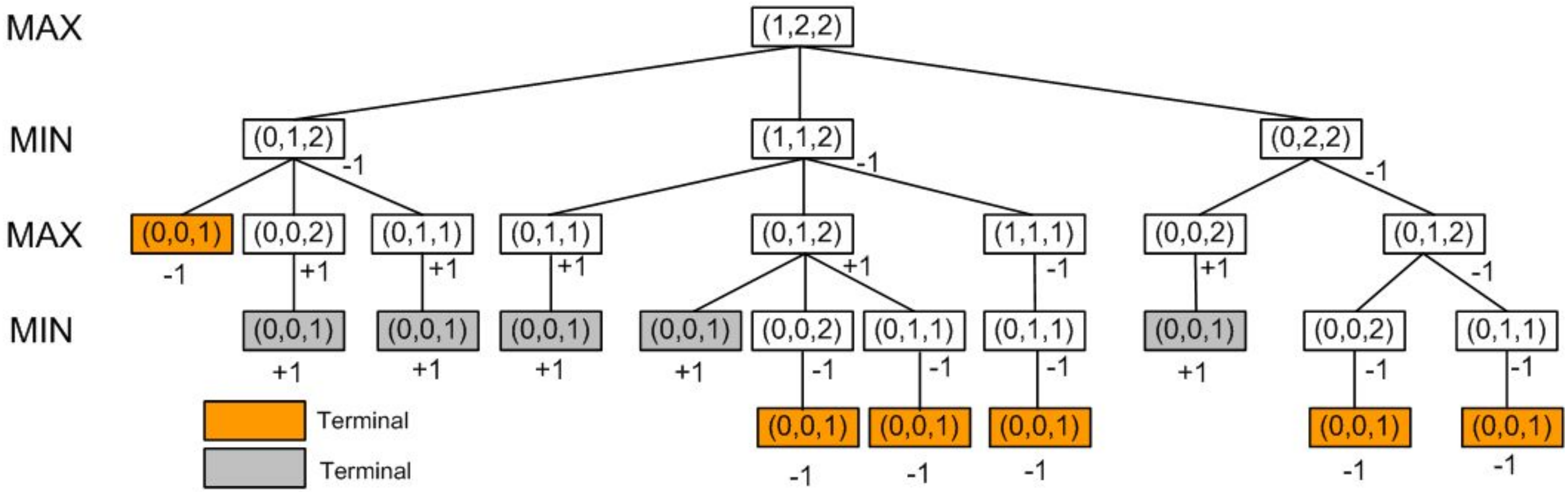
# Example: Game of NIM



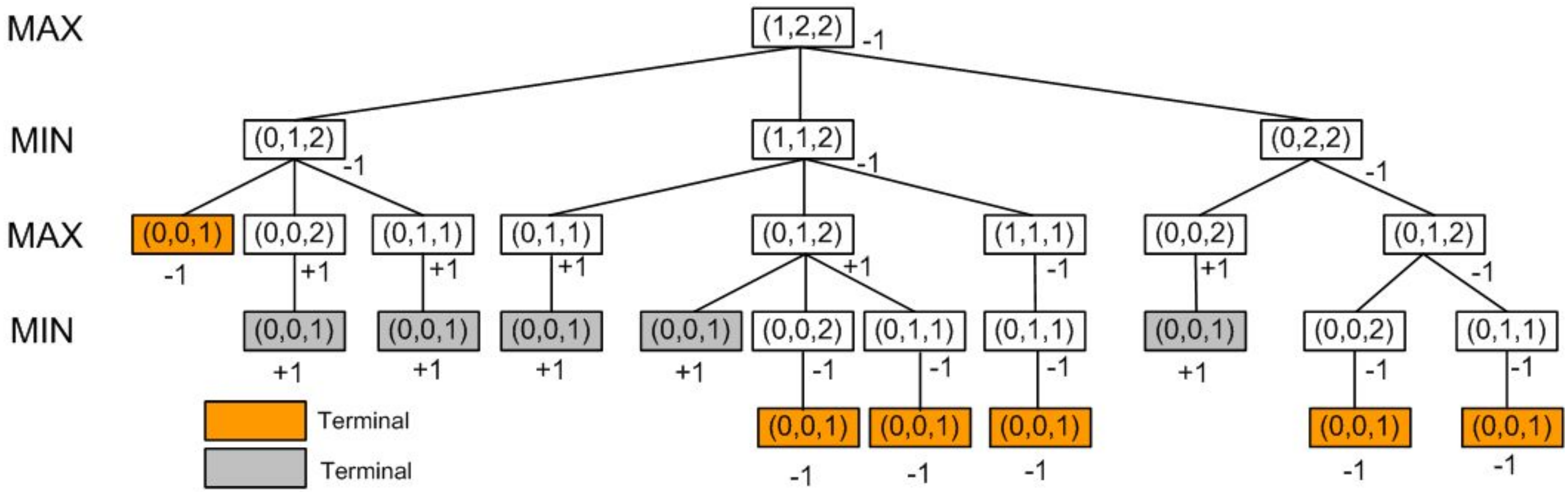
# Example: Game of NIM



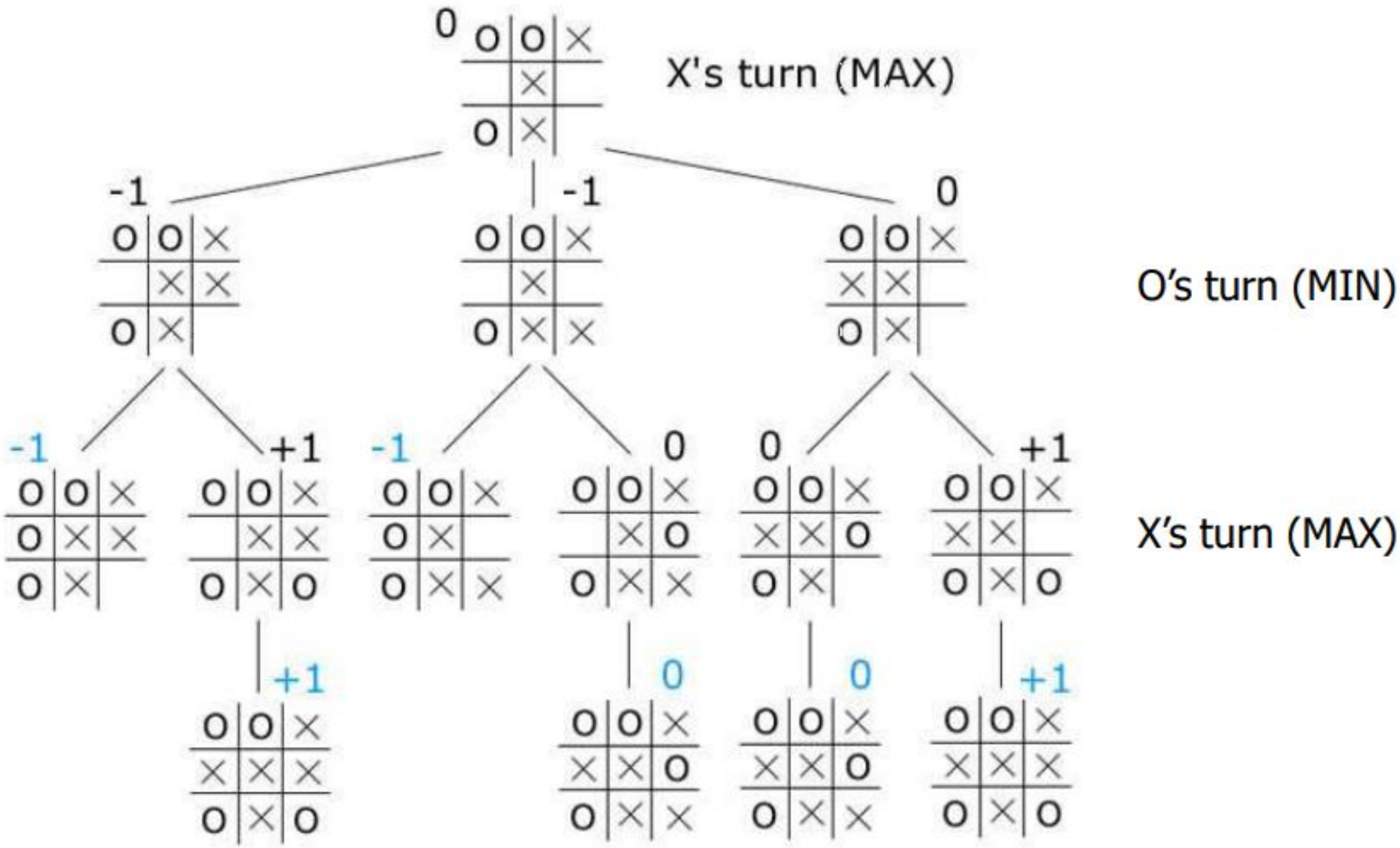
# Example: Game of NIM



# Example: Game of NIM



# Example: A Partial Game Tree for Tic-Tac-Toe



# Properties of Minimax

- Complete?
  - Yes, if the tree is finite
- Optimal?
  - Yes, if against an optimal opponent.
  - Otherwise?
- Assume the branching factor is k and the depth of the tree is d
  - Time complexity?
    - $O(k^d)$
  - Space complexity?
    - $O(kd)$ (depth-first exploration)

# Properties of Minimax

- For tic-tac-toe
  - $k = ?, d = ?$
- For chess
  - $k \approx 35, d \approx 100$ for “reasonable” games
  - Exact solution completely infeasible
- For GO
  - $k \approx 200, d \approx 100$ for “reasonable” games
- Can we do better?

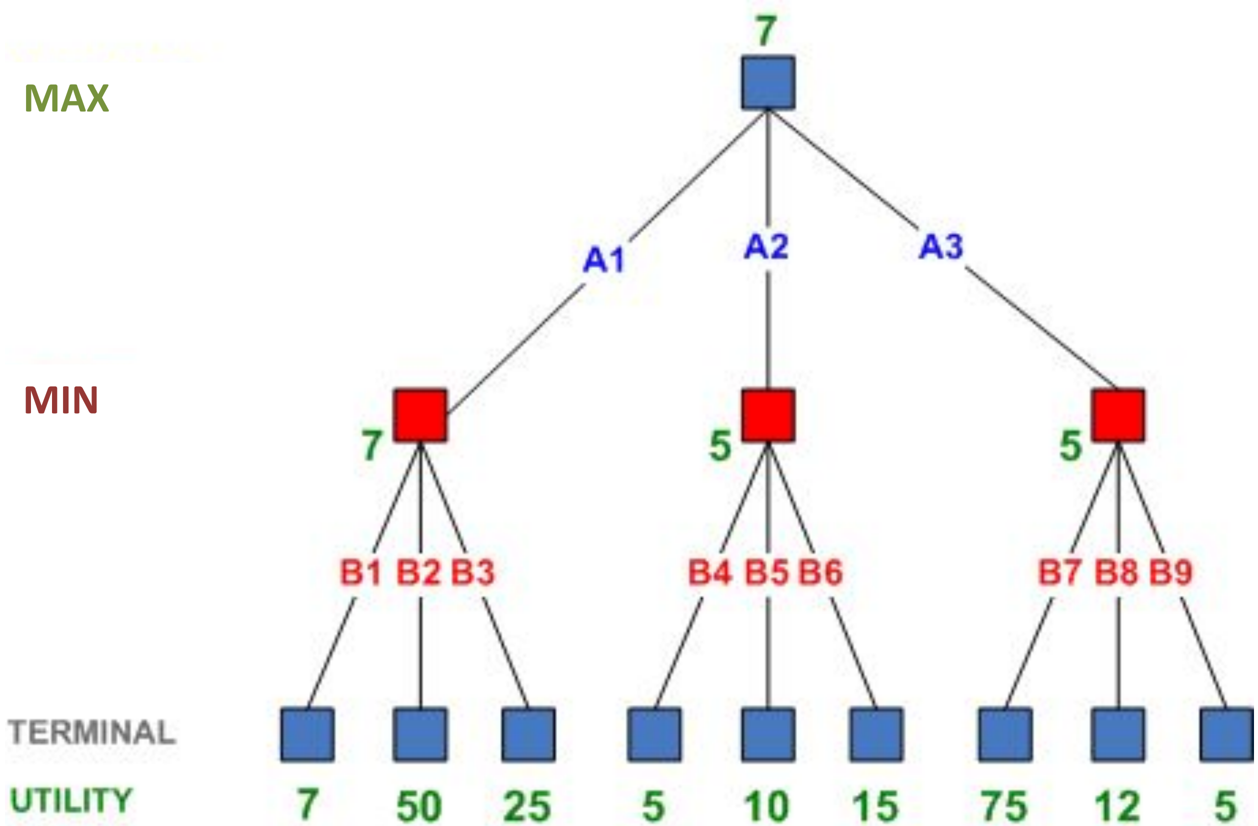
# α-β Pruning

Search optimisation techniques for choosing a good move (may not be optimal) when time is limited

# α-β Pruning

- The number of game states with minimax search is exponential in the # of moves
- Is it possible to compute the correct minimax decision without looking at every node in the game tree?
- Need to **prune** away branches that cannot possibly influence the final decision

# Motivating Example

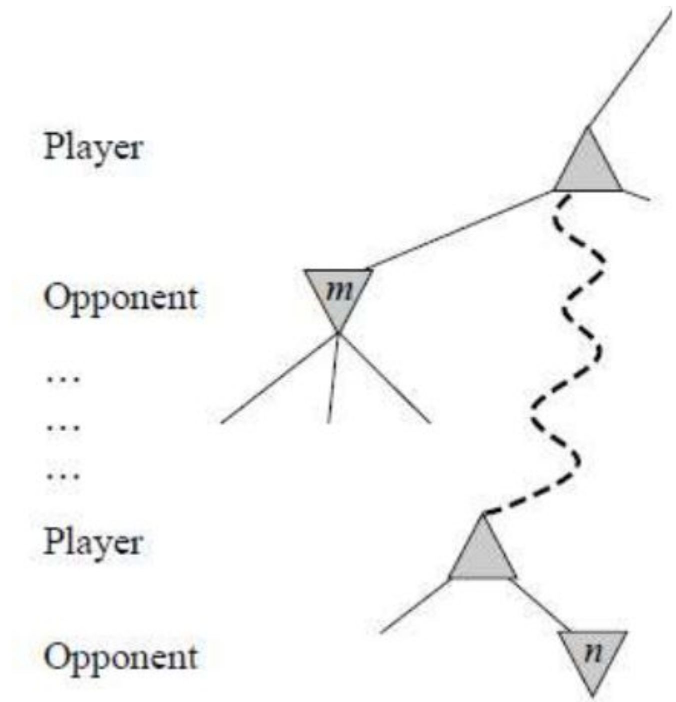



```
MiniMax(MAX)=
   MAX_Value(
      MIN(7,50,25),   //Child1
      MIN(5,10,15),   //Child2
      MIN(75,12,5)    //Child3
     )
```

MAX(**7, 5 or less, …**) = MAX(7,…) //after expanding **5**

# Basic Idea of α-β Pruning

- Consider a node **n** such that Player has a choice of moving to
- If Player has a **better choice m** either at the parent of n or at any choice point further up, then **n will never be reached in actual play**
- α-β pruning gets it name from the two parameters that describe bounds on the backed-up values



# α-β Pruning

- α = the value of the best (**highest-value**) choice we have found so far at any choice point along the path for **MAX**
- β = the value of the best (**lowest-value**) choice we have found so far at any choice point along the path for **MIN**
- α-β search updates the values of α and β as it goes along and **prunes the remaining branches** at a node as soon as the value of the current node is **worse than the current α or β for MAX or MIN** respectively

```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game

   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v
```

---

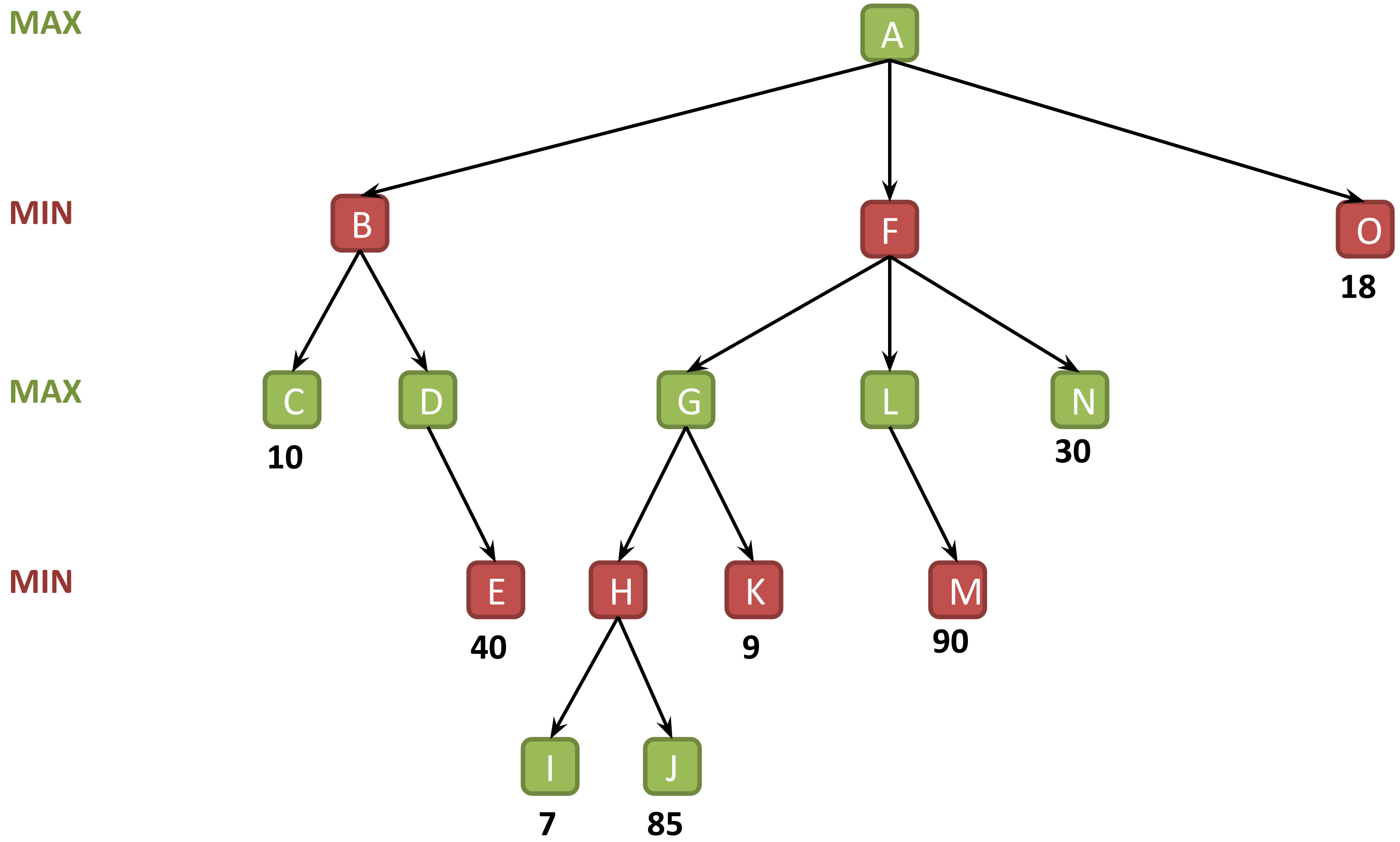
```
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

---

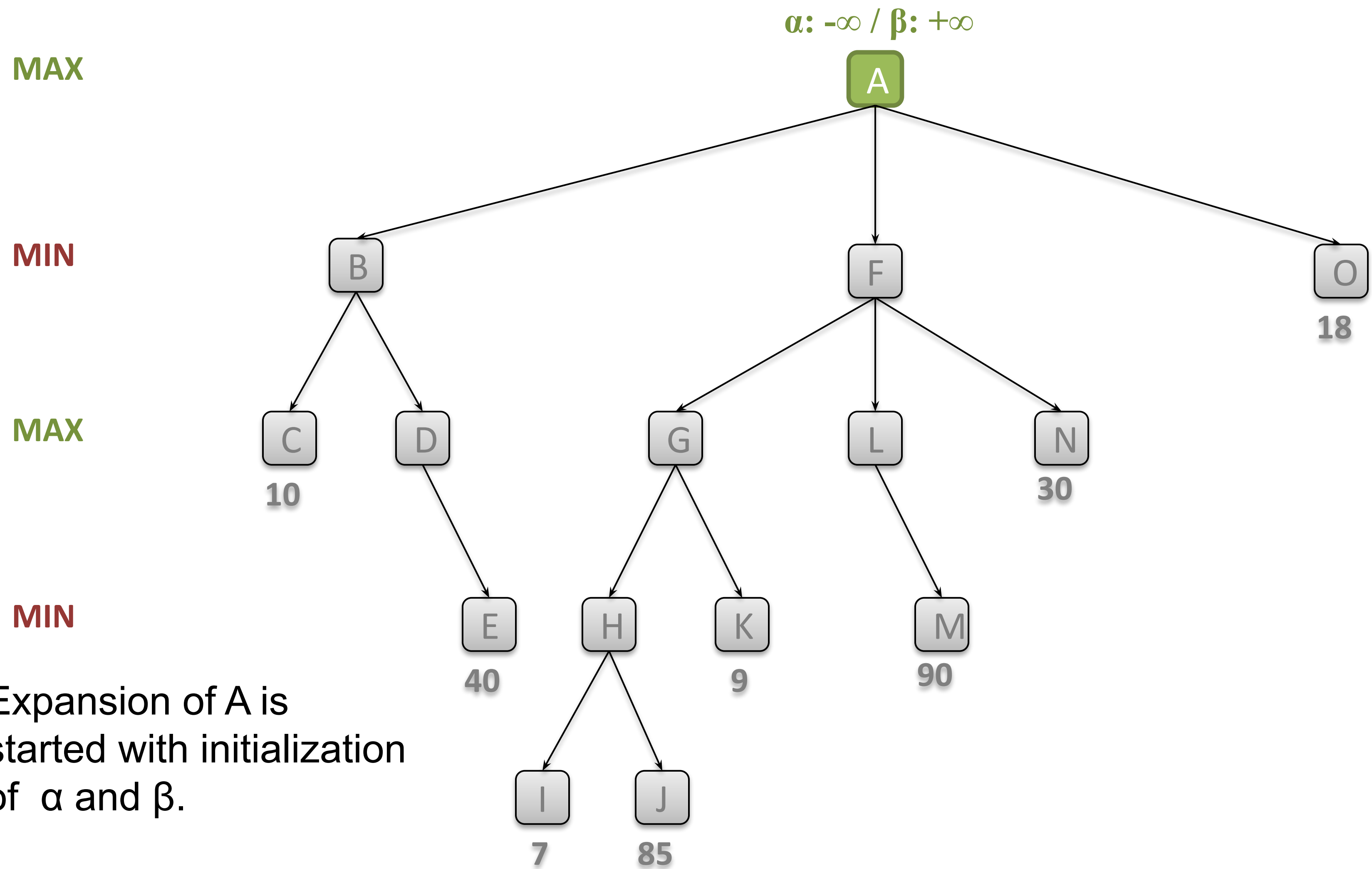
```
function MIN-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s, α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

# α-β Pruning Example

# α-β Pruning

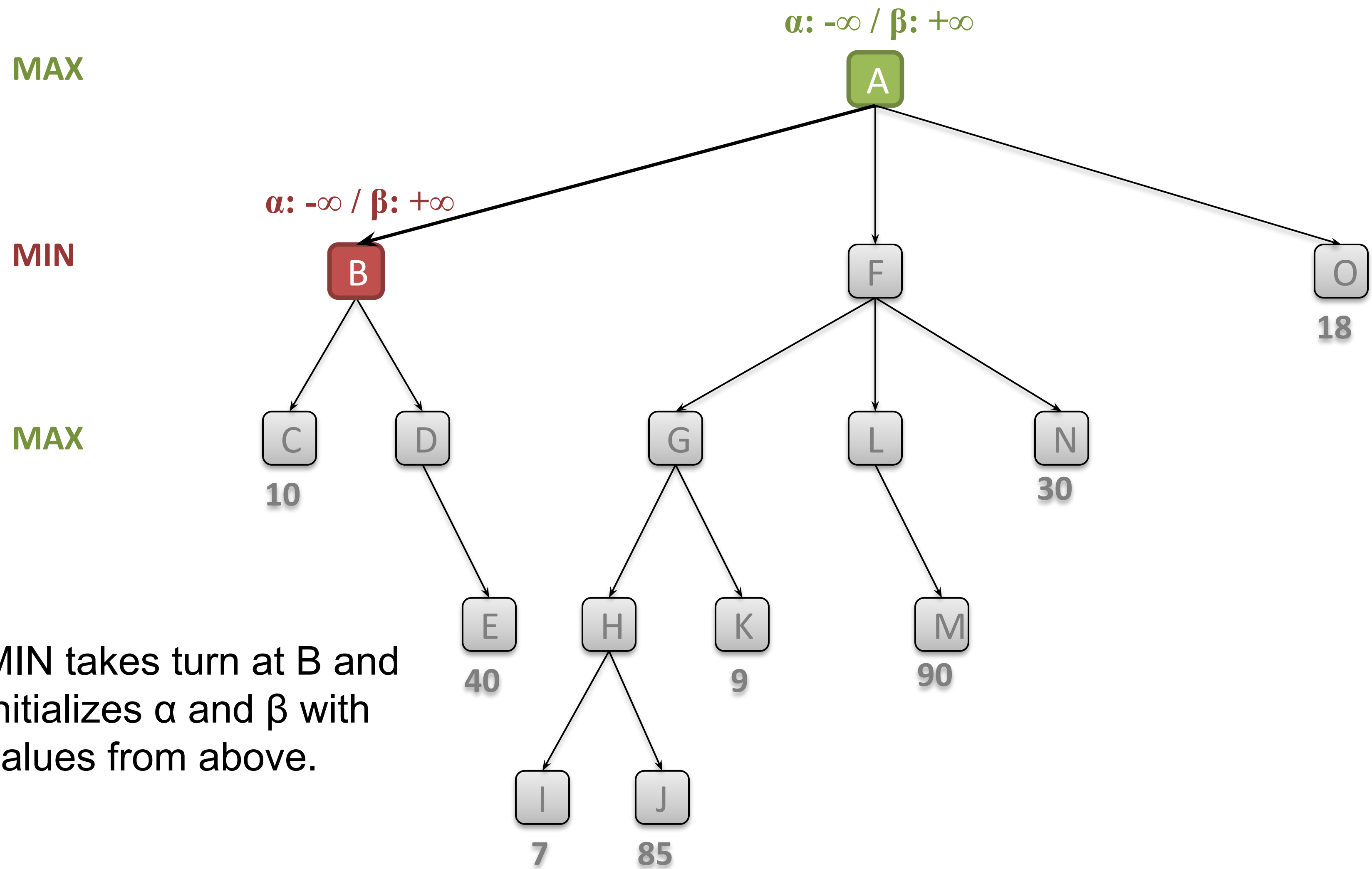


# α-β Pruning



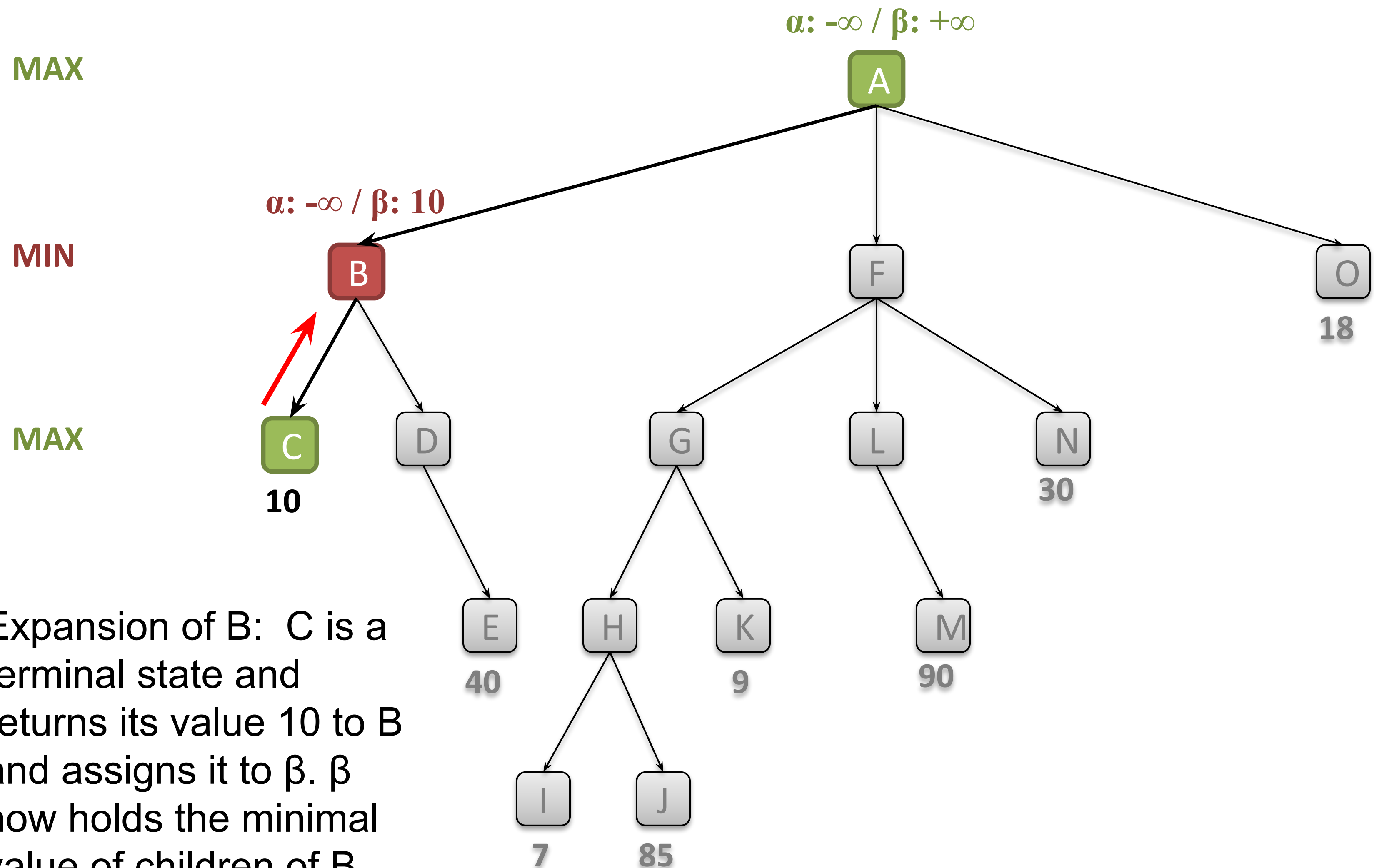

Expansion of A is started with initialization of α and β.

# α-β Pruning



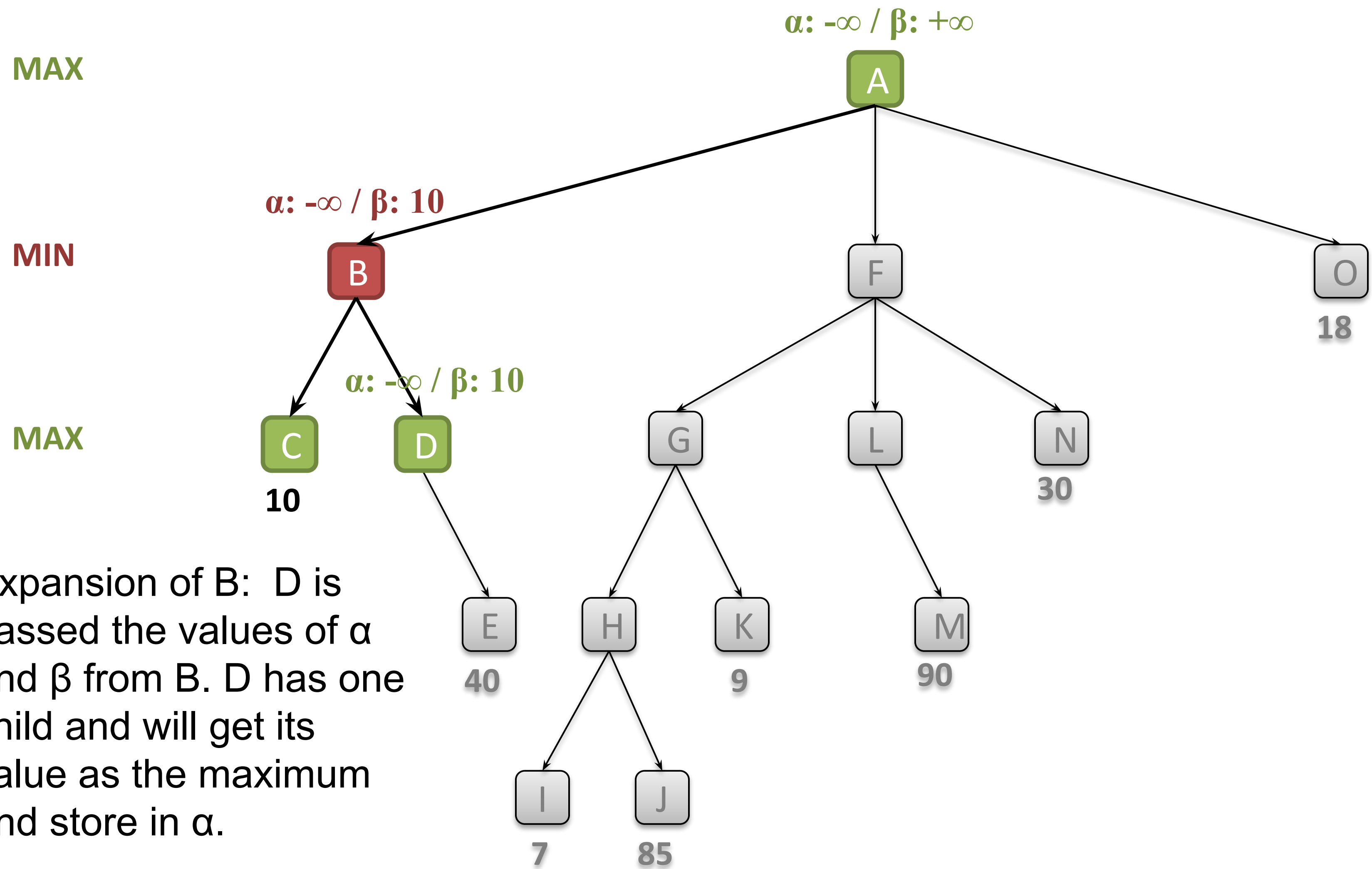

MIN takes turn at B and initializes α and β with values from above.

# α-β Pruning



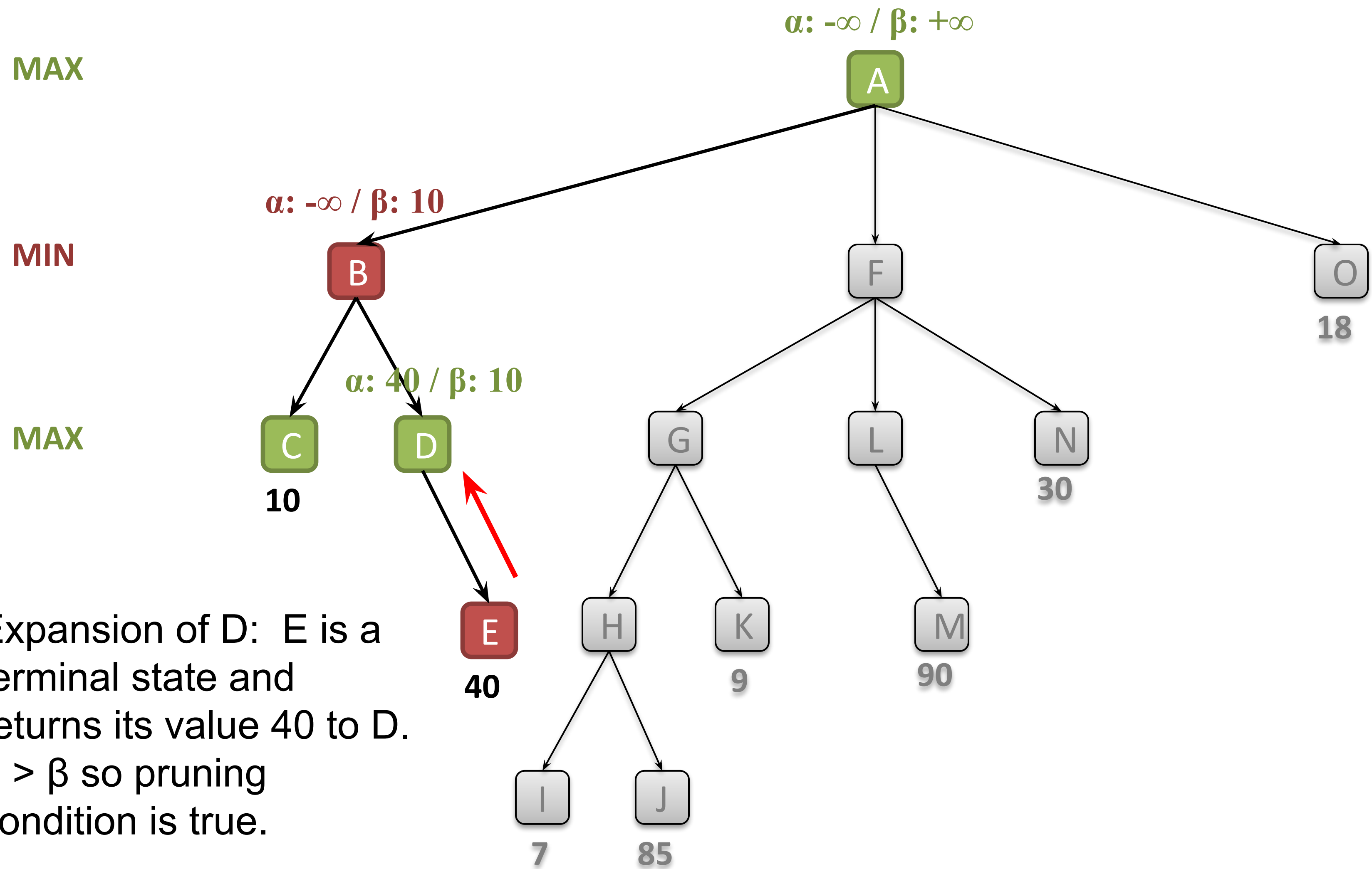

Expansion of B: C is a terminal state and returns its value 10 to B and assigns it to β. β now holds the minimal value of children of B.

# α-β Pruning



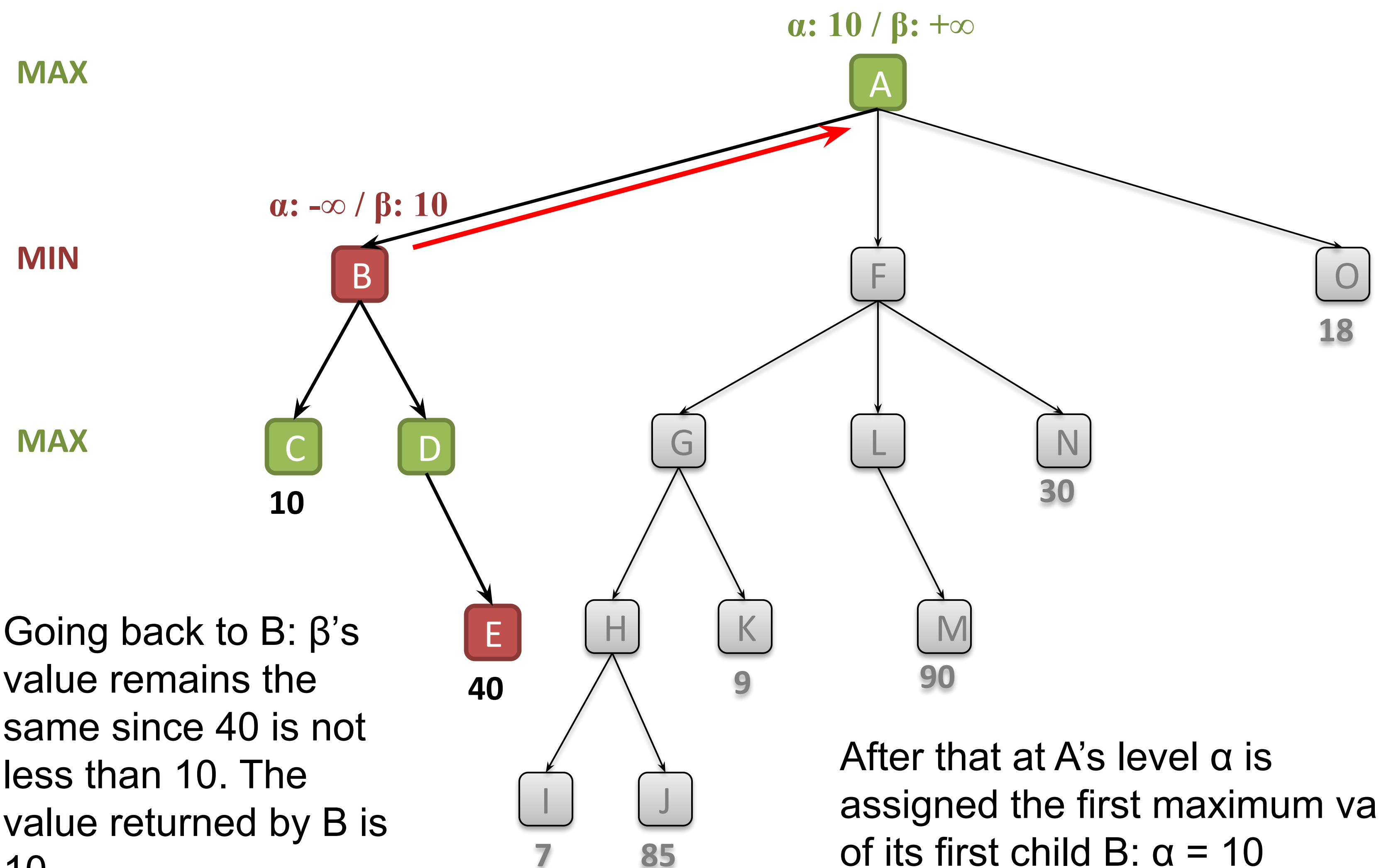

Expansion of B: D is passed the values of α and β from B. D has one child and will get its value as the maximum and store in α.

# α-β Pruning




Expansion of D: E is a terminal state and returns its value 40 to D. α > β so pruning condition is true.

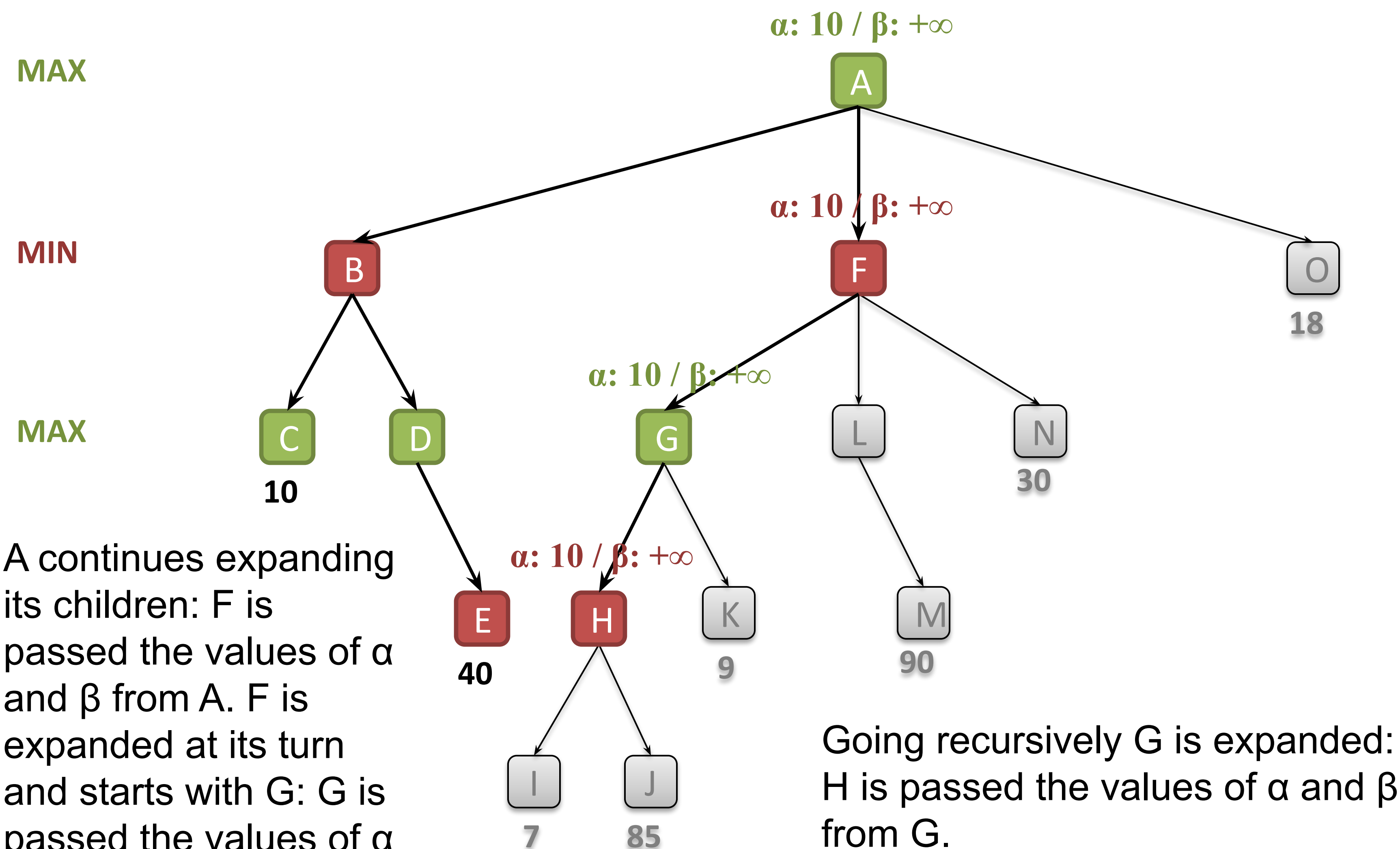
# α-β Pruning




Going back to B: β's value remains the same since 40 is not less than 10. The value returned by B is 10.

After that at A's level α is assigned the first maximum value of its first child B: α = 10

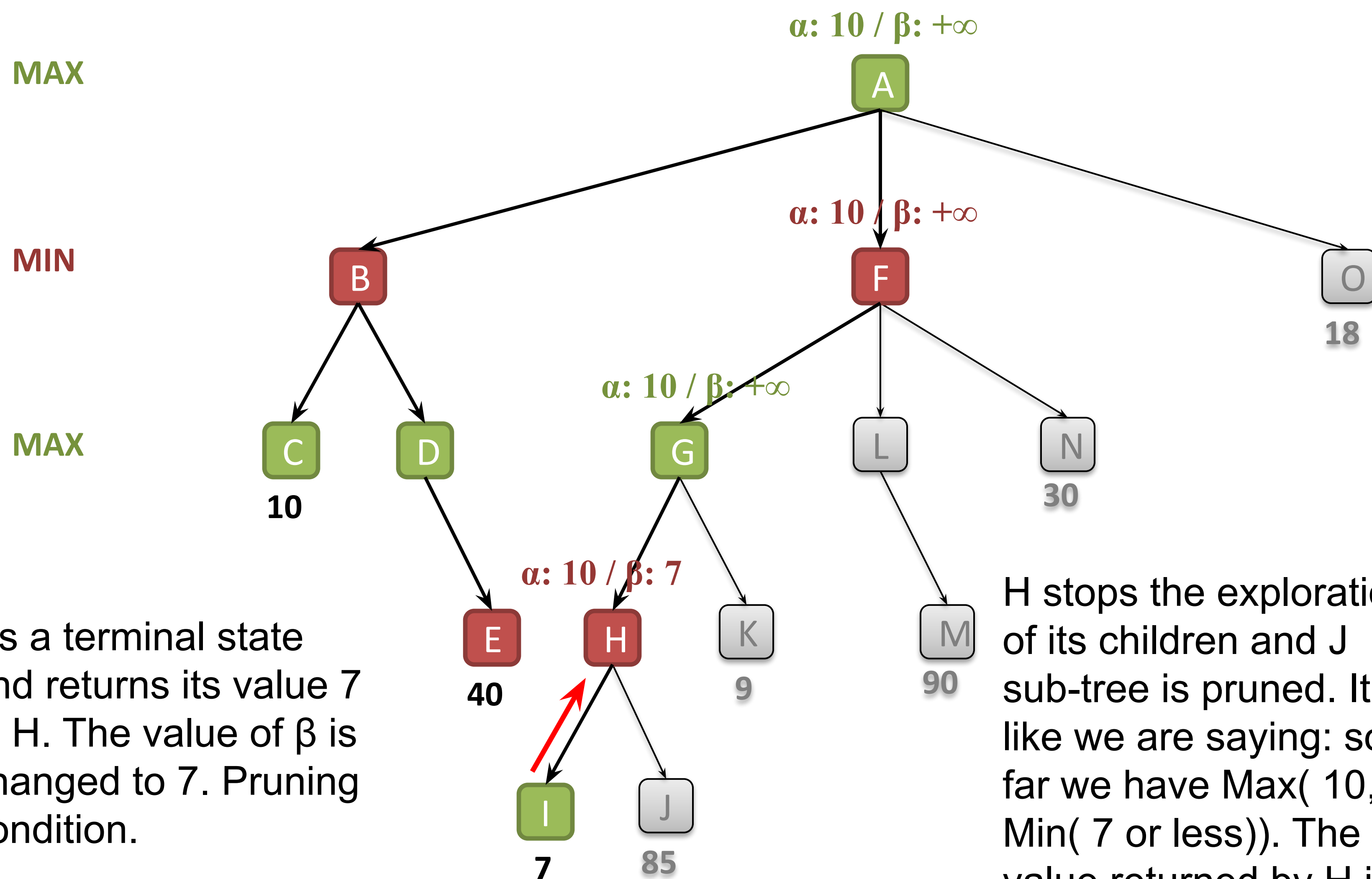
# α-β Pruning




A continues expanding its children: F is passed the values of α and β from A. F is expanded at its turn and starts with G: G is passed the values of α and β from F.

Going recursively G is expanded: H is passed the values of α and β from G.

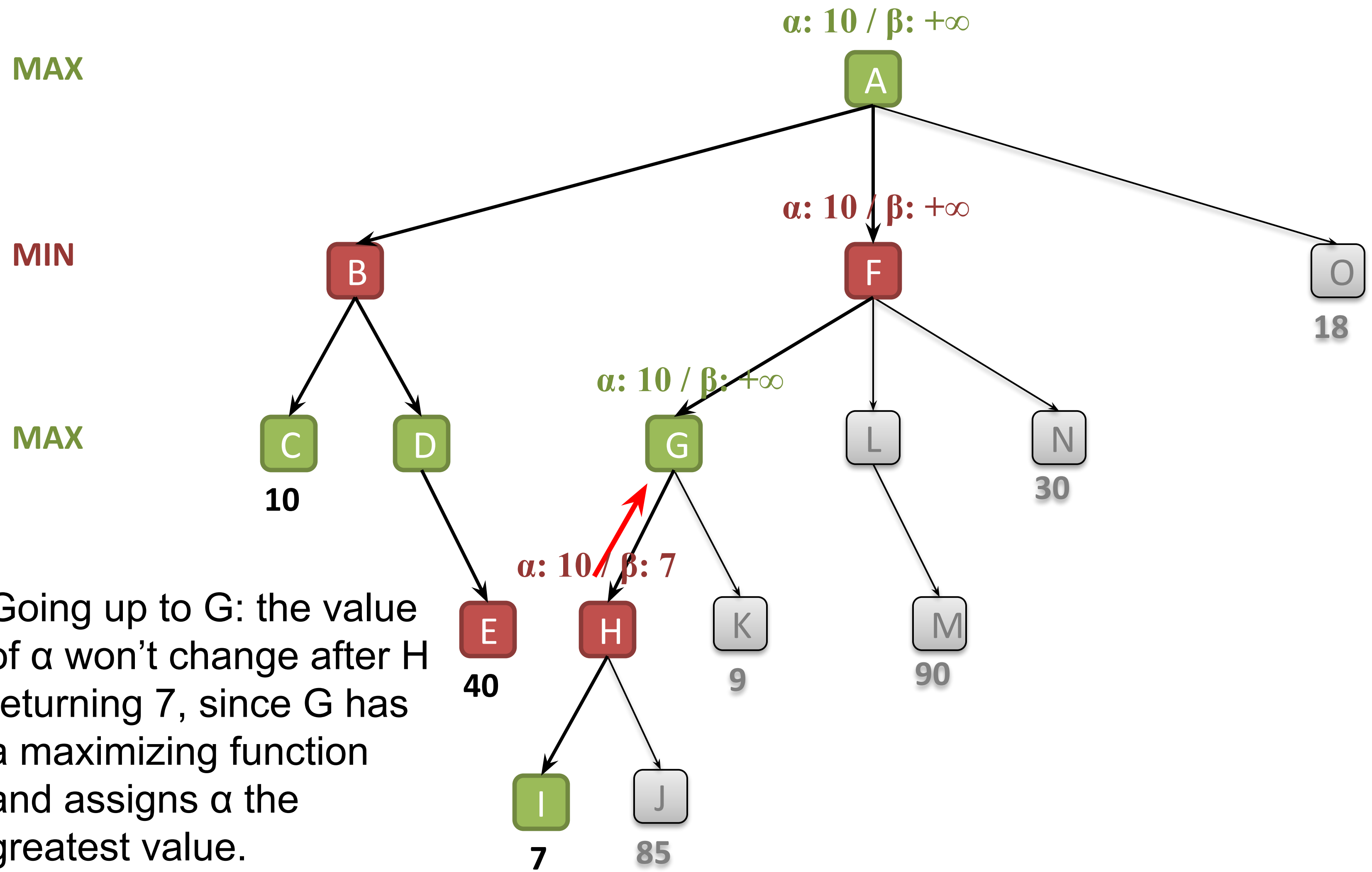
# α-β Pruning




I is a terminal state and returns its value 7 to H. The value of β is changed to 7. Pruning condition.

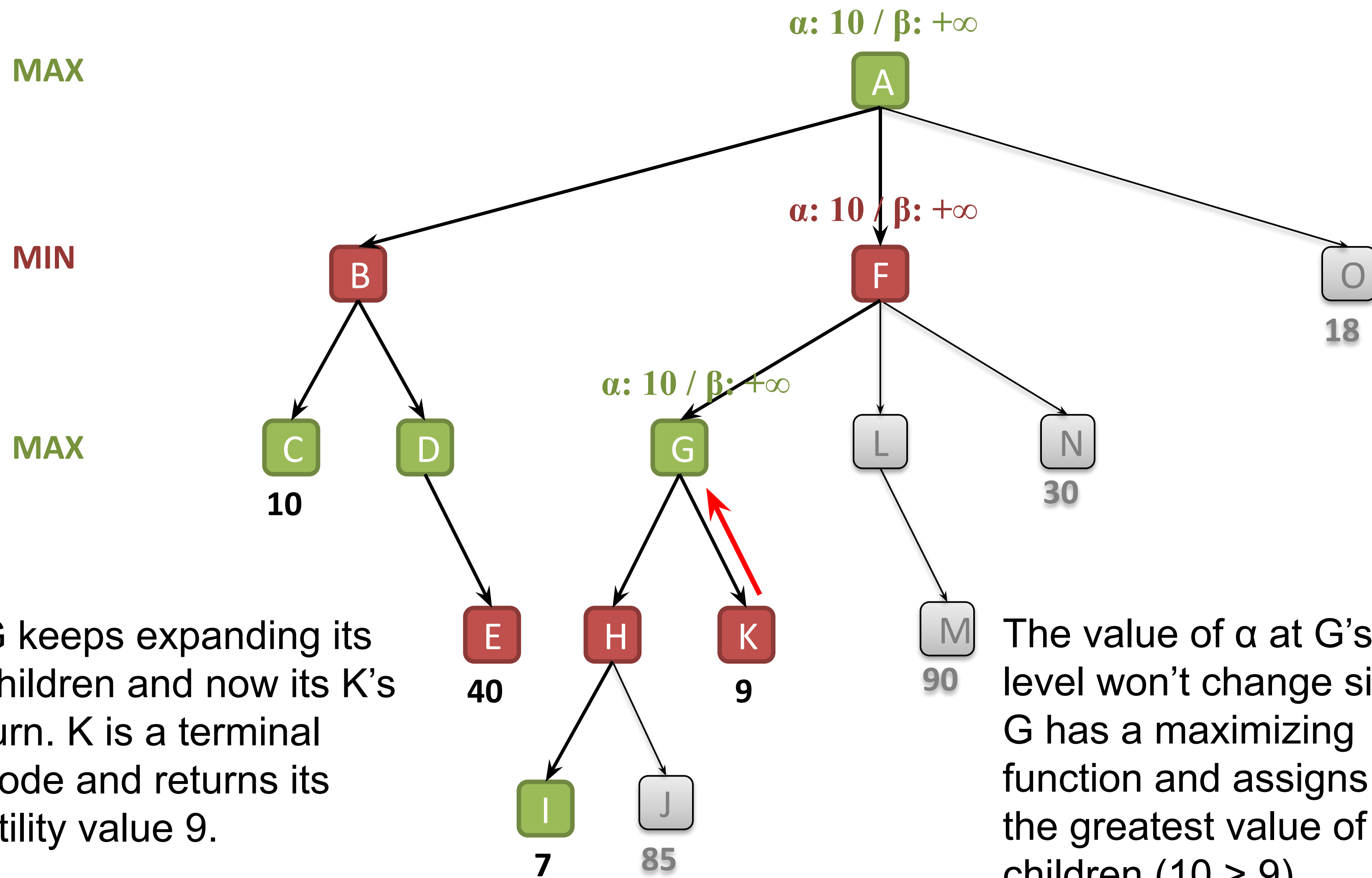
H stops the exploration of its children and J sub-tree is pruned. It's like we are saying: so far we have Max( 10, Min( 7 or less)). The value returned by H is 7.

# α-β Pruning




Going up to G: the value of α won't change after H returning 7, since G has a maximizing function and assigns α the greatest value.

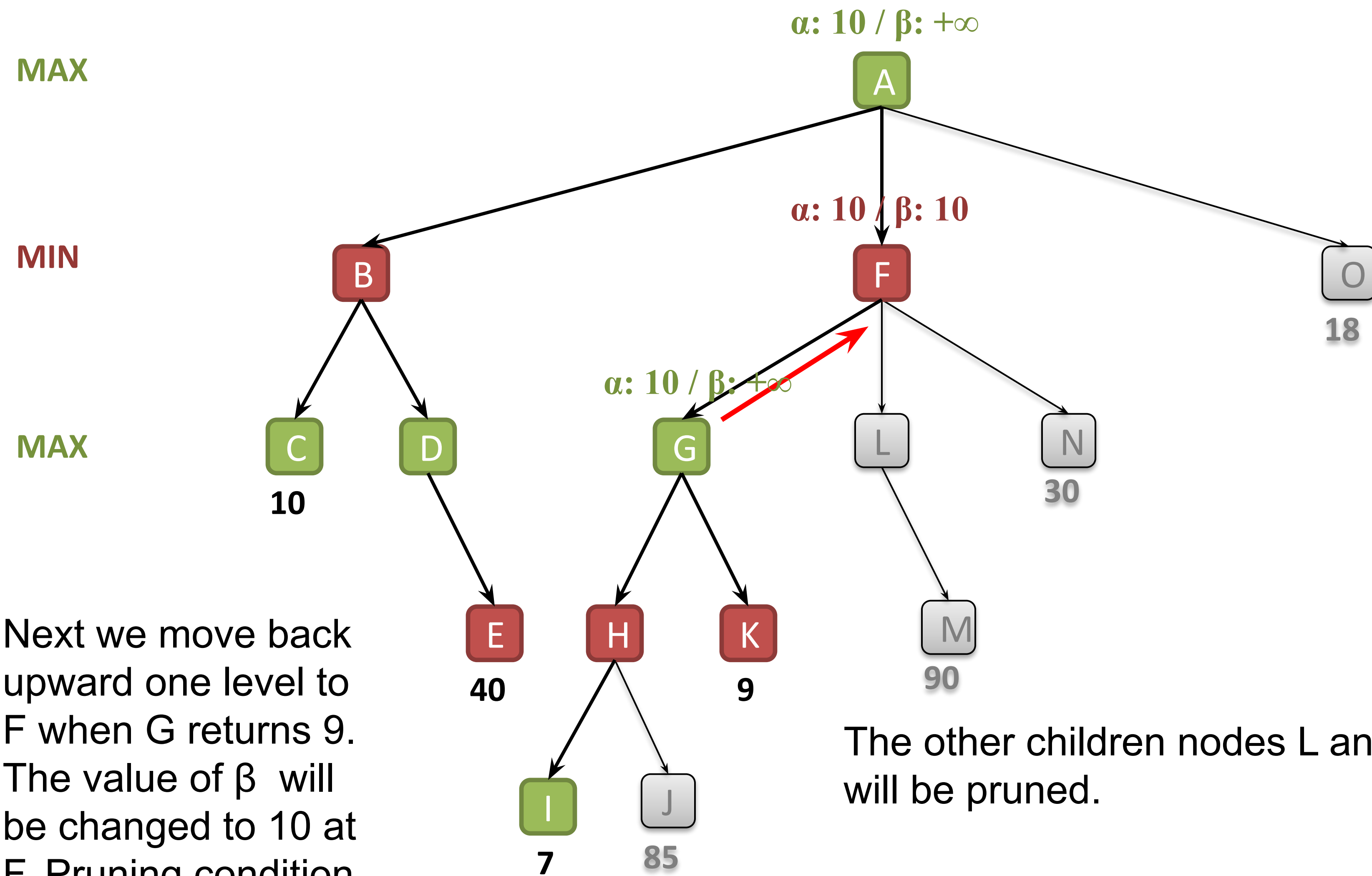
# α-β Pruning




G keeps expanding its children and now its K's turn. K is a terminal node and returns its utility value 9.

The value of α at G's level won't change since G has a maximizing function and assigns α the greatest value of its children (10 > 9).

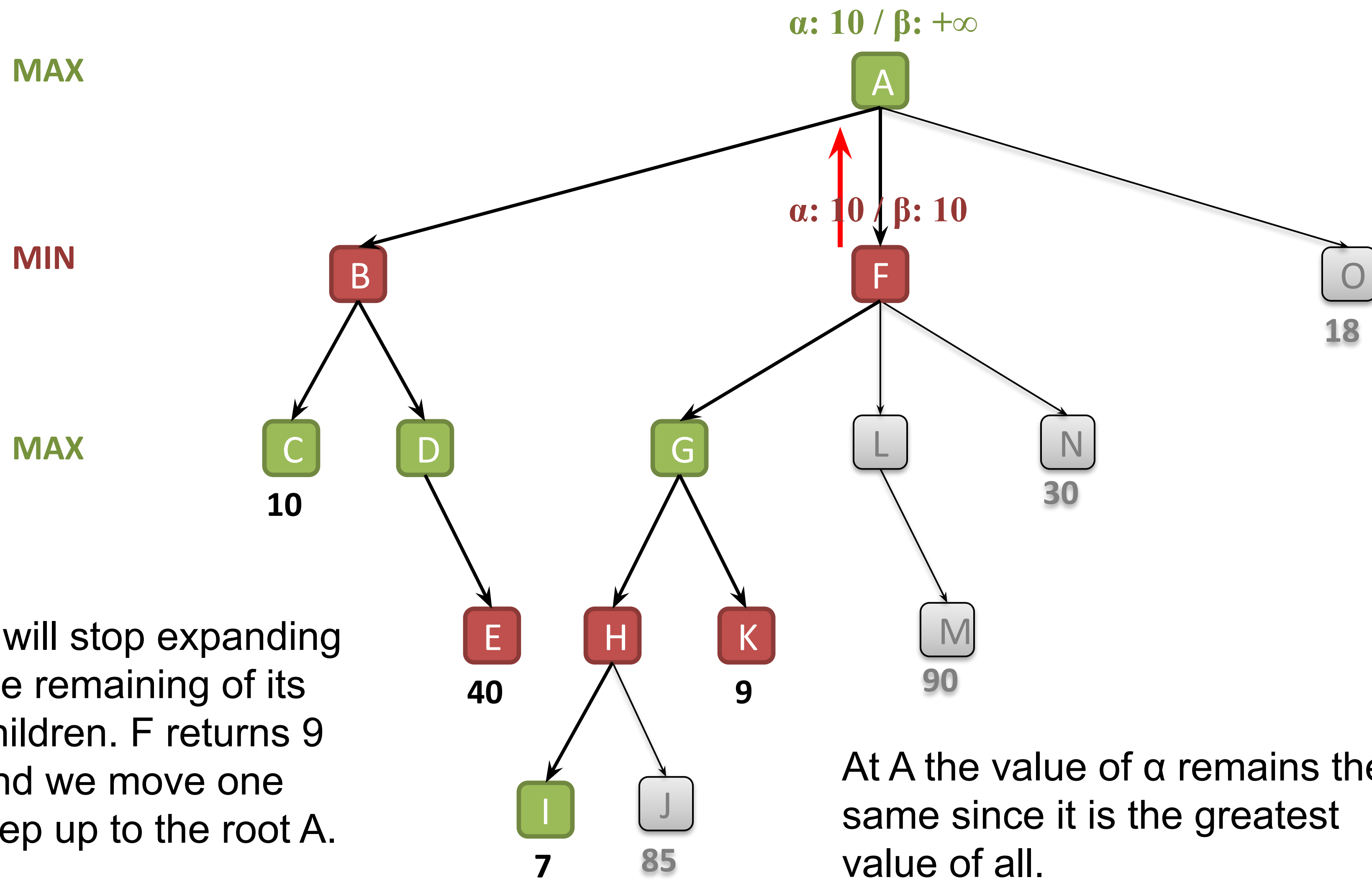
# α-β Pruning




Next we move back upward one level to F when G returns 9. The value of β will be changed to 10 at F. Pruning condition

The other children nodes L and M will be pruned.

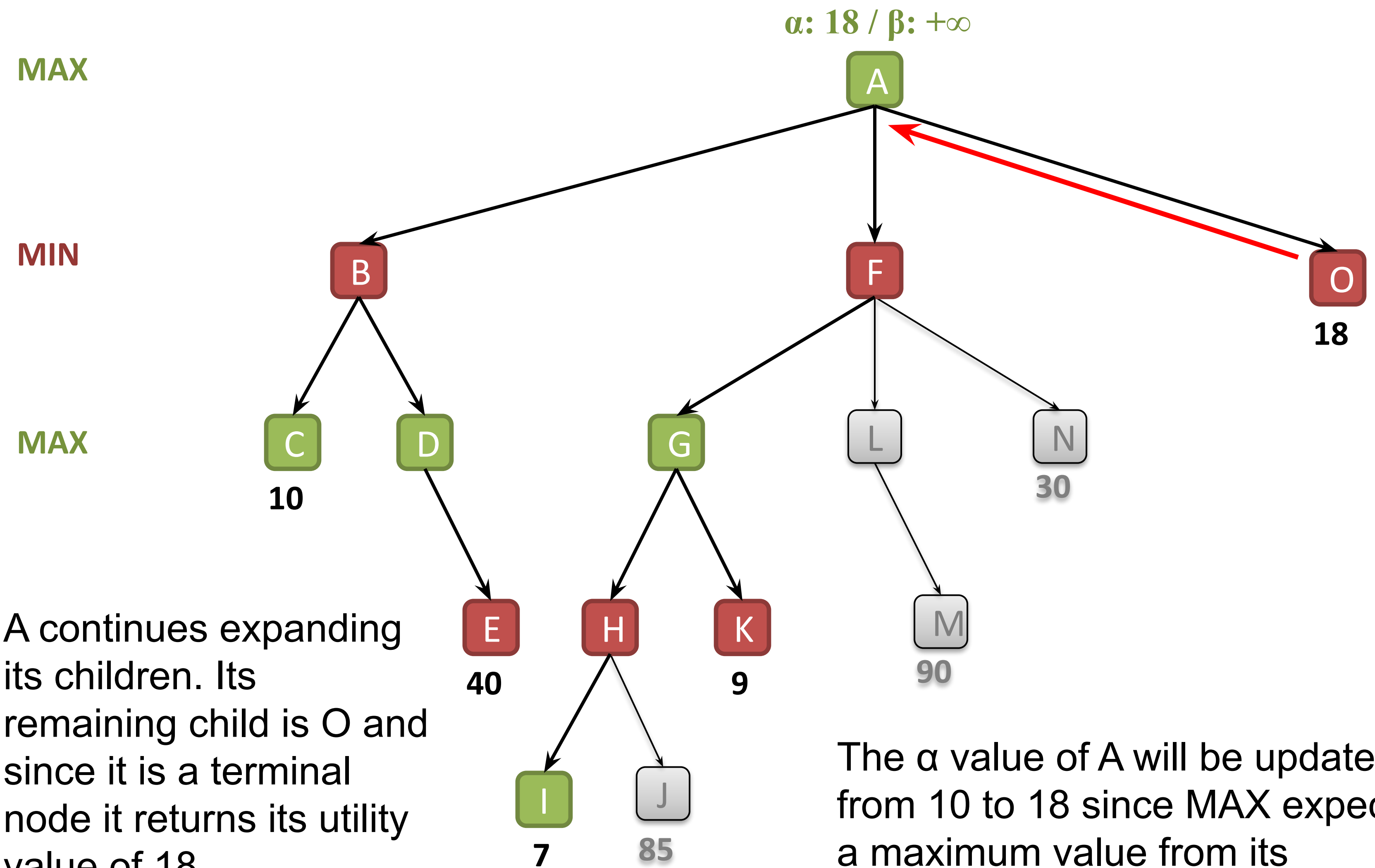
# α-β Pruning




F will stop expanding the remaining of its children. F returns 9 and we move one step up to the root A.

At A the value of α remains the same since it is the greatest value of all.

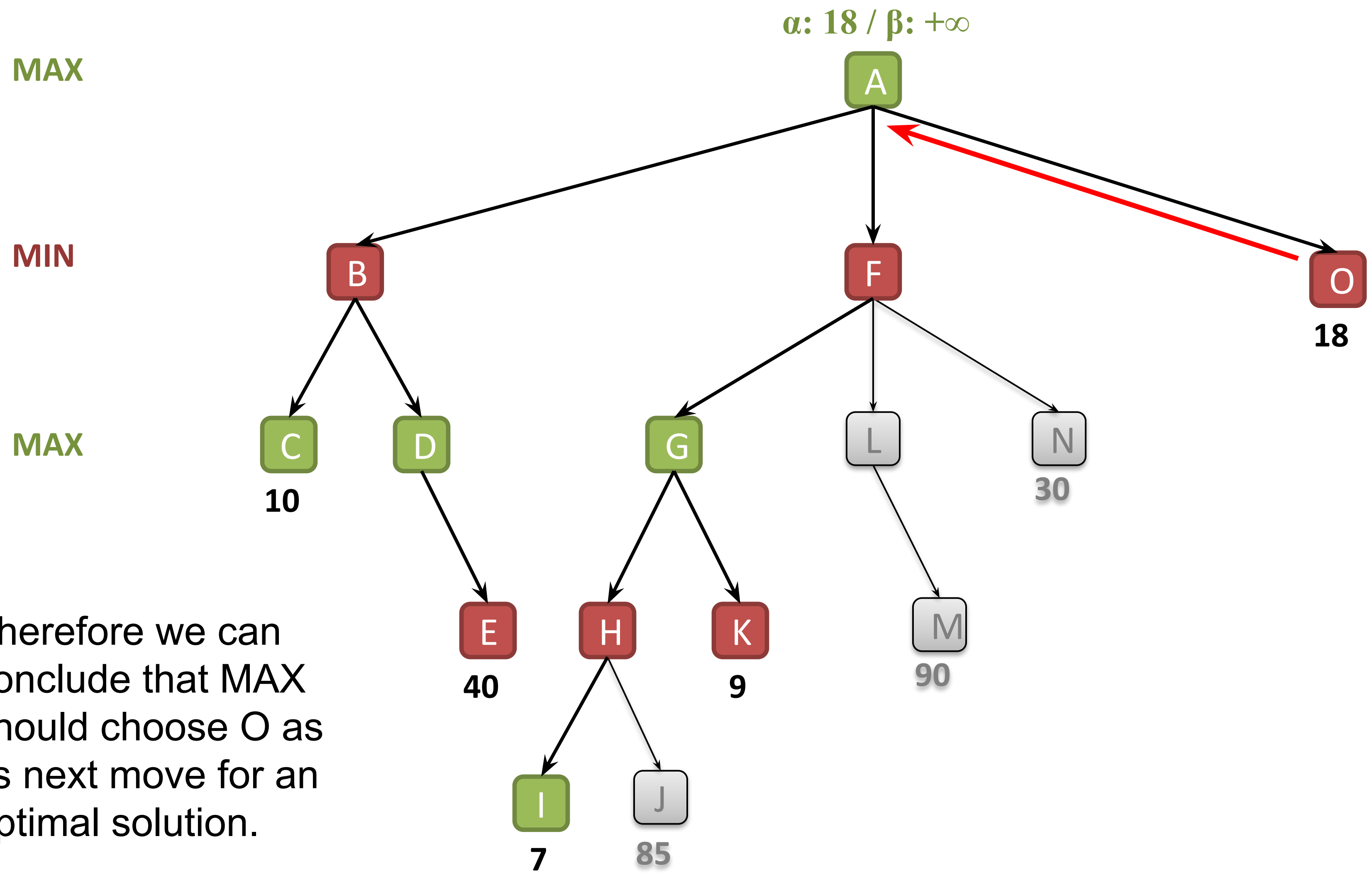
# α-β Pruning




A continues expanding its children. Its remaining child is O and since it is a terminal node it returns its utility value of 18.

The α value of A will be updated from 10 to 18 since MAX expects a maximum value from its children.

# α-β Pruning




Therefore we can conclude that MAX should choose O as its next move for an optimal solution.

# Properties of Alpha-Beta Pruning

- Pruning **does not** affect the final results
- Good move ordering improves effectiveness of pruning
- With “perfect ordering”, time complexity = $O(k^{d/2})$ => doubles depth of search
  - What’s the worse and average case time complexity?
  - Does it make sense then to have good heuristics for which nodes to expand first?

# Readings

- Artificial Intelligence a Modern Approach
  - Chapter 5: Adversarial Search