Game Playing

What kind of games?

- **Abstraction: To describe a game we must capture every relevant aspect of the game. Such as:**
  - Chess
  - Tic-tac-toe
  - …
- **Accessible environments: Such games are characterized by perfect information**
- **Search: game-playing then consists of a search through possible game positions**
- **Unpredictable opponent: introduces uncertainty thus game-playing must deal with contingency problems**

Improving effectiveness

- **To improve the effectiveness of a search-based problem-solving program two things can be done:**
  - Improve the generate procedure so that only good moves (or paths) are generated
  - Improve the test procedure so that the best moves (or paths) will be recognized and explored first

Searching for the next move

- **Complexity: many games have a huge search space**
  - **Chess:** $b = 35, m=100 \Rightarrow nodes = 35^{100}$

- **Resource (e.g., time, memory) limit: optimal solution not feasible/possible, thus must approximate**

1. **Pruning: makes the search more efficient by discarding portions of the search tree that cannot improve quality result.**

2. **Evaluation functions: heuristics to evaluate utility of a state without exhaustive search.**

Static Evaluation Function

- **In the amount of time available, it is usually possible to search a tree only ten or twenty moves deep (called ply).**
- **Then in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous.**
- **This is done using a static evaluation function, which uses whatever information it has to evaluate individual board positions by estimating how much likely they are to lead eventually to a win**

Two-player games

- **A game formulated as a search problem:**

  - Initial state: ?
  - Operators: ?
  - Terminal state: ?
  - Utility function: ?
    (**payoff function**)

Two-player games

- **A game formulated as a search problem:**

  - Initial state: board position and turn
  - Operators: definition of legal moves
  - Terminal state: conditions for when game is over
  - Utility function: (**payoff function**) a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win.

Mini Max search Procedure

- **For a simple one person game or puzzle, the A* algorithm can be used.**
- **It can be applied to reason forward from the current state as far as possible in the time allotted.**
- **But because of their adversarial nature, this procedure is inadequate for 2-person games such as chess.**
- **As values are pushed back up different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses**

Mini Max

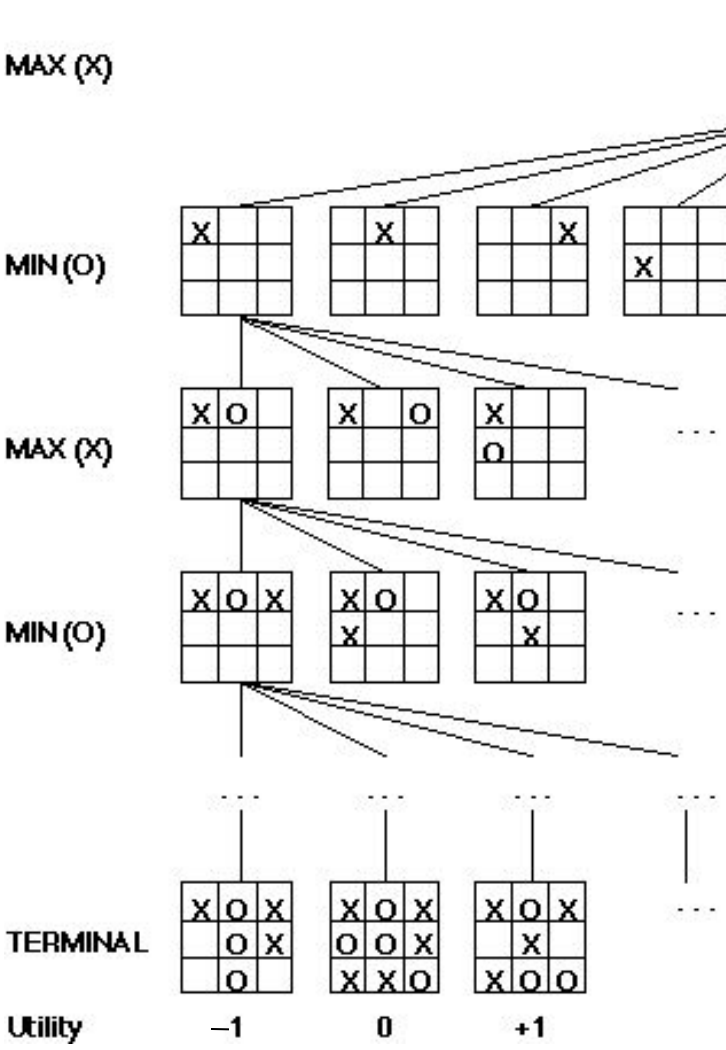
- **The mini max search procedure is a depth-first, depth-limited search procedure.**
- **The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions.**
- **Now we can apply the "static evaluation function" to those positions and simply choose the best one.**
- **After doing so, we can back up the value to the starting position to represent our evaluation of it.**

The minimax algorithm

- **Perfect play for deterministic environments with perfect information**
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the tree by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value

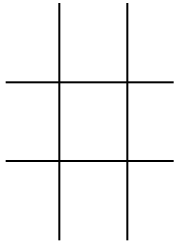
- **Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.**

Partial Game Tree for Tic-Tac-Toe

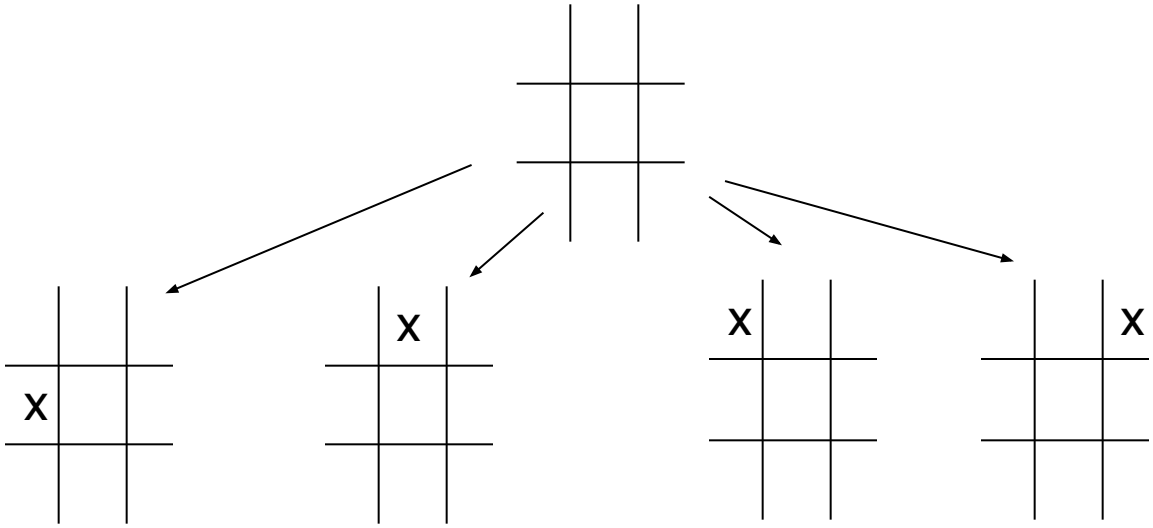


- f(n) = +1 if the position is a win for X.
- f(n) = -1 if the position is a win for O.
- f(n) = 0 if the position is a draw.

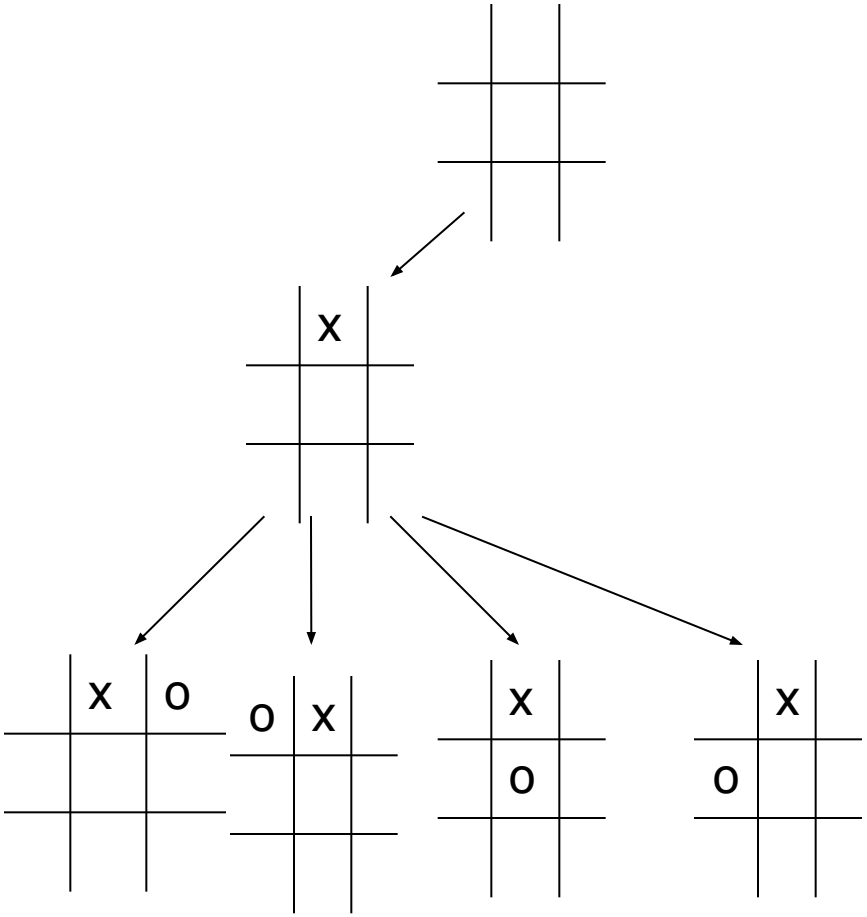
Generate Game Tree



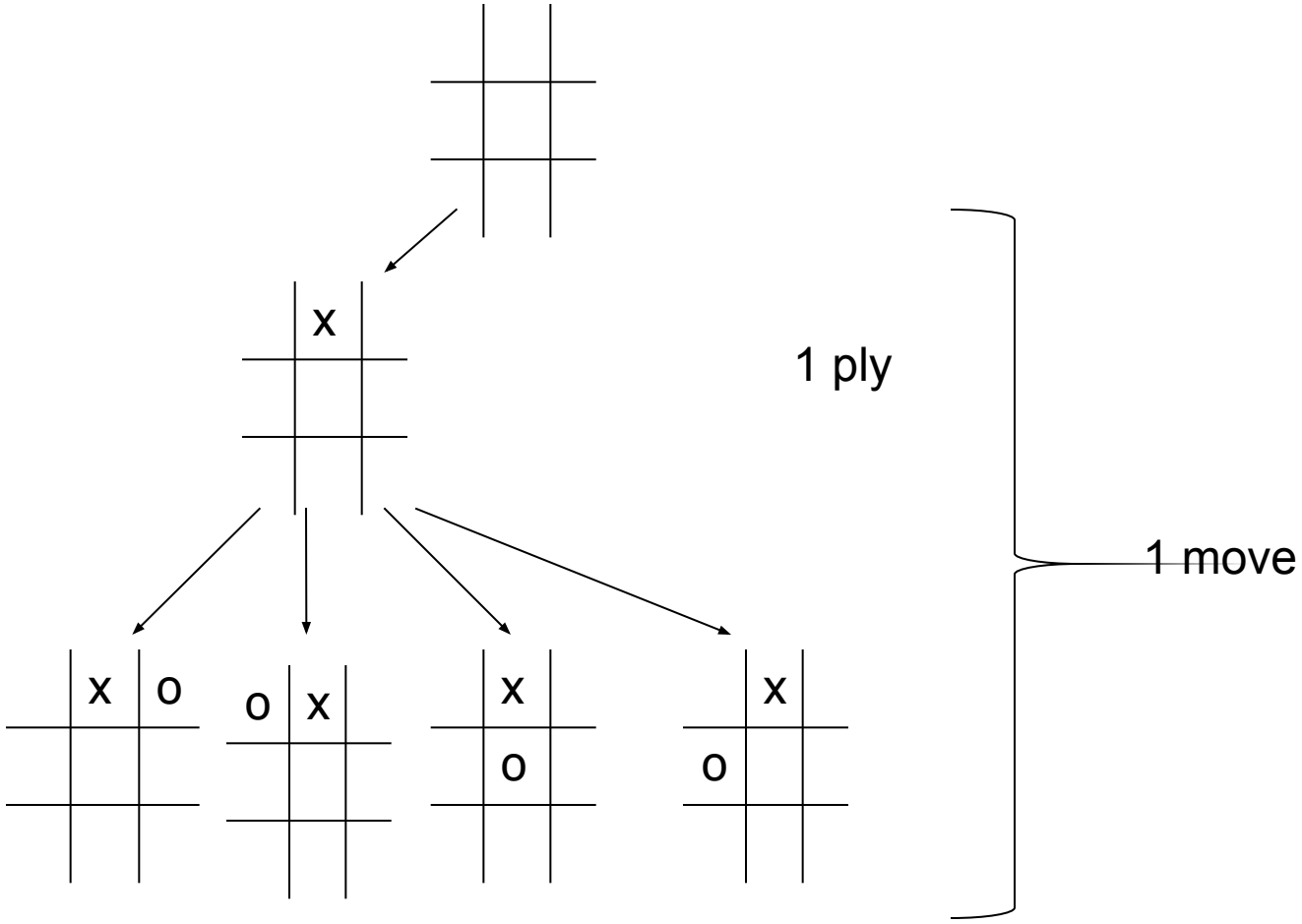
Generate Game Tree



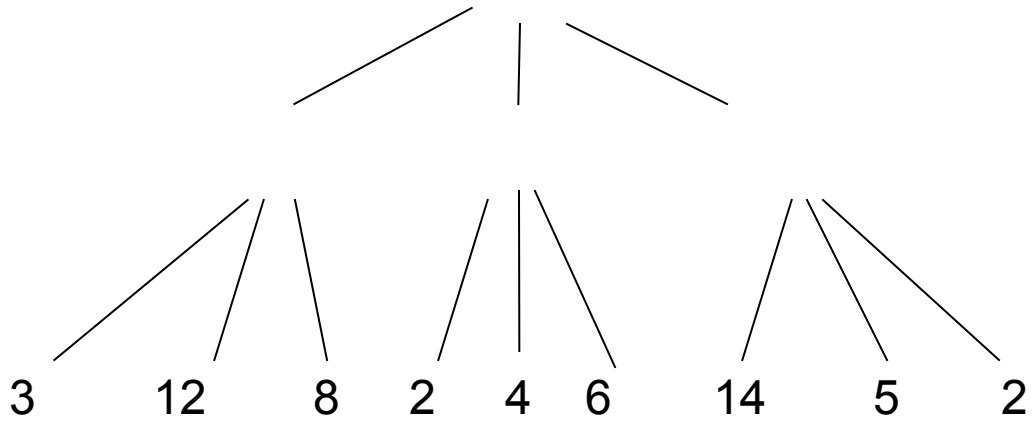
Generate Game Tree



Generate Game Tree

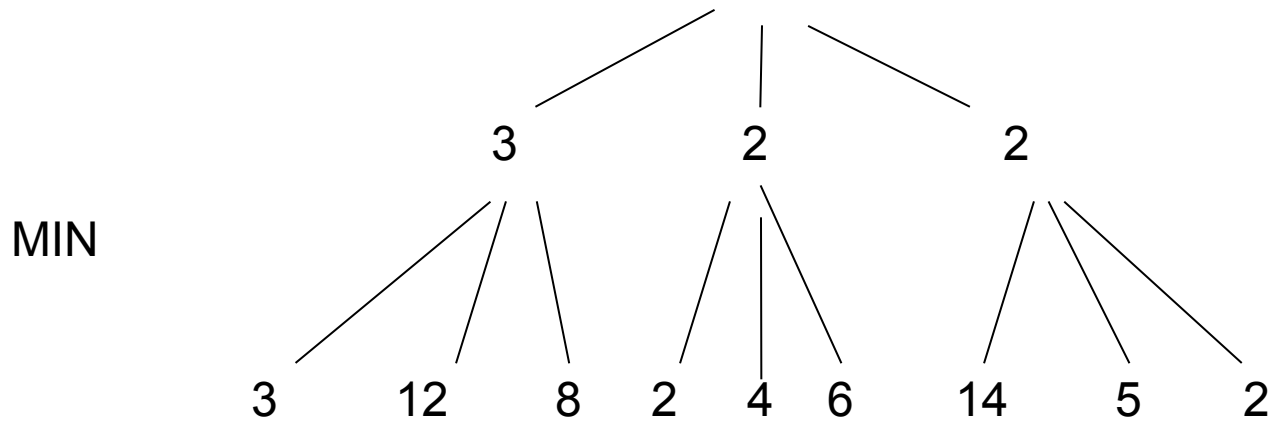


Minimax



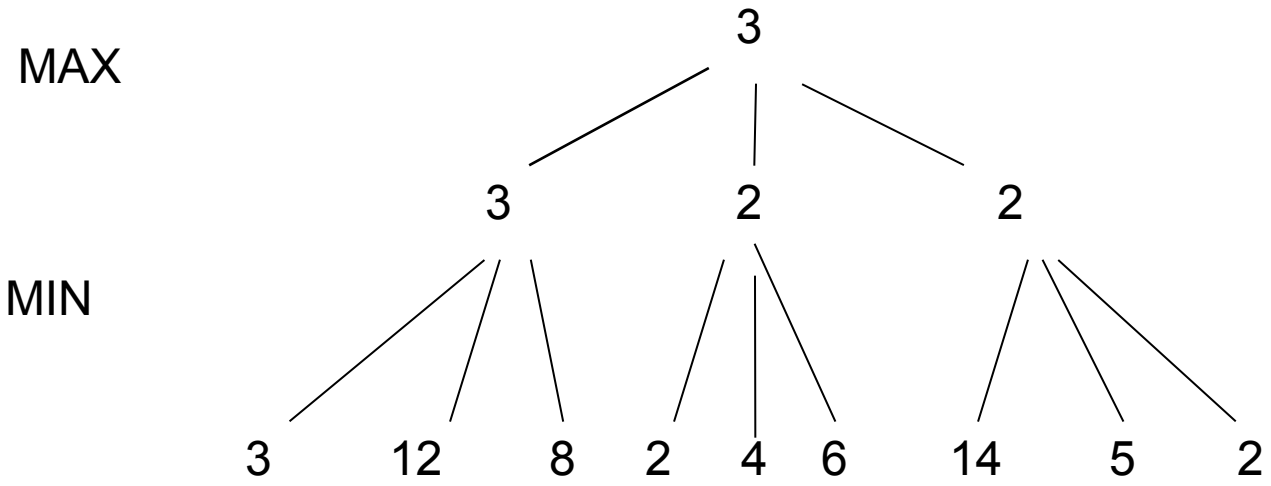
- Minimize opponent's chance
- Maximize your chance

Minimax



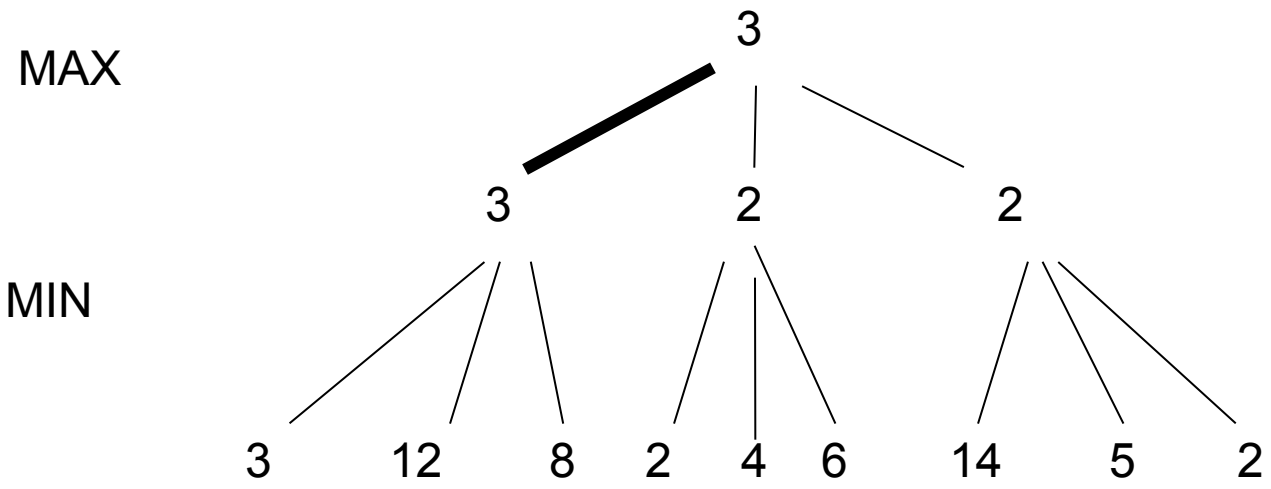
- Minimize opponent's chance
- Maximize your chance

Minimax



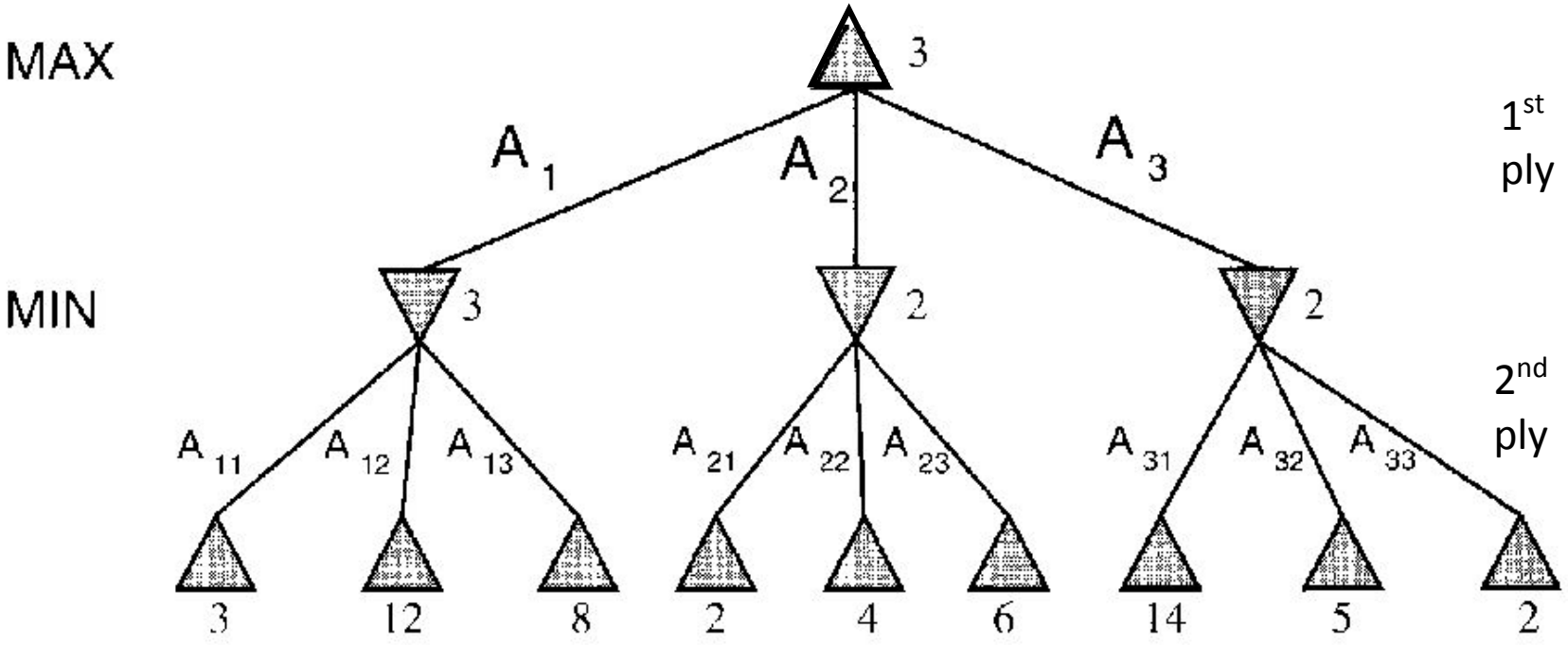
- Minimize opponent's chance
- Maximize your chance

Minimax

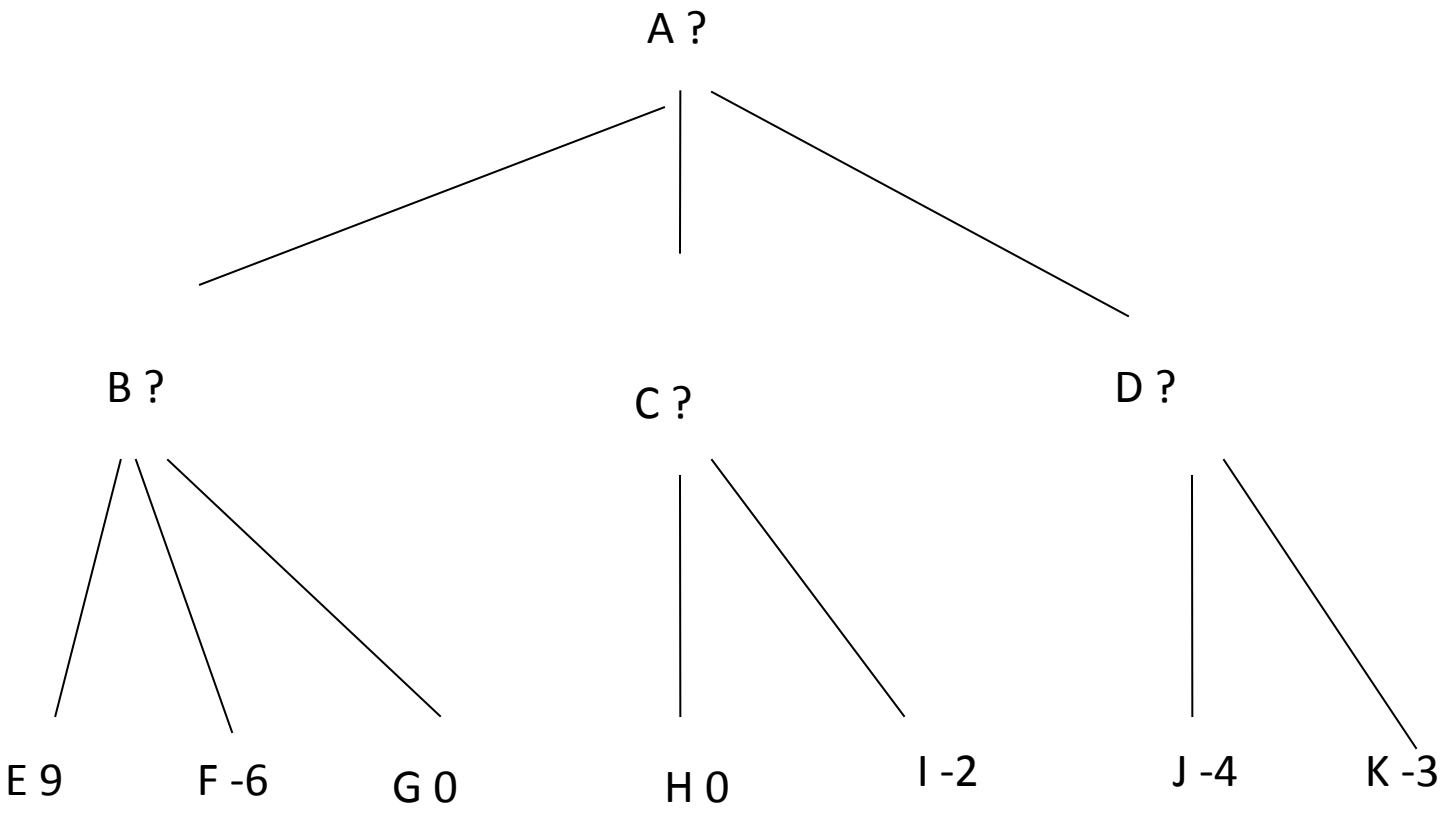


- Minimize opponent's chance
- Maximize your chance

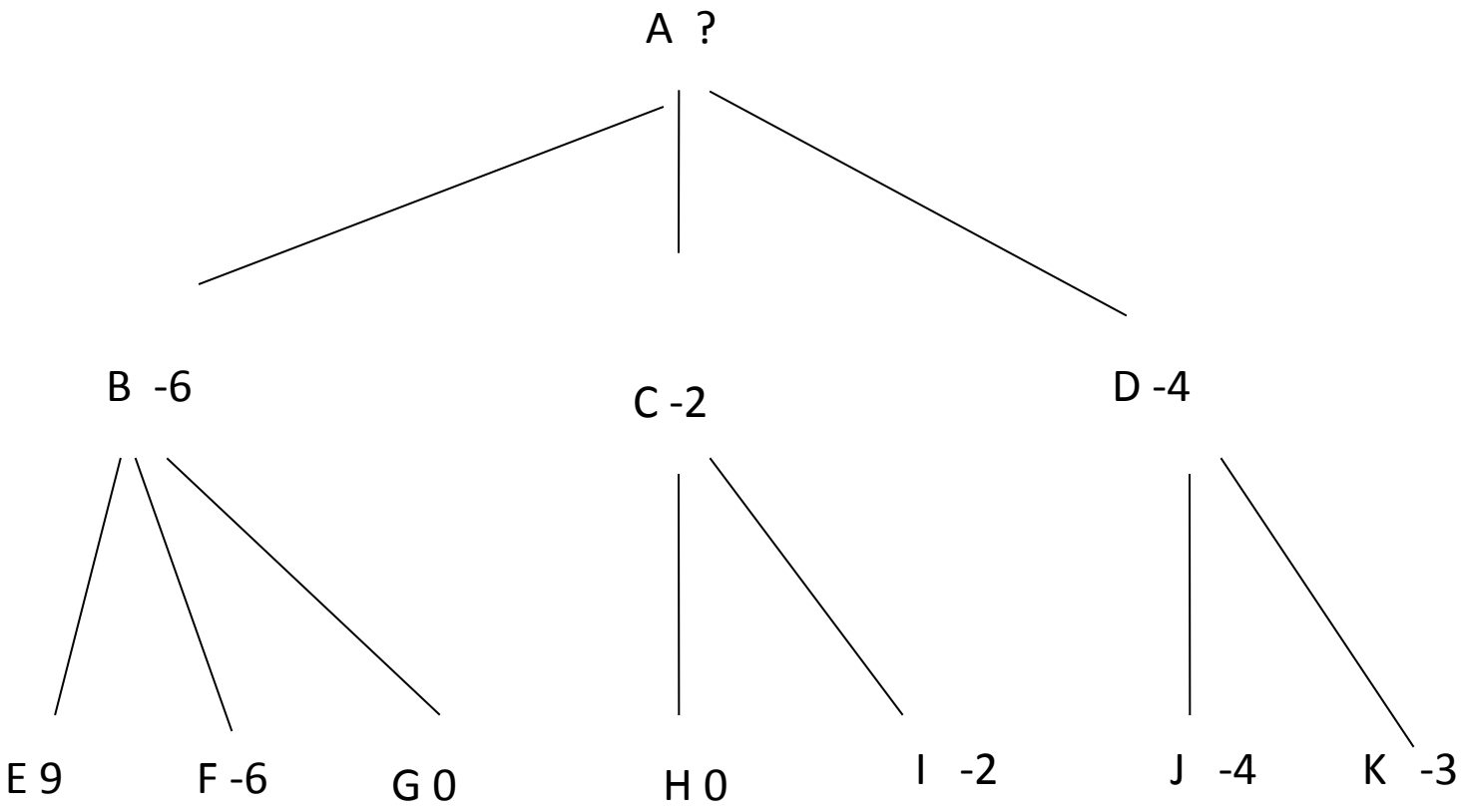
minimax = maximum of the minimum



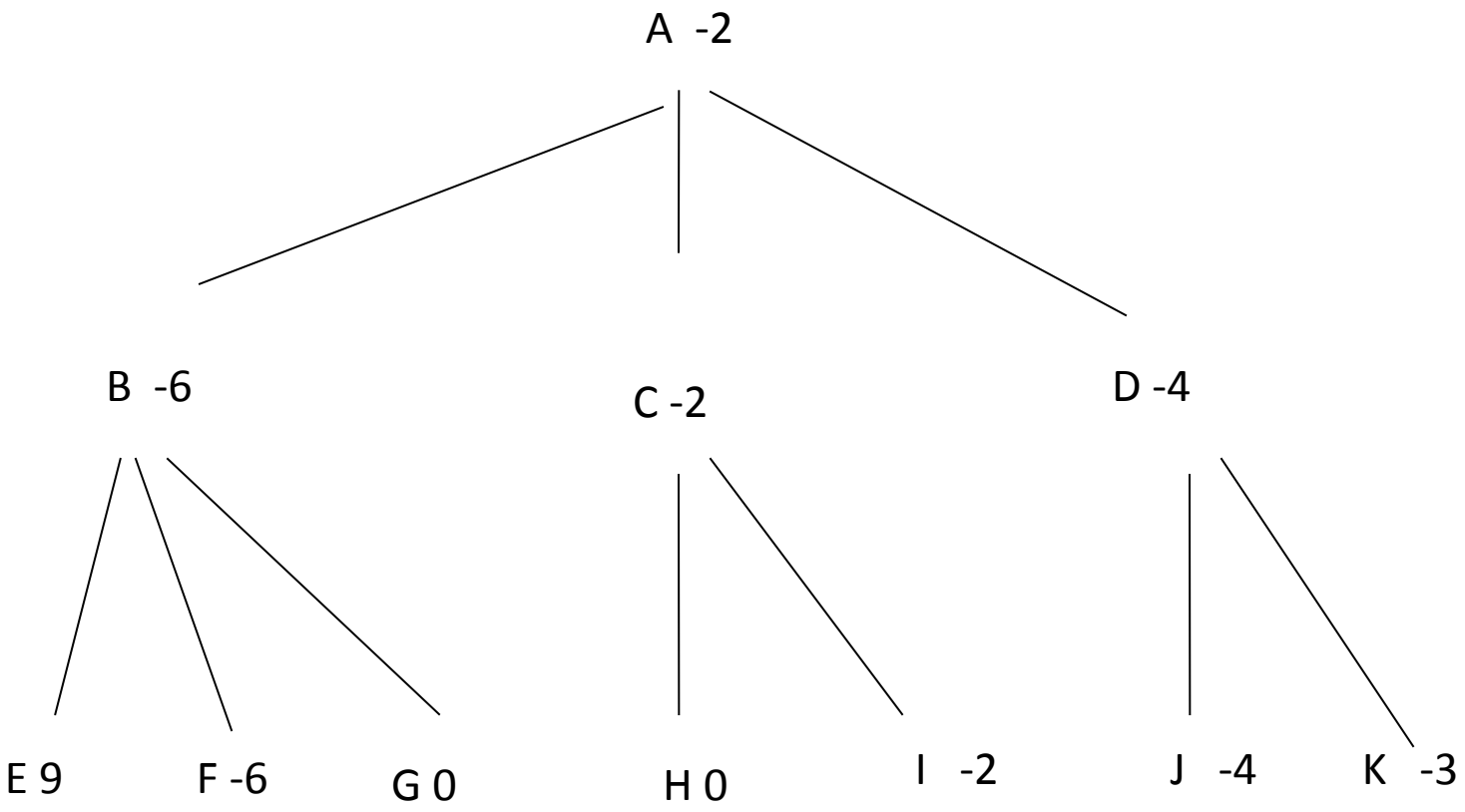
Mini Max Example



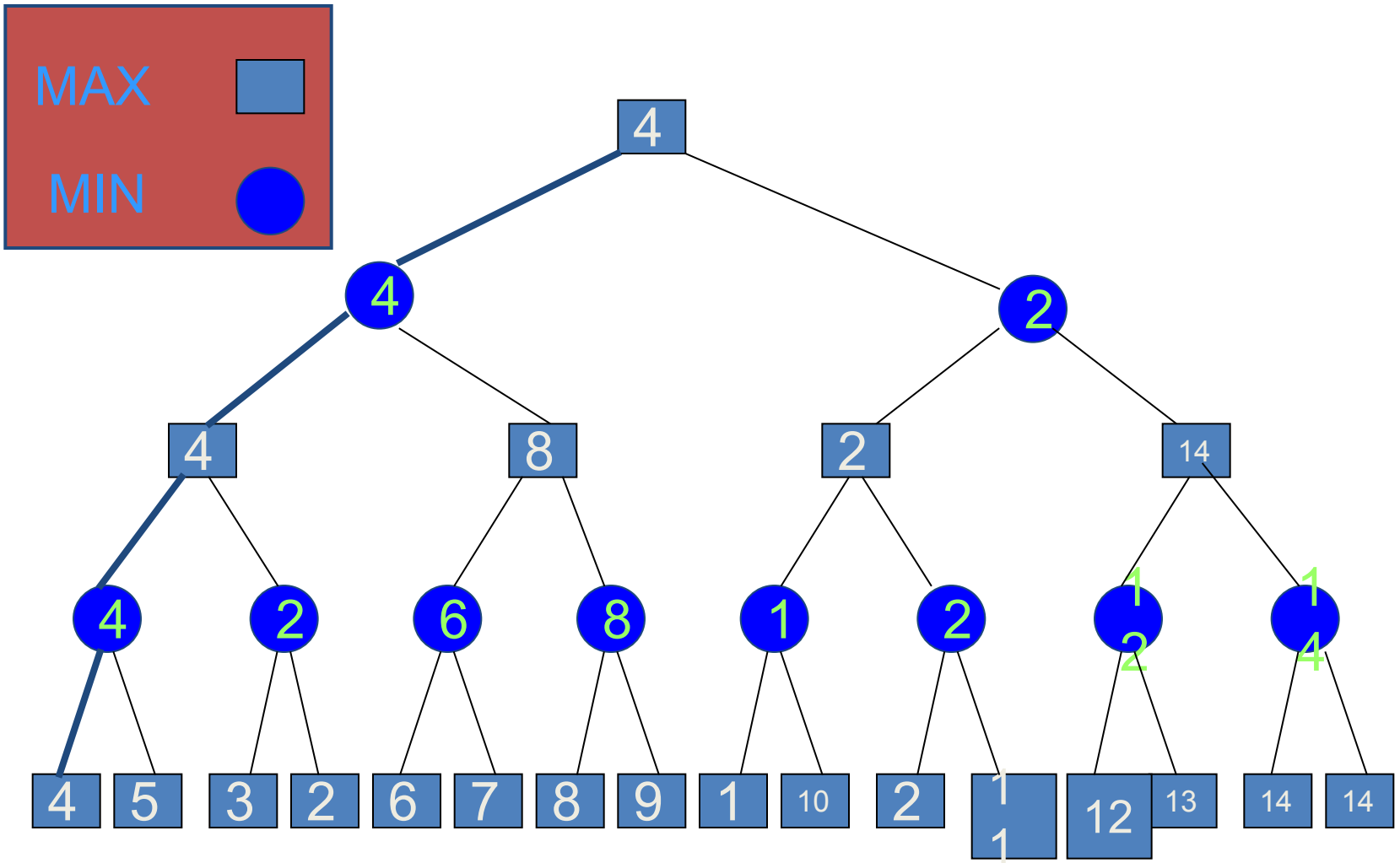
Mini Max Example

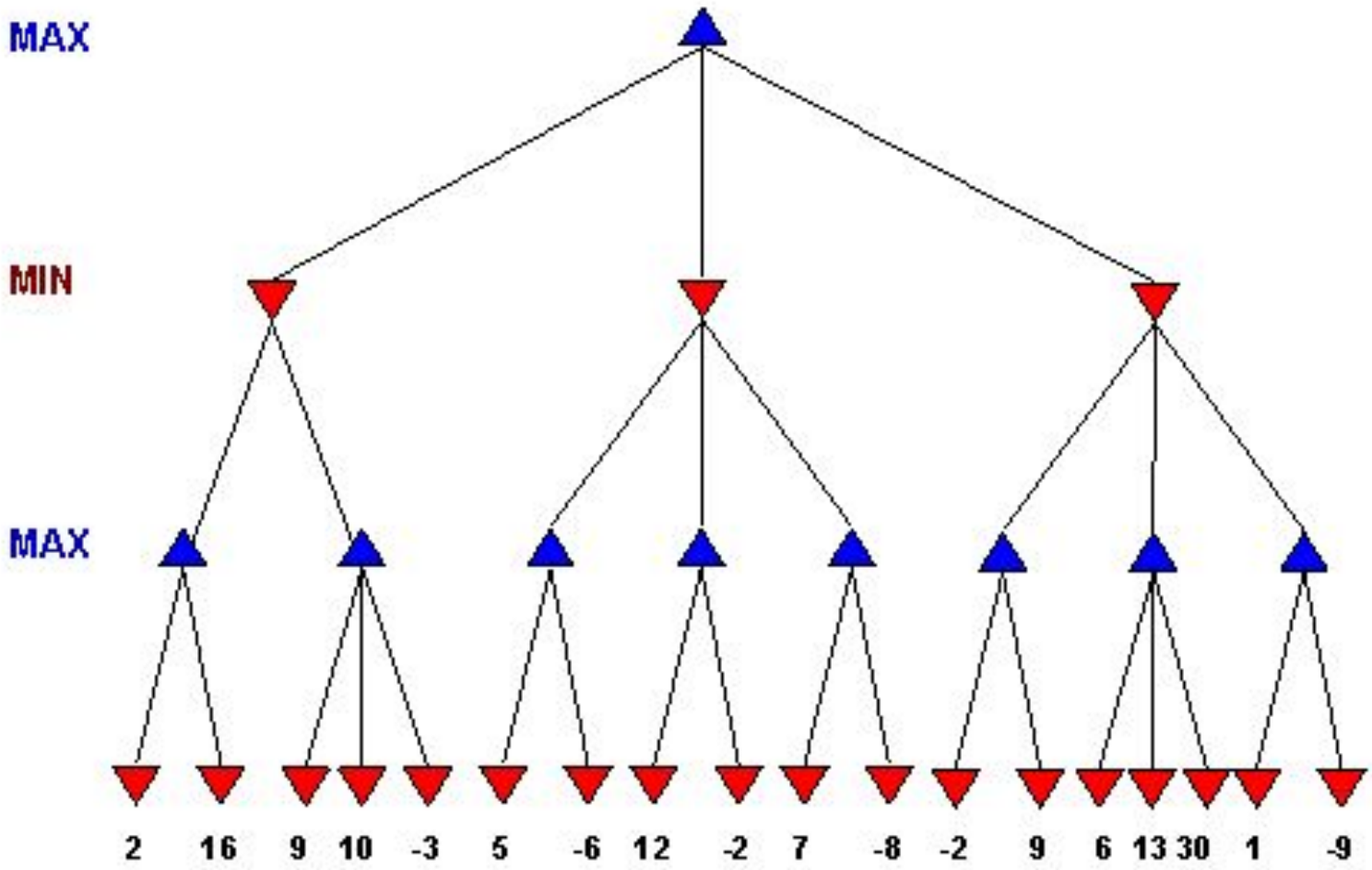


Mini Max Example



Minimax search example Minimax Tree




MAX
MIN
MAX
2
16
9
10
-3
5
-6
12
-2
7
-8
-2
9
6
13
30
1
-9

Minimax: Recursive implementation

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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

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

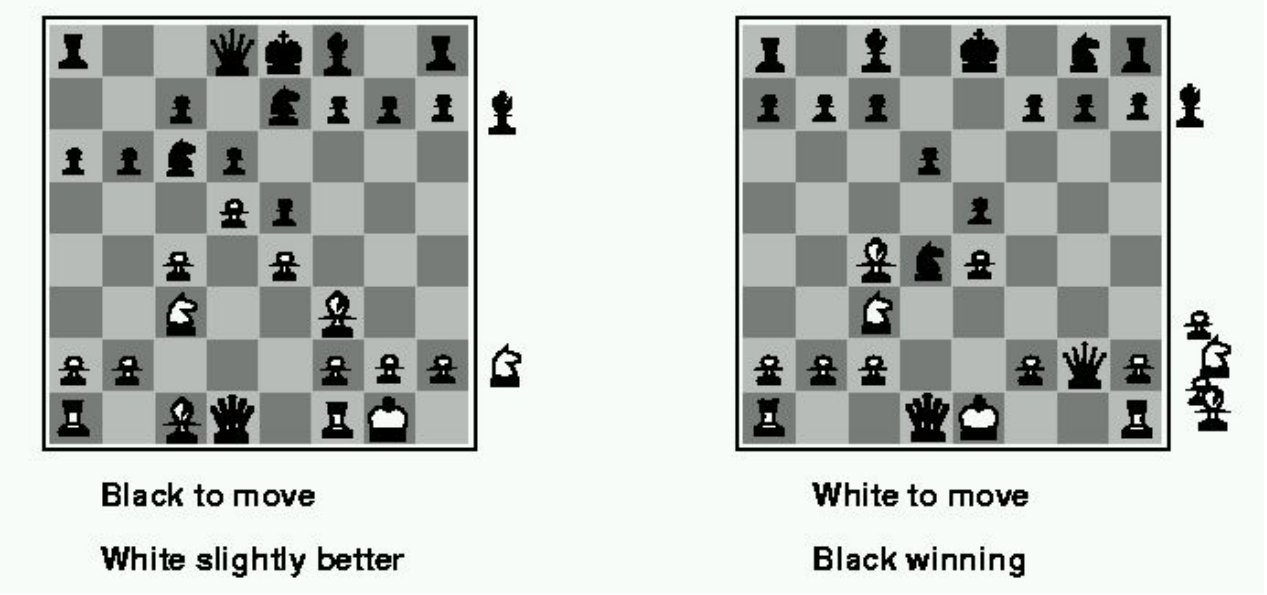
Complete: Yes, for finite state-space
Optimal: Yes

Time complexity: $O(b^m)$
Space complexity: O(bm) (= DFS Does not keep all nodes in memory.)

Move evaluation without complete search

- **Complete search is too complex and impractical**
- **Evaluation function: evaluates value of state using heuristics and cuts off search**
- **New MINIMAX:**
  - CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  - EVAL: evaluation function to replace utility function (e.g., number of chess pieces taken)

Evaluation functions



- **Weighted linear evaluation function: to combine *n* heuristics**
  $f = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$

E.g, *w's* could be the values of pieces (1 for prawn, 3 for bishop etc.)
***f's* could be the number of type of pieces on the board**

Minimax with cutoff: viable algorithm?

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

Search with an opponent

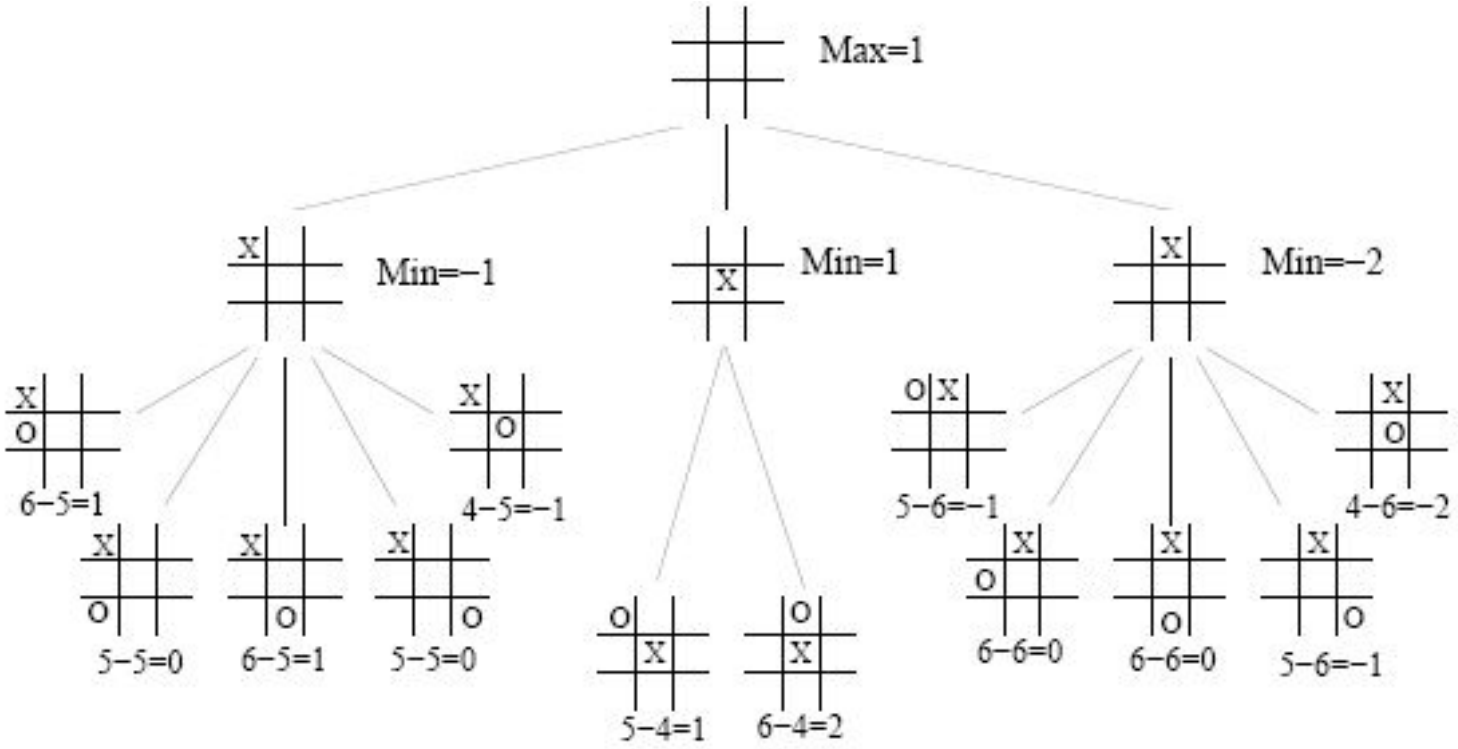
- **Heuristic approximation: defining an evaluation function which indicates how close a state is from a winning (or losing) move**
- **This function includes domain information.**
- **It does not represent a cost or a distance in steps.**
- **Conventionally:**
  - A winning move is represented by the value“+∞”.
  - A losing move is represented by the value “-∞”.
  - The algorithm searches with limited depth.

- **Each new decision implies repeating part of the search.**

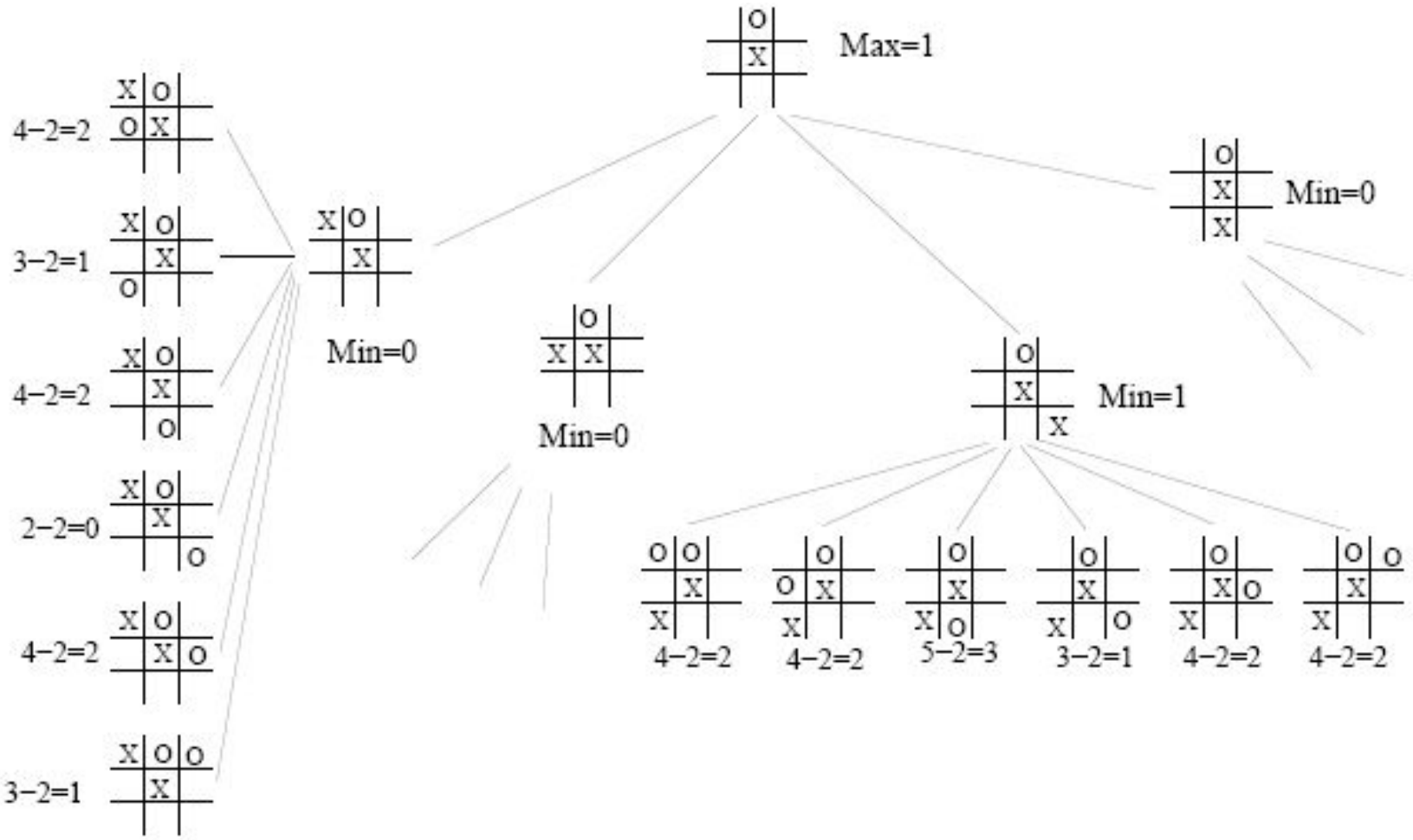
Example: tic-tac-toe

- **e (evaluation function → integer) = number of available rows, columns, diagonals for MAX - number of available rows, columns, diagonals for MIN**
- **MAX plays with "X" and desires maximizing e.**
- **MIN plays with "0" and desires minimizing e.**
- **Symmetries are taken into account.**
- **A depth limit is used (2, in the example).**

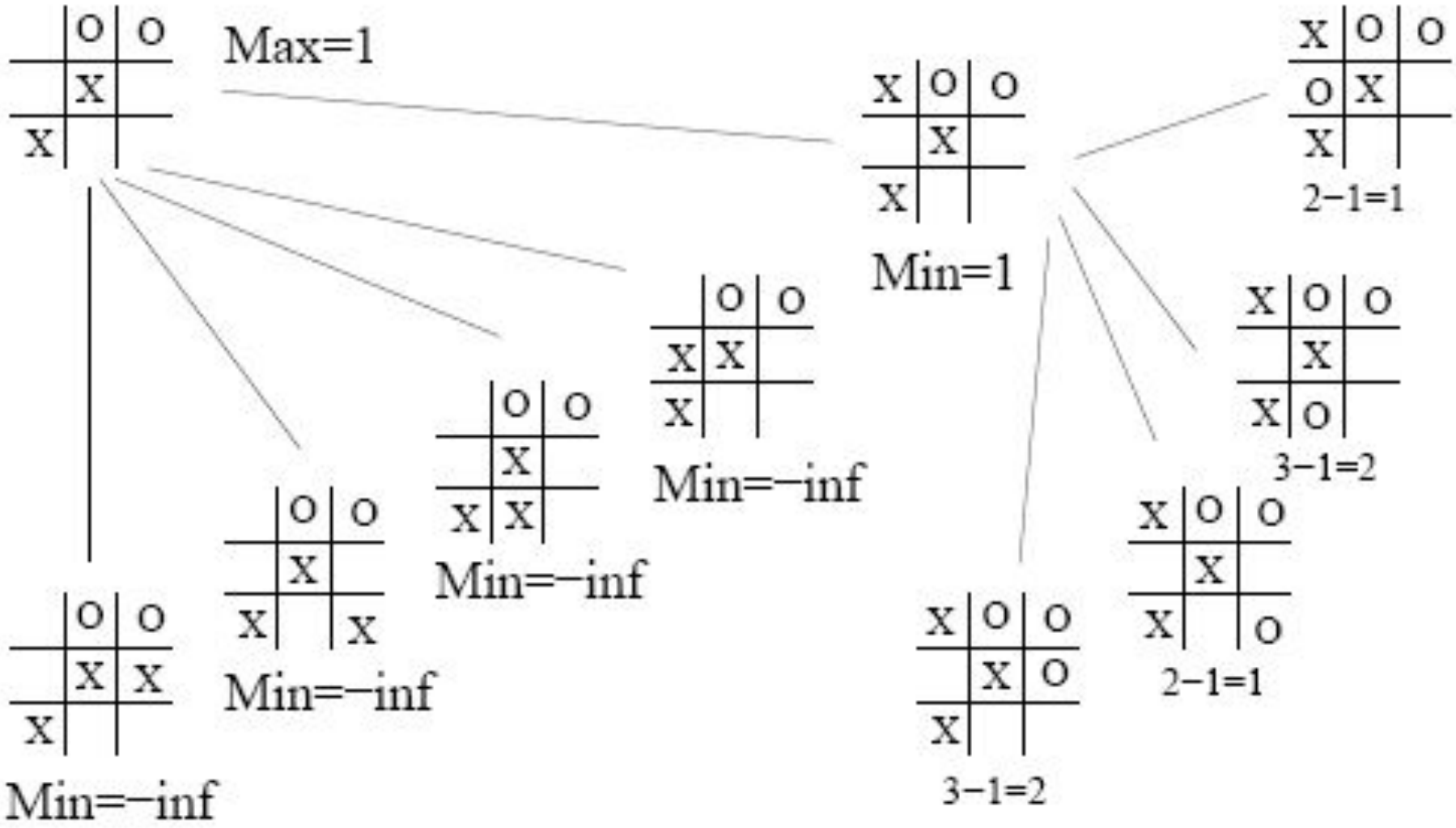
Example: tic-tac-toe



Example: tic-tac-toe



Example: tic-tac-toe



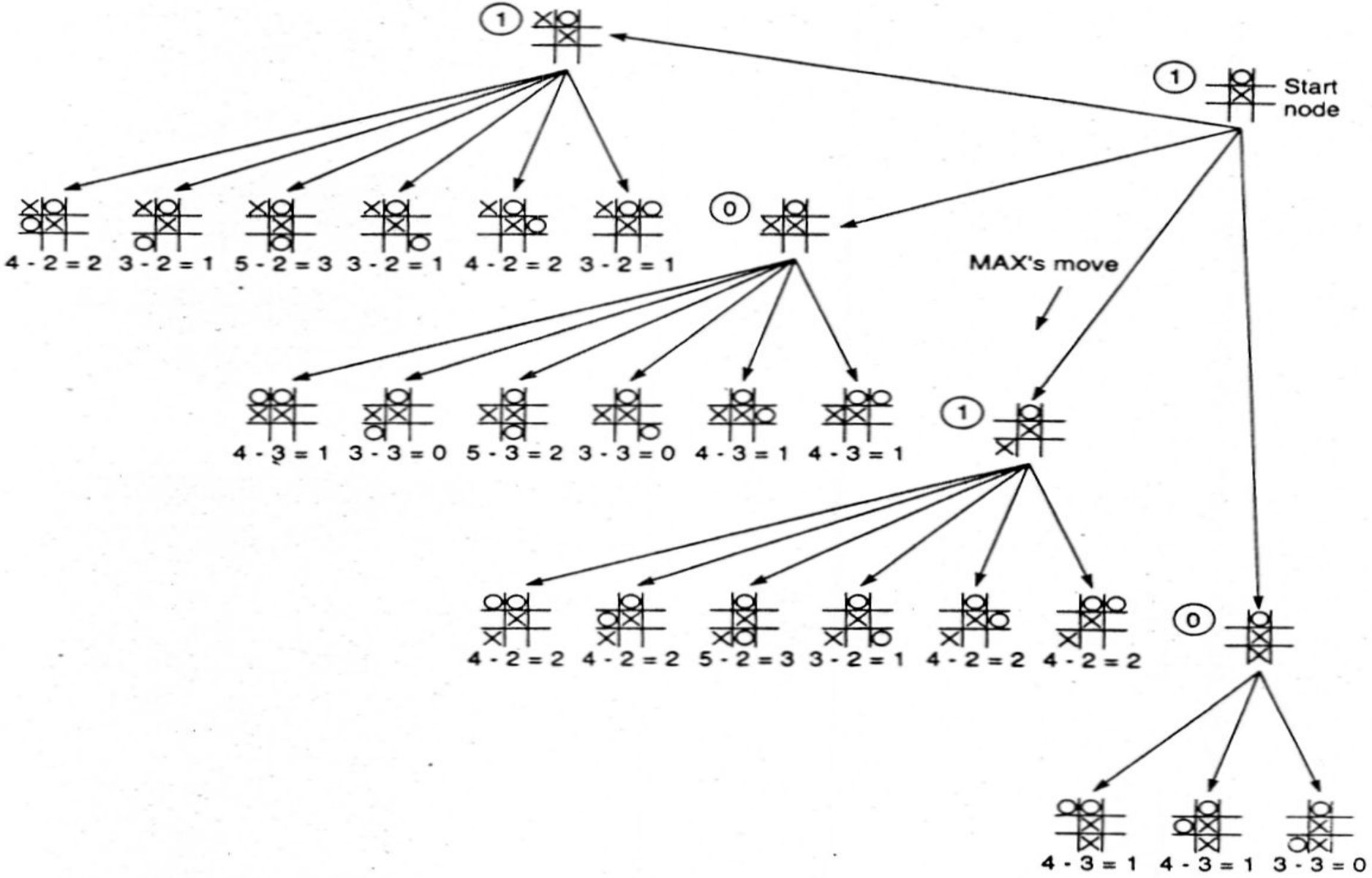


Figure 4.18 Two-ply minimax applied to **X**'s second move of tic-tac-toe.

Backup Values

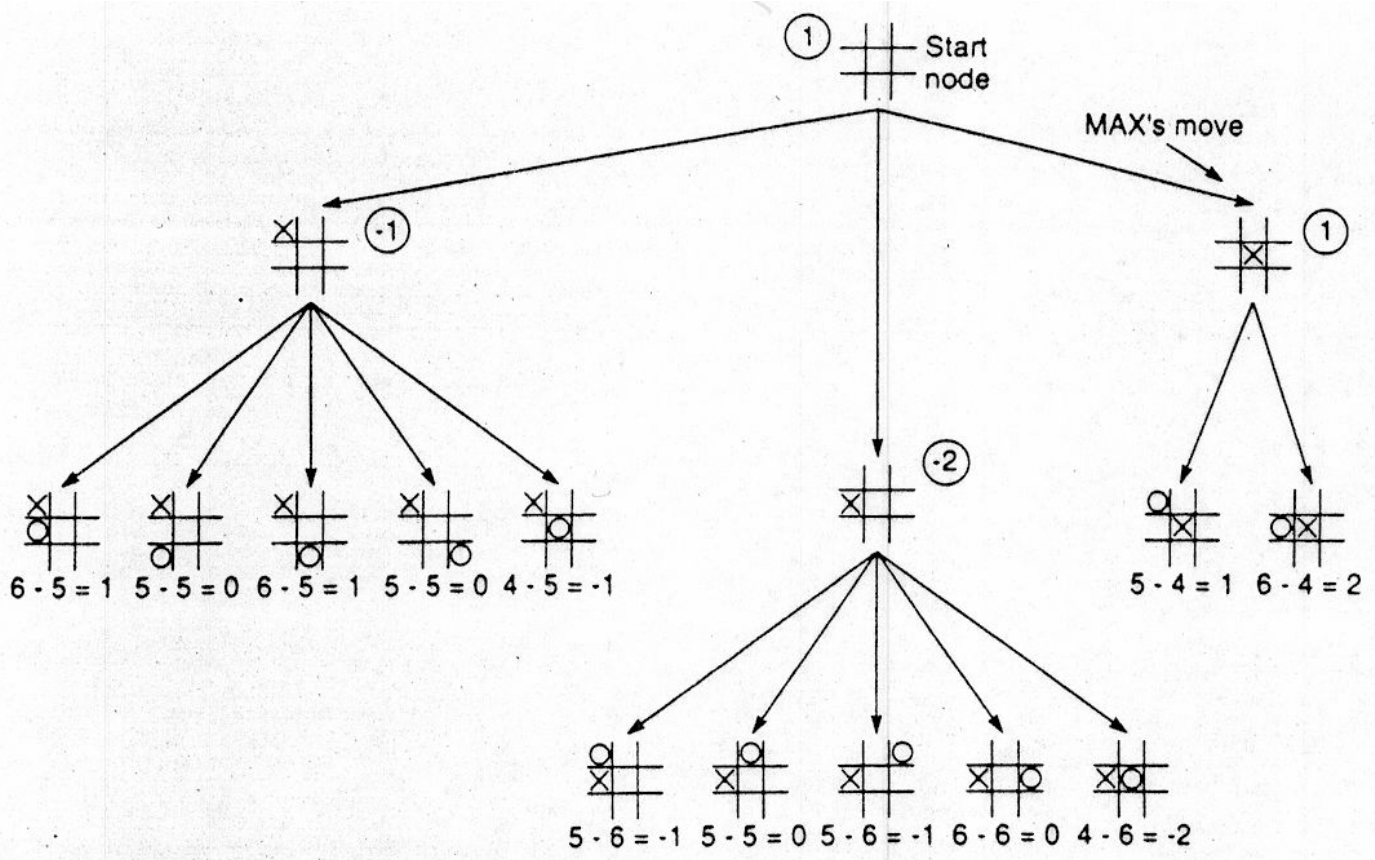


Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.

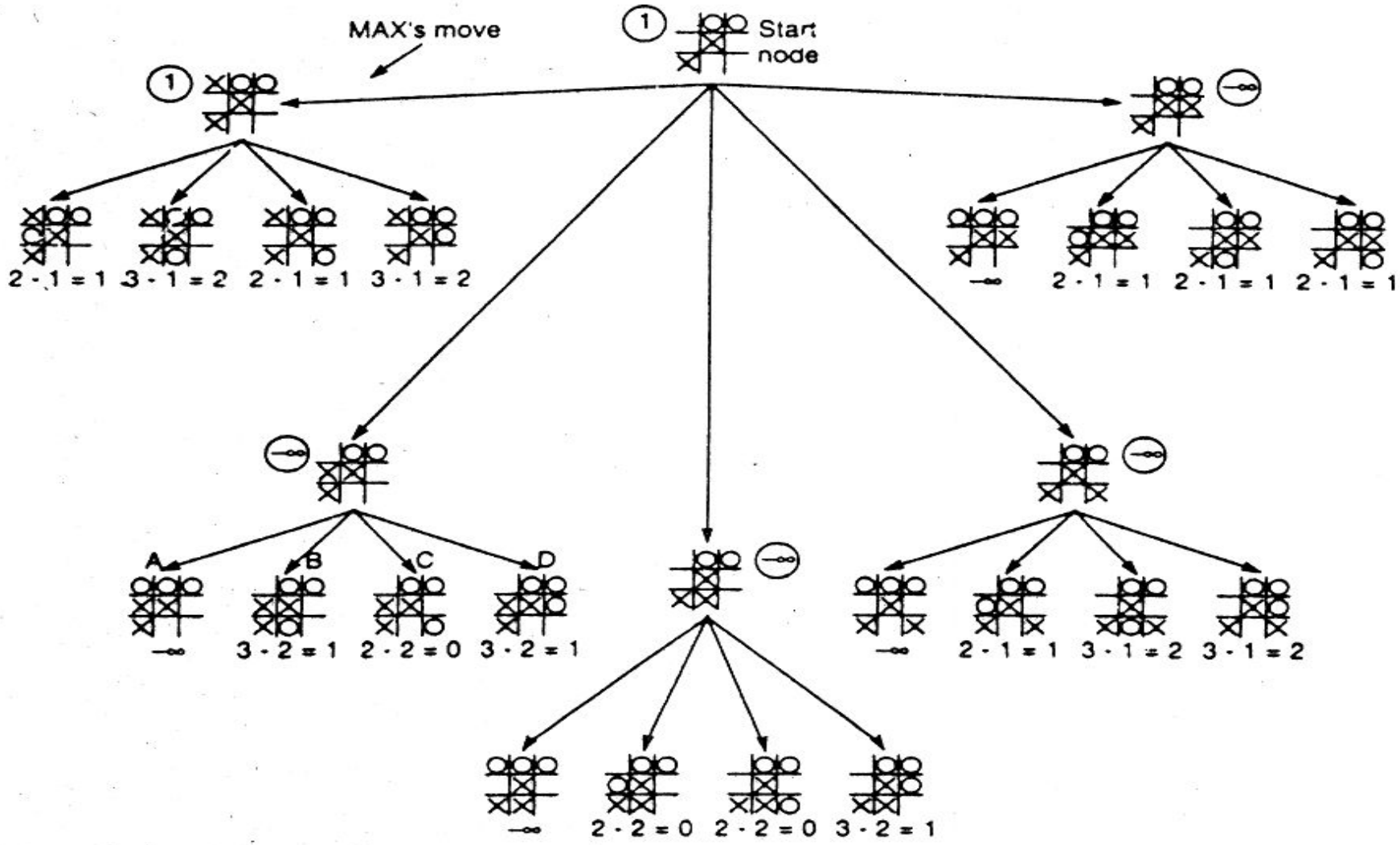


Figure 4.19 Two-ply minimax applied to **X**'s move near end game.

The minimax algorithm

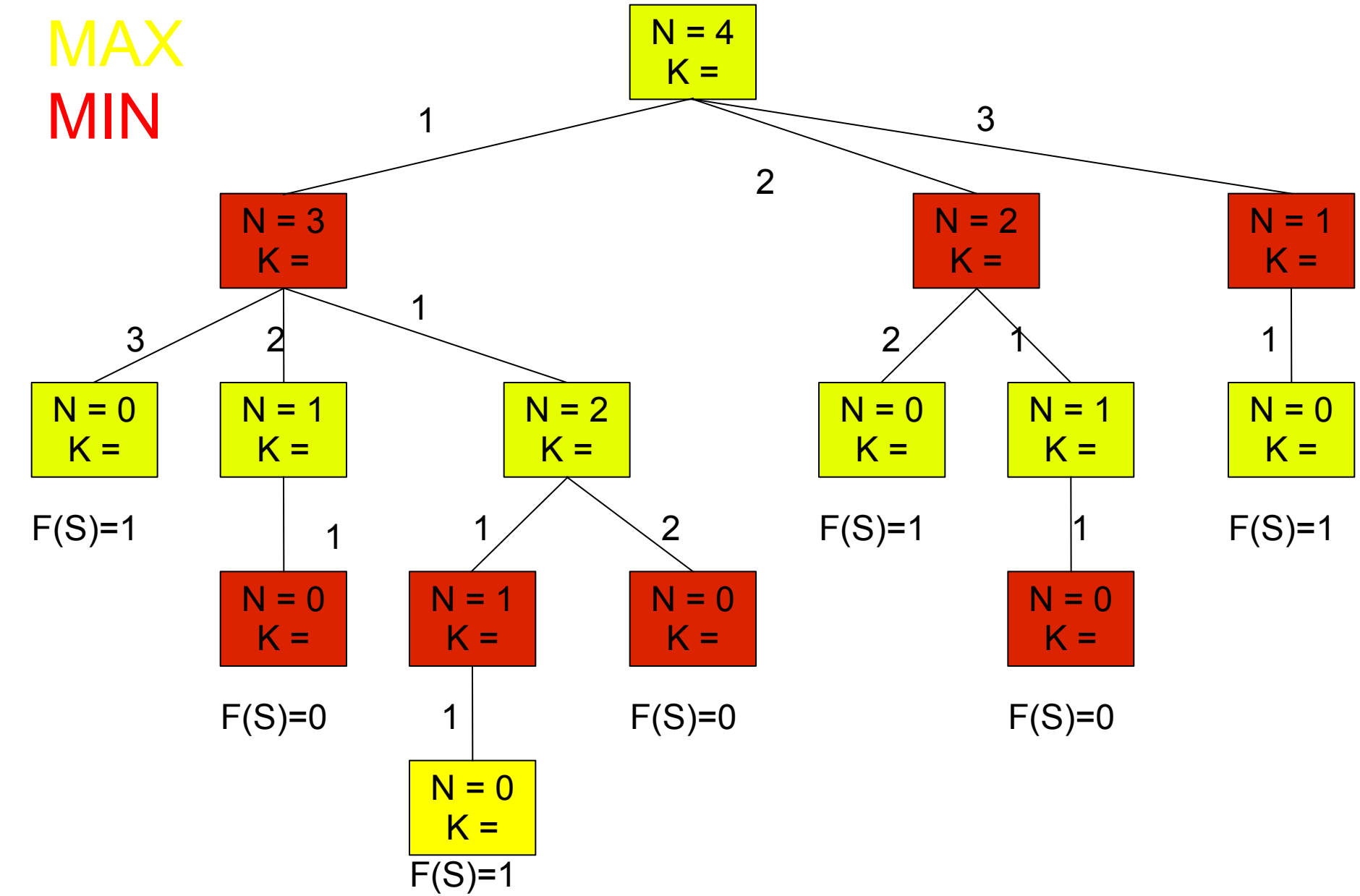
- **The minimax algorithm computes the minimax decision from the current state.**
- **It uses a simple recursive computation of the minimax values of each successor state:**
  - directly implementing the defining equations.
- **The recursion proceeds all the way down to the leaves of the tree.**
- **Then the minimax values are backed up through the tree as the recursion unwinds.**

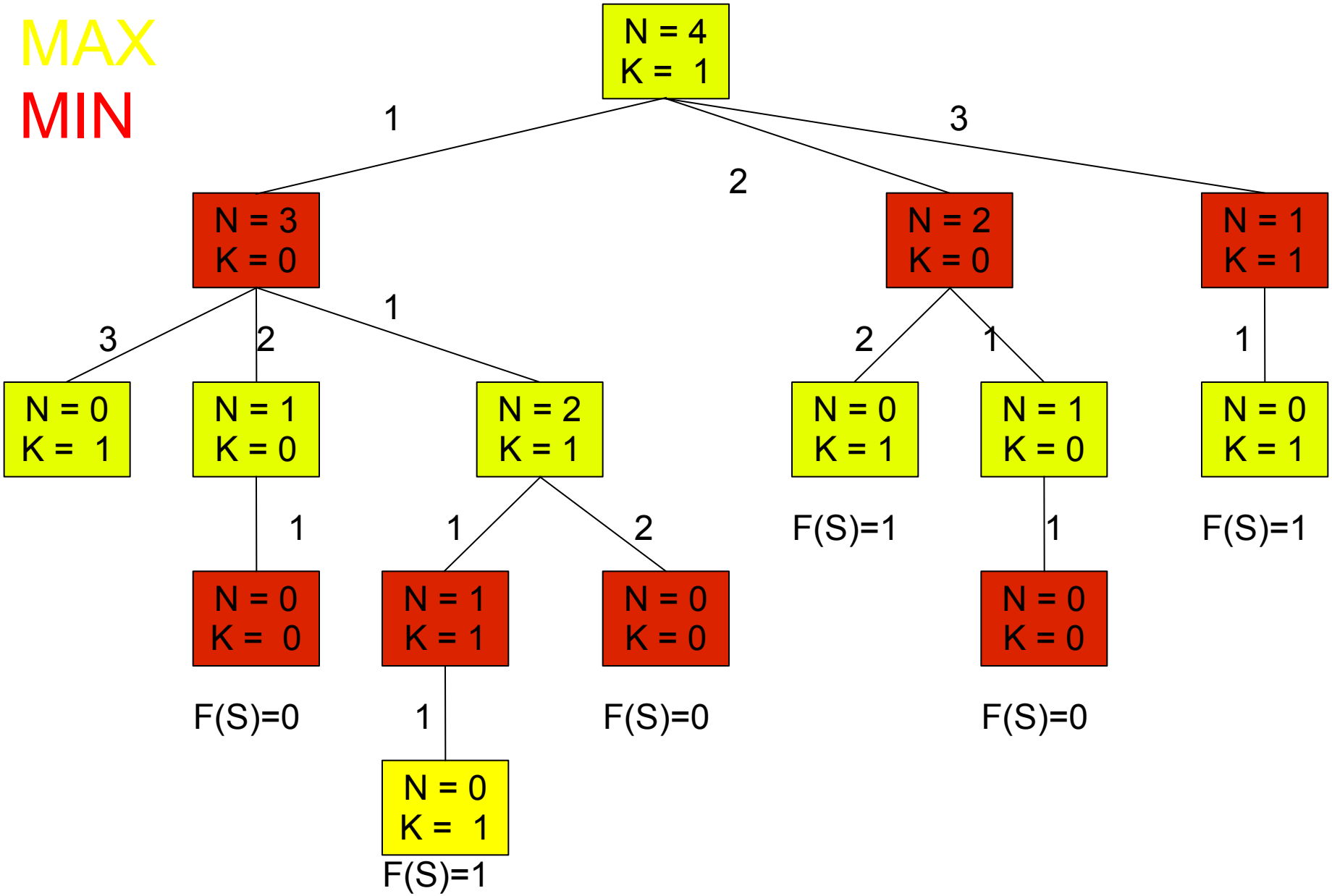
Example

- **Coins game**
  - There is a stack of N coins
  - In turn, players take 1, 2, or 3 coins from the stack
  - The player who takes the last coin loses

Coins Game: Formal Definition

- **Initial State: The number of coins in the stack**
- **Operators:**
  1. Remove one coin
  2. Remove two coins
  3. Remove three coins
- **Terminal Test: There are no coins left on the stack**
- **Utility Function: F(S)**
  - F(S) = 1 if MAX wins, 0 if MIN wins


MAX
MIN
N = 4
K =
1
2
3
N = 3
K =
N = 2
K =
N = 1
K =
3
2
1
2
1
1
N = 0
K =
N = 1
K =
N = 2
K =
N = 0
K =
N = 1
K =
N = 0
K =
F(S)=1
1
1
2
F(S)=1
1
F(S)=1
N = 0
K =
N = 1
K =
N = 0
K =
N = 0
K =
F(S)=0
1
F(S)=0
F(S)=0
N = 0
K =
F(S)=1


MAX
MIN
N = 4
K = 1
1
2
3
N = 3
K = 0
N = 2
K = 0
N = 1
K = 1
3
2
1
2
1
1
N = 0
K = 1
N = 1
K = 0
N = 2
K = 1
N = 0
K = 1
N = 1
K = 0
N = 0
K = 1
1
1
2
1
F(S)=1
F(S)=1
N = 0
K = 0
N = 1
K = 1
N = 0
K = 0
N = 0
K = 0
F(S)=0
1
F(S)=0
F(S)=0
N = 0
K = 1
F(S)=1

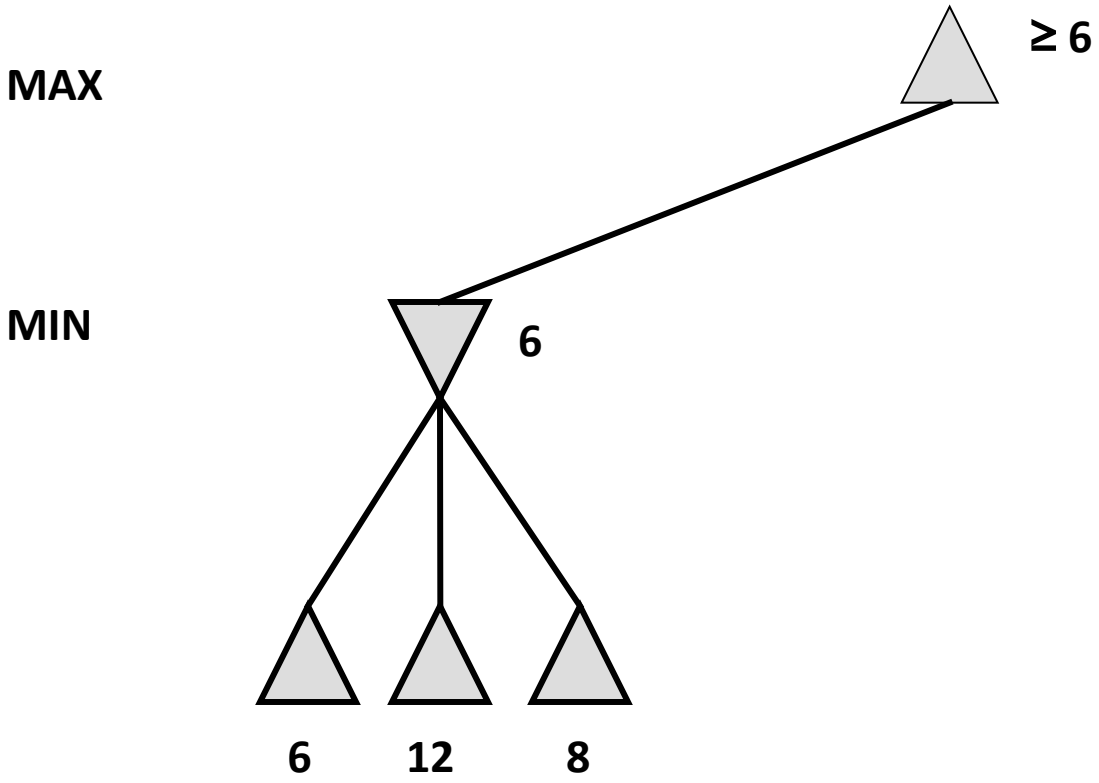
α-β pruning: search cutoff

- **Pruning: eliminating a branch of the search tree from consideration without exhaustive examination of each node**

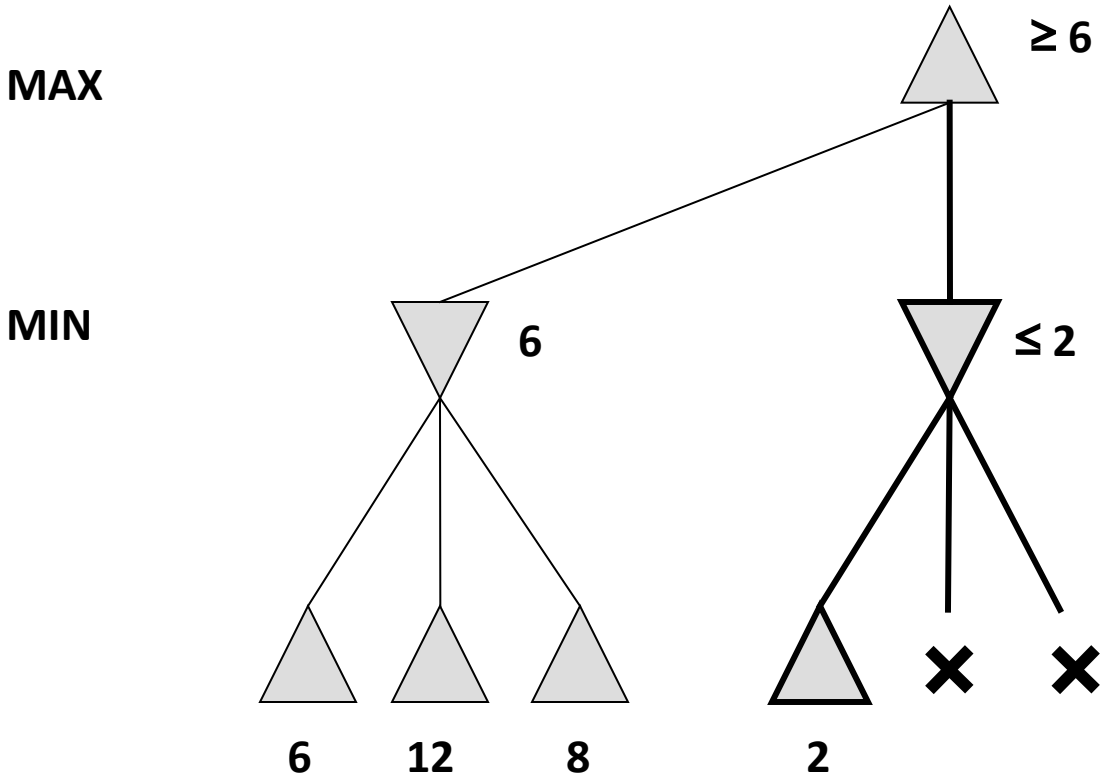
- **α-β pruning: the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.**

- **Does it work? Yes, in roughly cuts the branching factor from b to √b resulting in double as far look-ahead than pure minimax**

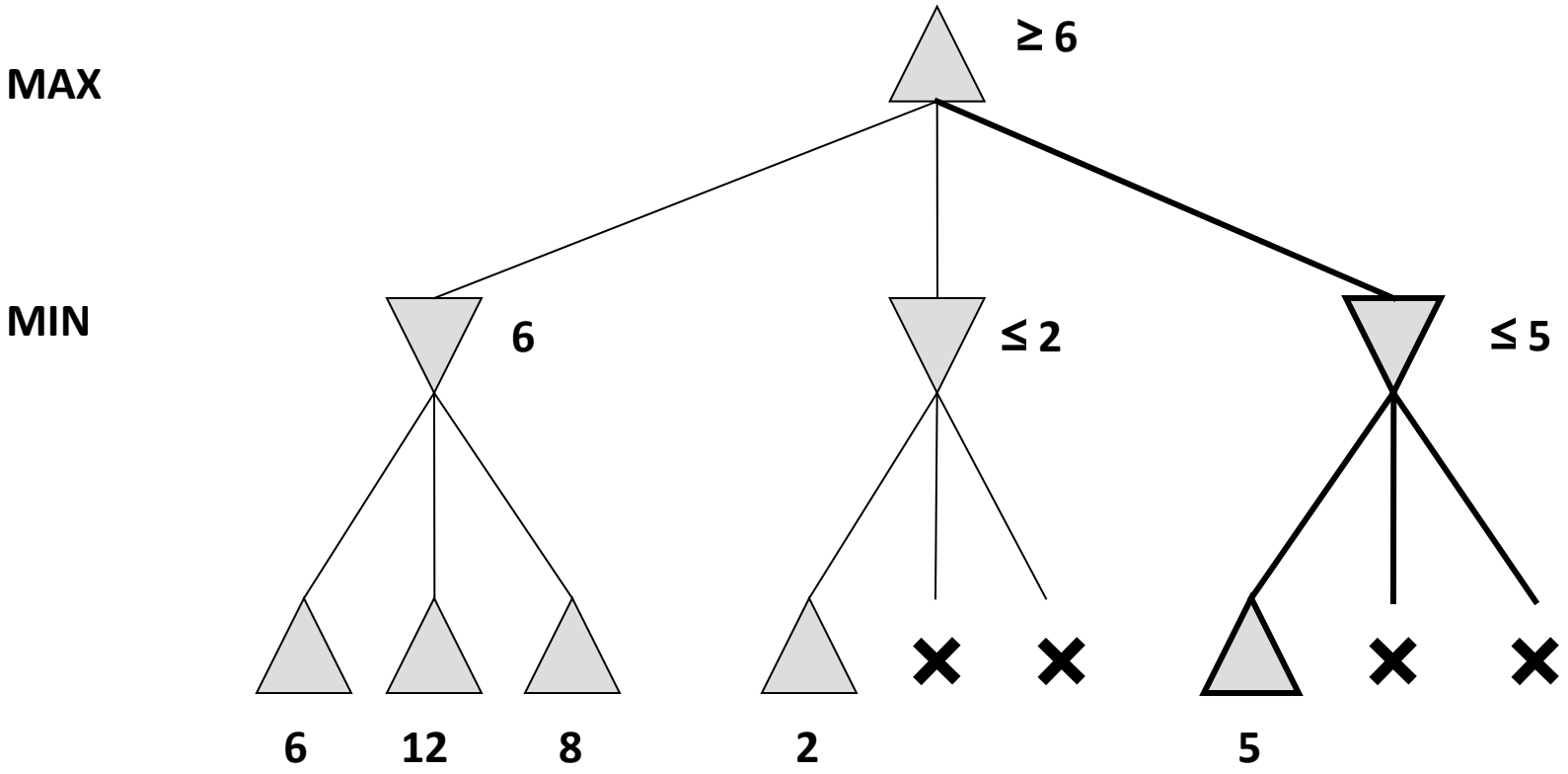
α-β pruning: example



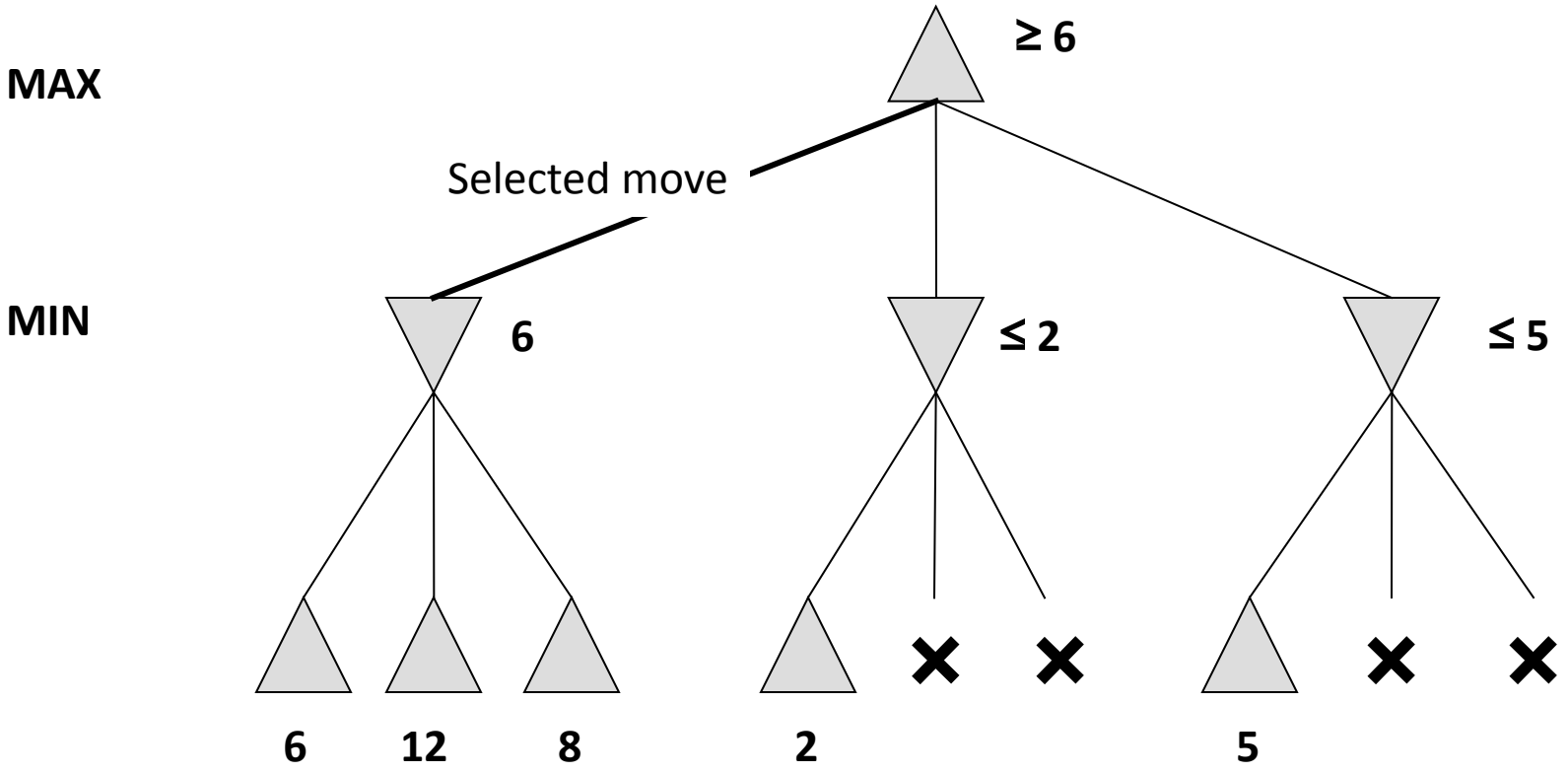
α-β pruning: example



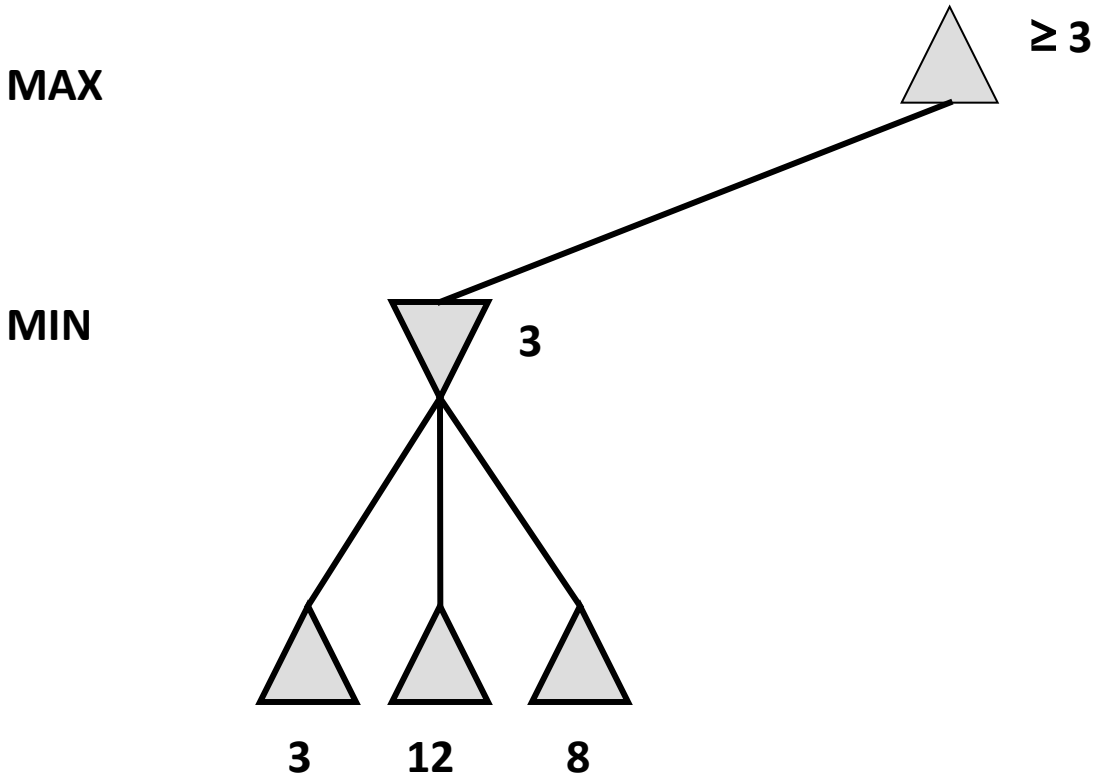
α-β pruning: example



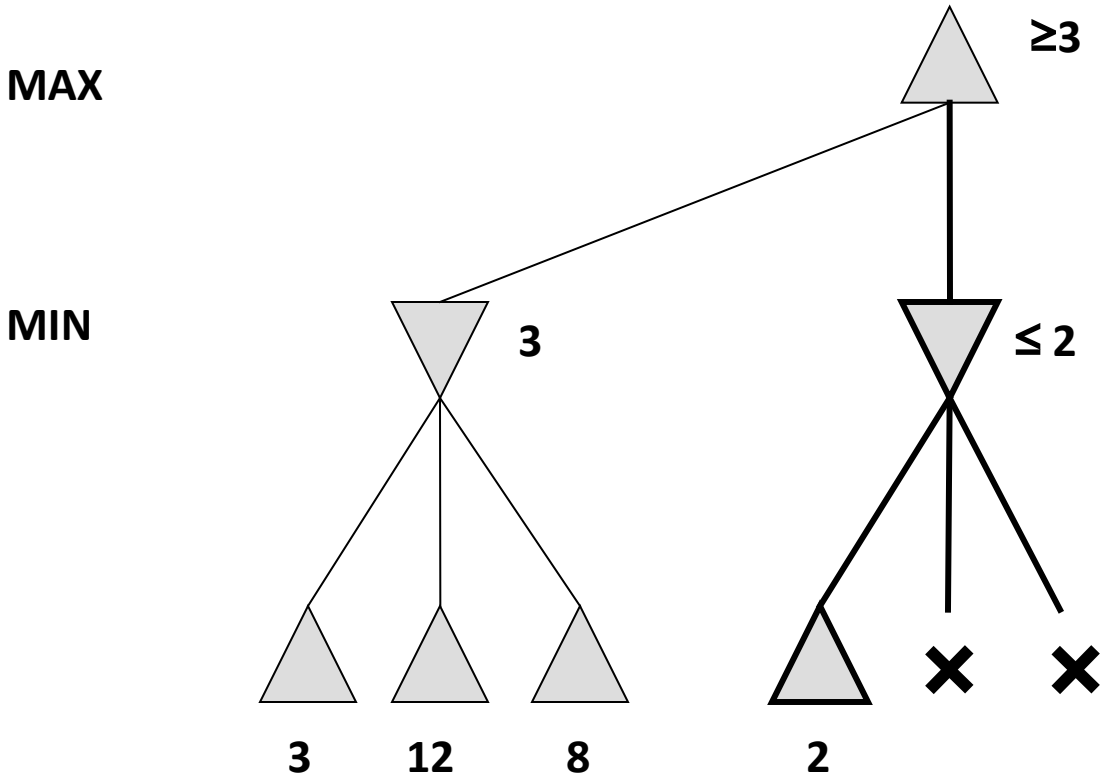
α-β pruning: example



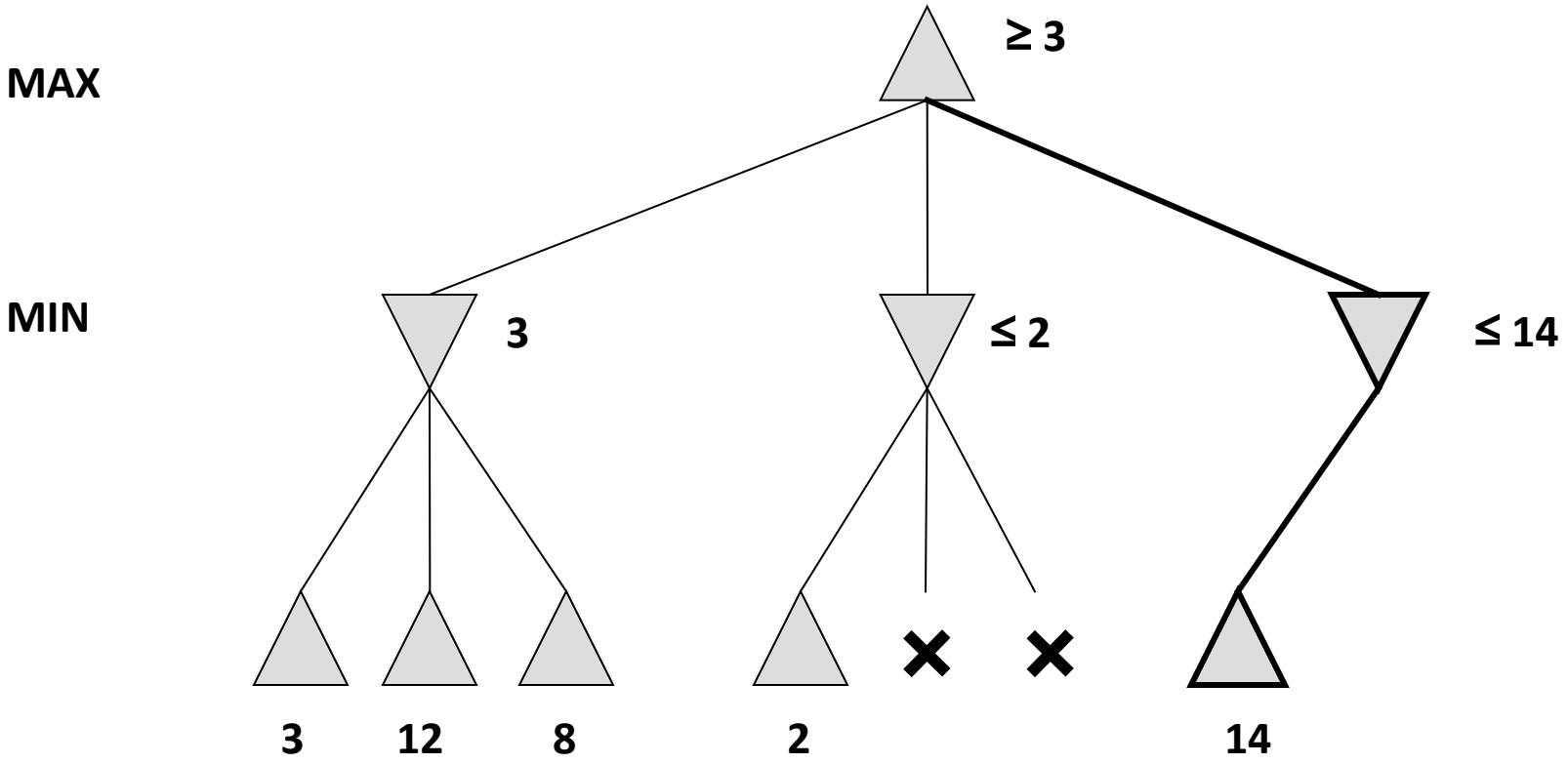
α-β pruning: example 2



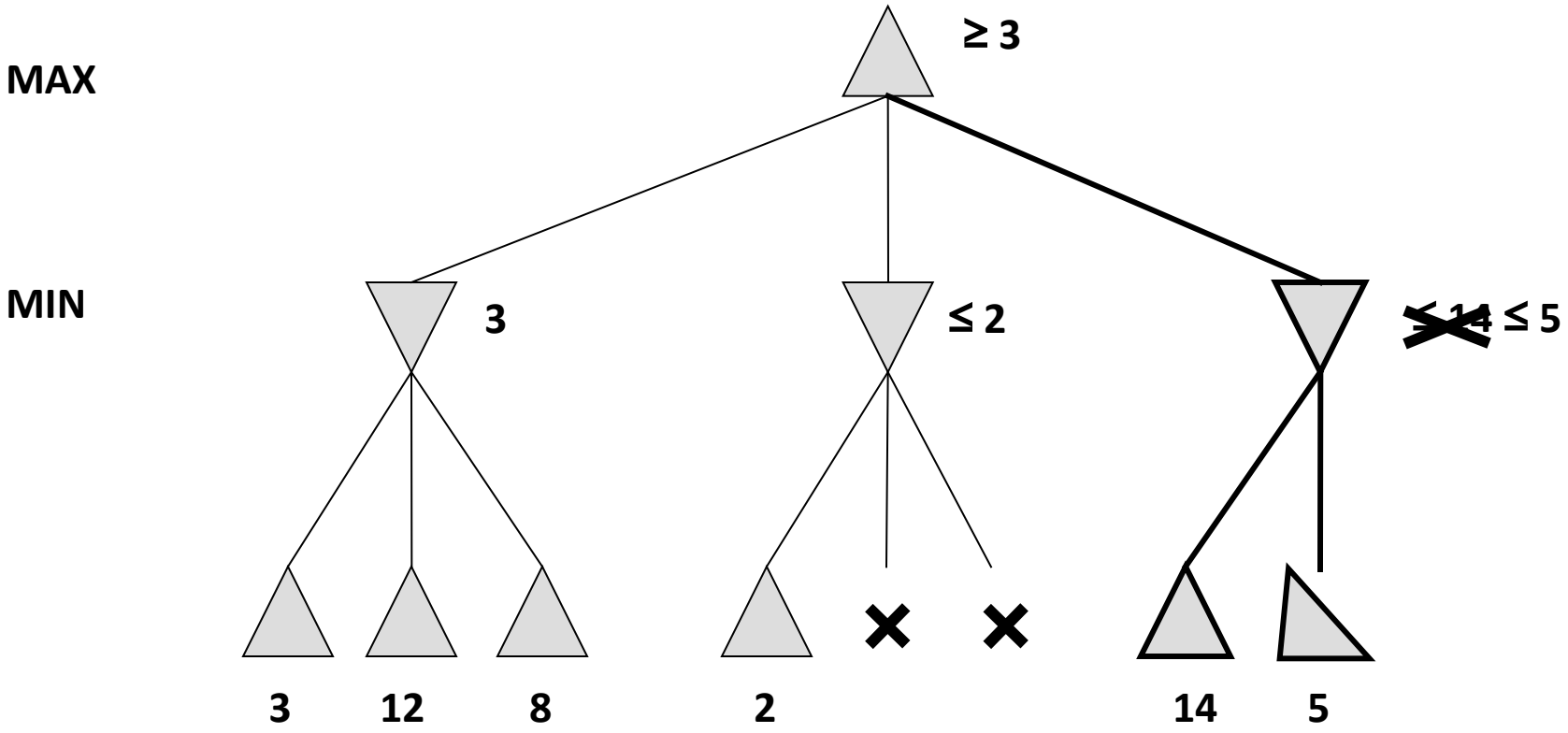
α-β pruning: example 2



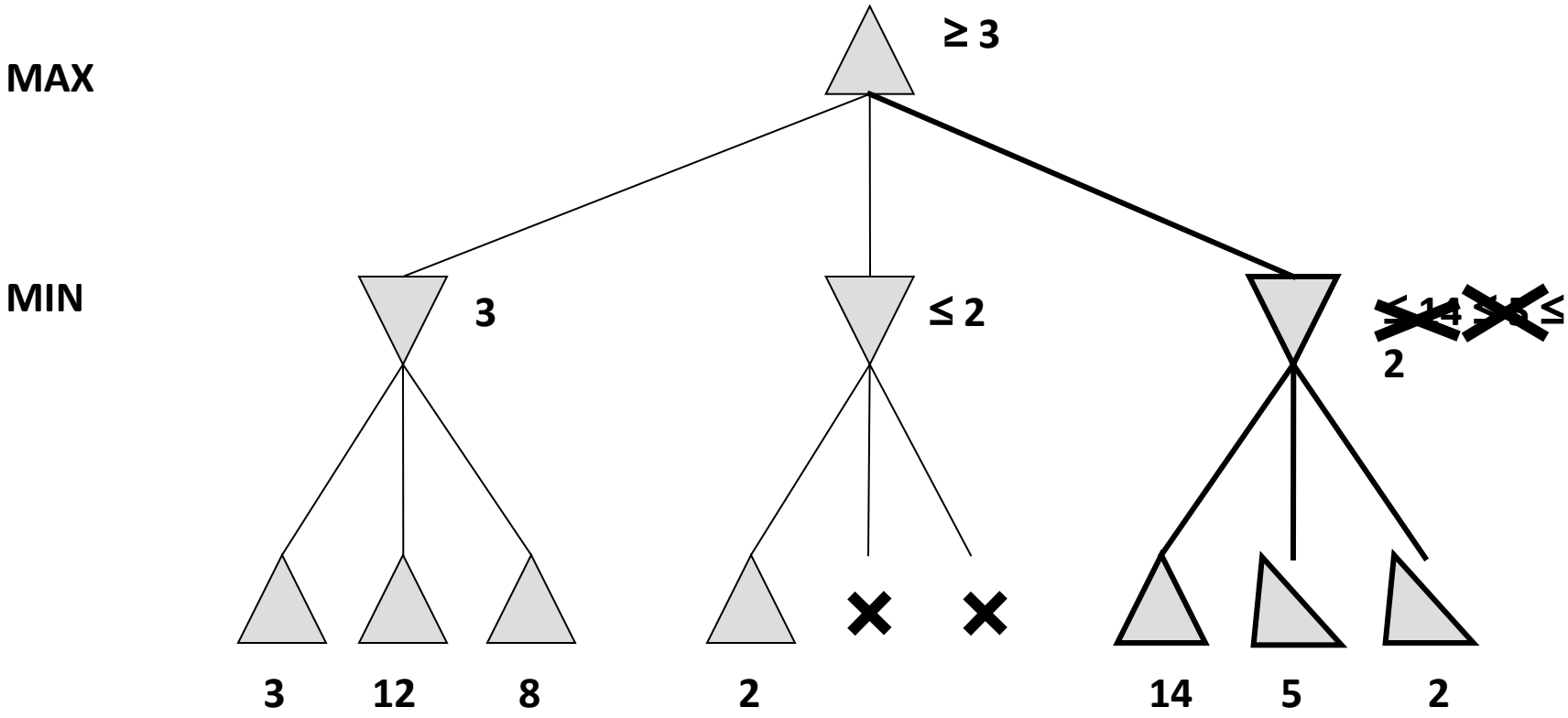
α-β pruning: example 2



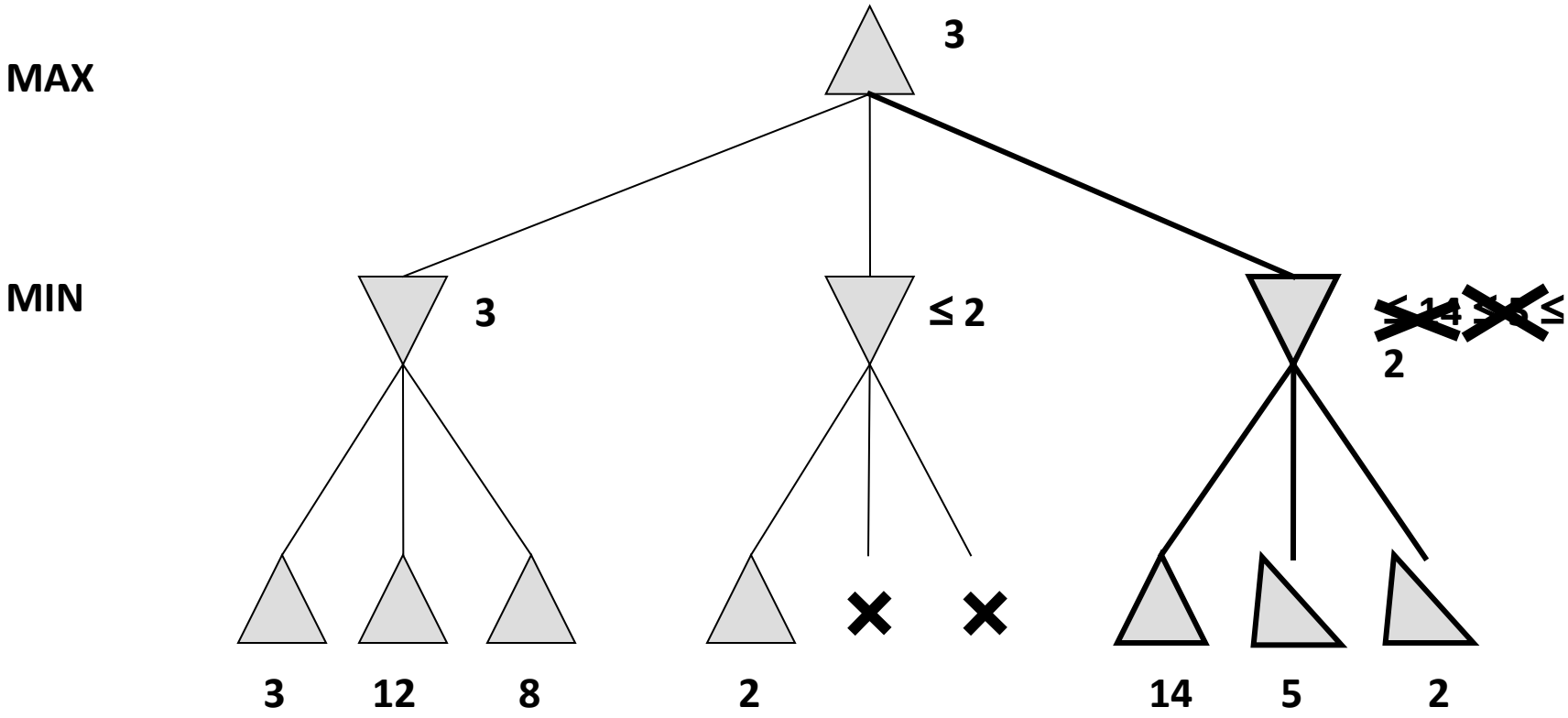
α-β pruning: example 2



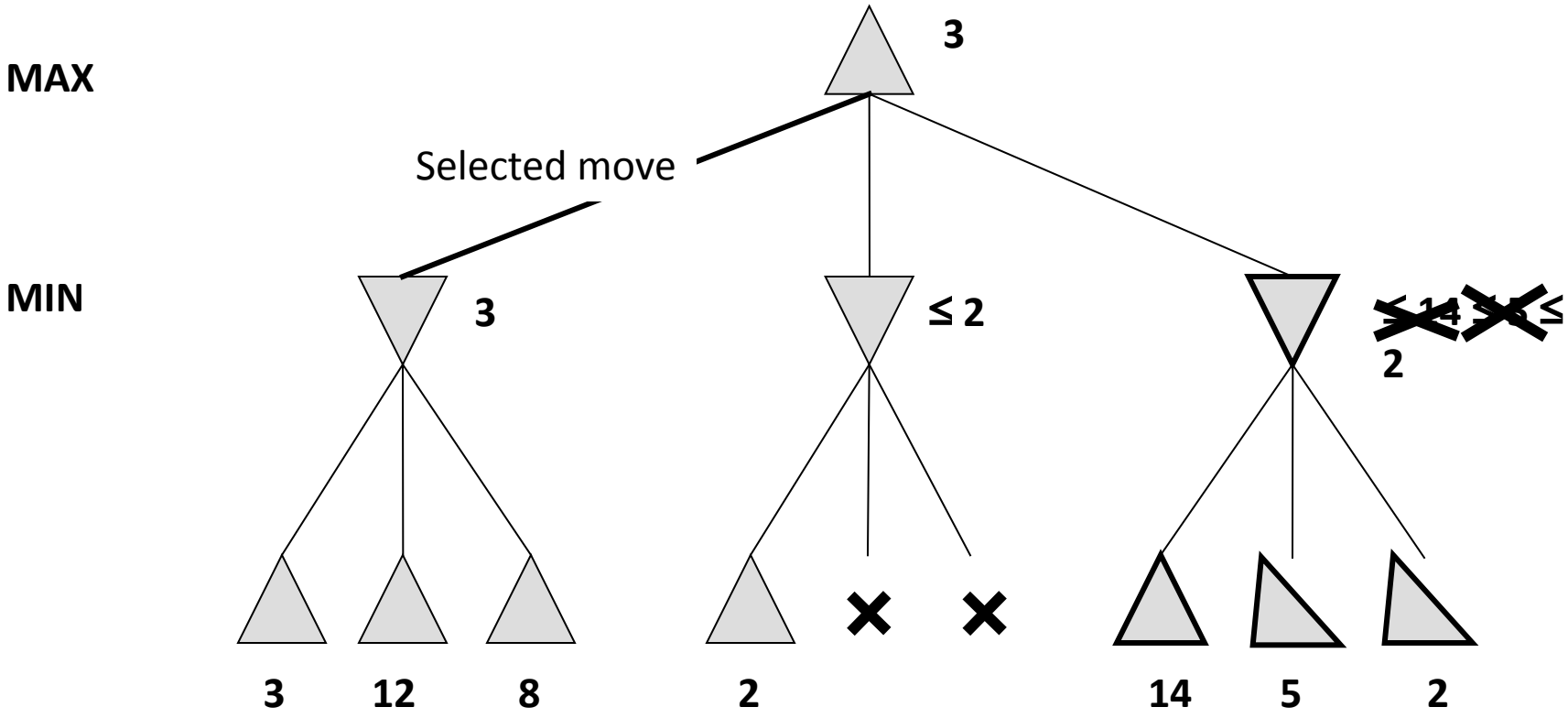
α-β pruning: example 2



α-β pruning: example 2



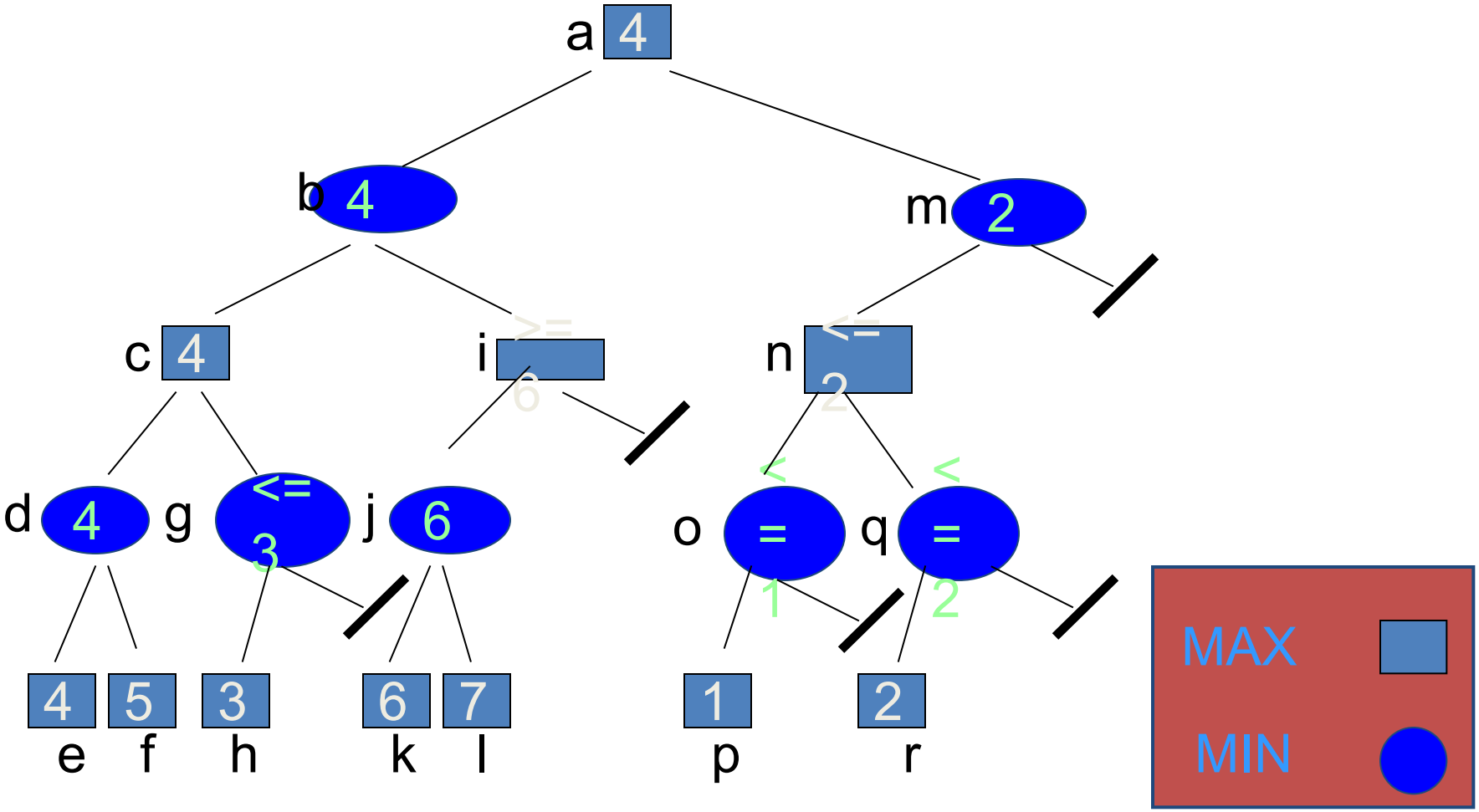
α-β pruning: example 2



Alpha-Beta Pruning Example

α Pruning node>= β

β Pruning node<= α



```
function ALPHA-BETA-DECISION(state) returns an action
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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
  same as MAX-VALUE but with roles of α, β reversed
```

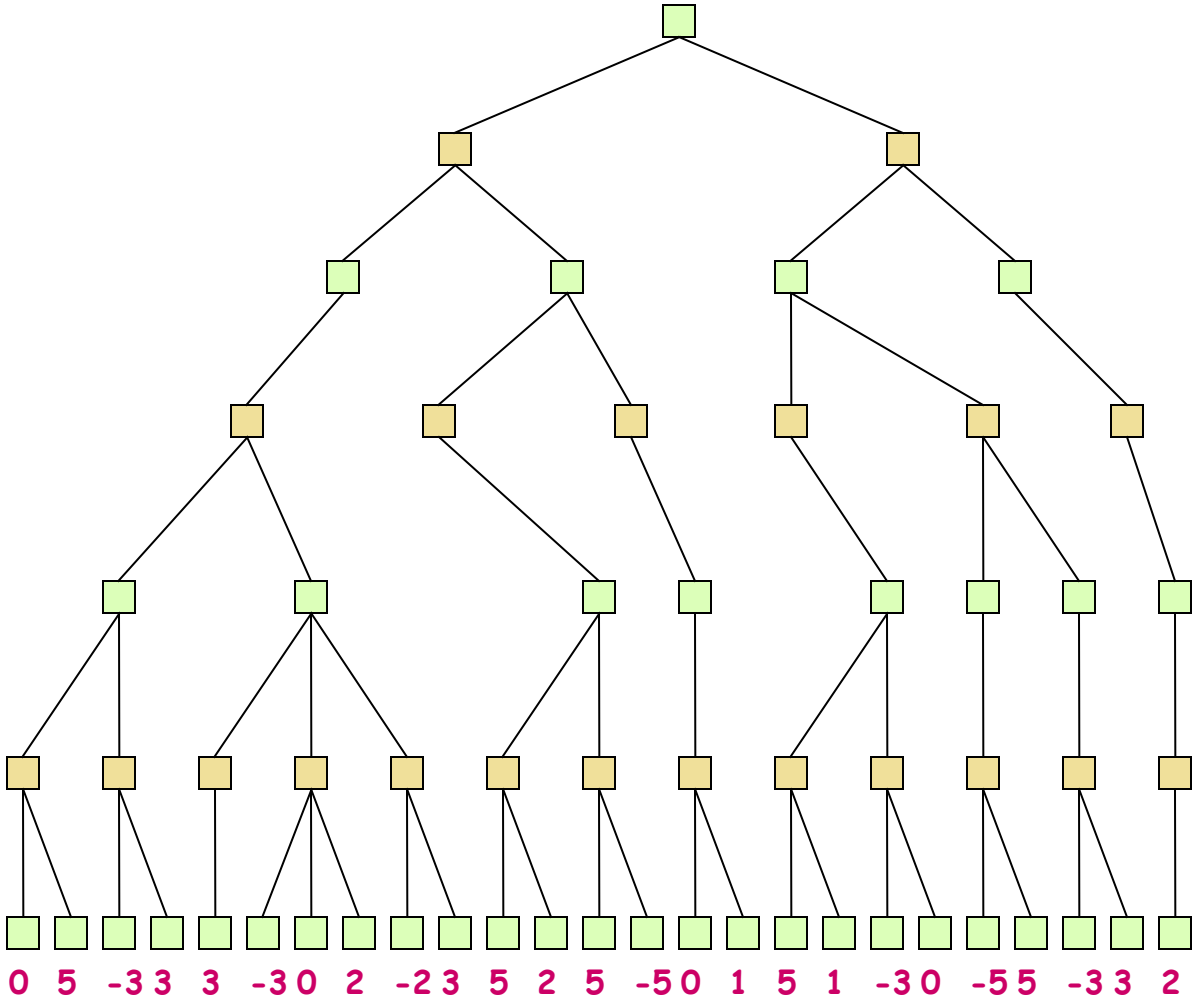
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



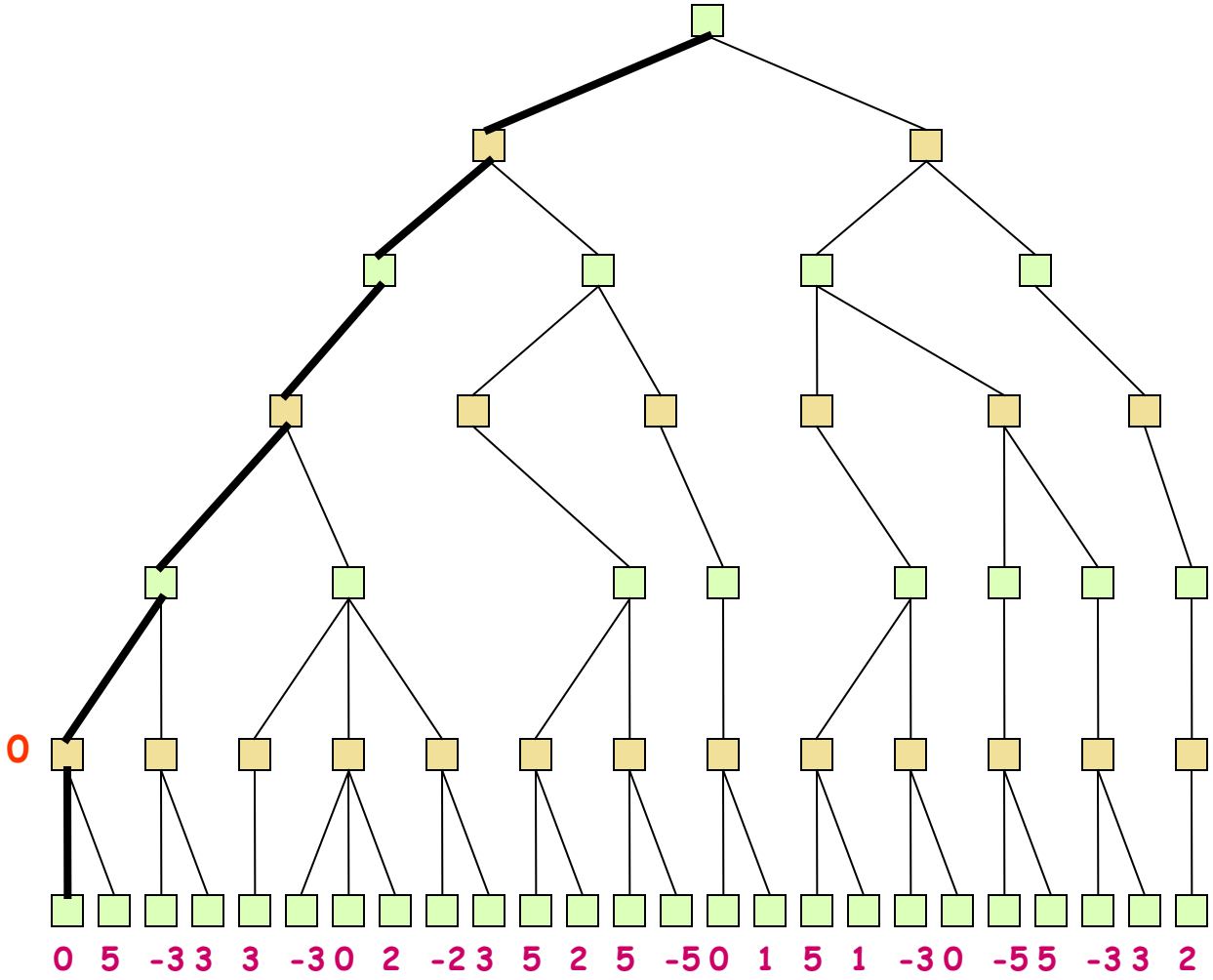
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



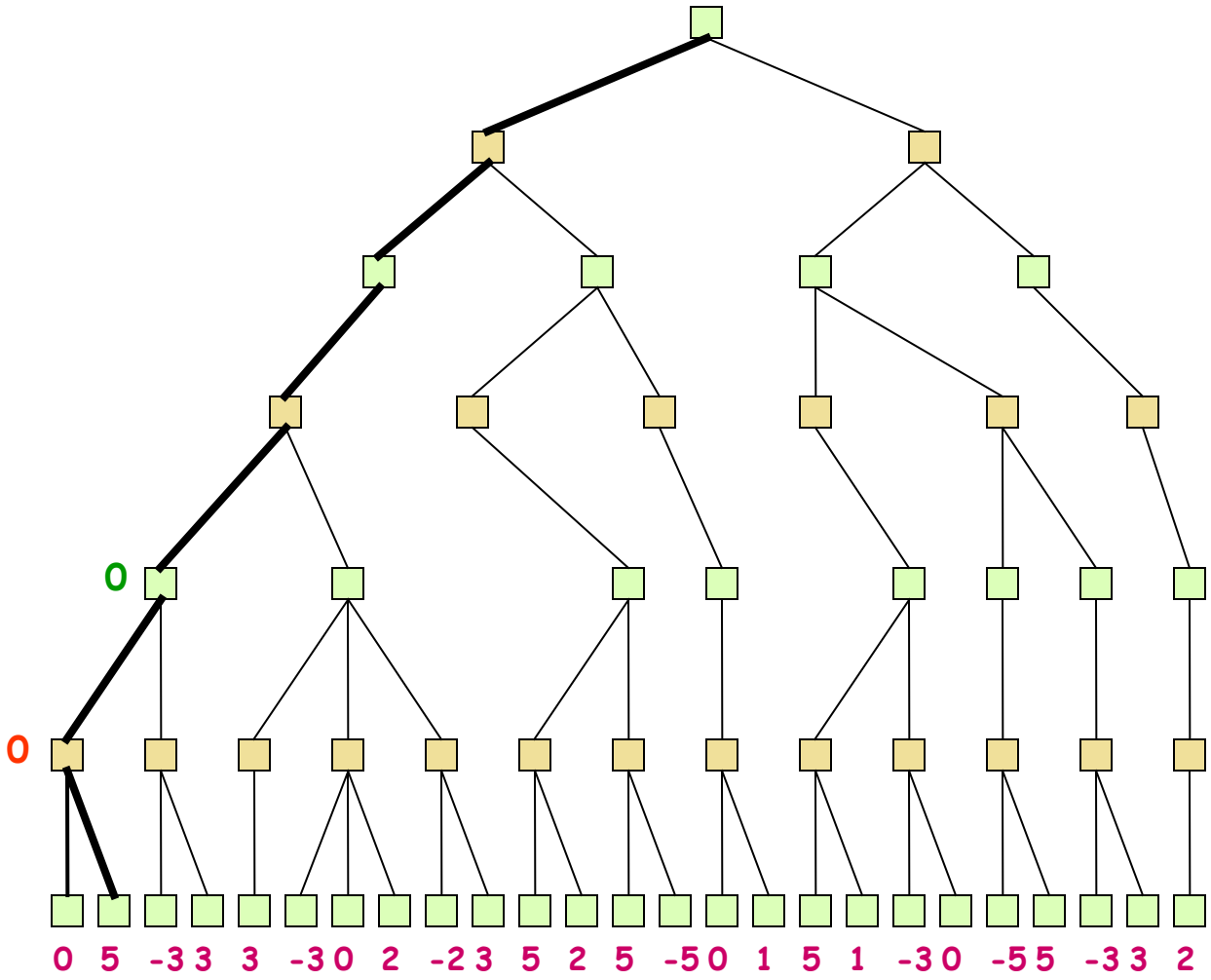
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



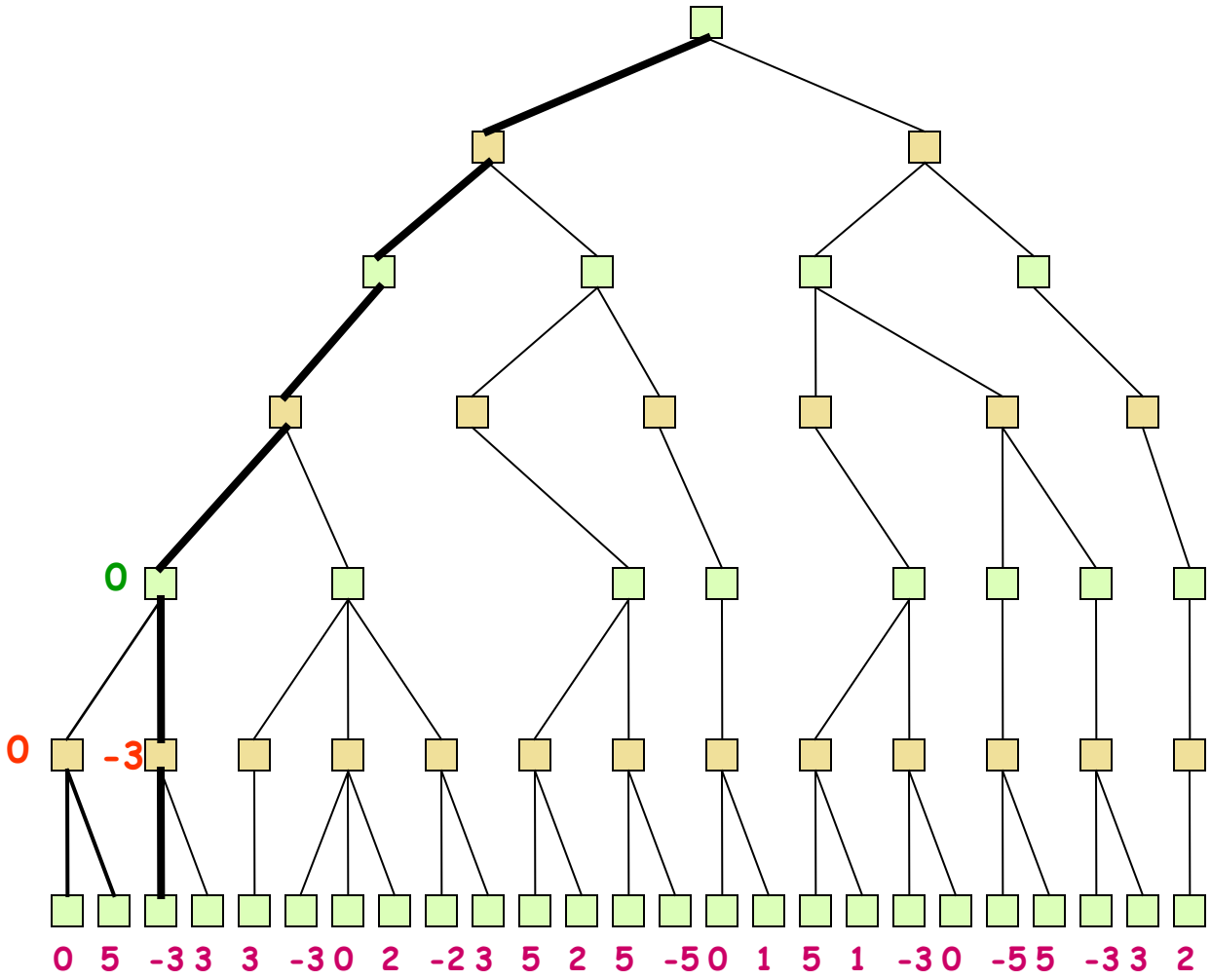
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



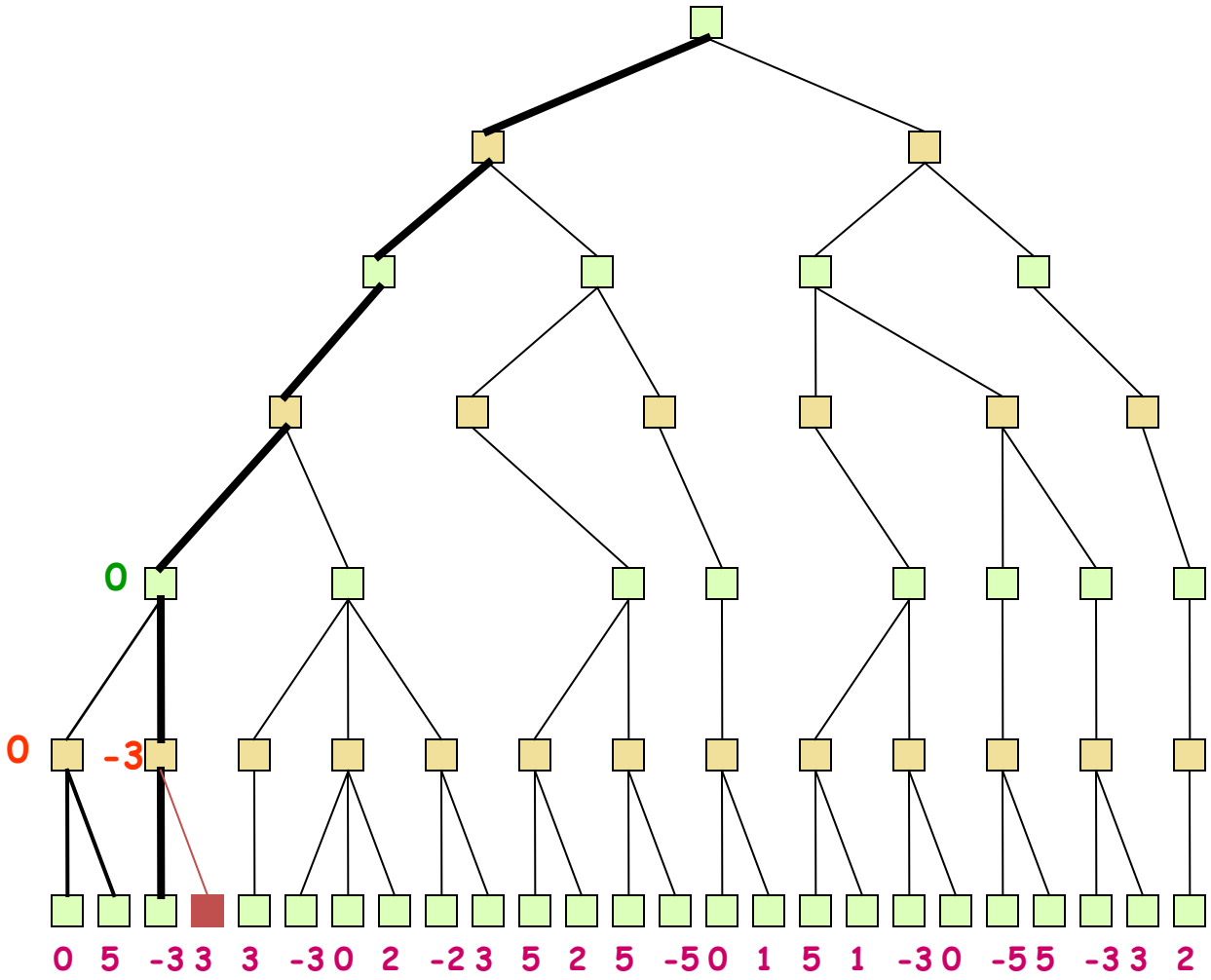
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



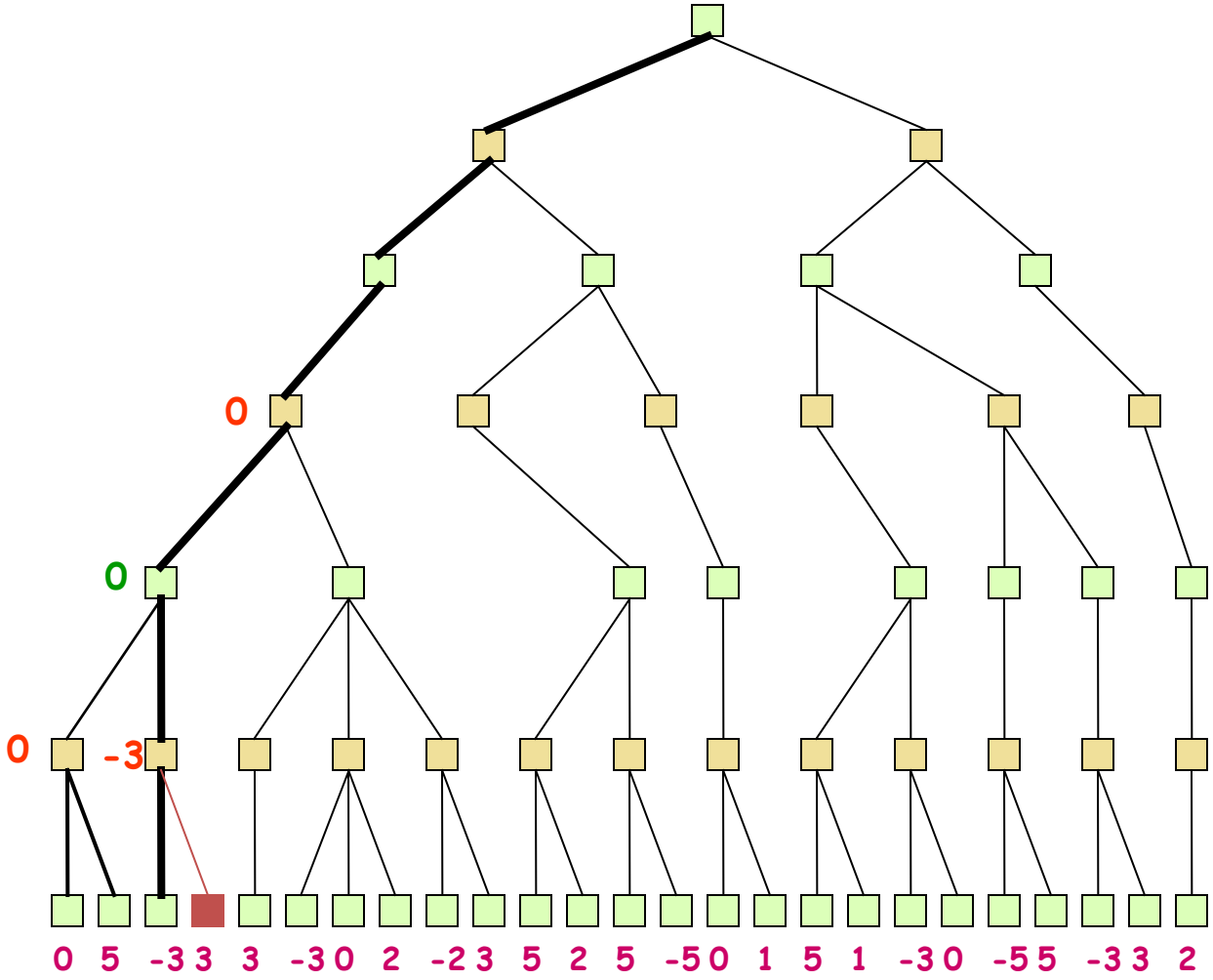
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



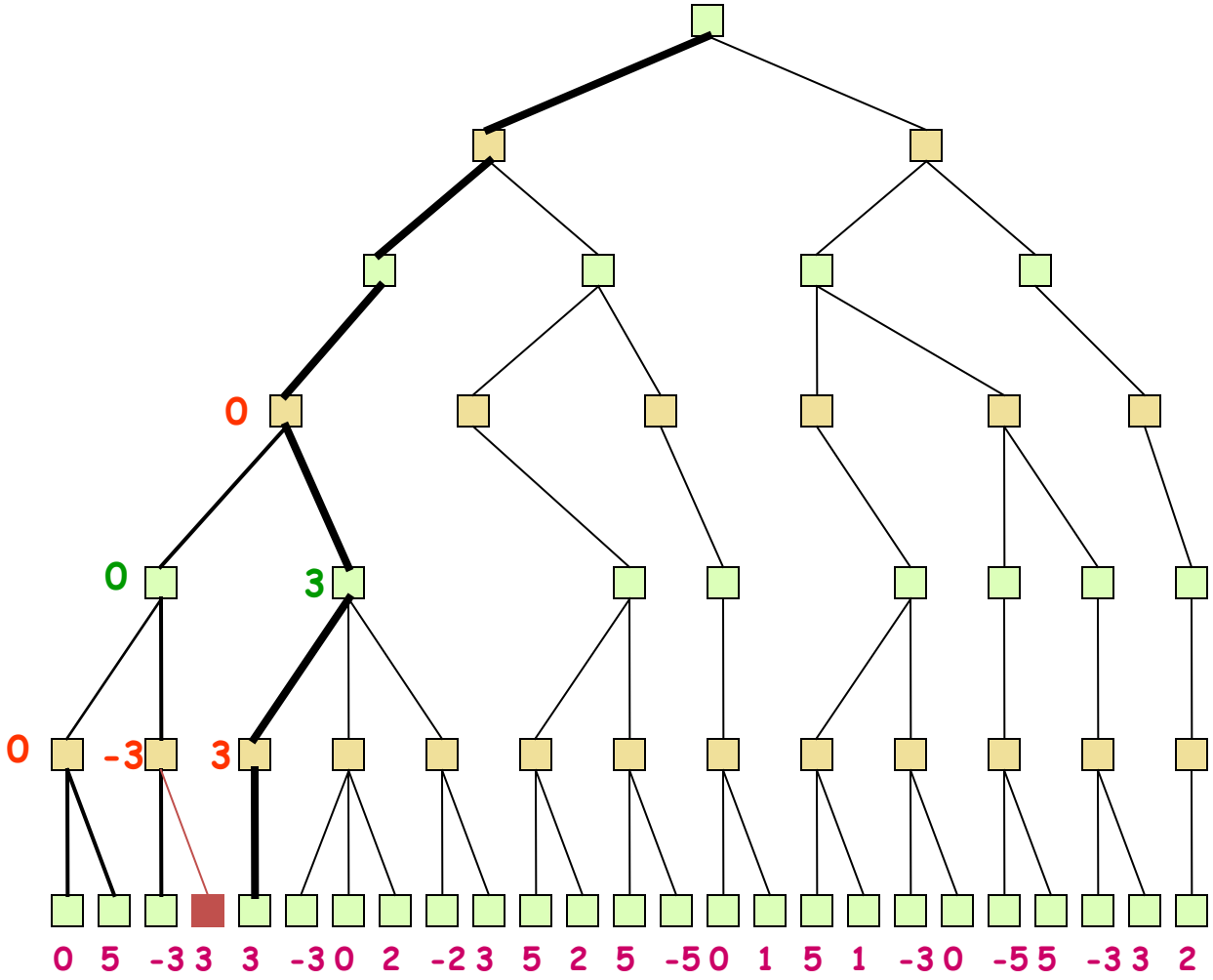
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



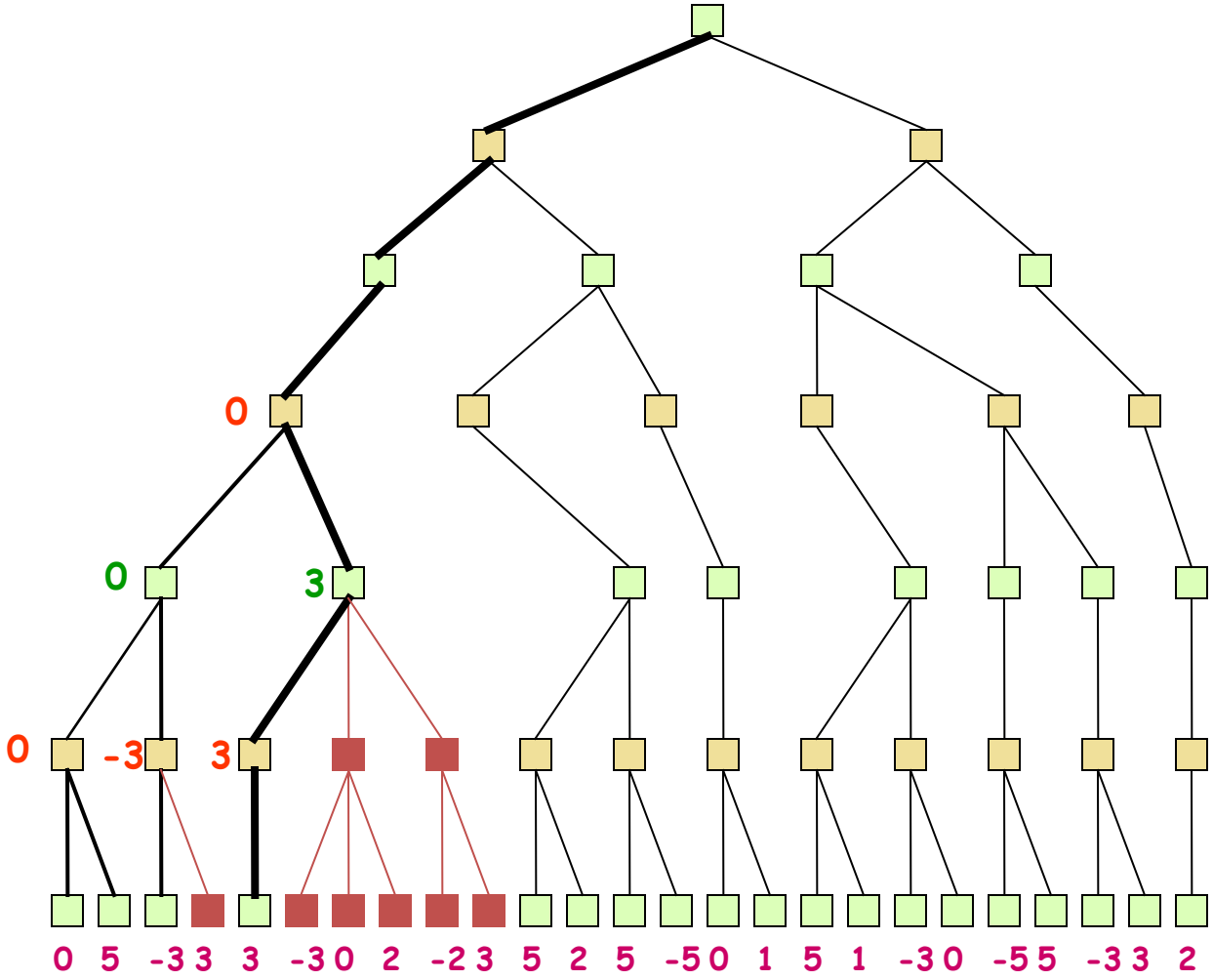
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



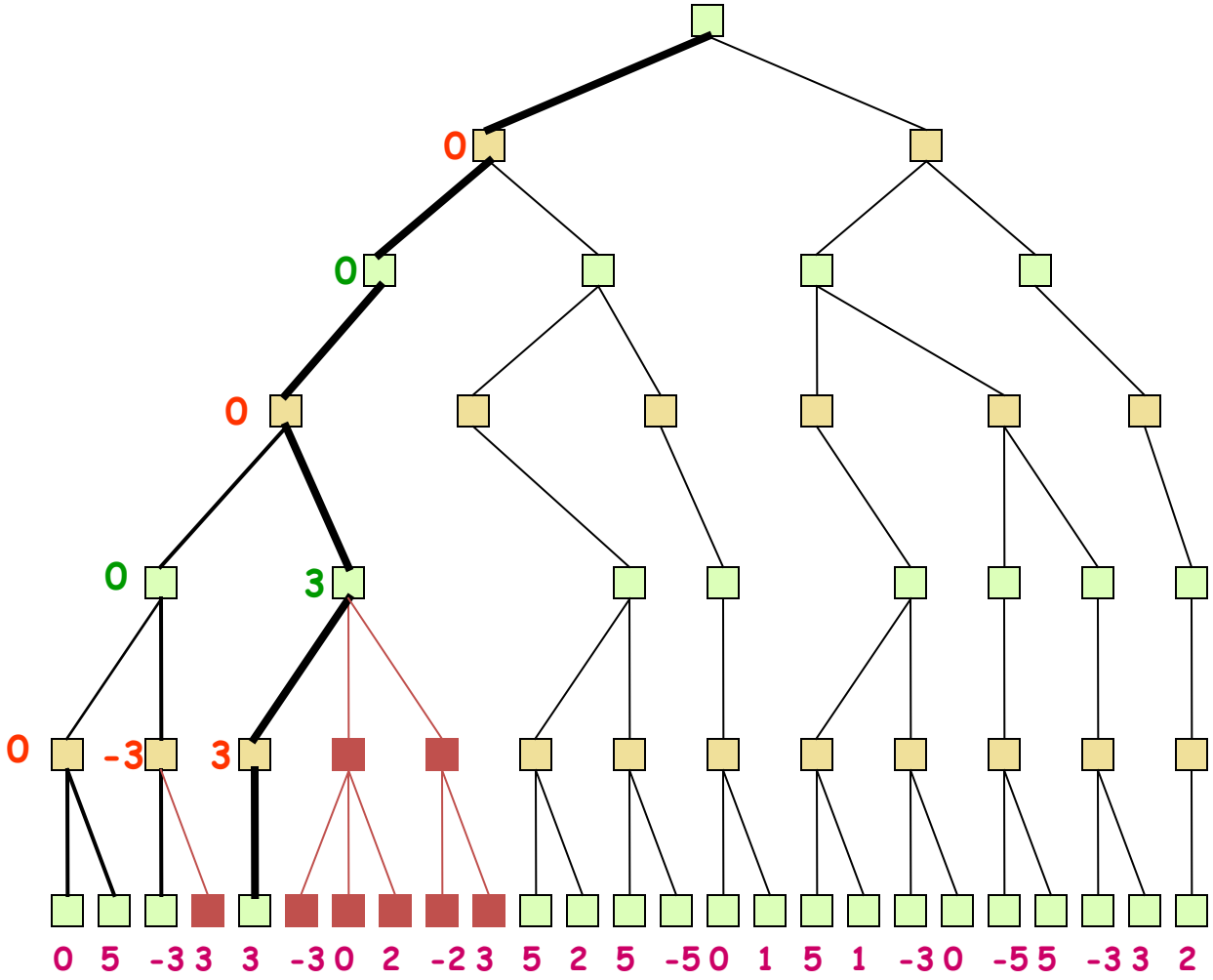
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



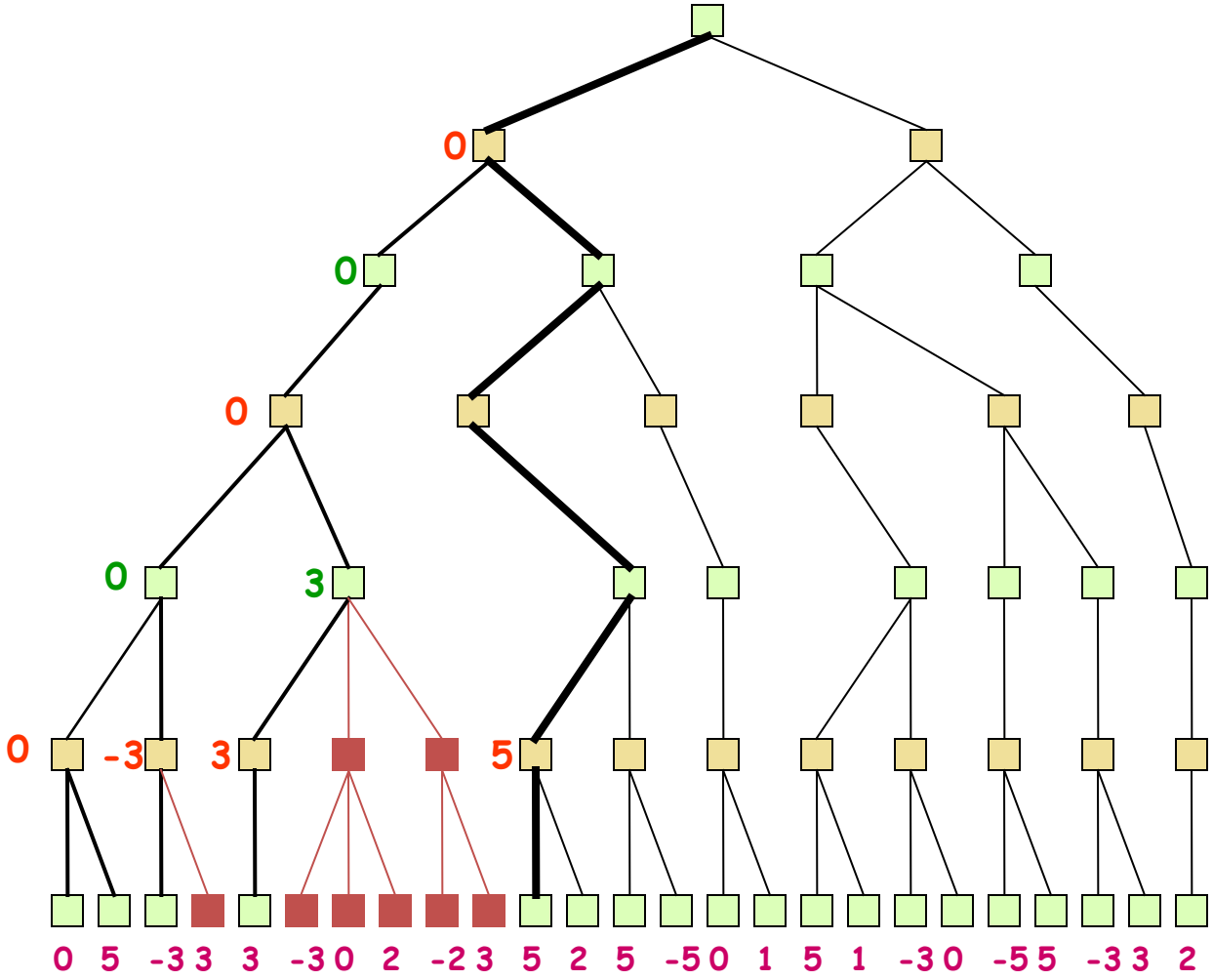
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



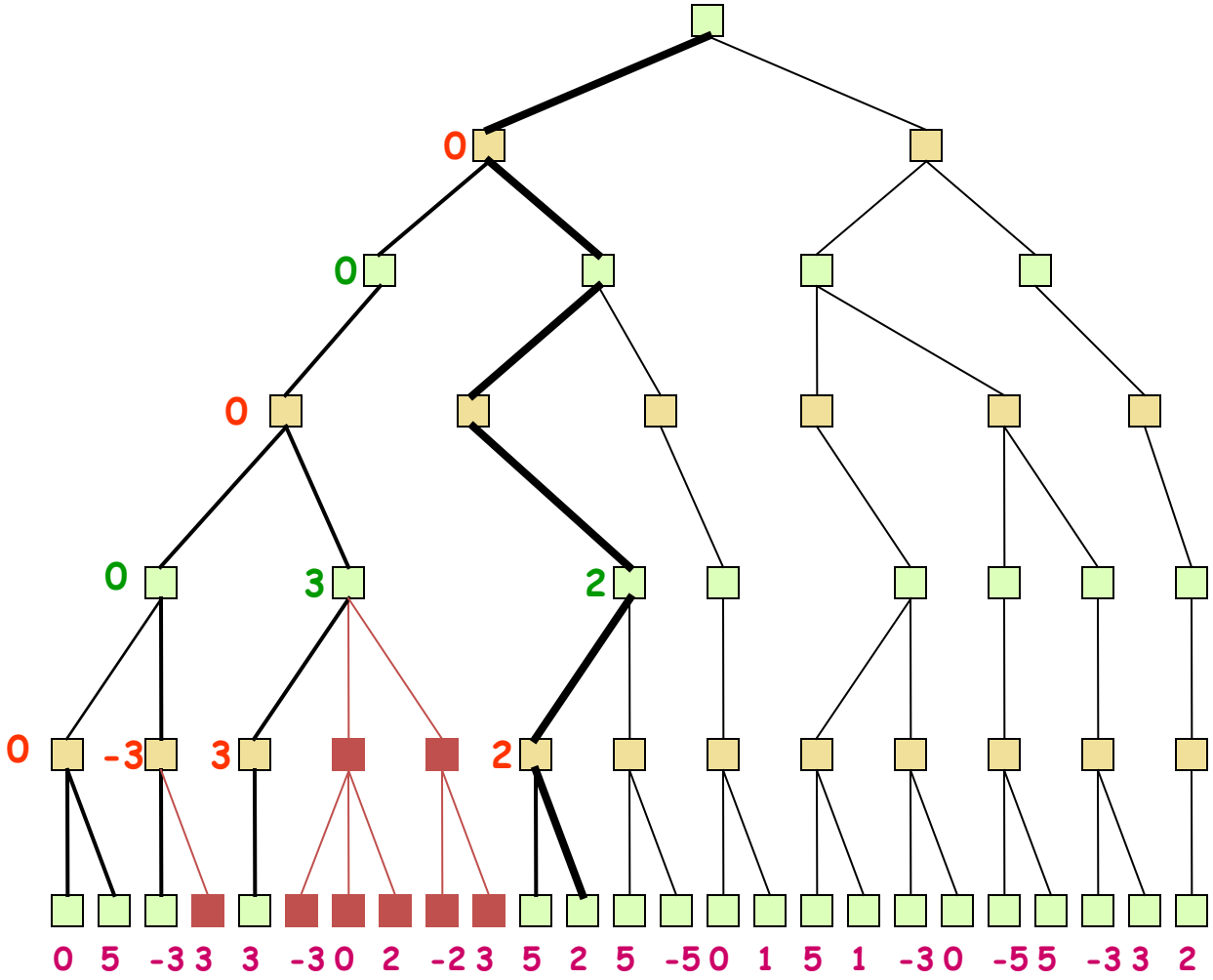
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



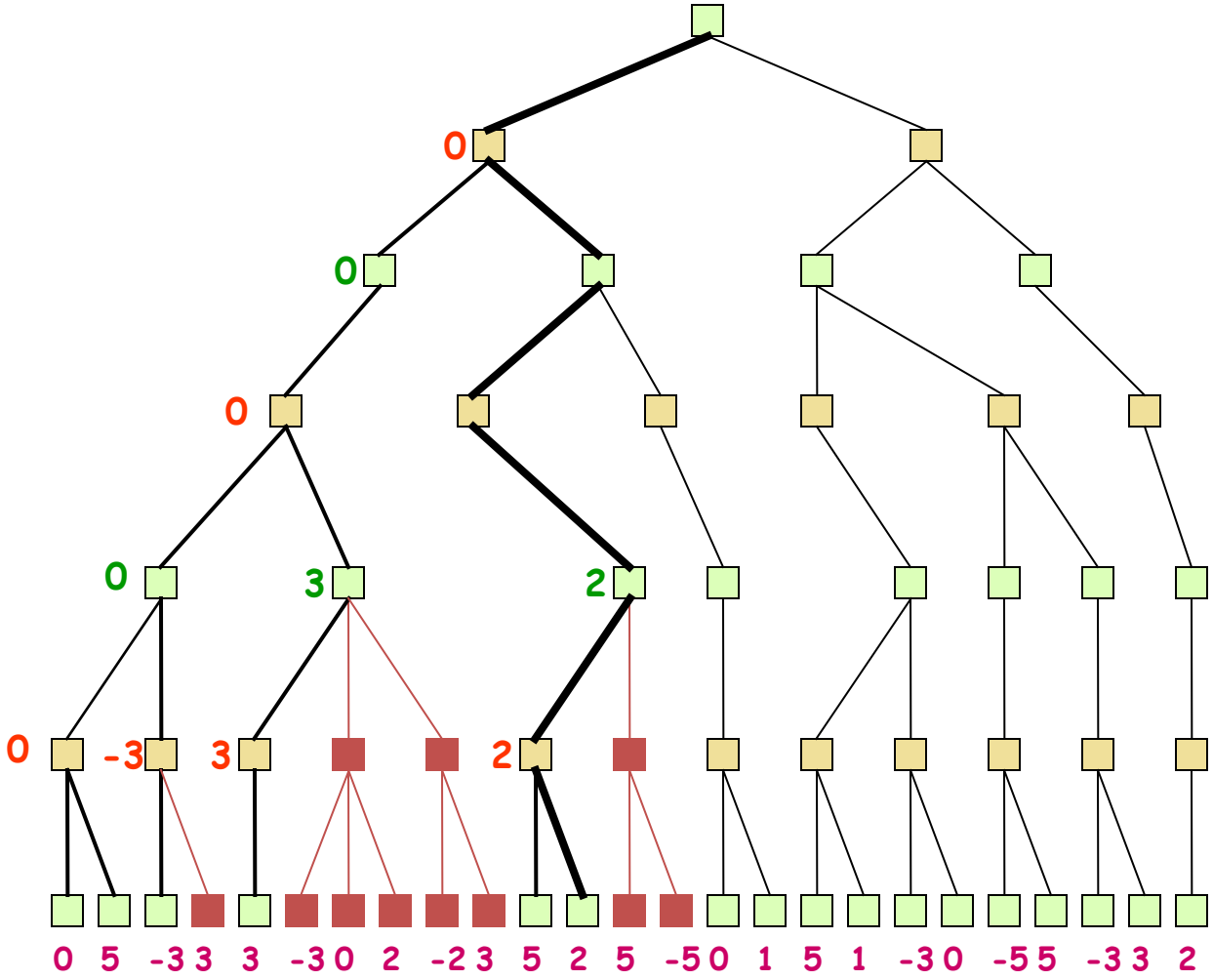
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



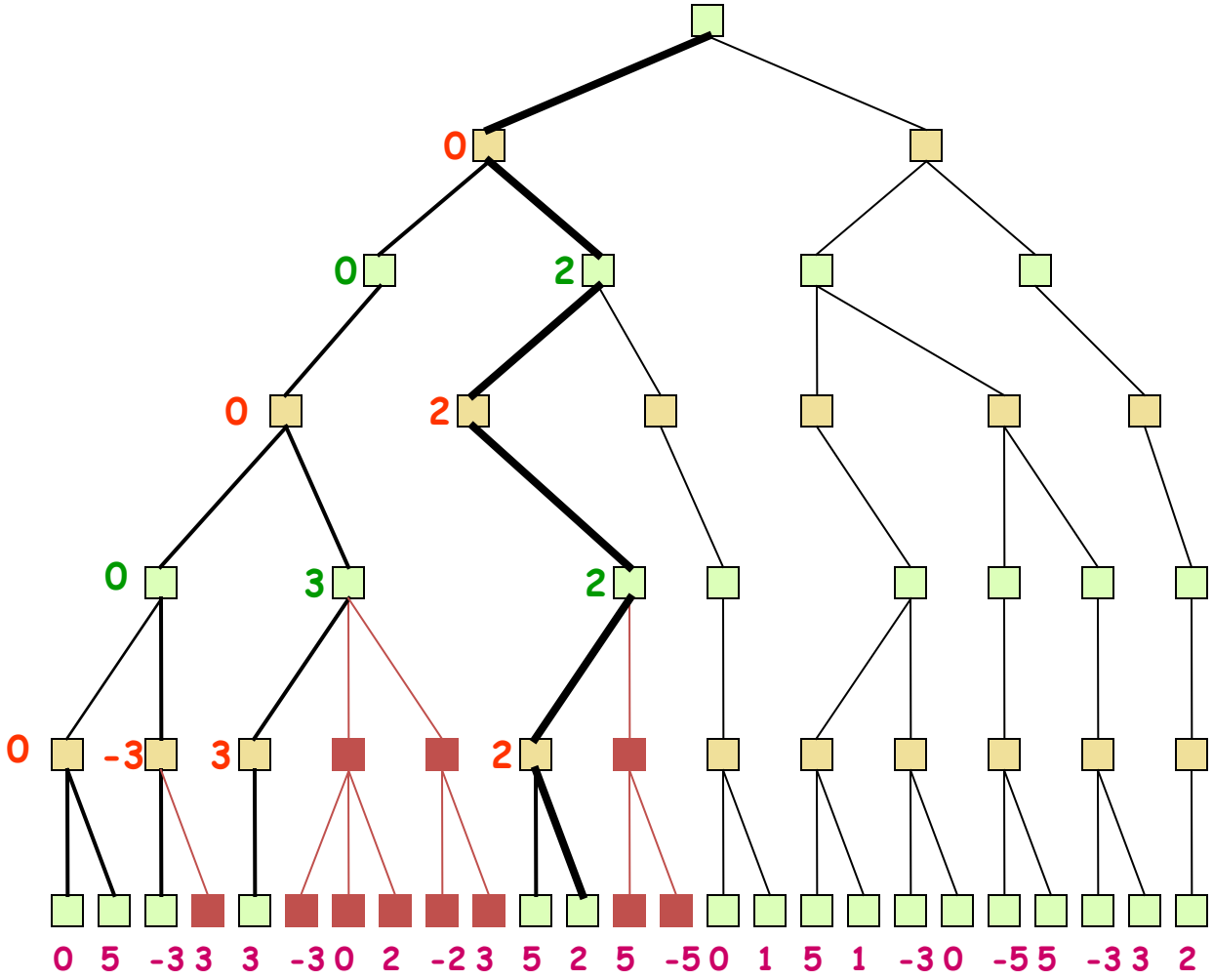
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



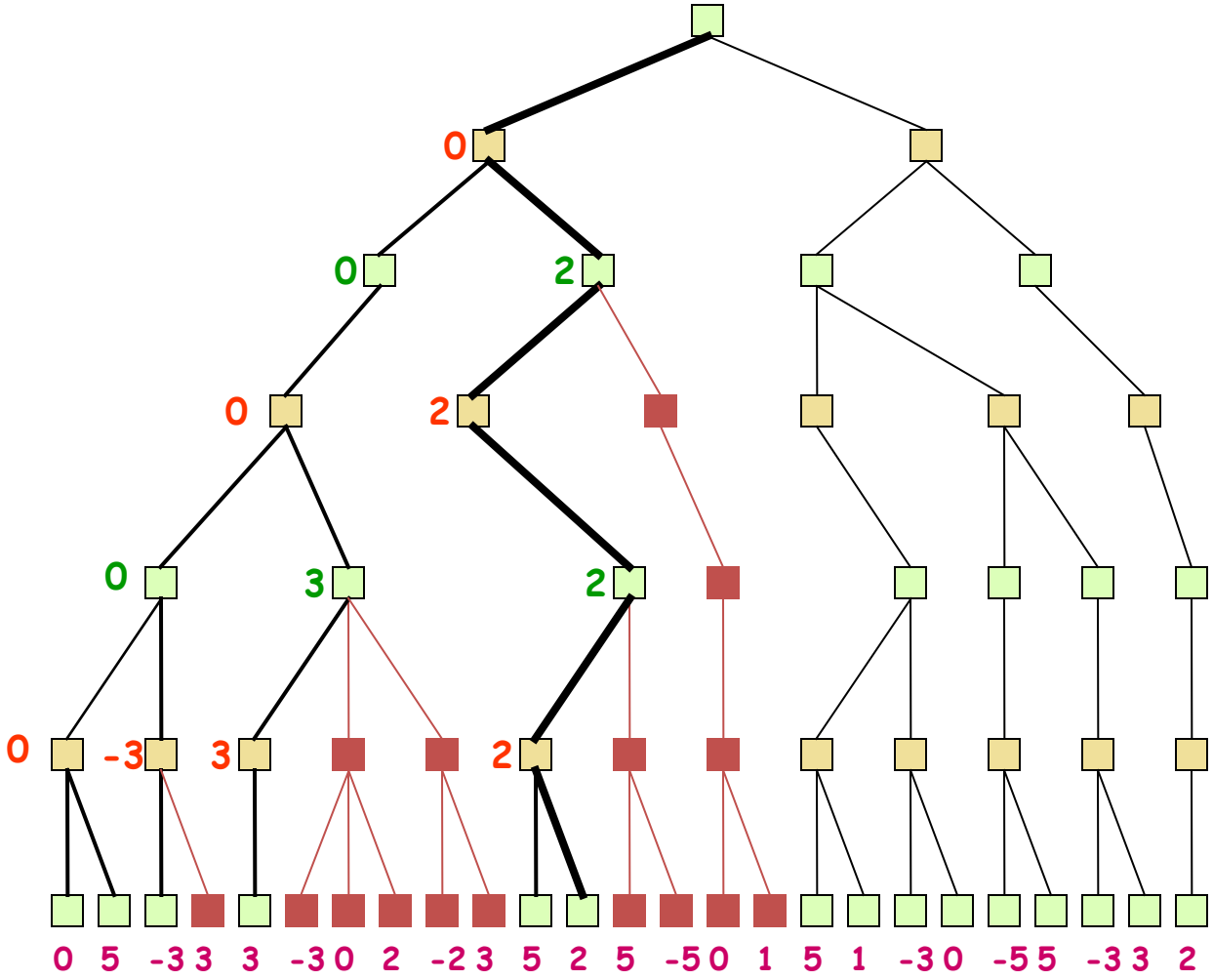
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



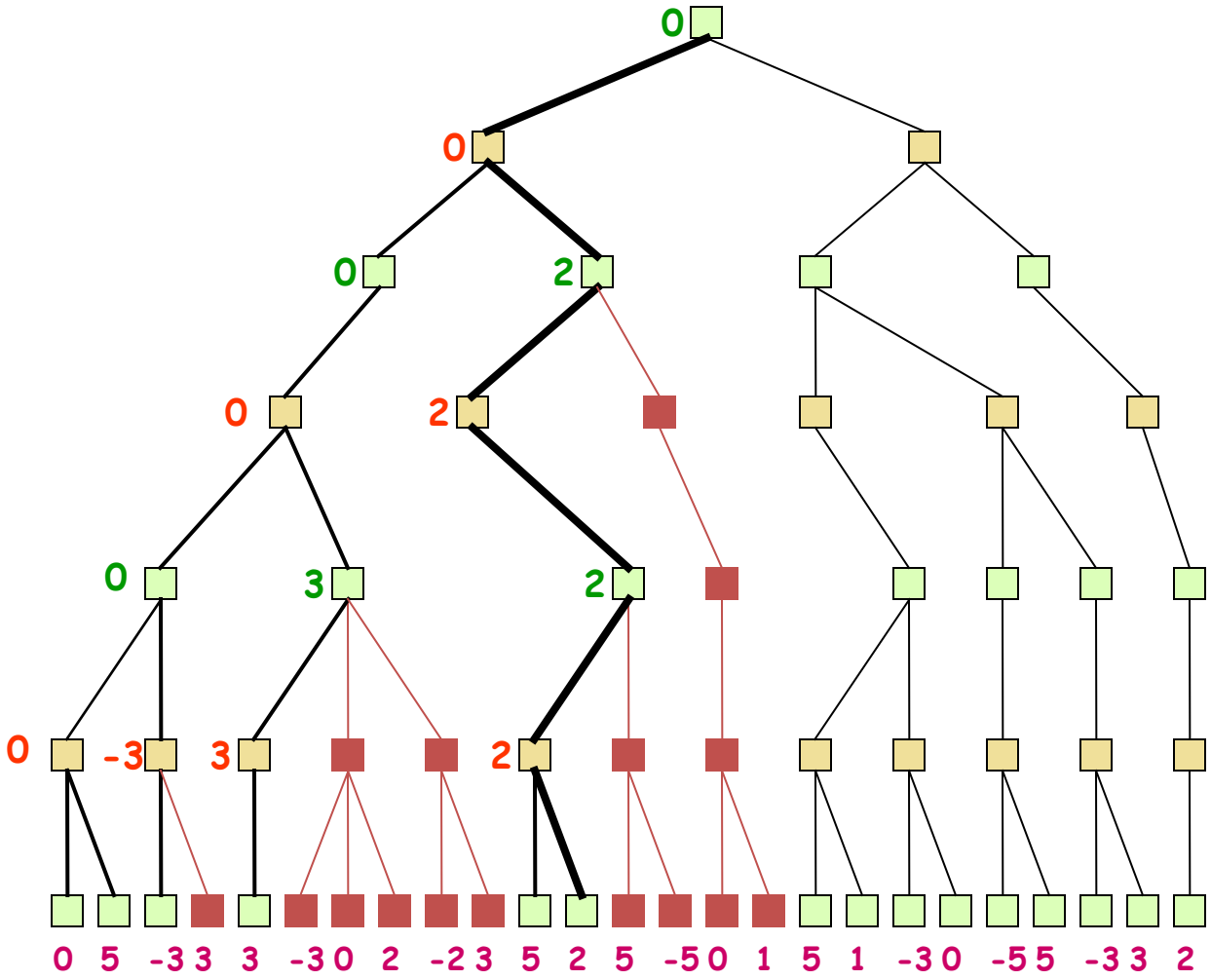
α-β pruning: example 3

α *Pruning node>=* β

β *Pruning node<=* α

MAX

MIN



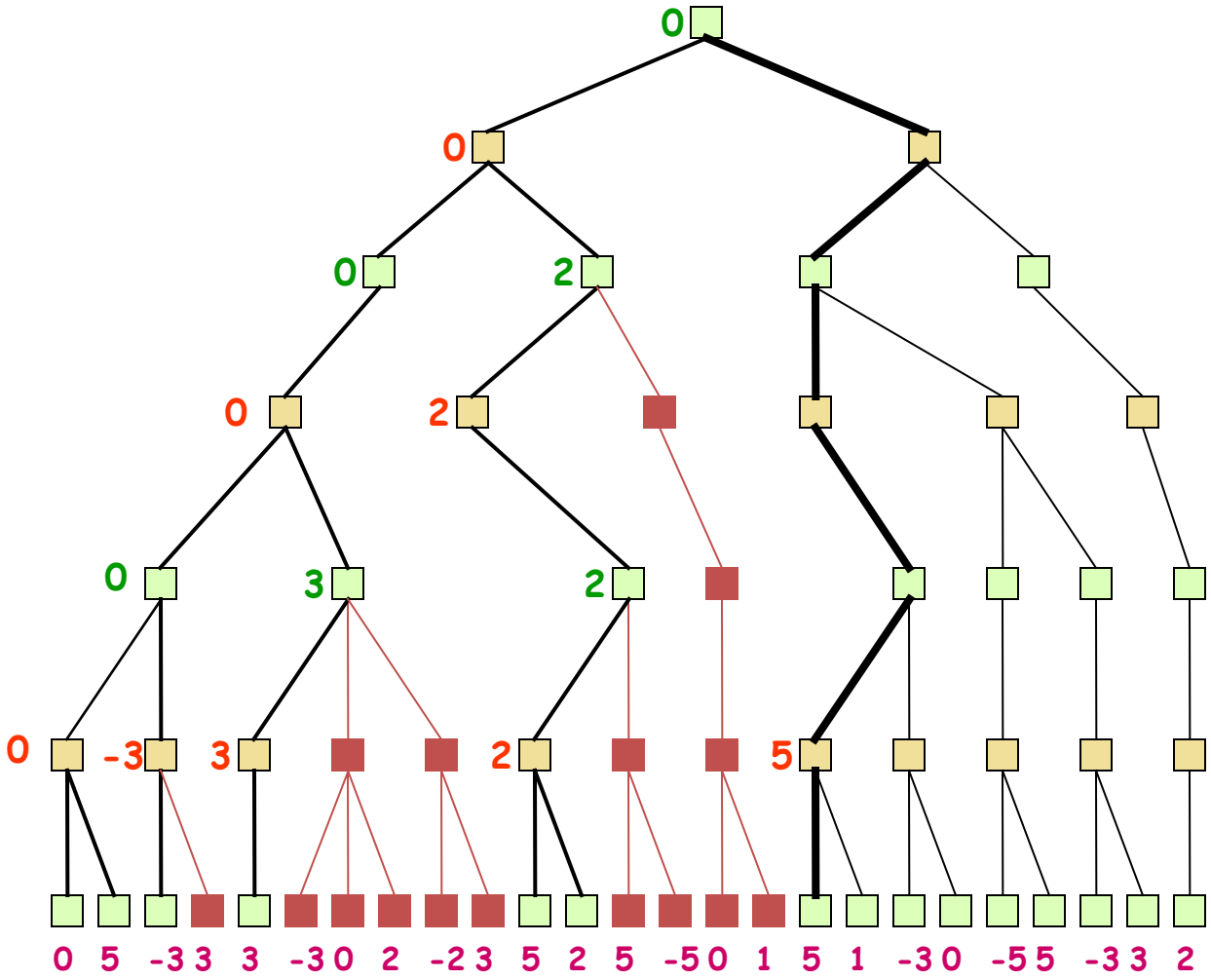
α-β pruning: example 3

α *Pruning* *node*>= β

β *Pruning* *node*<= α

MAX

MIN



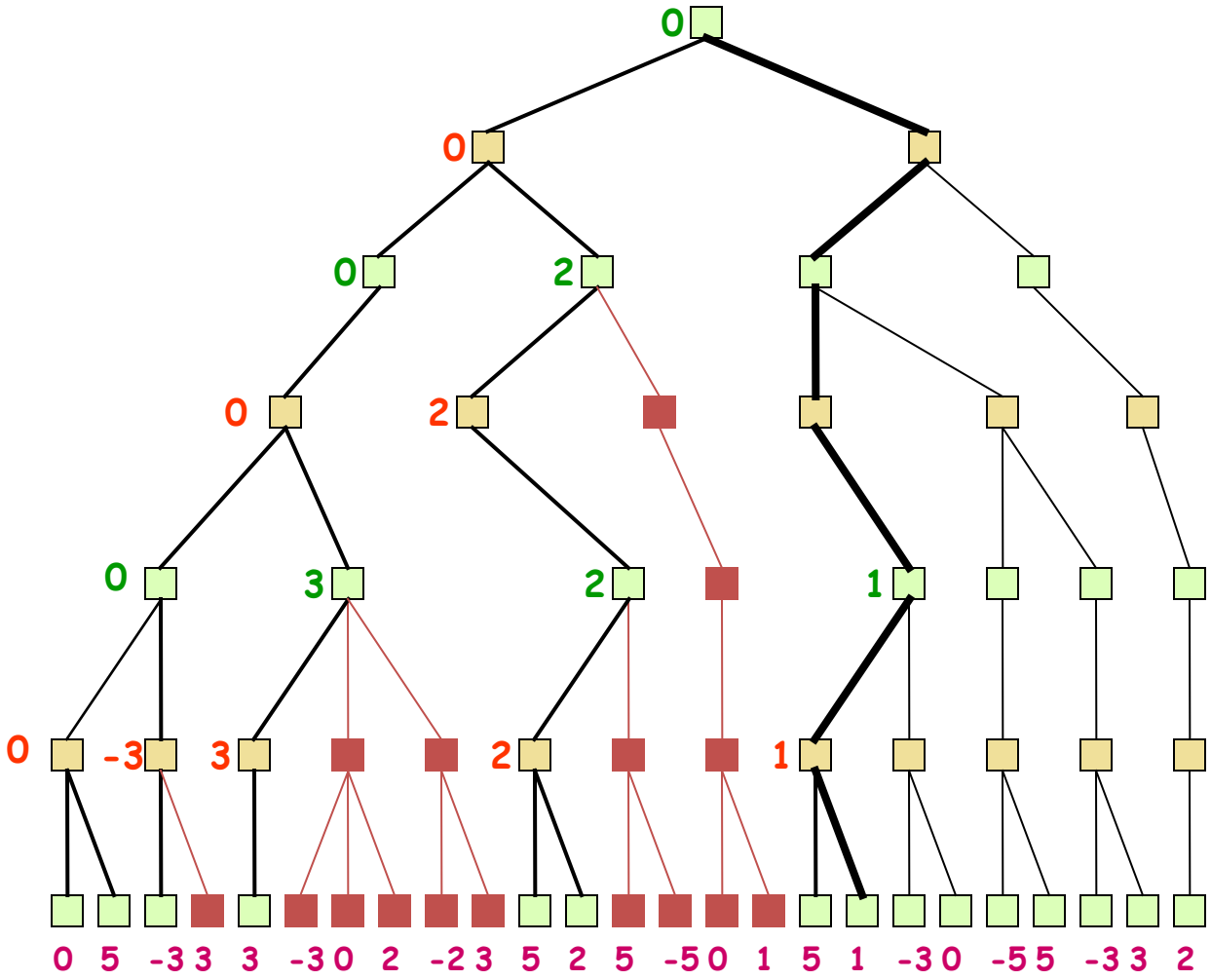
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



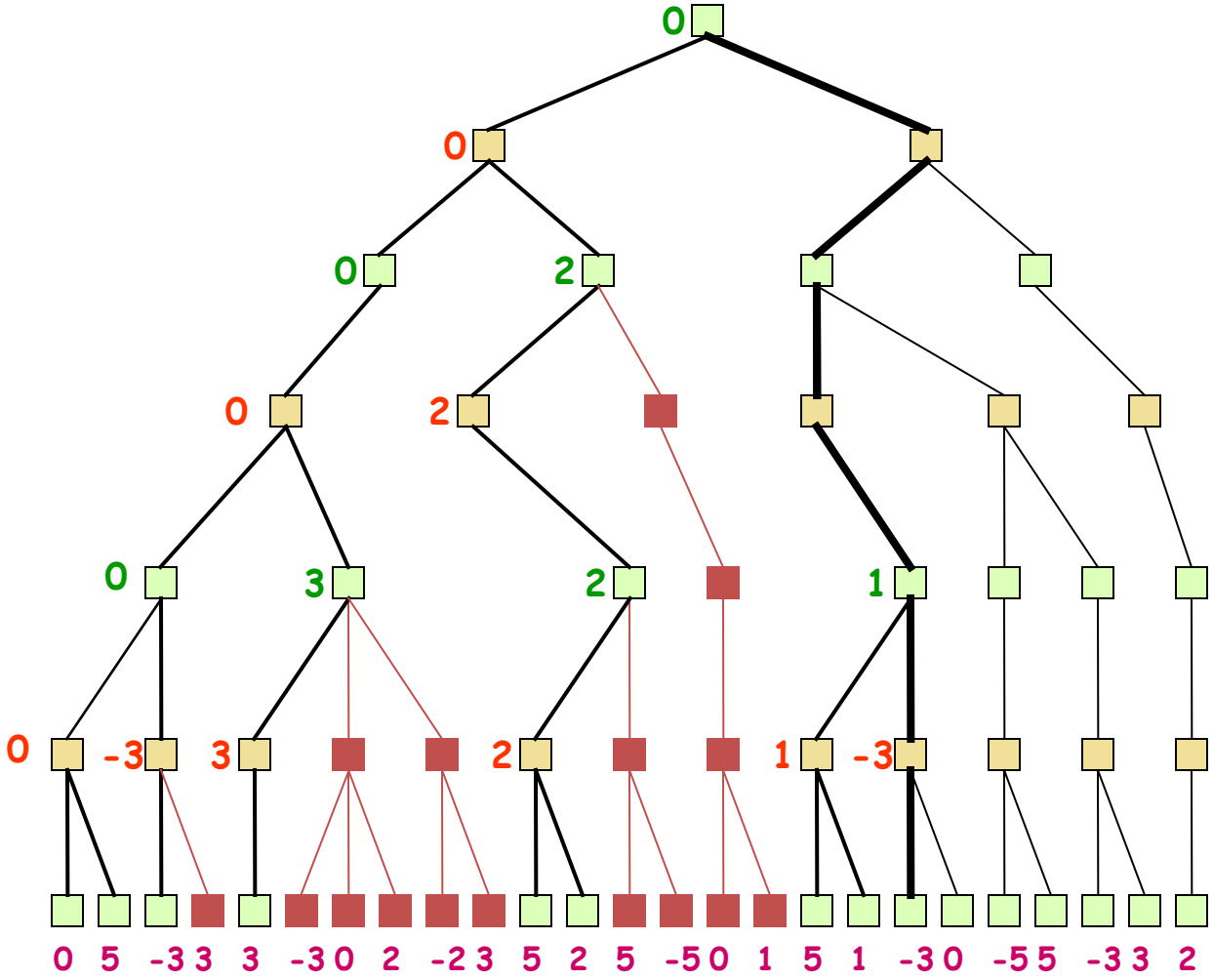
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



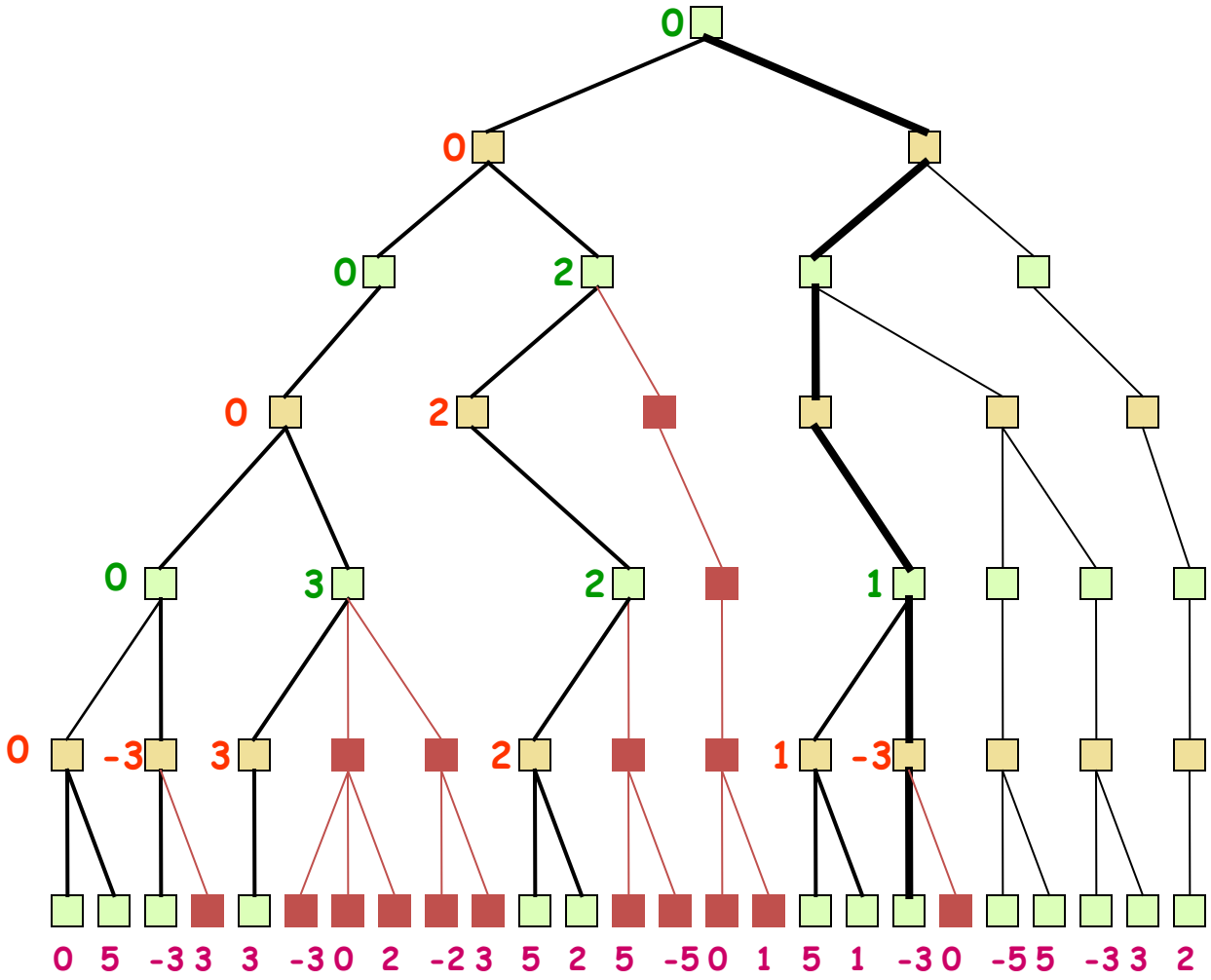
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



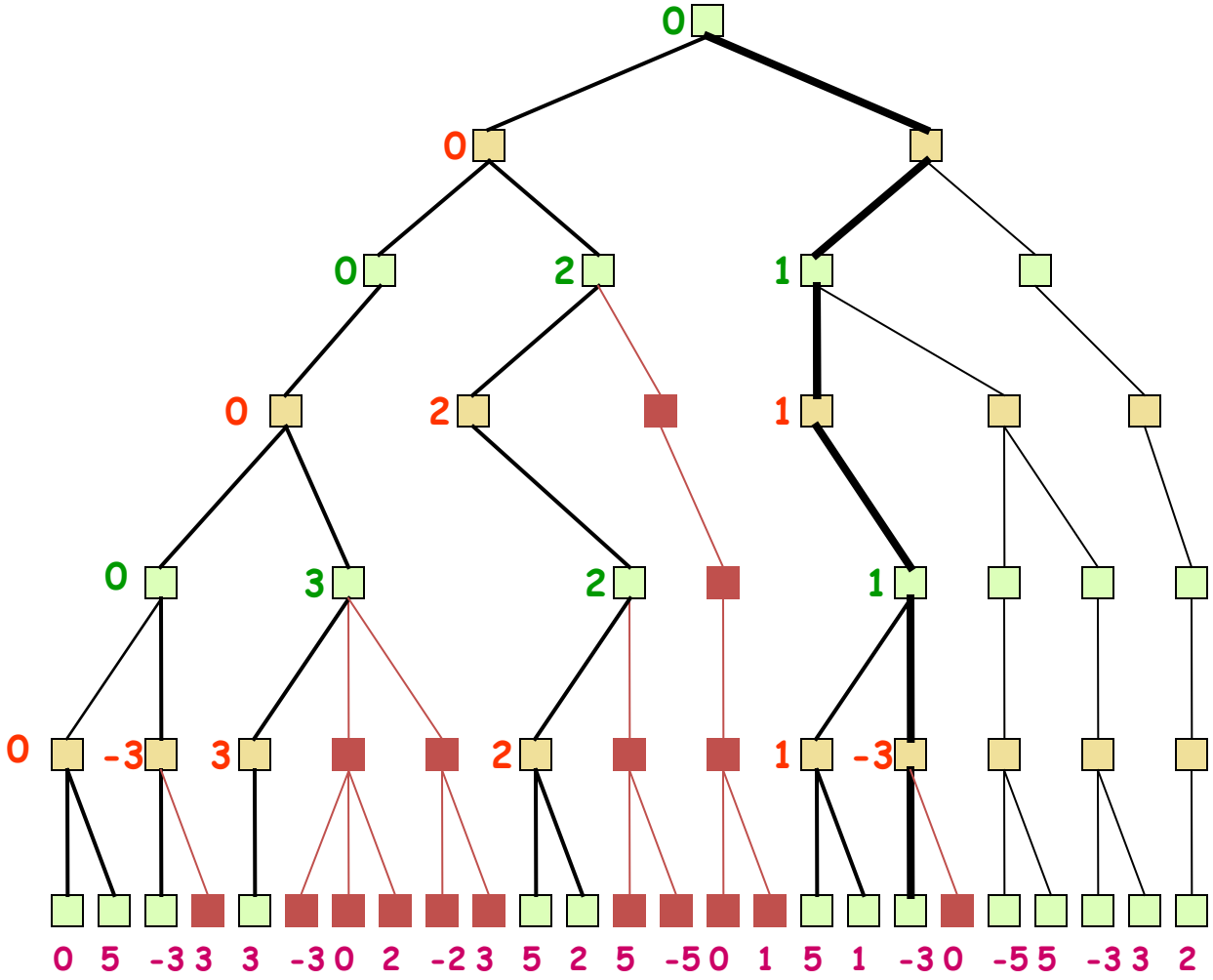
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



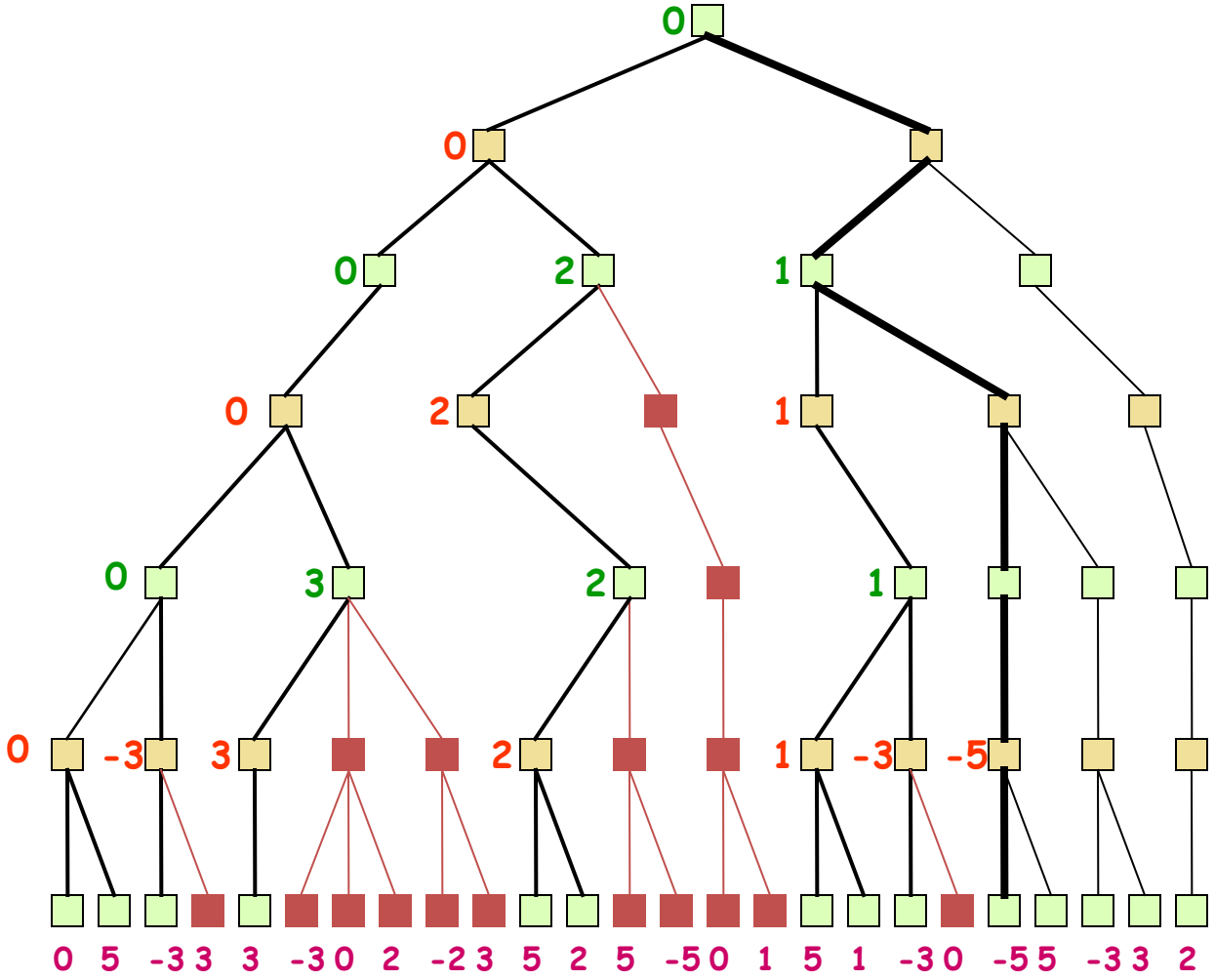
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



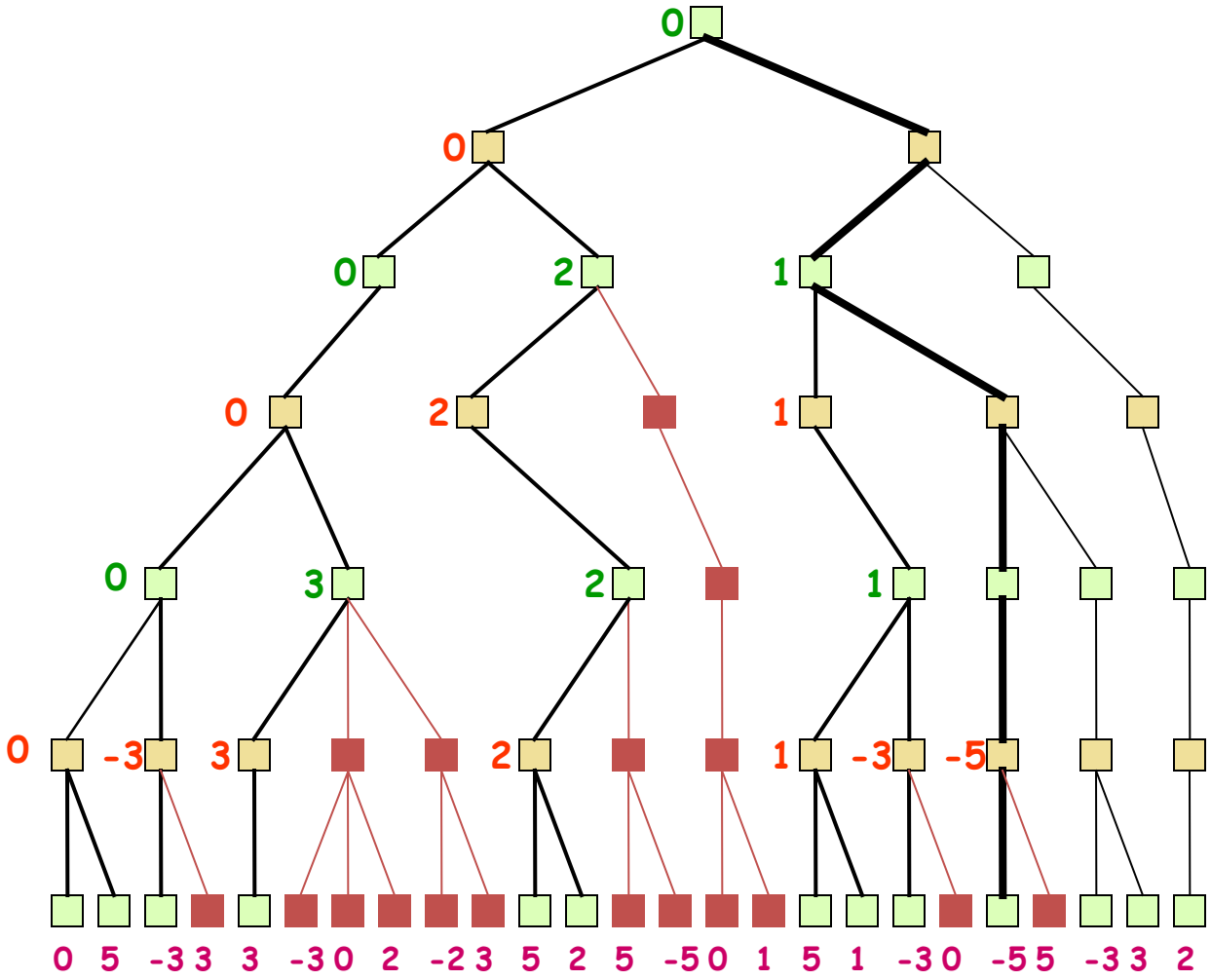
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



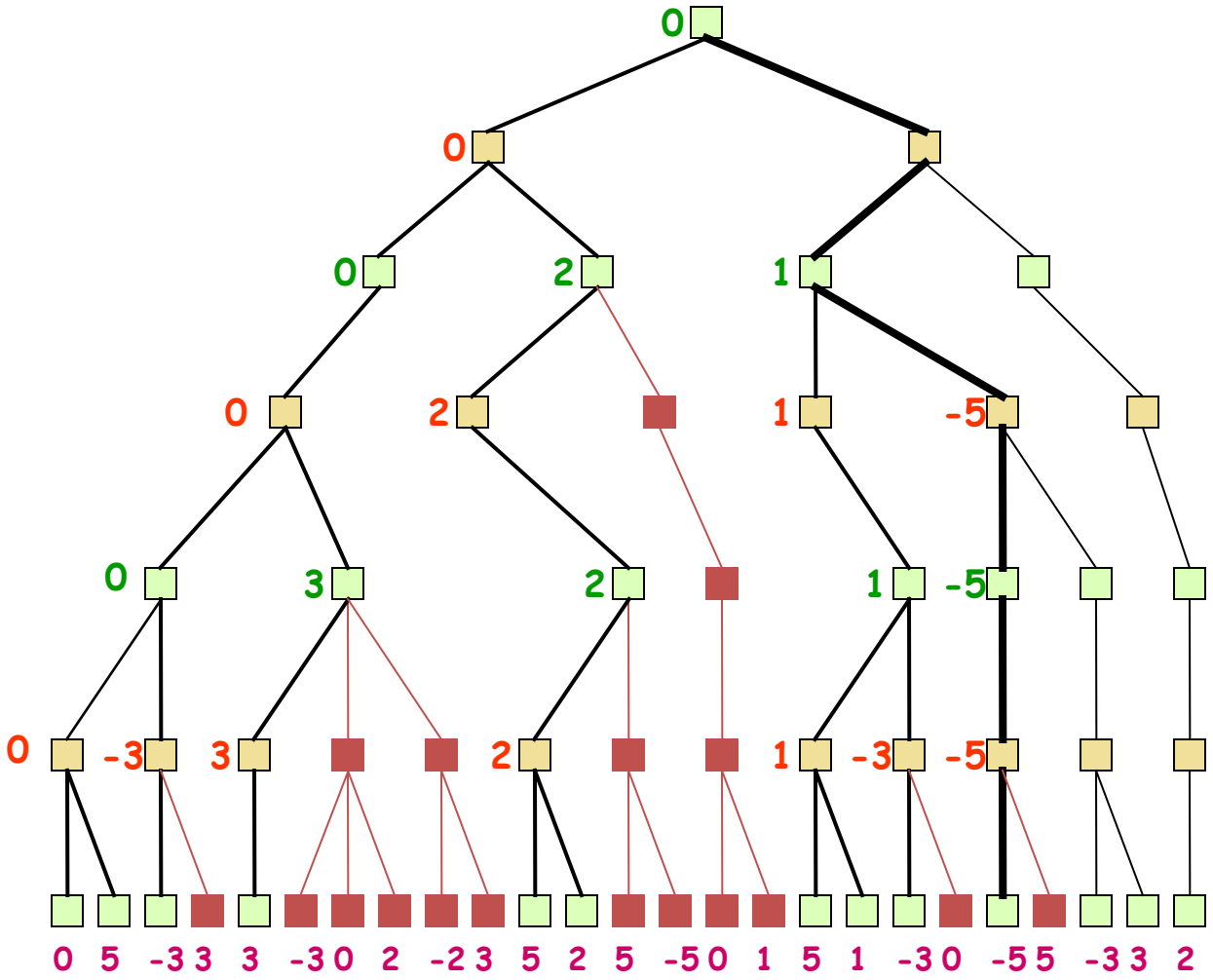
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



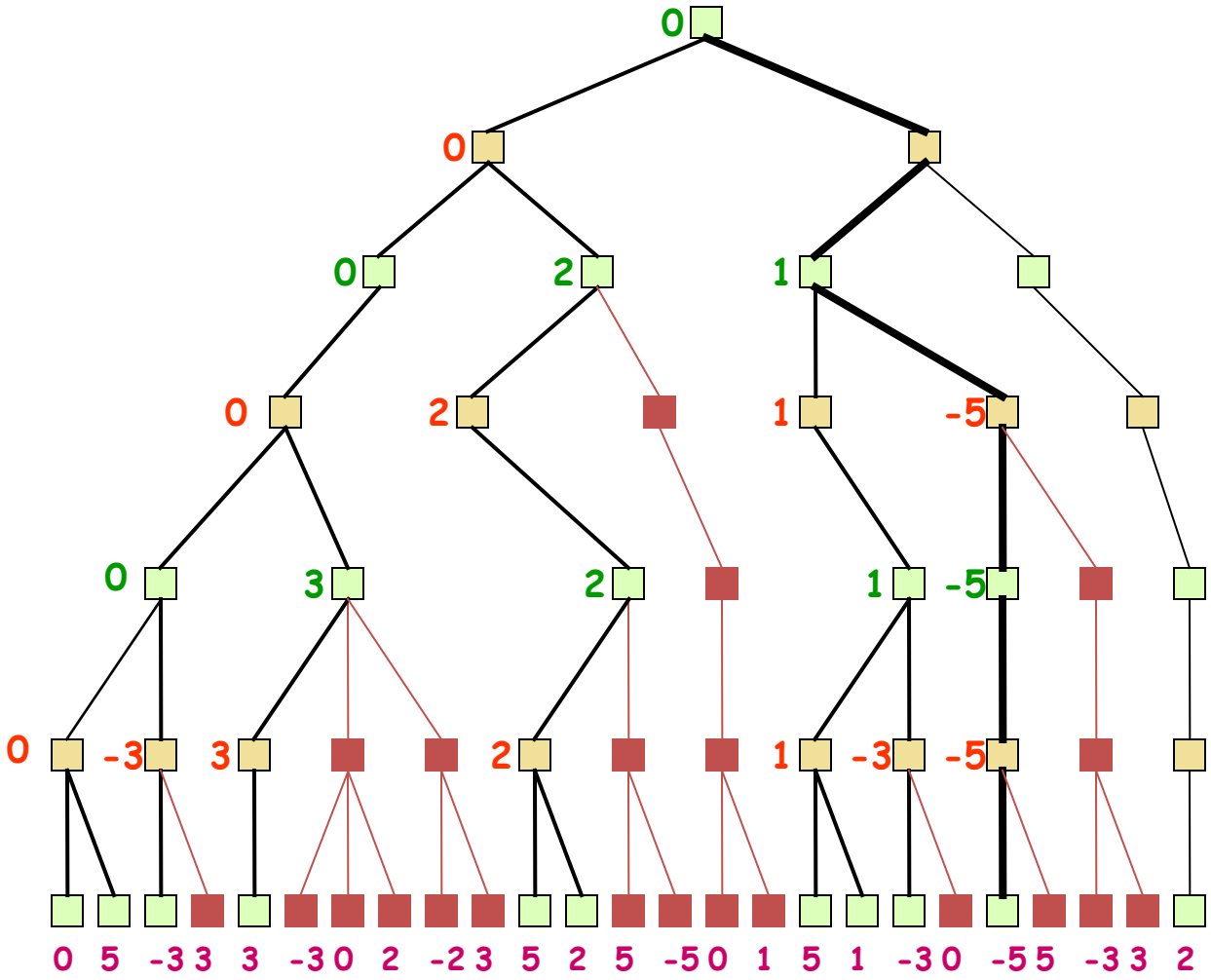
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



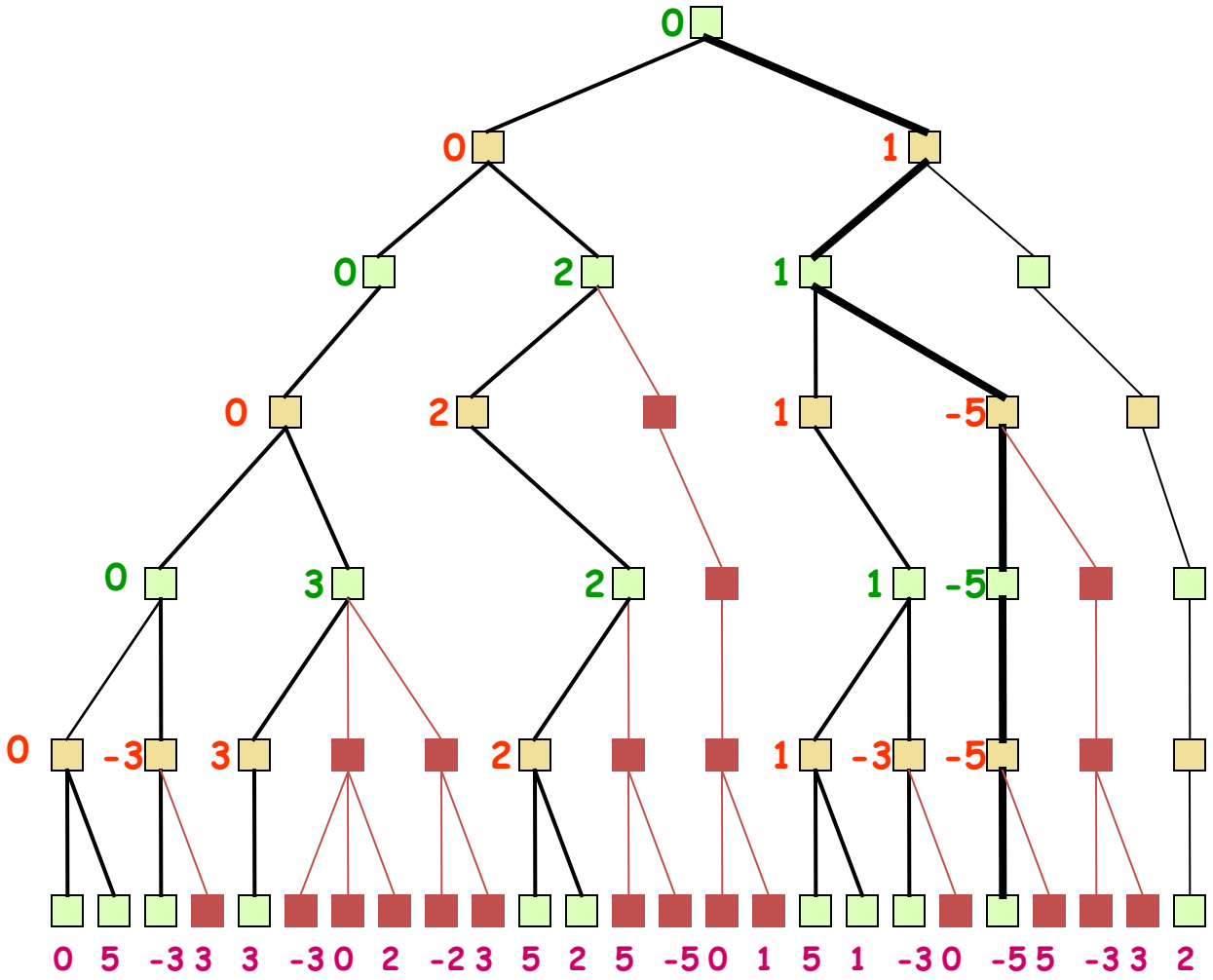
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



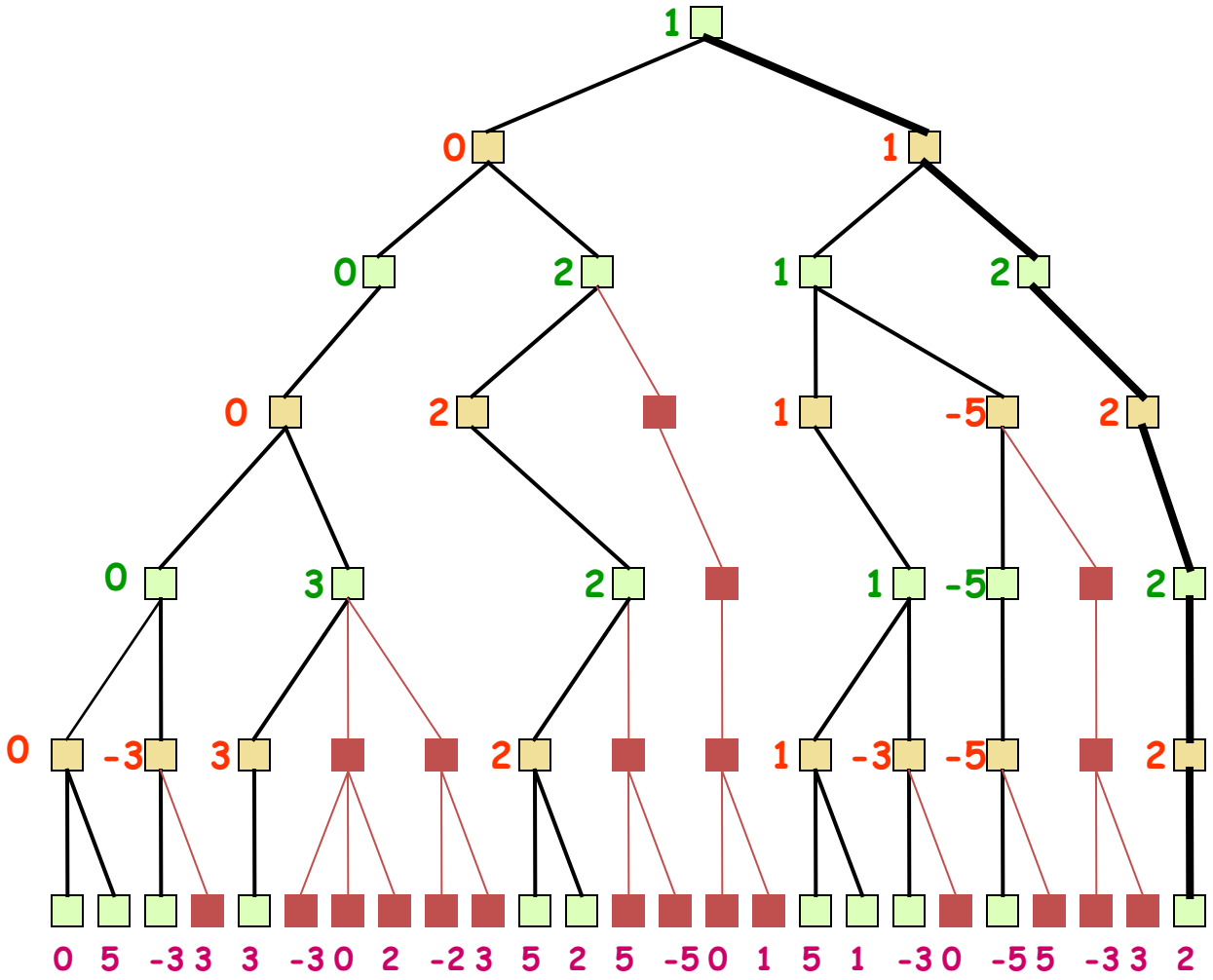
α-β pruning: example 3

α *Pruning* *node>=* β

β *Pruning* *node<=* α

MAX

MIN



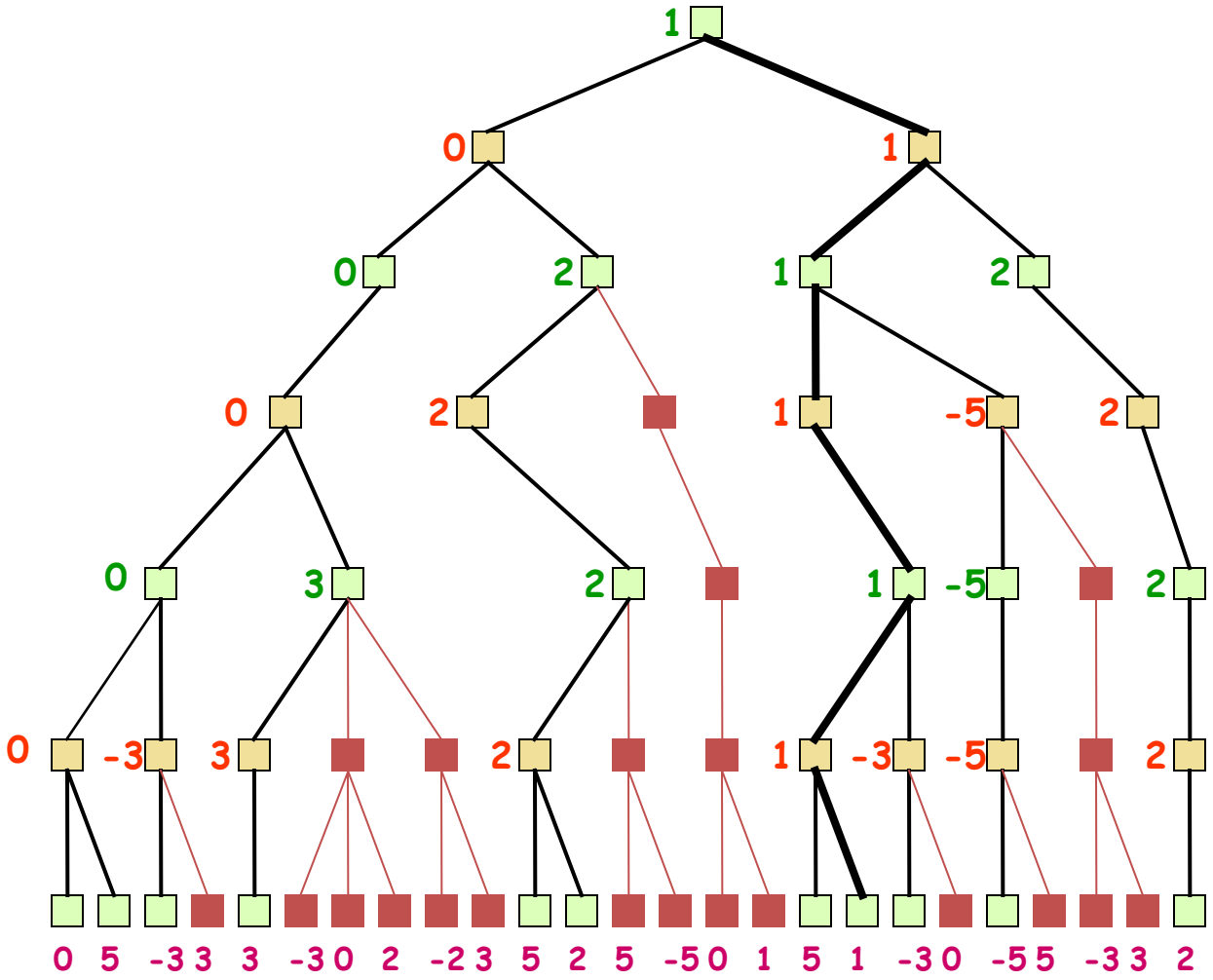
α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α

MAX

MIN



α-β algorithm:

```
function ALPHA-BETA-DECISION(state) returns an action
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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
  same as MAX-VALUE but with roles of α, β reversed
```

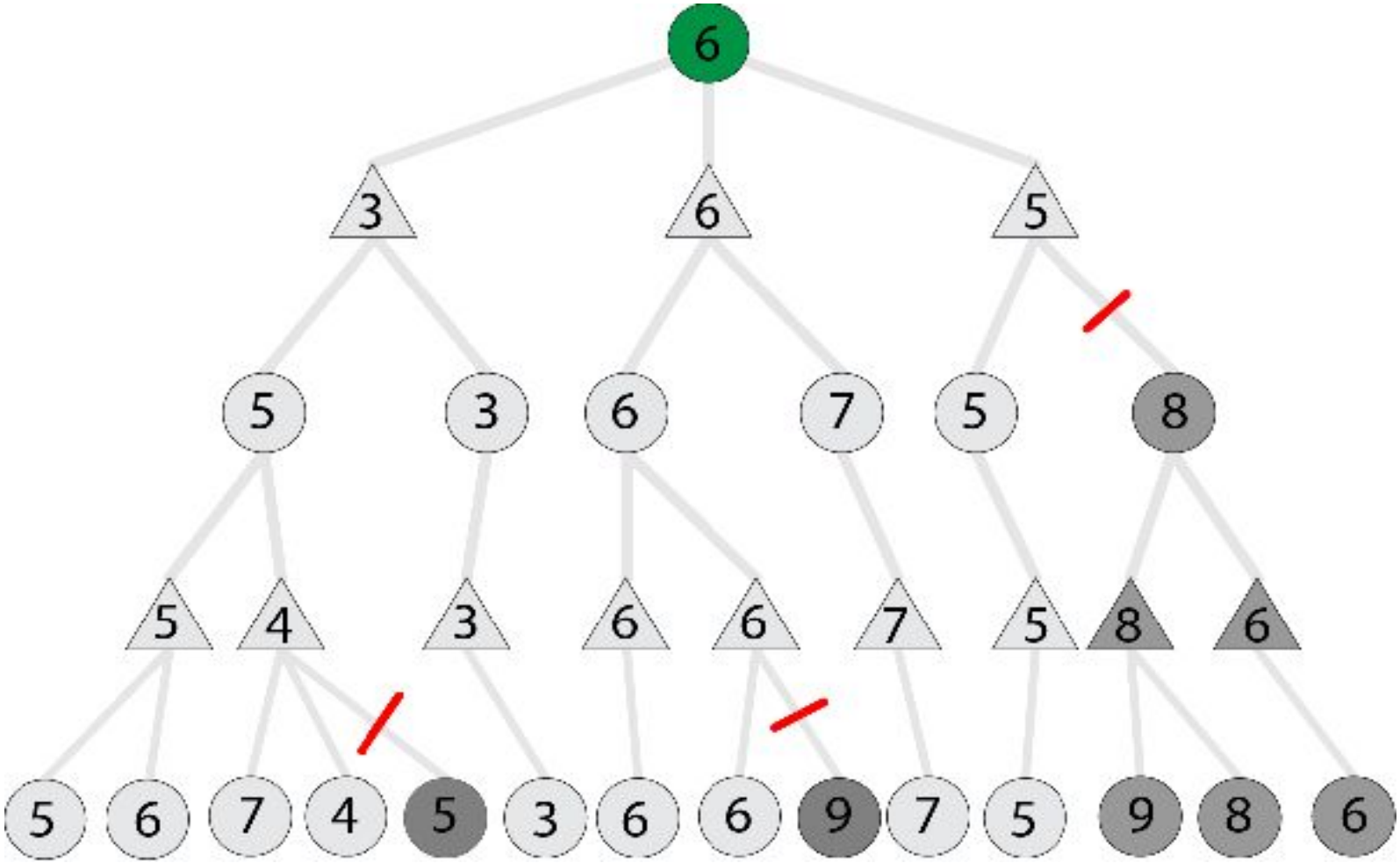
Effectiveness of alpha-beta

- **Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal computation**
- **Worst case: no pruning, examining b^d leaf nodes, where each node has b children and a d-ply search is performed**
- **Best case: examine only $(2b)^{d/2}$ leaf nodes.**
  - Result is you can search twice as deep as minimax!
- **Best case is when each player's best move is the first alternative generated**
- **In Deep Blue, they found empirically that alpha-beta pruning meant that the average branching factor at each node was about 6 instead of about 35!**

α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α



α-β pruning: example 3

α Pruning node>= β

β Pruning node<= α