Adversarial search

- **Goal:** To decide the best movement to play in a game
- There are many kinds of games:
  - Number of players, information known by the players, chance, indeterminism, cooperation/competition, limited resources, ...
- We are going to focus on
  - Two players.
  - Alternating decisions (MAX player, MIN player)
  - Perfect information
  - For example: chess, go, checkers, othello,...

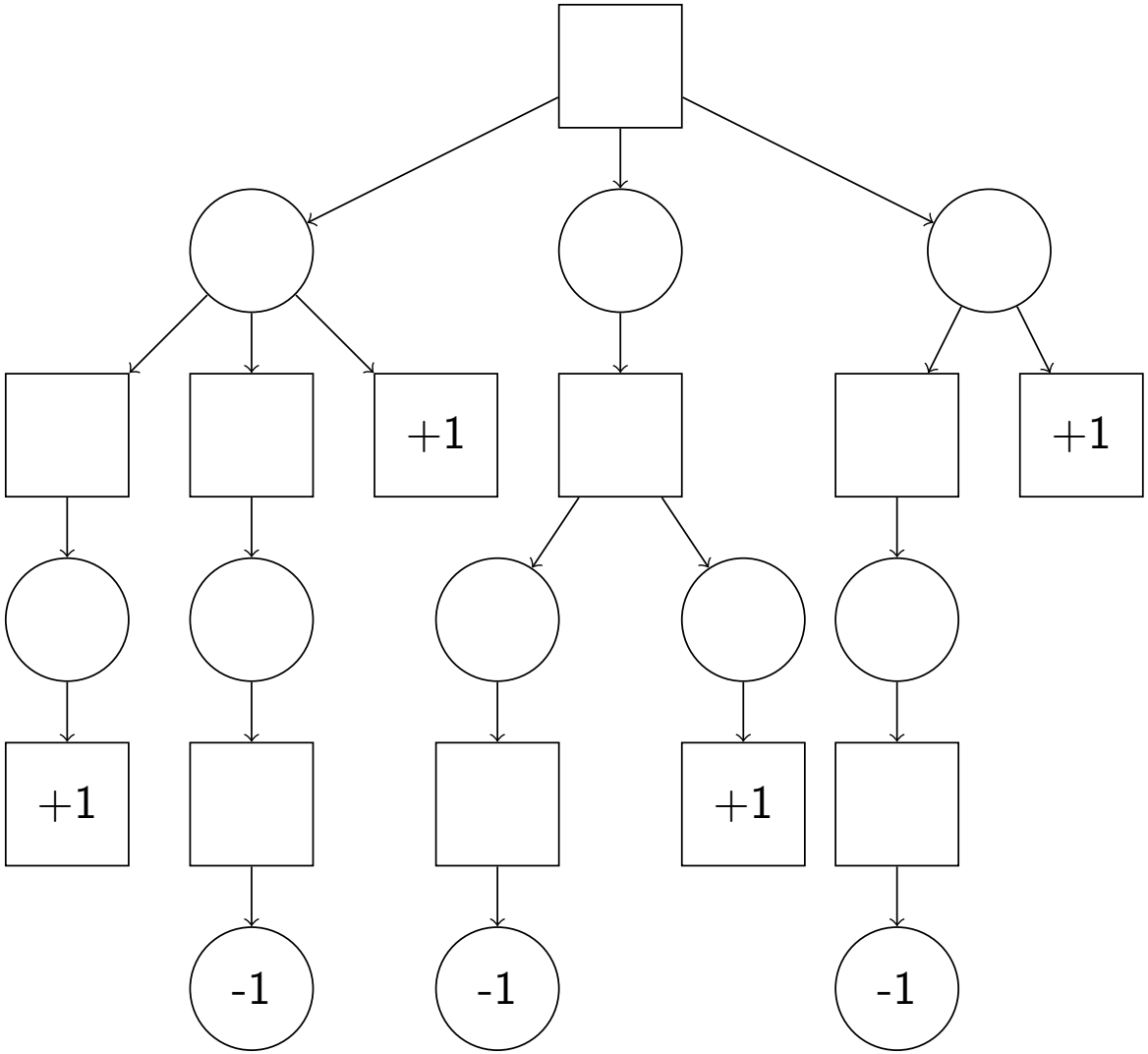
Representation of a game

- Can be defined as state space problems
  - State = Elements of the game
  - Goal states = Winning states (Defined by its properties)
  - Actions of the game = Rules of the game
- They are problems with special characteristics
  - The accesibility of the states depends on the actions chosen by the opponent
  - Two kind of solutions (one for each player)
  - No optimality criteria (all solutions are equally good, the length of the path is not important)

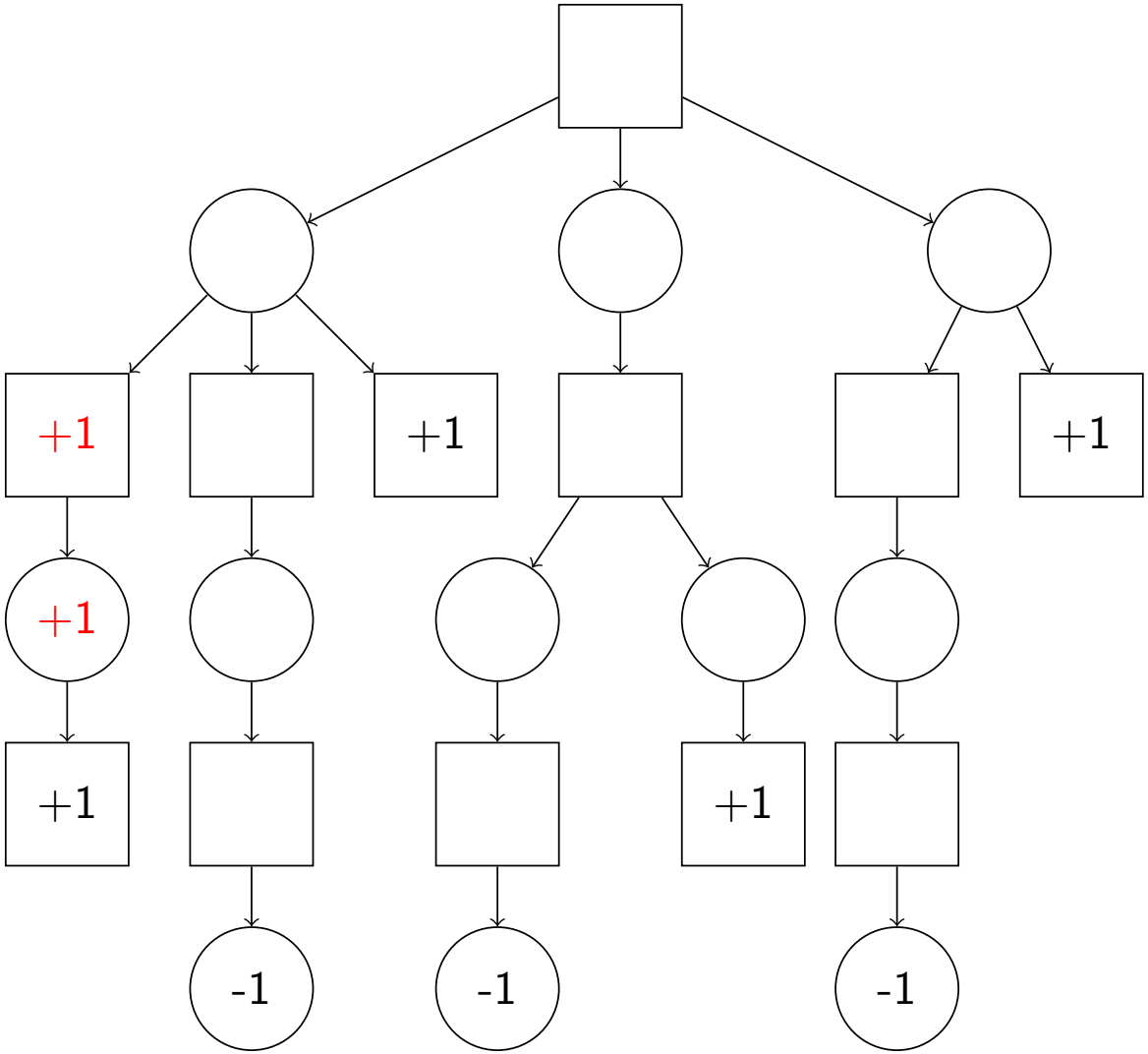
Adversarial search - Brute Force

- The trivial approach is to generate the whole game search tree
- Terminal states are labeled depending on the winner (MAX=+1, MIN=-1)
- The goal is to find a set of movements that allows the MAX player to reach a winning state
- The value of the states are propagated from the terminal states to the root of the search tree
- The path that leads to a winning state is chosen
- A depth-first search is the best choice (minimizes the space cost)
- For games minimally complex this seach is unfeasible (eg.: chess $O(2^{35})$, go $O(2^{300})$)

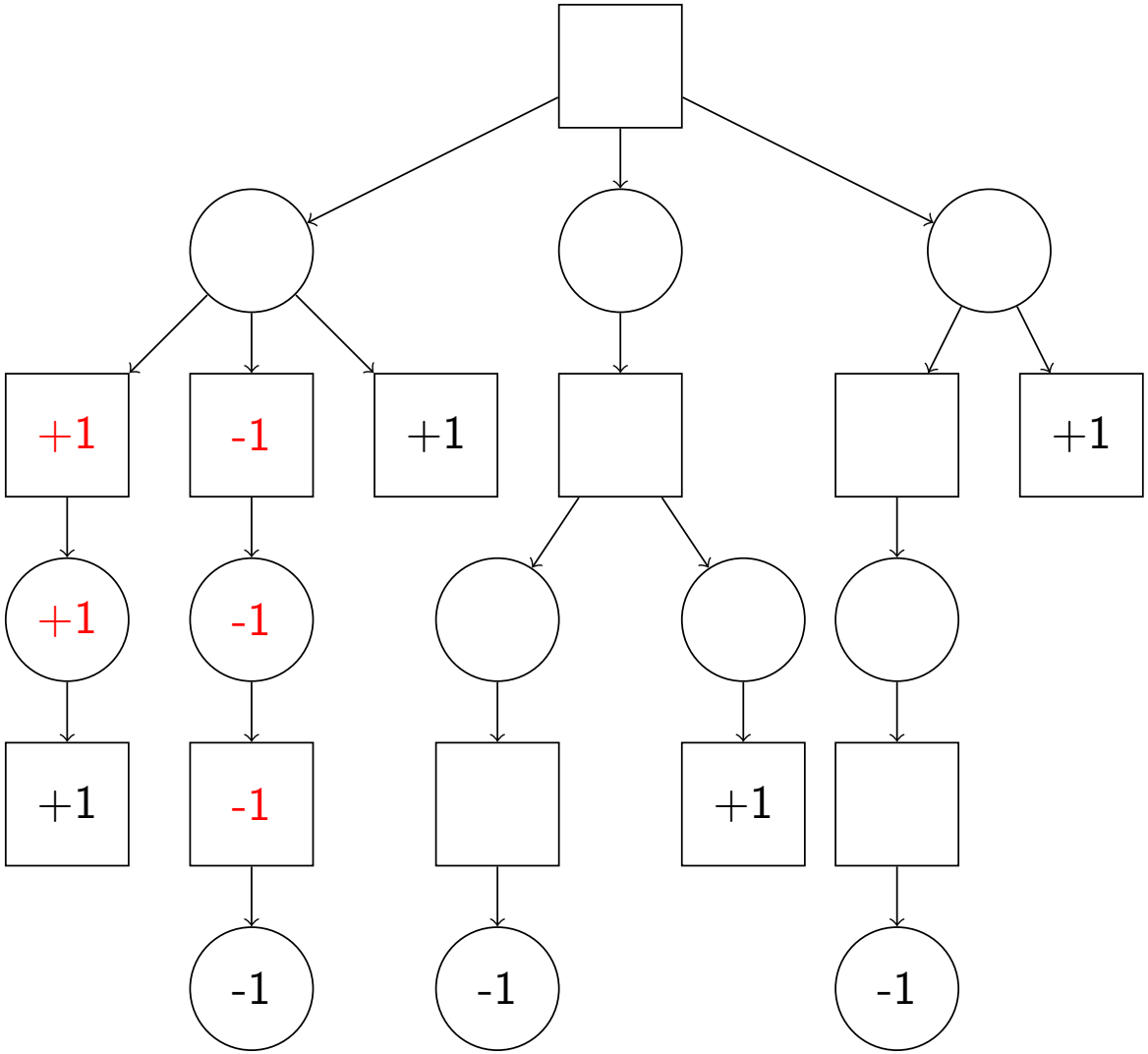
Adversarial search



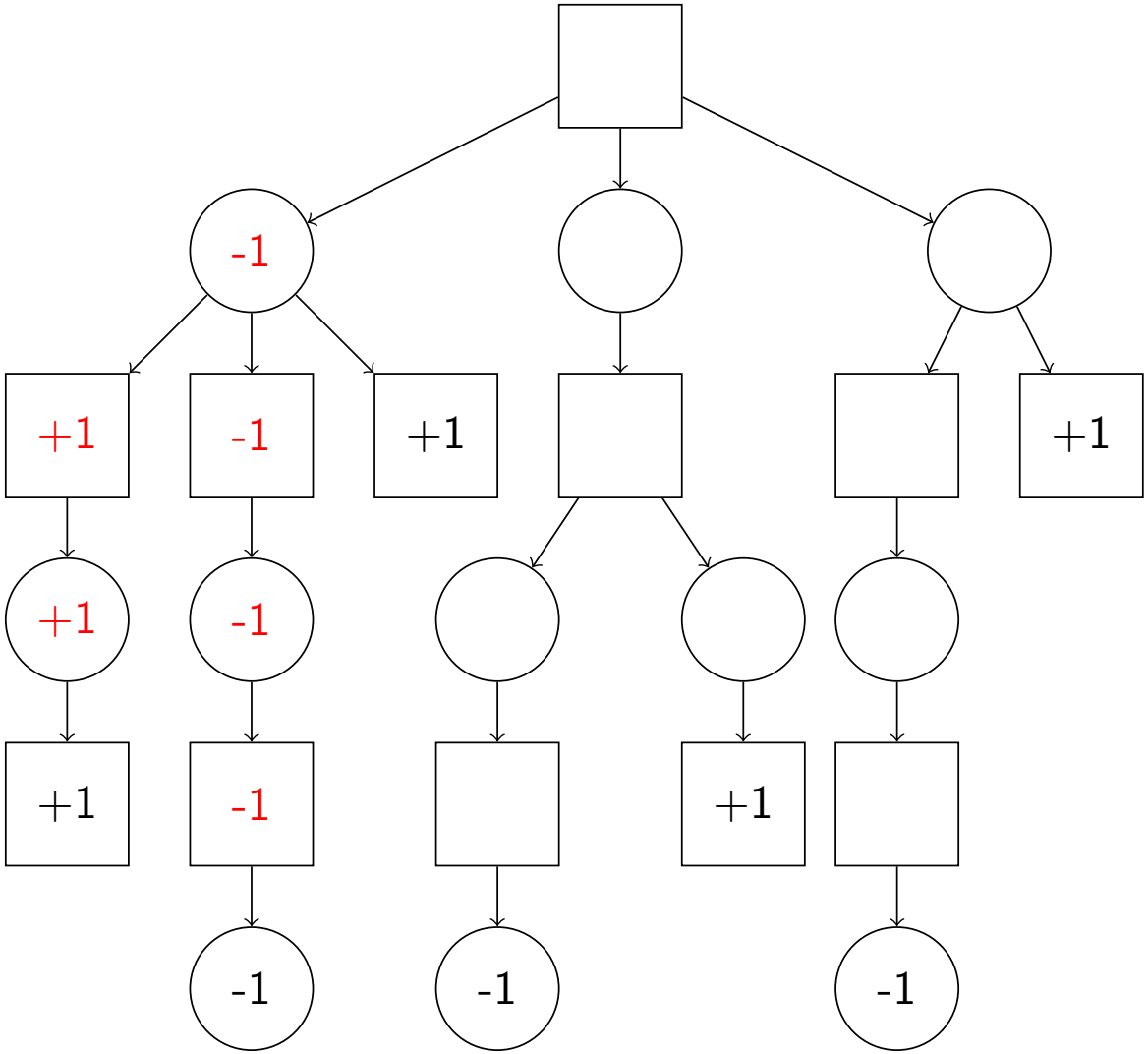
Adversarial search



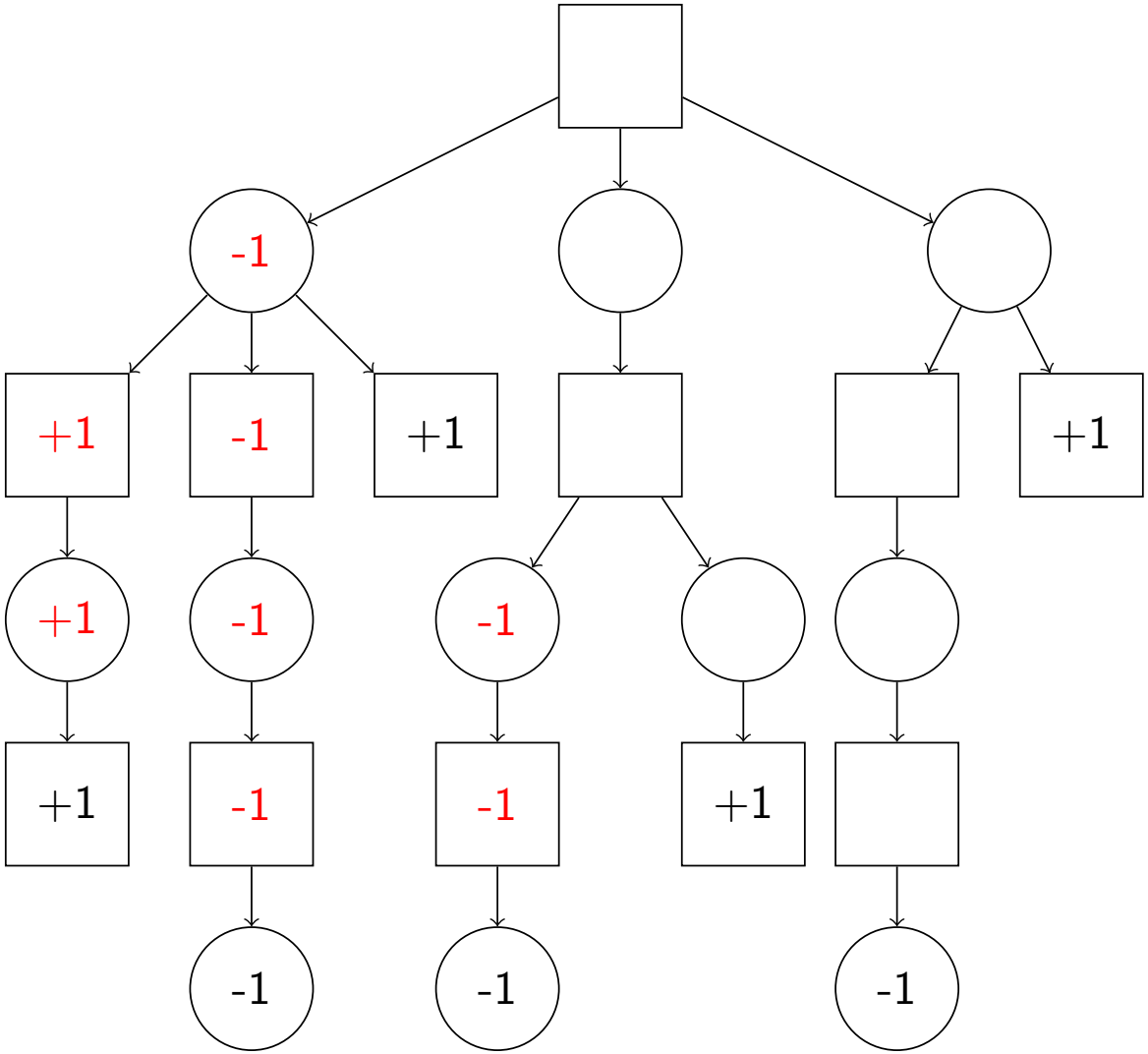
Adversarial search



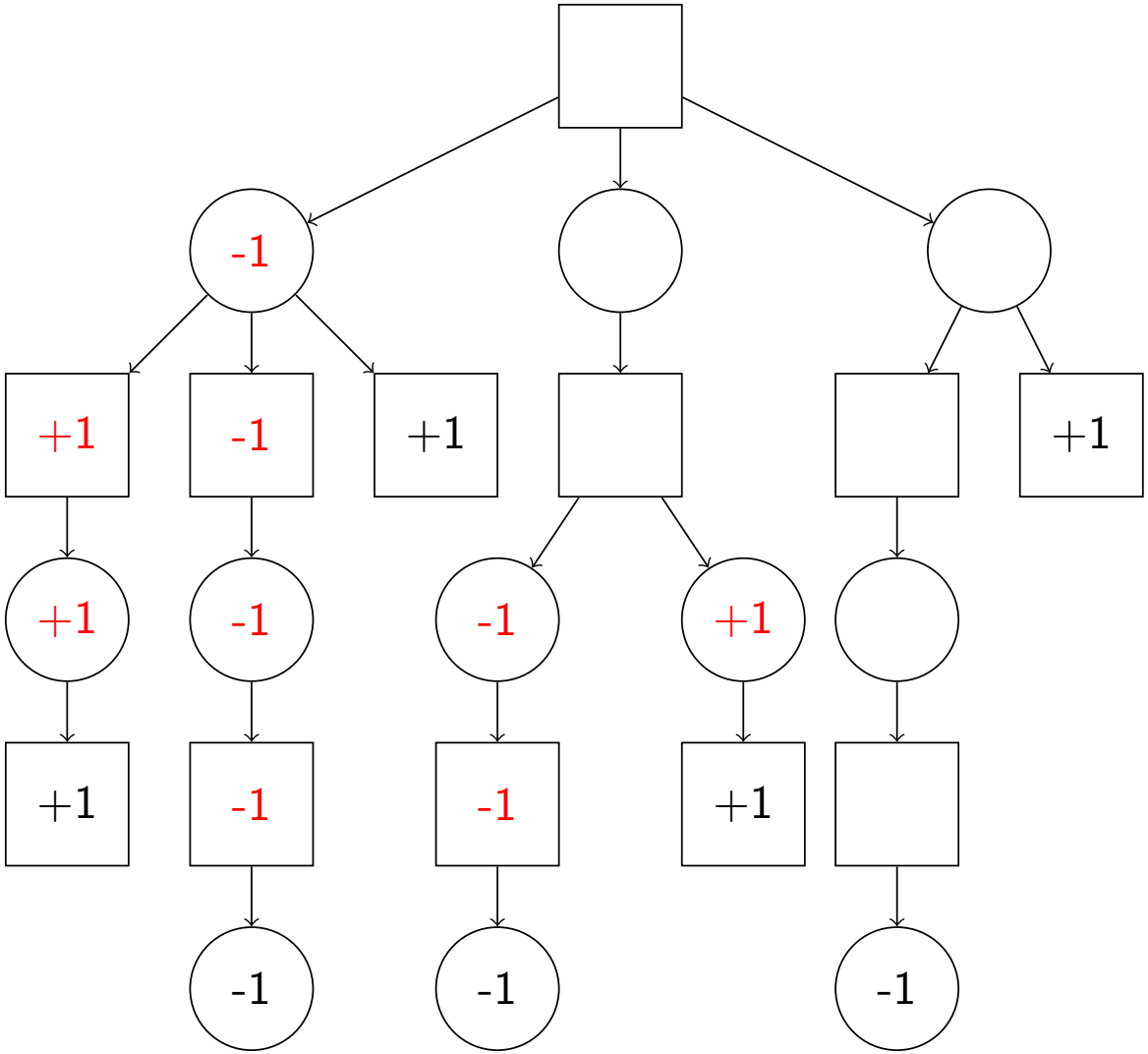
Adversarial search



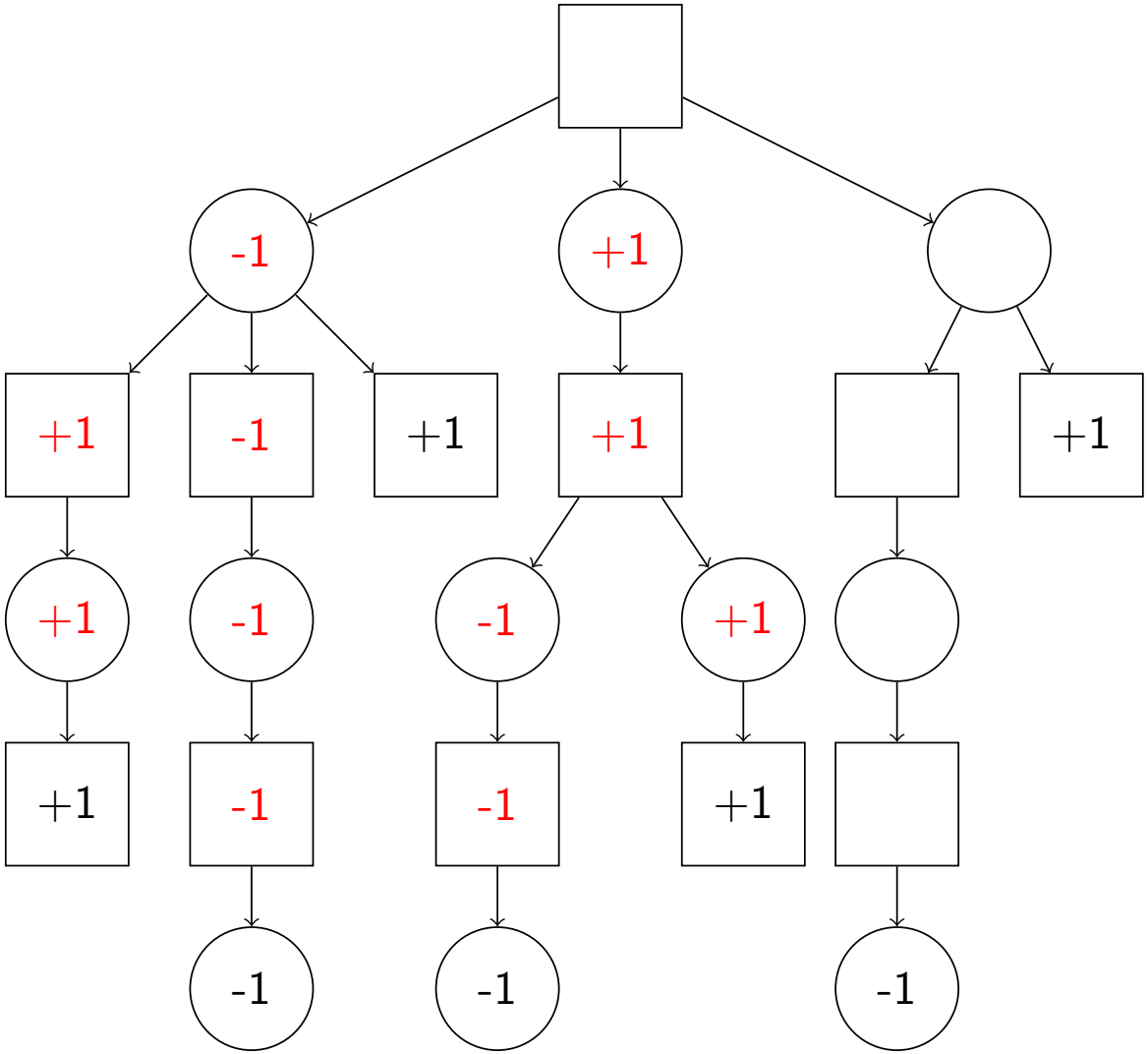
Adversarial search



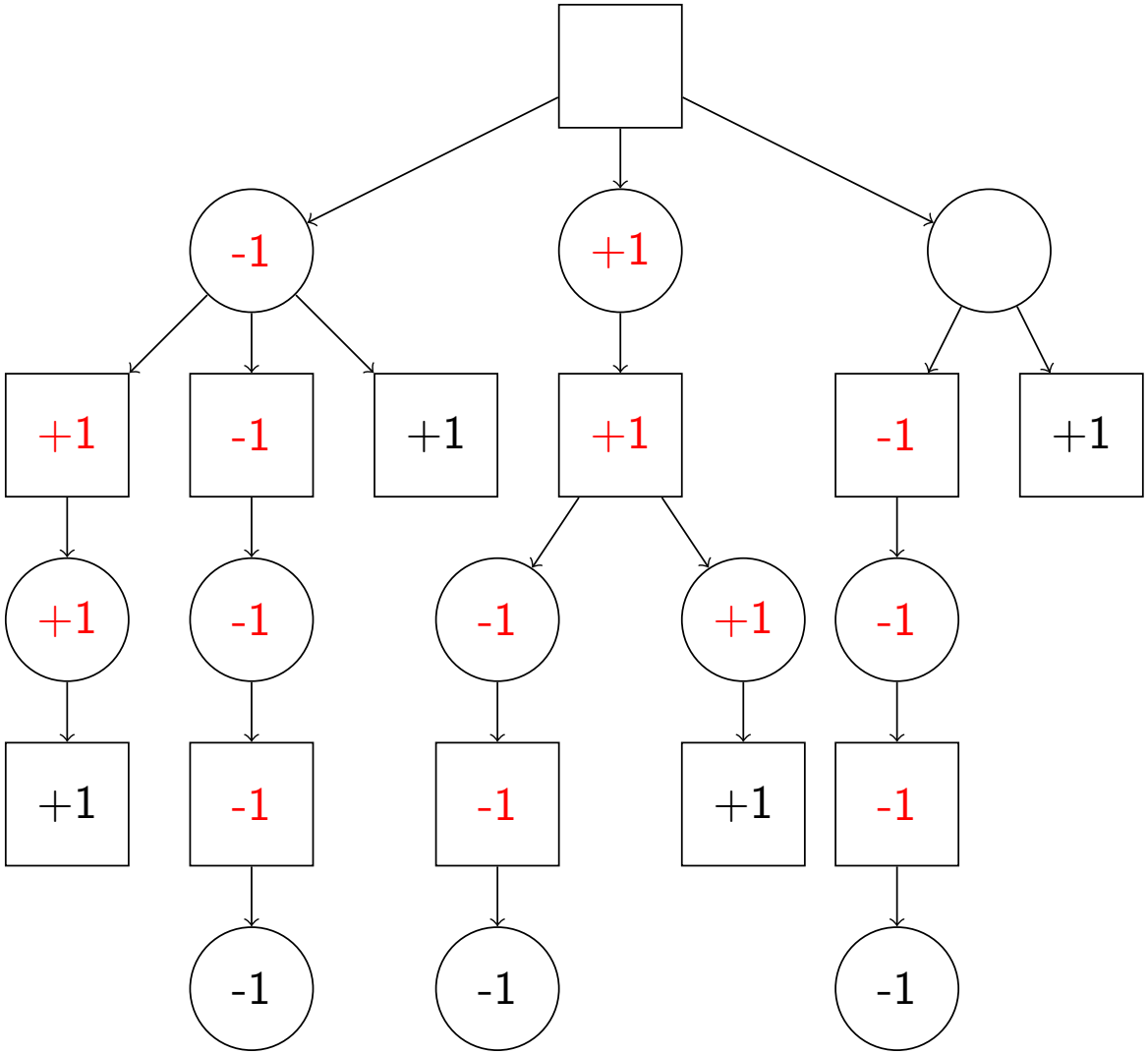
Adversarial search



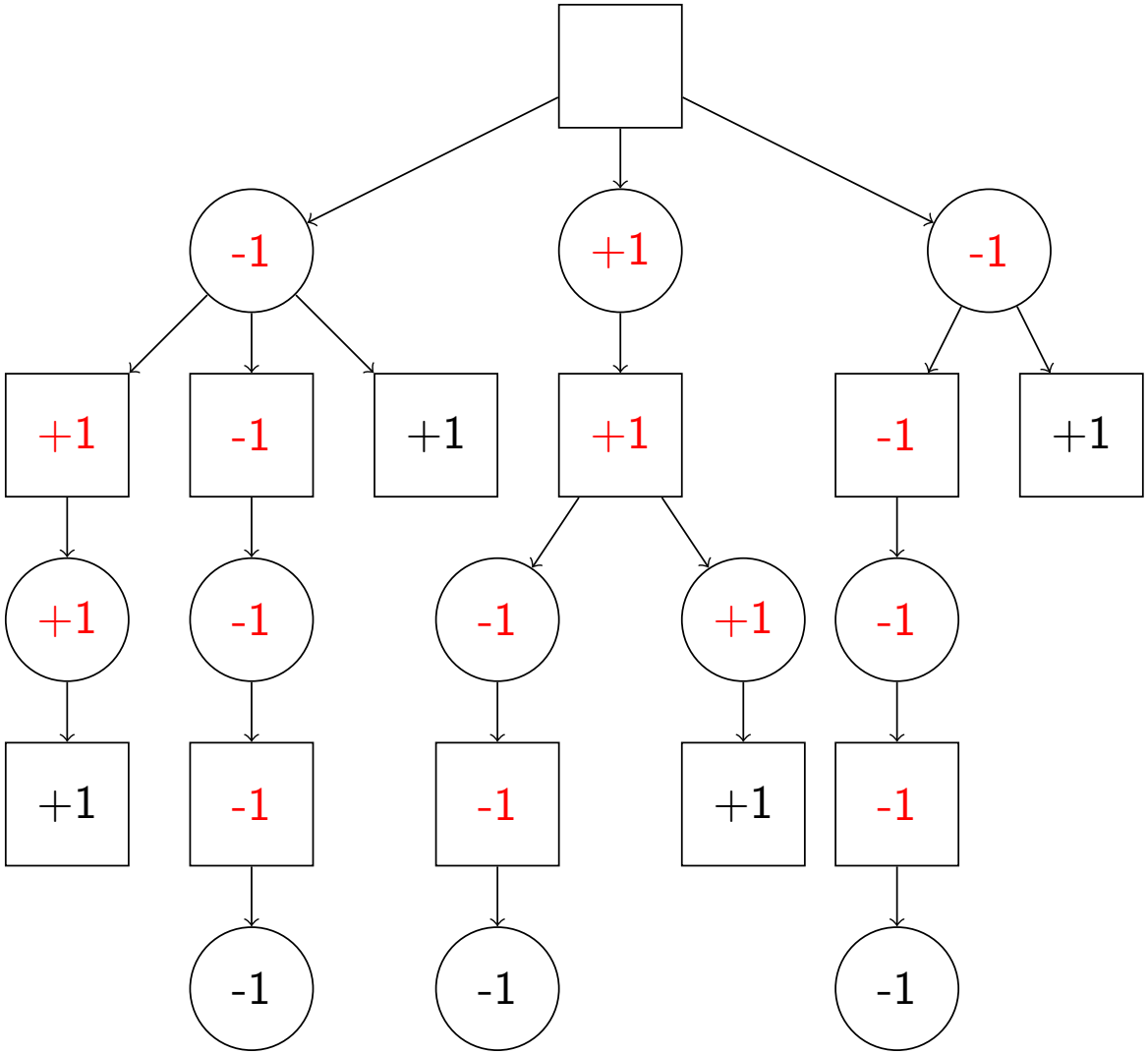
Adversarial search



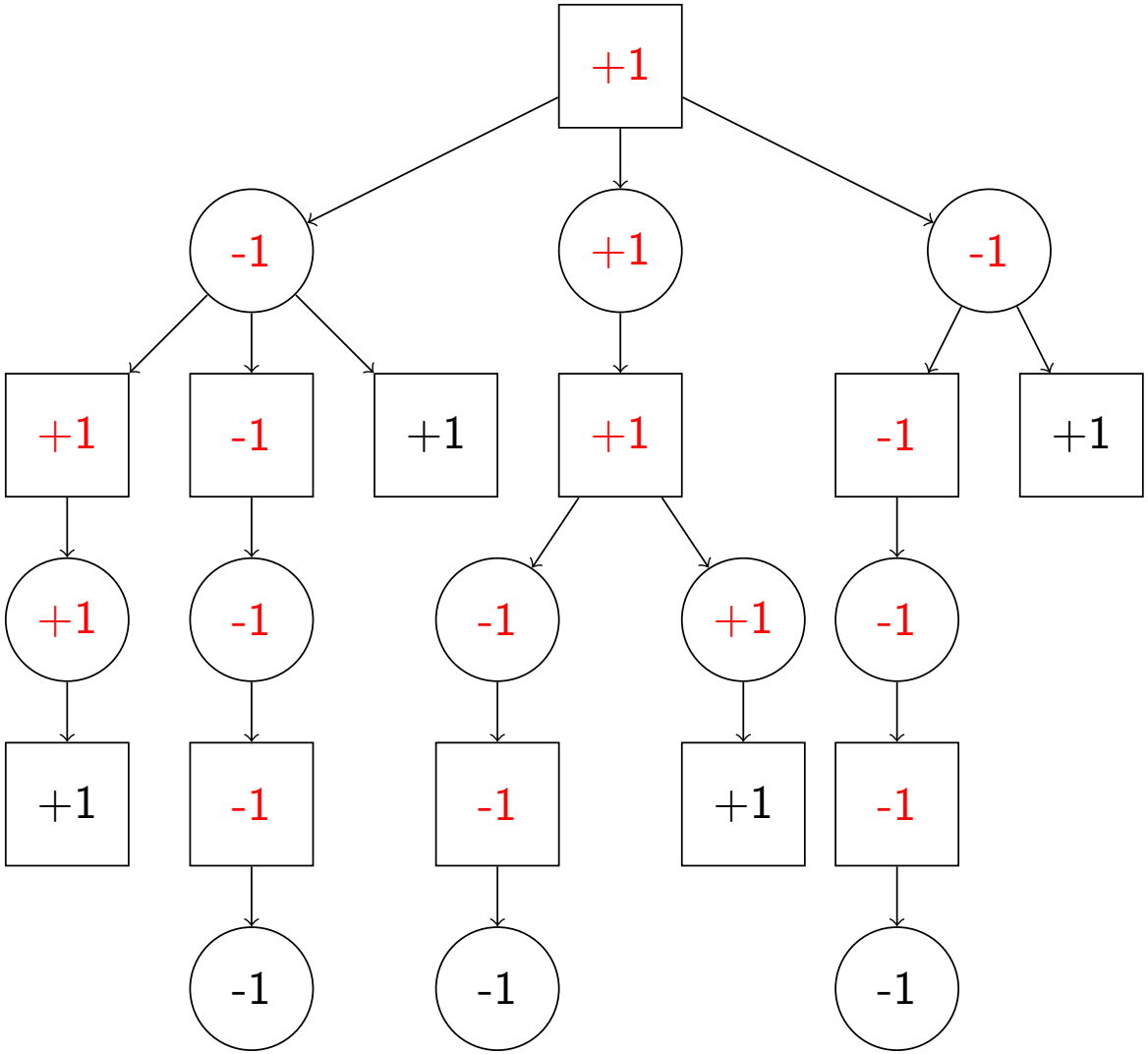
Adversarial search



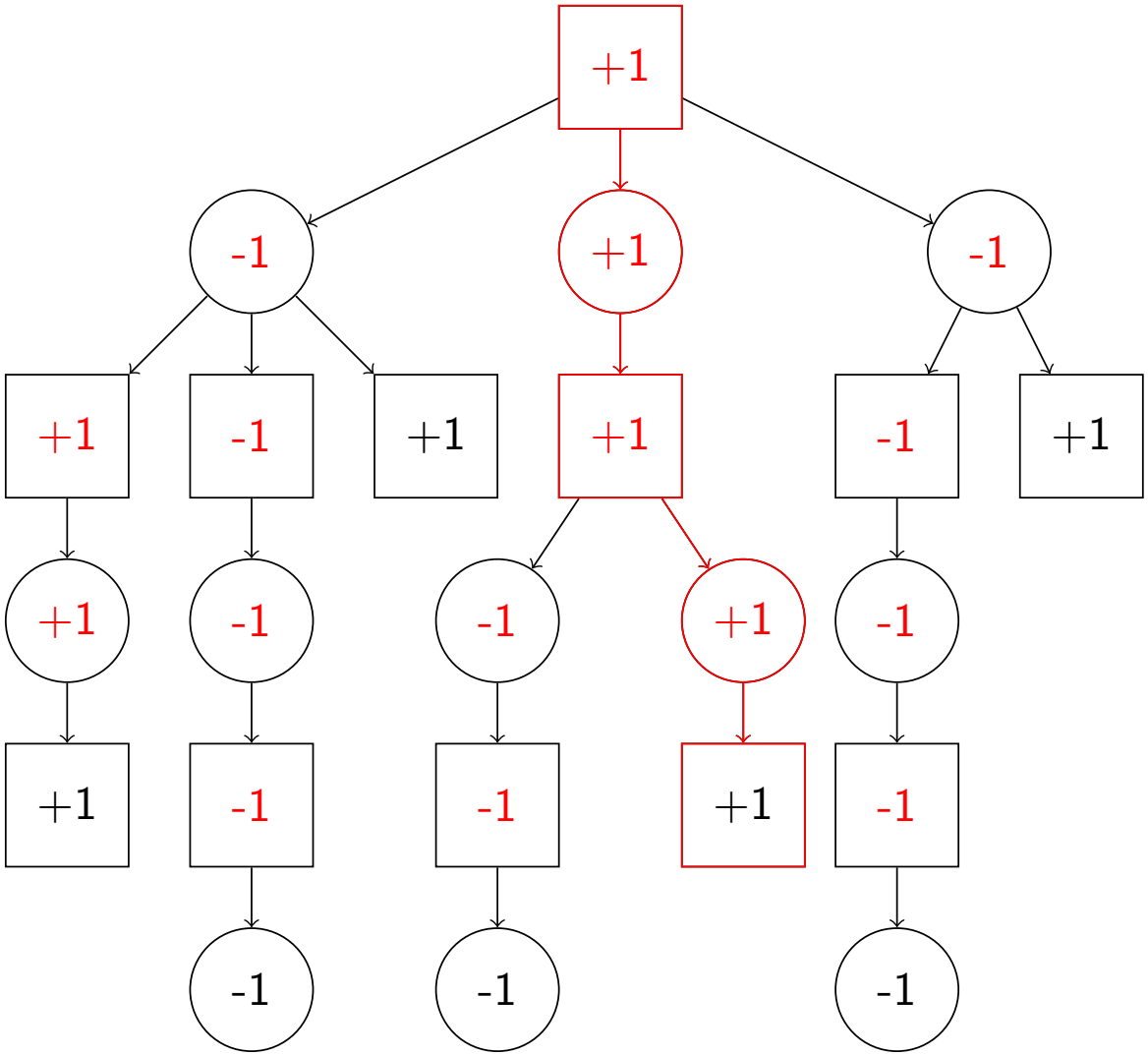
Adversarial search



Adversarial search



Adversarial search



Adversarial search - Heuristic Approach

- An heuristic function evaluates using domain knowledge how far is a state from being a winning state
- The value of this function is neither a cost nor a path length
- By convention the winning states are evaluated to $+\infty$ and the losing states to $-\infty$
- A depth-limited search algorithm is used to decide the current movement
- Each new decision needs to perform another search
- The deeper the algorithm searches the better the algorithm plays

Minimax algorithm

Function: MiniMax (g)

```
movr:move; max,maxc:integer
max ← −∞
foreach mov ∈ possible_moves(g) do
    cmax ← min_value(apply(mov,g))
    if cmax > max then
        max ← cmax
        movr ← mov
    end
end
return movr
```

g: Representation of the state (position of the pieces, maximum depth to explore, current player, ...)
possibles_moves(g): generates a list of all possible moves from the current state
apply(mov,g): Generates the state resulting from playing the move in the current state

- A depth limited depth-first search of the tree of moves is explored
- Depending on the level a function that returns the maximum or minimum of the evaluation of the descendants is applied (valueMax, valueMin)
- The search always starts with the MAX player
- We assume that the evaluation function is the same for both players

Minimax Algorithm

Function: valueMax (g)

```
vmax:integer
if final_state(g) then
    return value(g)
else
    vmax ← −∞
    foreach mov ∈ possible_moves(g) do
        vmax ← max(vmax,
        valueMin(apply(mov,g))
    end
    return vmax
end
```

Function: valueMin (g)

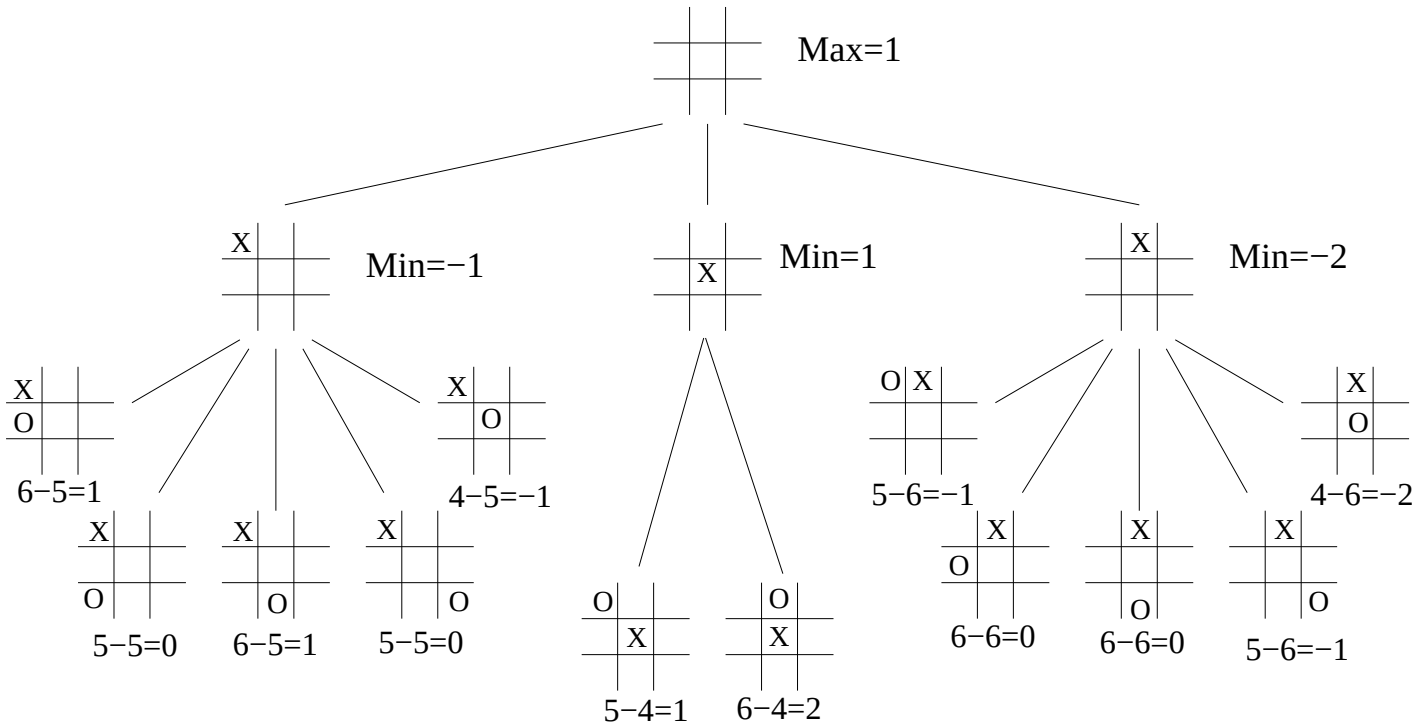
```
vmin:integer
if final_state(g) then
    return value(g)
else
    vmin ← +∞
    foreach mov ∈ possible_moves(g) do
        vmin ← min(vmin,
        valueMax(apply(mov,g))
    end
    return vmin
end
```

final_state: Determines if the current state is final (maximum depth, winning state)
value(g): Returns the computed value of the heuristic function for the current state

Example: Tic-tac-toe (1)

e = number of complete rows, columns and diagonals available to MAX - number of complete rows, columns and diagonals available to MIN
MAX plays **X** and needs to maximize *e*, MIN plays **O** and needs to minimize *e*

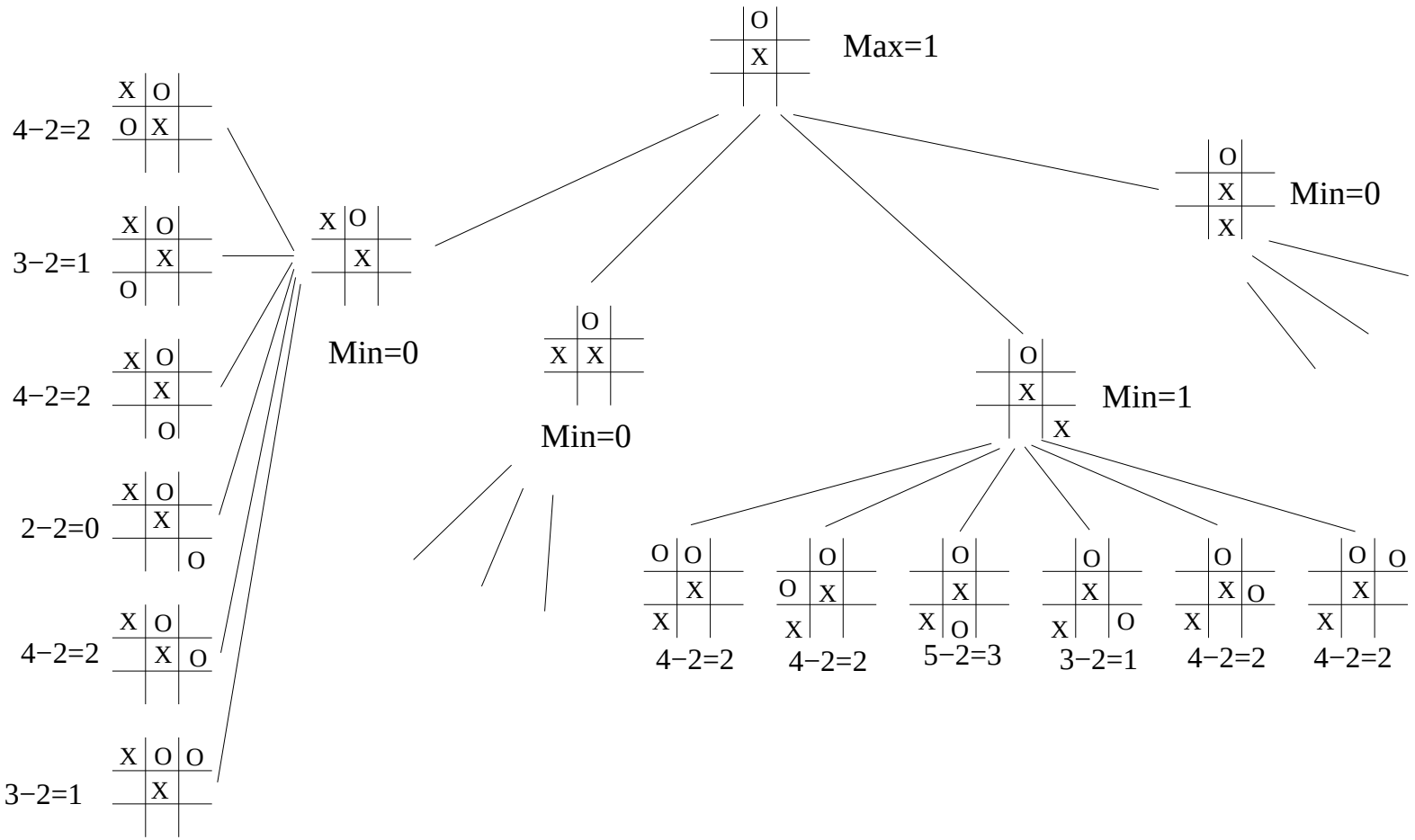
Higher absolute values mean better states for the current player, Symmetries are take in account, a maximum depth is given (in the example is 2)



The best first play for max is to cover the central position

Example: Tic-tac-toe (2)

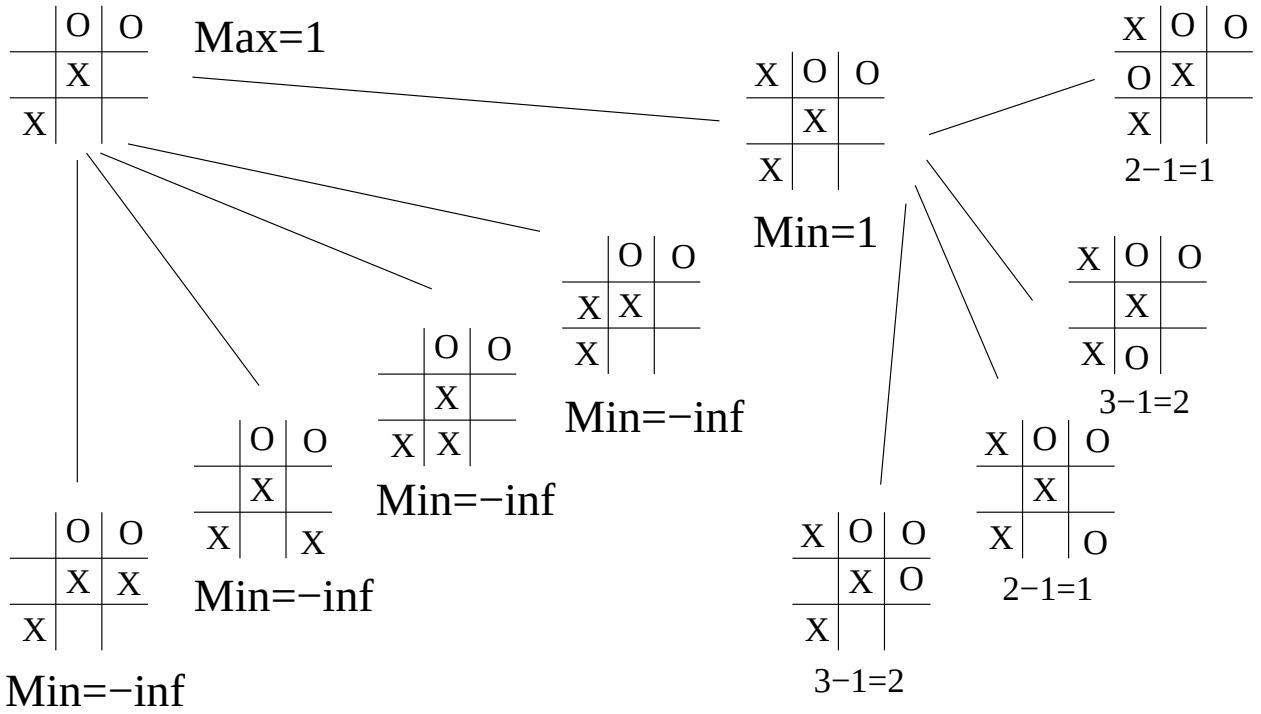
Assuming that the move of MIN is to cover the first position of the central column



The best move for MAX is to cover the inferior left corner

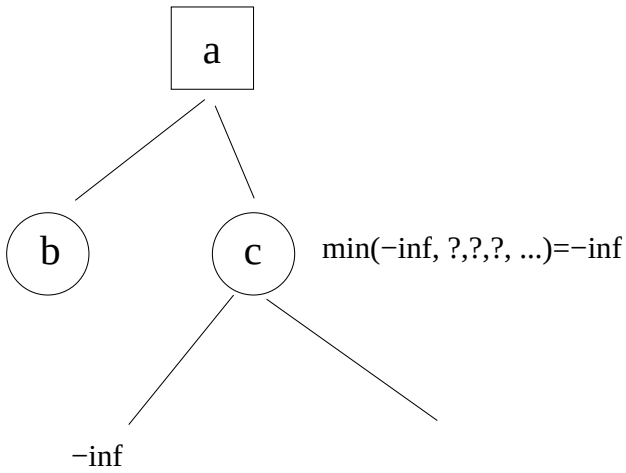
Example: Tic-tac-toe (3)

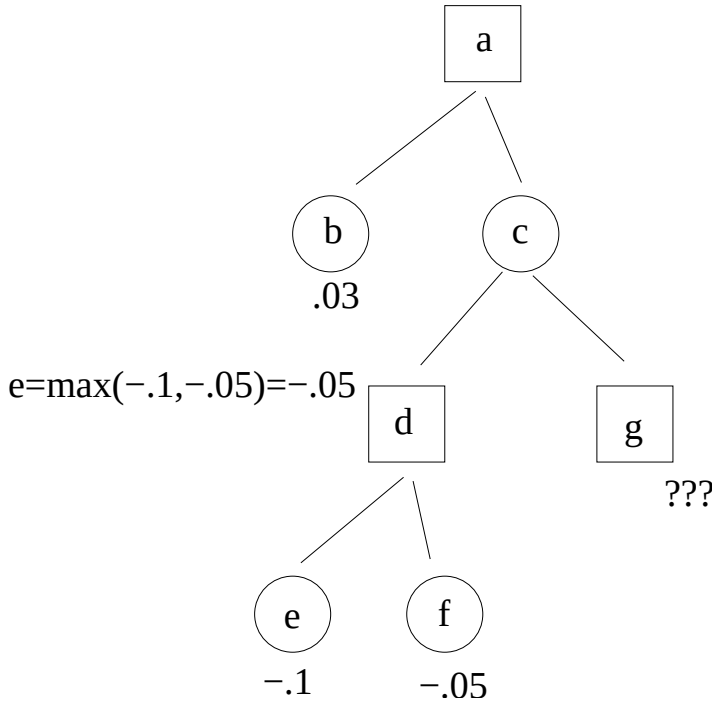
Assuming that the move of MIN is to cover the superior right corner



The best move for MAX is to cover the superior left corner

Minimax with $\alpha\beta$ pruning

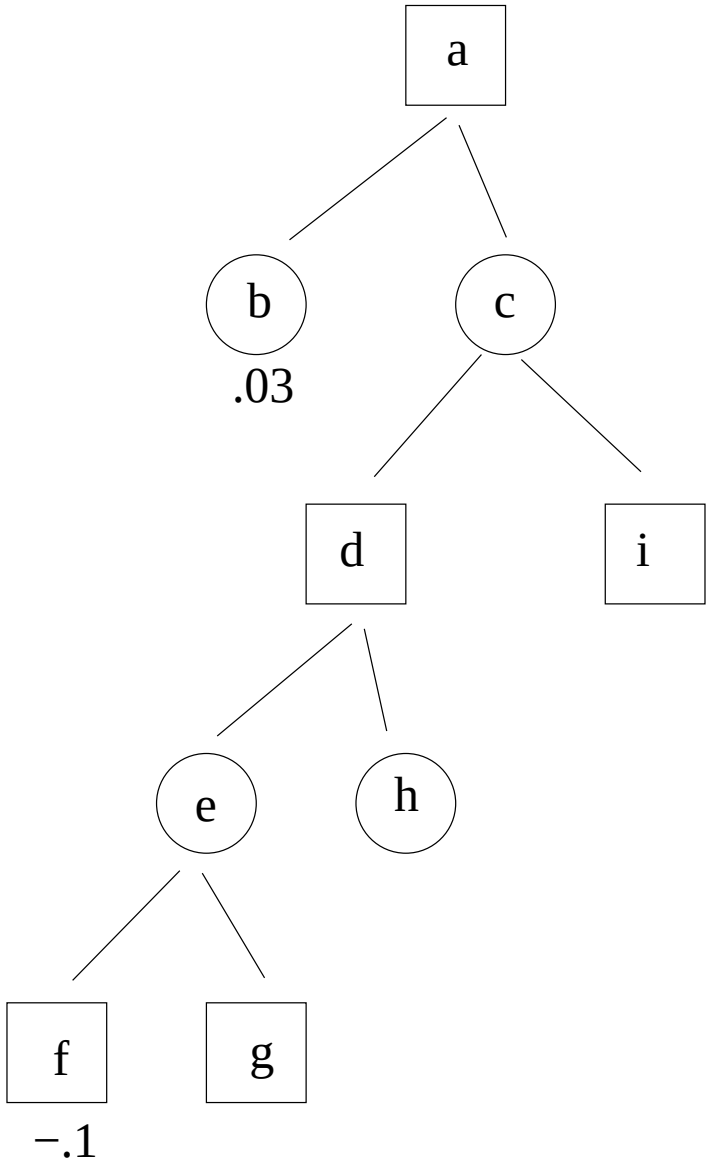




It is not necessary to explore the successors of **c** because we already have the best possible evaluation

In **c** we have $e = min(-.05, v_g)$, thus in **a** we have $e = max(.03, min(-.05, v_g)) = .03$
All descendants of **g** are pruned because they will not change the decision

$\alpha\beta$ pruning Minimax



e(e) = min(−.1,g)

Because **b** has an evaluation of .03 any value less than that is of no use ⇒ **g** does not need to be explored

e(d) = max(e(e), h) ⇒ **h** needs to be explored

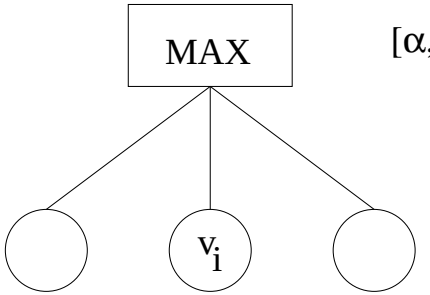
The minimum value obtained exploring the max states is the α limit and gives the inferior bound of e(n).

The maximum value obtained exploring the min states is the β limit and gives the superior bound of e(n)

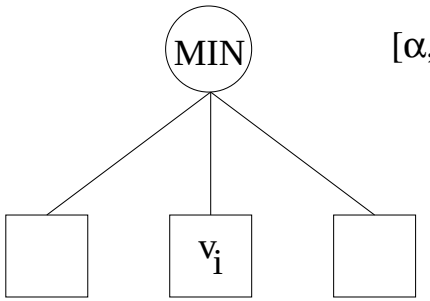
In the example, the node **a** (max) has currently a minimum value of .03 obtained from its descendants **b**

Notice that the value of a max node comes from the evaluation of its min nodes (and viceversa)

Minimax with $\alpha\beta$ pruning



if $v_i > \alpha$ then update α
if $v_i \geq \beta$ then β pruning
return α



if $v_i < \beta$ then update β
if $vi \leq \alpha$ then α pruning
return β

The α and β limits are passed to the siblings in order of expansion
α is the lower bound for the max nodes
β is the upper bound for the min nodes
Pruning depends on the order of exploration

$\alpha\beta$ pruning Minimax algorithm

Function: valueMax (g,α,β)

```
if final_state(g) then
    return value(g)
else
    foreach mov ∈ possible_moves(g) do
        α ← max(α,valueMin(apply(mov,g) ,α,β))
        if α ≥ β then
            return β
        end
    end
    return α
end
```

Function: valueMin (g,α,β)

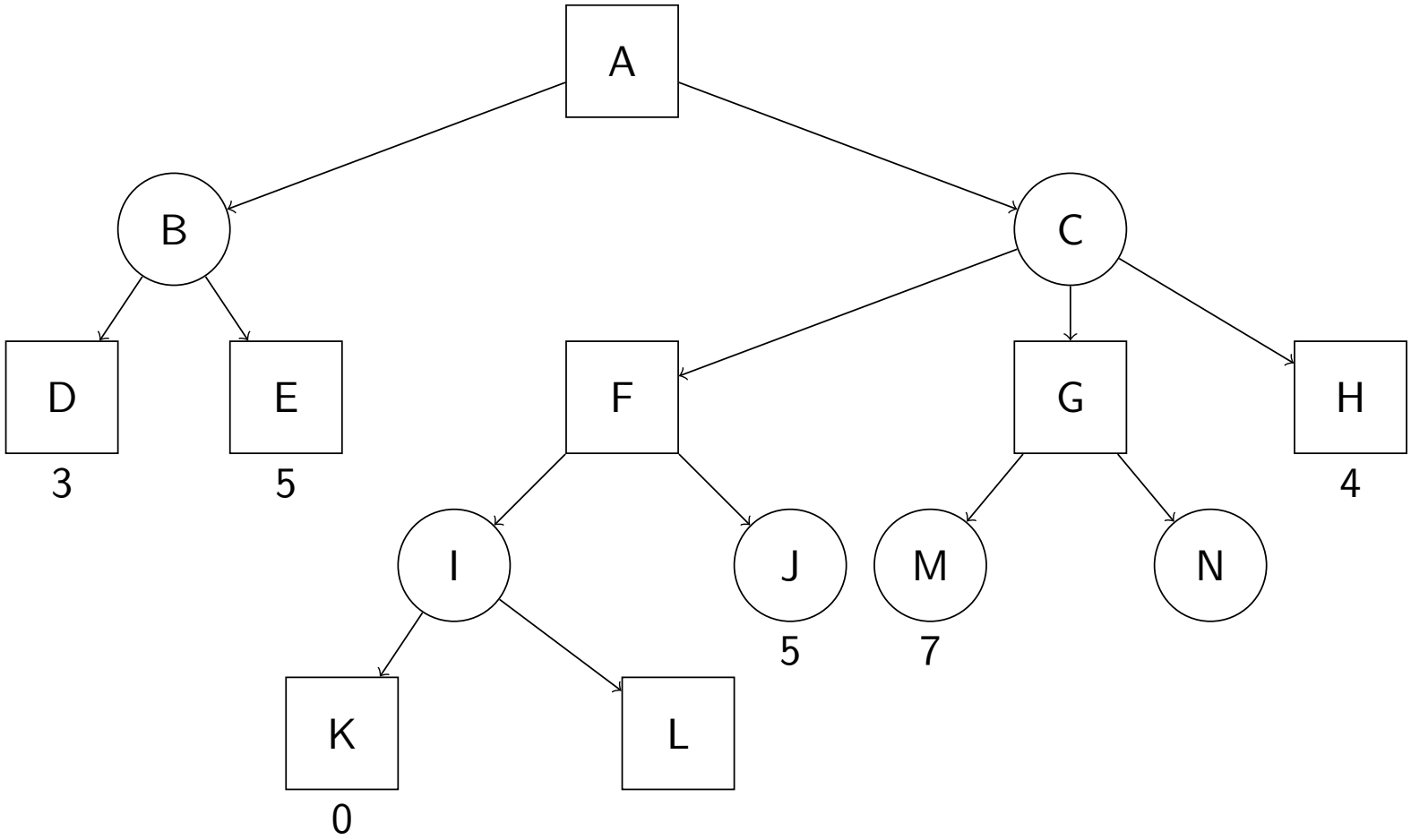
```
if final_state(g) then
    return value(g)
else
    foreach mov ∈ possible_moves(g) do
        β ← min(β,valueMax(apply(mov,g) ,α,β))
        if α ≥ β then
            return α
        end
    end
    return β
end
```

The search is initiated calling the function valueMax with $\alpha = -\infty$ $\beta = +\infty$

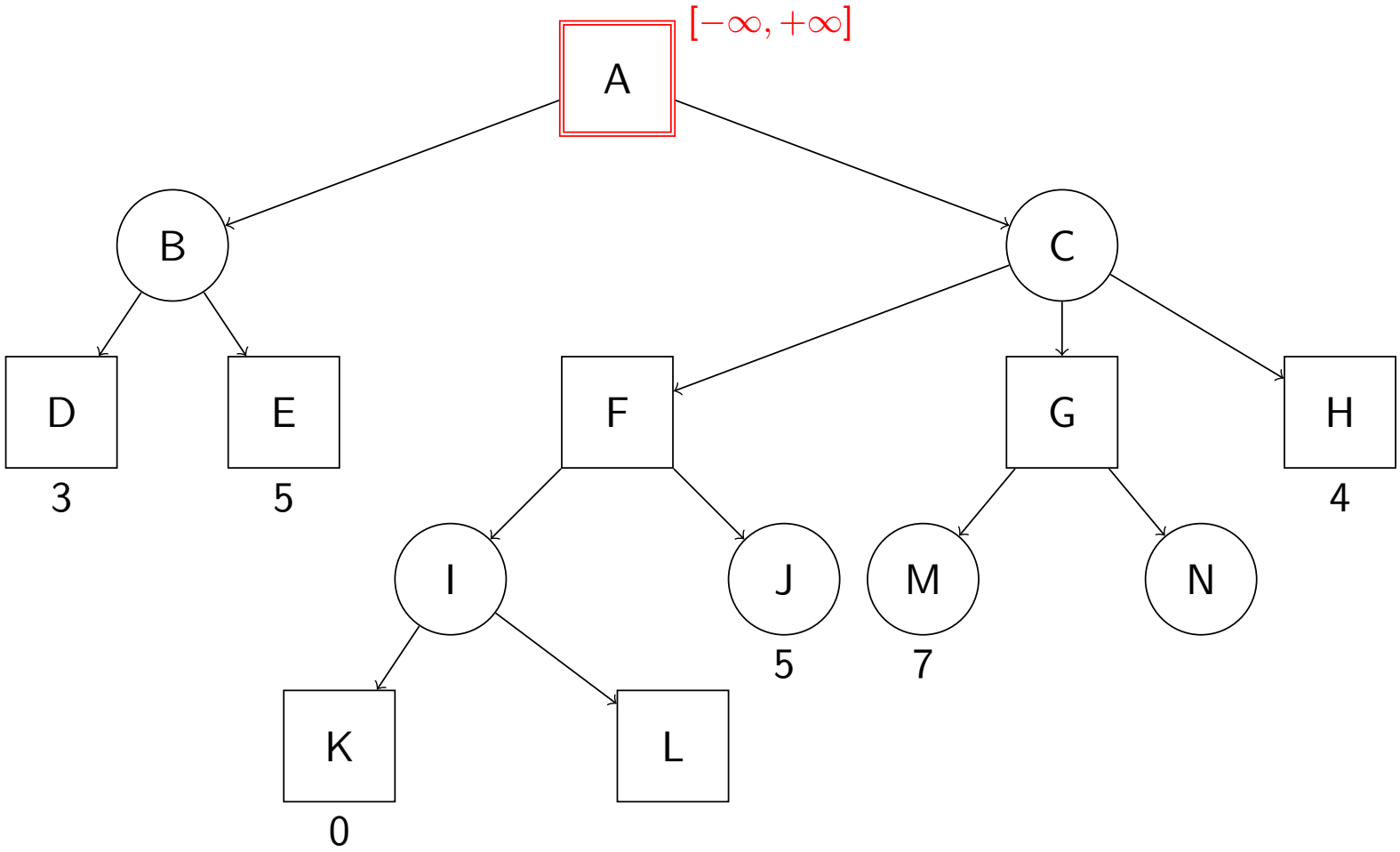
In the function valueMax α is the parameter that is updated and β is the value of the best movement

In the function valueMin β is the parameter that is updated and α is the value of the best movement

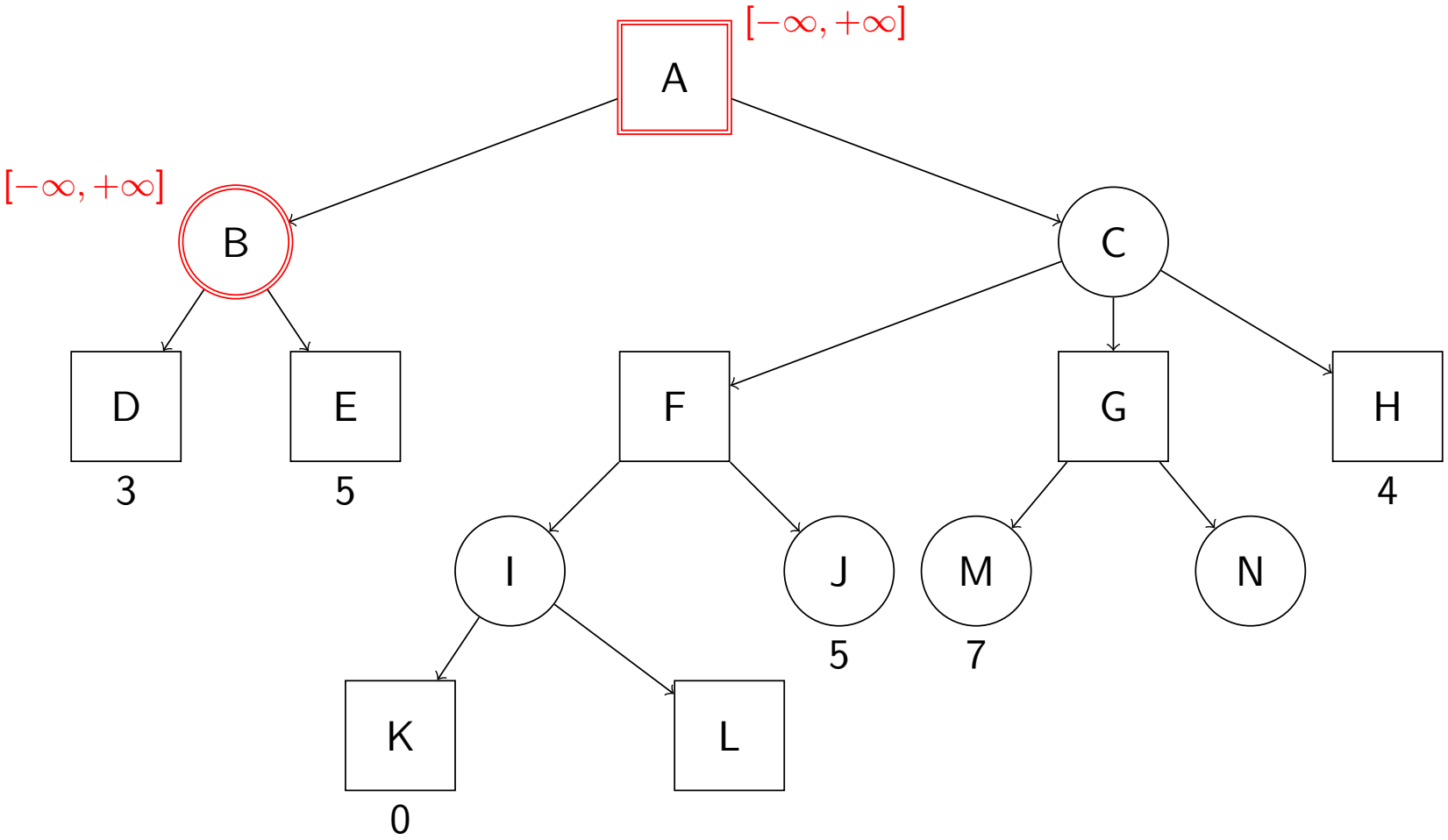
$\alpha\beta$ pruning: Example



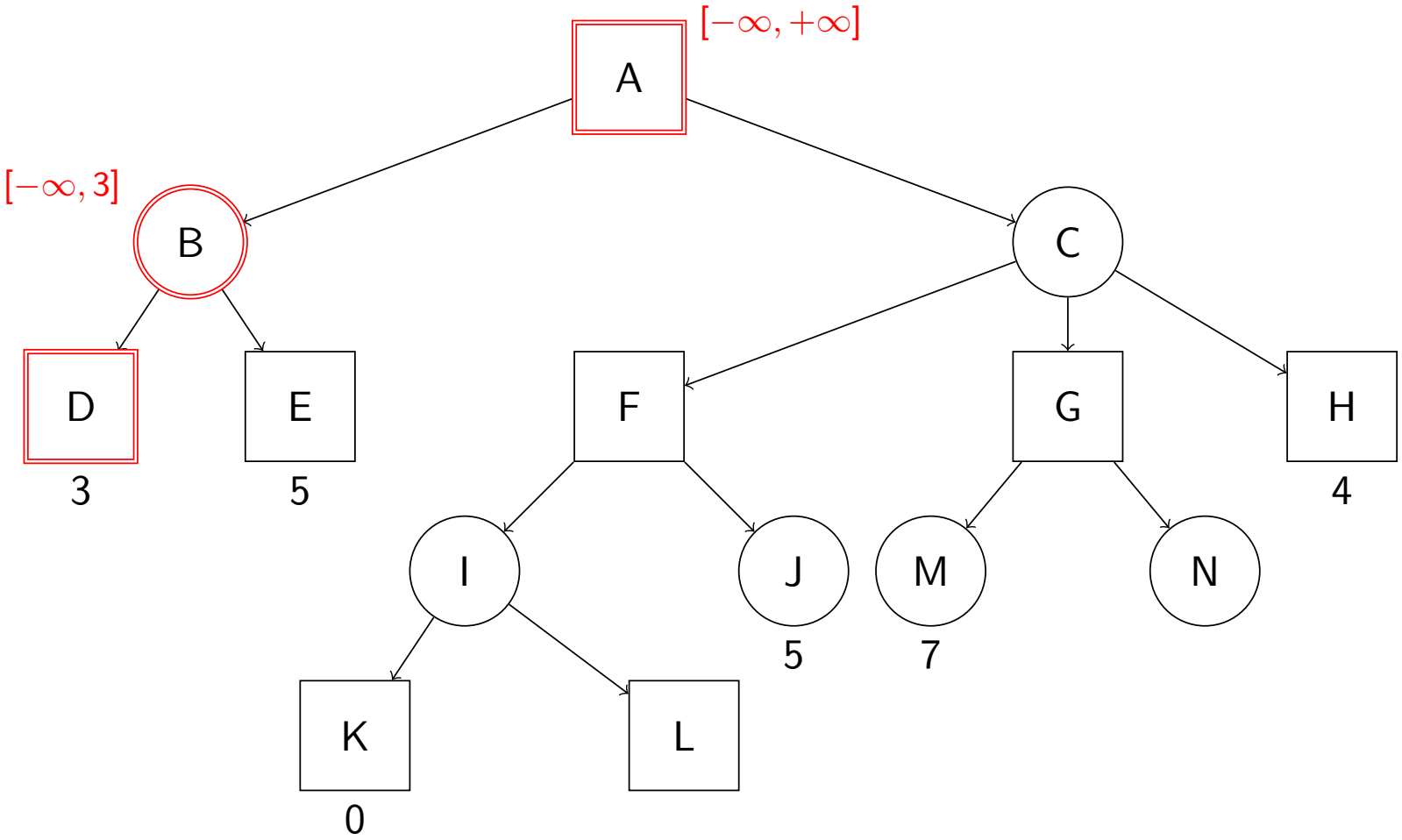
$\alpha\beta$ pruning: Example



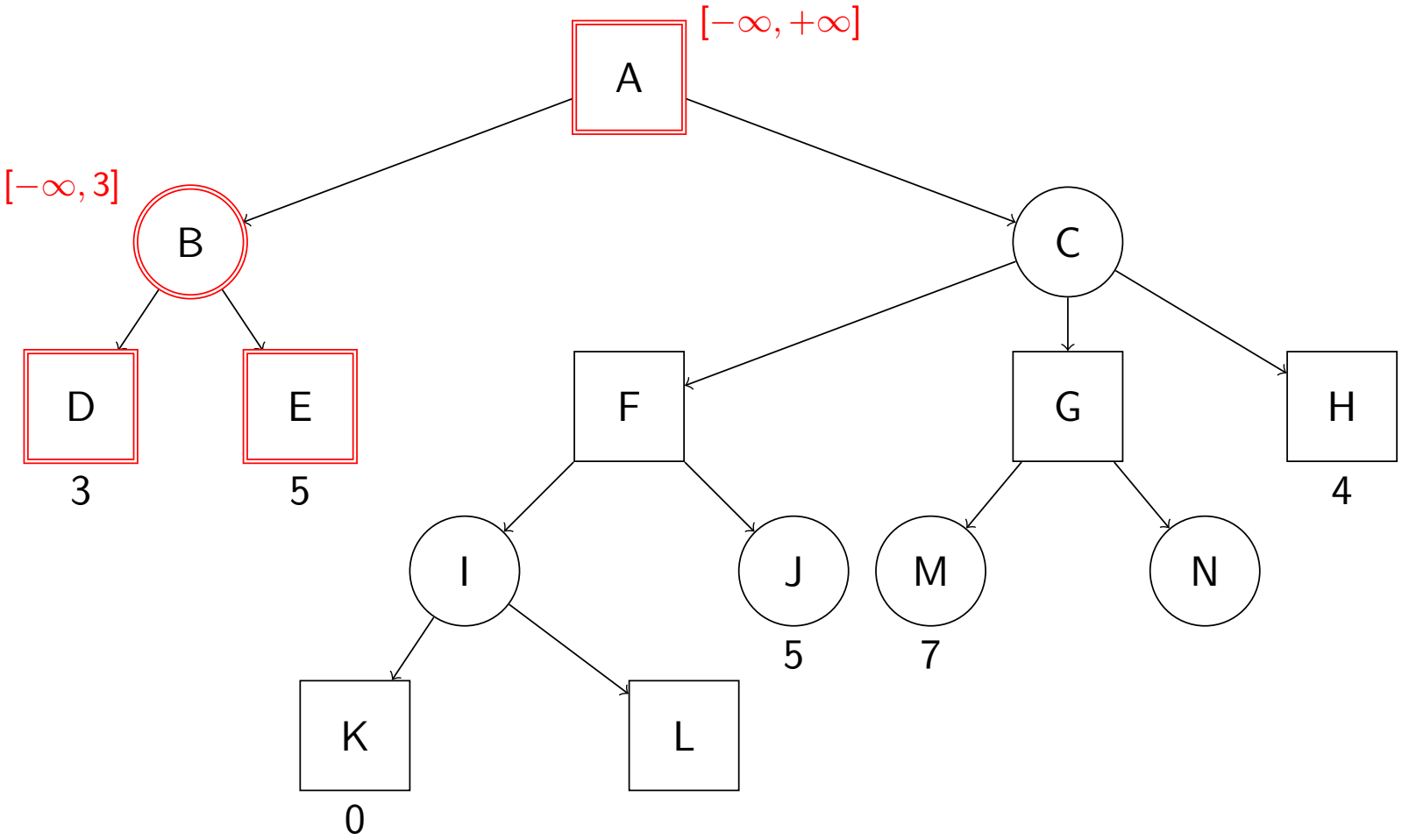
$\alpha\beta$ pruning: Example



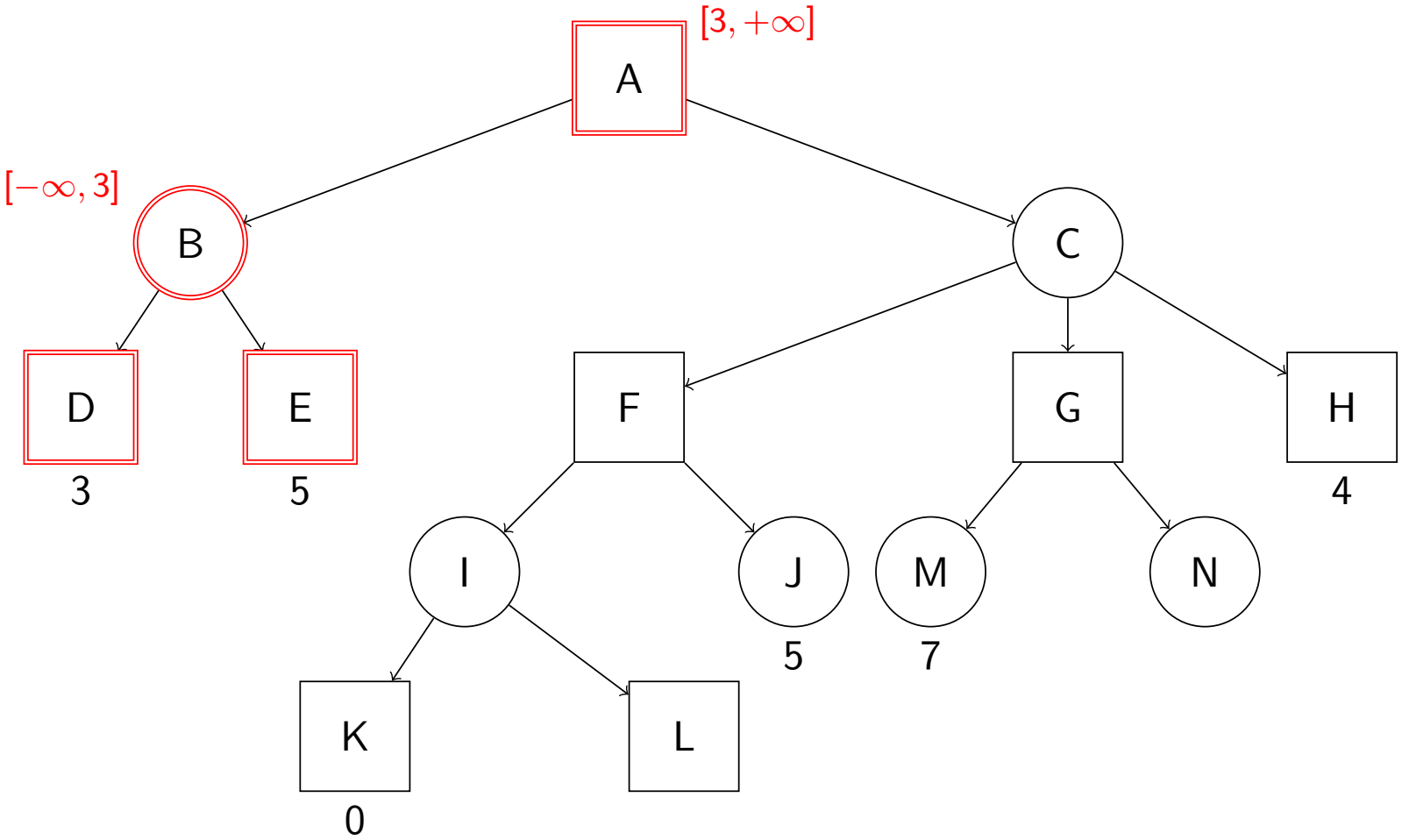
$\alpha\beta$ pruning: Example



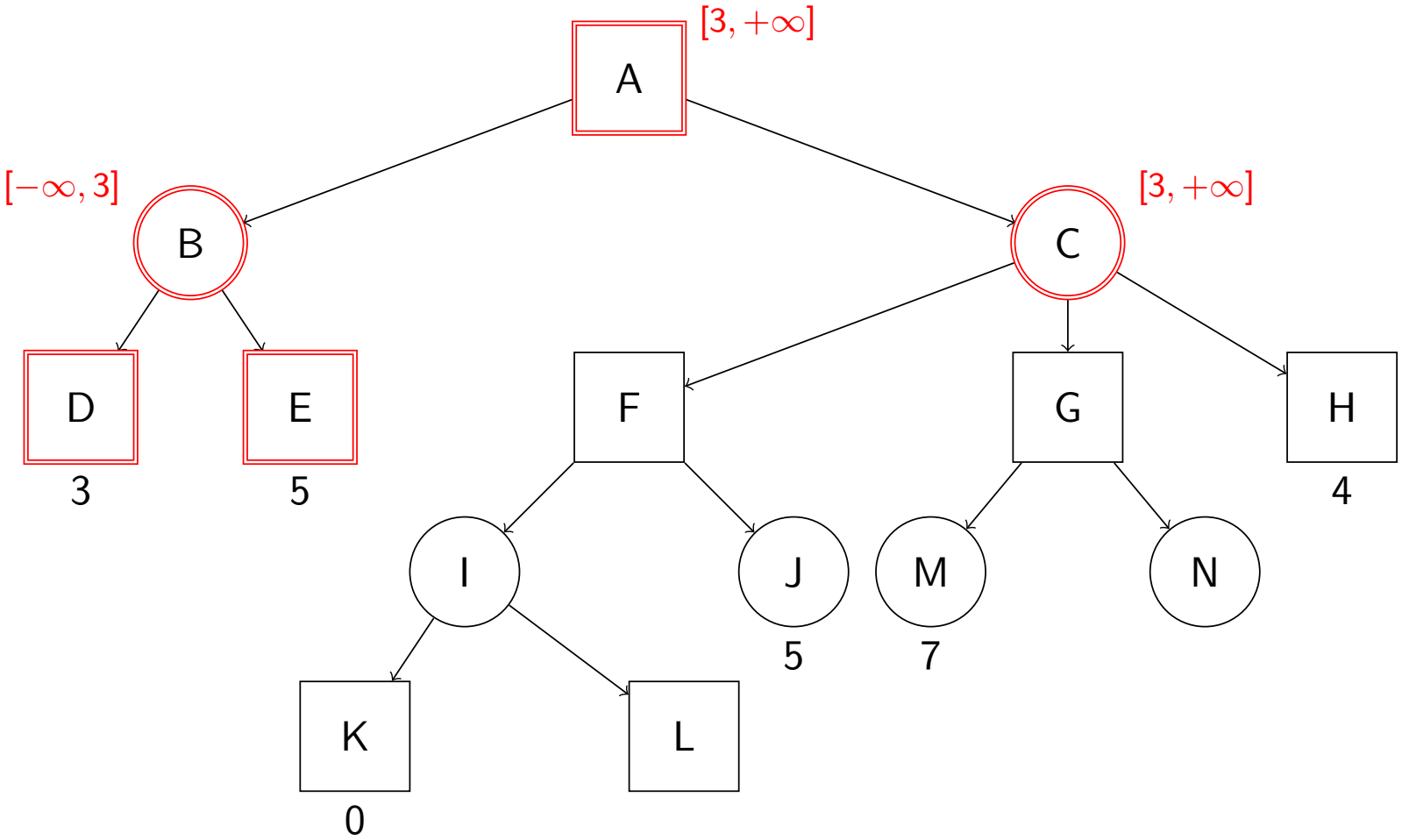
$\alpha\beta$ pruning: Example



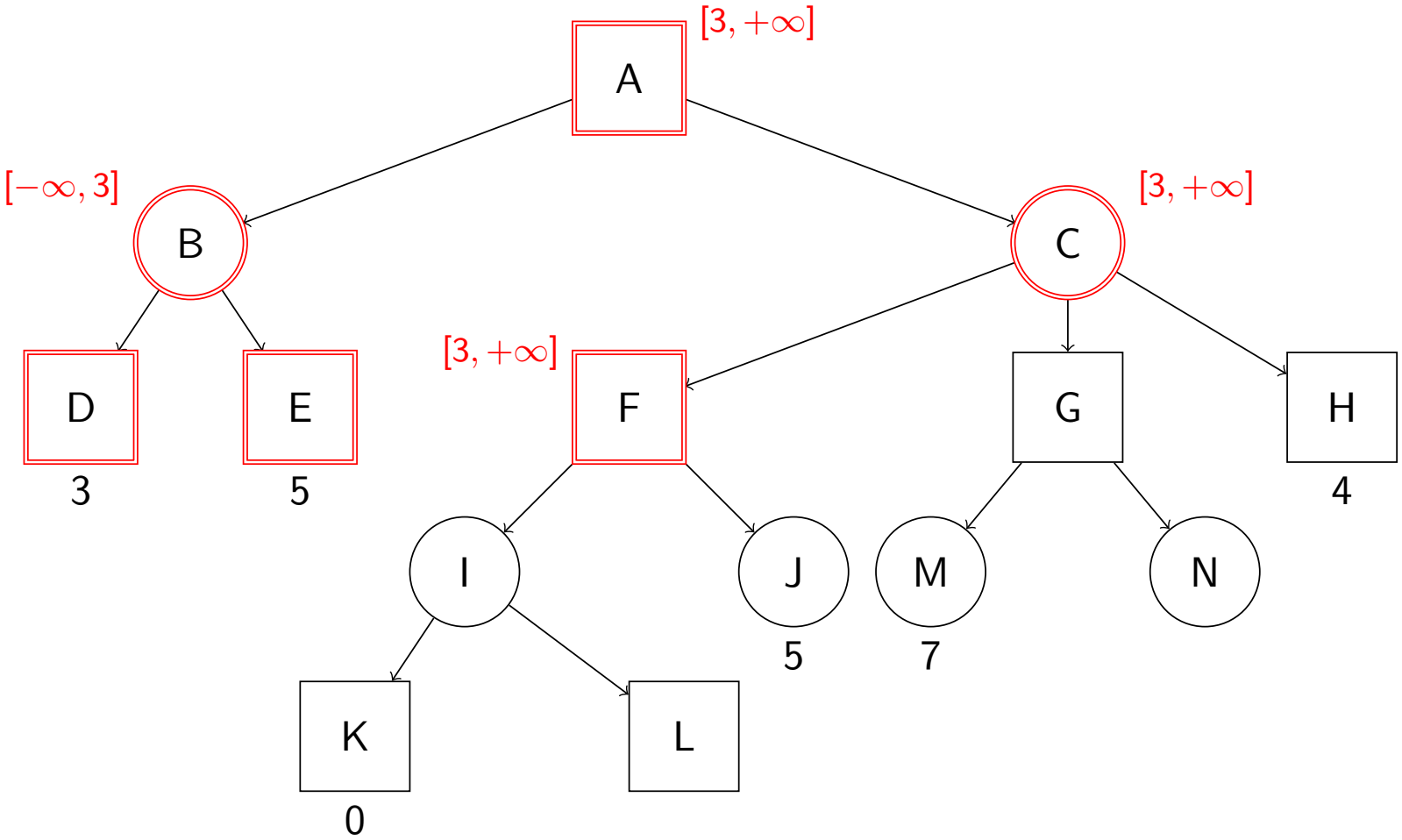
$\alpha\beta$ pruning: Example



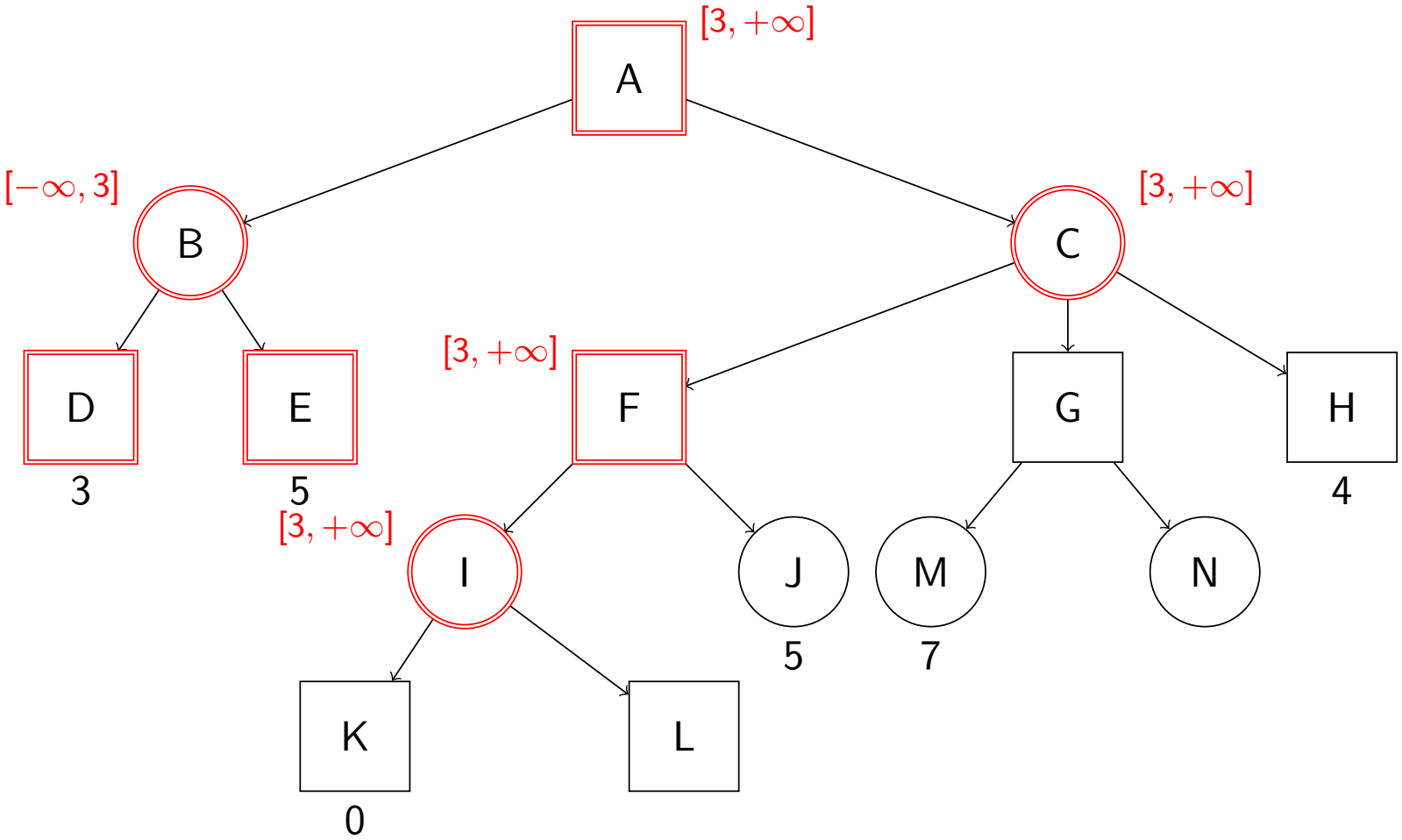
$\alpha\beta$ pruning: Example



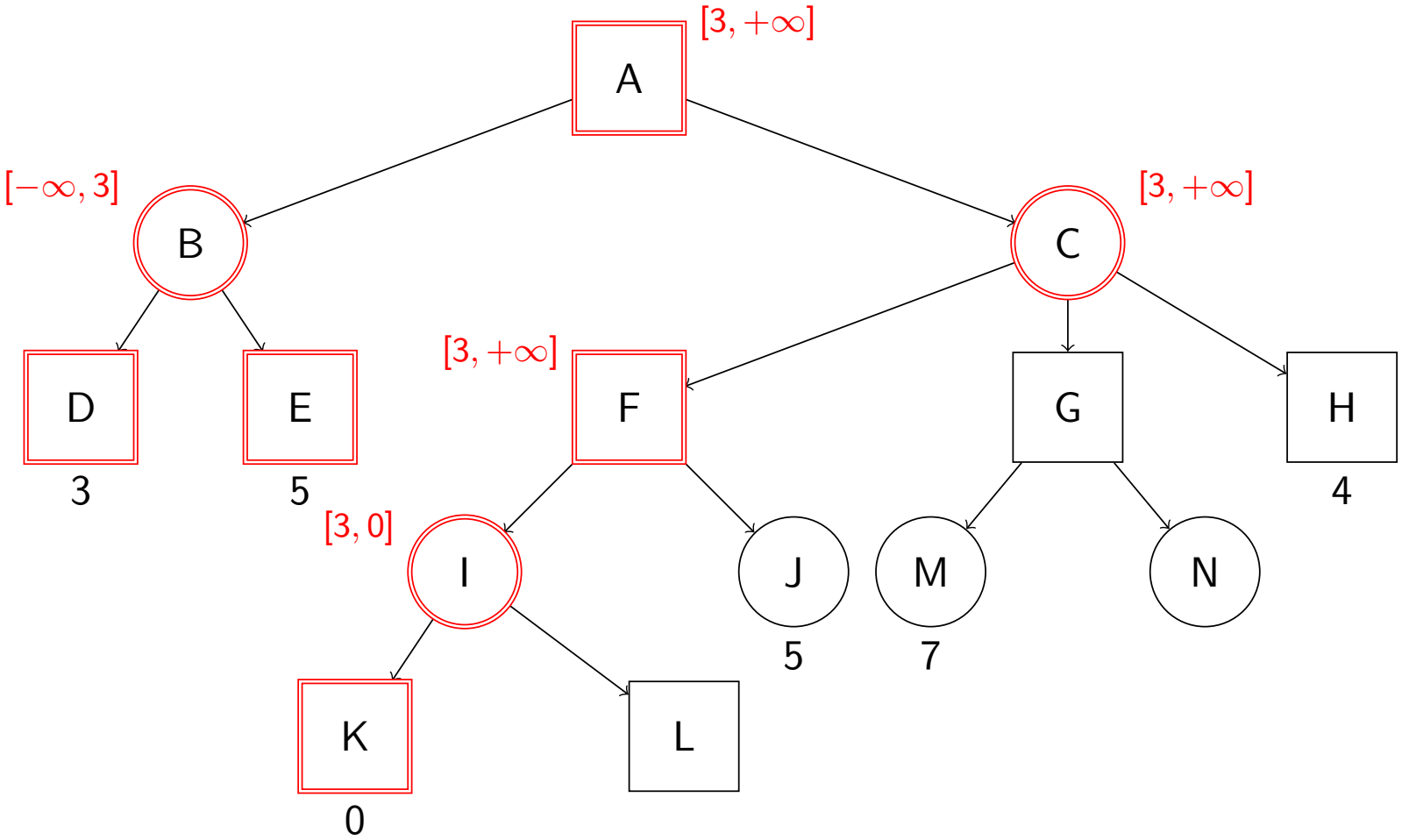
$\alpha\beta$ pruning: Example



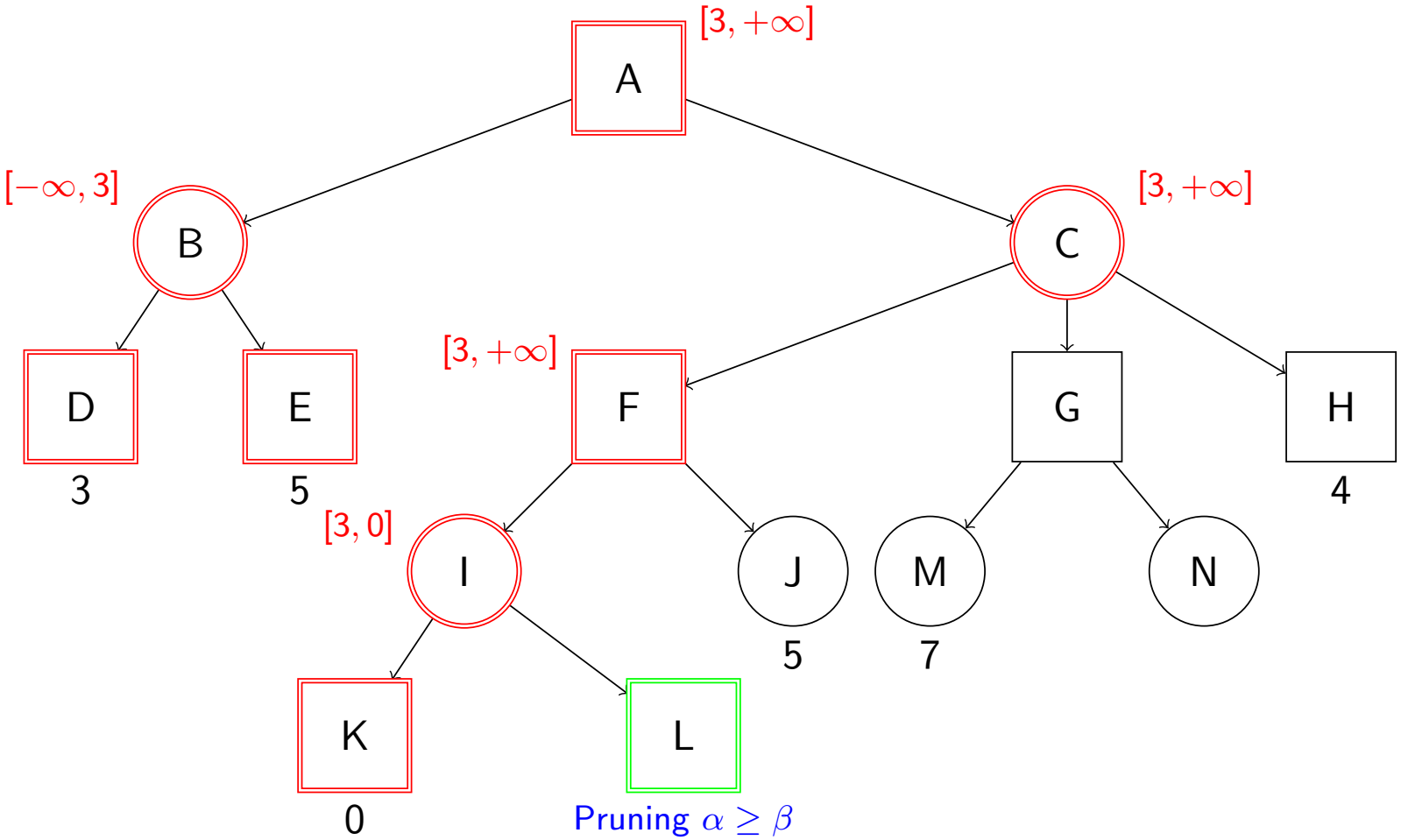
$\alpha\beta$ pruning: Example



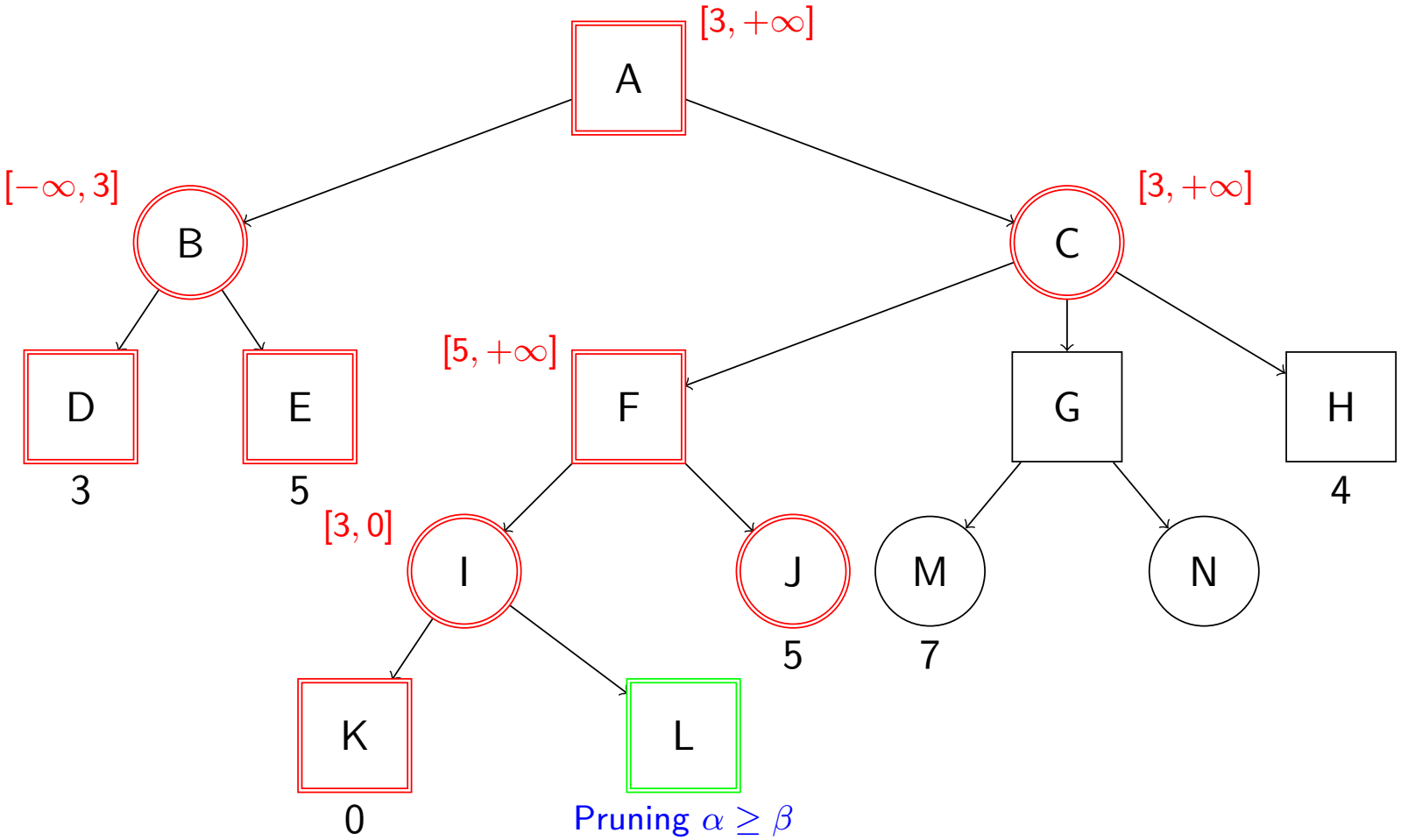
$\alpha\beta$ pruning: Example



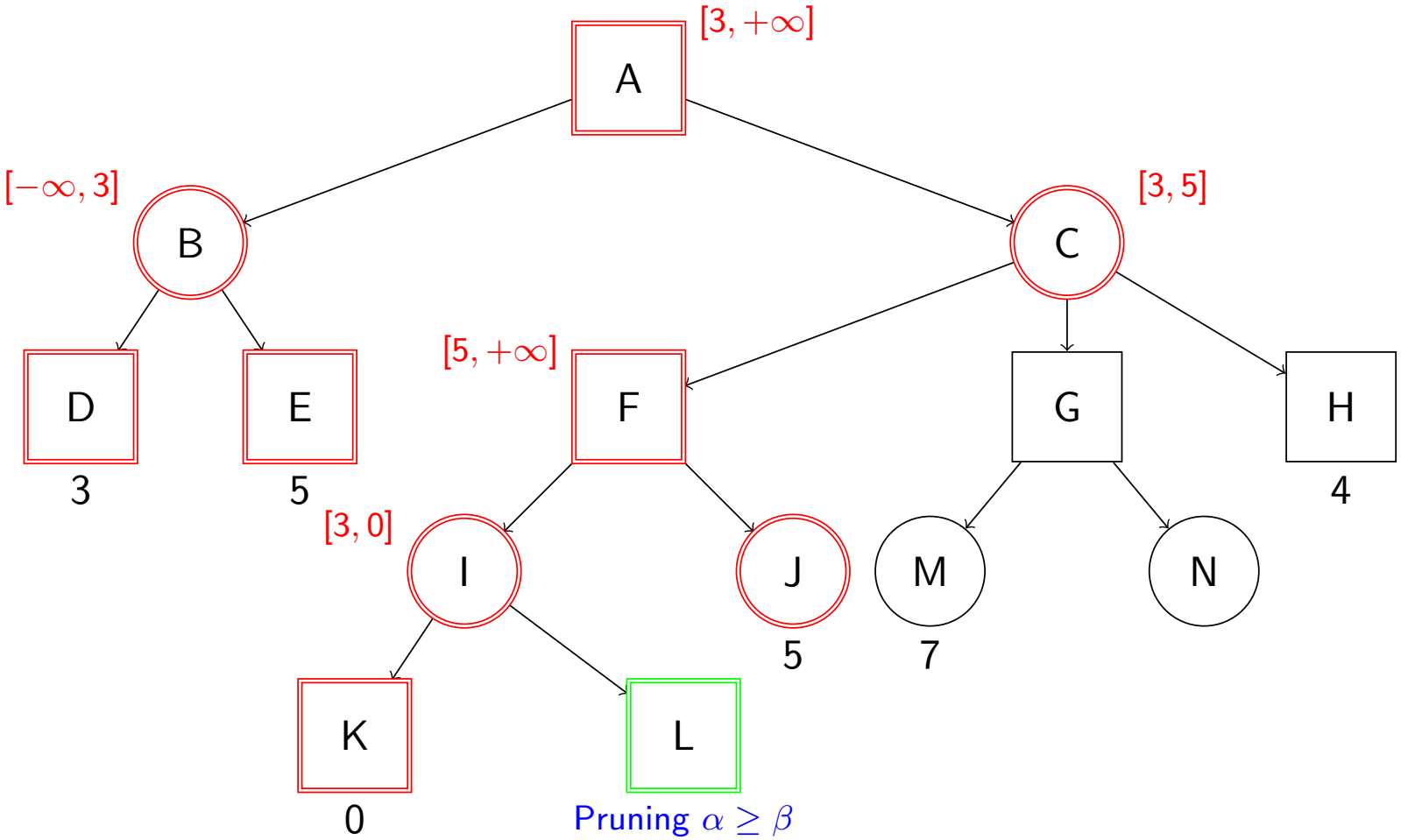
$\alpha\beta$ pruning: Example



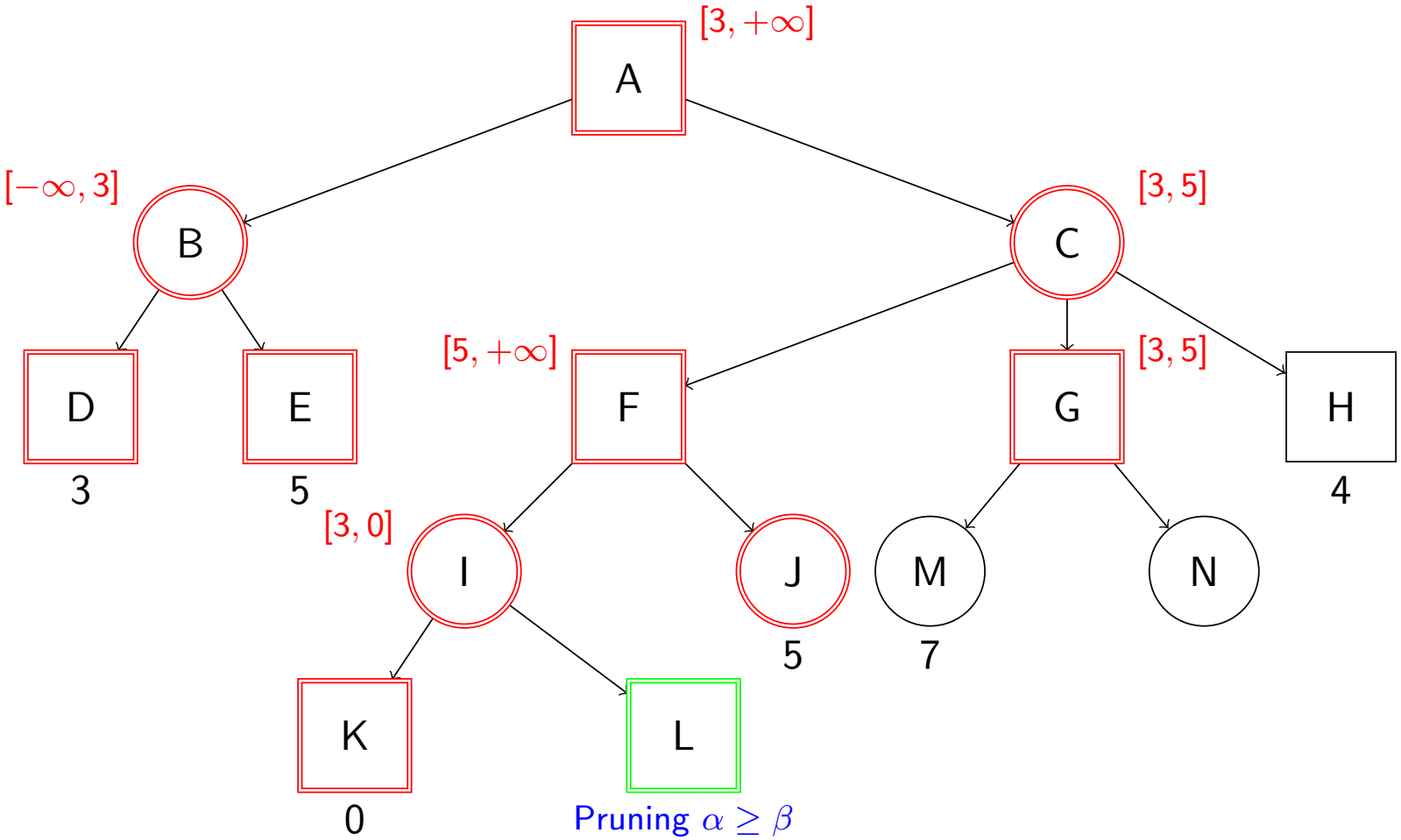
$\alpha\beta$ pruning: Example



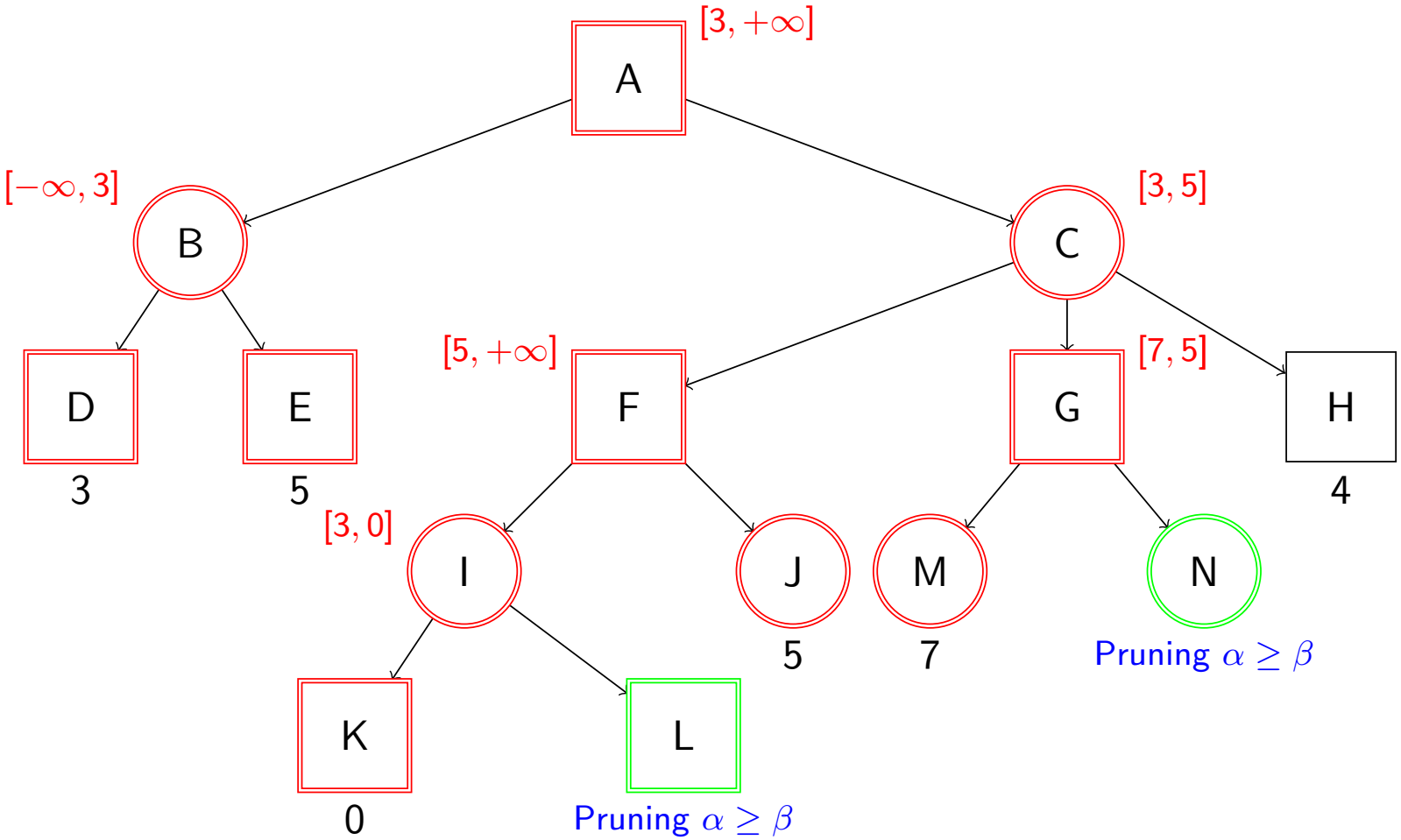
$\alpha\beta$ pruning: Example



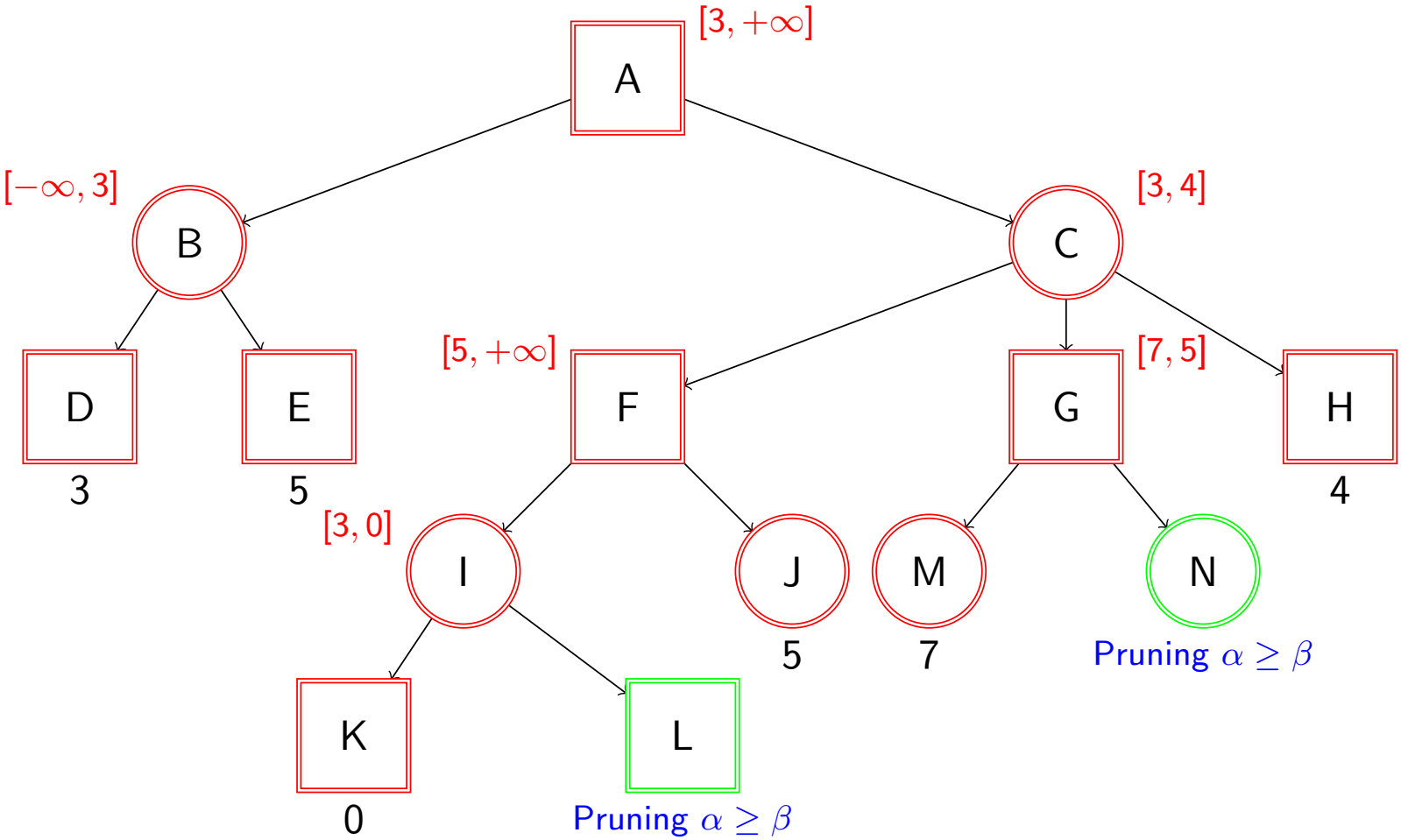
$\alpha\beta$ pruning: Example



$\alpha\beta$ pruning: Example



$\alpha\beta$ pruning: Example



$\alpha\beta$ pruning: Example