CS 480

Introduction to Artificial Intelligence

September 16th, 2021

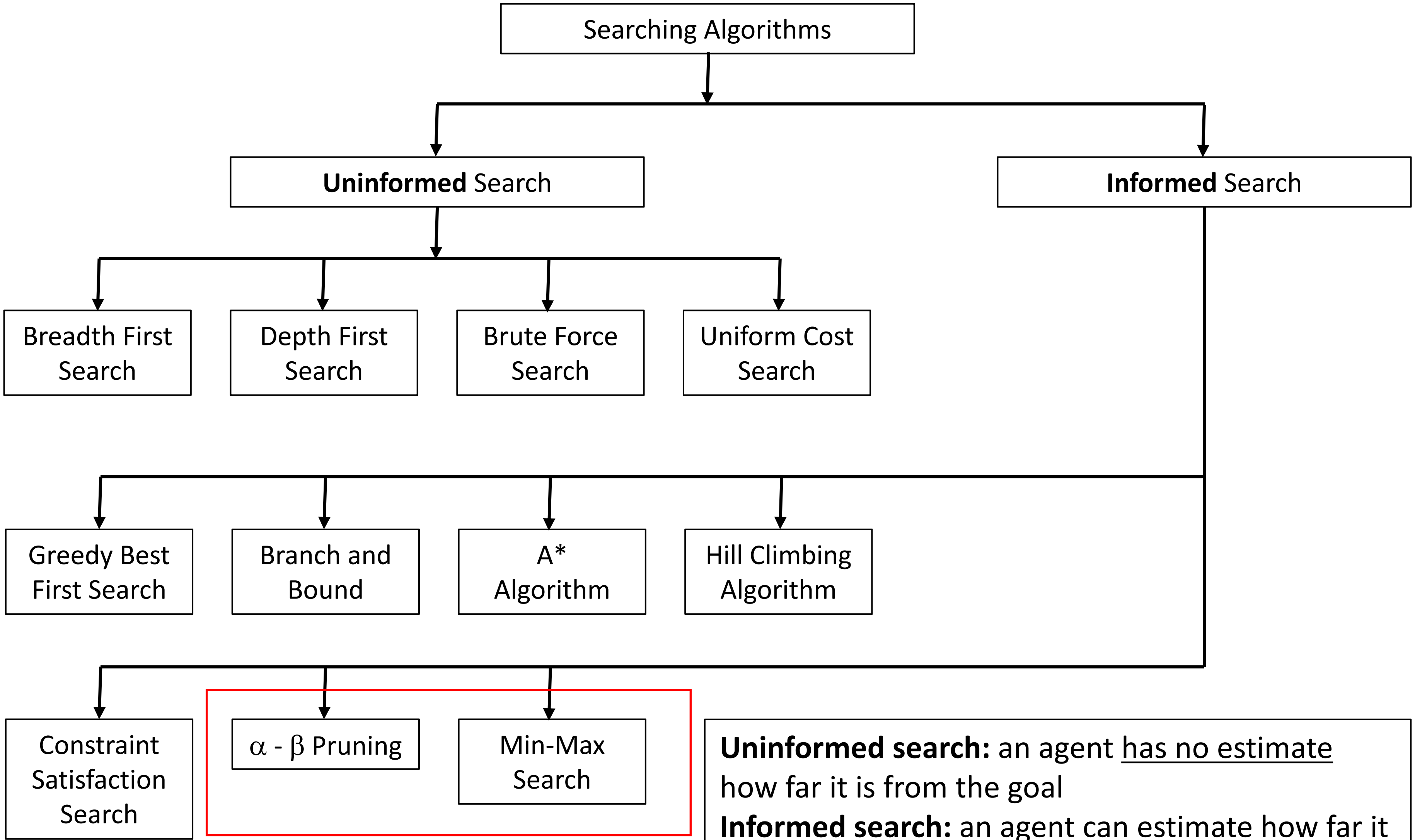
Announcements / Reminders

- **Written Assignment #01 is posted**
  - **due on Wednesday (09/22/21) at 11:00 PM CST**
- **Contribute to the discussion on Blackboard, please**
- **Please follow the Week 04 To Do List instructions**

Plan for Today

- **Adversarial Search: MinMax / α-β Pruning**
- **Constraint Satisfaction Problems: Introduction**

Selected Searching Algorithms

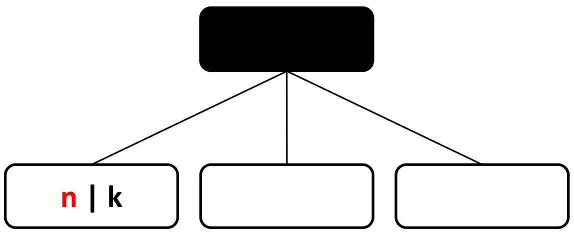


Uninformed search: an agent <u>has no estimate</u> how far it is from the goal
Informed search: an agent <u>can estimate</u> how far it is from the goal

MinMax: Assigning MINMAX Values

CASE 1:
State n is Terminal Node

ISTERMINAL(n) = true
TOMOVE(n) = MAX or MIN



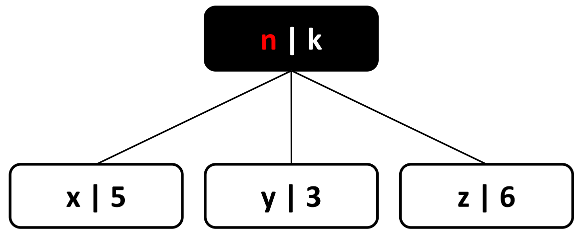
TERMINAL/LEAF

$$k = MINMAX(n) = UTILITY(n)$$

$$= \textit{utility value of this state for MAX Player}$$

CASE 2:
State n is a Non-Terminal Node and it is MIN Player's move

ISTERMINAL(n) = false
TOMOVE(n) = MIN



$$k = MINMAX(n) =$$

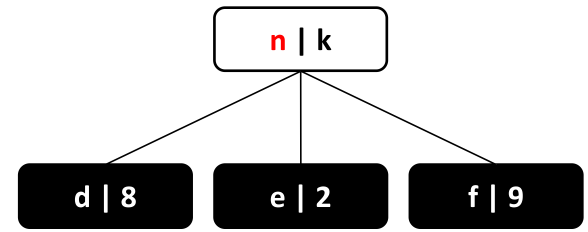
$$= min_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

$$= min(MINMAX(x), MINMAX(y), MINMAX(z))$$

$$= min(5, 3, 6)$$

CASE 3:
State n is a Non-Terminal Node and it is MAX Player's move

ISTERMINAL(n) = false
TOMOVE(n) = MAX



$$k = MINMAX(n) =$$

$$= max_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

$$= max(MINMAX(d), MINMAX(e), MINMAX(f))$$

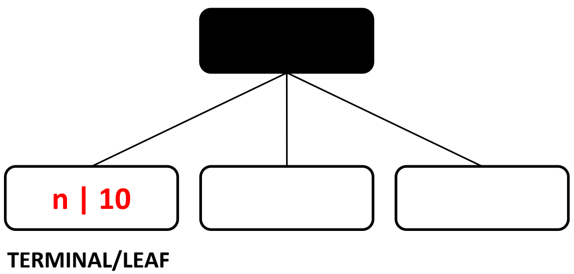
$$= max(8, 2, 9)$$

$$MINMAX(n) = \begin{cases} UTILITY(n, MAX), if\ ISTERMINAL(n) \\ max_{a \in ACTIONS(n)} MINMAX(RESULT(n, a)), if\ TOMOVE(s) = MAX \\ min_{a \in ACTIONS(n)} MINMAX(RESULT(n, a)), if\ TOMOVE(s) = MIN \end{cases}$$

MinMax: Assigning MINMAX Values

CASE 1:
State n is Terminal Node

ISTERMINAL(n) = true
TOMOVE(n) = MAX or MIN



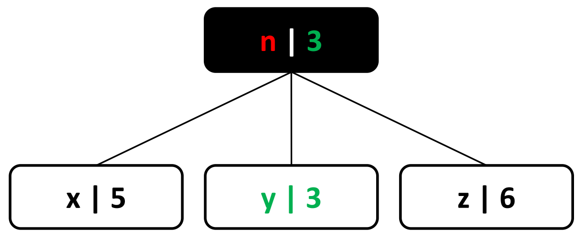
$$k = MINMAX(n) = UTILITY(n)$$

$$= \textit{utility value of this state for MAX Player}$$

$$= 10$$

CASE 2:
State n is a Non-Terminal Node and it is MIN Player's move

ISTERMINAL(n) = false
TOMOVE(n) = MIN



$$k = MINMAX(n) =$$

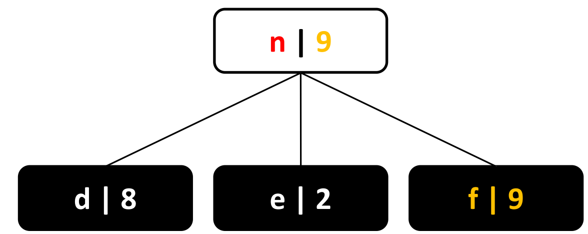
$$= min_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

$$= min(MINMAX(x), MINMAX(y), MINMAX(z))$$

$$= min(5, 3, 6) = 3$$

CASE 3:
State n is a Non-Terminal Node and it is MAX Player's move

ISTERMINAL(n) = false
TOMOVE(n) = MAX



$$k = MINMAX(n) =$$

$$= max_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

$$= max(MINMAX(d), MINMAX(e), MINMAX(f))$$

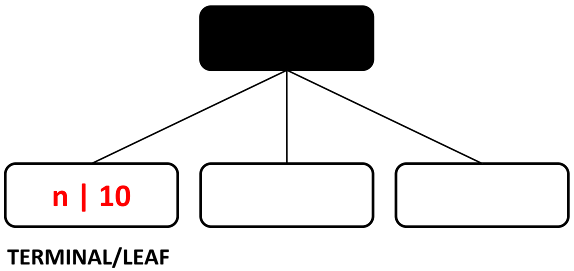
$$= max(8, 2, 9) = 9$$

$$MINMAX(n) = \begin{cases} UTILITY(n, MAX), \textit{ if } ISTERMINAL(n) \\ max_{a \in ACTIONS(n)} MINMAX(RESULT(n, a)), \textit{ if } TOMOVE(s) = MAX \\ min_{a \in ACTIONS(n)} MINMAX(RESULT(n, a)), \textit{ if } TOMOVE(s) = MIN \end{cases}$$

MinMax: Assigning MINMAX Values

CASE 1:
State n is Terminal Node

ISTERMINAL(n) = true
TOMOVE(n) = MAX or MIN



$$k = MINMAX(n) = UTILITY(n)$$

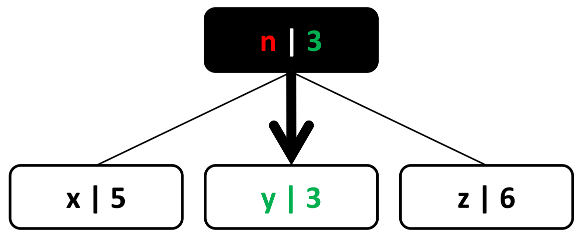
$$= \textit{utility value of this state for MAX Player}$$

$$= 10$$

What does it mean?
Utility of node n, to MAX Player, is 10 (if the game gets here, this is what MAX Player will receive)

CASE 2:
State n is a Non-Terminal Node and it is MIN Player's move

ISTERMINAL(n) = false
TOMOVE(n) = MIN



$$k = MINMAX(n) =$$

$$= min_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

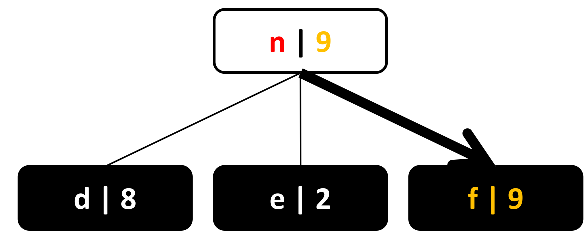
$$= min(MINMAX(x), MINMAX(y), MINMAX(z))$$

$$= min(5, 3, 6) = 3$$

What does it mean?
At node n, MIN Player will choose a move from n to y to MINIMIZE MAX Player's utility

CASE 3:
State n is a Non-Terminal Node and it is MAX Player's move

ISTERMINAL(n) = false
TOMOVE(n) = MAX



$$k = MINMAX(n) =$$

$$= max_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

$$= max(MINMAX(d), MINMAX(e), MINMAX(f))$$

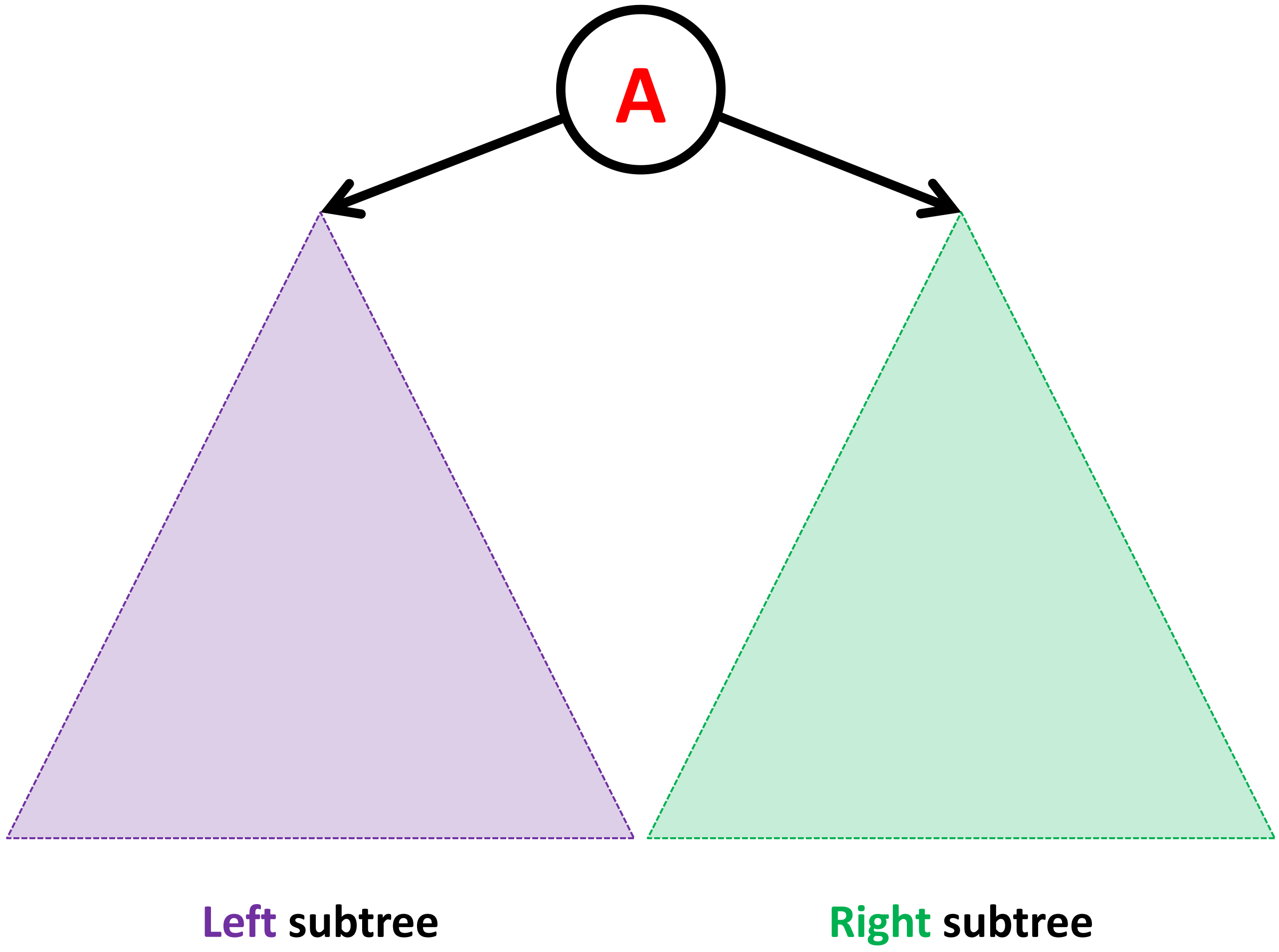
$$= max(8, 2, 9) = 9$$

What does it mean?
At node n, MAX Player will choose a move from n to f to MAXIMIZE MAX Player's utility

MinMax Algorithm: Pseudocode

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Search Tree: Recursive Structure

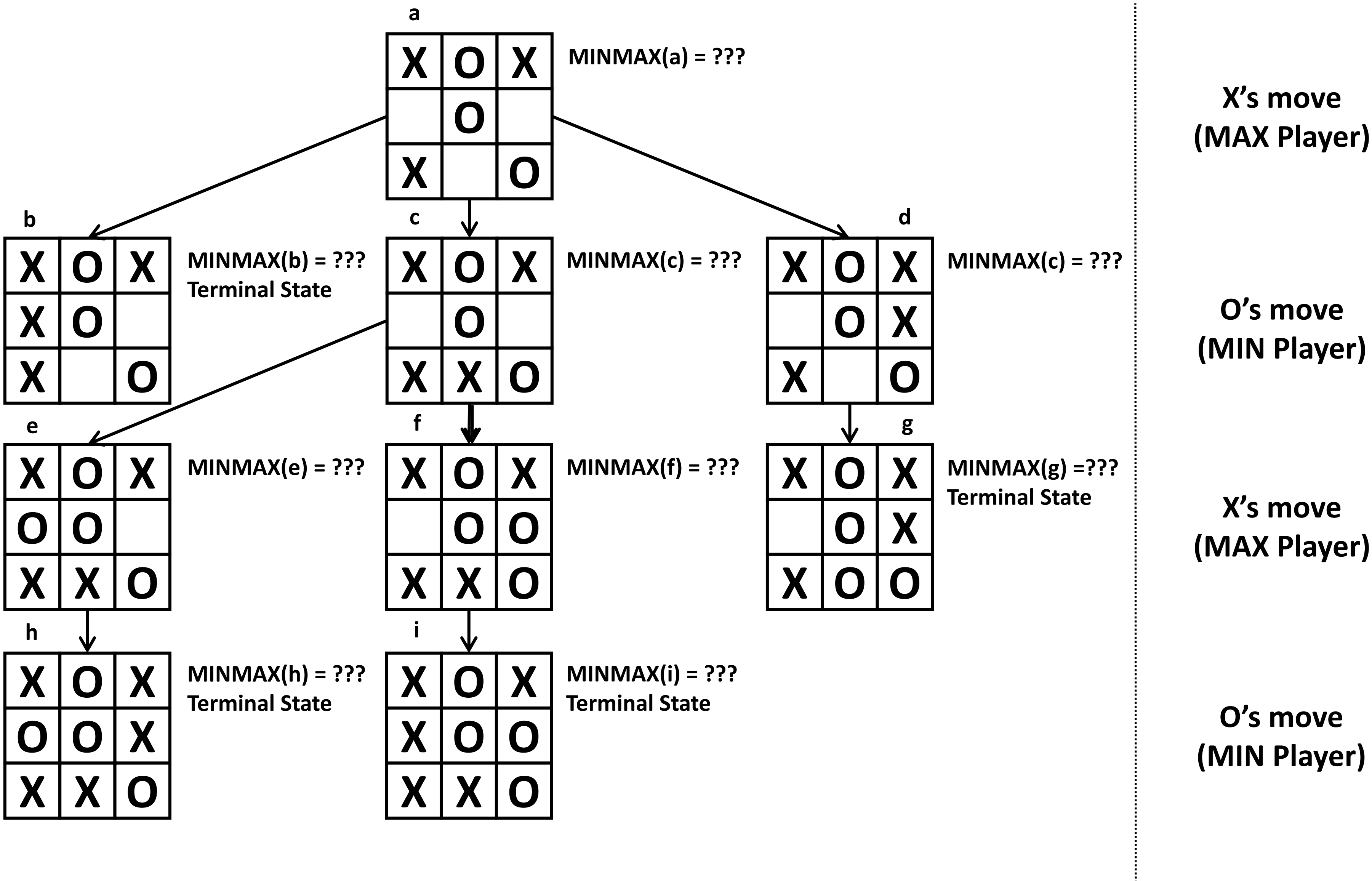


MinMax Algorithm: Pseudocode

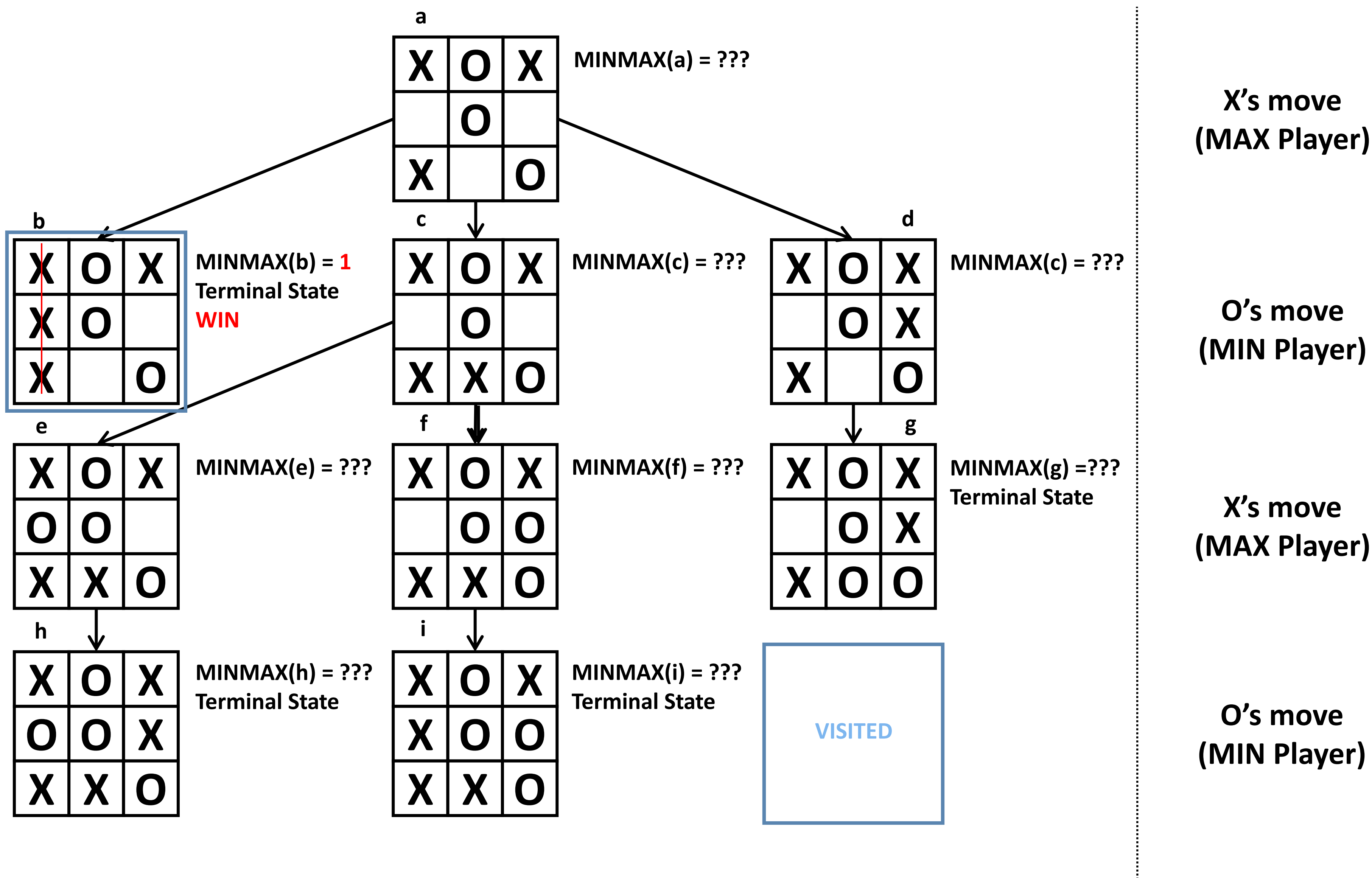
```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

RECURSION

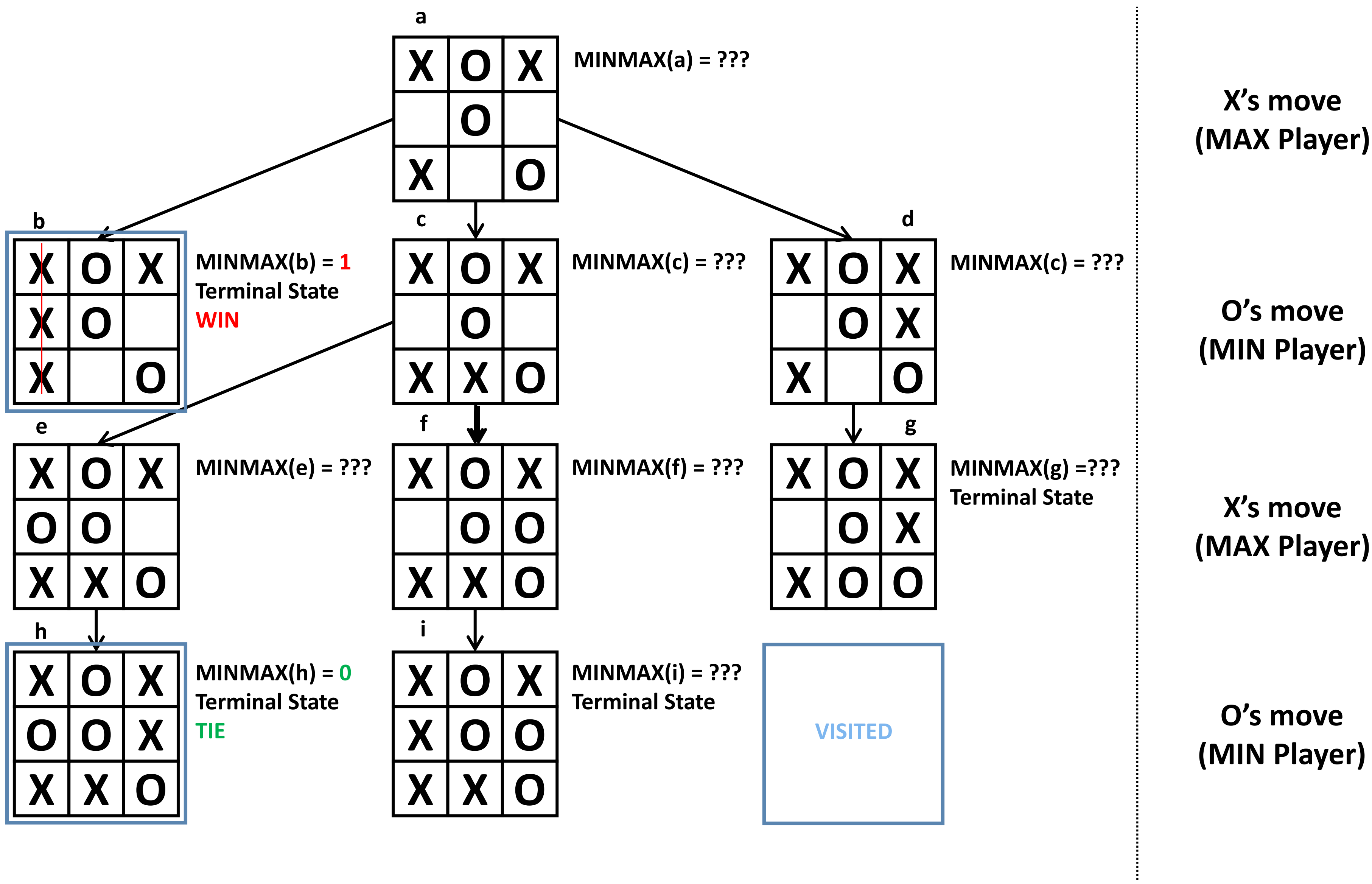
MinMax Algorithm: Tic Tac Toe



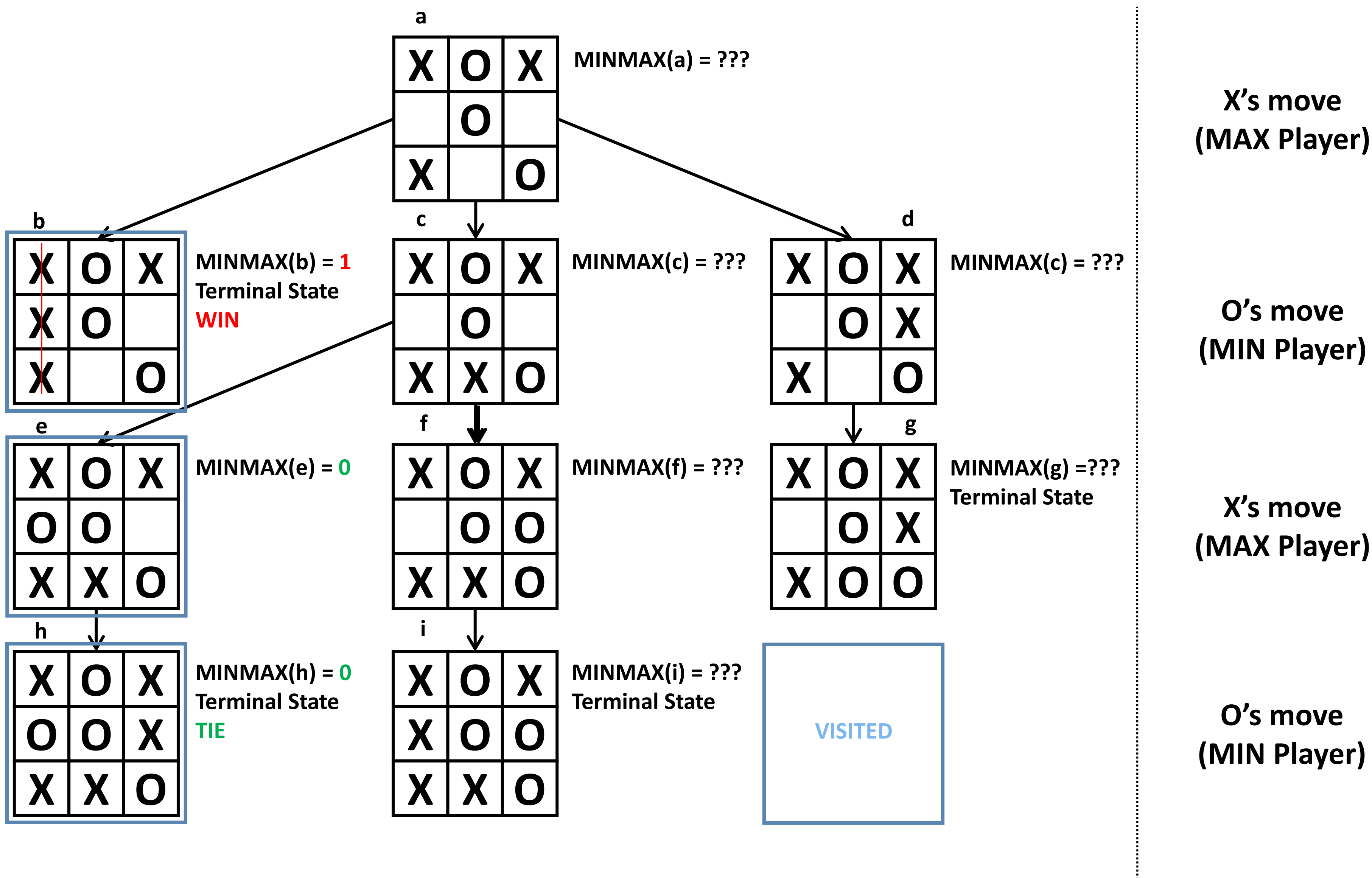
MinMax Algorithm: Tic Tac Toe



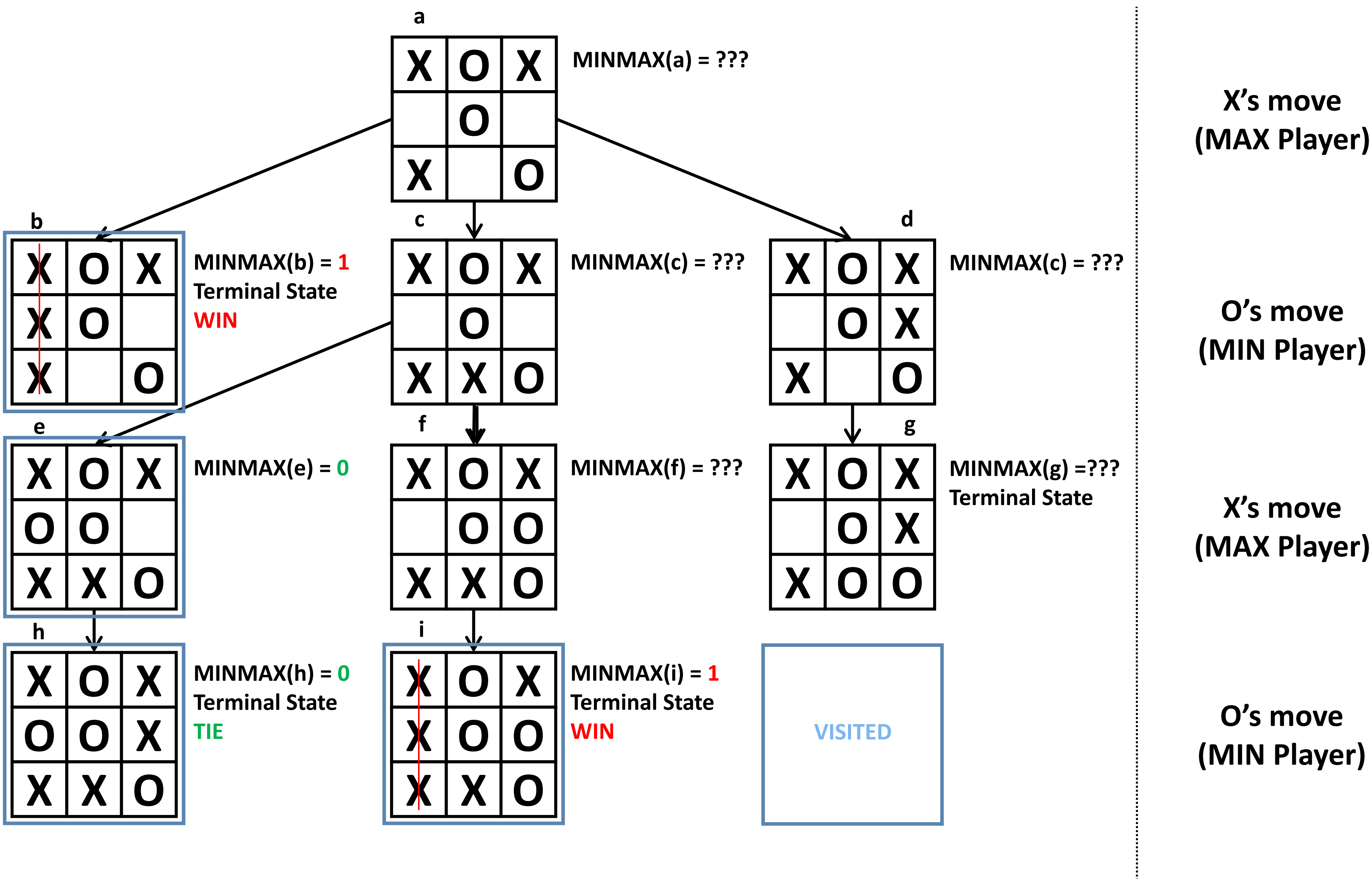
MinMax Algorithm: Tic Tac Toe



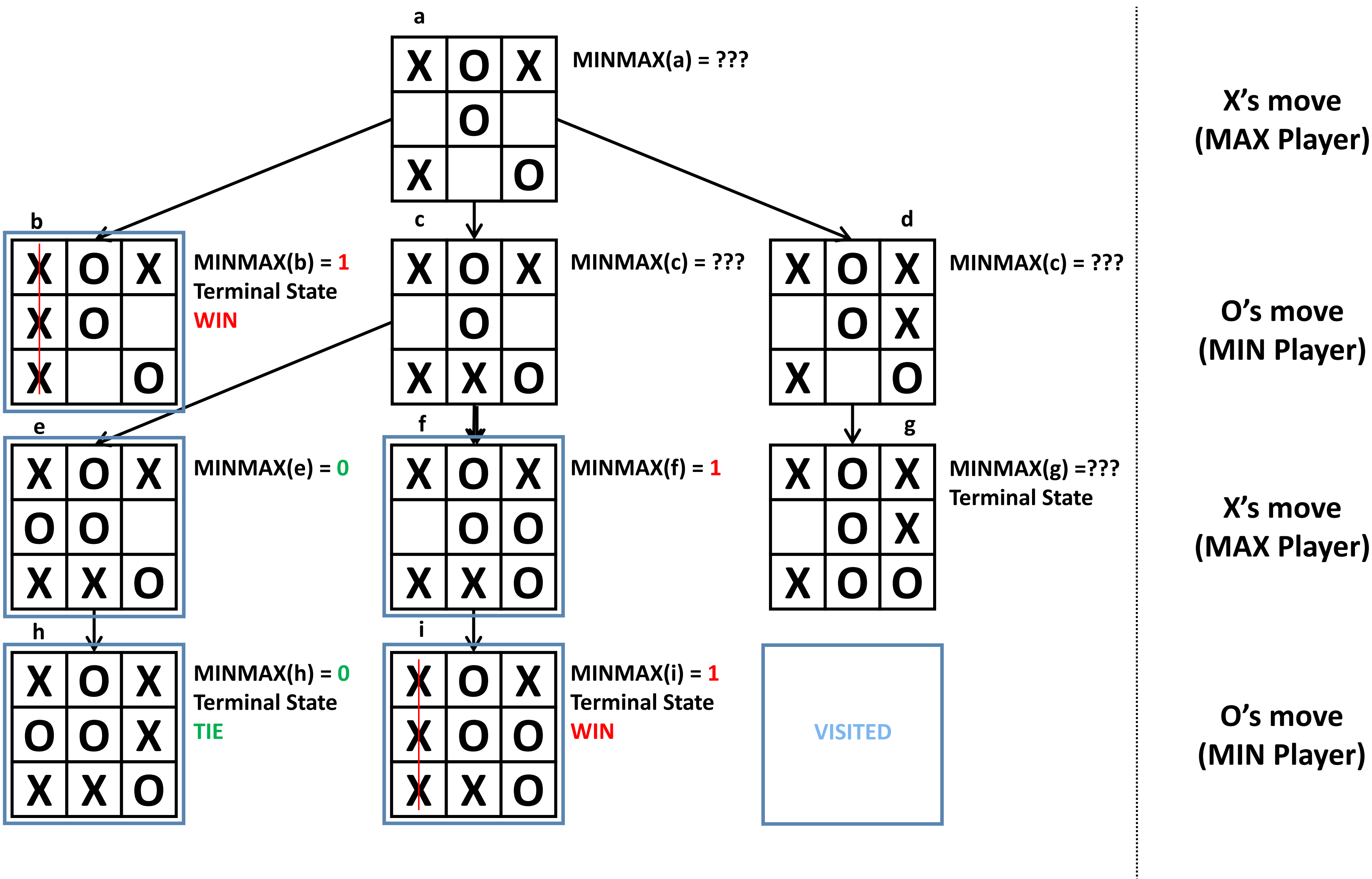
MinMax Algorithm: Tic Tac Toe



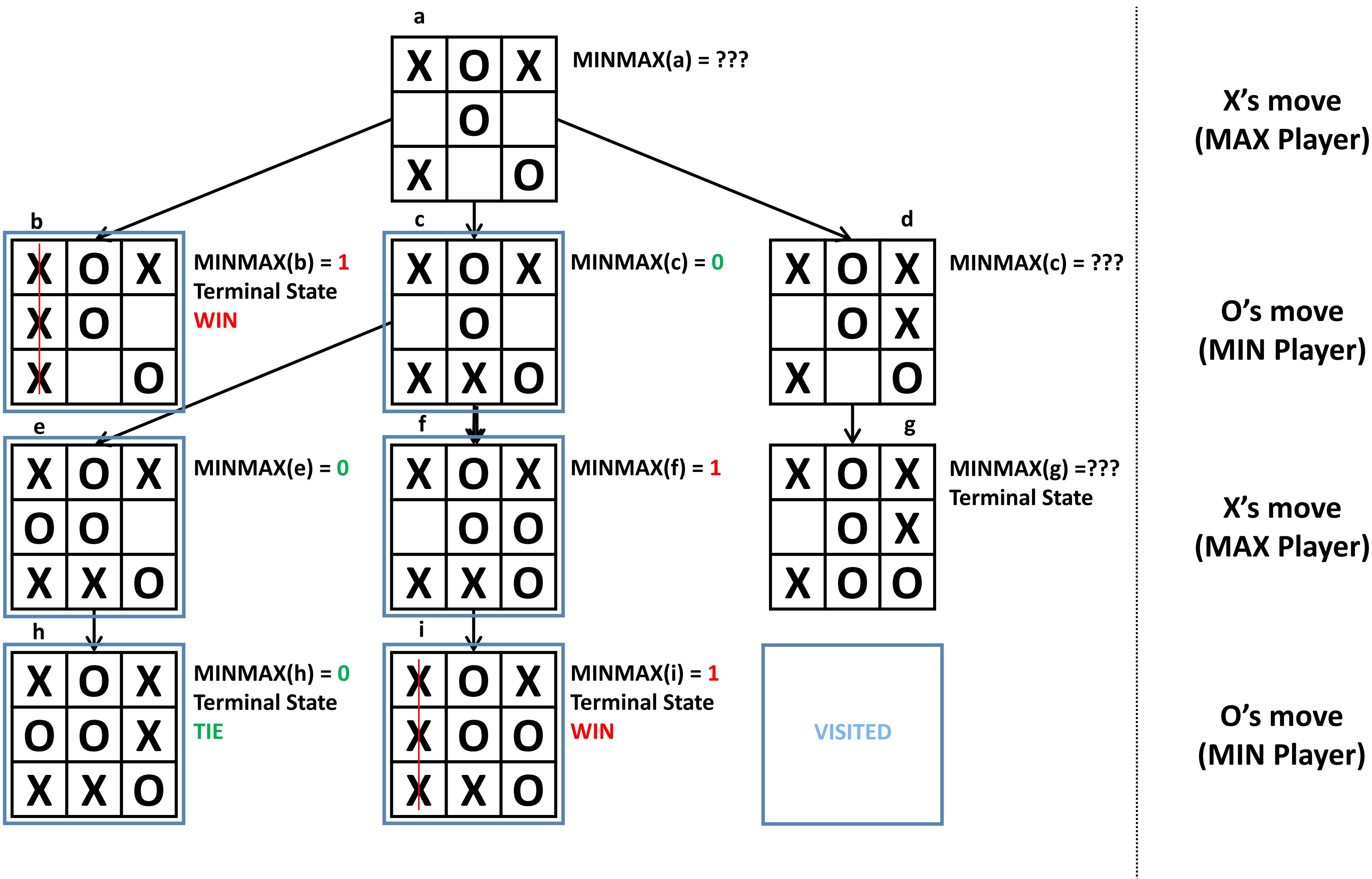
MinMax Algorithm: Tic Tac Toe



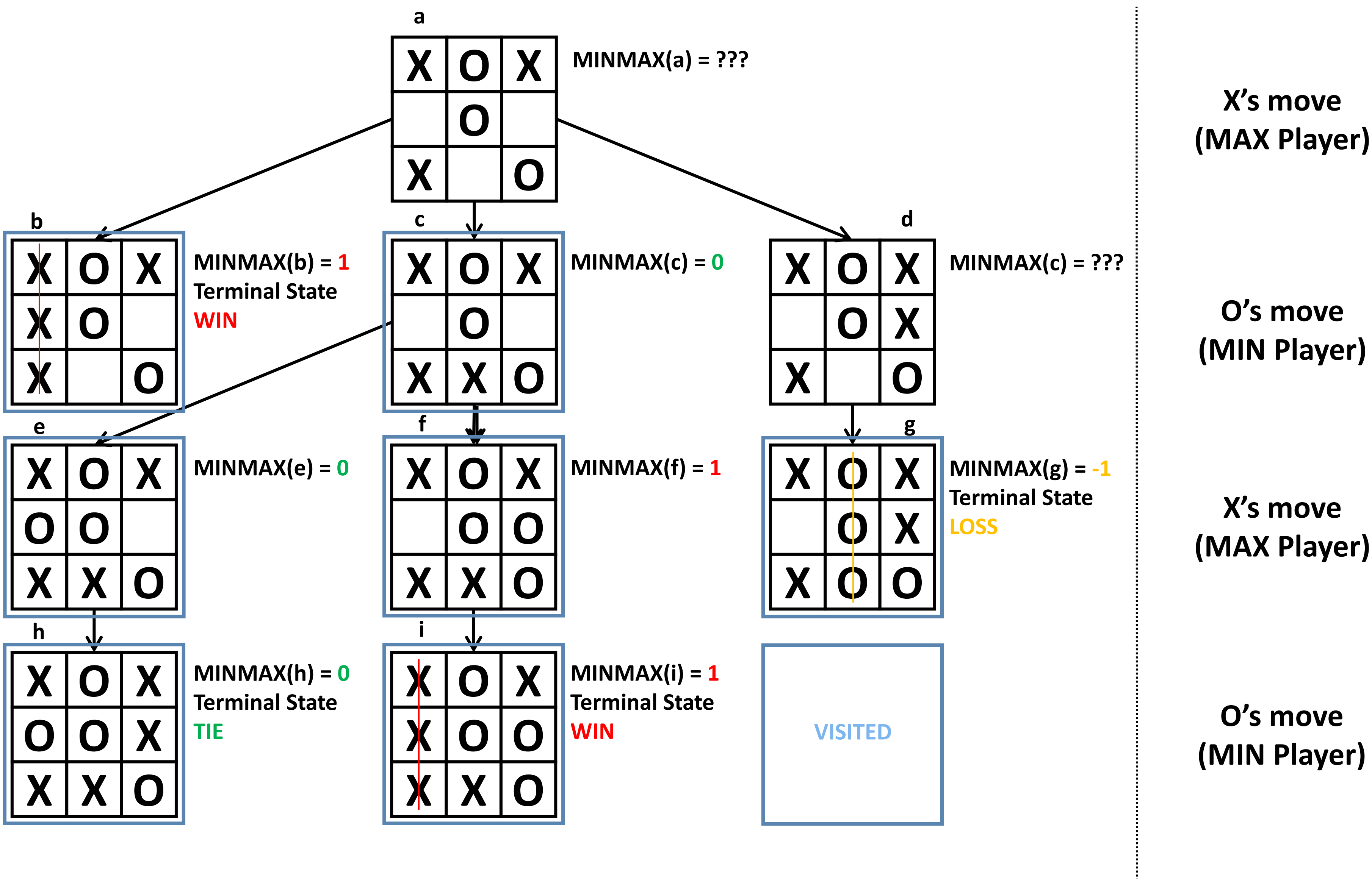
MinMax Algorithm: Tic Tac Toe



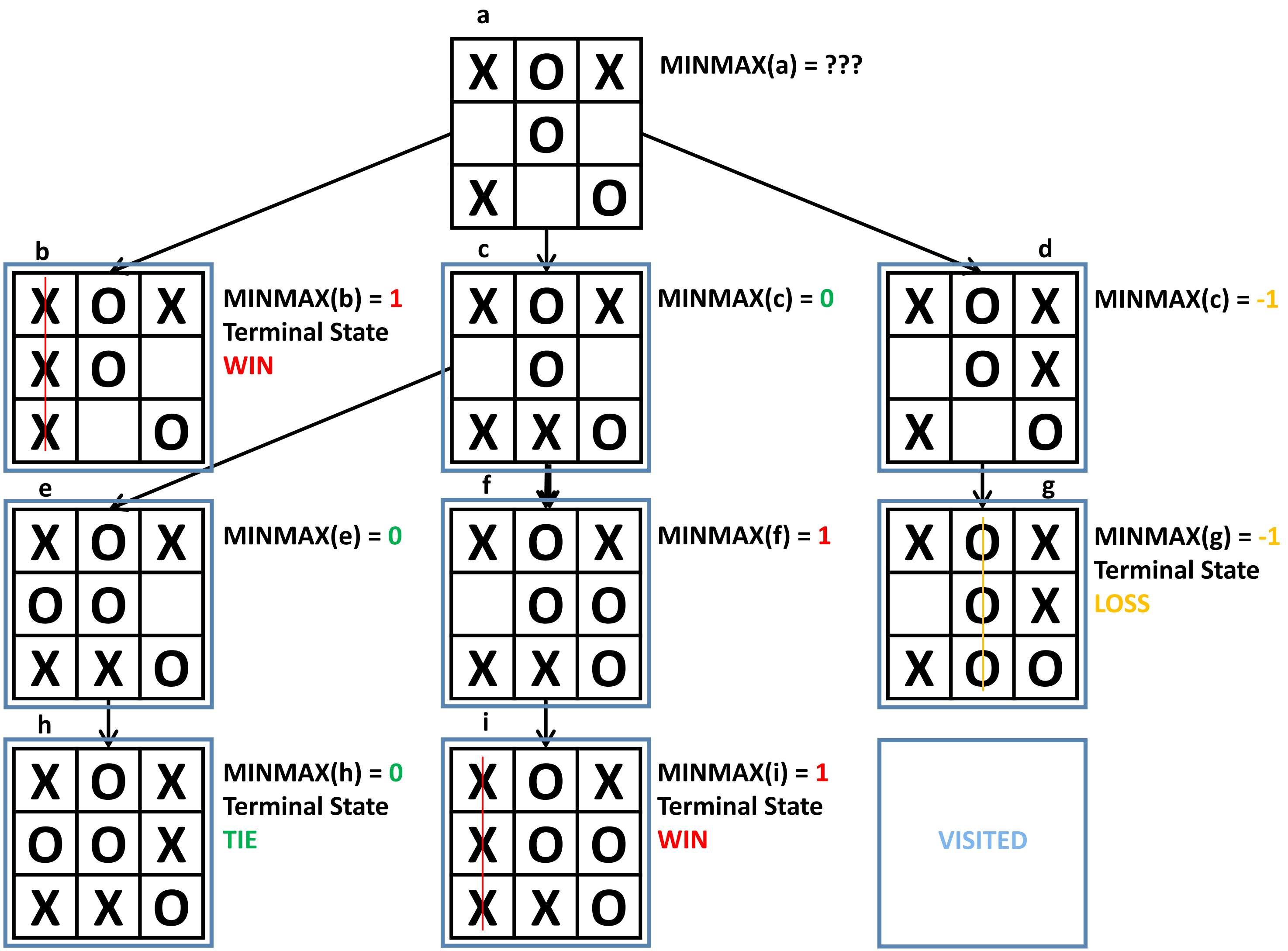
MinMax Algorithm: Tic Tac Toe



MinMax Algorithm: Tic Tac Toe



MinMax Algorithm: Tic Tac Toe



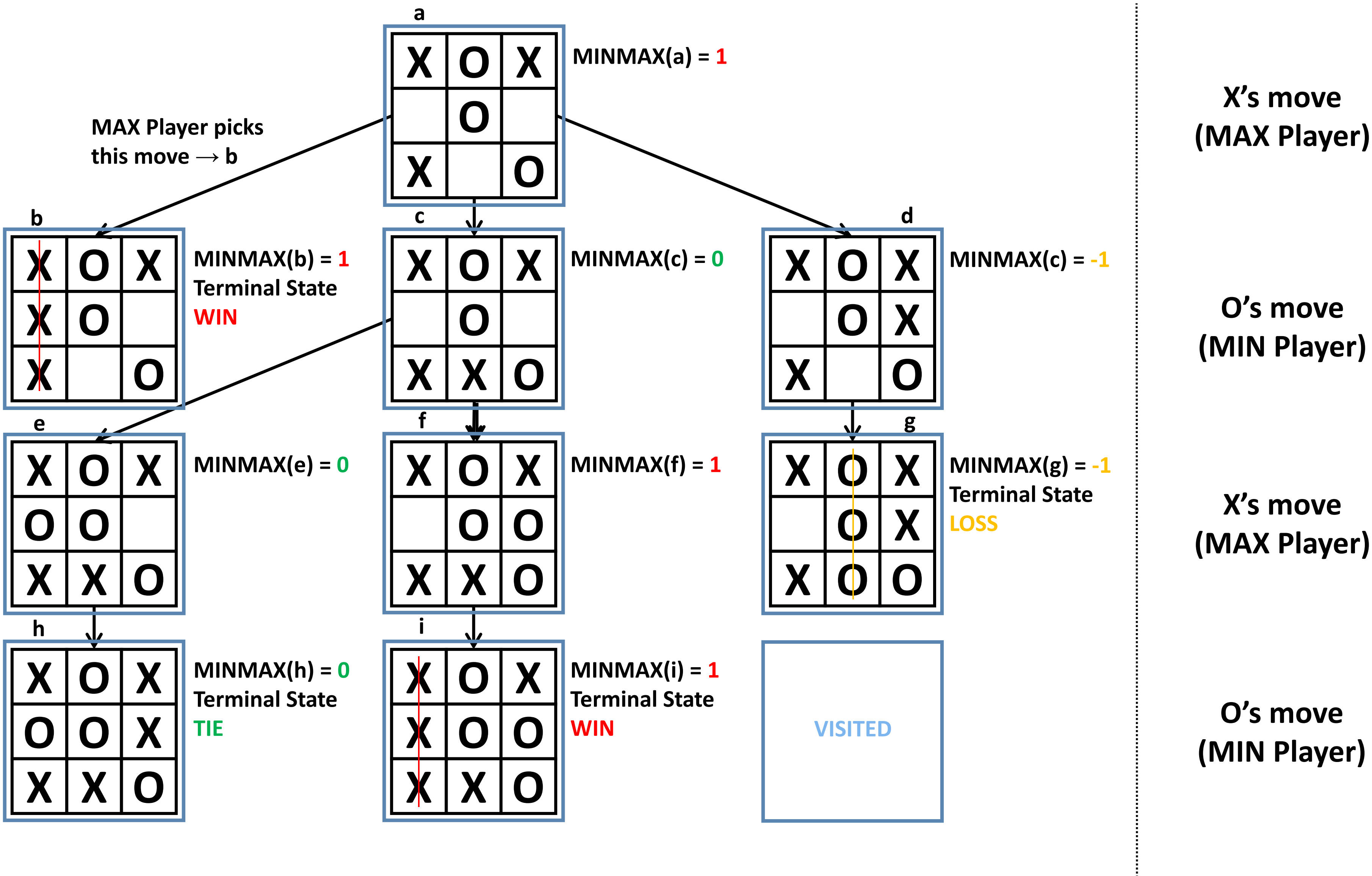
X's move
(MAX Player)

O's move
(MIN Player)

X's move
(MAX Player)

O's move
(MIN Player)

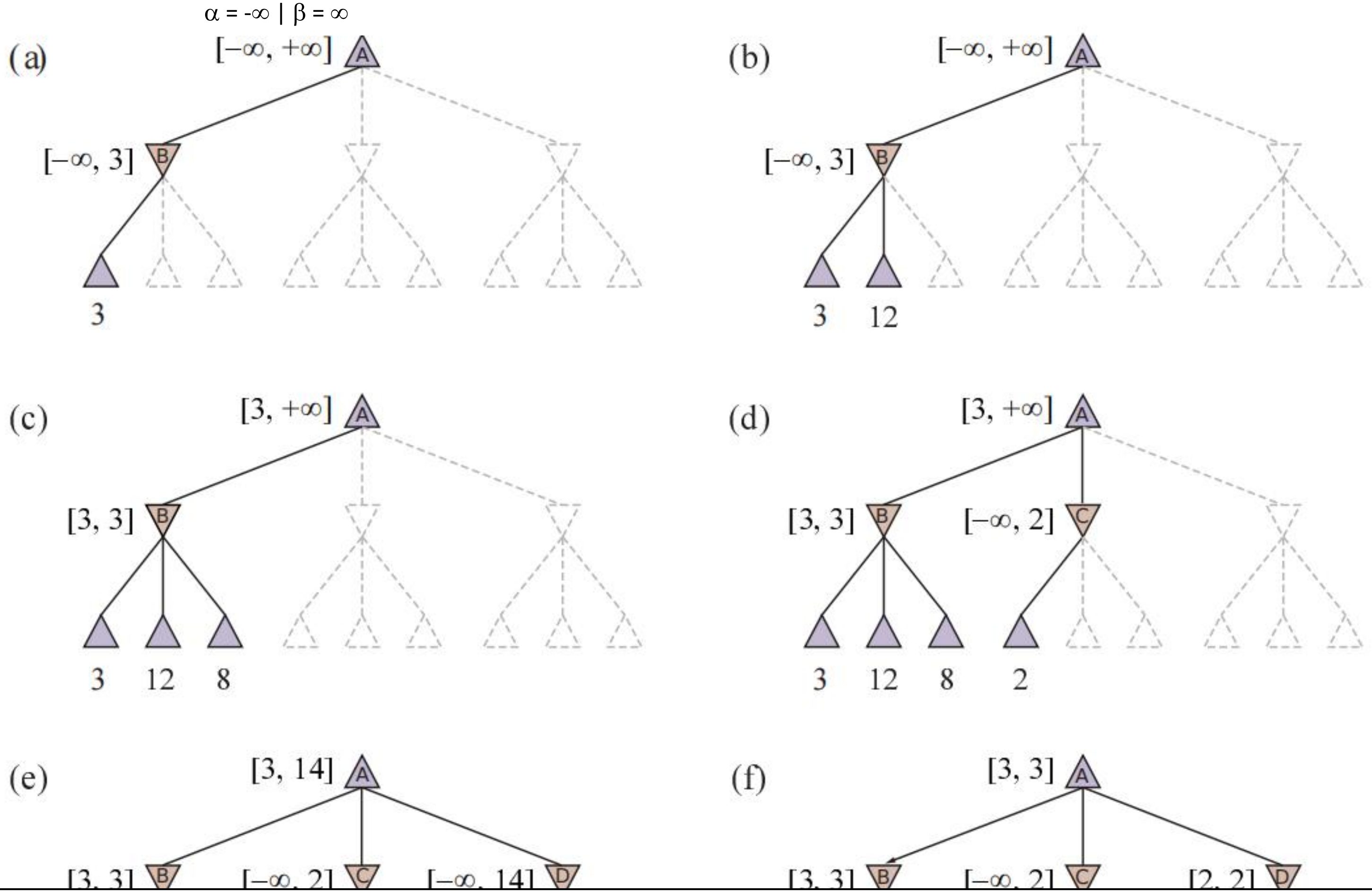
MinMax Algorithm: Tic Tac Toe



MinMax with α-β: Pseudocode

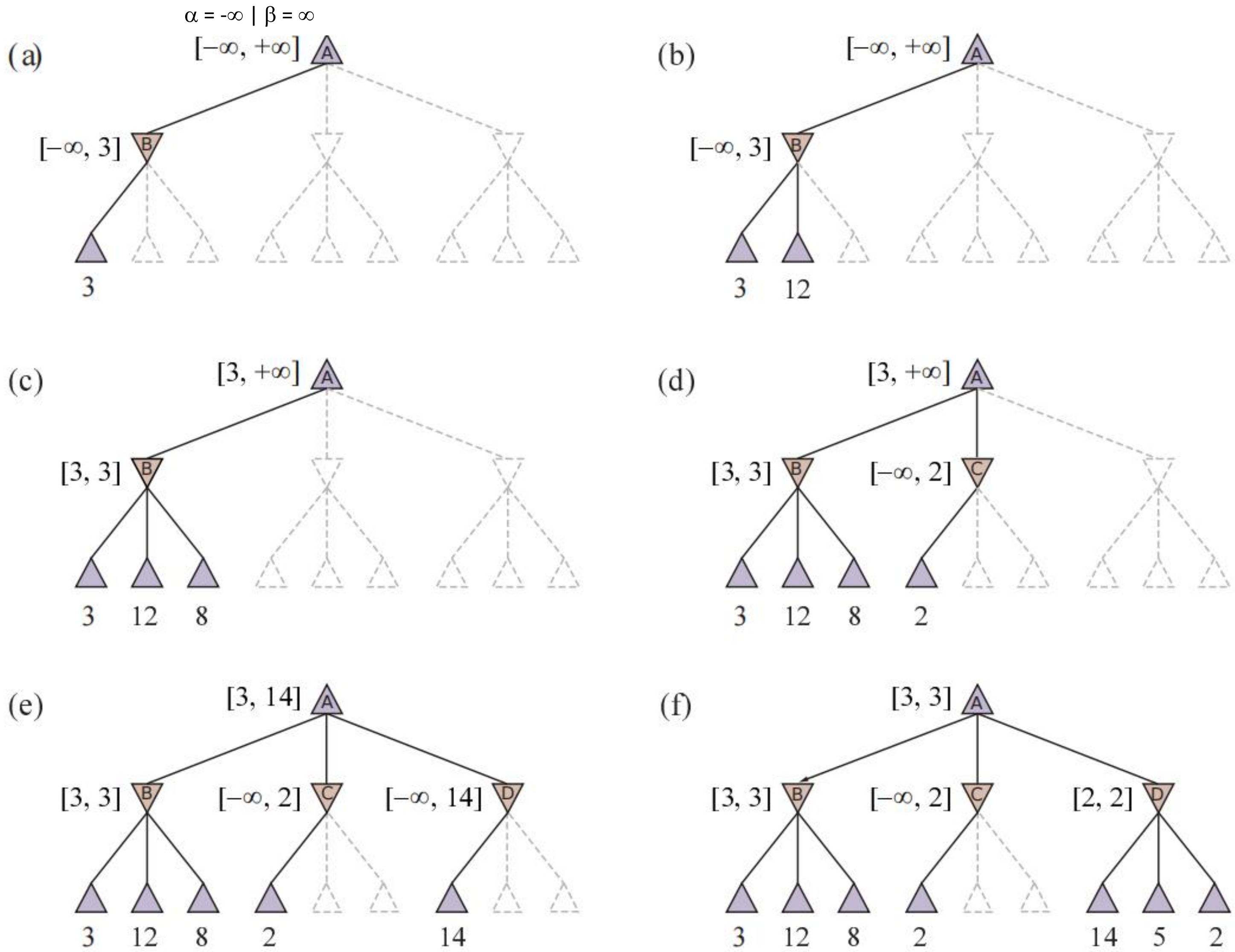
```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Example MinMax with 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 player ("at least")
β: the value of the best (lowest-value) choice we have found so far at any choice point along the path for MIN player ("at most")

Example MinMax with α-β Pruning



MinMax with α-β: Pseudocode

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

RECURSION

MinMax with α-β: Pseudocode

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move
```

```
function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move
```

MAX Player's move

```
function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

MIN Player's move

MinMax with α-β: Pseudocode

function ALPHA-BETA-SEARCH(*game*, *state*) **returns** an action
  player $\leftarrow$ *game*.TO-MOVE(*state*)
  value, *move* $\leftarrow$ MAX-VALUE(*game*, *state*, $-\infty$, $+\infty$)
  return *move*

function MAX-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow -\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MIN-VALUE(*game*, *game*.RESULT(*state*, *a*), α, β)
    if $v2 > v$ **then**
      v, *move* $\leftarrow$ *v2*, *a*
      $\alpha \leftarrow$ MAX(α, v)
    if $v \geq \beta$ **then return** *v*, *move*
  return *v*, *move*

Go through all legal actions/moves (subtrees) recursively

MAX Player's move

function MIN-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MAX-VALUE(*game*, *game*.RESULT(*state*, *a*), α, β)
    if $v2 < v$ **then**
      v, *move* $\leftarrow$ *v2*, *a*
      $\beta \leftarrow$ MIN(β, v)
    if $v \leq \alpha$ **then return** *v*, *move*
  return *v*, *move*

Go through all legal actions/moves (subtrees) recursively

MIN Player's move

MinMax with α-β: Pseudocode

```
function ALPHA-BETA-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state, −∞, +∞)
   return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
      if v2 > v then
         v, move ← v2, a
         α ← MAX(α, v)
      if v ≥ β then return v, move
   return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
      if v2 < v then
         v, move ← v2, a
         β ← MIN(β, v)
      if v ≤ α then return v, move
   return v, move
```

MAX Player's move

Go through all legal actions/moves (subtrees) recursively

If higher MINMAX(subtree) value found
store a as the best move
update bound α (within this recursive call only!)

MIN Player's move

Go through all legal actions/moves (subtrees) recursively

If lower MINMAX(subtree) value found:
store a as the best move
update bound β (within this recursive call only!)

MinMax with α-β: Pseudocode

function ALPHA-BETA-SEARCH($game$, $state$) **returns** an action
  player ← $game$.TO-MOVE($state$)
  $value$, $move$ ← MAX-VALUE($game$, $state$, $-\infty$, $+\infty$)
  return $move$

function MAX-VALUE($game$, $state$, α, β) **returns** a ($utility$, $move$) pair
  if $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
  $v \leftarrow -\infty$
  for each a **in** $game$.ACTIONS($state$) **do**
    $v2$, $a2$ ← MIN-VALUE($game$, $game$.RESULT($state$, a), α, β)
    if $v2 > v$ **then**
      v, $move \leftarrow v2$, a
      $\alpha \leftarrow$ MAX(α, v)
    if $v \geq \beta$ **then return** v, $move$
  return v, $move$

If higher MINMAX(subtree) value found
store a as the best move
update bound α (within this recursive call only!)

Go through all legal actions/moves (subtrees) recursively

MAX Player does NOT change bound β here!

MAX Player's move

function MIN-VALUE($game$, $state$, α, β) **returns** a ($utility$, $move$) pair
  if $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
  $v \leftarrow +\infty$
  for each a **in** $game$.ACTIONS($state$) **do**
    $v2$, $a2$ ← MAX-VALUE($game$, $game$.RESULT($state$, a), α, β)
    if $v2 < v$ **then**
      v, $move \leftarrow v2$, a
      $\beta \leftarrow$ MIN(β, v)
    if $v \leq \alpha$ **then return** v, $move$
  return v, $move$

If lower MINMAX(subtree) value found:
store a as the best move
update bound β (within this recursive call only!)

Go through all legal actions/moves (subtrees) recursively

MIN Player does NOT change bound α here!

MIN Player's move

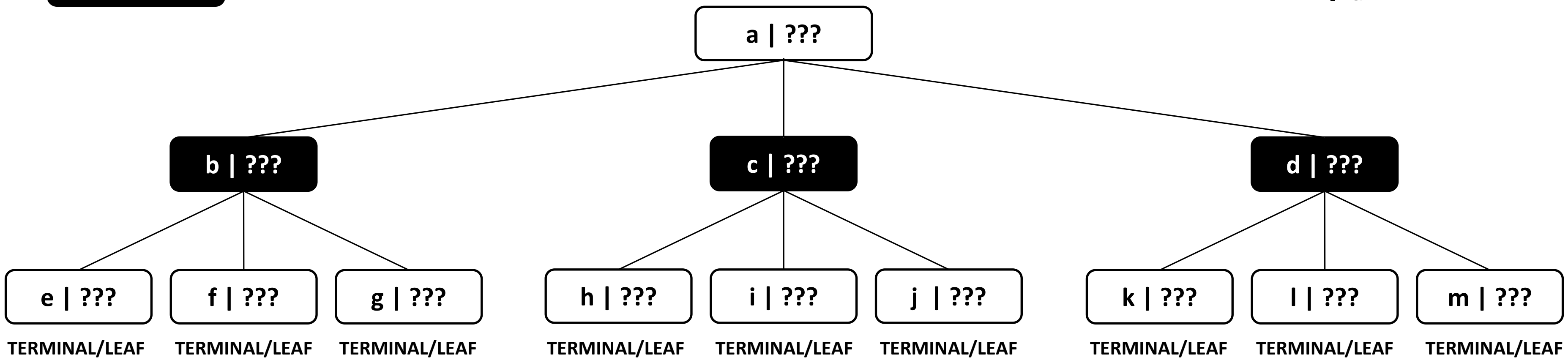
MinMax with α-β: Example

n | MINMAX(n) — MAX player state / move / turn

n | MINMAX(n) — MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = -\infty$

MINMAX(a) can be at most: $\beta_a = \infty$

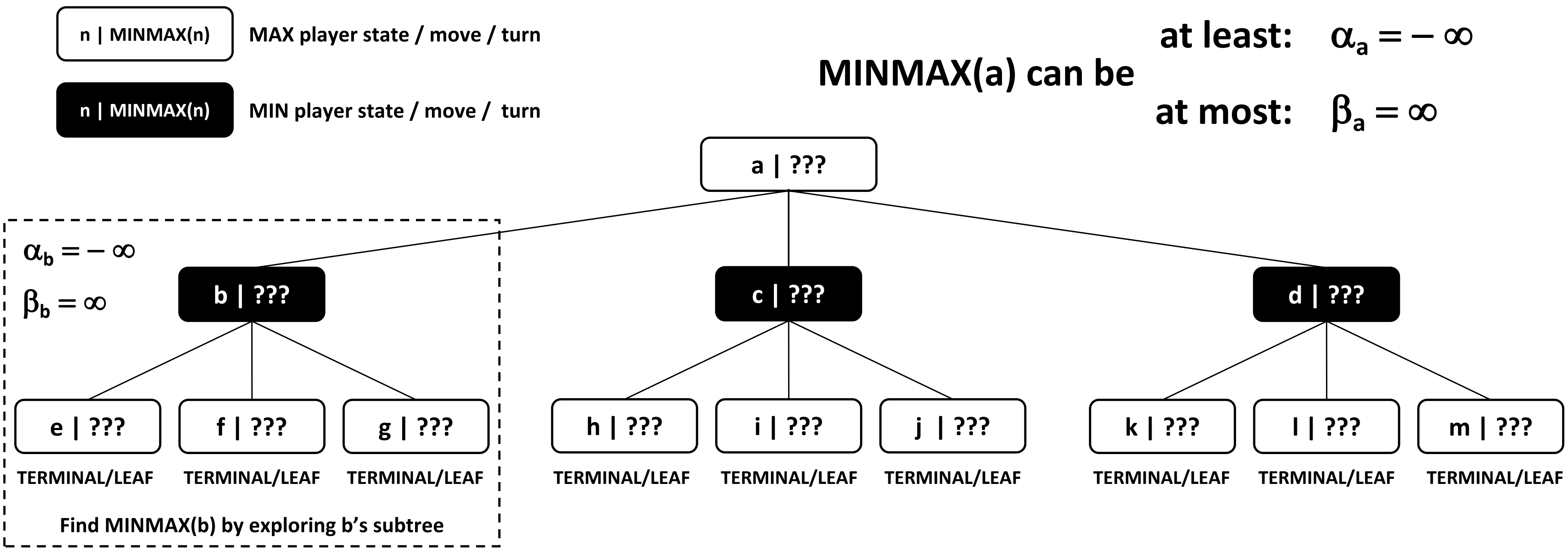


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

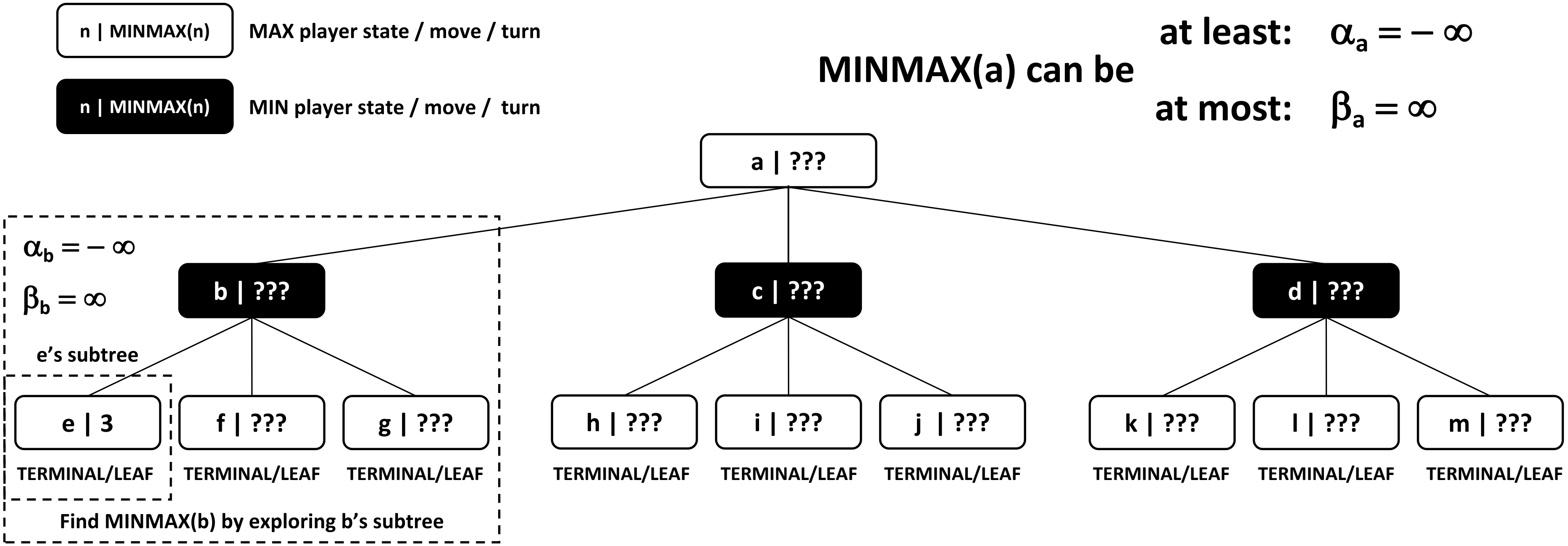
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

- **MIN Player (at node b) has not seen any successor MINMAX values yet → min MINMAX seen: v = ∞**
- **v > α_a ($\infty > -\infty$) → we can keep exploring b's subtree**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

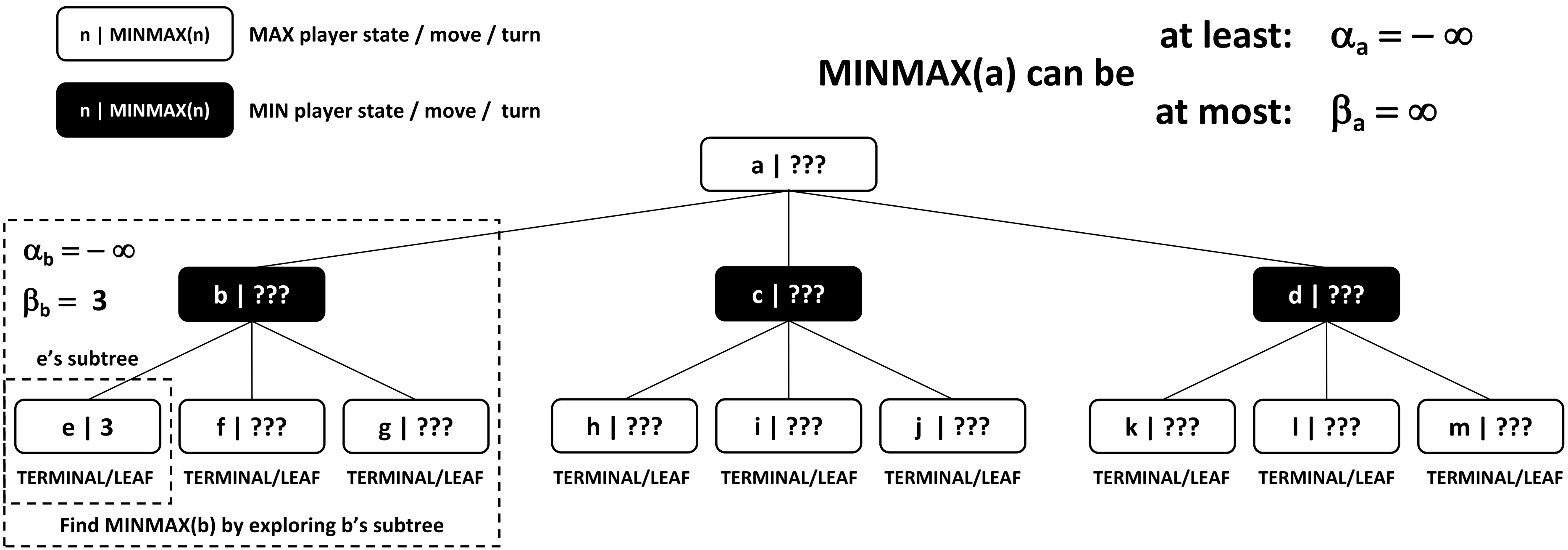
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

- **We need to analyze e's subtree**
- **Node e is a terminal node (Case 1) → MINMAX(e) = UTILITY(e) = 3 | v2 = MINMAX(e) = 3**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

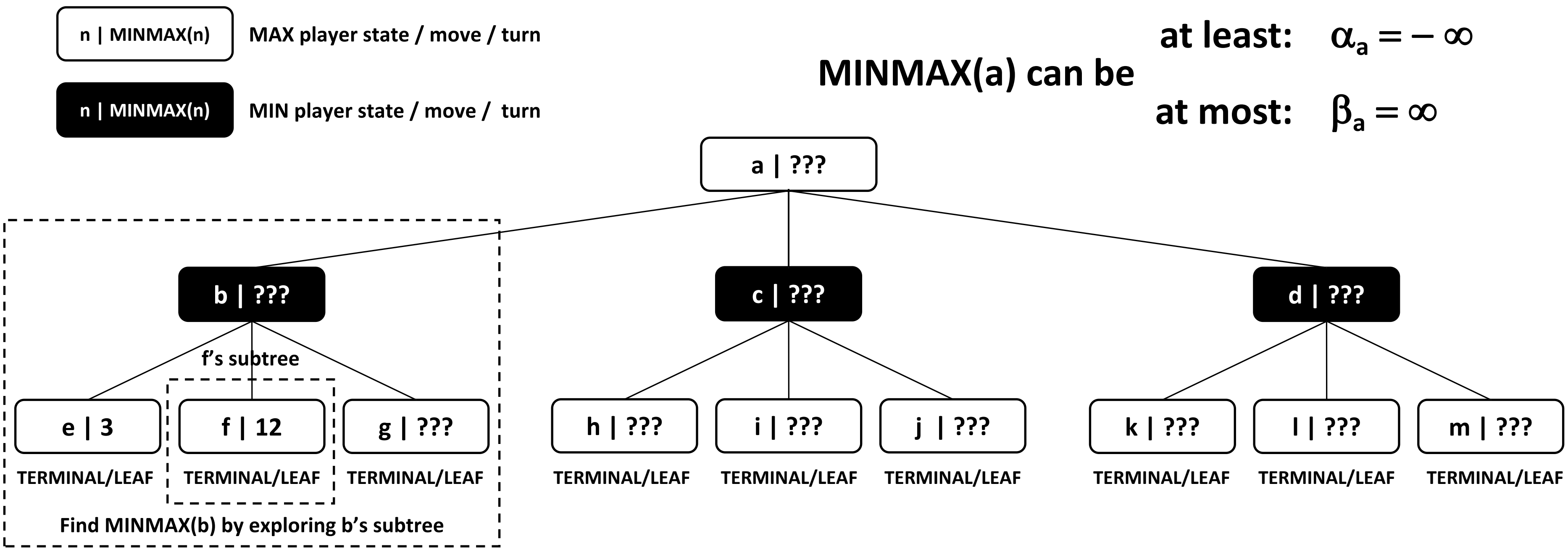
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

- **v2 < v ($3 < \infty$) → v = v2 = 3 → $\beta_b = \min(\beta_b, v) = \min(\infty, 3) = 3$**
- **v > α_a ($3 > -\infty$) → we can keep exploring b's subtree**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

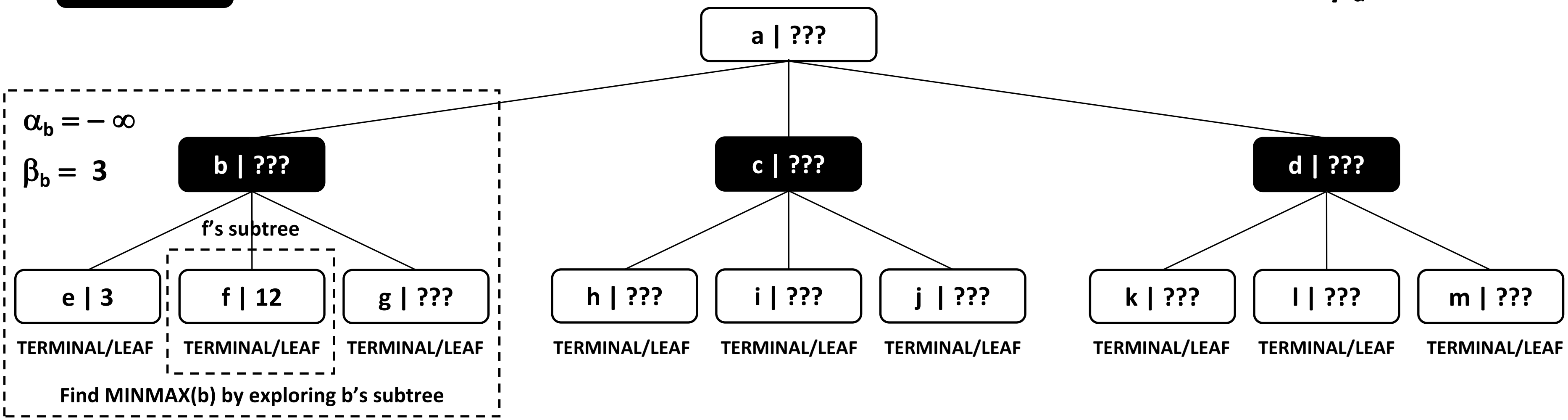
- **We need to analyze f's subtree**
- **Node f is a terminal node (Case 1) → MINMAX(f) = UTILITY(f) = 12 | v2 = MINMAX(f) = 12**

MinMax with α-β: Example

n | MINMAX(n)	MAX player state / move / turn
n | MINMAX(n)	MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = -\infty$

MINMAX(a) can be at most: $\beta_a = \infty$



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

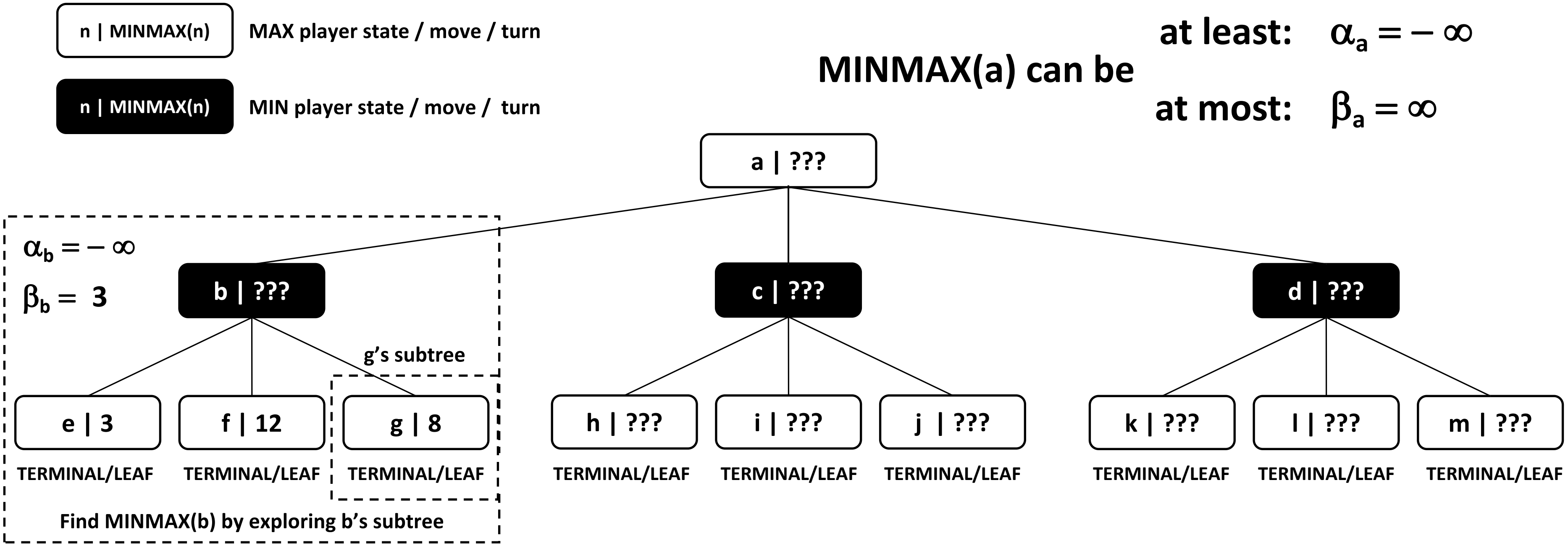
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

- **v2 > v (12 > 3) → MINMAX(f) is not "better" than MINMAX(e) → no changes**
- **v > α_a (3 > –∞) → we can keep exploring b's subtree**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

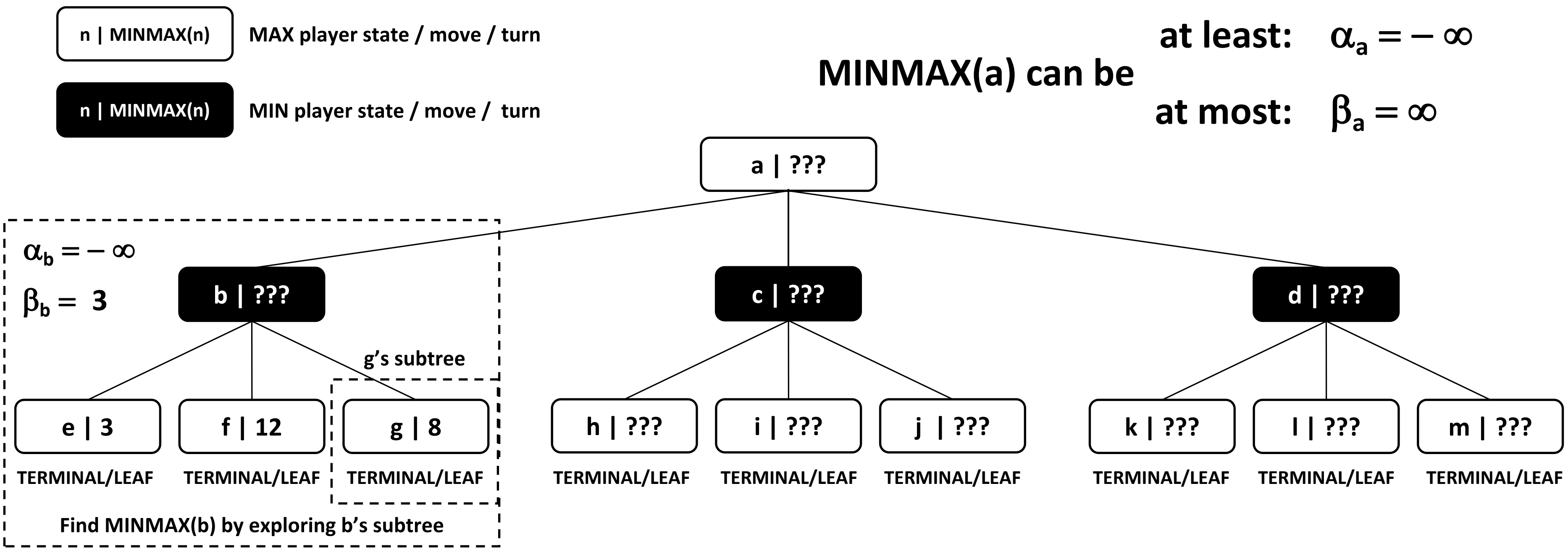
- MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN
- MAX Player's decision: not enough information yet.

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

- We need to analyze g's subtree
- Node g is a terminal node (Case 1) → MINMAX(g) = UTILITY(g) = 8 | v2 = MINMAX(g) = 8

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

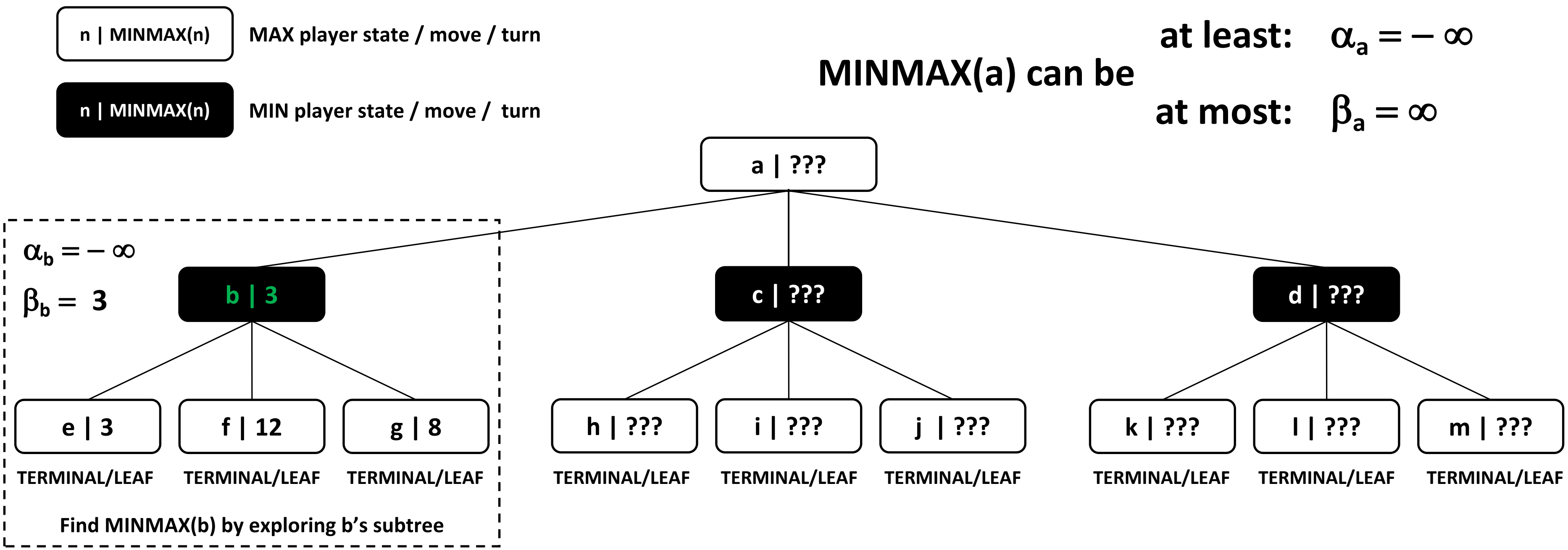
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player needs to explore b's subtree:

- **v2 > v (8 > 3) → MINMAX(g) is not "better" than MINMAX(e) → no changes**
- **v > α_a (3 > $-\infty$) → we could keep exploring b's subtree, but all b's subtrees are explored now**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

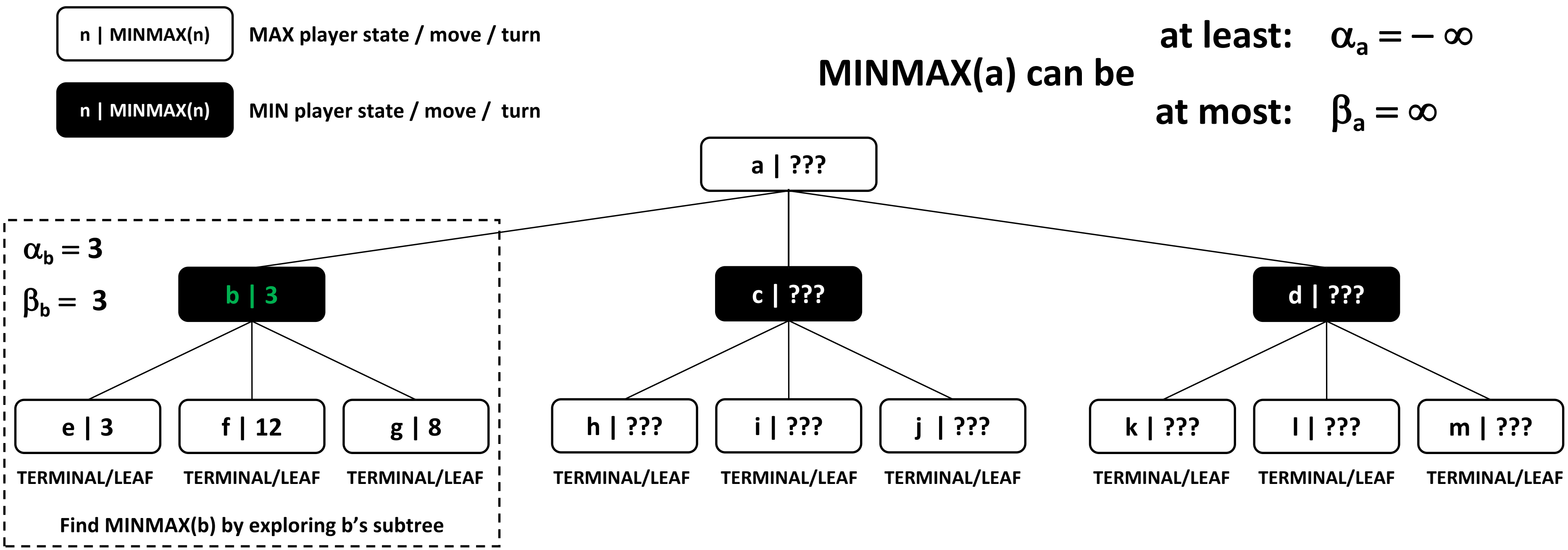
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player explored entire b's subtree:

- **MINMAX(b) = min(MINMAX(e), MINMAX(f), MINMAX(g)) = 3 (Case 2)**
- **v > α_a (3 > –∞) → we could keep exploring b's subtree, but all b's subtrees are explored now**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(???, ???, ???) → can't be established

MIN Player explored entire b's subtree:

- **We know the exact value of MINMAX(b) → α_b = 3**

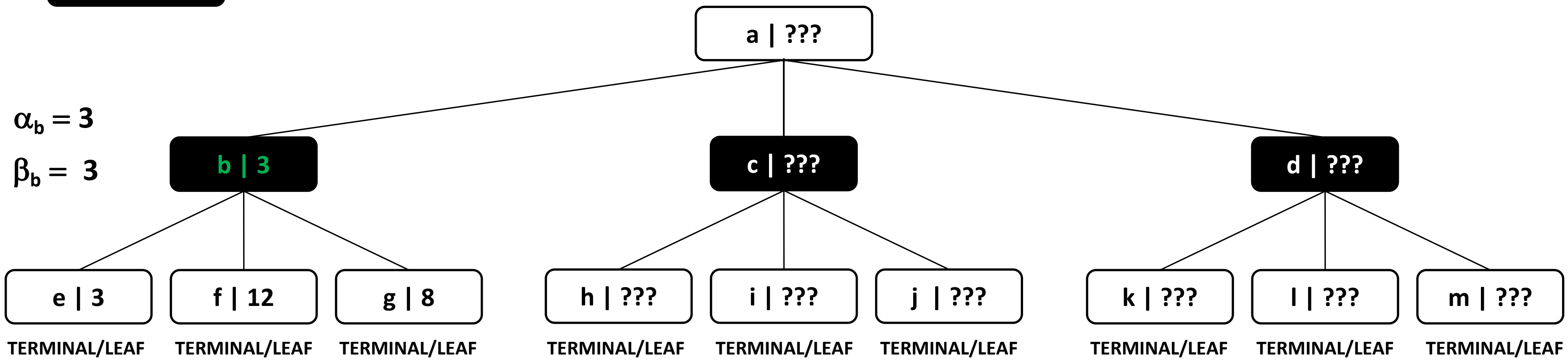
MinMax with α-β: Example

n | MINMAX(n) — MAX player state / move / turn

n | MINMAX(n) — MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = 3$

at most: $\beta_a = \infty$

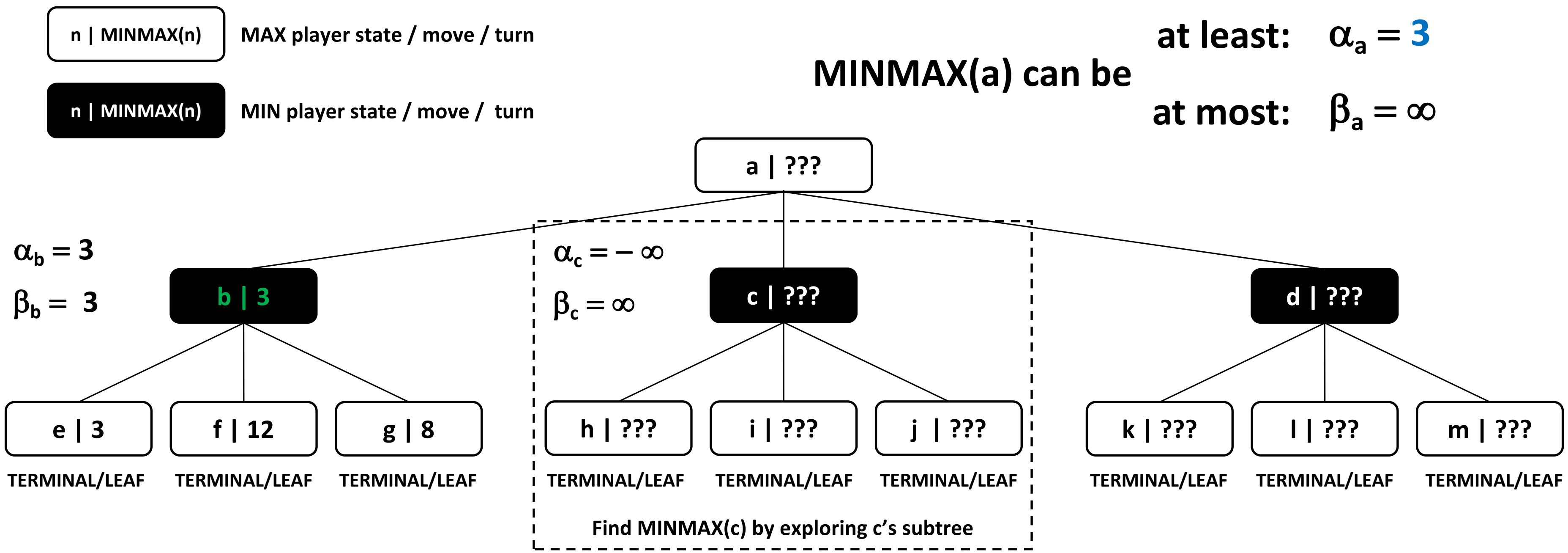


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

MINMAX(a) = max(MINMAX(b), MINMAX(c), MINMAX(d)) = max(3, ???, ???) → can't be established, but MAX Player now knows that it will be AT LEAST 3 (3 OR HIGHER)→ $\alpha_a = 3$

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

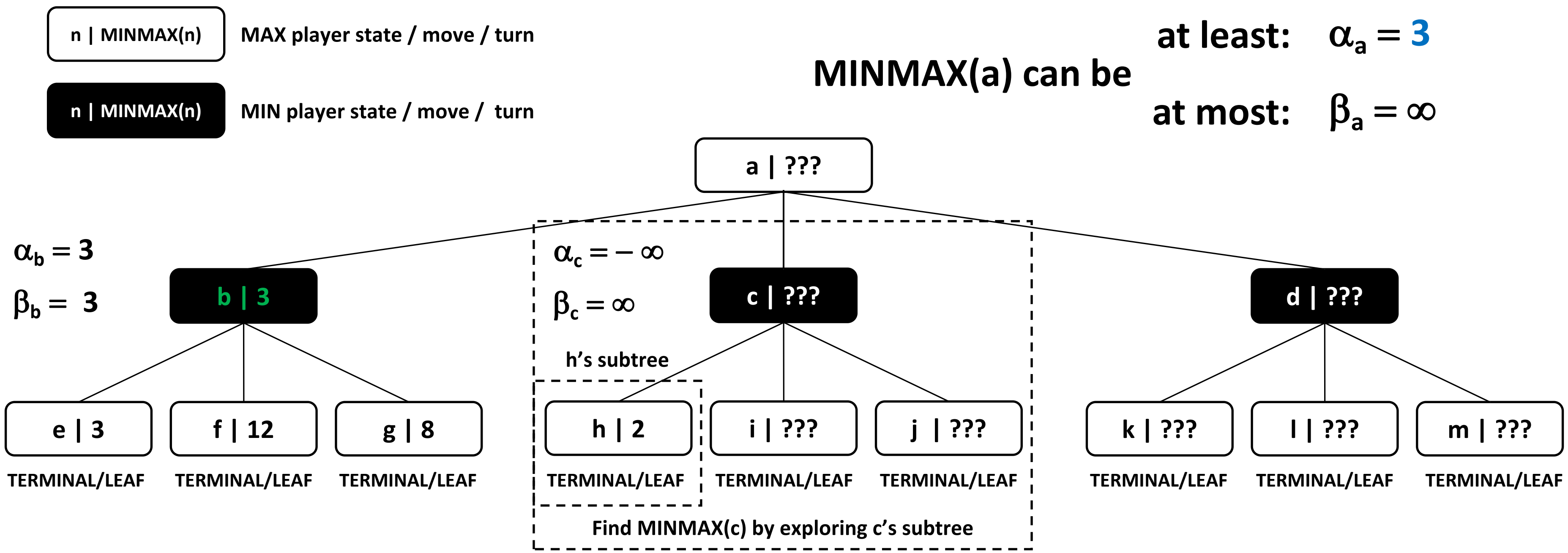
- MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN
- MAX Player's decision: not enough information yet.

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ???, ???) $\rightarrow$ can't be established

MIN Player needs to explore c's subtree:

- MIN Player (at node c) has not seen any successor MINMAX values yet $\rightarrow$ min MINMAX seen: v = ∞
- v > α_a (∞ > 3) $\rightarrow$ we can keep exploring c's subtree

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN
- MAX Player's decision: not enough information yet.

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ???, ???) → can't be established

MIN Player needs to explore c's subtree:

- We need to analyze h's subtree
- Node h is a terminal node (Case 1) → MINMAX(h) = UTILITY(h) = 2 | v2 = MINMAX(h) = 2

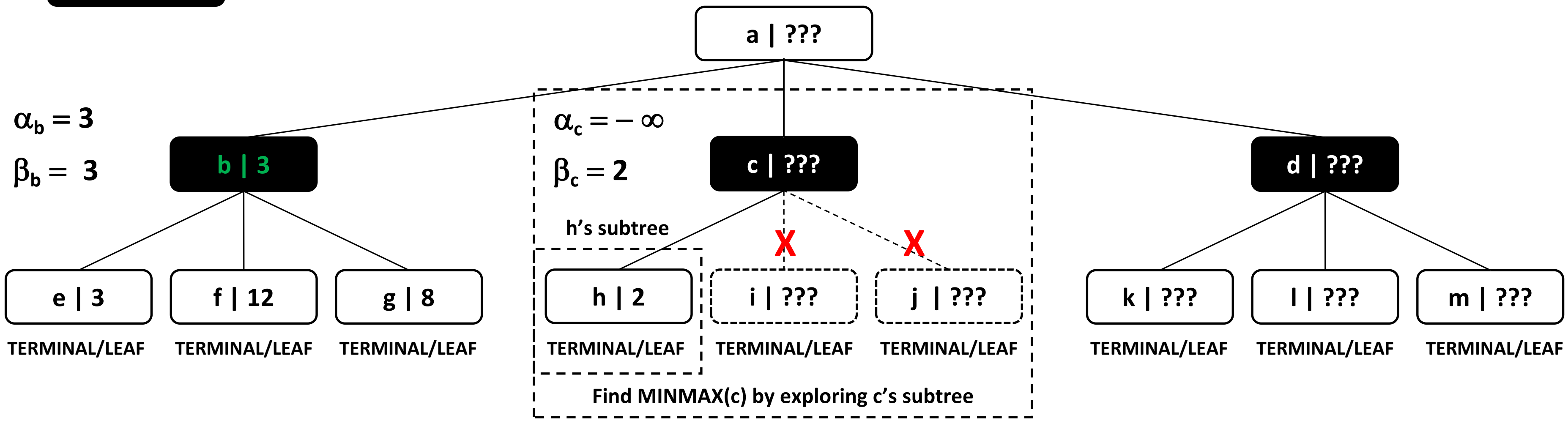
MinMax with α-β: Example

n | MINMAX(n) — MAX player state / move / turn

n | MINMAX(n) — MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = 3$

MINMAX(a) can be at most: $\beta_a = \infty$



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

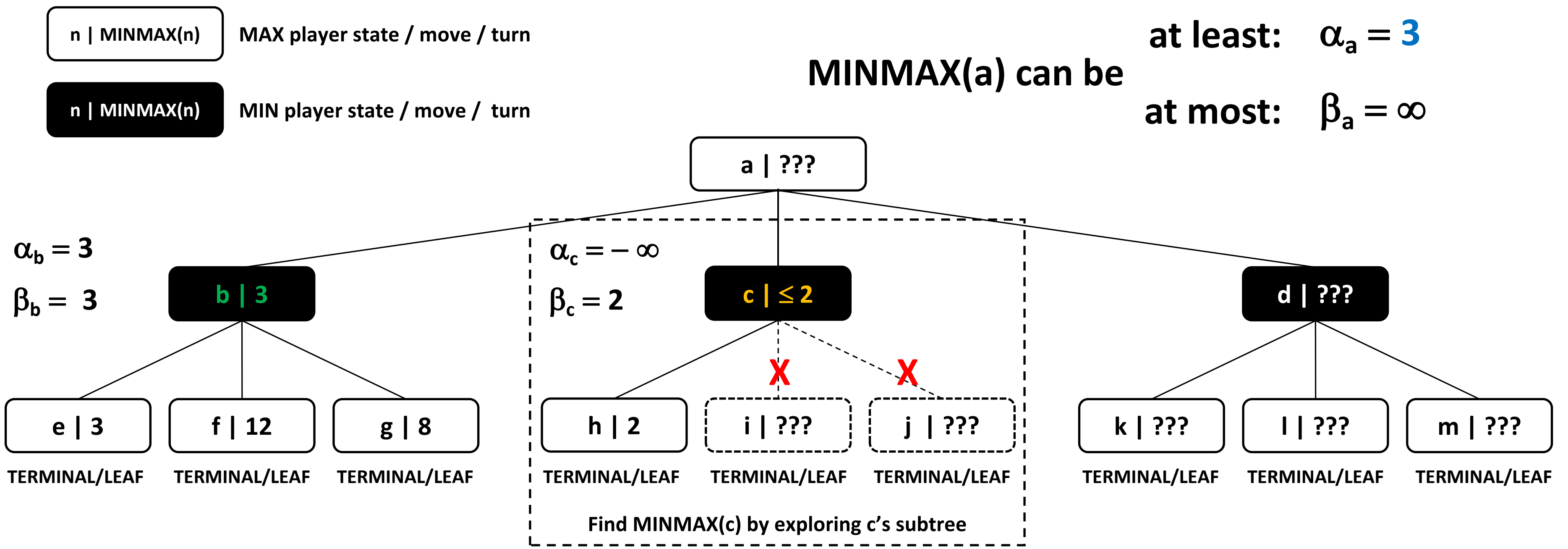
- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ???, ???) → can't be established

MIN Player needs to explore c's subtree:

- **v2 < v ($2 < \infty$) → v = v2 = 2 → $\beta_c = \min(\beta_c, v) = \min(\infty, 2) = 2$**
- **v < α_a (2 < 3) → we cannot keep exploring c's subtree → prune remaining branches**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

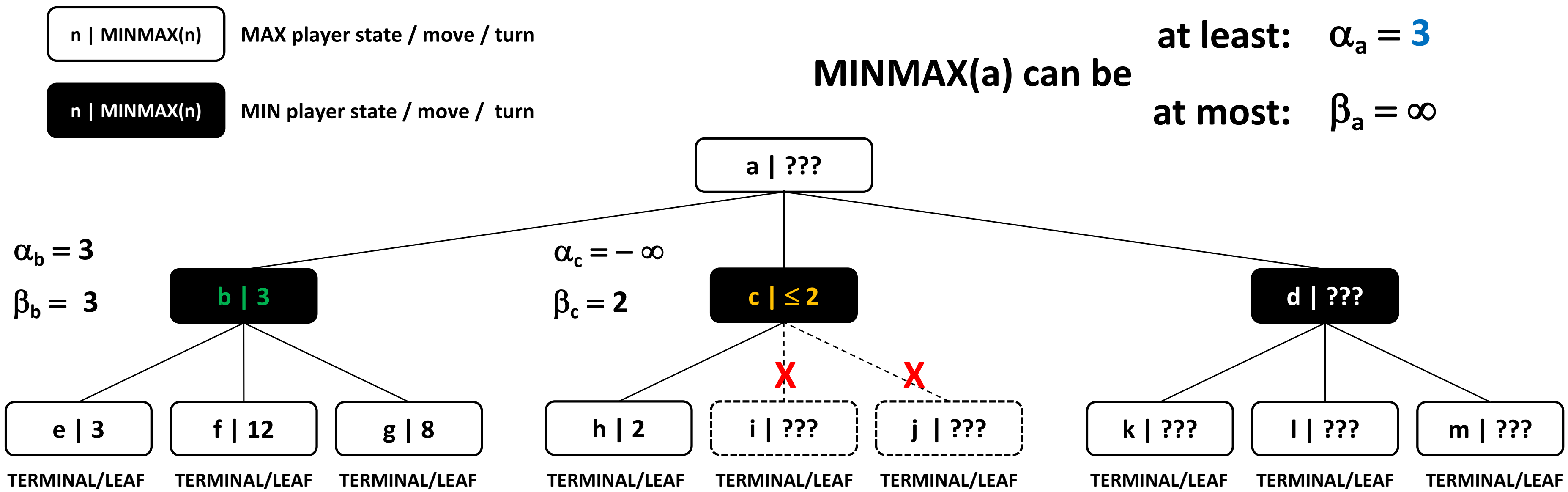
- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ???, ???) → can't be established

MIN Player explored c's subtree as far as it was necessary:

- **We know that MINMAX(c) ≤ 2**

MinMax with α-β: Example

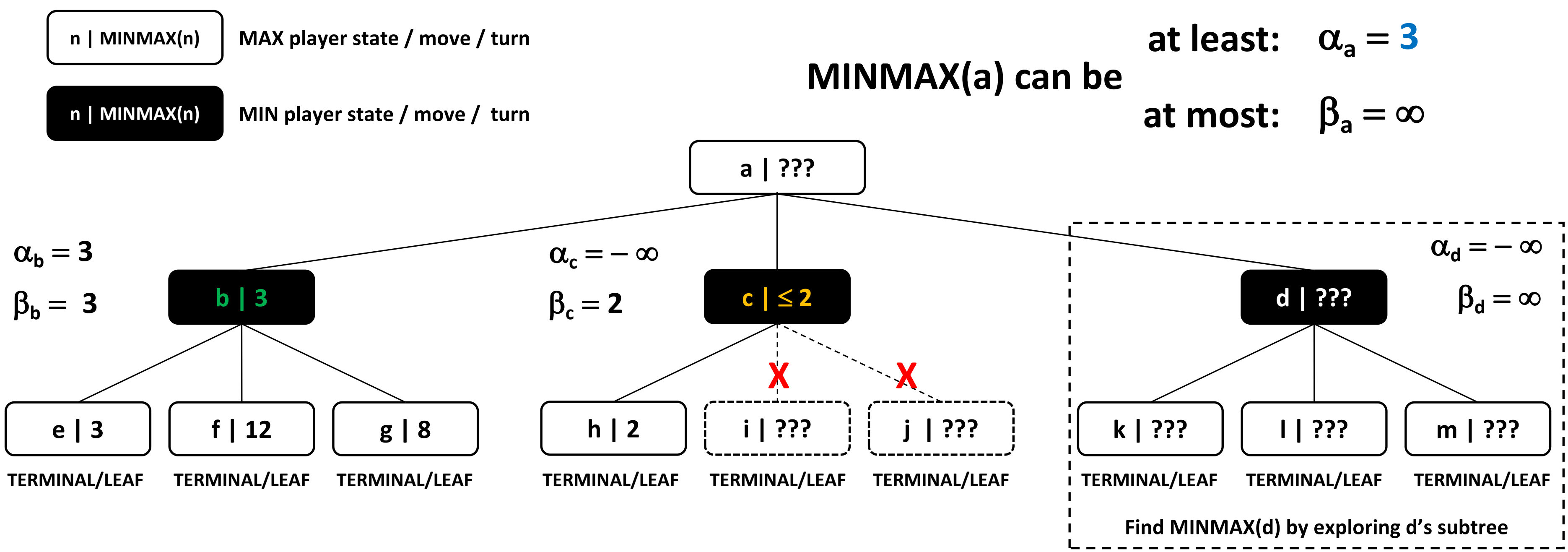


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ???) → can't be established

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

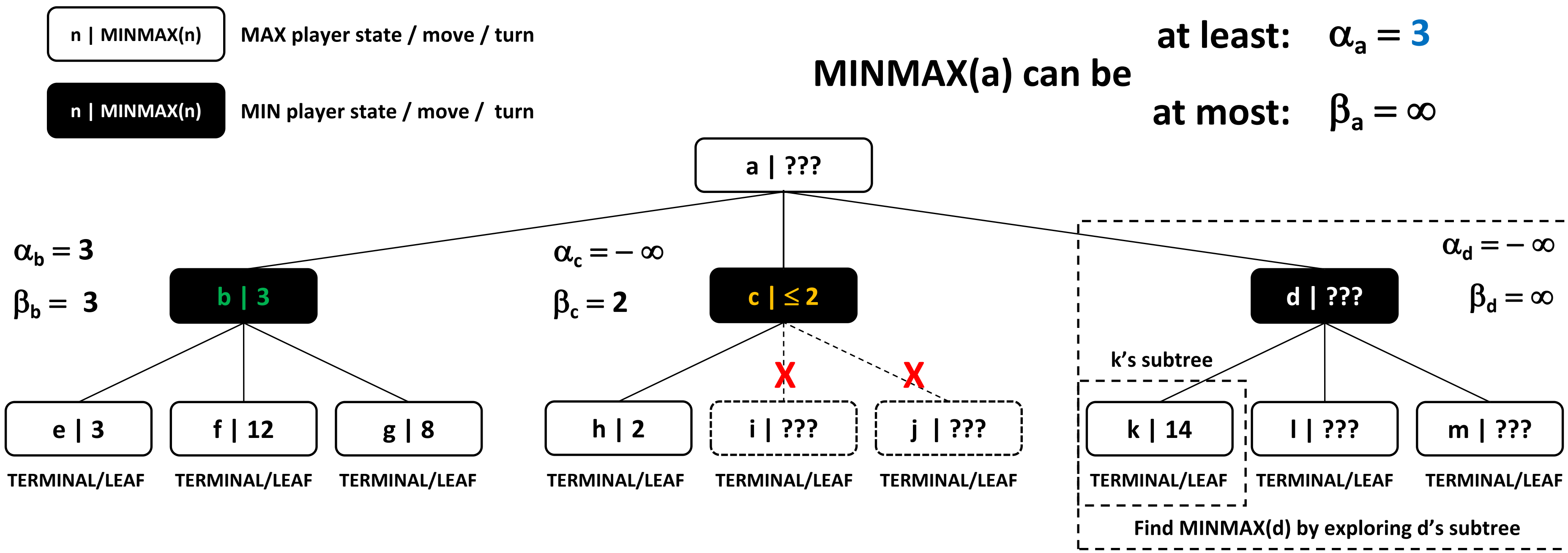
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ???) → can't be established

MIN Player needs to explore d's subtree:

- **MIN Player (at node d) has not seen any successor MINMAX values yet → min MINMAX seen: v = ∞**
- **$v > \alpha_a$ (∞ > 3) → we can keep exploring d's subtree**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ???) → can't be established

MIN Player needs to explore d's subtree:

- **We need to analyze k's subtree**
- **Node k is a terminal node (Case 1) → MINMAX(k) = UTILITY(k) = 14 | v2 = MINMAX(k) = 14**

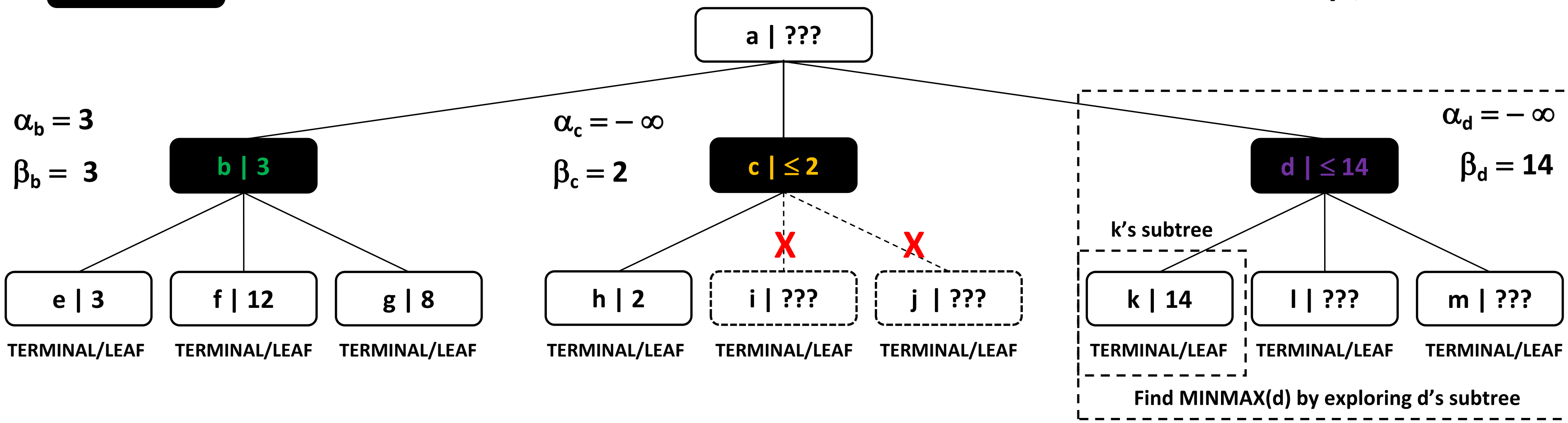
MinMax with α-β: Example

n | MINMAX(n) — MAX player state / move / turn

n | MINMAX(n) — MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = 3$

MINMAX(a) can be at most: $\beta_a = \infty$



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ???) → can't be established

MIN Player needs to explore d's subtree:

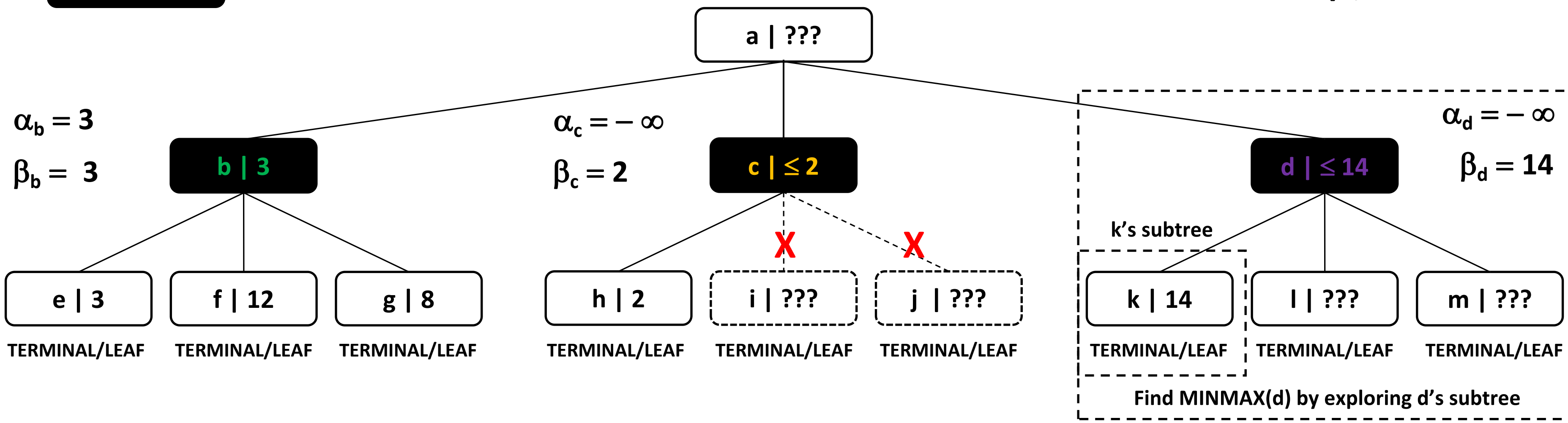
- **v2 < v (14 < ∞) → v = v2 = 14 → $\beta_d = \min(\beta_d, v) = \min(\infty, 14) = 14$**
- **$v > \alpha_a$ (14 > 3) → we can keep exploring d's subtree → we also know that MINMAX(d) ≤ 14**

MinMax with α-β: Example

n | MINMAX(n)	MAX player state / move / turn
n | MINMAX(n)	MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = 3$

at most: $\beta_a = 14$



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

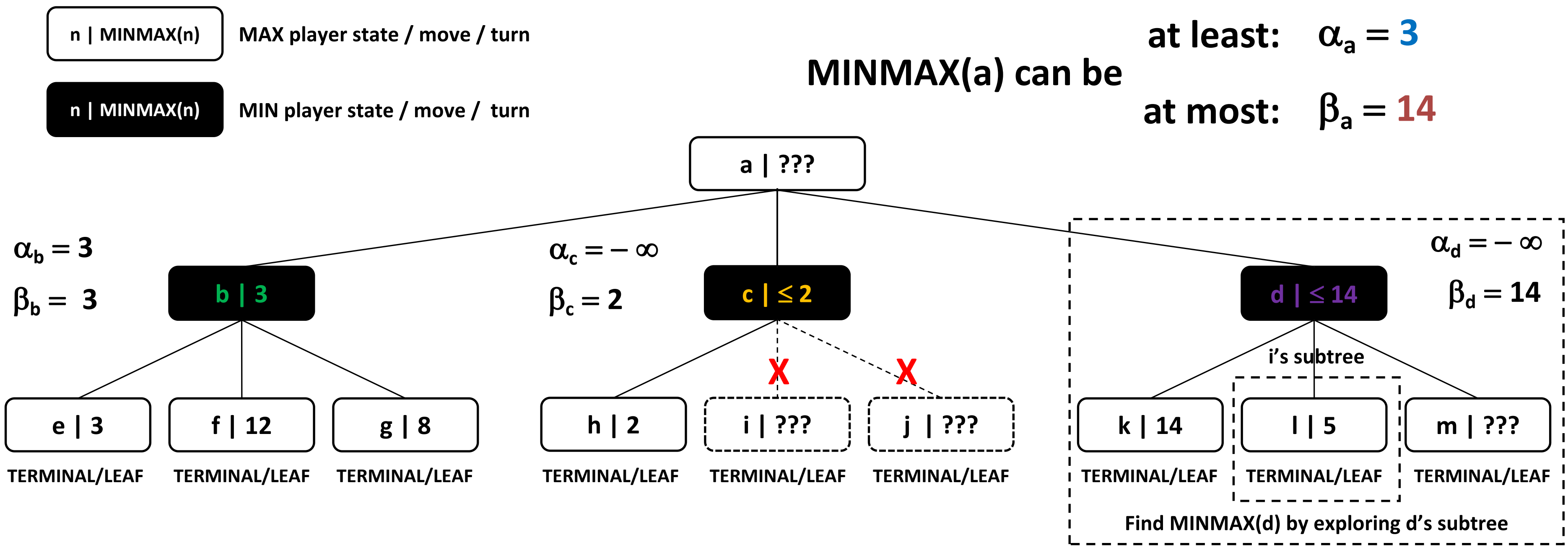
- MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 14
- MAX Player's decision: not enough information yet.

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ≤ 14) → can't be established

MIN Player needs to explore d's subtree:

- we know that MINMAX(d) ≤ 14 → this tells us that MINMAX(a) cannot be > 14 → β_a = 14

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

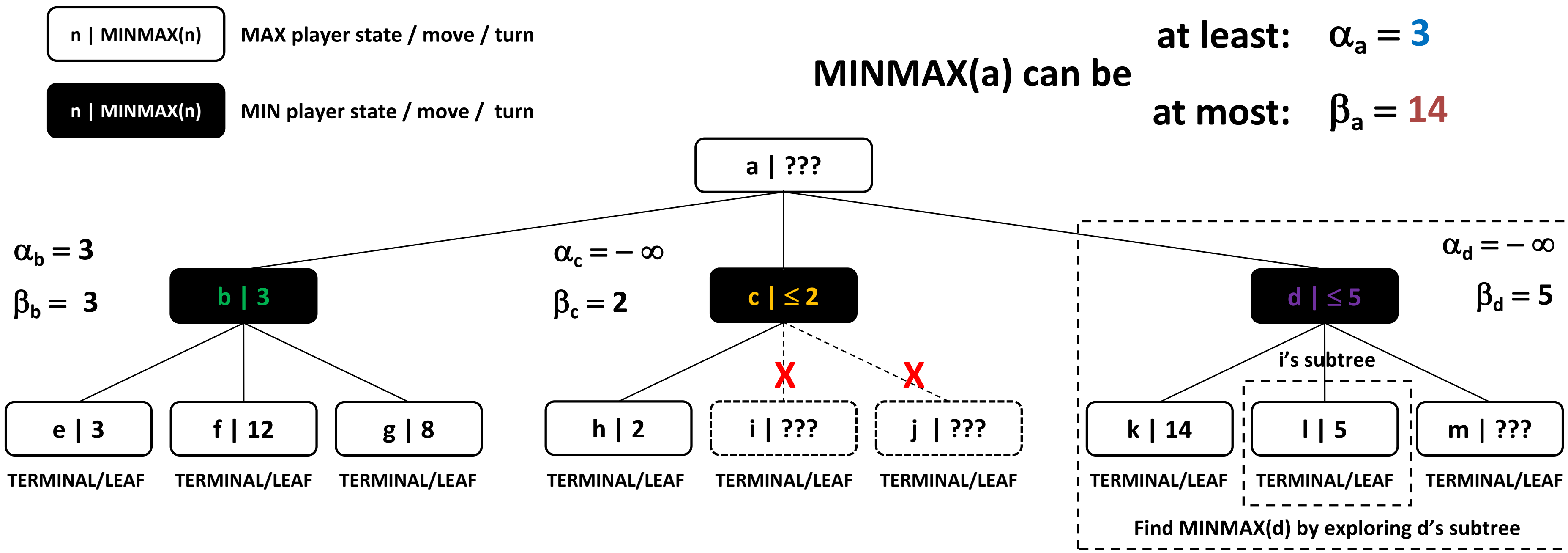
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 14**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ≤ 14) → can't be established

MIN Player needs to explore d's subtree:

- **We need to analyze l's subtree**
- **Node l is a terminal node (Case 1) → MINMAX(l) = UTILITY(l) = 5 | v2 = MINMAX(l) = 5**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 14**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ≤ 14) → can't be established

MIN Player needs to explore d's subtree:

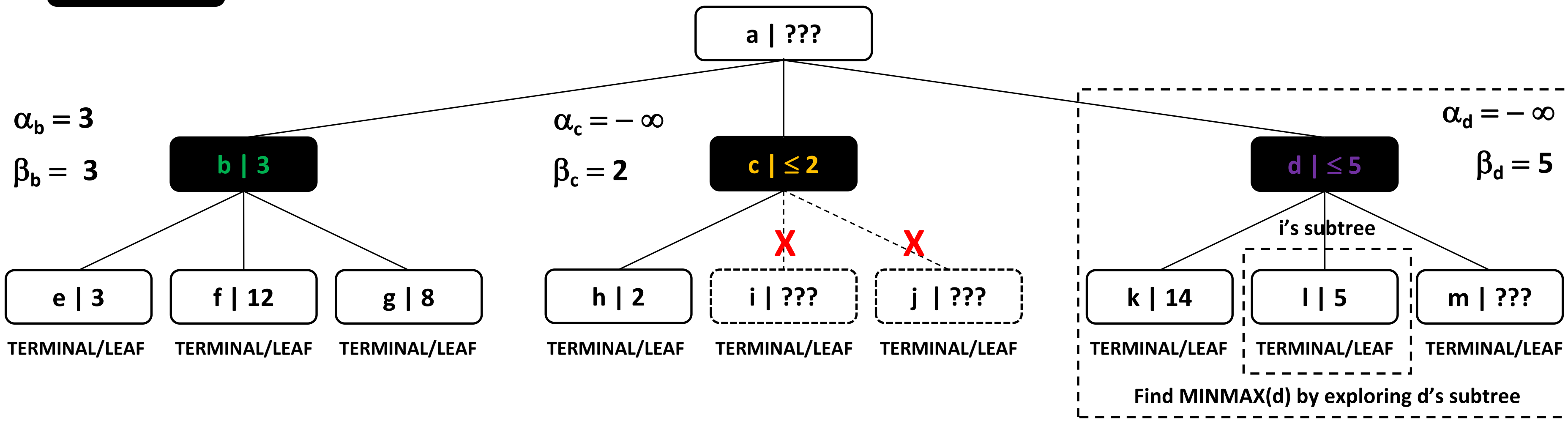
- **v2 < v (5 < 14) → v = v2 = 5 → β_d = min(β_d, v) = min(∞, 5) = 5**
- **v > α_a (5 > 3) → we can keep exploring d's subtree → we also know that MINMAX(d) ≤ 5**

MinMax with α-β: Example

n | MINMAX(n)	MAX player state / move / turn
n | MINMAX(n)	MIN player state / move / turn

MINMAX(a) can be at least: $\alpha_a = 3$

at most: $\beta_a = 5$



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

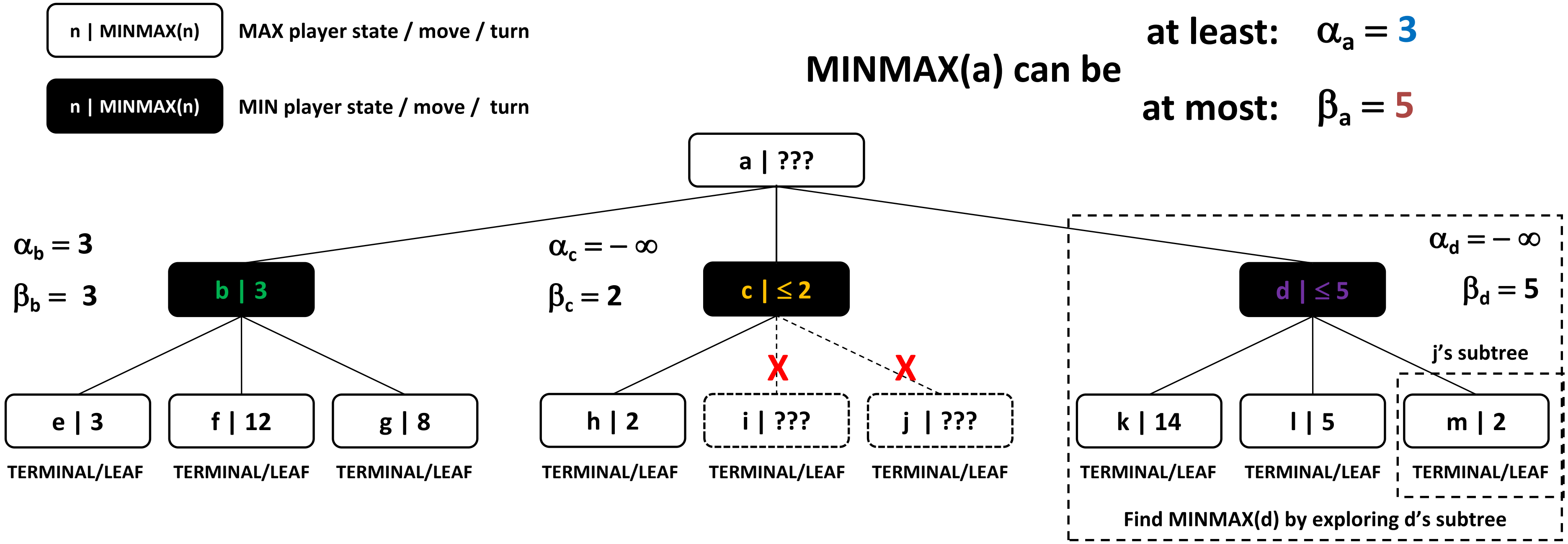
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 5**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ≤ 5) → can't be established

MIN Player needs to explore d's subtree:

- **we know that MINMAX(d) ≤ 5 → this tells us that MINMAX(a) cannot be > 5 → β_a = 5**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

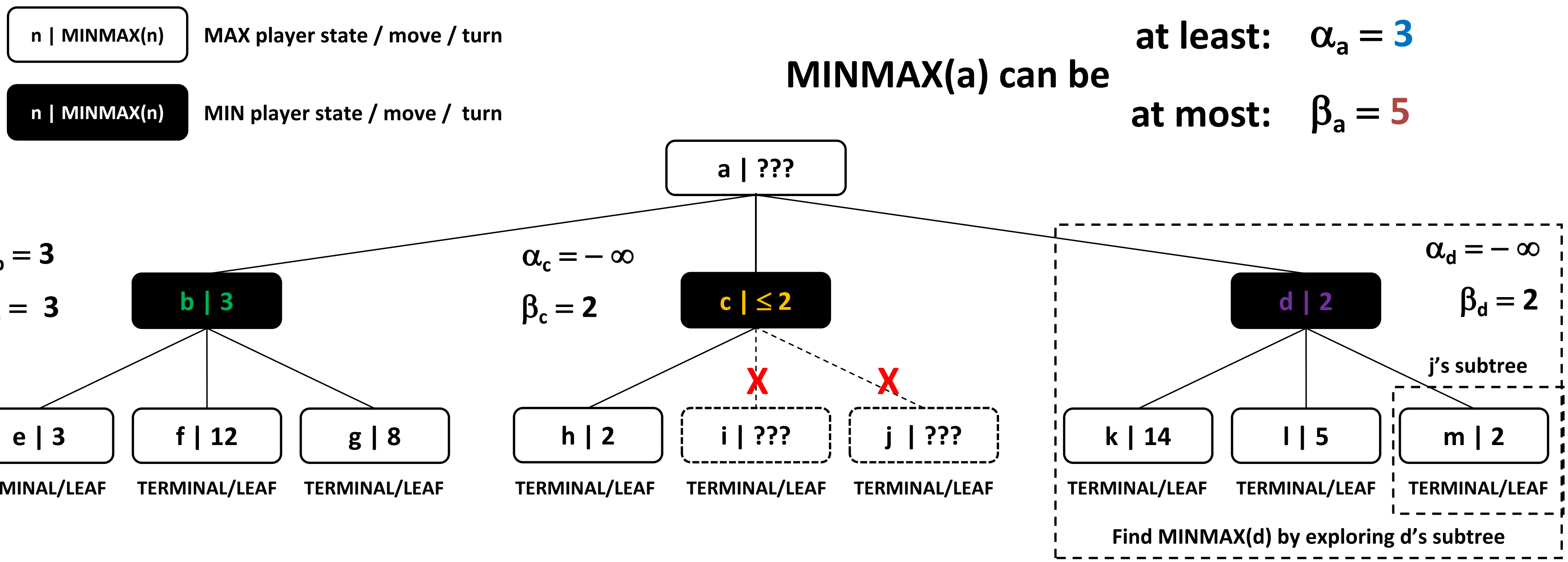
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 5**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ≤ 5) → can't be established

MIN Player needs to explore d's subtree:

- **We need to analyze m's subtree**
- **Node m is a terminal node (Case 1) → MINMAX(m) = UTILITY(m) = 2 | v2 = MINMAX(m) = 2**

MinMax with α-β: Example



MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

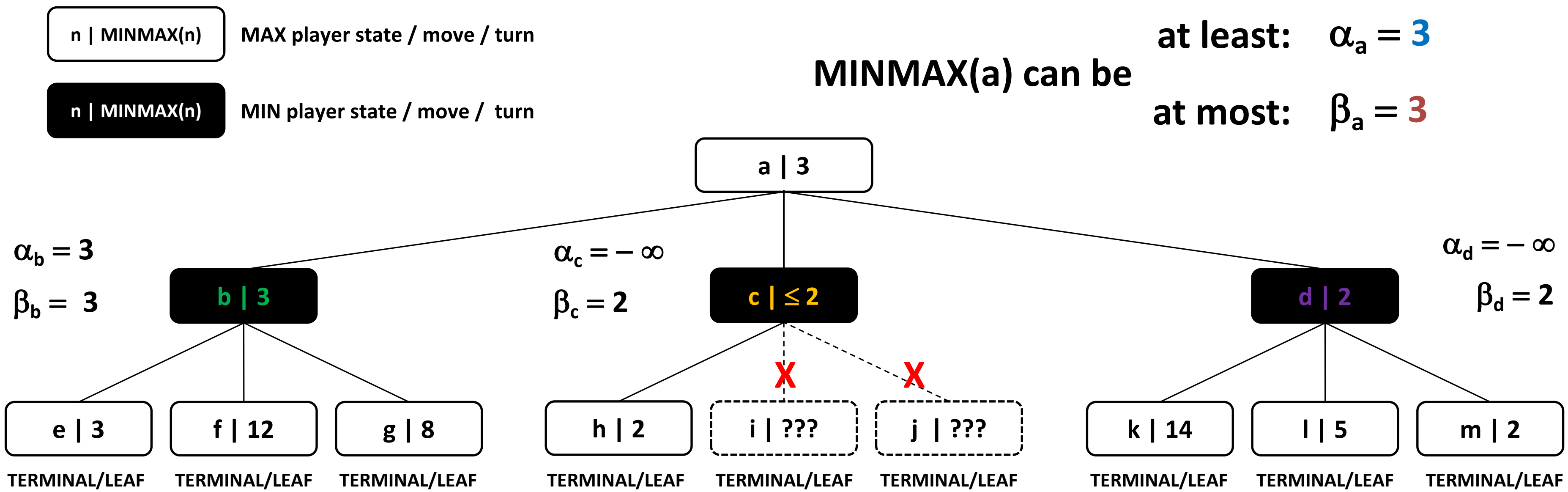
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 5**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ≤ 2, ≤ 5) → can't be established

MIN Player needs to explore d's subtree:

- **v2 < v (2 < 5) → v = v2 = 2 → β_d = min(β_d, v) = min(∞, 2) = 2**
- **v < α_a (2 < 3) → we cannot keep exploring d's subtree → we also know that MINMAX(d) = 2**

MinMax with α-β: Example



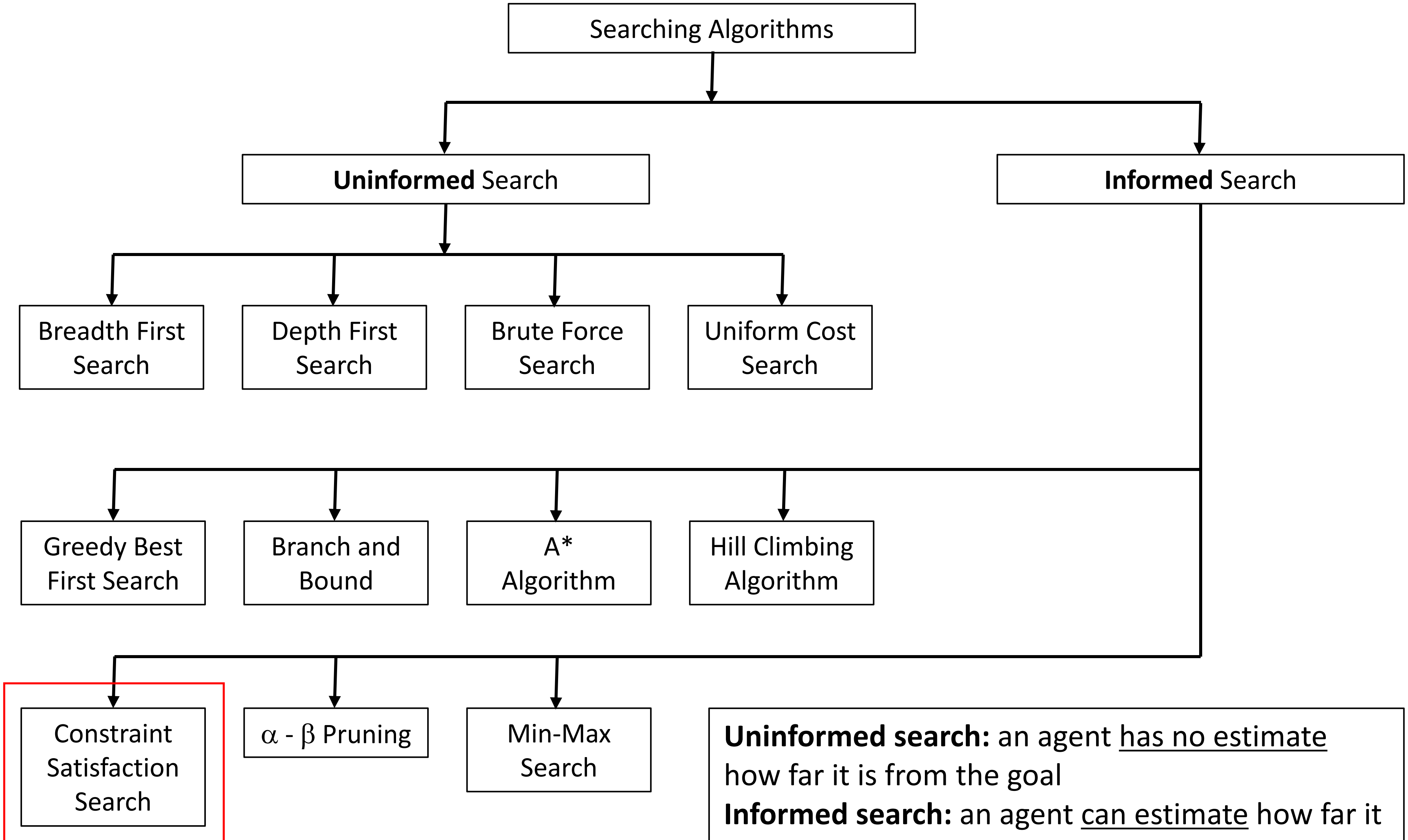
Find MINMAX(d) by exploring d's subtree

MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **MINMAX(b) = 3 | MINMAX(c) = $\leq$ 2 | MINMAX(d) = 2**
- **MAX Player's decision: choose move b, because:**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, $\leq$ 2, 2) = 3
- **Since we know MINMAX(a), we can update β_a for completeness $\rightarrow$ β_a = 3**

Selected Searching Algorithms



Uninformed search: an agent has no estimate how far it is from the goal
Informed search: an agent can estimate how far it is from the goal

Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) consists of three components:

- **a set of variables** $X = \{X_1, ..., X_n\}$
- **a set of domains** $D = \{D_1, ..., D_n\}$
- **a set of constraints** C **that specify allowable combinations of values**

- **A domain** D_i **is a set of allowable values** $\{v1, ..., vk\}$ **for variable** X_i
- **A constraint** C_j **is a** $\langle scope, relation \rangle$ **pair, for example** $\langle (X1, X2), X1 > X2 > \rangle$

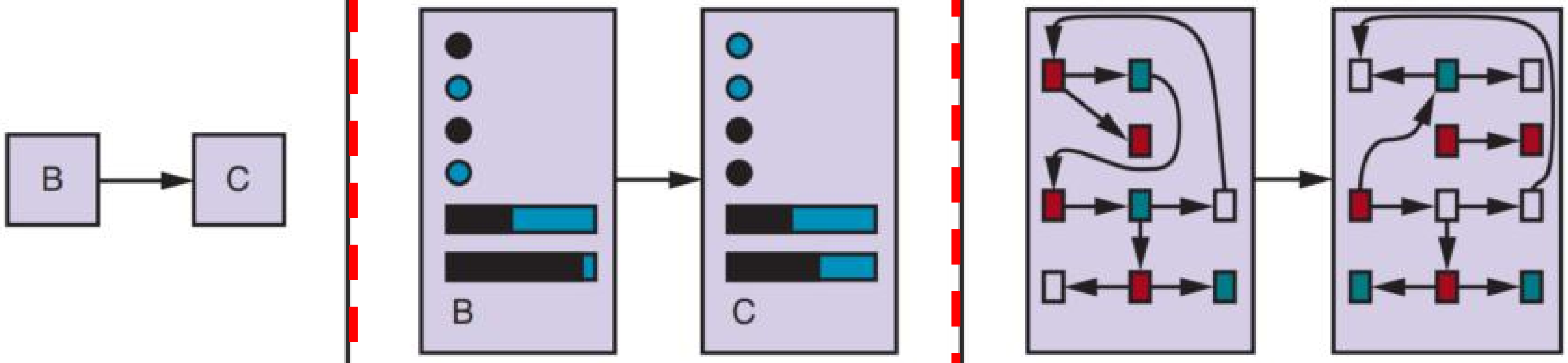
Constraint Satisfaction Problem

The goal is to find an assignment (variable = value):

$$\{X_1 = v_1, \ldots, X_n = v_n\}$$

- **If NO constraints violated: consistent assignment**
- **If ALL variables have a value: complete assignment**
- **If SOME variables have NO value: partial assignment**
- **SOLUTION: consistent and complete assignment**
- **PARTIAL SOLUTION: consistent and partial assignment**

State Representations



(a) Atomic | (b) Factored | (c) Structured

- Searching
- Hidden Markov models
- Markov decision process
- Finite state machines

- Constraint satisfaction algorithms
- Propositional logic
- Planning
- Bayesian algorithms
- Some machine learning algorithms

- Relational database algorithms
- First-order logic
- First-order probability models
- Natural language understanding (some)

CSP Example: Map Coloring

Problem:

Color this map in a way that no two neighbors have same color

Variables:

X = {WA, NT, Q, NSW, V, SA, T}

Variable Domains:

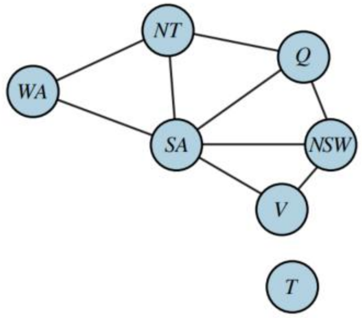
D_{WA} = {RED, GREEN, BLUE}
D_{NT} = {RED, GREEN, BLUE}
D_{Q} = {RED, GREEN, BLUE}
D_{NSW} = {RED, GREEN, BLUE}
D_{V} = {RED, GREEN, BLUE}
D_{SA} = {RED, GREEN, BLUE}
D_{T} = {RED, GREEN, BLUE}

Constraints (Rules):

- **Neighboring regions have to have DISTINCT colors:**

**CONSTRAINTS = C = {SA ≠ WA, SA ≠ NT, SA ≠ Q,
SA ≠ NSW, SA ≠ V, WA ≠ NT, NT ≠ Q, Q ≠ NSW, NSW ≠ V}**

Constraint Graph:



CSP Example: Sudoku (3x3 for now)

Problem:

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$

Variables:

$X = \{x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{2,2}, x_{2,3}, x_{3,1}, x_{3,2}, x_{3,3}\}$

Variable Domains:

$D_{x1,1} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x1,2} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x1,3} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x2,1} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x2,2} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x2,3} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x3,1} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x3,2} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$D_{x3,3} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

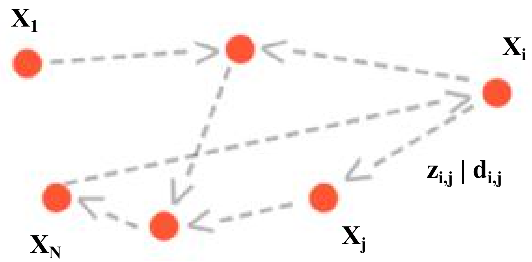
Constraints (Rules):

- **Each value {1, 2, 3, 4, 5, 6, 7, 8, 9} can appear EXACTLY once:**

$\text{CONSTRAINTS} = C = \{x_{1,1} \neq x_{1,2}, x_{1,1} \neq x_{1,3}, x_{1,1} \neq x_{2,1}, x_{1,1} \neq x_{2,2}, x_{1,1} \neq x_{2,3}, x_{1,2} \neq x_{1,3}, x_{1,2} \neq x_{2,1}, x_{1,2} \neq x_{2,2}, x_{1,2} \neq x_{2,3}, x_{1,2} \neq x_{3,1}, x_{1,2} \neq x_{3,2}, x_{1,3} \neq x_{2,1}, x_{1,3} \neq x_{2,2}, x_{1,3} \neq x_{2,3}, x_{1,3} \neq x_{3,1}, x_{1,3} \neq x_{3,2}, x_{1,3} \neq x_{3,3}, x_{2,1} \neq x_{2,2}, x_{2,1} \neq x_{2,3}, x_{2,1} \neq x_{3,1}, x_{2,1} \neq x_{3,2}, x_{2,1} \neq x_{3,3}, x_{2,2} \neq x_{2,3}, x_{2,2} \neq x_{3,1}, x_{2,2} \neq x_{3,2}, x_{2,2} \neq x_{3,3}, x_{2,3} \neq x_{3,1}, x_{2,3} \neq x_{3,2}, x_{2,3} \neq x_{3,3}, x_{3,1} \neq x_{3,2}, x_{3,1} \neq x_{3,3}, x_{3,2} \neq x_{3,3}\}$

CSP Example: Traveling Salesman

Problem:



There are:

- N cities (vertices)
- N(N-1) links (edges)
- Each link has some positive cost d
- Total path (tour) cost is COST

Variables:

$Z = \{z_{1,2}, z_{1,3}, \ldots, z_{N-1,N}\}$
$D = \{d_{1,2}, d_{1,3}, \ldots, d_{N-1,N}\}$

Variable Domains:

$D_{zi,j} = \{\text{traveled}, \text{notTraveled}\}$

or better:

$D_{zi,j} = \{1, 0\}$

$D_{di,j} = R_+$

Constraints (Rules):

- Exit each city EXACTLY once:

$$\sum_{j=1}^{N} z_{i,j} = 1$$

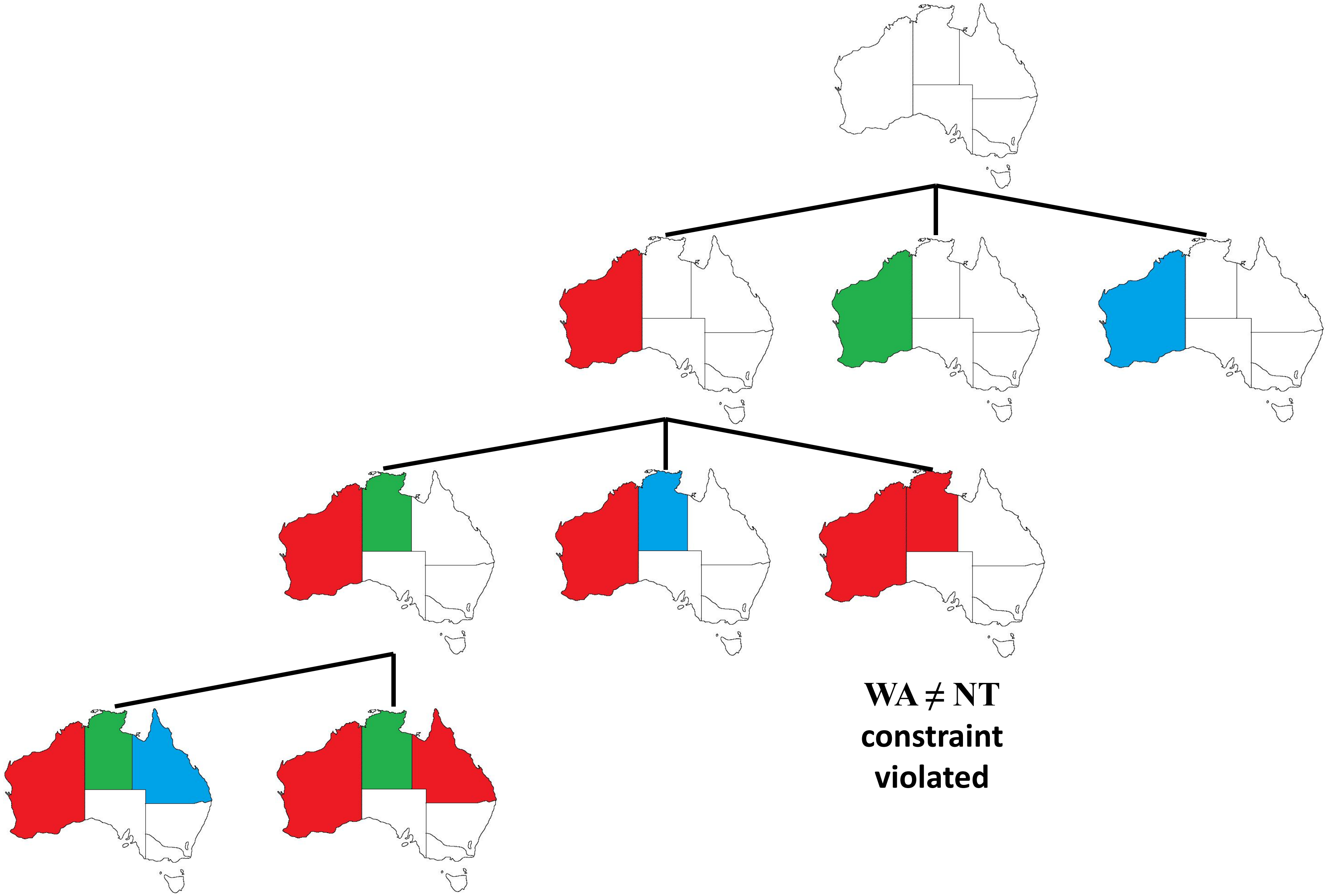
- Enter each city EXACTLY once:

$$\sum_{i=1}^{N} z_{i,j} = 1$$

- Cost of tour is at most C:

$$\sum_{i=1}^{N} \sum_{j=1}^{N} z_{i,j} d_{i,j} \leq COST$$

CSP as a Tree Search Problem



CSP as a Tree Search Problem