CS 581

Advanced Artificial Intelligence

January 24, 2024

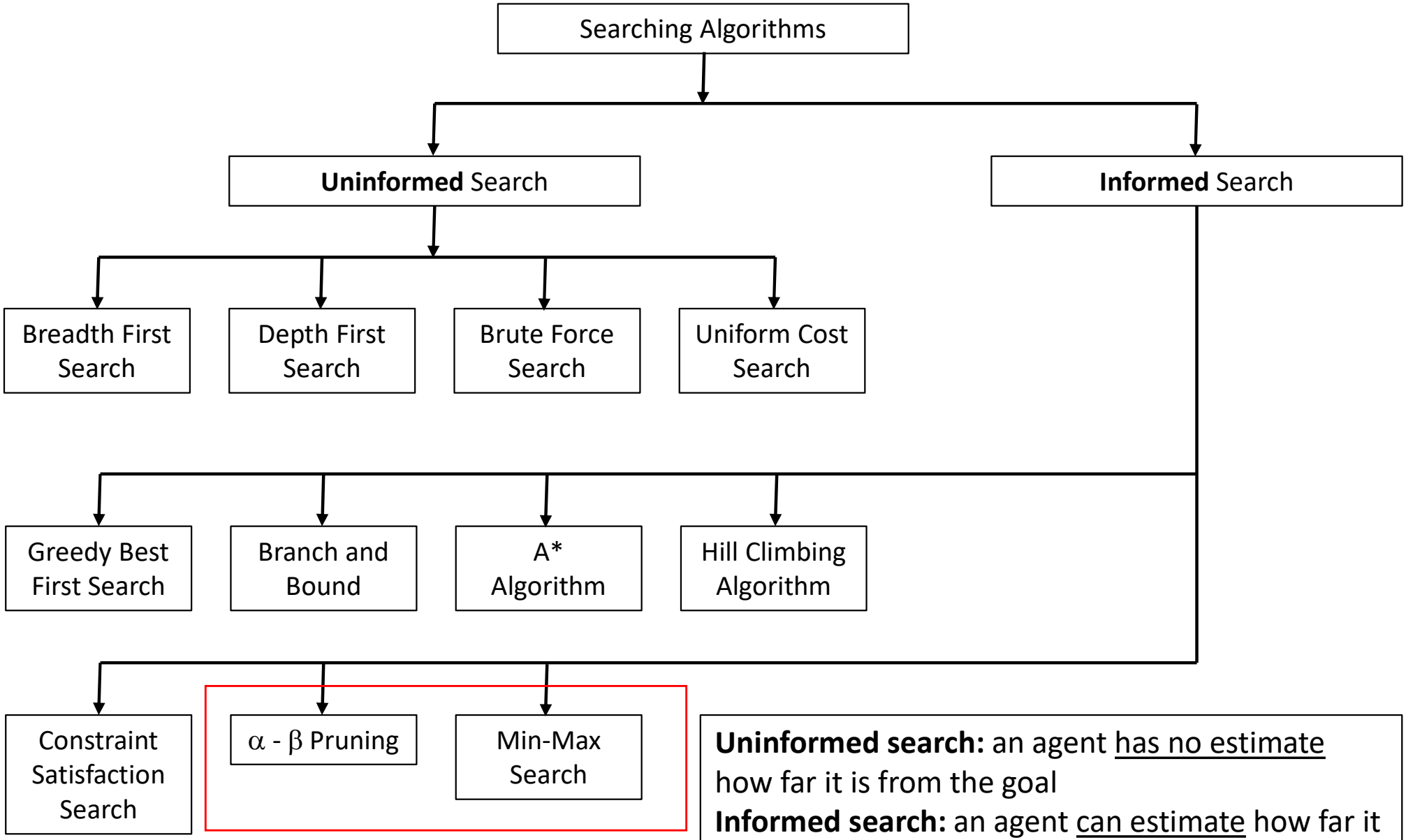
Announcements / Reminders

- Please follow the Week 02 To Do List instructions (if you haven't already)
- Written Assignment #01: to be posted soon
- Programming Assignment #01: to be posted soon

Plan for Today

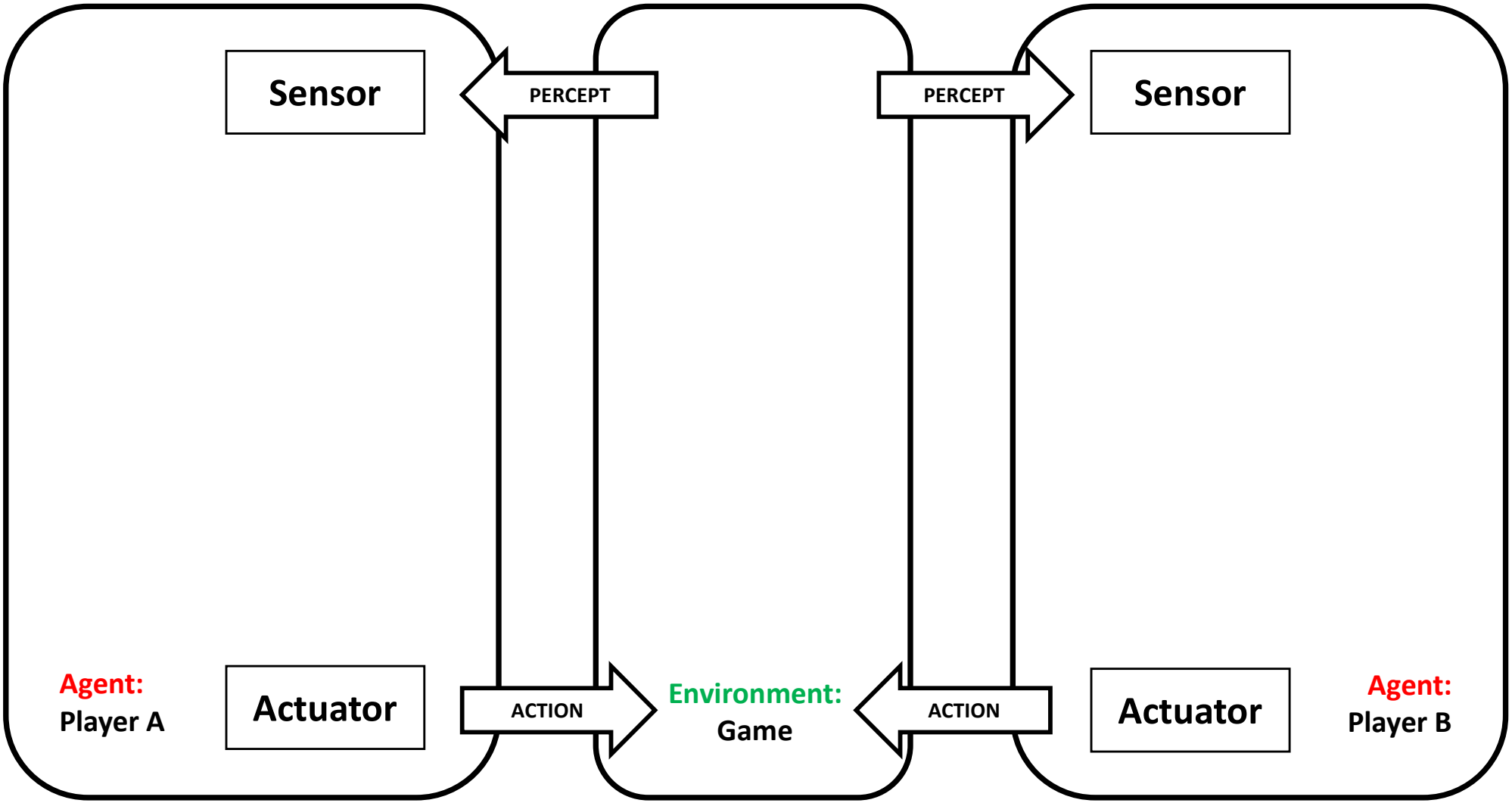
- Local Search Problems [DELAYED]
- Problem Solving: Adversarial Search
- Adversarial Search: MinMax / α-β Pruning
- Constraint Satisfaction Problems (CSP)

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

Two-player Games



Perfect Information Zero Sum Games

- Perfect information = fully observable
- Multiagent: number of players is 2 or more
- Multiagent: agents are competitve
- Zero-sum: “winner takes all”
- Examples:
  - Tic Tac Toe
  - Chess

Two Player Games: Env Assumptions

Works with a "Simple Environment":

- Fully observable
- ~~Single agent~~ Mulitagent (competitive!)
- Deterministic
- Static
- Episodic / sequential
- Discrete
- Known to the agent

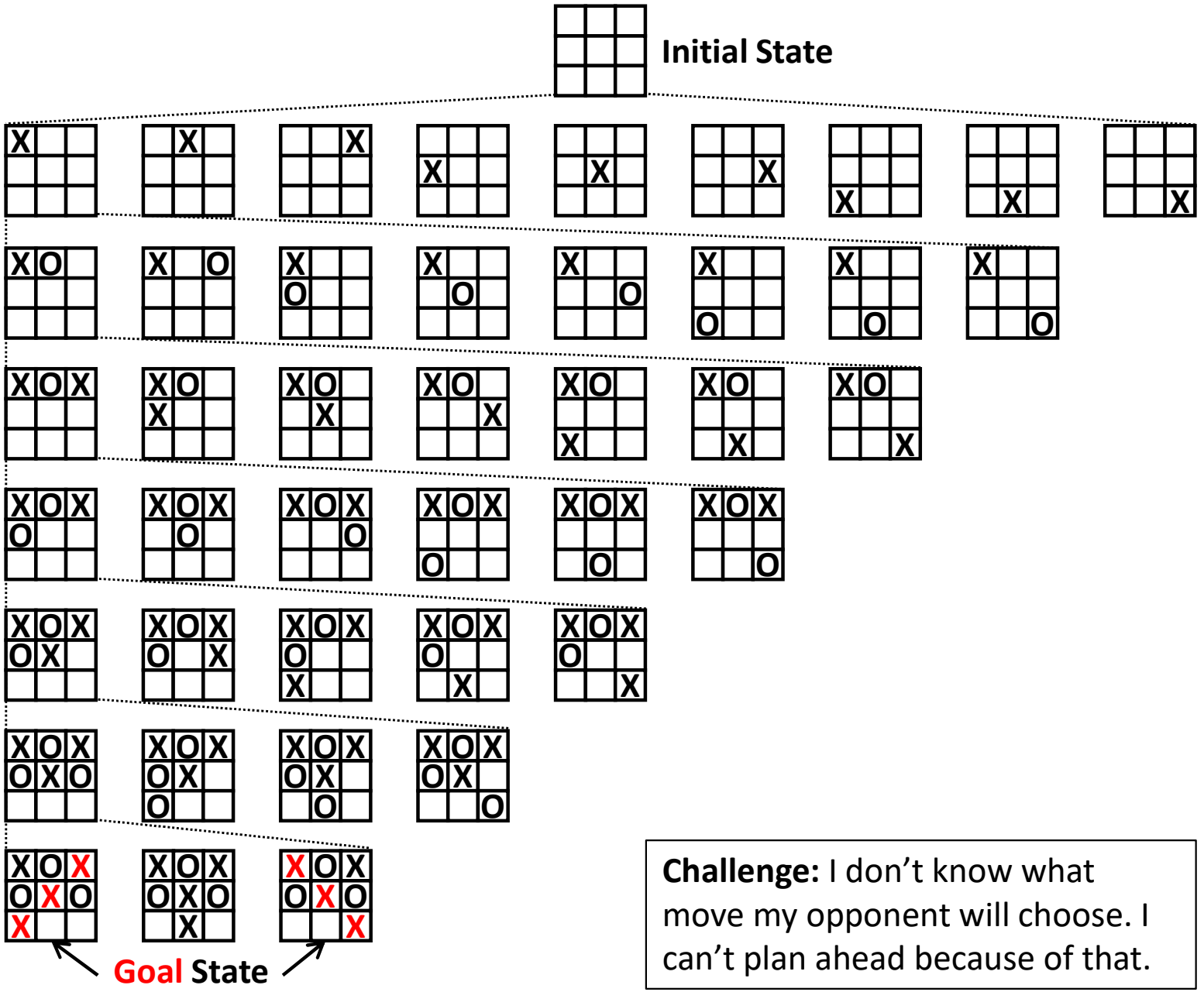
Defining Zero Sum Game Problem

- Define a set of possible states: State Space
- Specify how will you track Whose Move / Turn it is
- Specify Initial State
- Specify Goal State(s) (there can be multiple)
- Define a FINITE set of possible Actions (legal moves) for EACH state in the State Space
- Come up with a Transition Model which describes what each action does
- Come up with a Terminal Test that verifies if the game is over
- Specify the Utility (Payoff / Objective) Function: a function that defines the final numerical value to player *p* when the game ends in terminal state *s*

MinMax Algorithm: the Idea

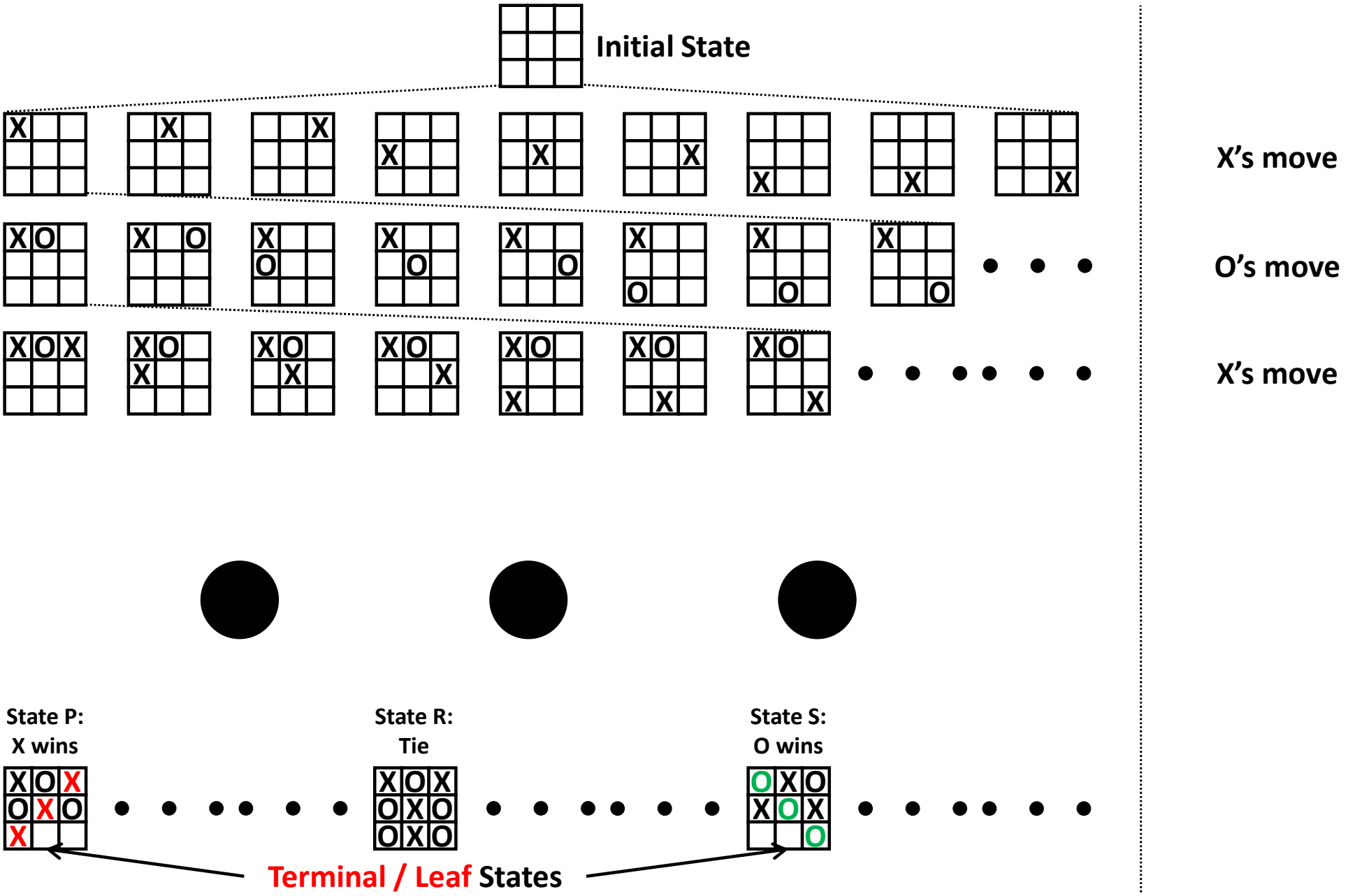
I don't know what move my opponent will choose, but I am going to ASSUME that it is going to be the best / optimal option

Tic Tac Toe: Zero Sum Game (2 Players)

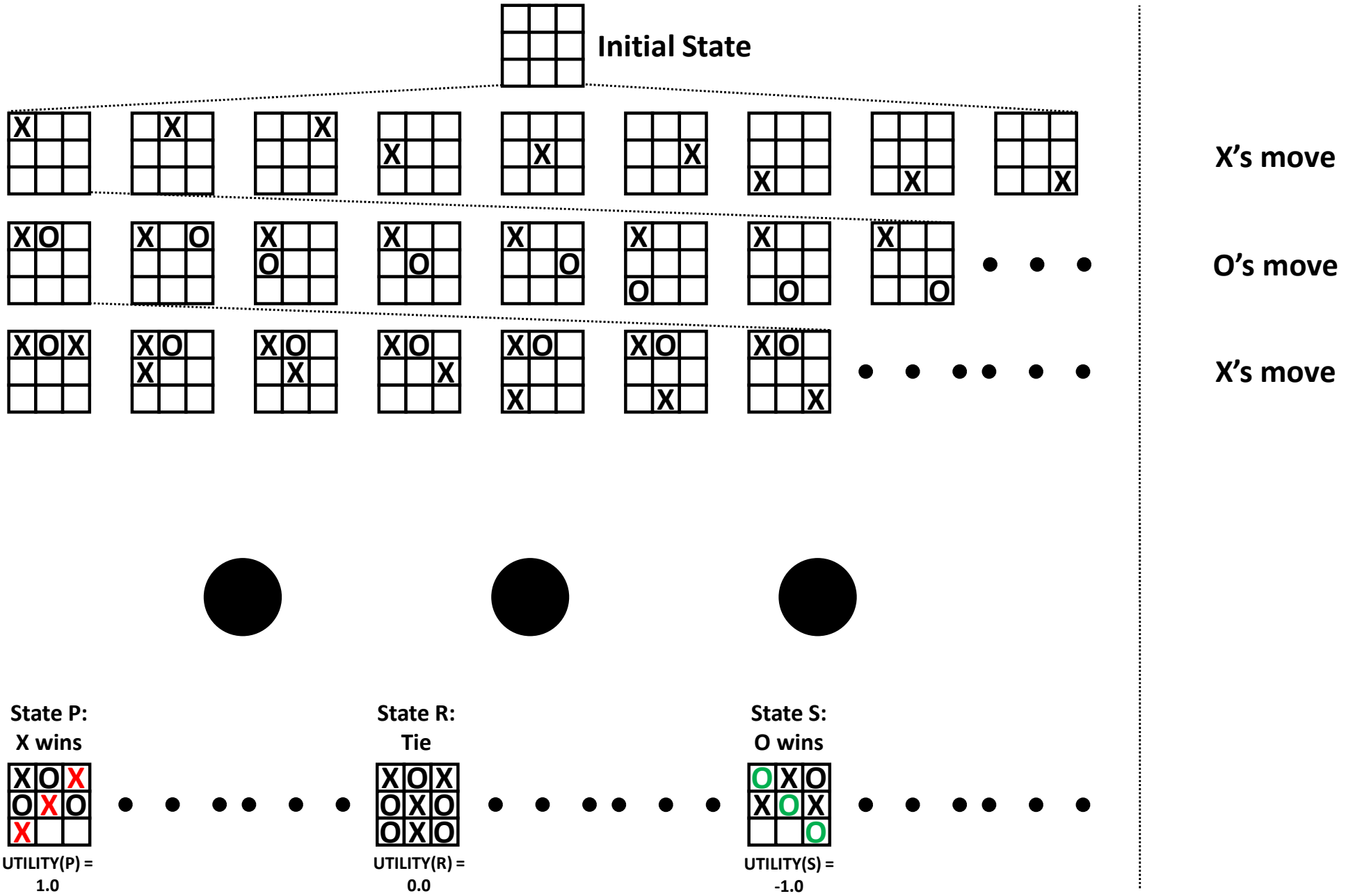


Challenge: I don't know what move my opponent will choose. I can't plan ahead because of that.

Tic Tac Toe: Zero Sum Game (2 Players)



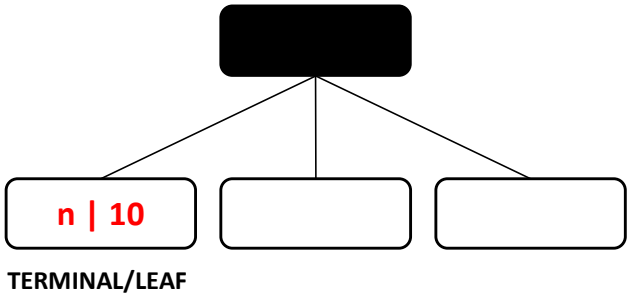
Tic Tac Toe: Zero Sum Game (2 Players)



MinMax: Assigning MINMAX Values

CASE 1:
State n is Terminal Node

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



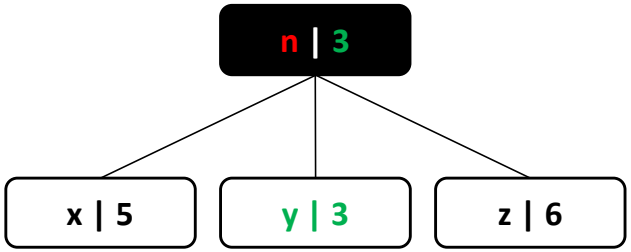
$$k = MINMAX(n) = UTILITY(n)$$

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

$$= 10$$

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

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



$$k = MINMAX(n) =$$

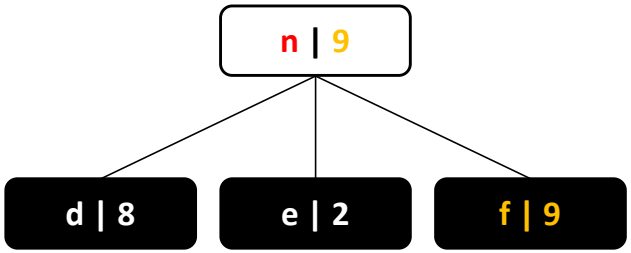
$$= \min_{a \in ACTIONS(n)} MINMAX(RESULT(n, a))$$

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

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

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

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



$$k = MINMAX(n) =$$

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

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

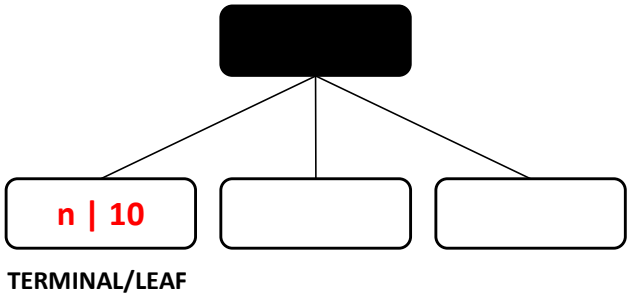
$$= \max(8, 2, 9) = 9$$

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

MinMax: Assigning MINMAX Values

CASE 1:
State n is Terminal Node

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



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

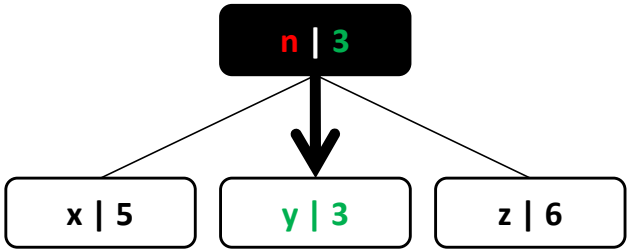
$$= \textit{utility value of this state for MAX Player}$$

$$= 10$$

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



$$k = MINMAX(n) =$$

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

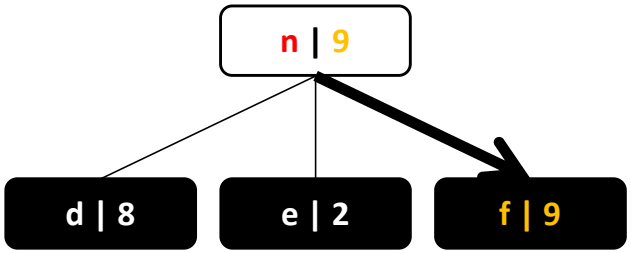
$$= \min(MINMAX(x), MINMAX(y), MINMAX(z))$$

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

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



$$k = MINMAX(n) =$$

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

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

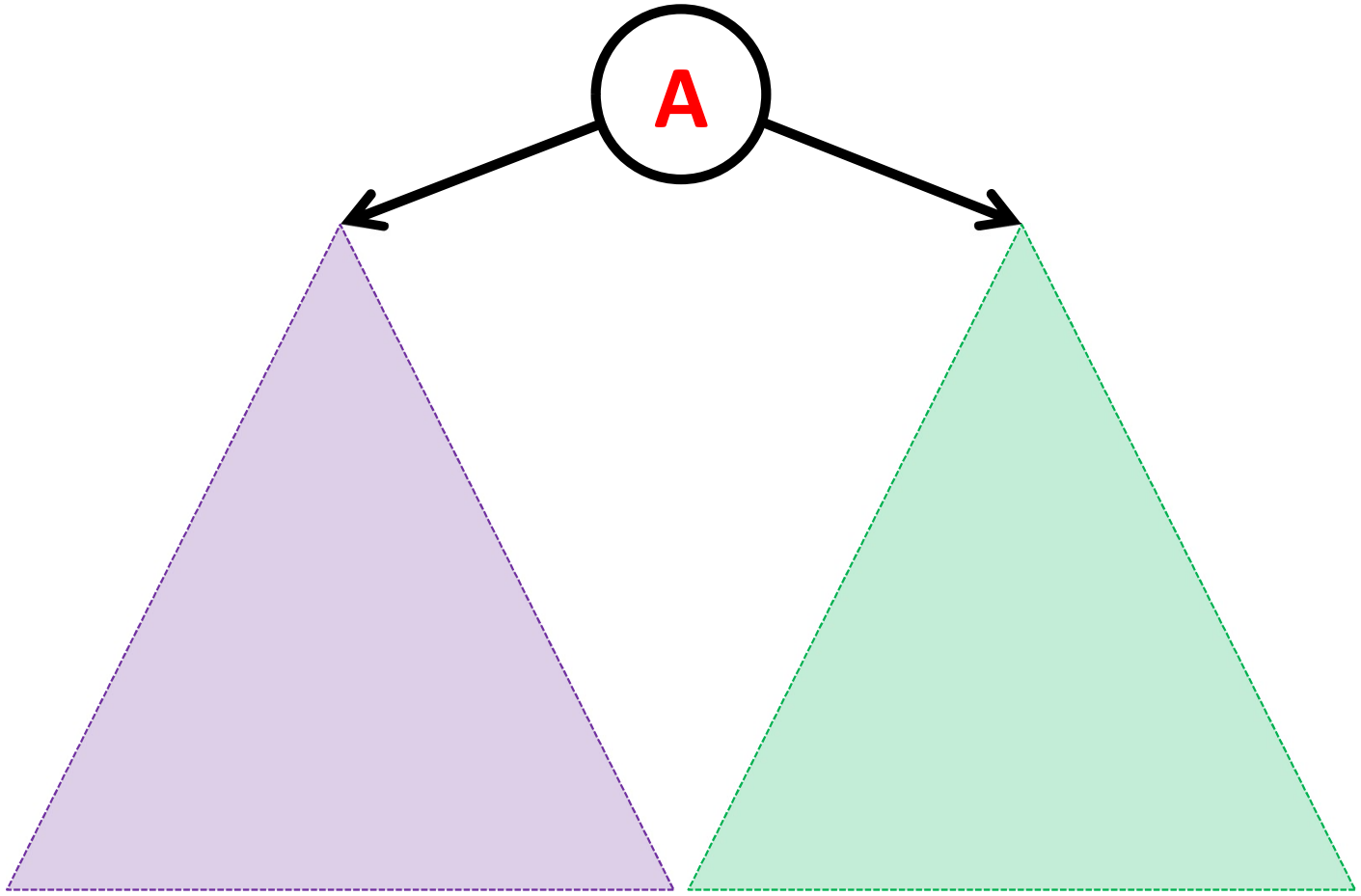
$$= \max(8, 2, 9) = 9$$

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

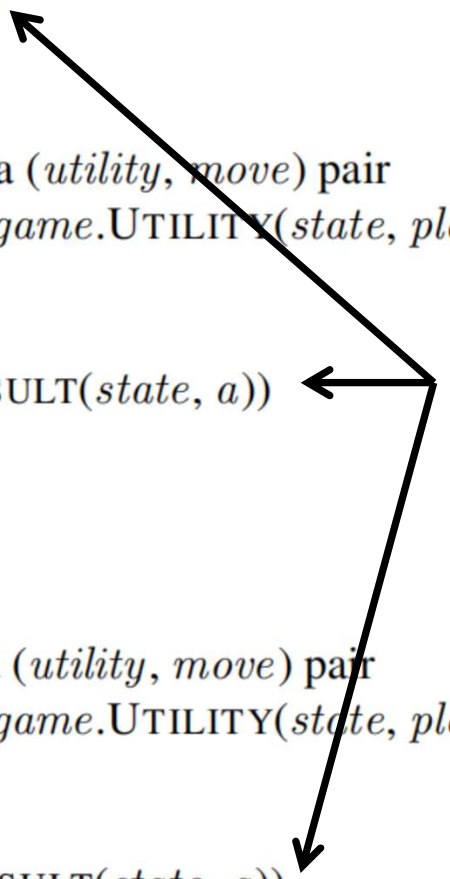
```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

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

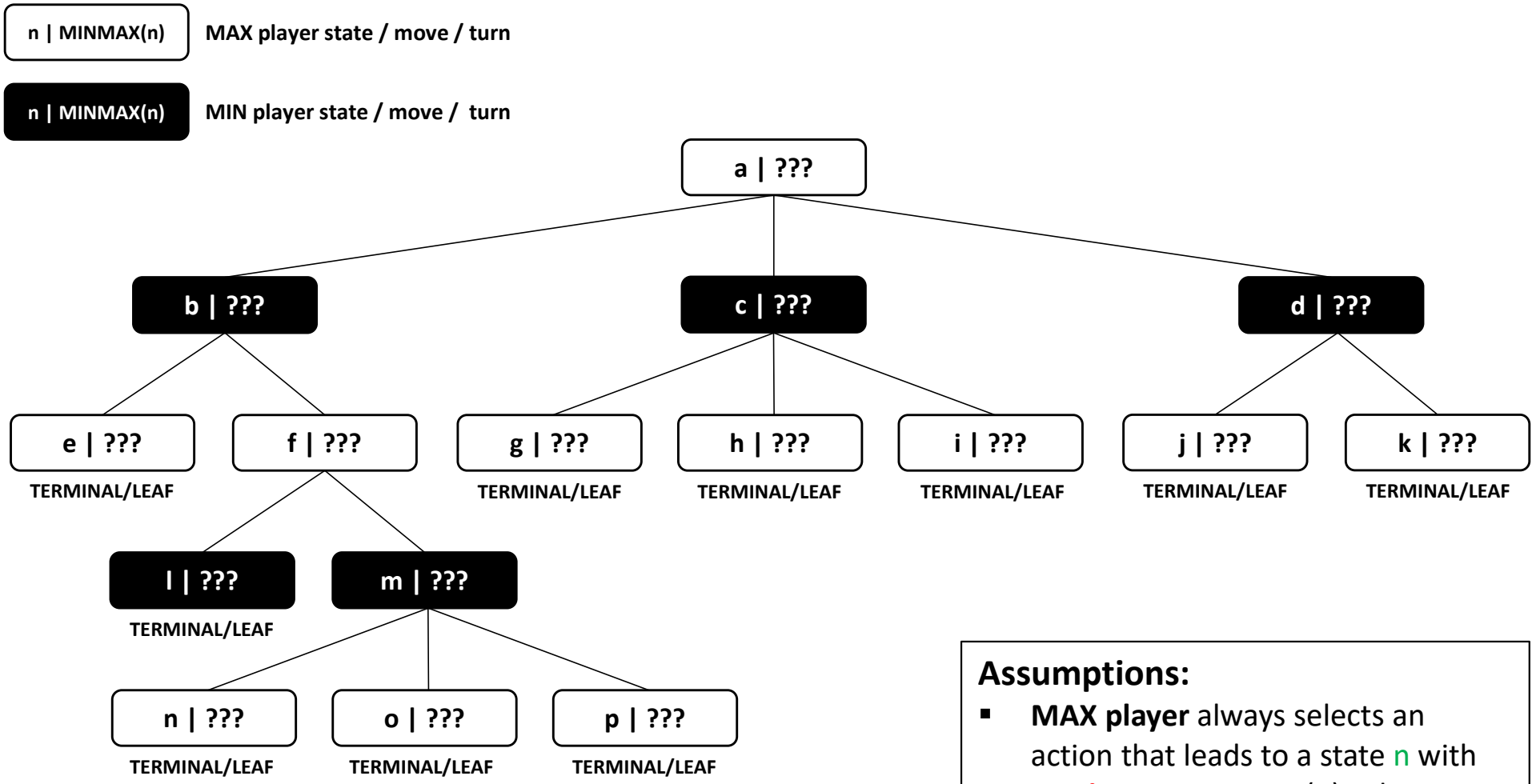


MinMax Algorithm: Pseudocode

```
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
```



Example MinMax Search Tree



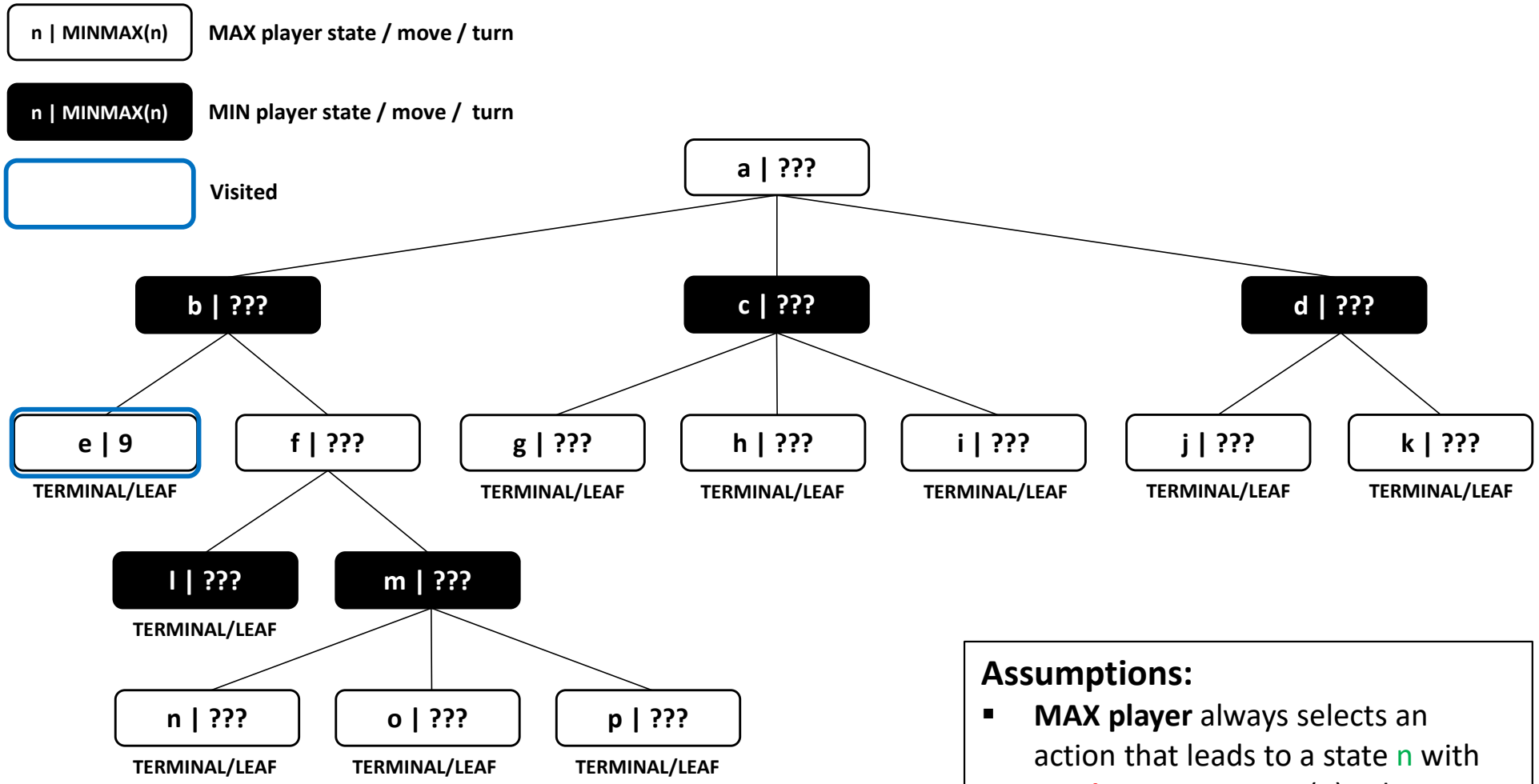
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



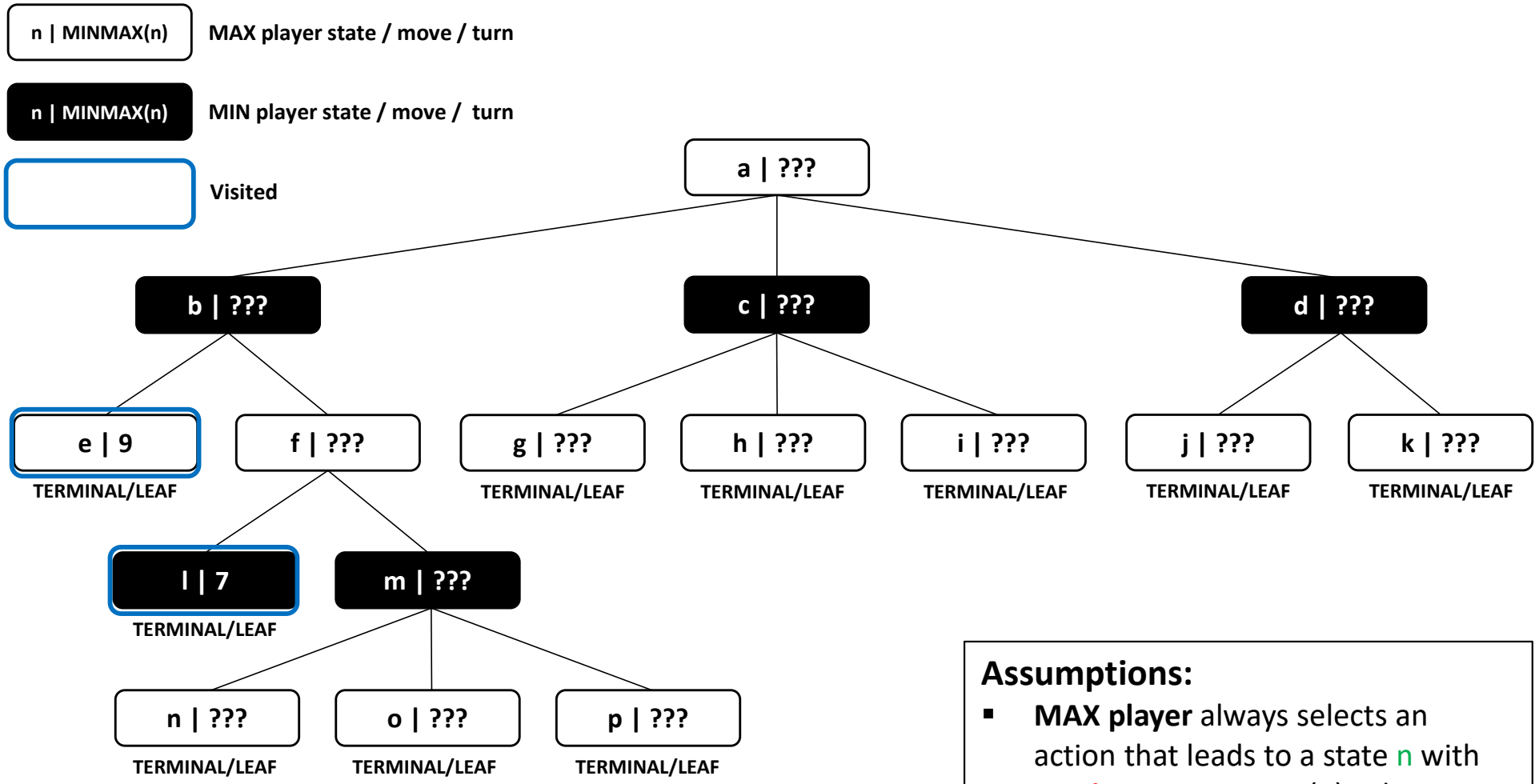
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



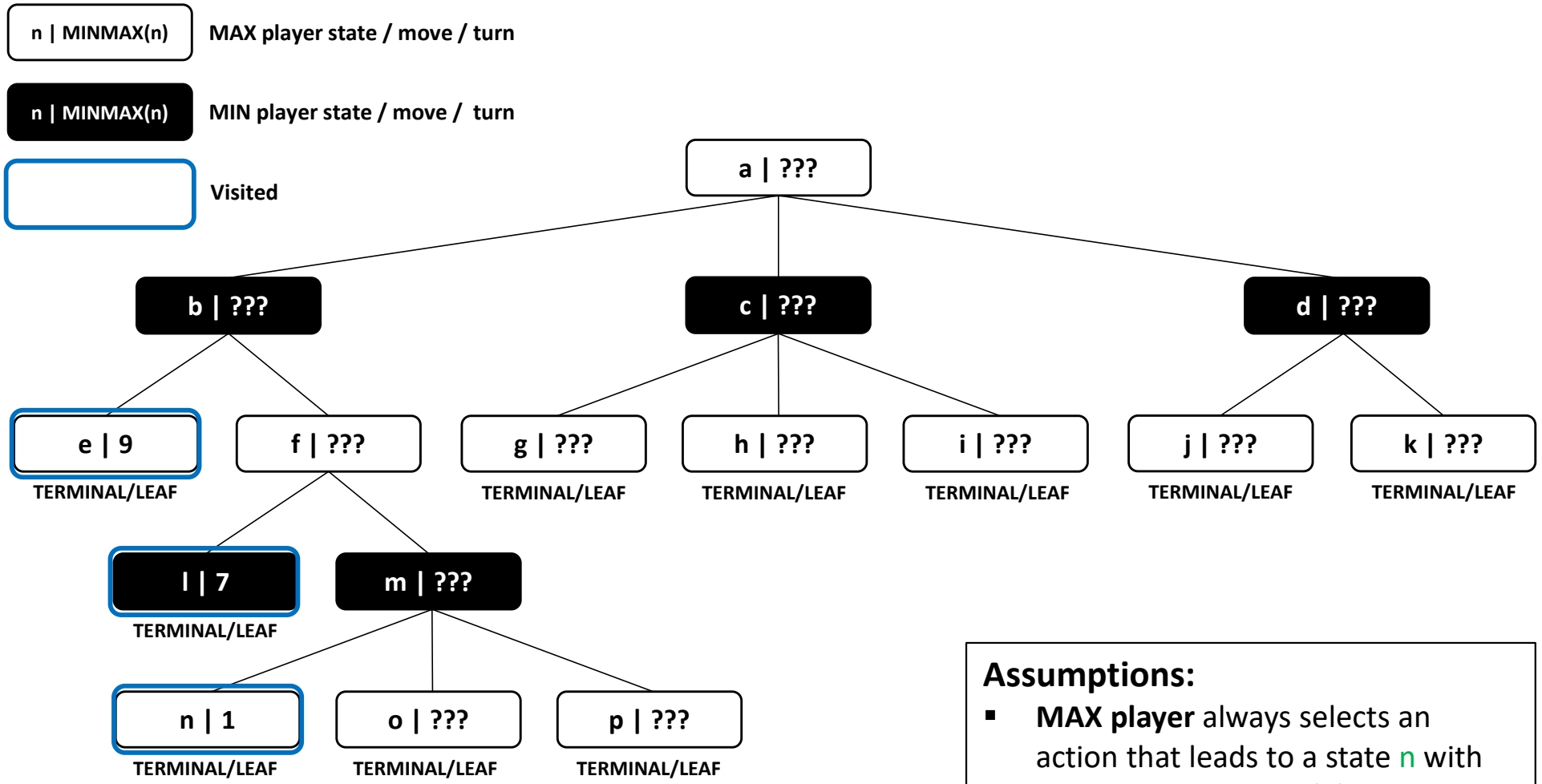
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



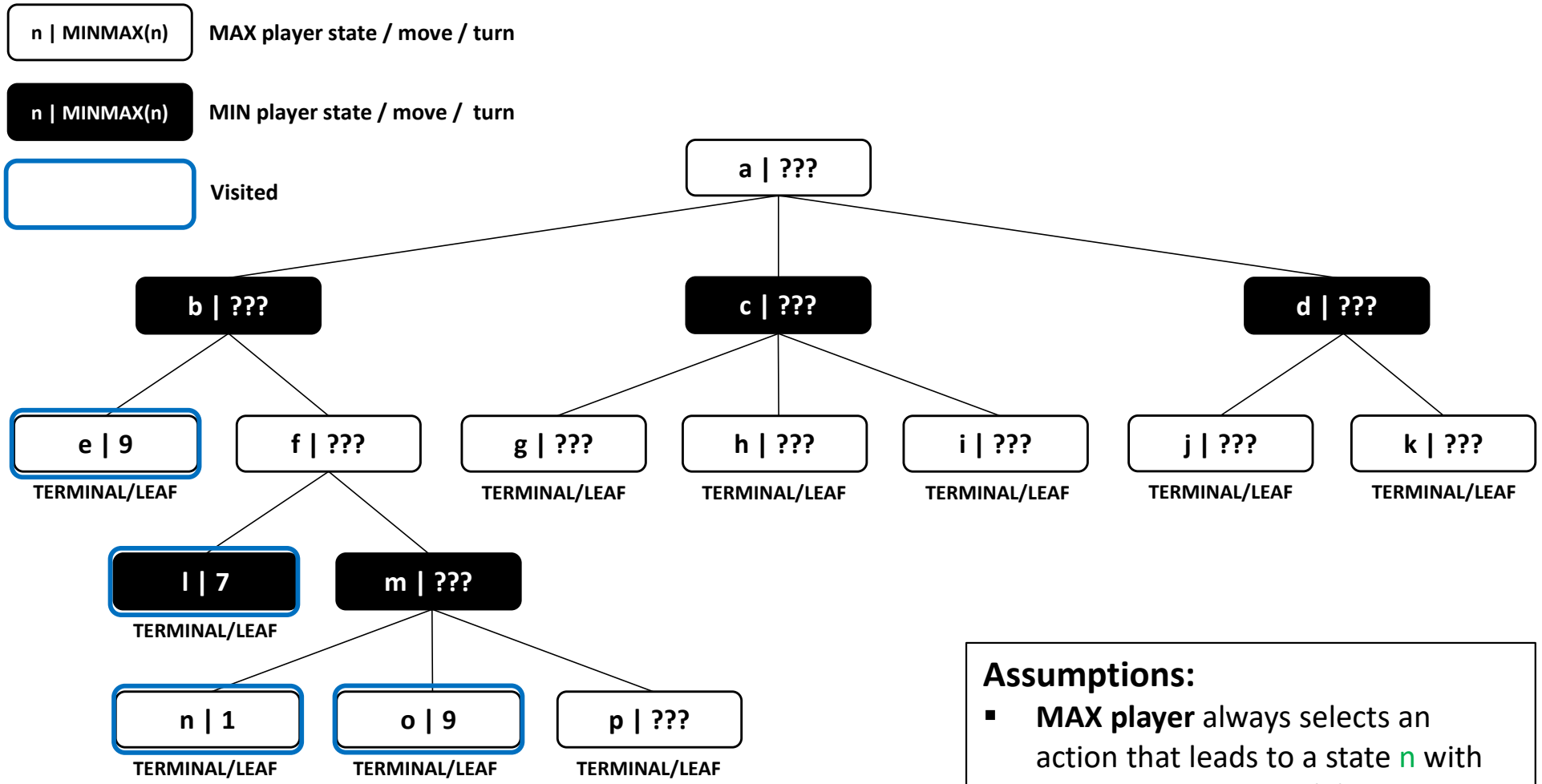
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



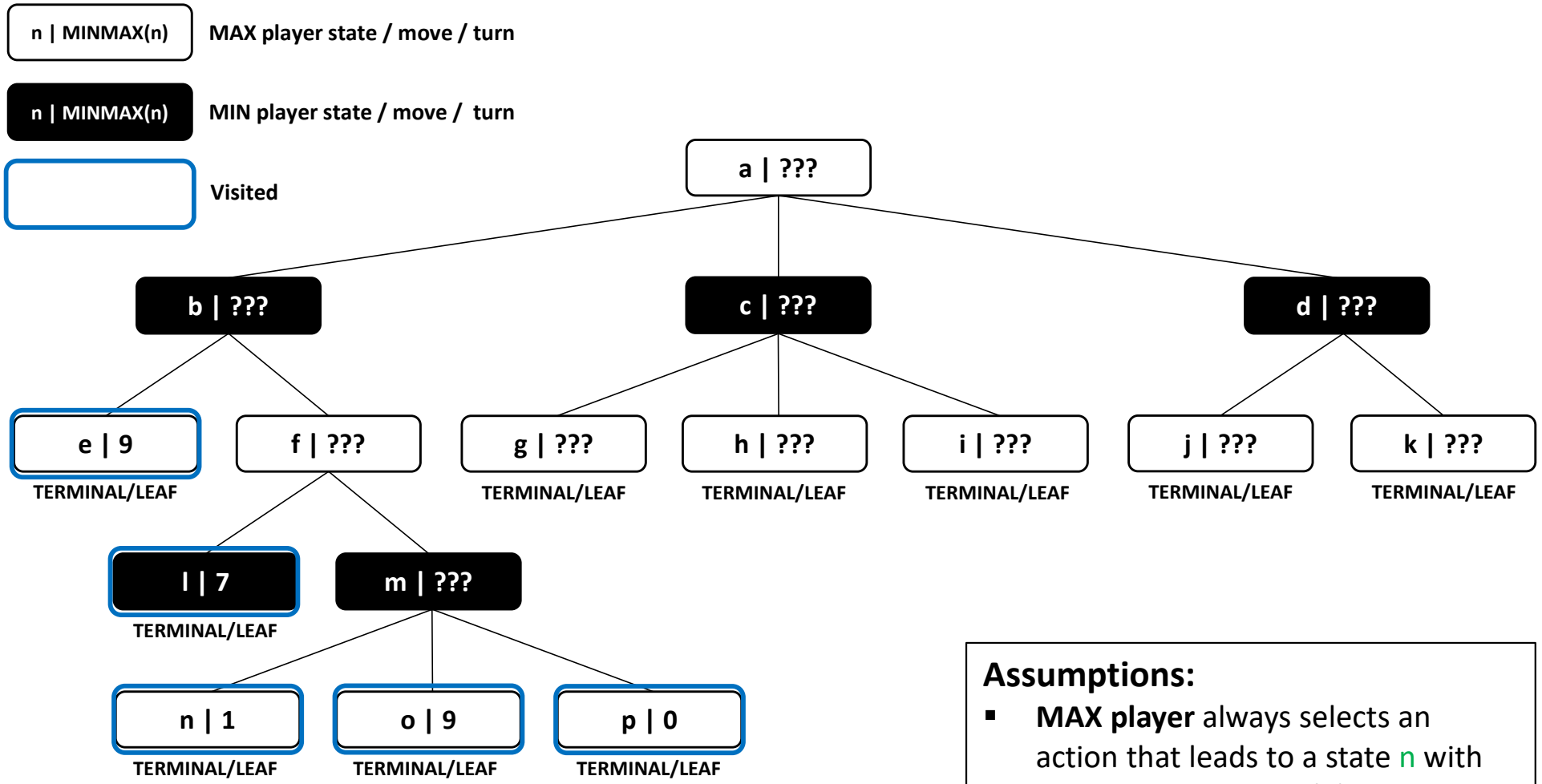
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



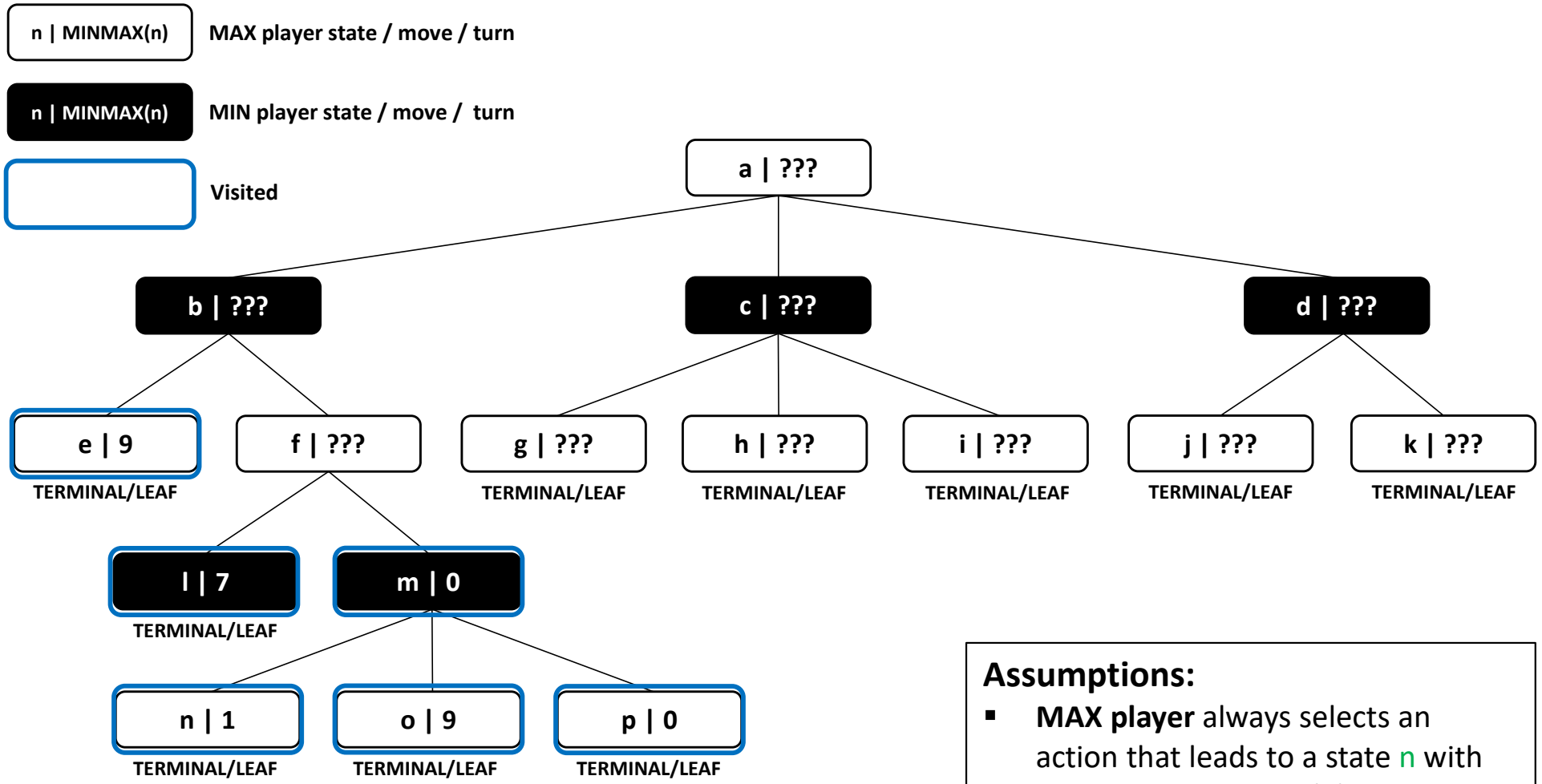
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



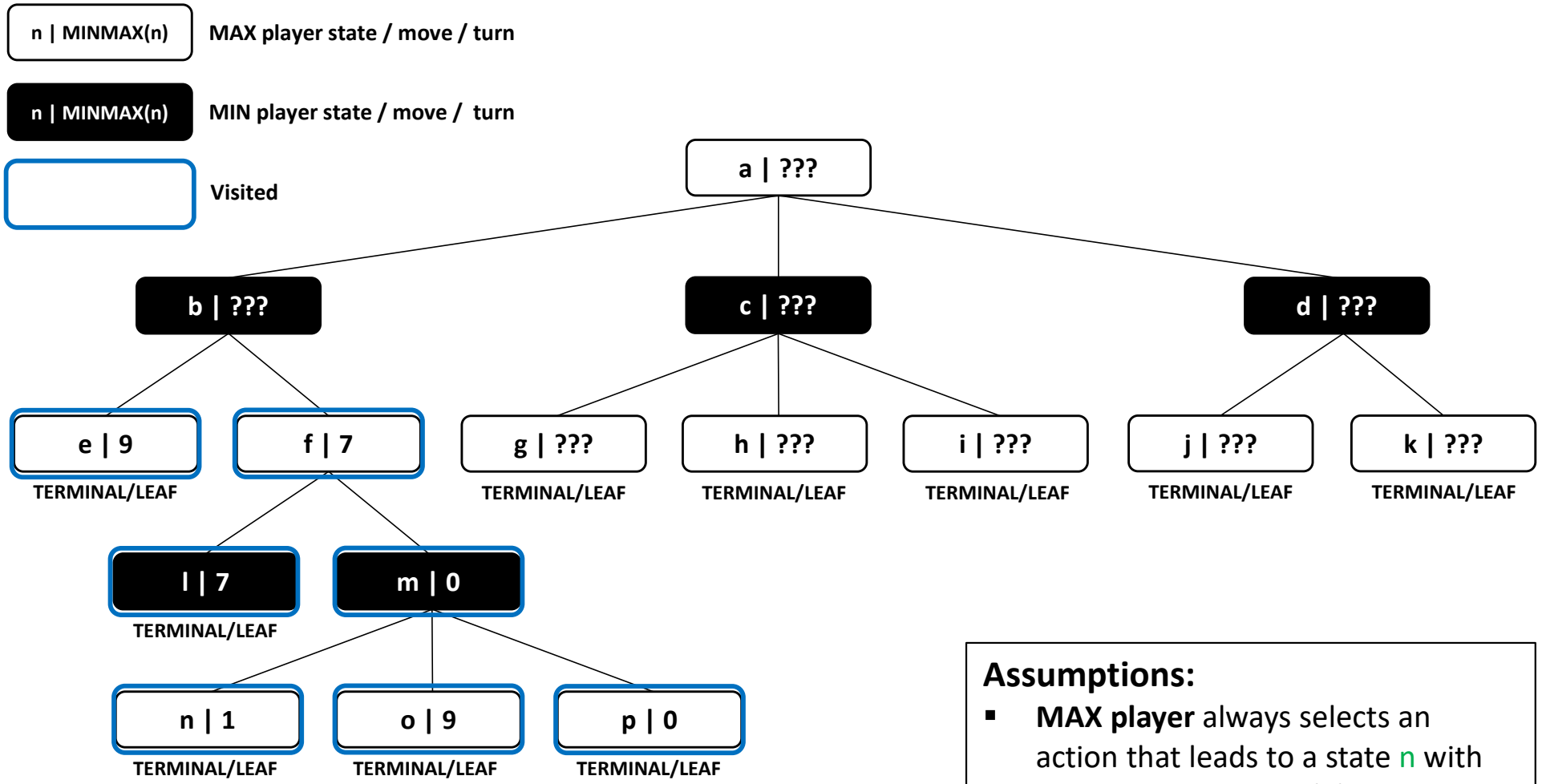
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



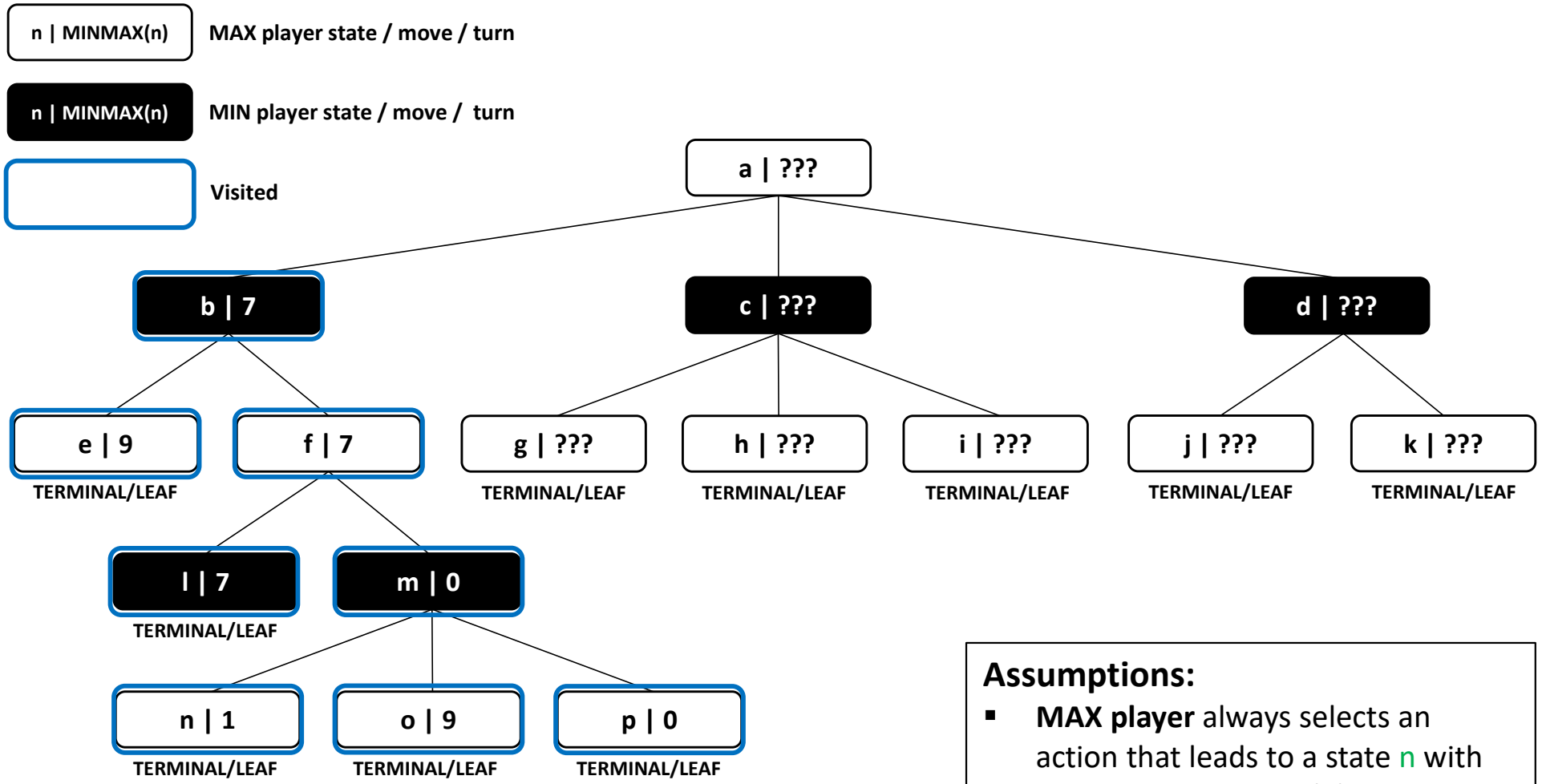
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



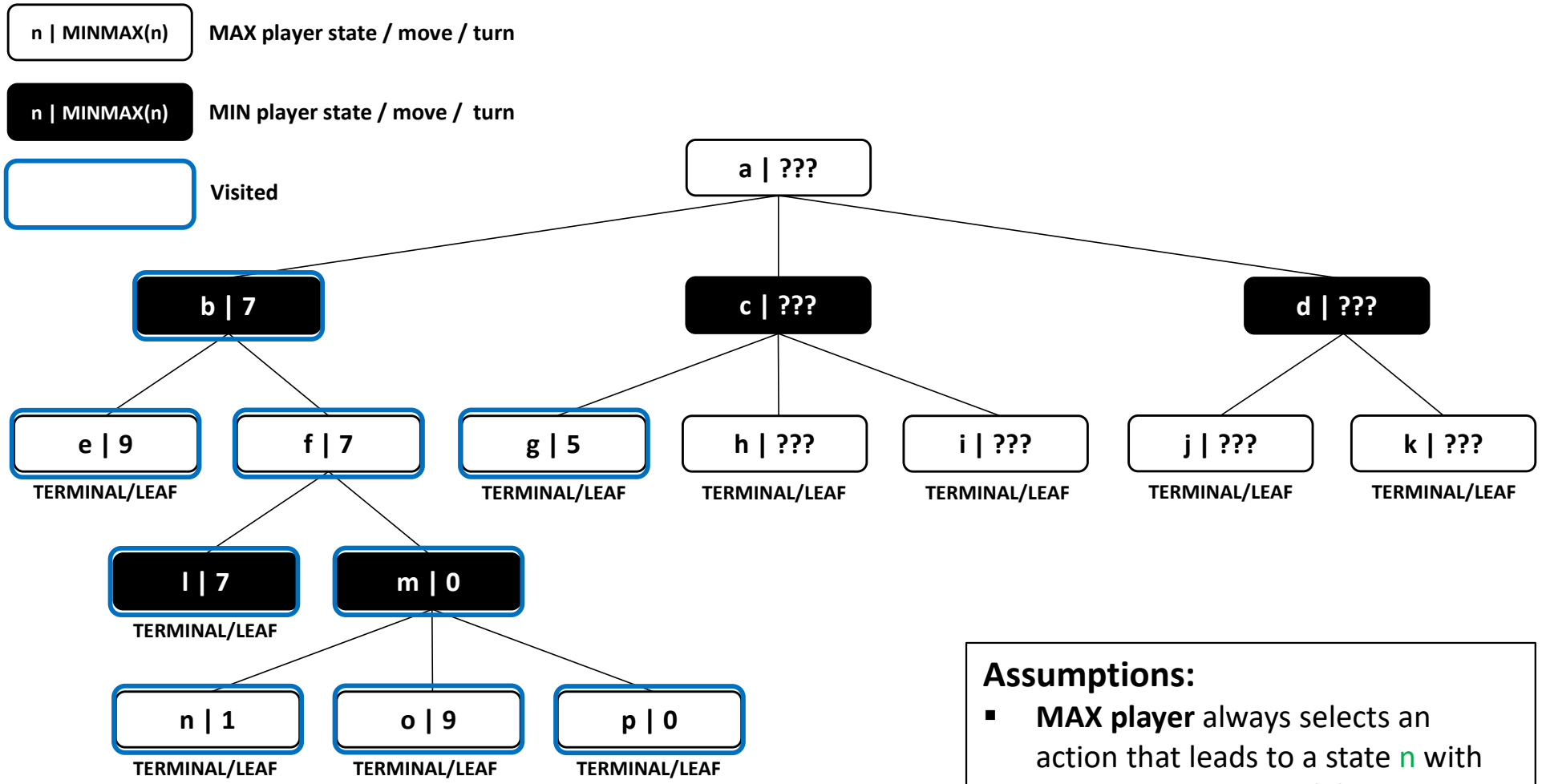
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



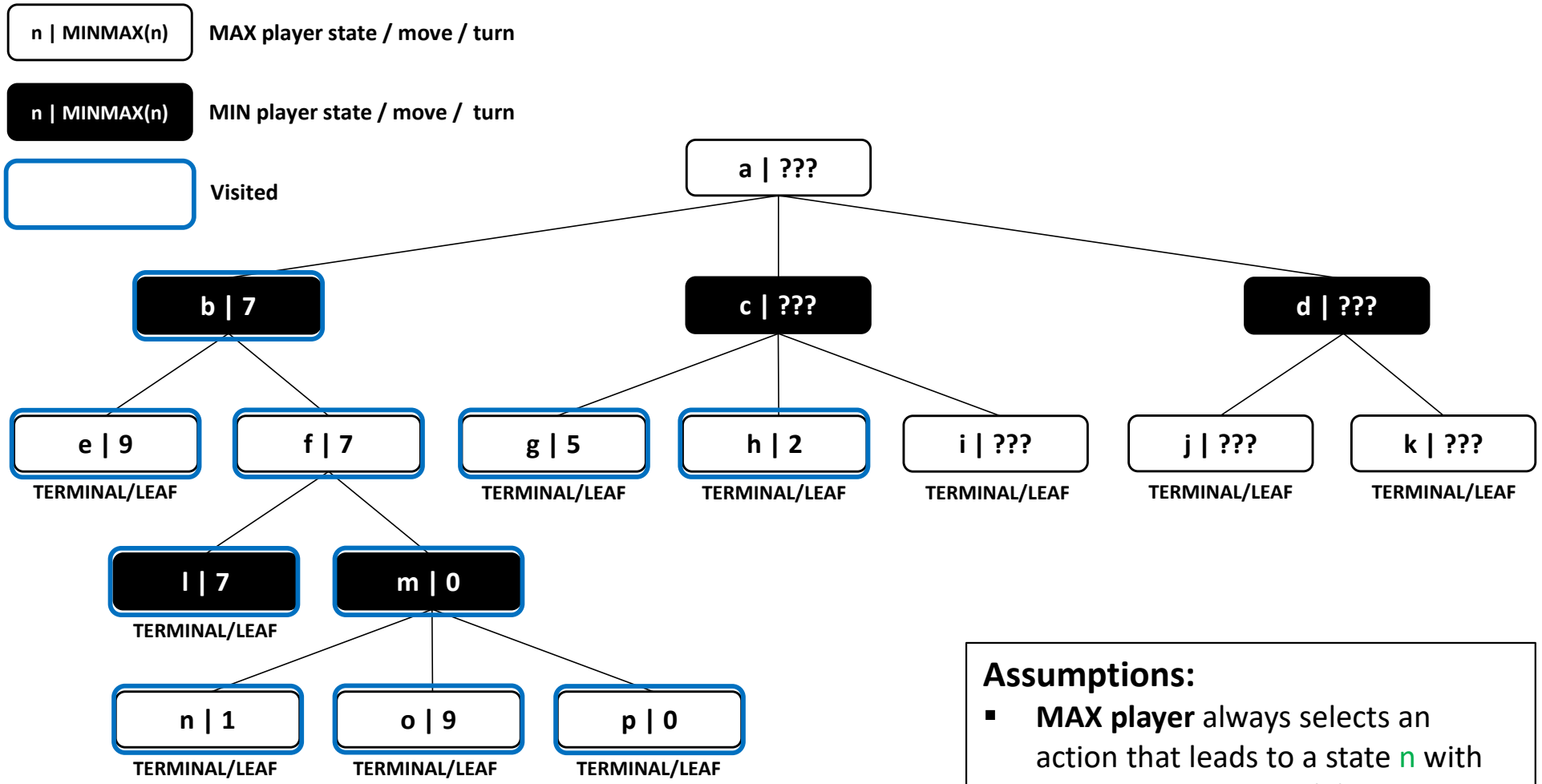
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



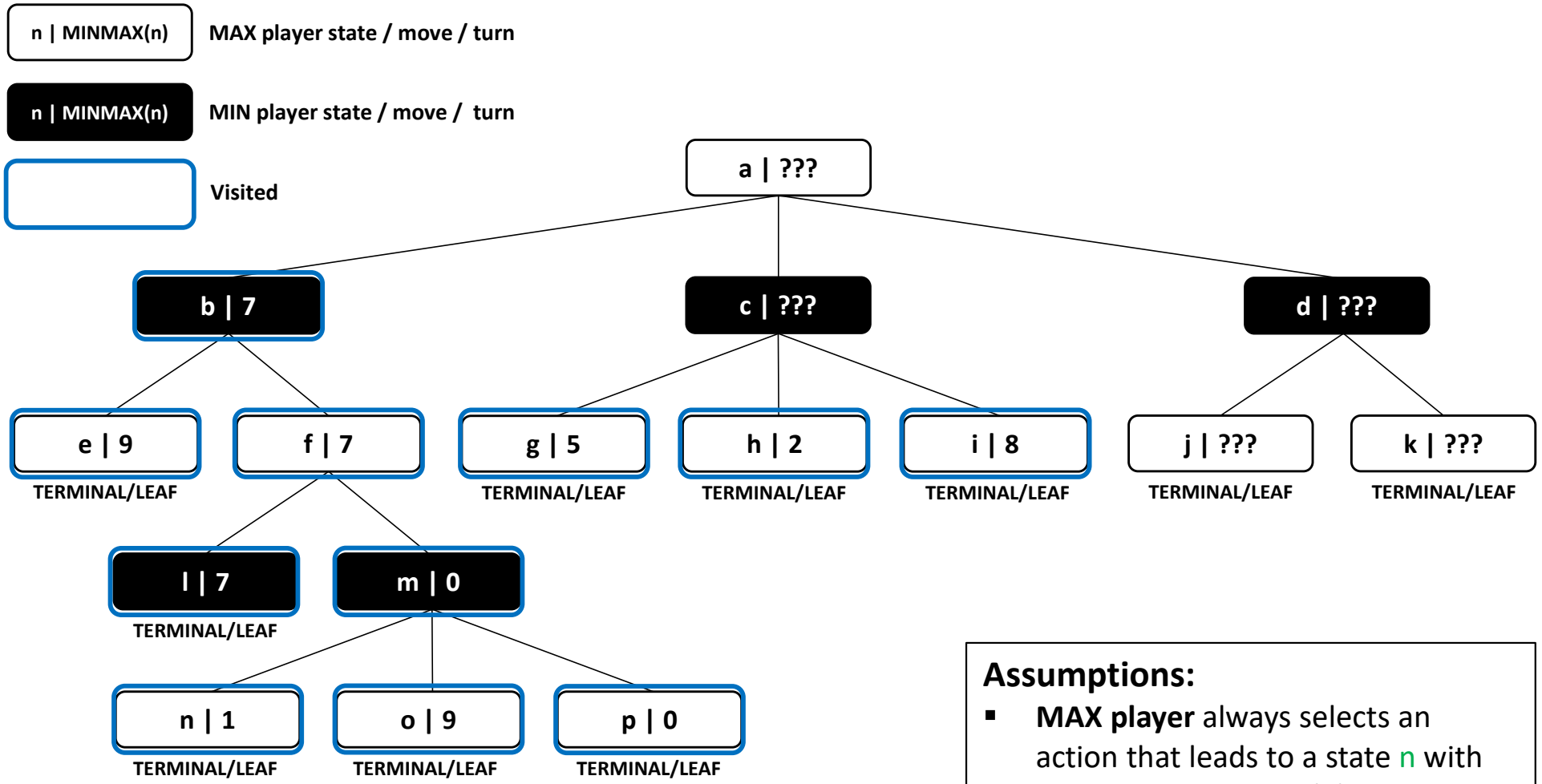
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



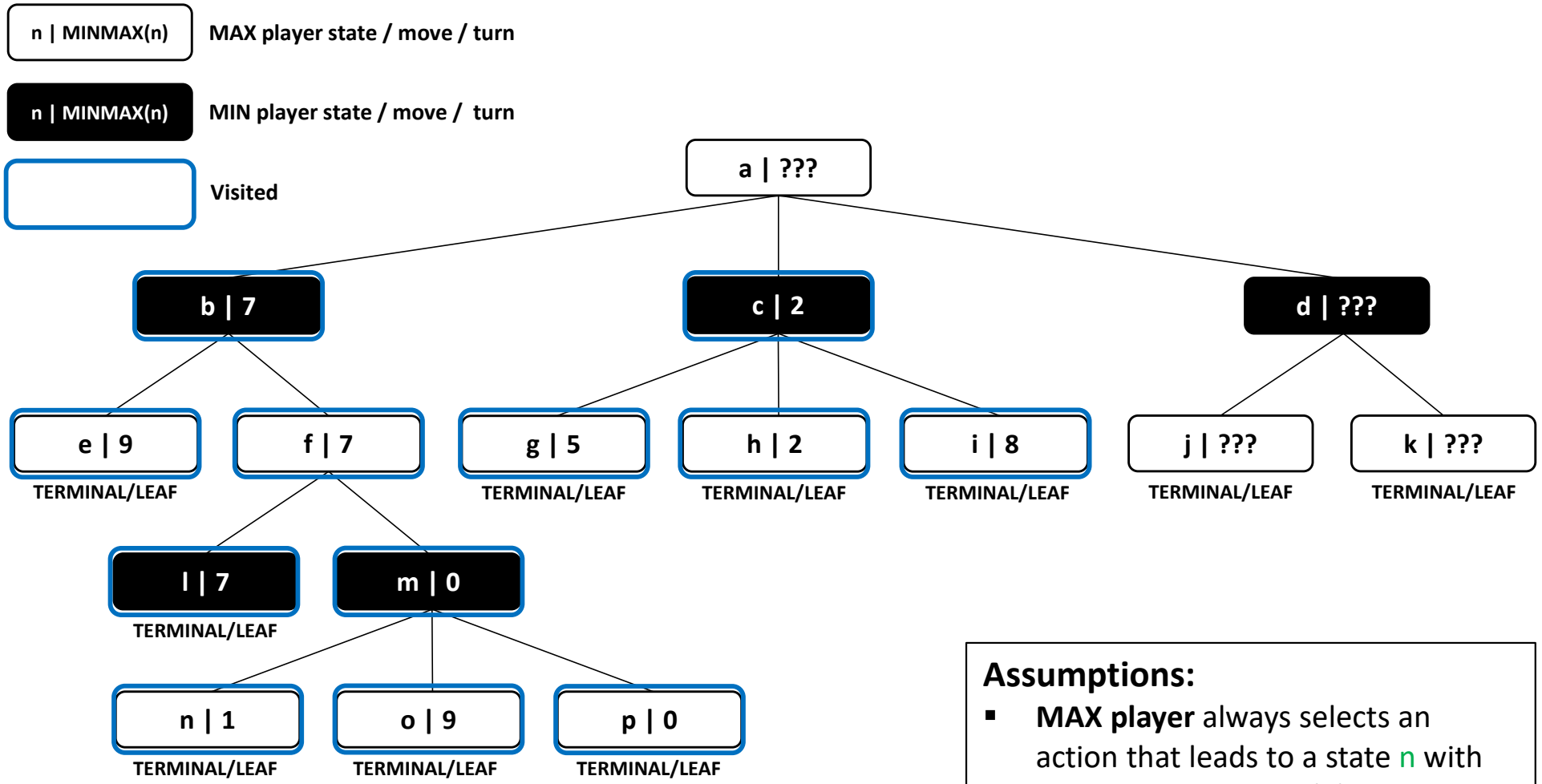
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



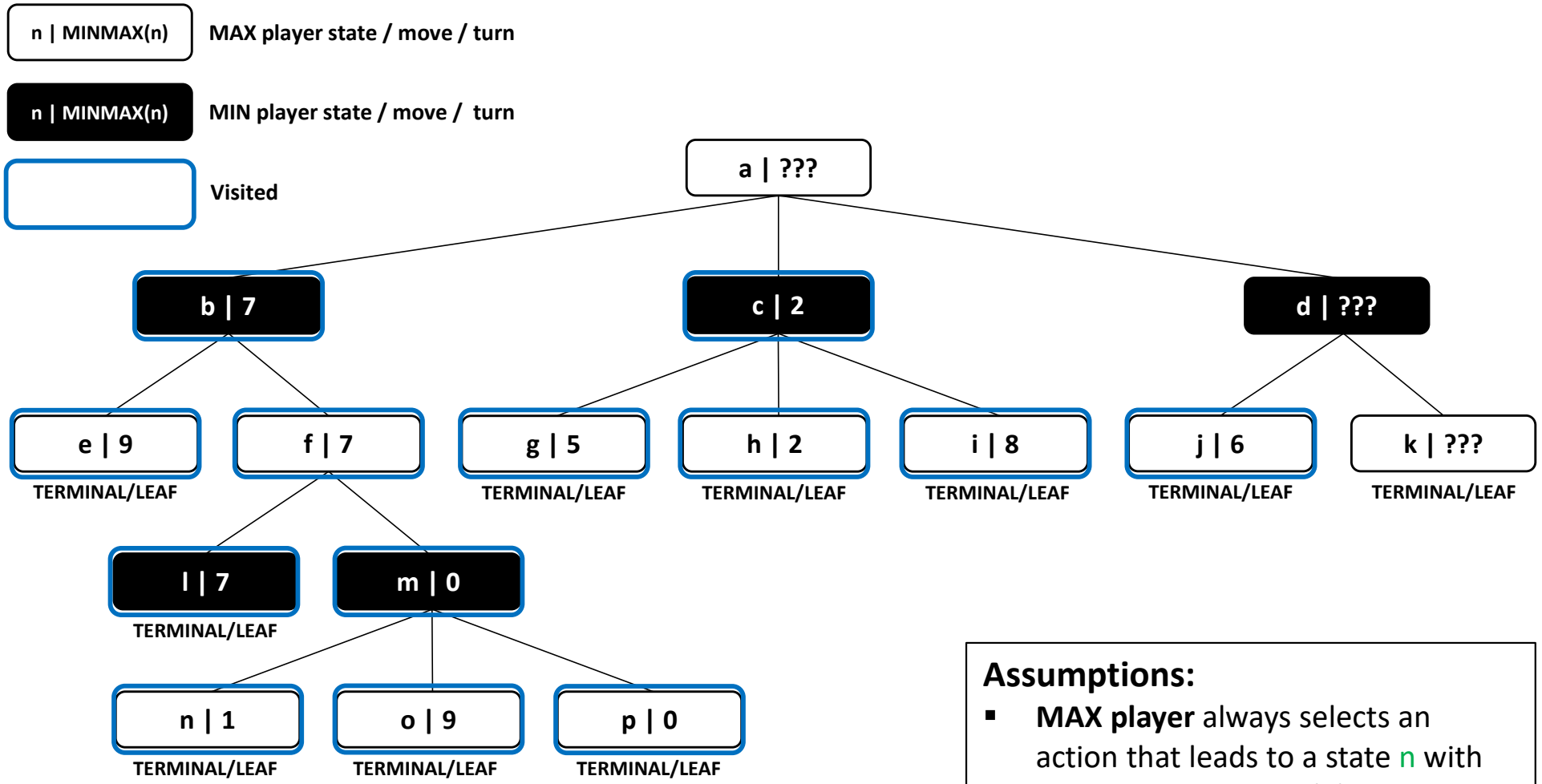
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



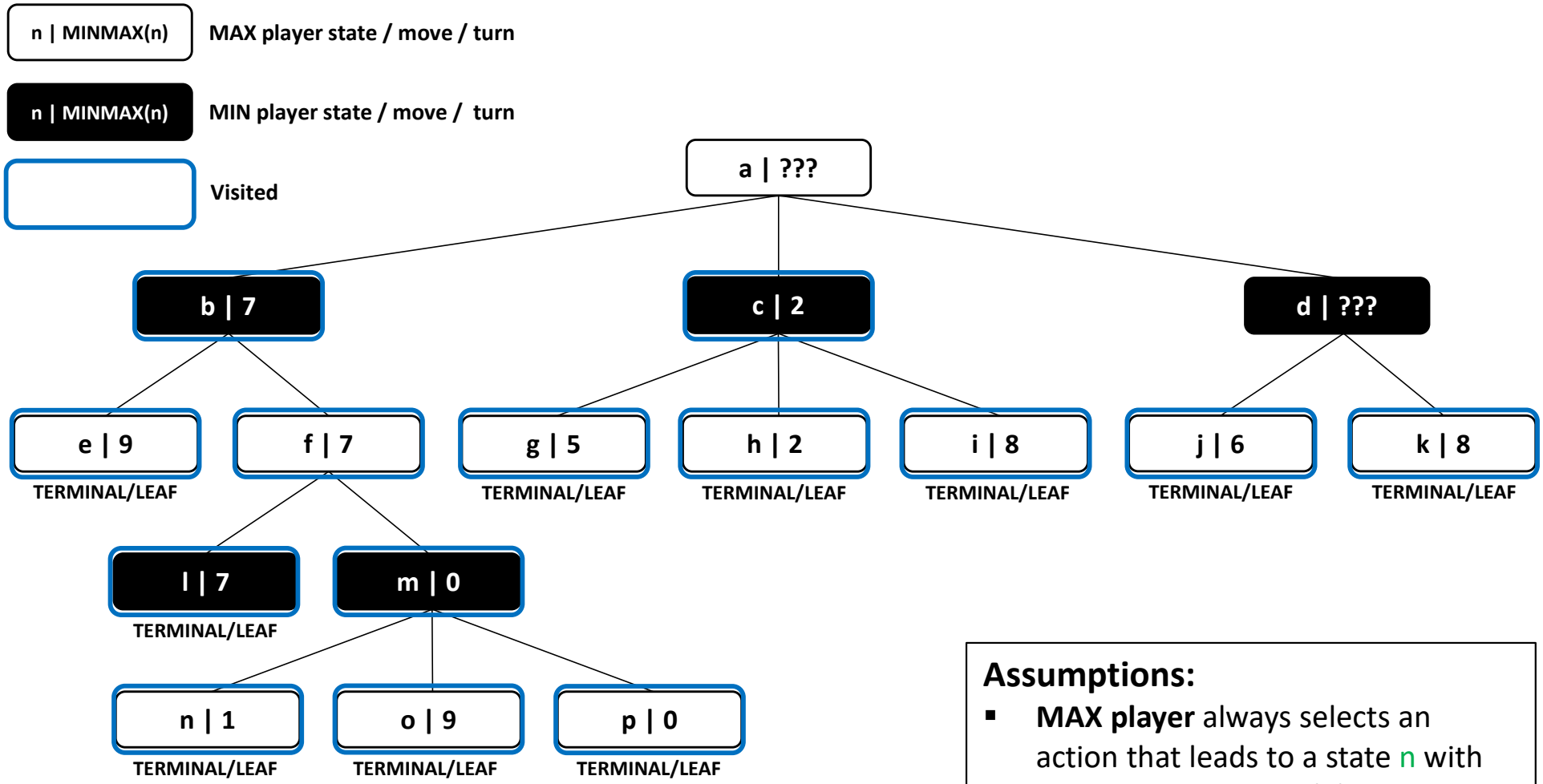
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



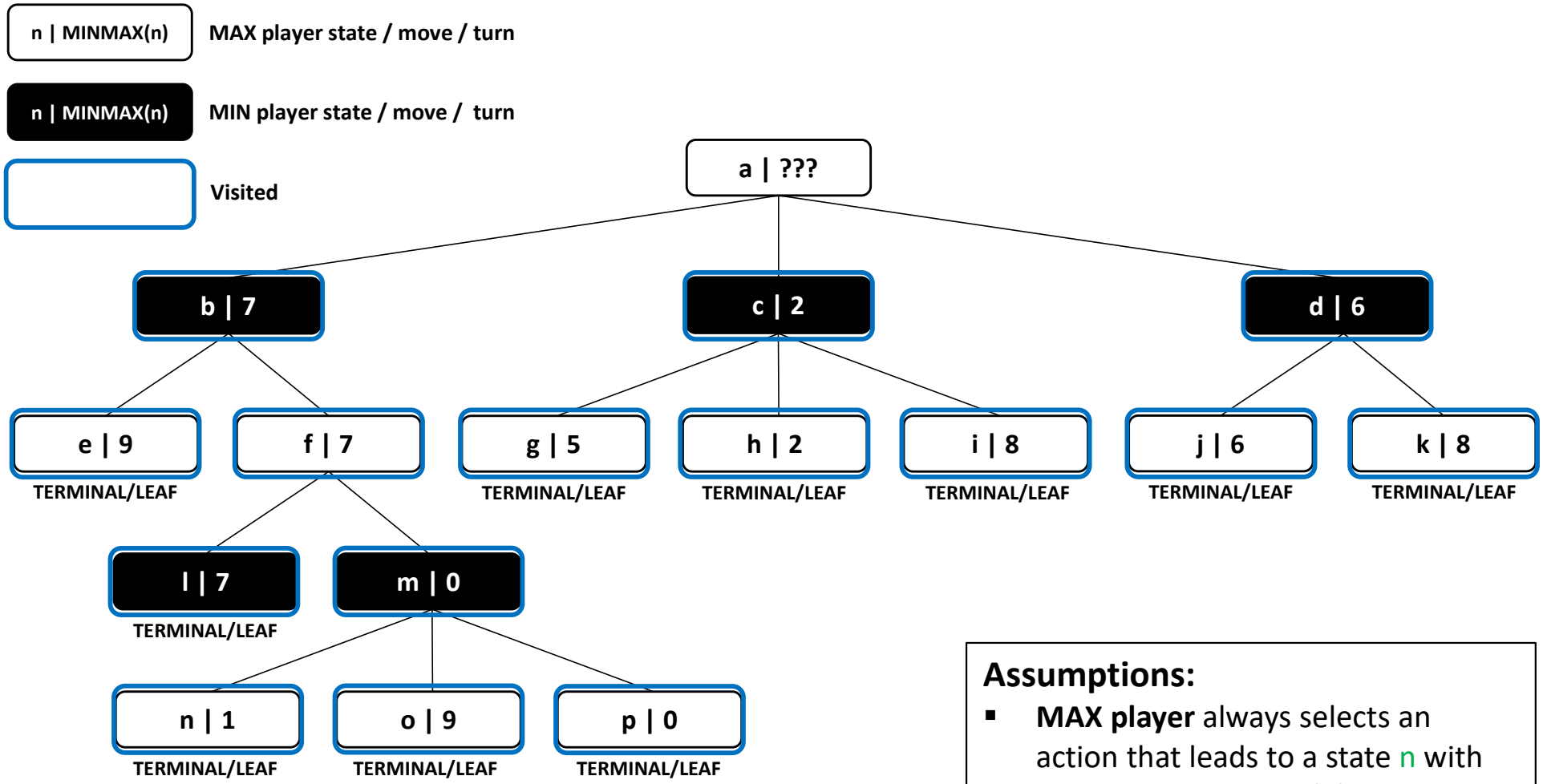
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



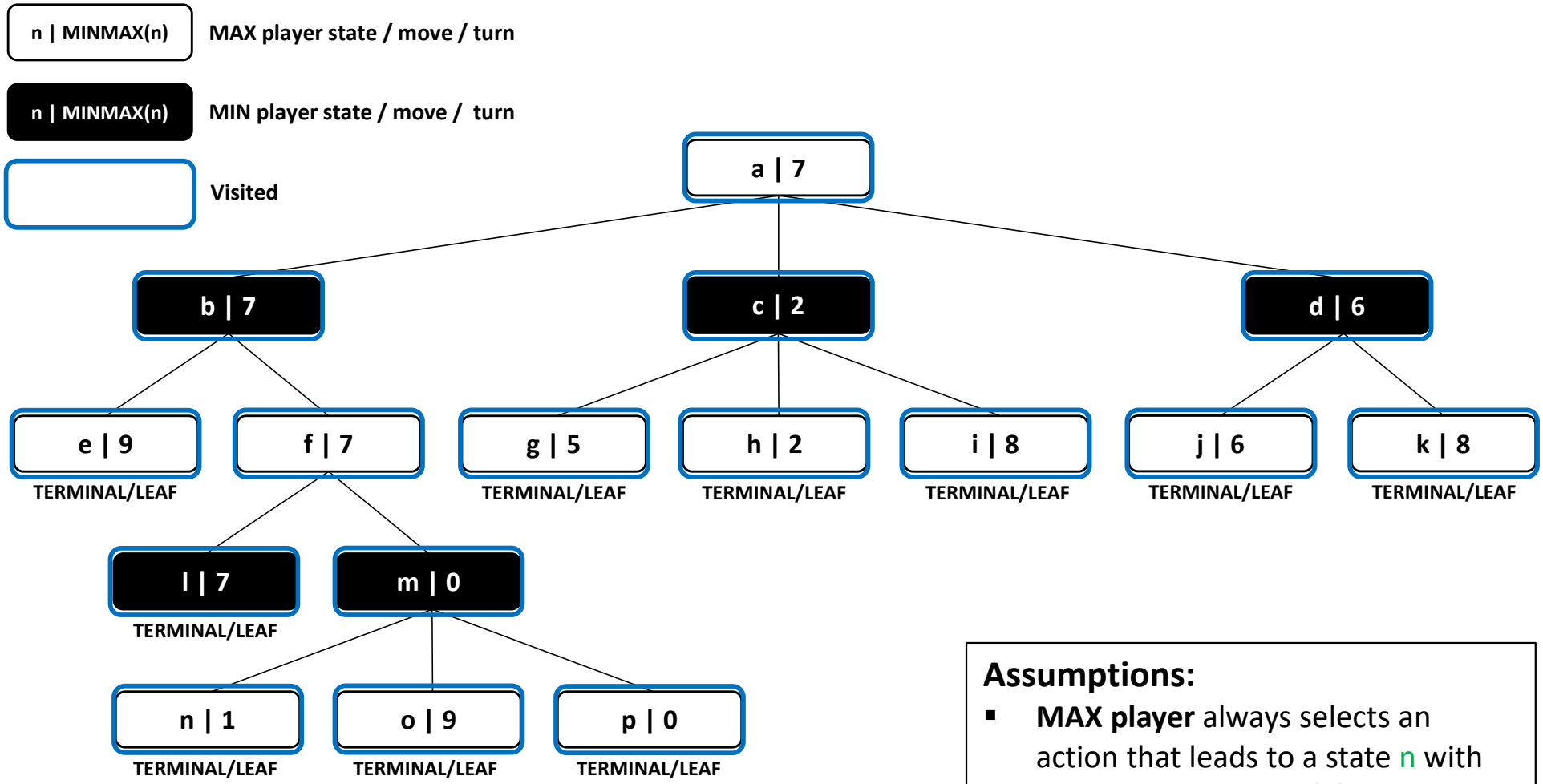
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



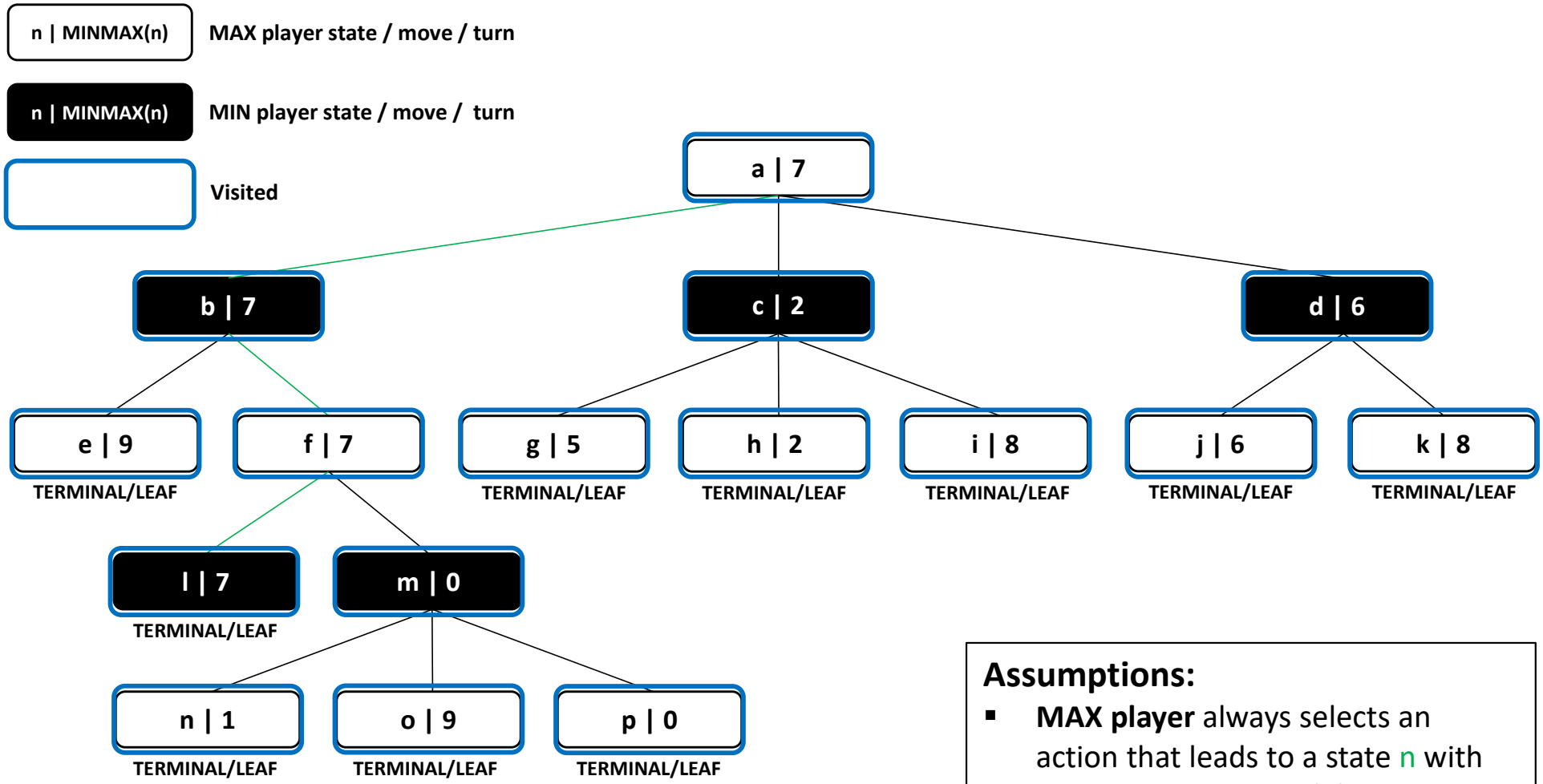
$$MINMAX(n) = MINMAX(State_n)$$

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

Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

Example MinMax Search Tree



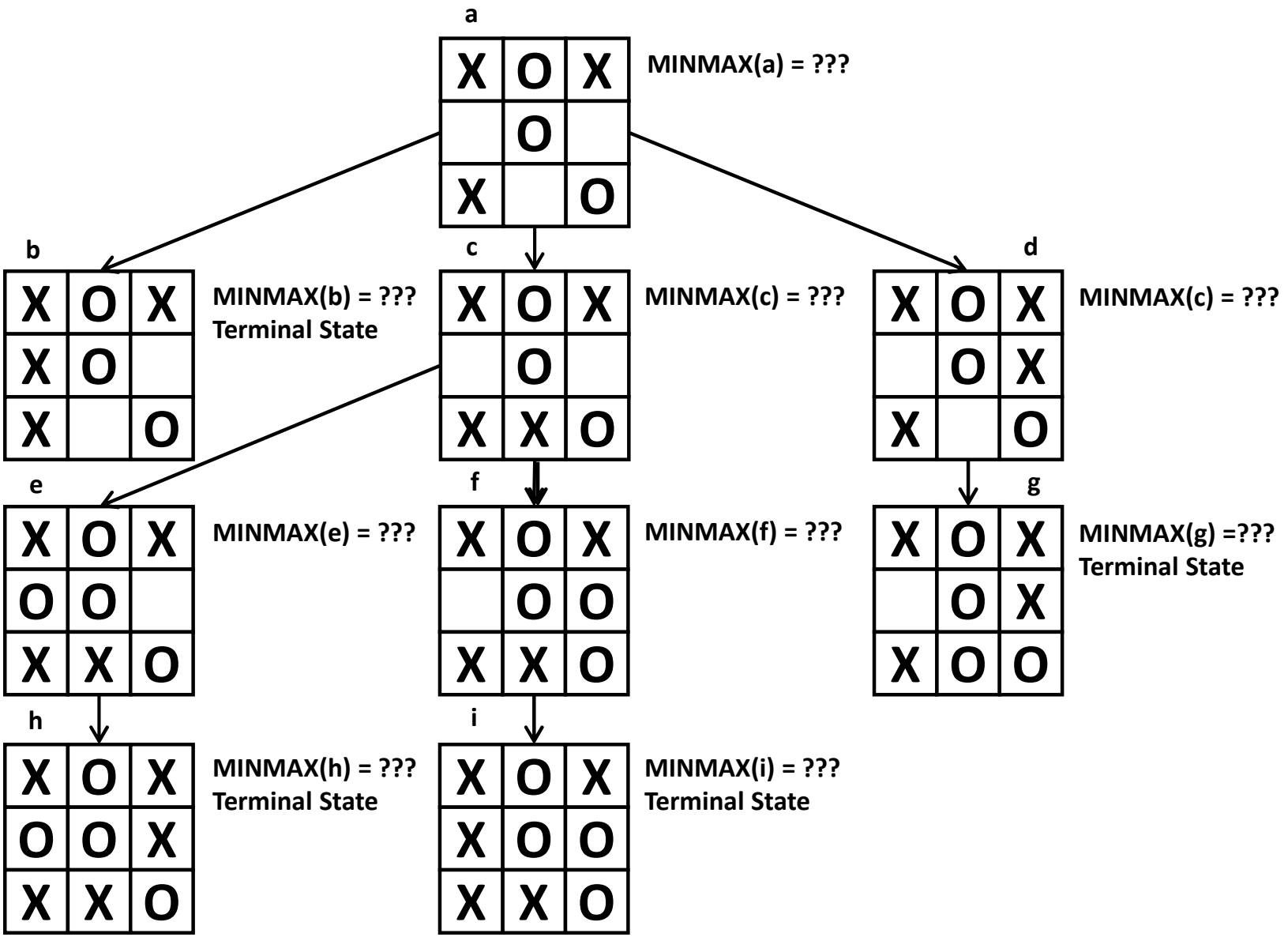
$$MINMAX(n) = MINMAX(State_n)$$

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

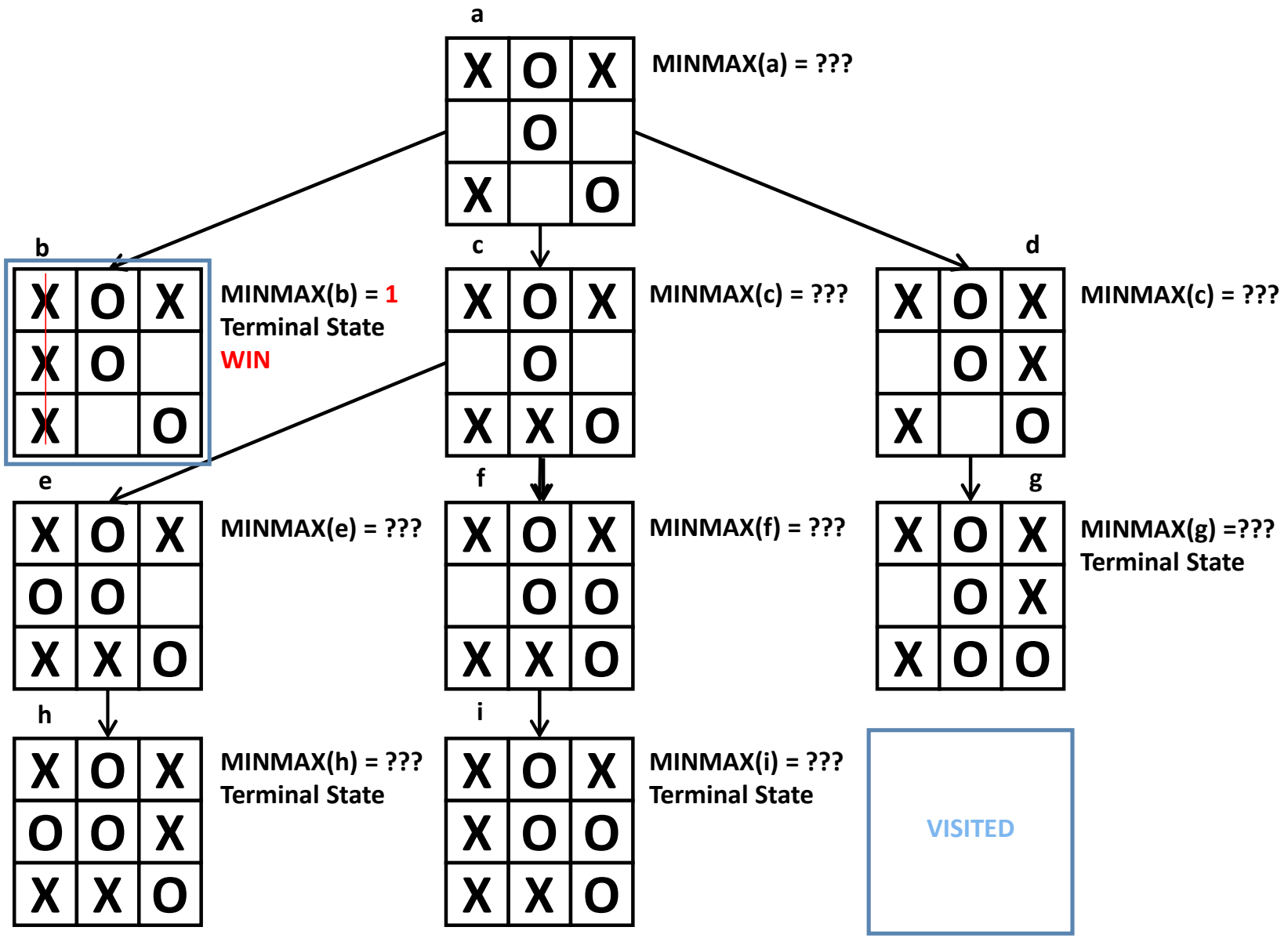
Assumptions:

- **MAX player** always selects an action that leads to a state n with **maximum** MINIMAX(n) value
- **MIN player** always selects an action that leads to a state n with **minimum** MINIMAX(n) value
- **BOTH players** always play optimally

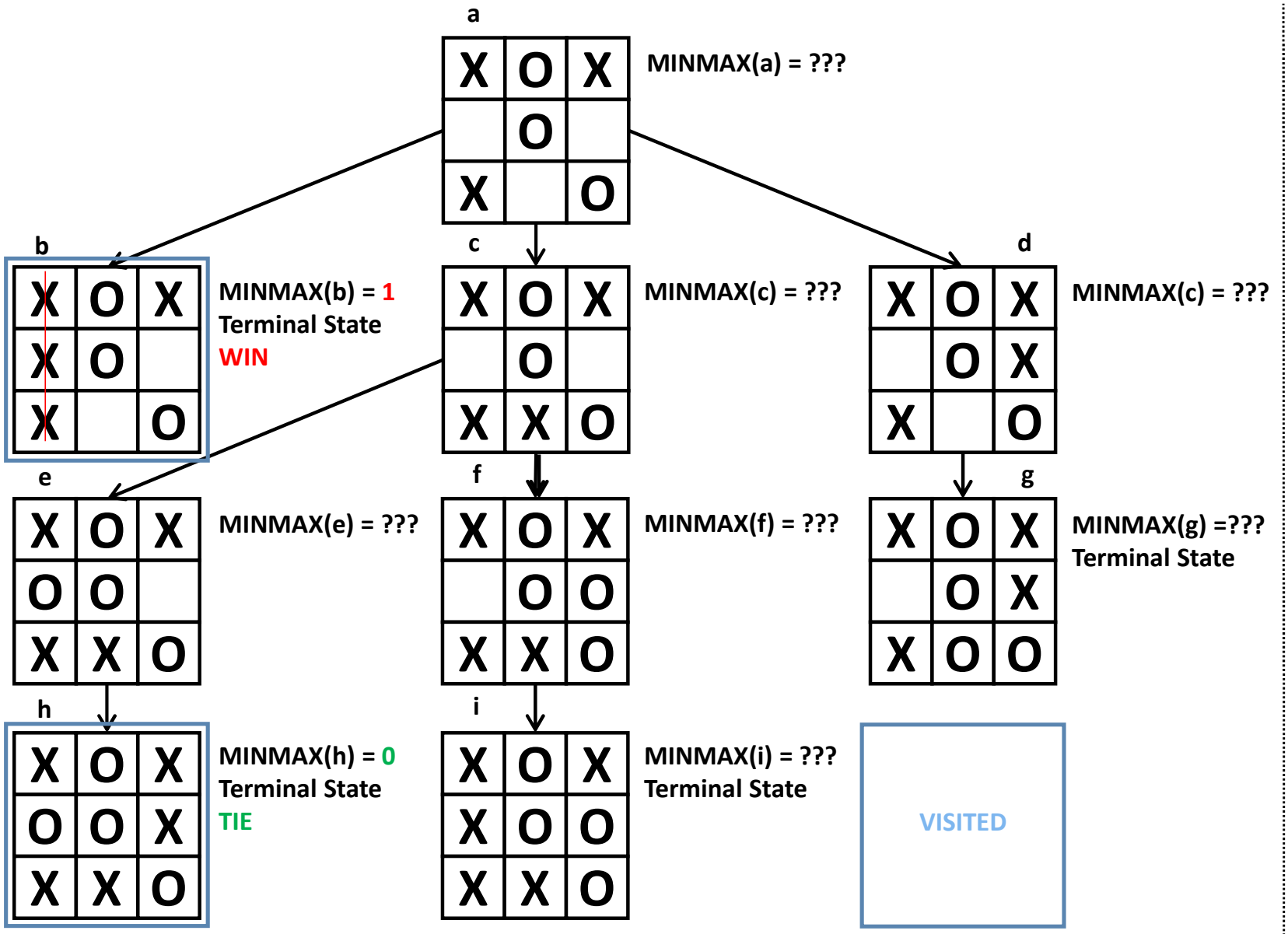
MinMax Algorithm: Tic Tac Toe



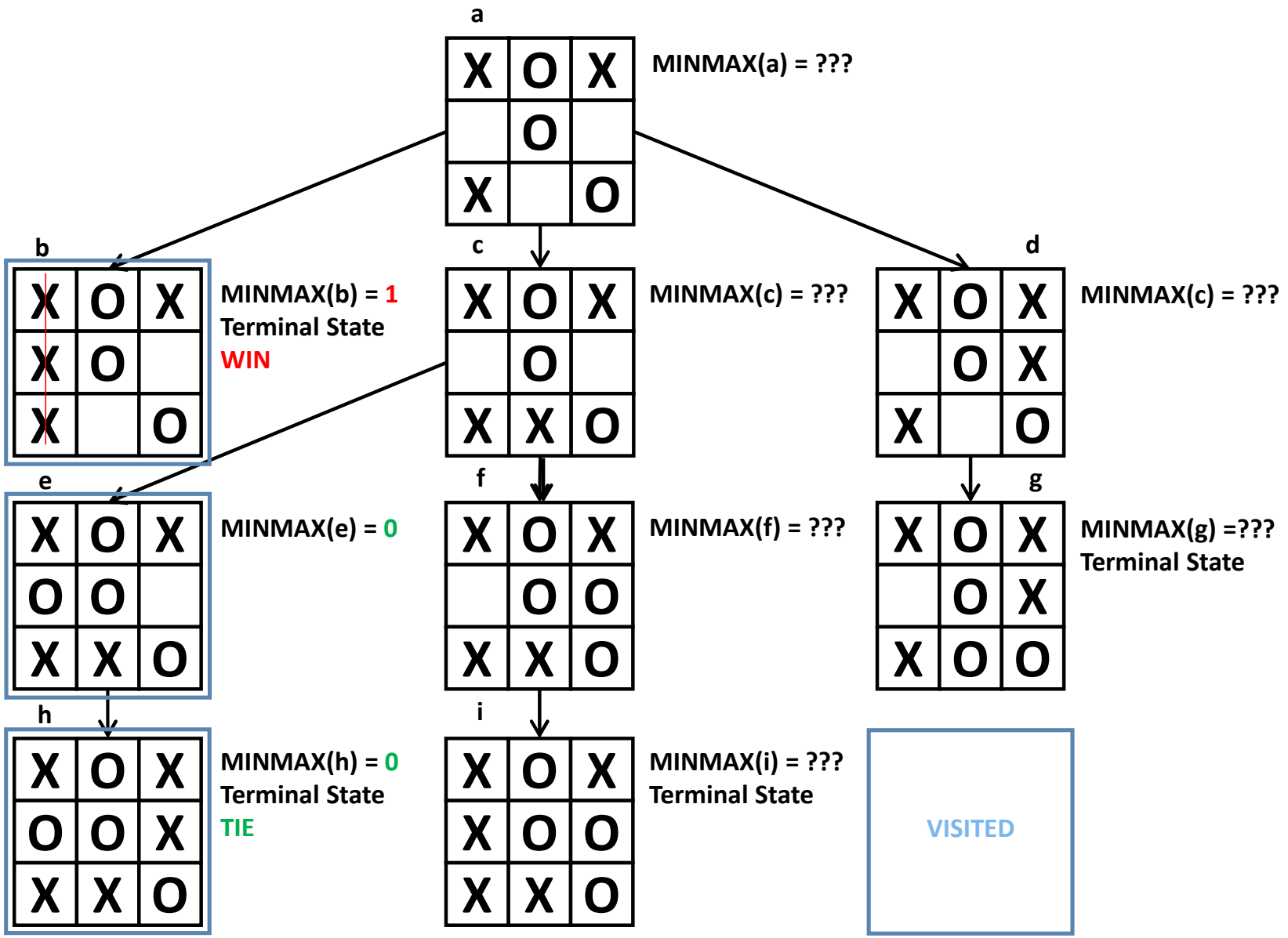
MinMax Algorithm: Tic Tac Toe



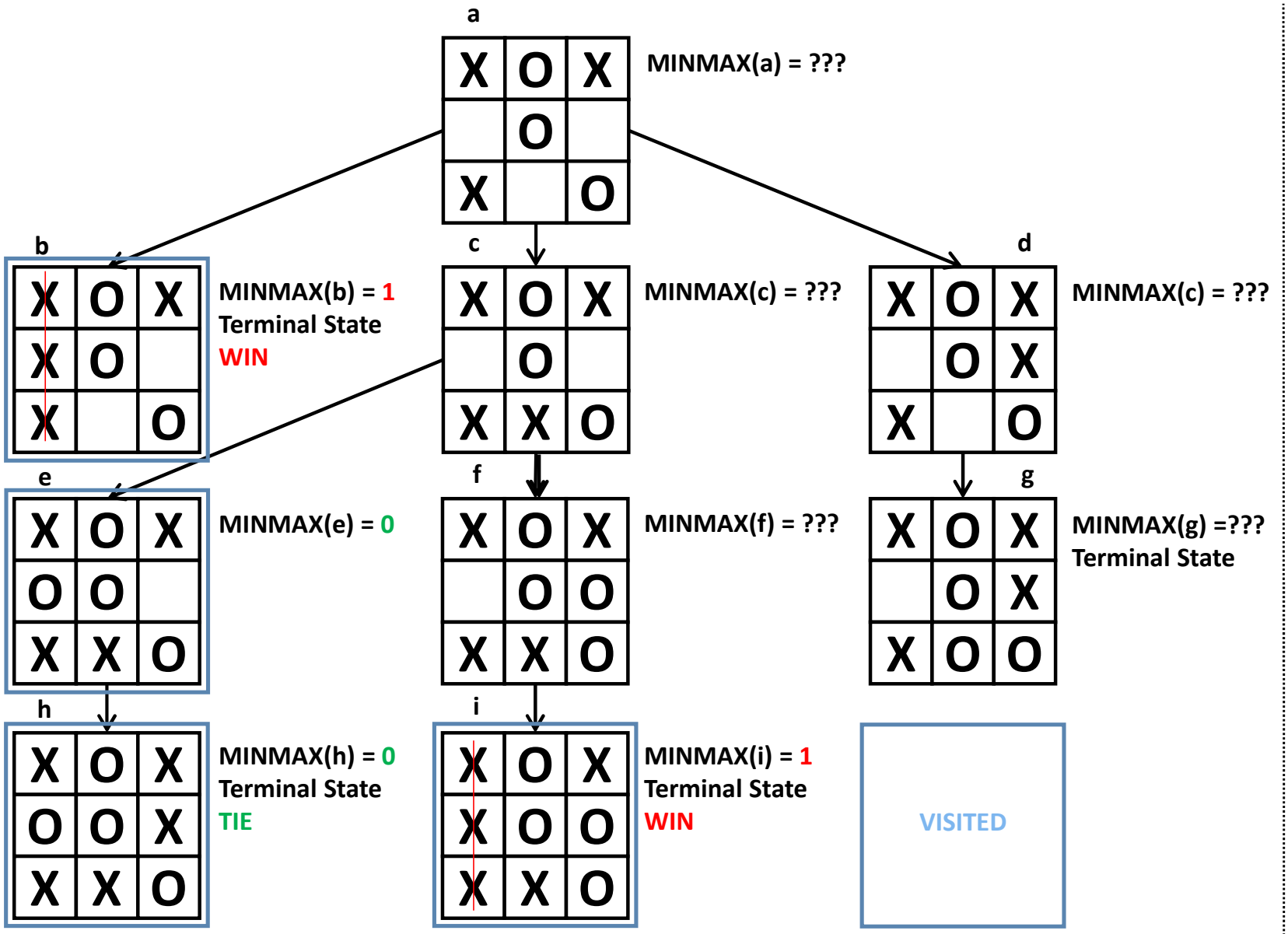
MinMax Algorithm: Tic Tac Toe



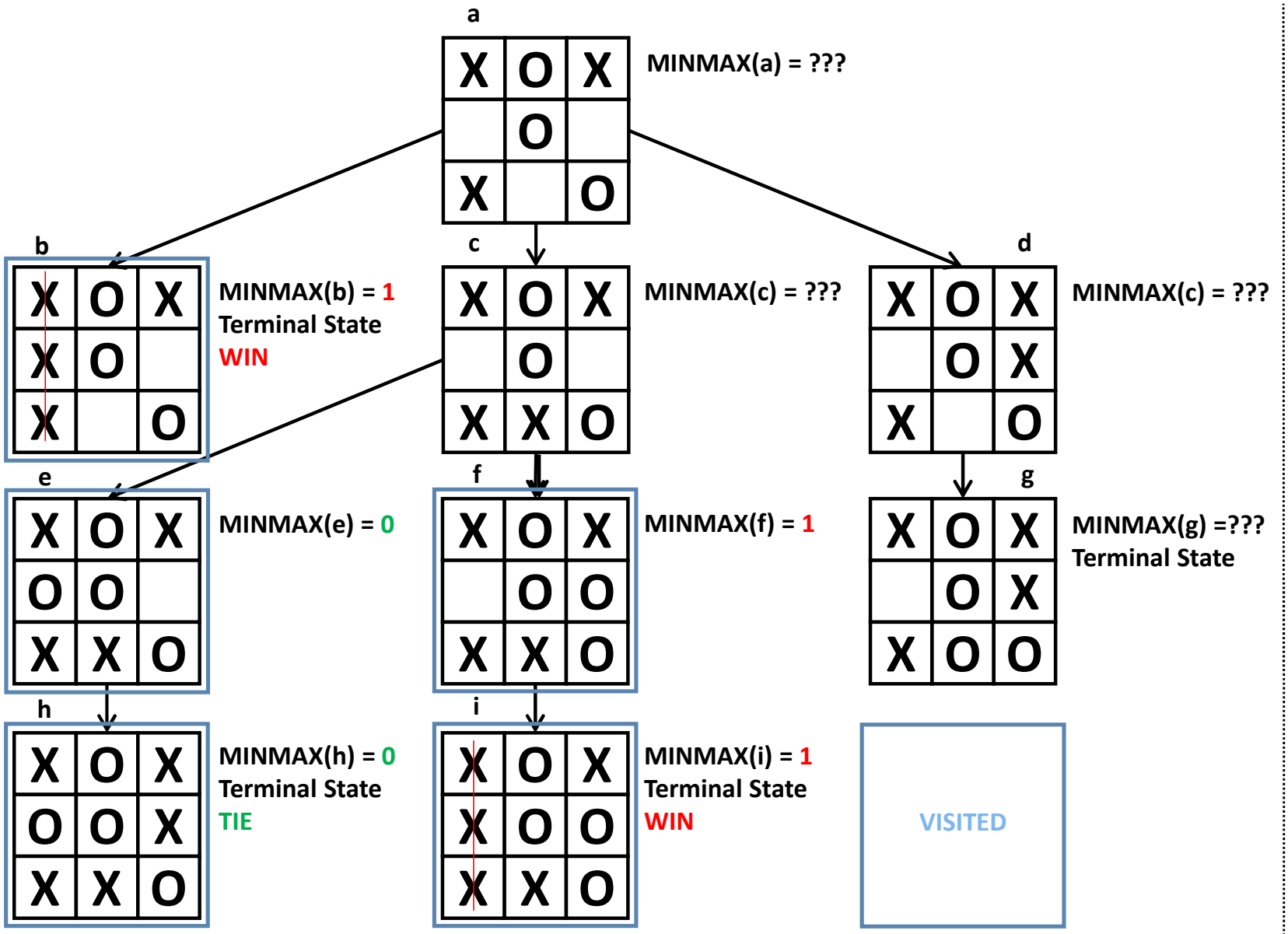
MinMax Algorithm: Tic Tac Toe



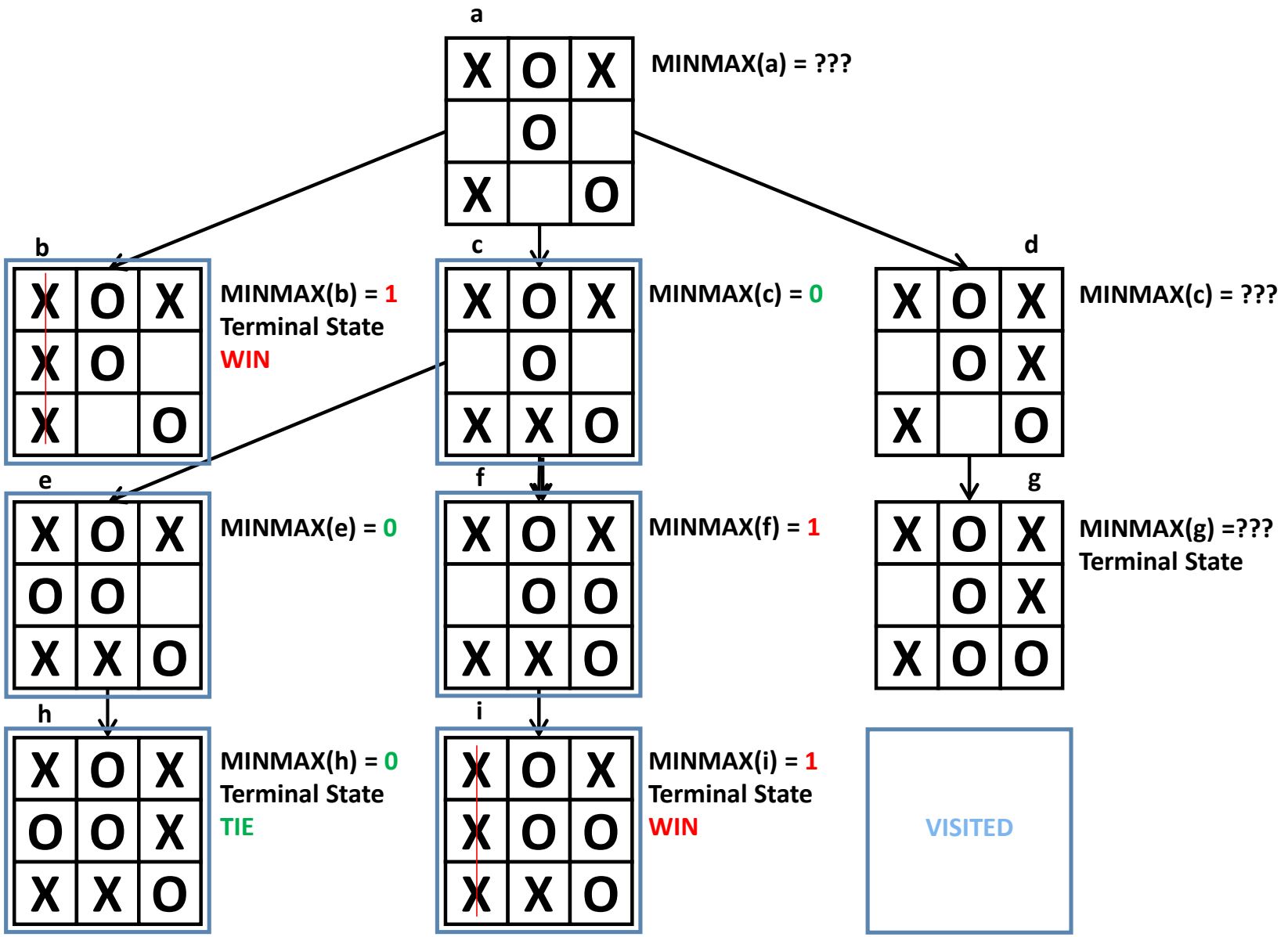
MinMax Algorithm: Tic Tac Toe



MinMax Algorithm: Tic Tac Toe



MinMax Algorithm: Tic Tac Toe



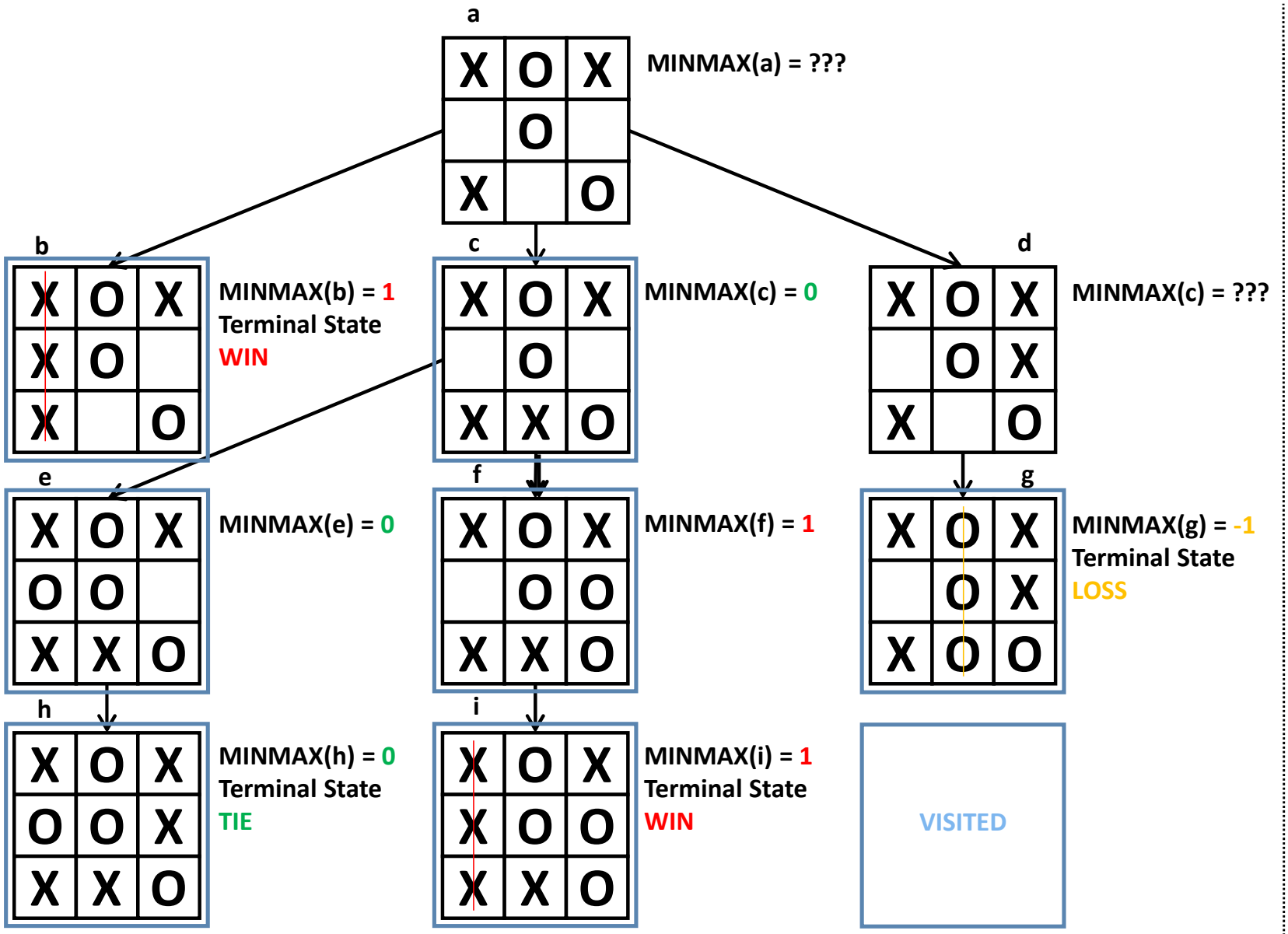
X's move
(MAX Player)

O's move
(MIN Player)

X's move
(MAX Player)

O's move
(MIN Player)

MinMax Algorithm: Tic Tac Toe



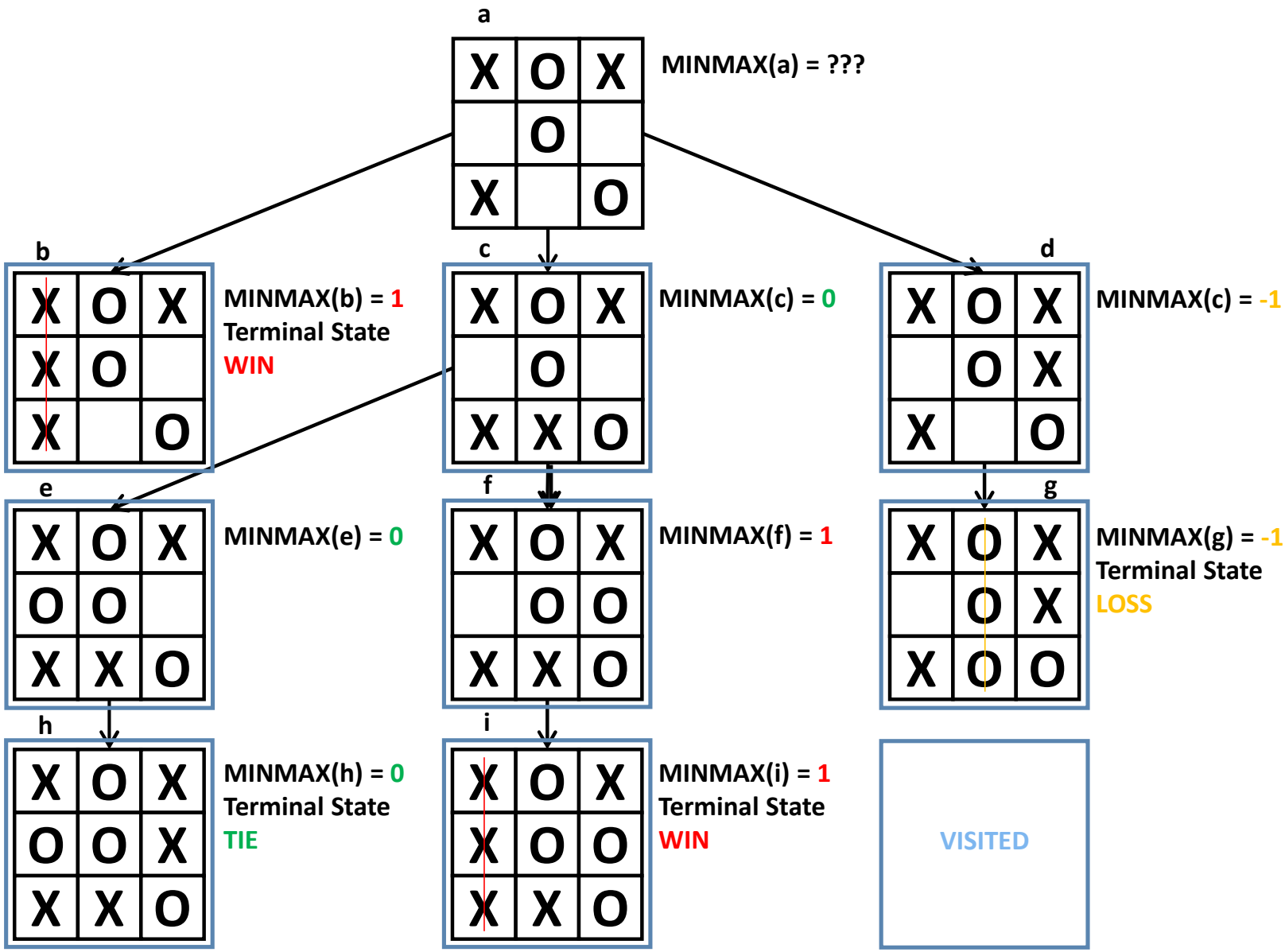
X's move
(MAX Player)

O's move
(MIN Player)

X's move
(MAX Player)

O's move
(MIN Player)

MinMax Algorithm: Tic Tac Toe



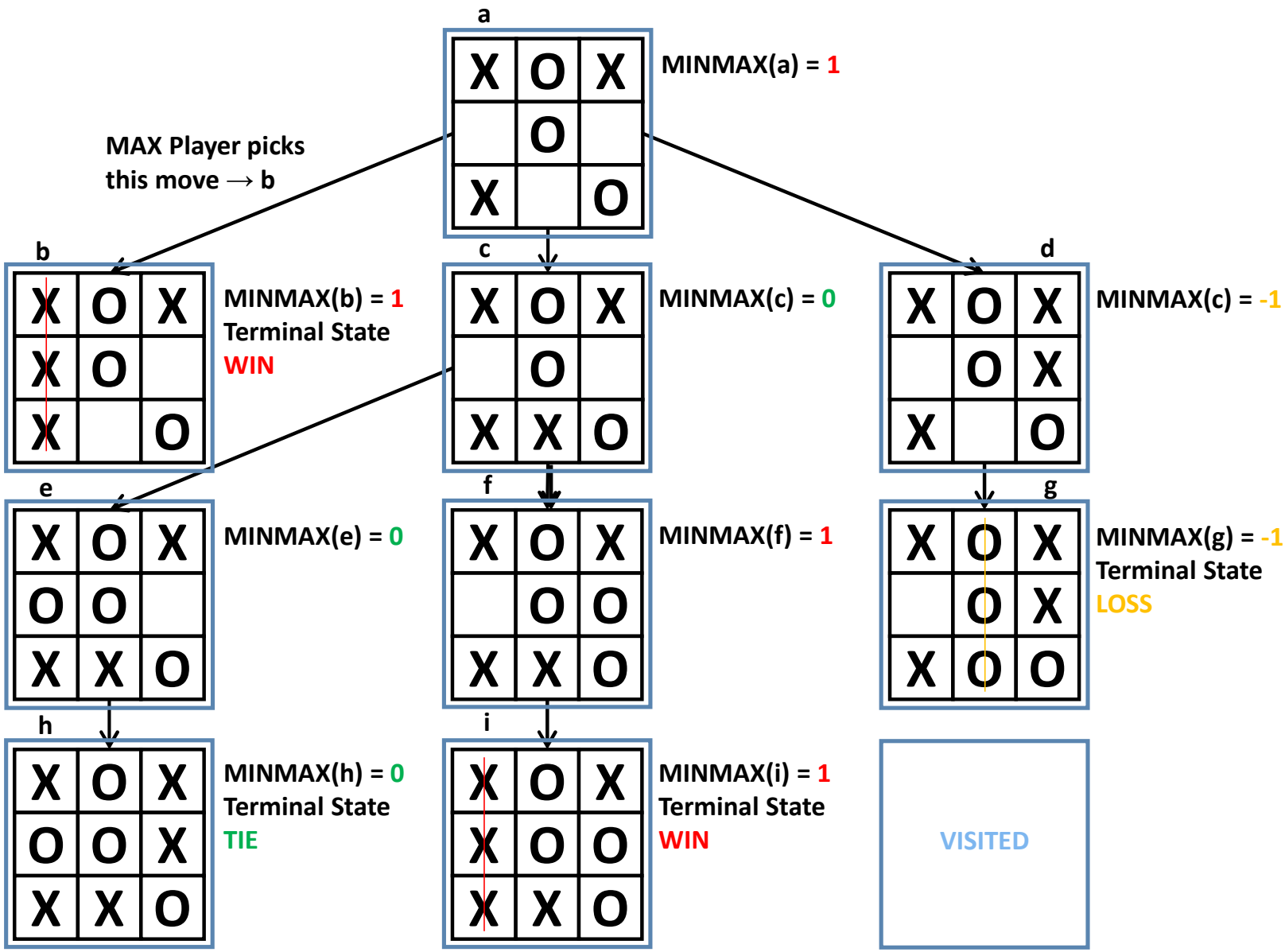
X's move
(MAX Player)

O's move
(MIN Player)

X's move
(MAX Player)

O's move
(MIN Player)

MinMax Algorithm: Tic Tac Toe



X's move (MAX Player)

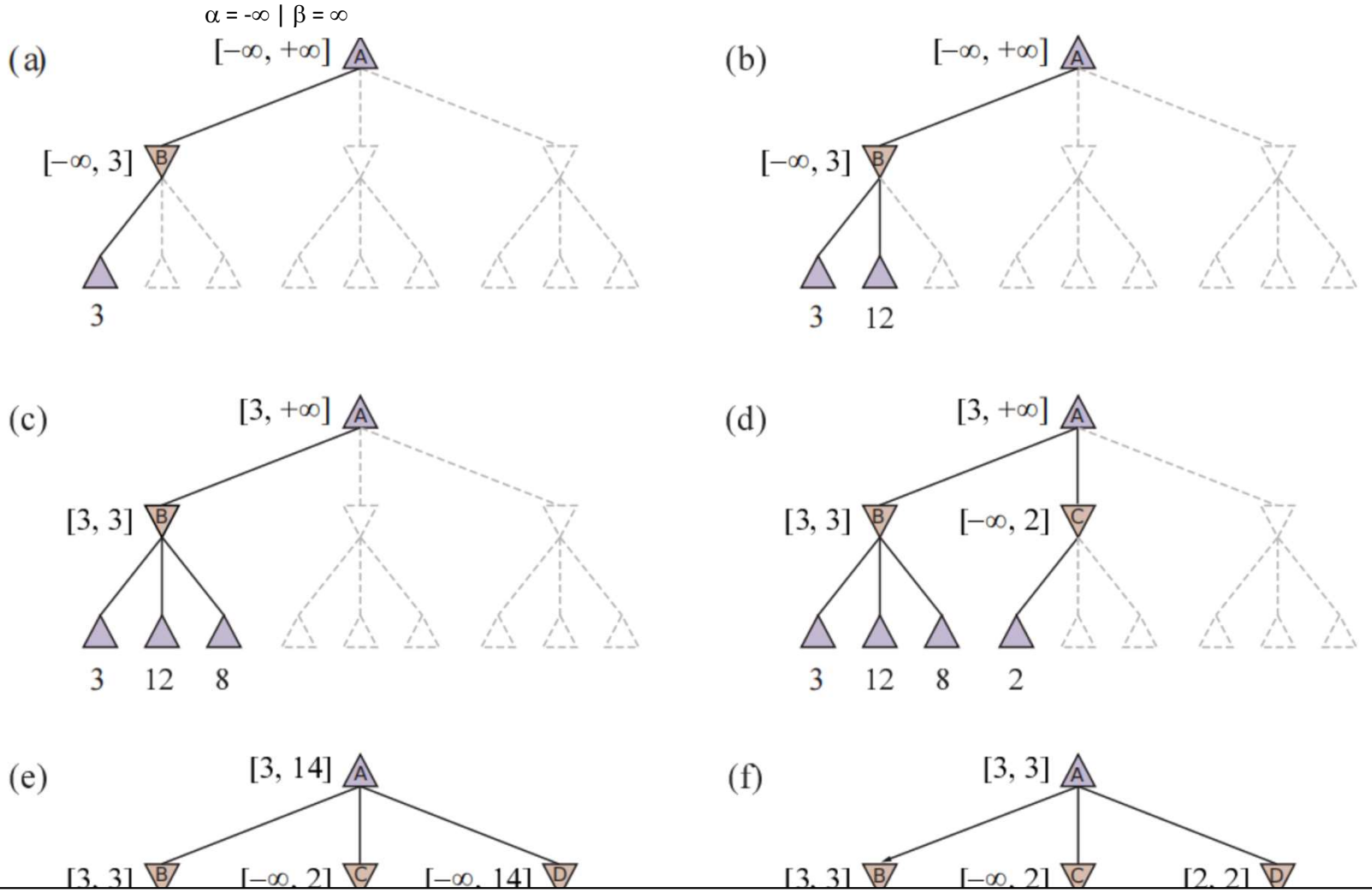
O's move (MIN Player)

X's move (MAX Player)

O's move (MIN Player)

MinMax: What is the Challenge?

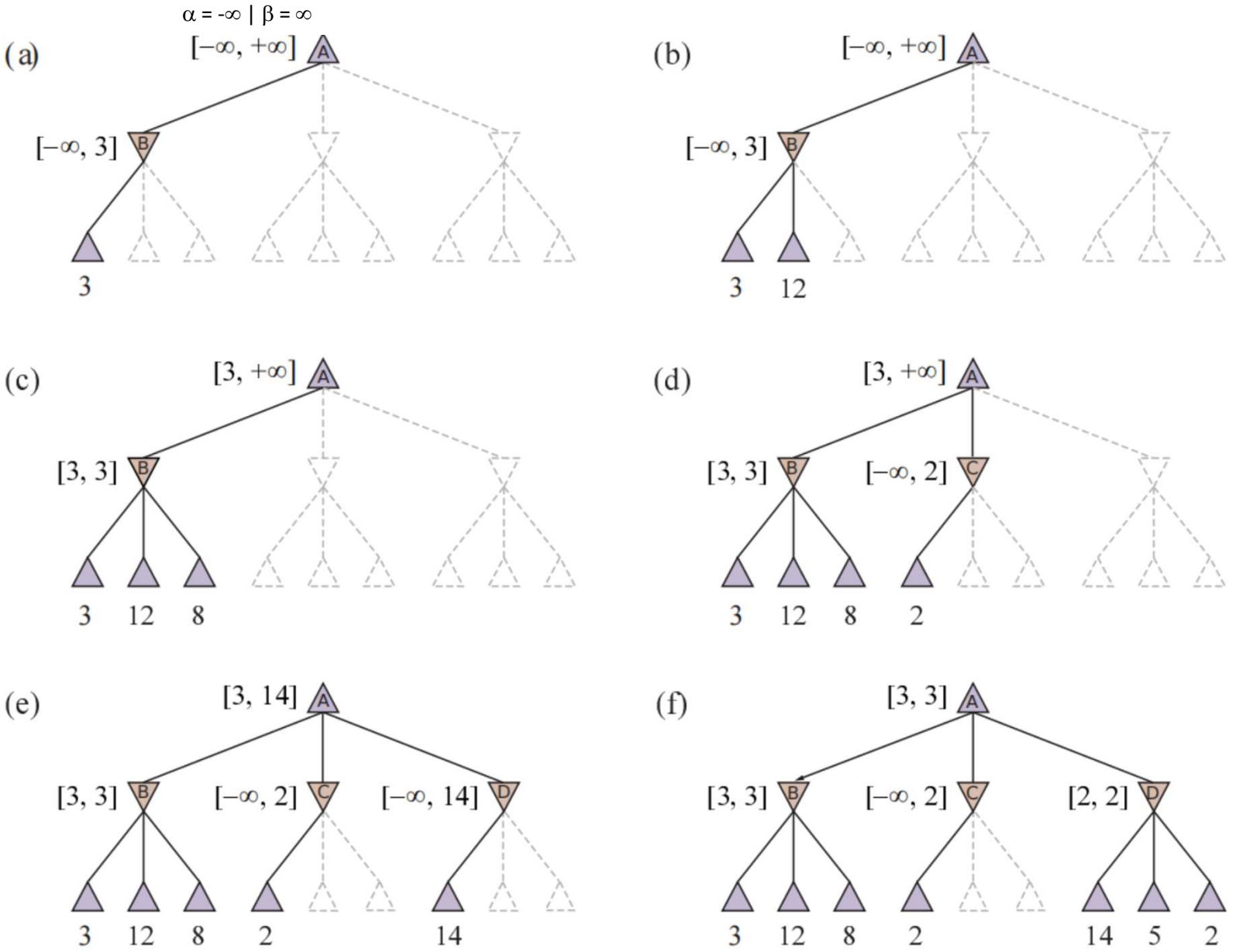
Example MinMax with α-β Pruning



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")

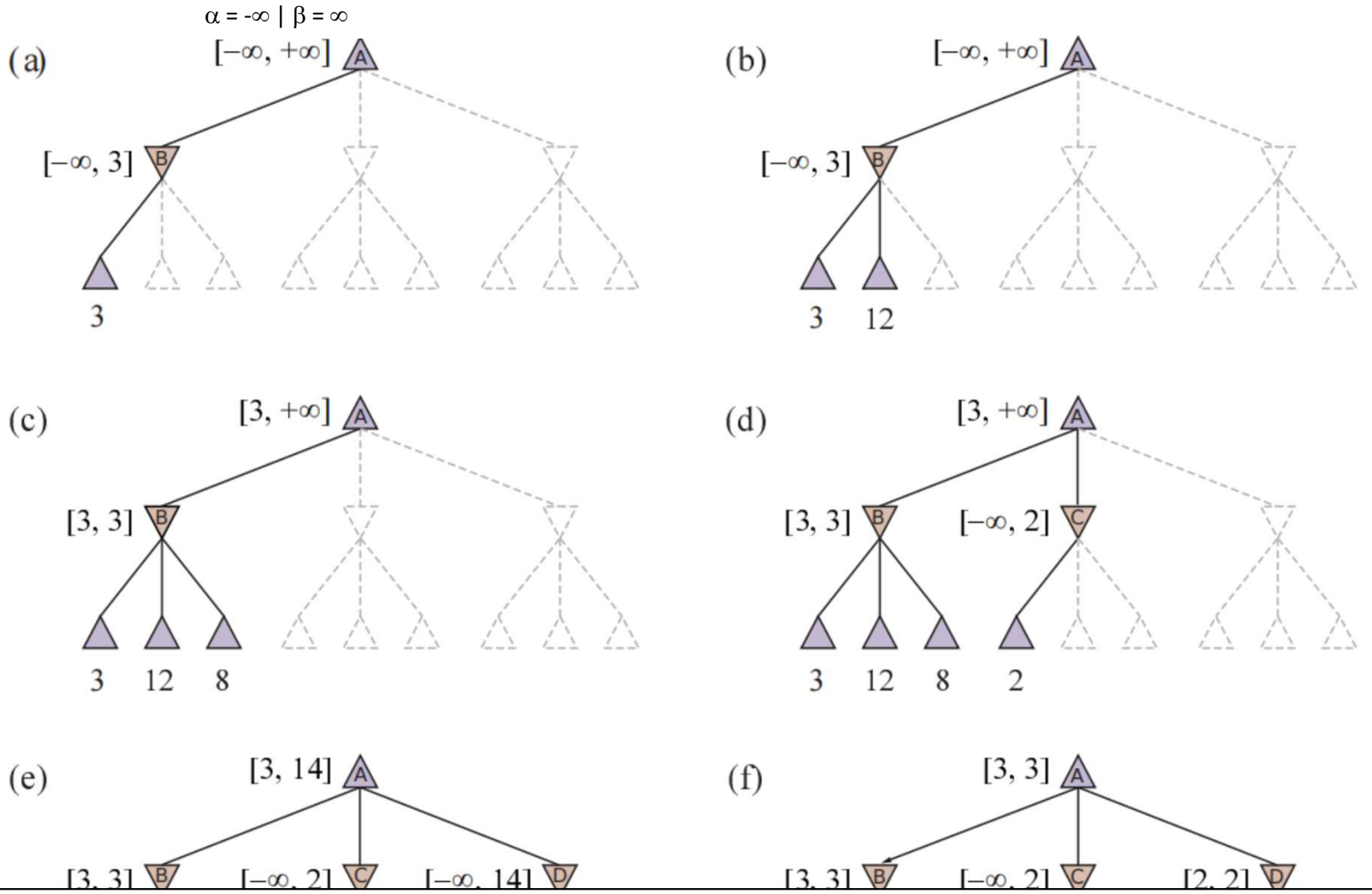
Example MinMax with α-β Pruning



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
```

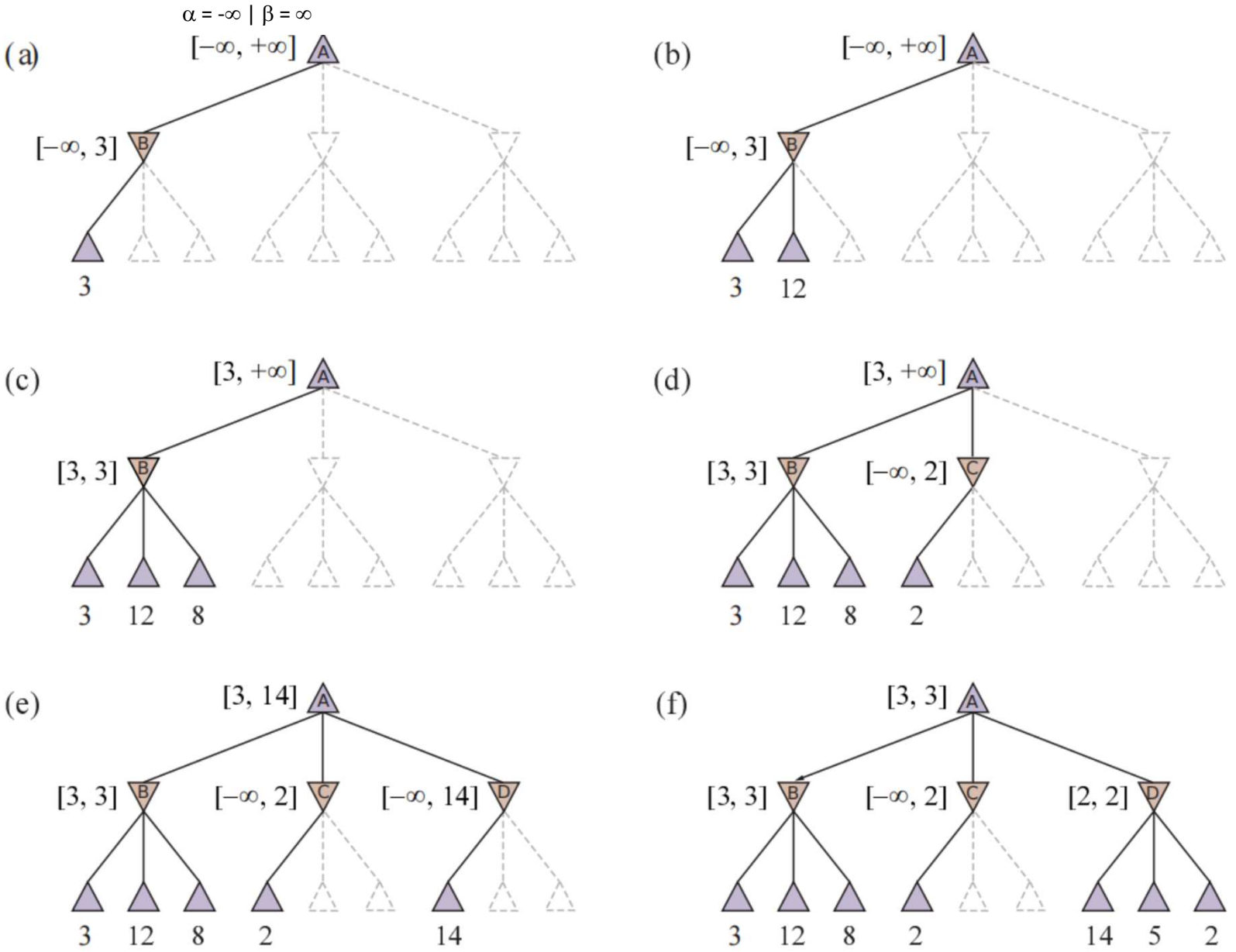
Example MinMax with α-β Pruning



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")

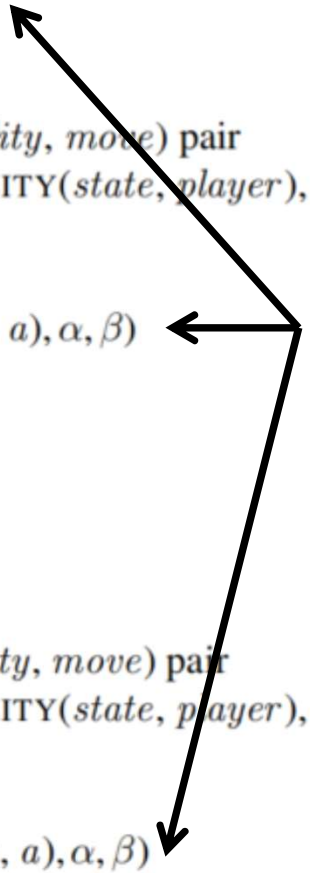
β: the value of the best (lowest-value) choice we have found so far at any choice point along the path for MIN player ("at most")

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 ← 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
```

Go through all legal actions/moves (subtrees) recursively

MAX Player's 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
```

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!)

MAX 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!)

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
```

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!)

MAX Player does NOT change bound β here!

MAX Player's move

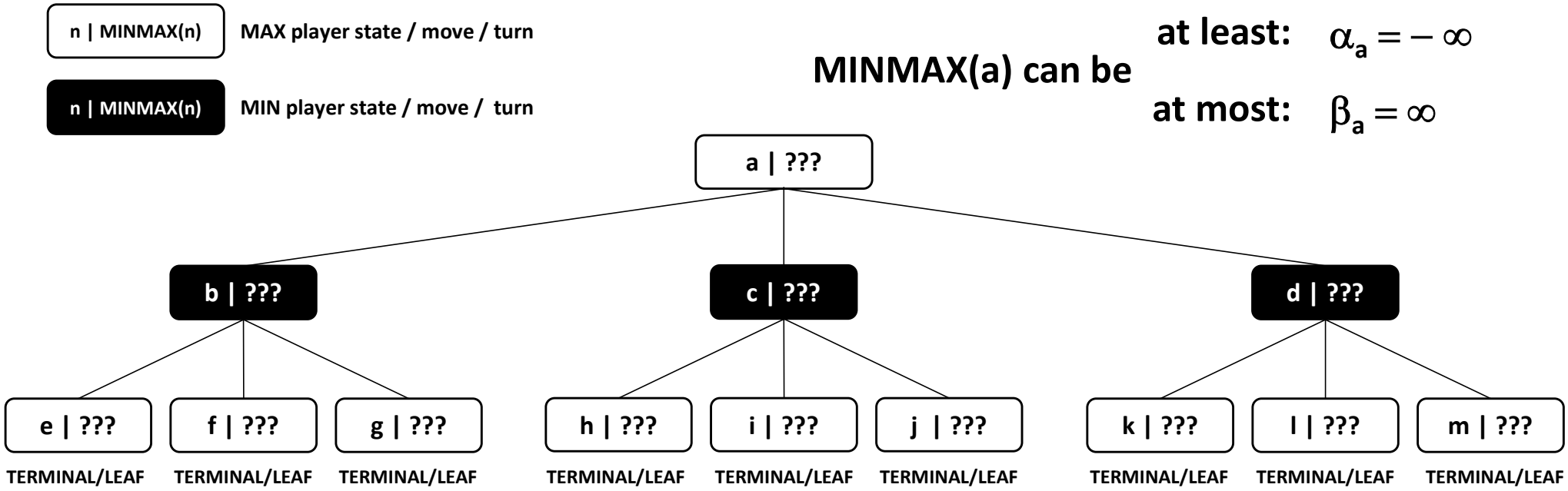
Go through all legal actions/moves (subtrees) recursively

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

MIN Player does NOT change bound α here!

MIN Player's move

MinMax with α-β: Example

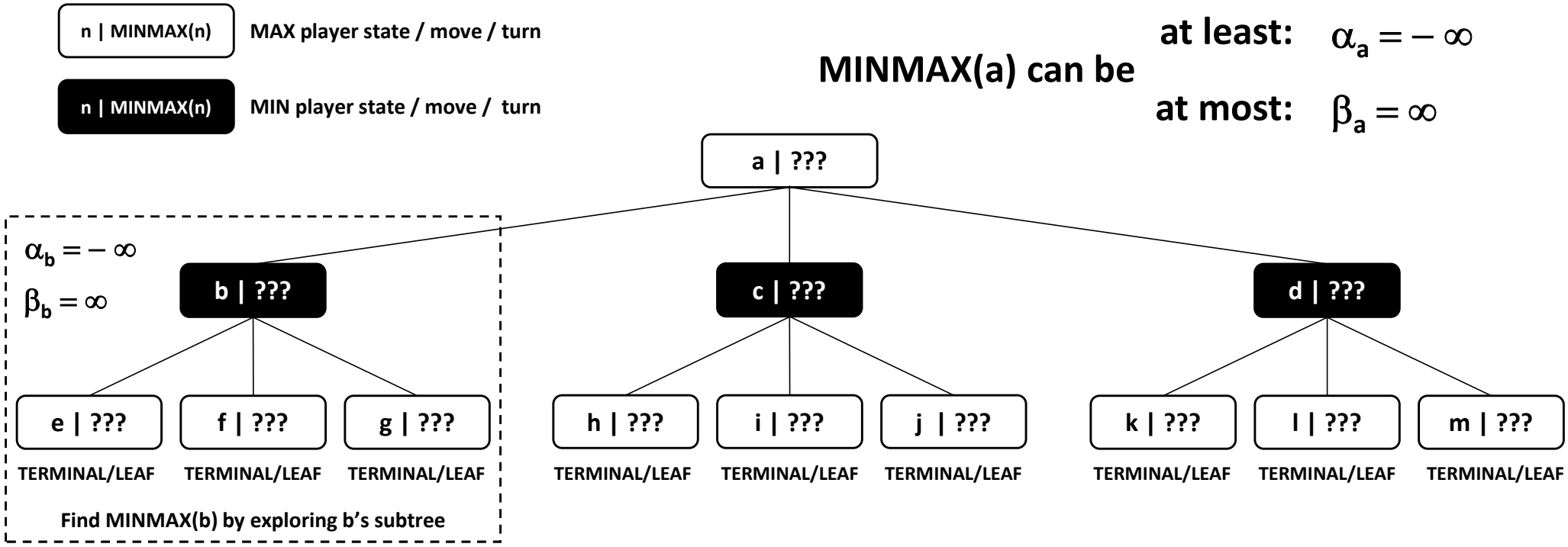


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **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

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:

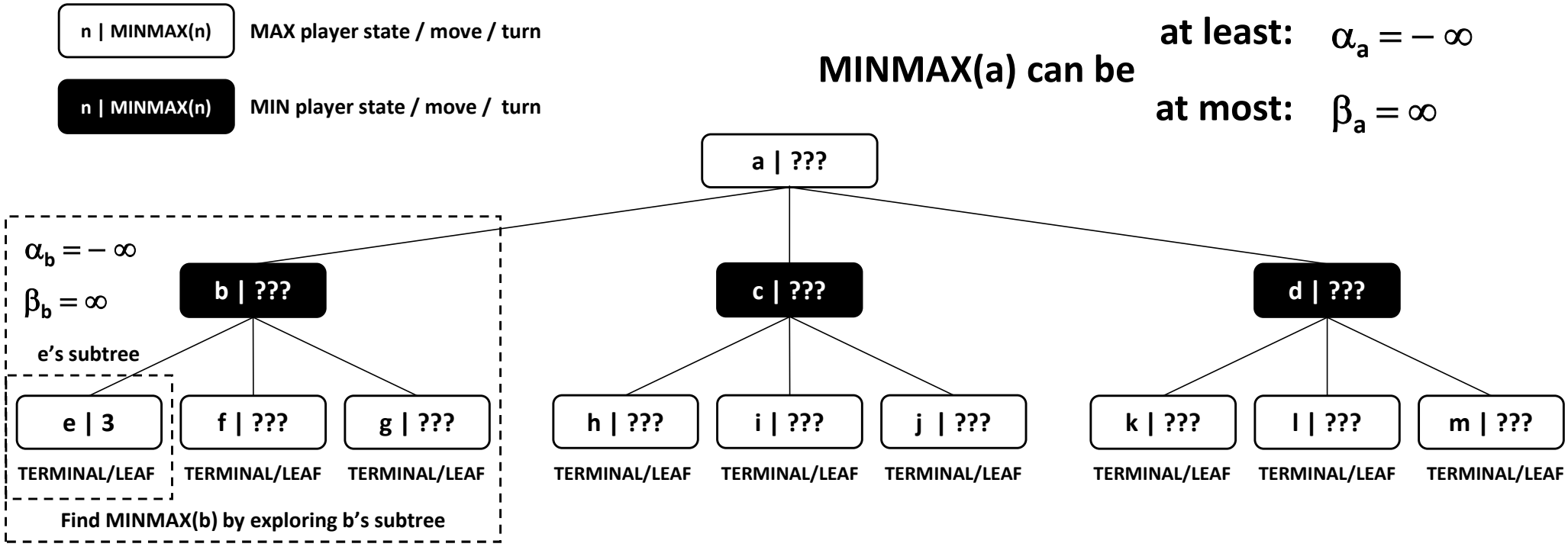
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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:

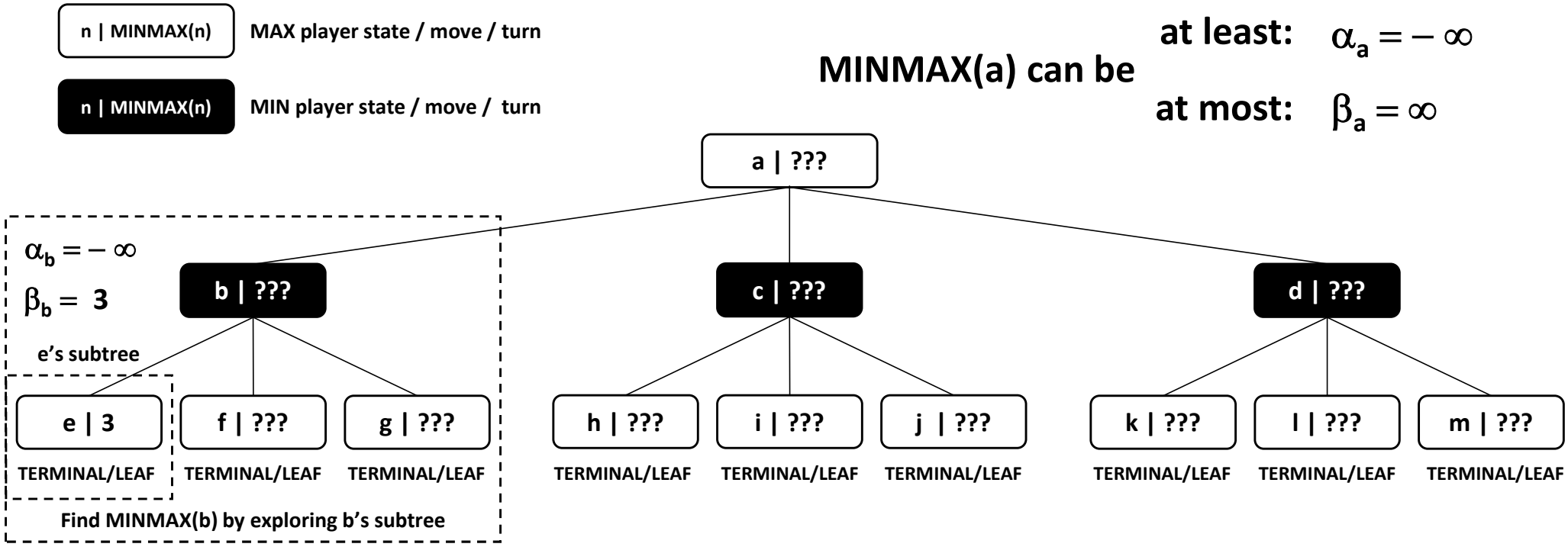
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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:

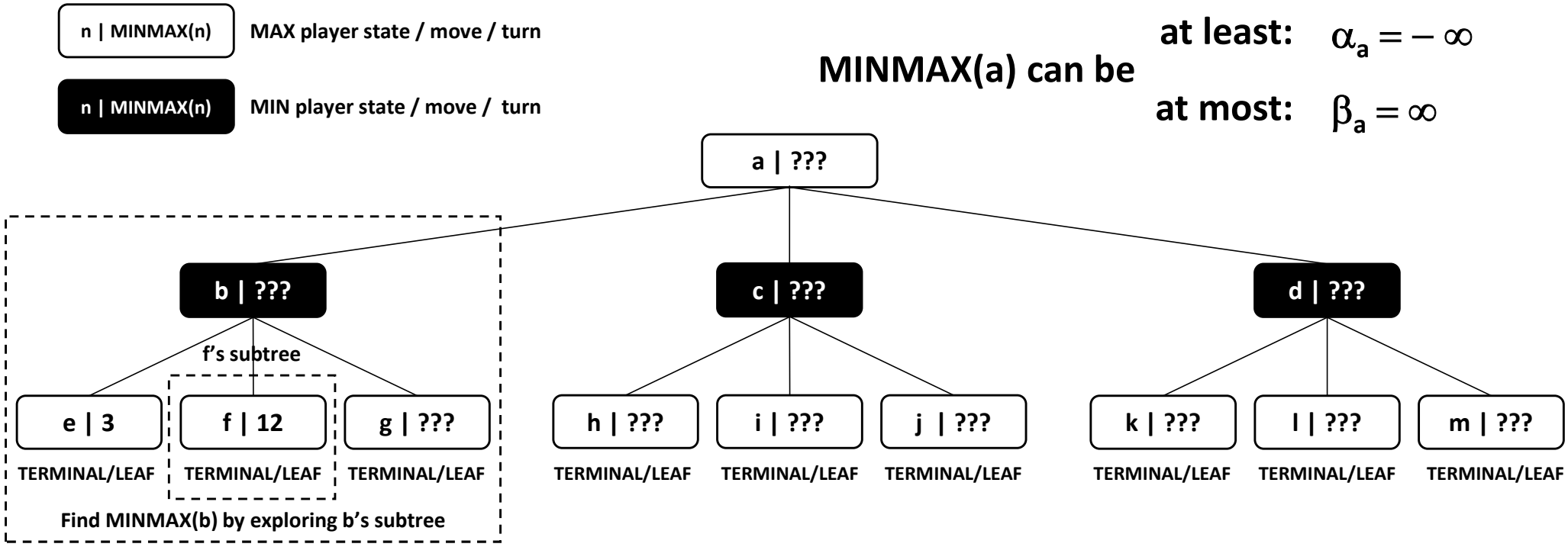
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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 < ∞) → v = v2 = 3 → β_b = min(β_b, v) = min(∞, 3) = 3**
- **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:

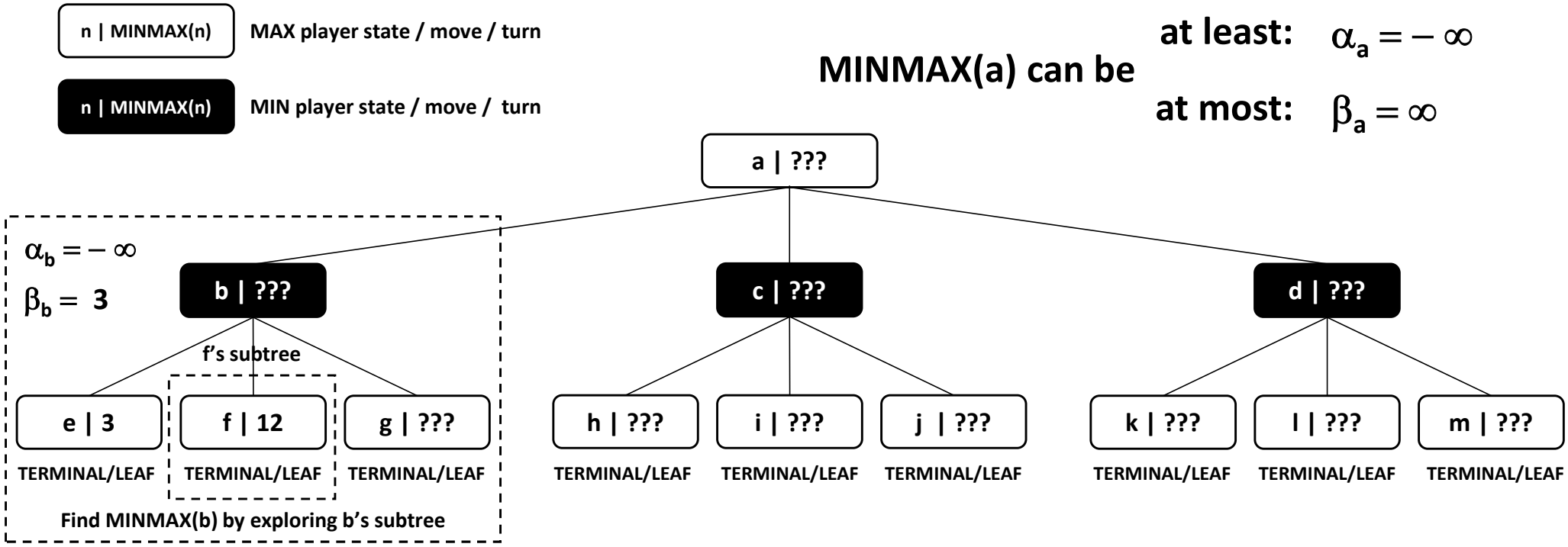
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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 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



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:

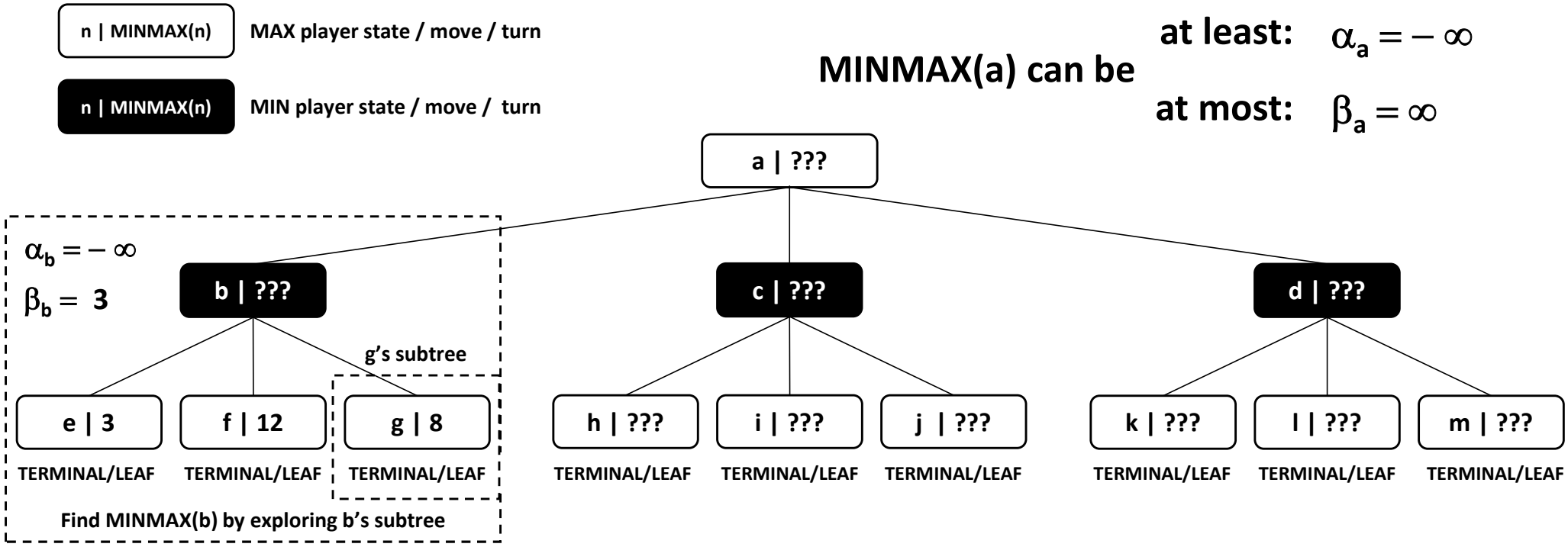
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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 > $-\infty$) → 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:

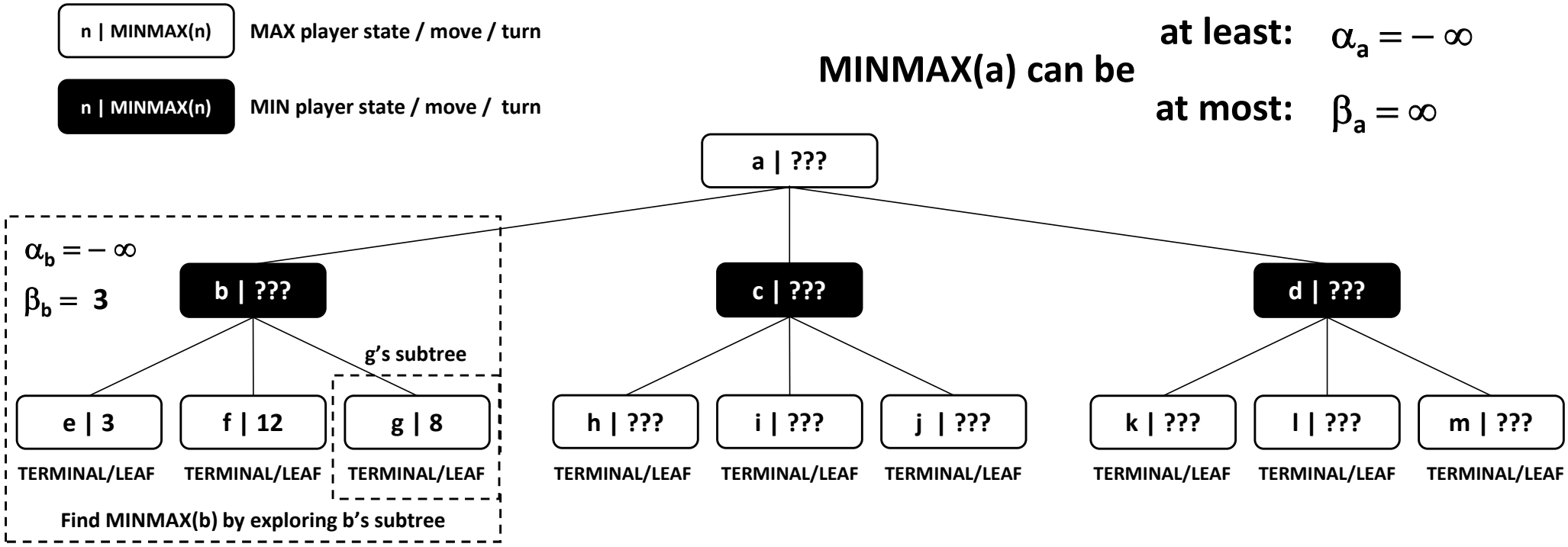
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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 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:

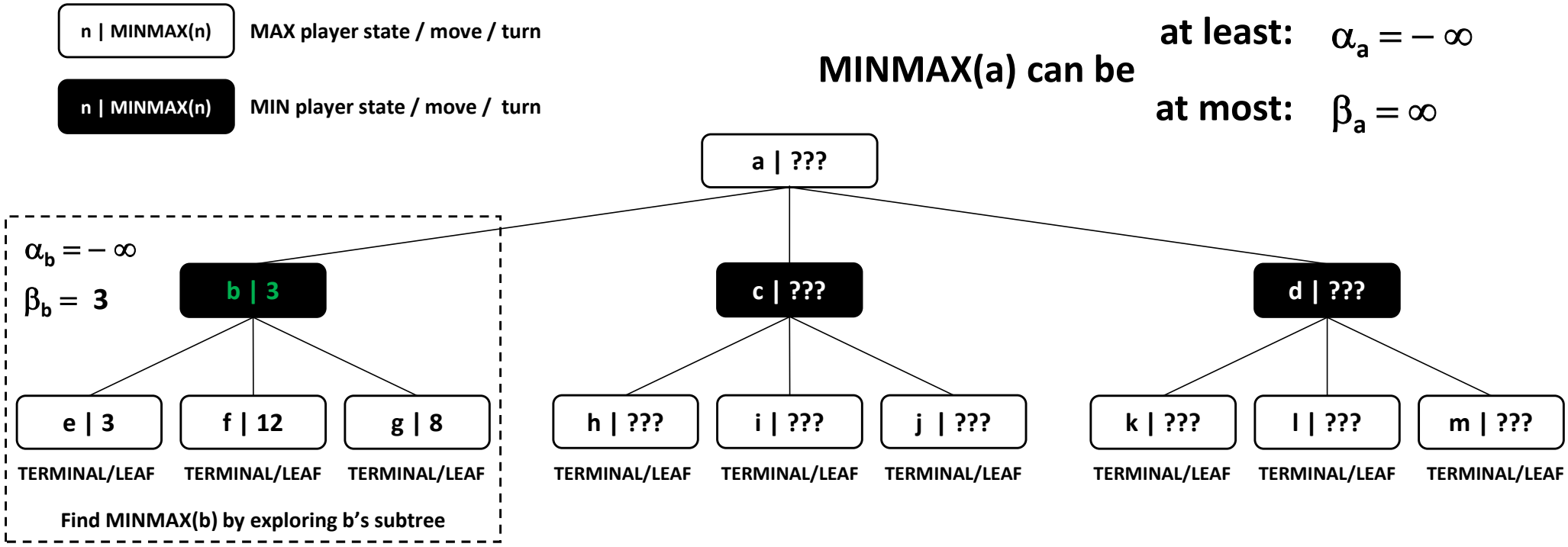
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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:

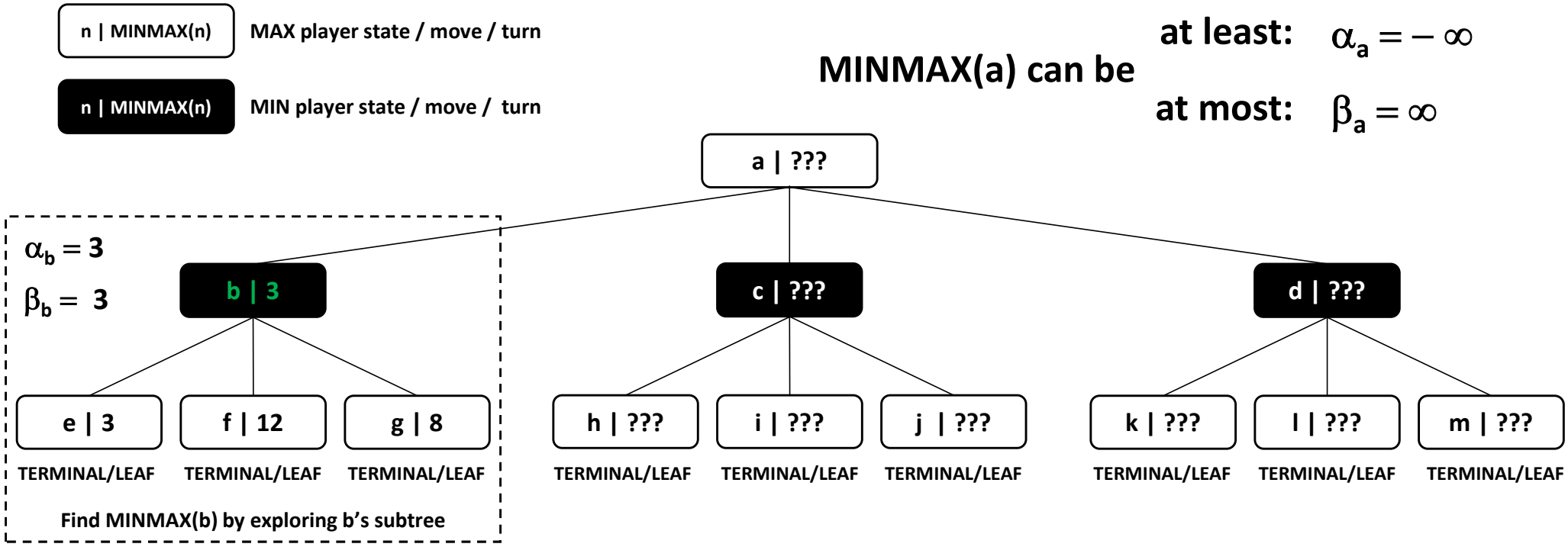
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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)**
- **$v > \alpha_a$ $(3 > -\infty)$ → 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:

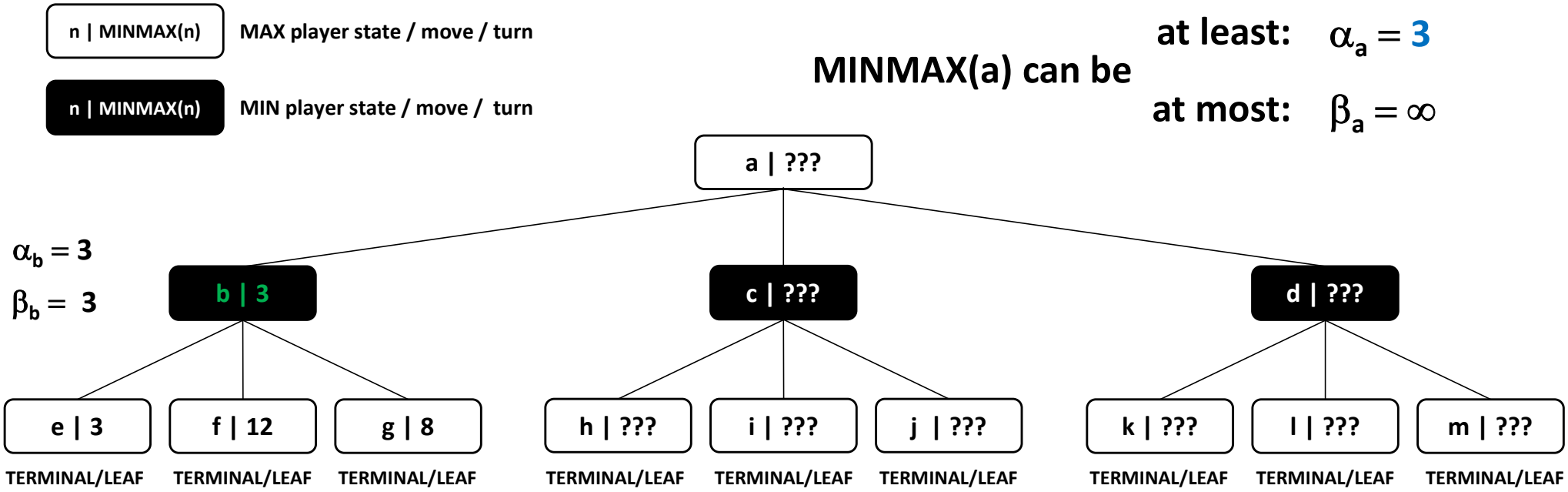
- **MINMAX(b) = UNKNOWN | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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

MIN Player explored entire b's subtree:

- **We know the exact value of MINMAX(b) → $\alpha_b = 3$**

MinMax with α-β: Example

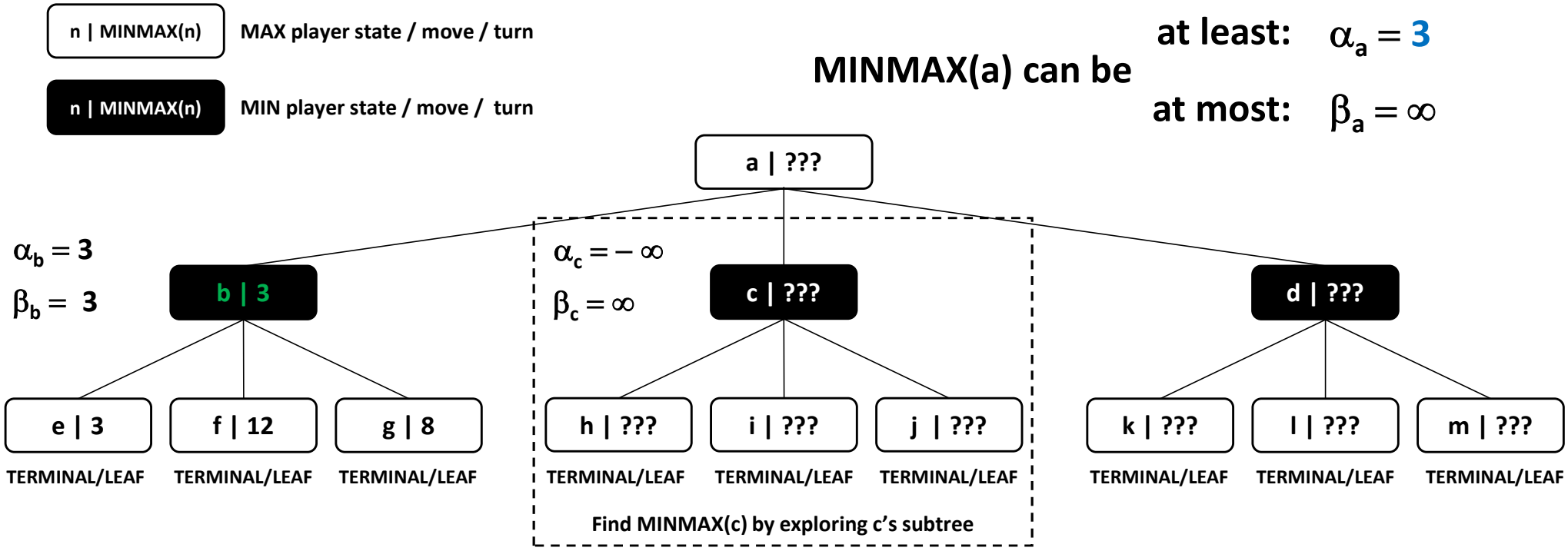


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

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

MINMAX(a) = max(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:

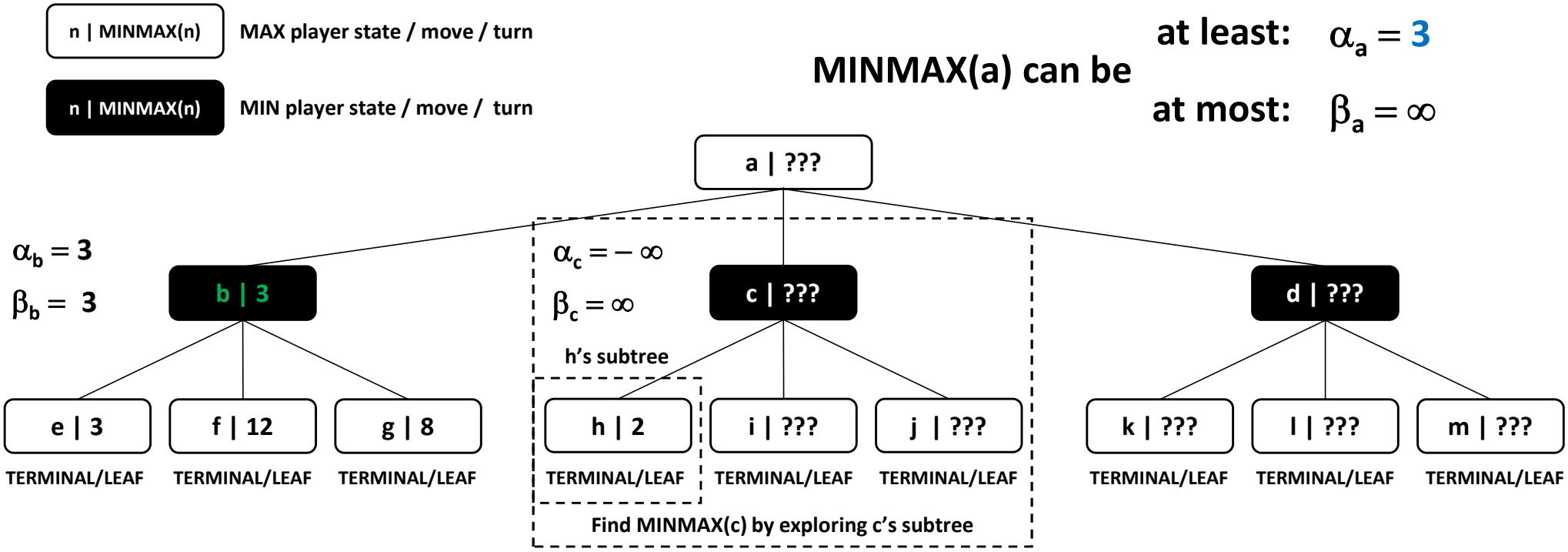
- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  MINMAX(a) = max(3, MINMAX(c), MINMAX(d)) = max(3, ???, ???) → 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 → min MINMAX seen: v = ∞**
- **v > α_a (∞ > 3) → 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:

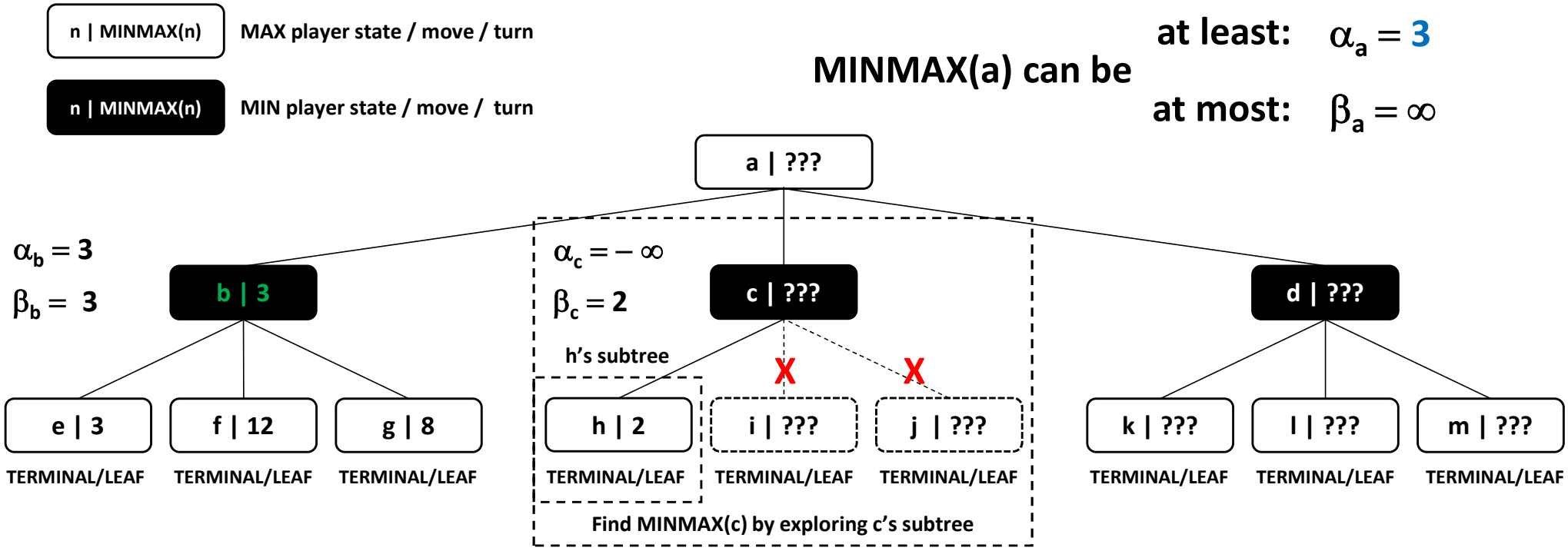
- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  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



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:

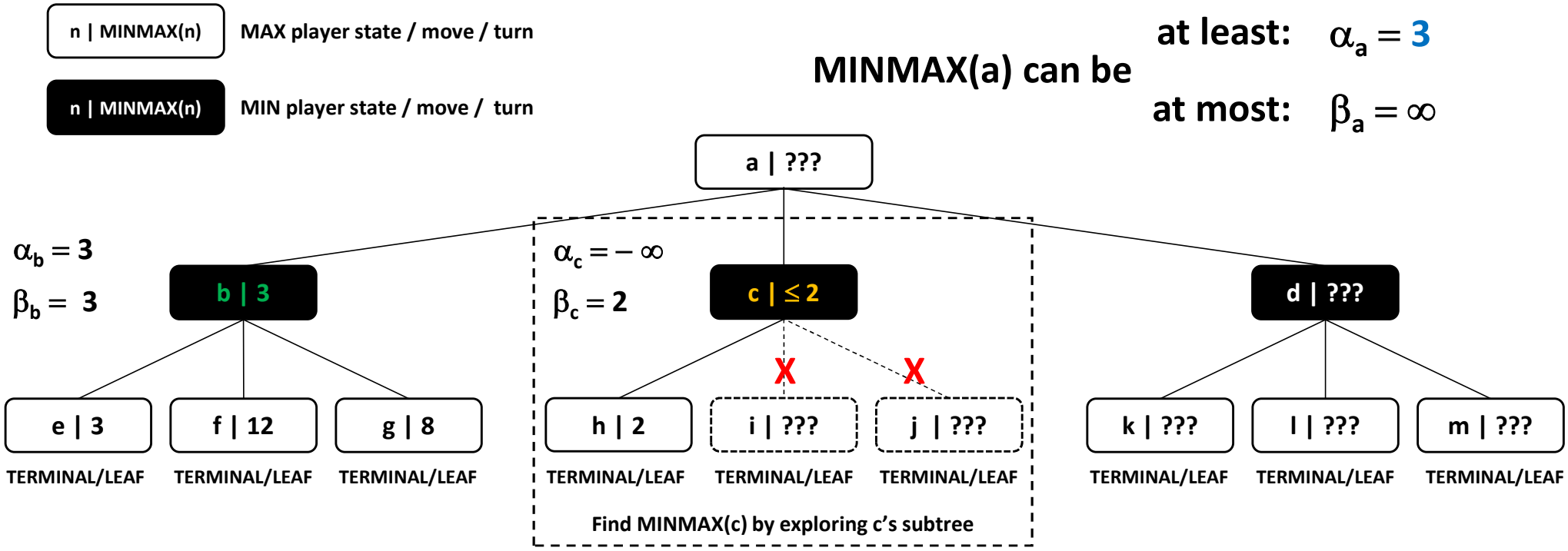
- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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

MIN Player needs to explore c's subtree:

- **$v2 < v$ $(2 < \infty) \rightarrow v = v2 = 2 \rightarrow \beta_c = \min(\beta_c, v) = \min(\infty, 2) = 2$**
- **$v < \alpha_a$ $(2 < 3) \rightarrow$ 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:

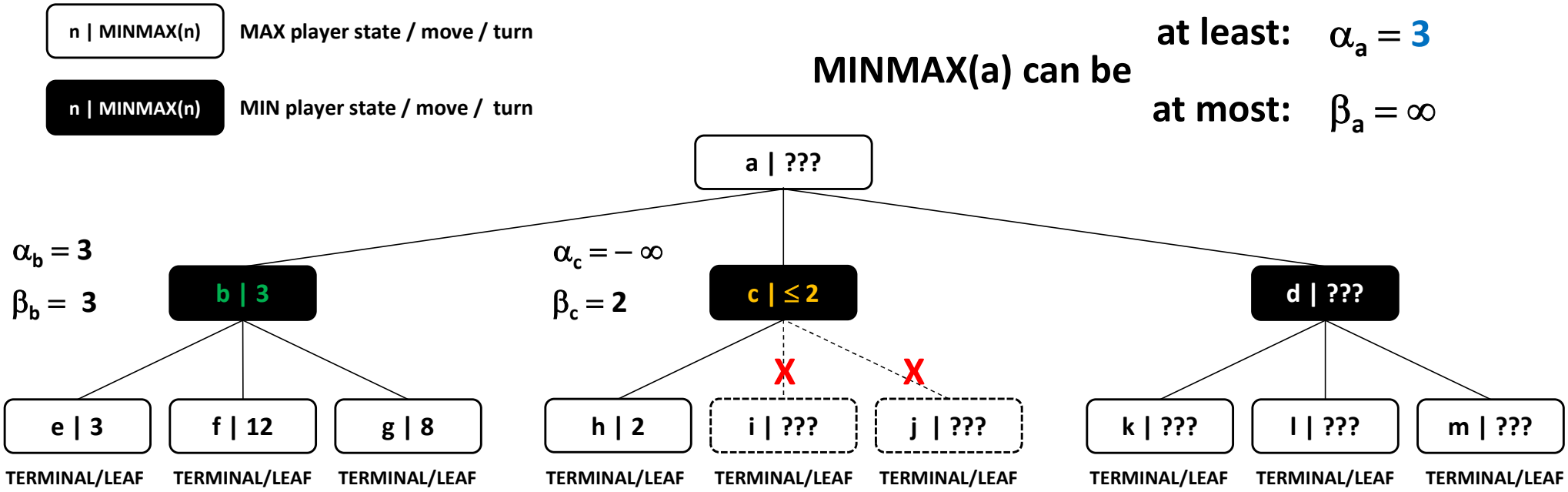
- **MINMAX(b) = 3 | MINMAX(c) = UNKNOWN | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  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

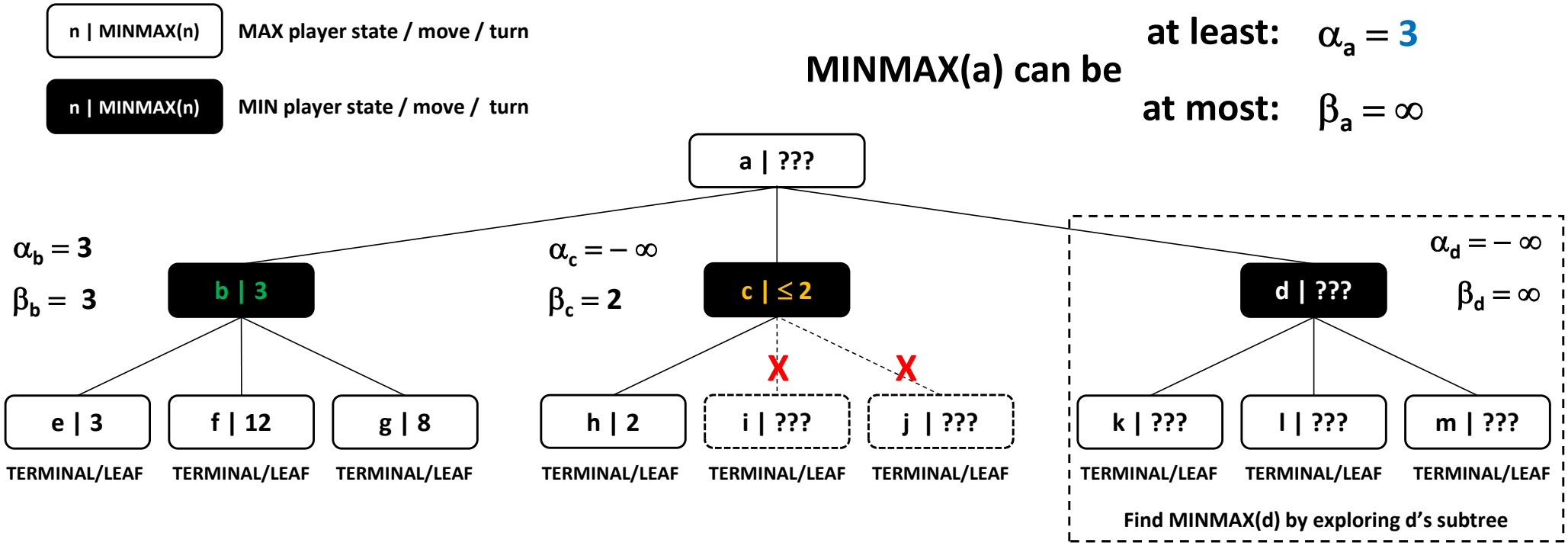


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

- **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:

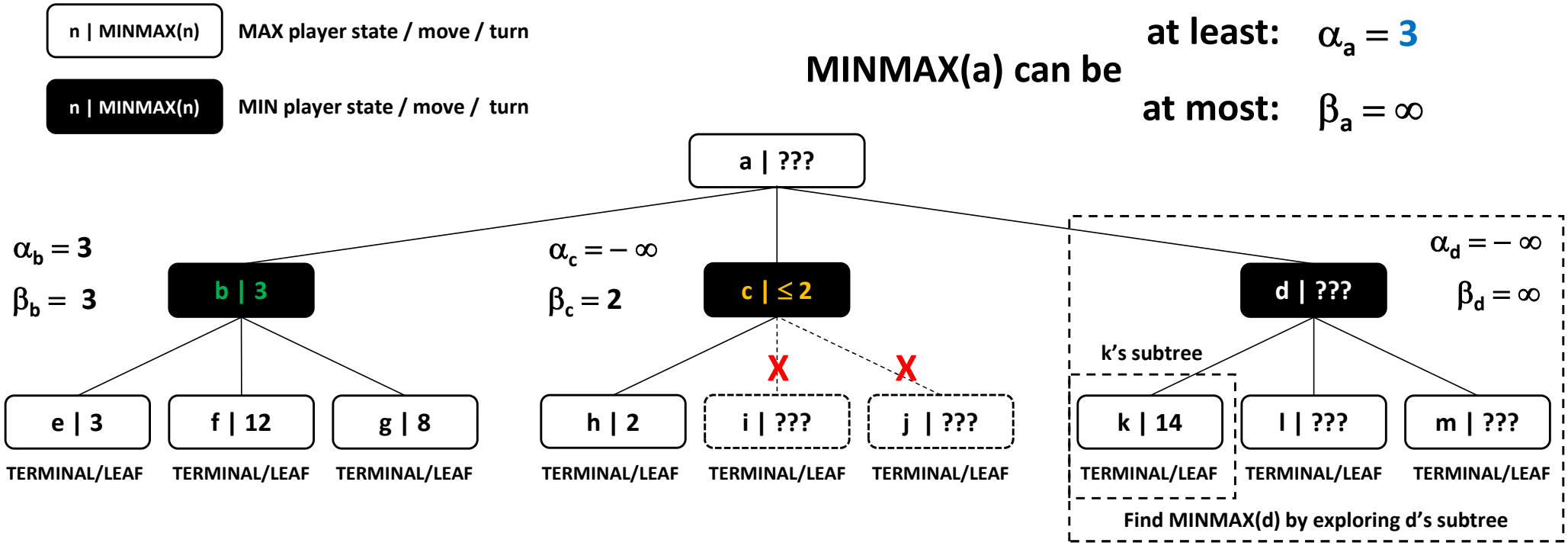
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  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 > α_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:

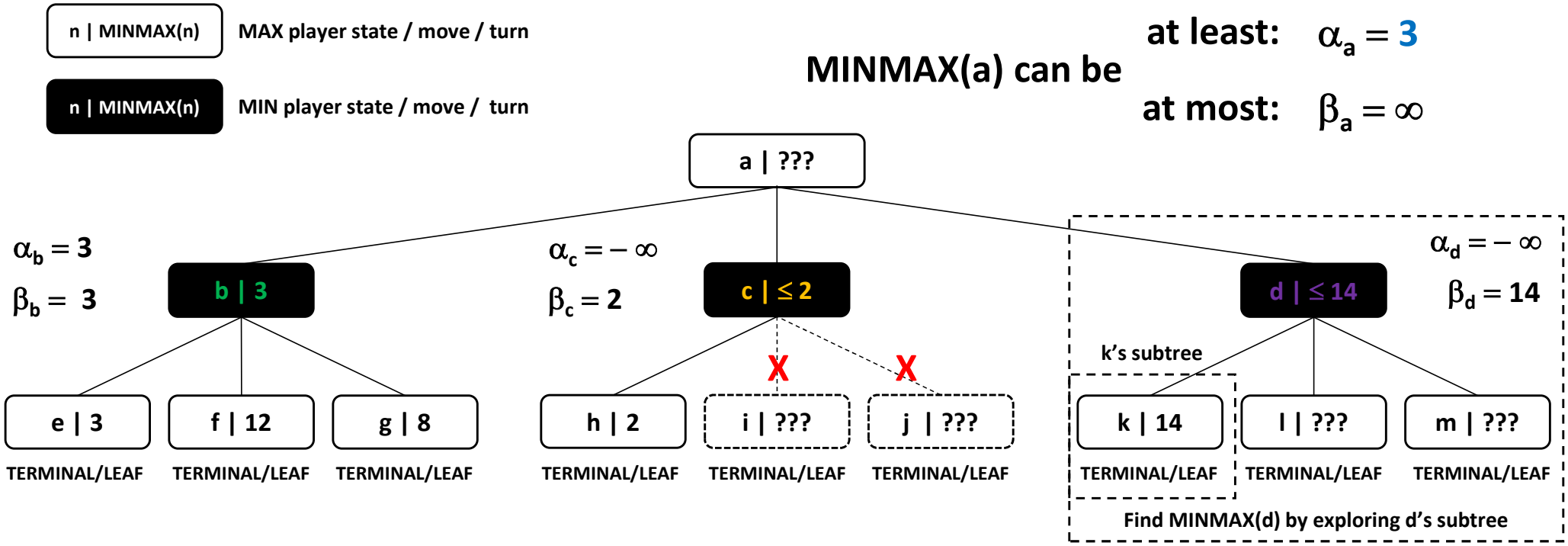
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

  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



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:

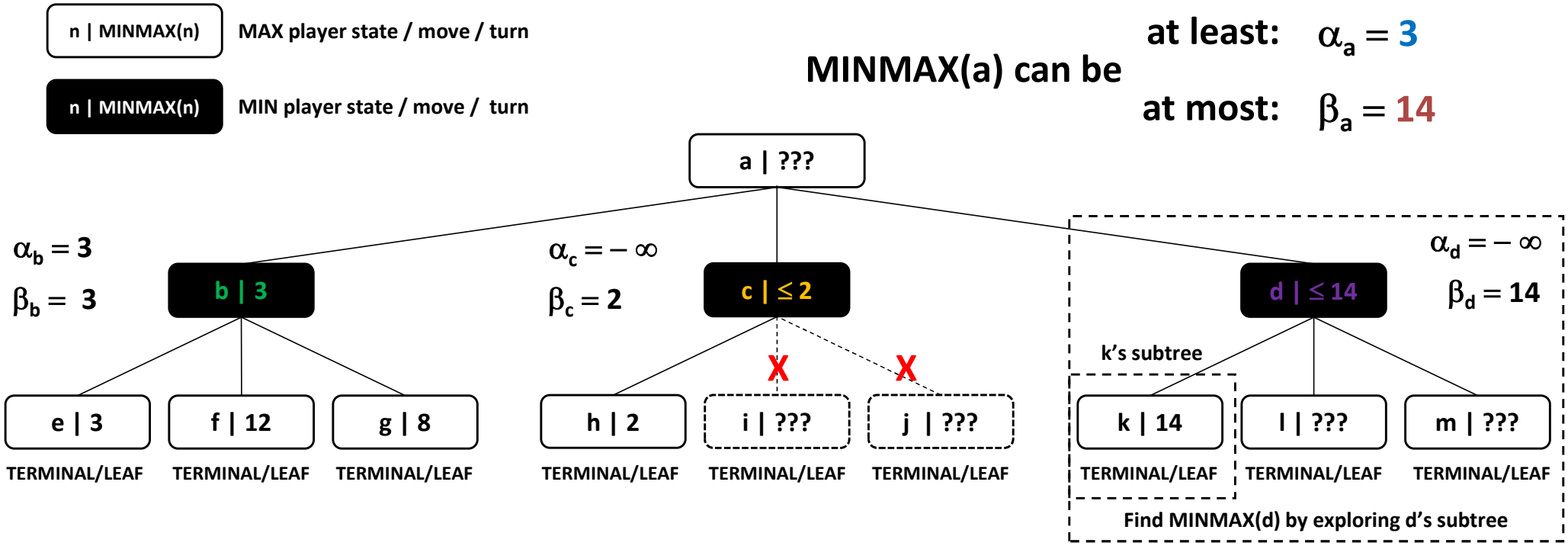
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = UNKNOWN**
- **MAX Player's decision: not enough information yet.**

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

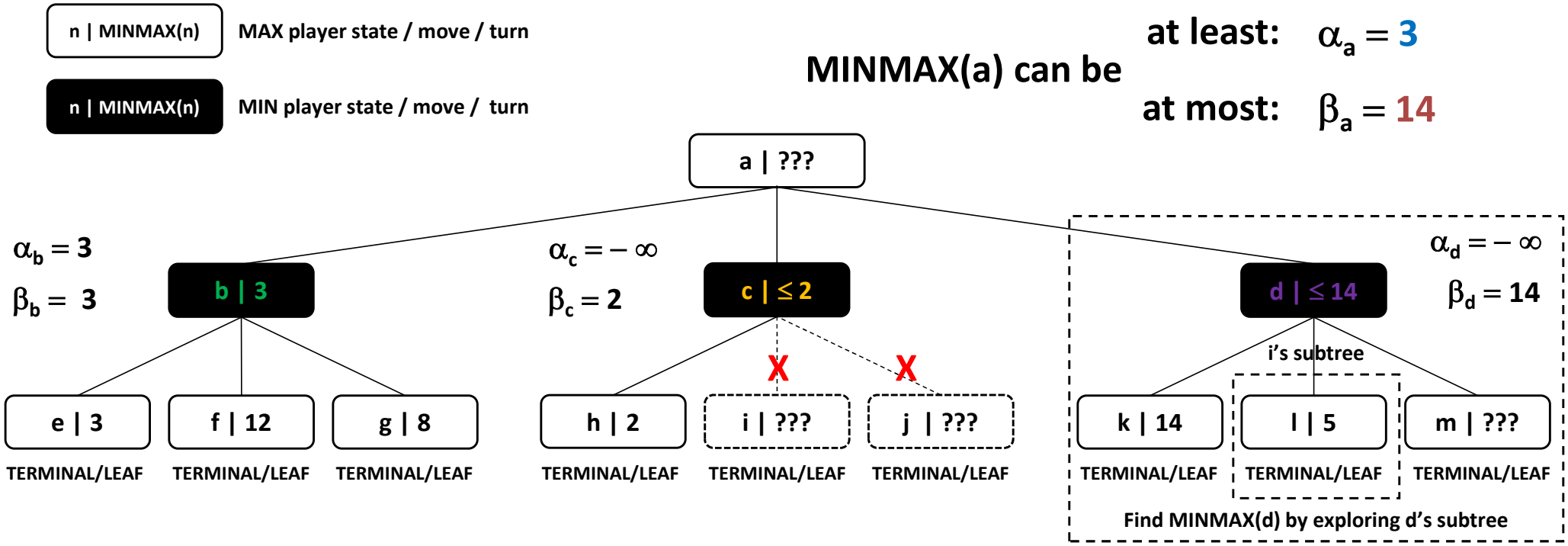
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 14**
- **MAX Player's decision: not enough information yet.**

  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 know that MINMAX(d) ≤ 14 → this tells us that MINMAX(a) cannot be > 14 → $\beta_a = 14$**

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:

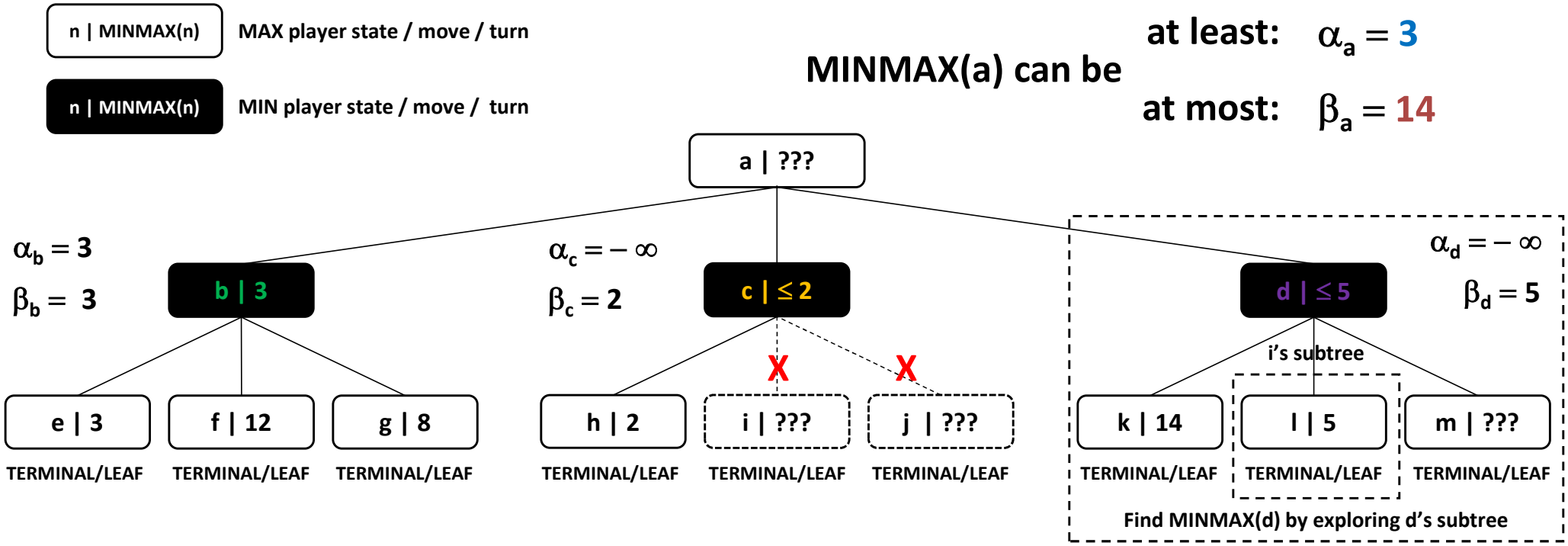
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 14**
- **MAX Player's decision: not enough information yet.**

  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:

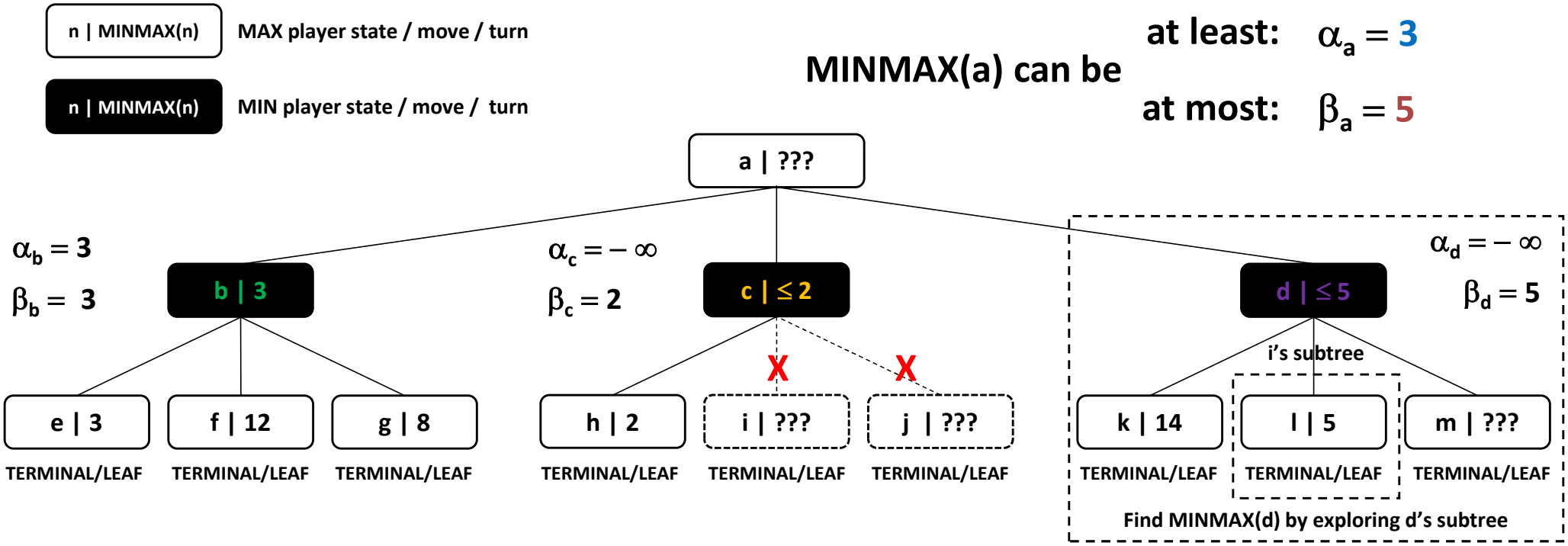
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 14**
- **MAX Player's decision: not enough information yet.**

  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



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:

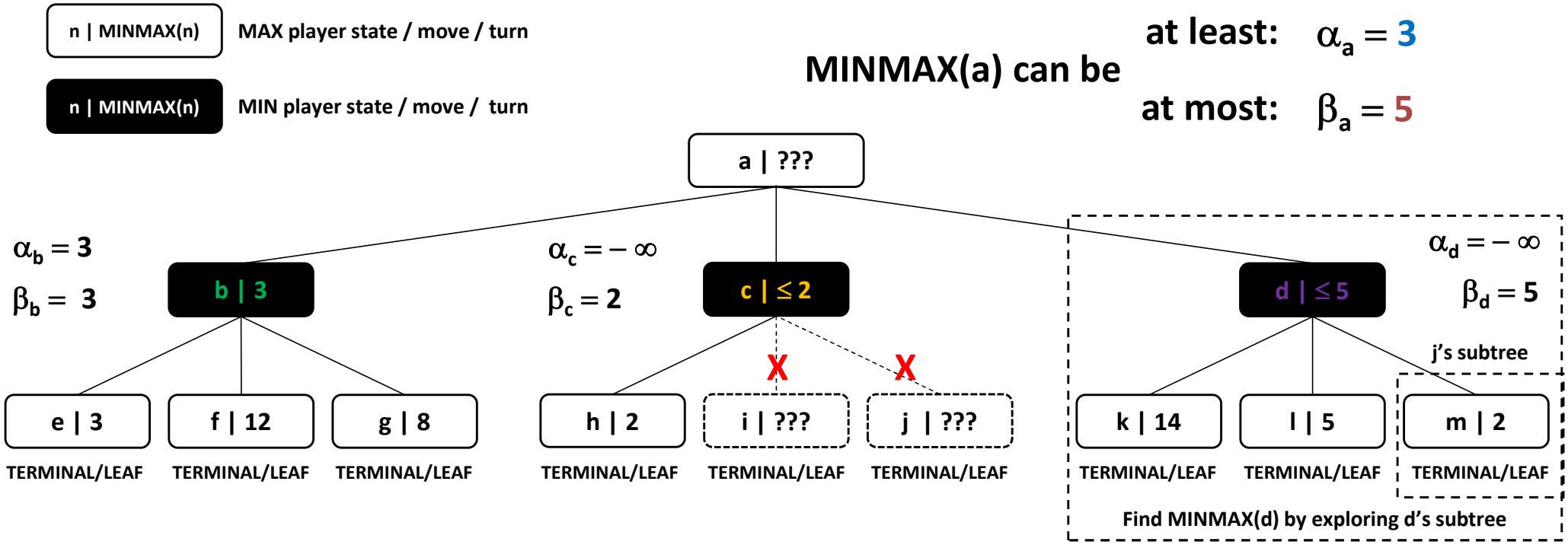
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 5**
- **MAX Player's decision: not enough information yet.**

  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 → $\beta_a = 5$**

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:

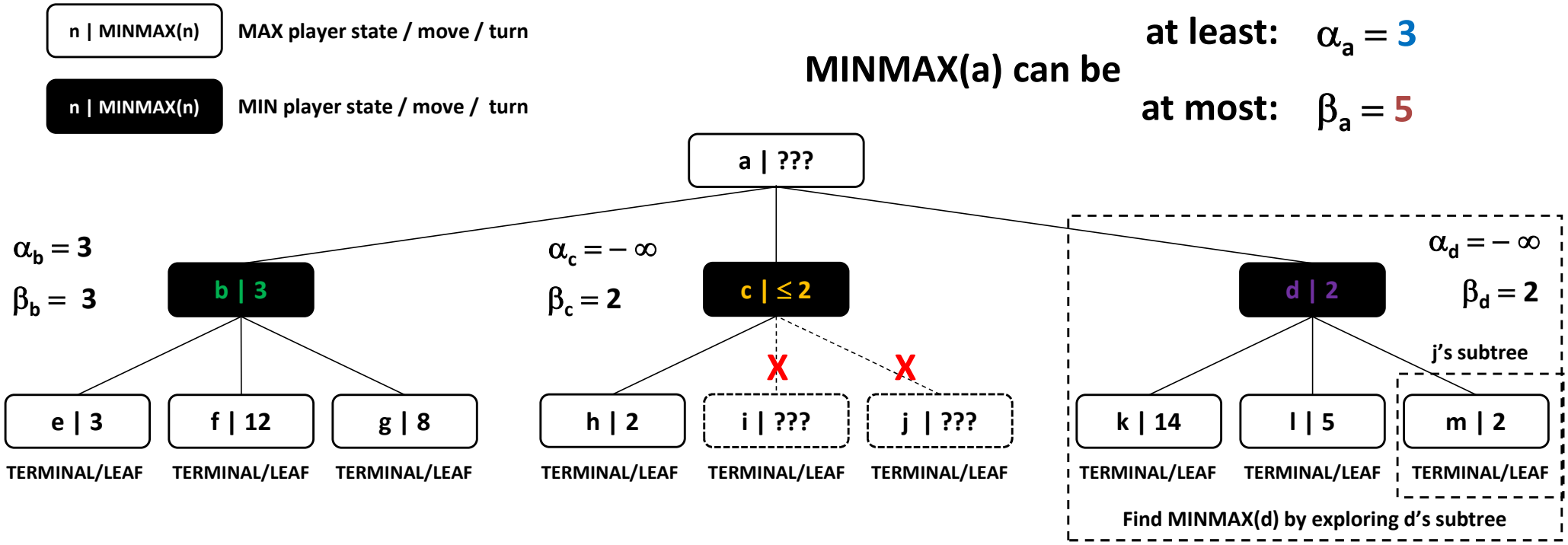
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 5**
- **MAX Player's decision: not enough information yet.**

  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:

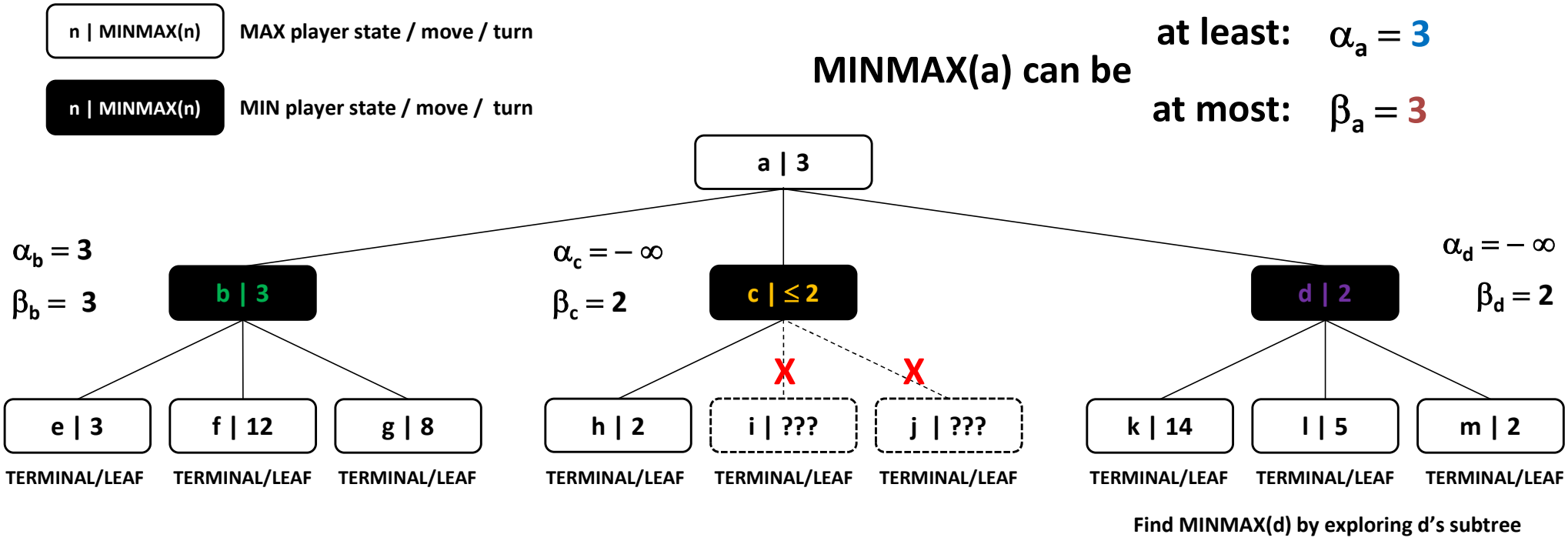
- **MINMAX(b) = 3 | MINMAX(c) = ≤ 2 | MINMAX(d) = ≤ 5**
- **MAX Player's decision: not enough information yet.**

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

MIN Player needs to explore d's subtree:

- **$v2 < v$ $(2 < 5) \rightarrow v = v2 = 2 \rightarrow \beta_d = \min(\beta_d, v) = \min(\infty, 2) = 2$**
- **$v < \alpha_a$ $(2 < 3) \rightarrow$ we cannot keep exploring d's subtree → we also know that MINMAX(d) = 2**

MinMax with α-β: Example

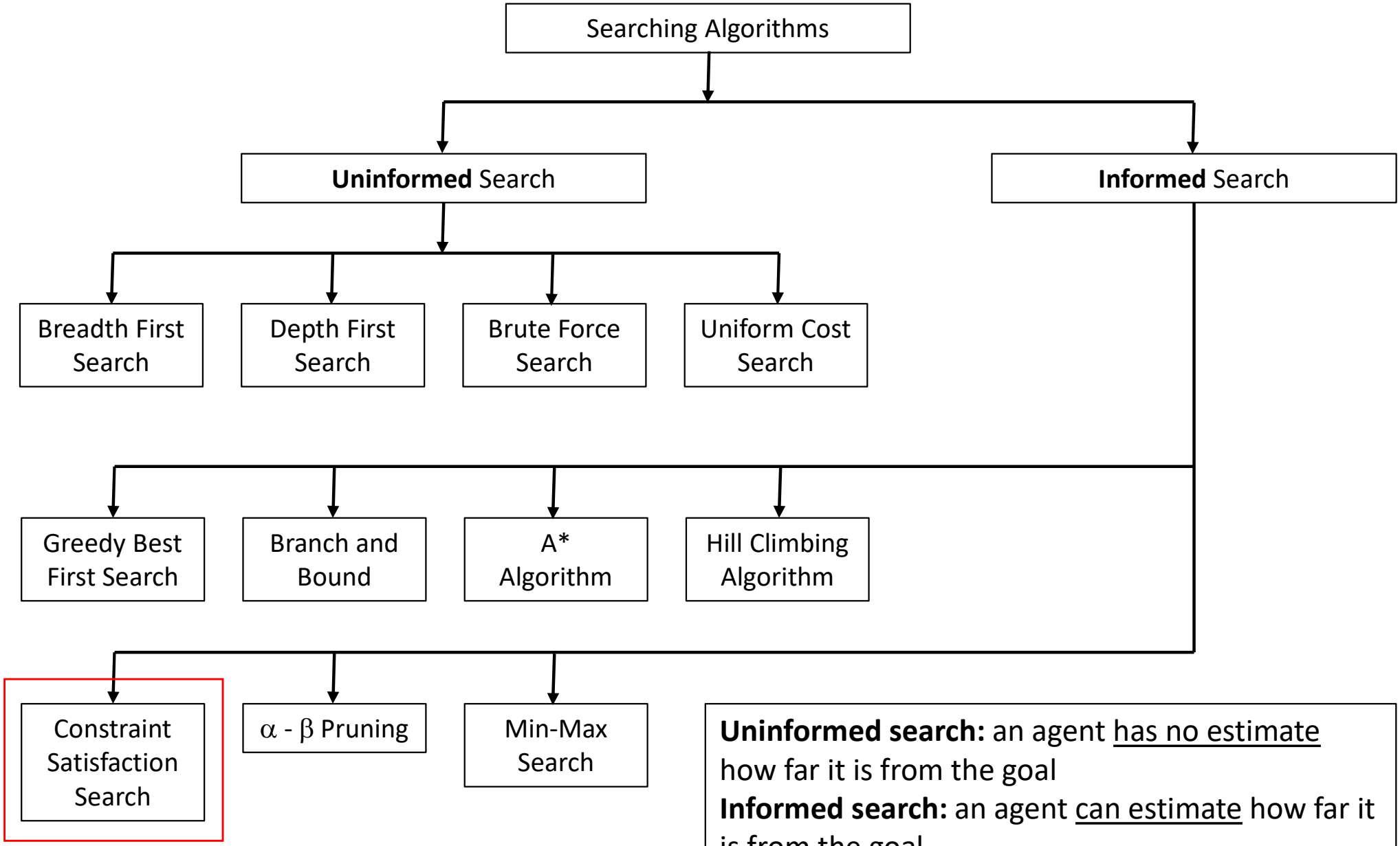


MAX Player wants to maximize the utility of the game by choosing the right move from state a to one of three successor states: b, c, d. It will choose the state with maximum MAXMIN value. Currently:

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

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

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** $\langle \text{scope, relation} \rangle$ **pair, for example** $\langle (X1, X2), X1 > X2 \rangle$

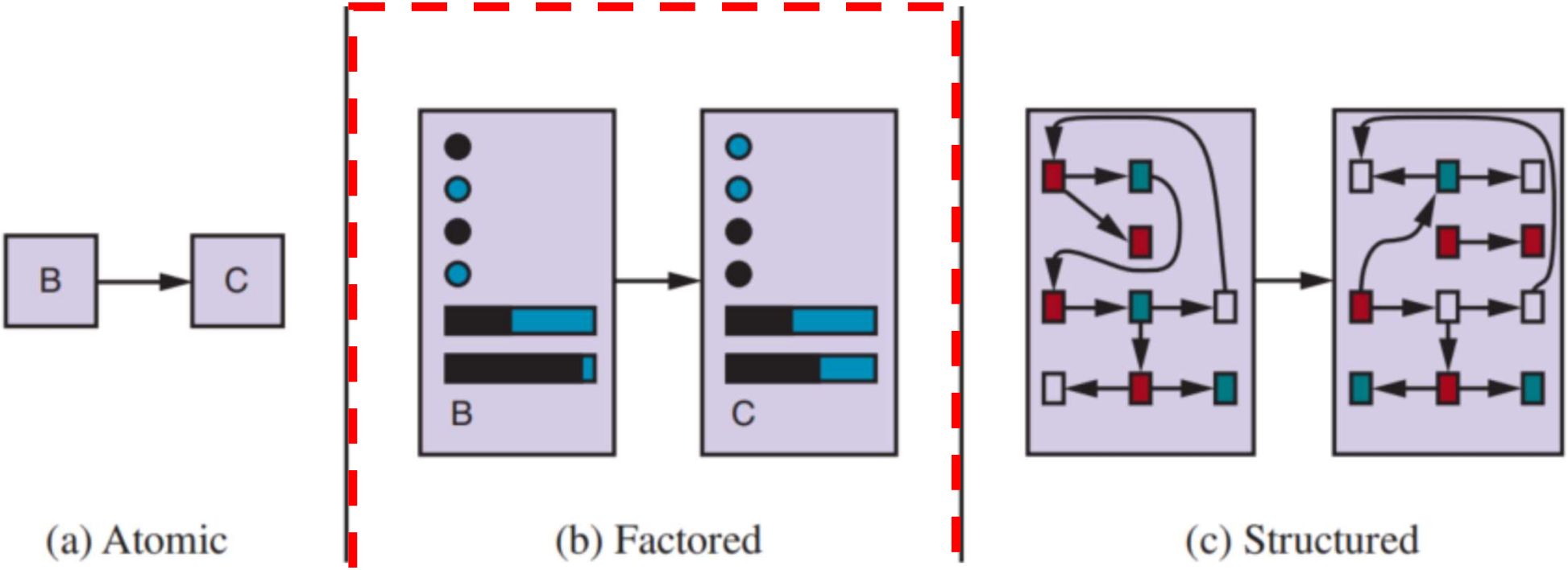
Constraint Satisfaction Problem

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



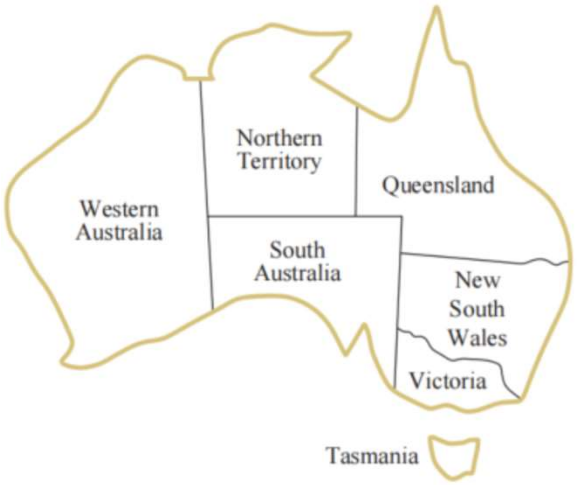
- Searching
- Hidden Markov models
- Markov decision process
- Finite state machines

- 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:

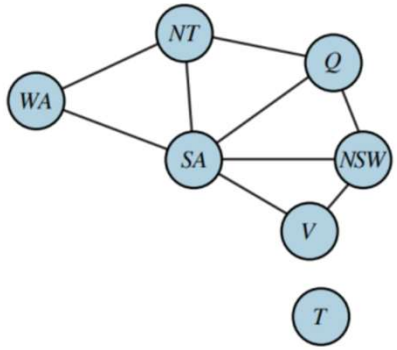
$D_{WA} = \{RED, GREEN, BLUE\}$
$D_{NT} = \{RED, GREEN, BLUE\}$
$D_{Q} = \{RED, GREEN, BLUE\}$
$D_{NSW} = \{RED, GREEN, BLUE\}$
$D_{V} = \{RED, GREEN, BLUE\}$
$D_{SA} = \{RED, GREEN, BLUE\}$
$D_{T} = \{RED, GREEN, BLUE\}$

Constraints (Rules):

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

$CONSTRAINTS = C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$

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:

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

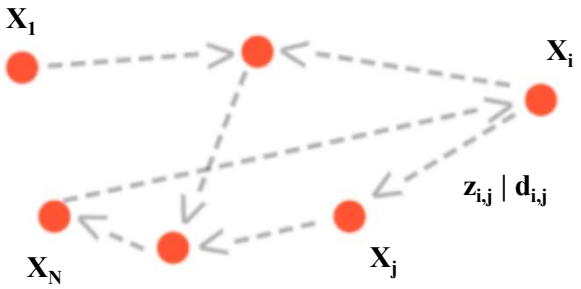
Constraints (Rules):

- **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:

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

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: Variable Types

- **Domains can be:**
  - **finite, for example:** $\{1, 2, 3, 5, 8, 20\}$ **(simpler)**
  - **infinite, for example: a set of all integers**
- **Variables can be:**
  - **discrete, for example:** $X = \{X_1, ..., X_n\}$ **(simpler)**
  - **continuous, for example:** R_+
- **Constraints can be:**
  - **unary (involve single variable), for example:** $X_1 = 5$
  - **binary (involve two variables), for example:** $X_1 = X_2$
  - **higher order (involve > 2 variables), for example:** $X_1 = X_2 * X_3$
- **Soft constraints (preferences: green over blue) possible**

CSP as a Search Problem

CSP is a variant of a search problem you already know. The problem can be restated / updated with:

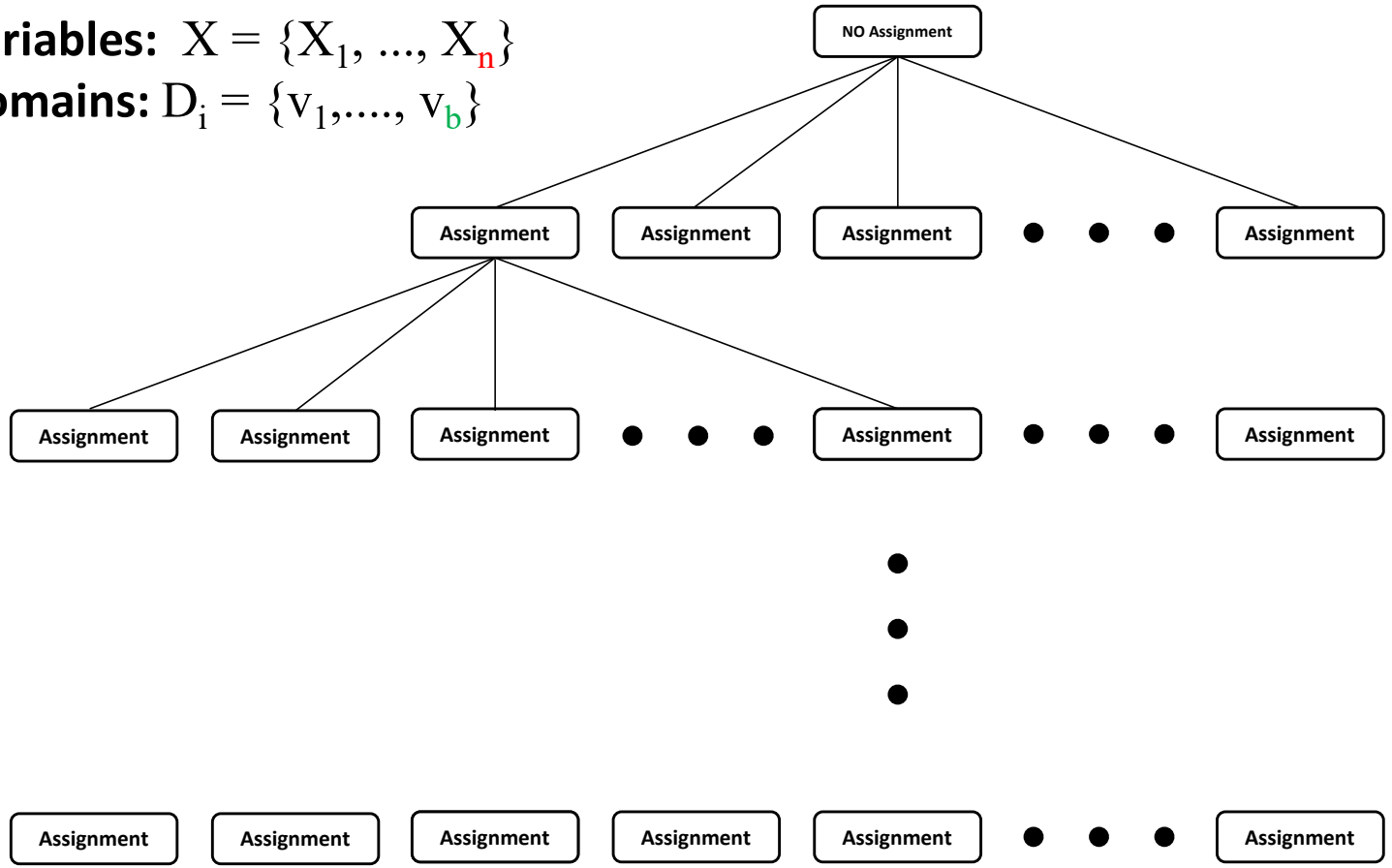
- **Initial state: the empty assignment { }, in which all variables are unassigned.**
- **Successor function: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.**
- **Goal test: the current assignment is complete.**
- **Path cost: a constant cost (e.g., 1) for every step.**

CSP Search Tree: Idea

CSP Problem:

Variables: $X = \{X_1, ..., X_n\}$

Domains: $D_i = \{v_1, ..., v_b\}$



0 variable assigned

1 variables assigned

2 variables assigned

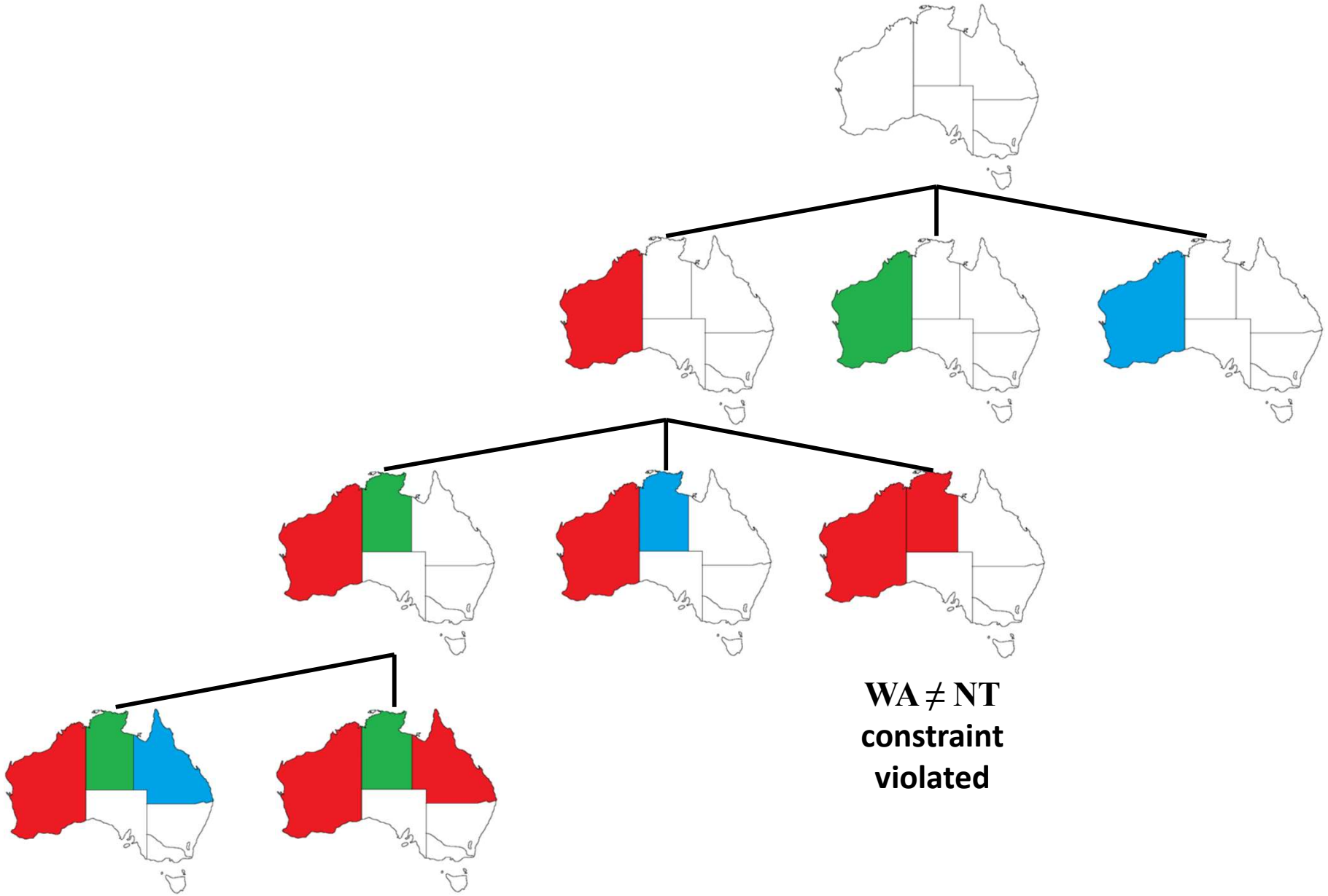
ALL (n) variables assigned

Tree leaves are COMPLETE assignments

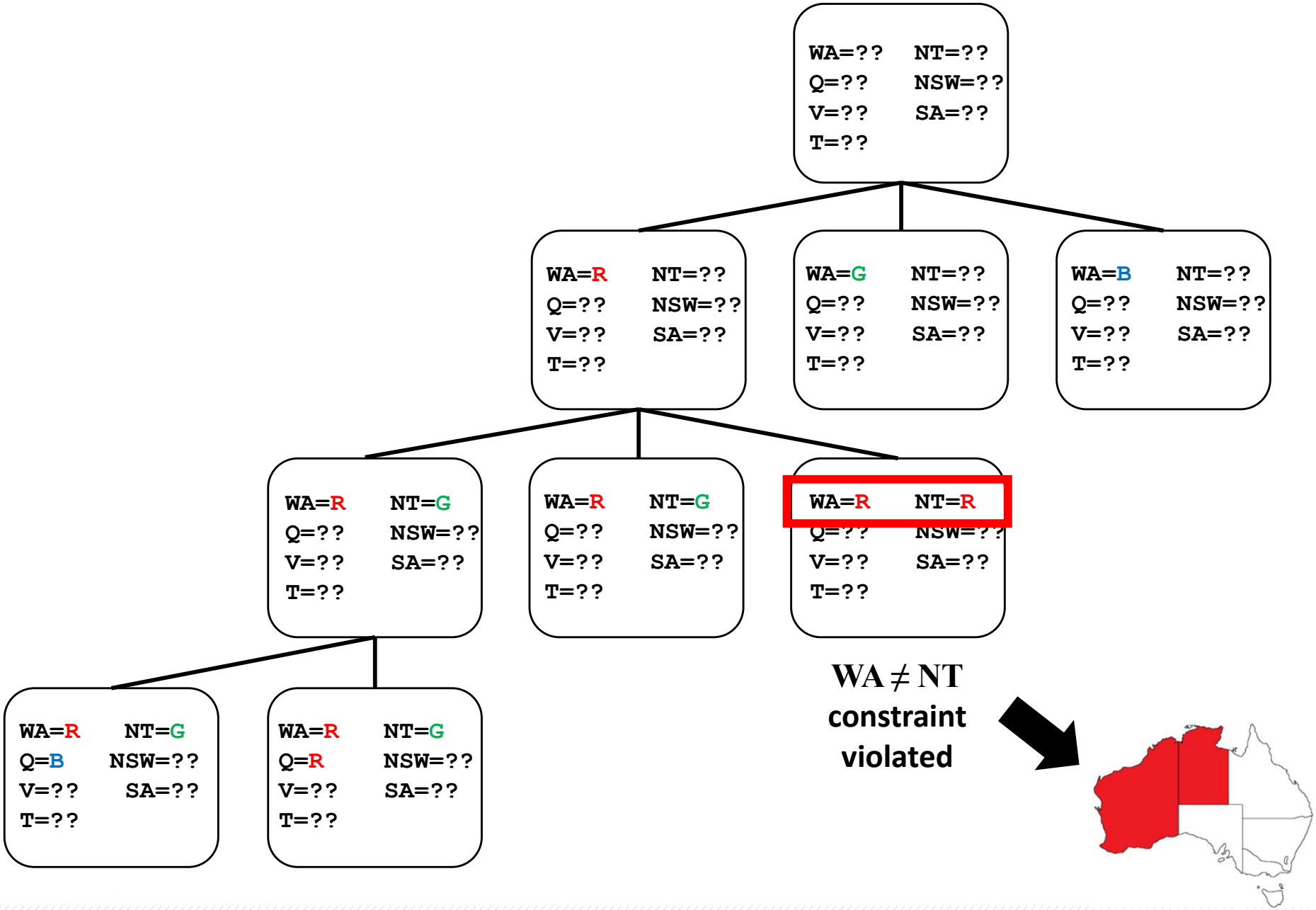
The sequence of variable assignments does NOT matter*

***(when you disregard performance)**

CSP as a Tree Search Problem



CSP as a Tree Search Problem

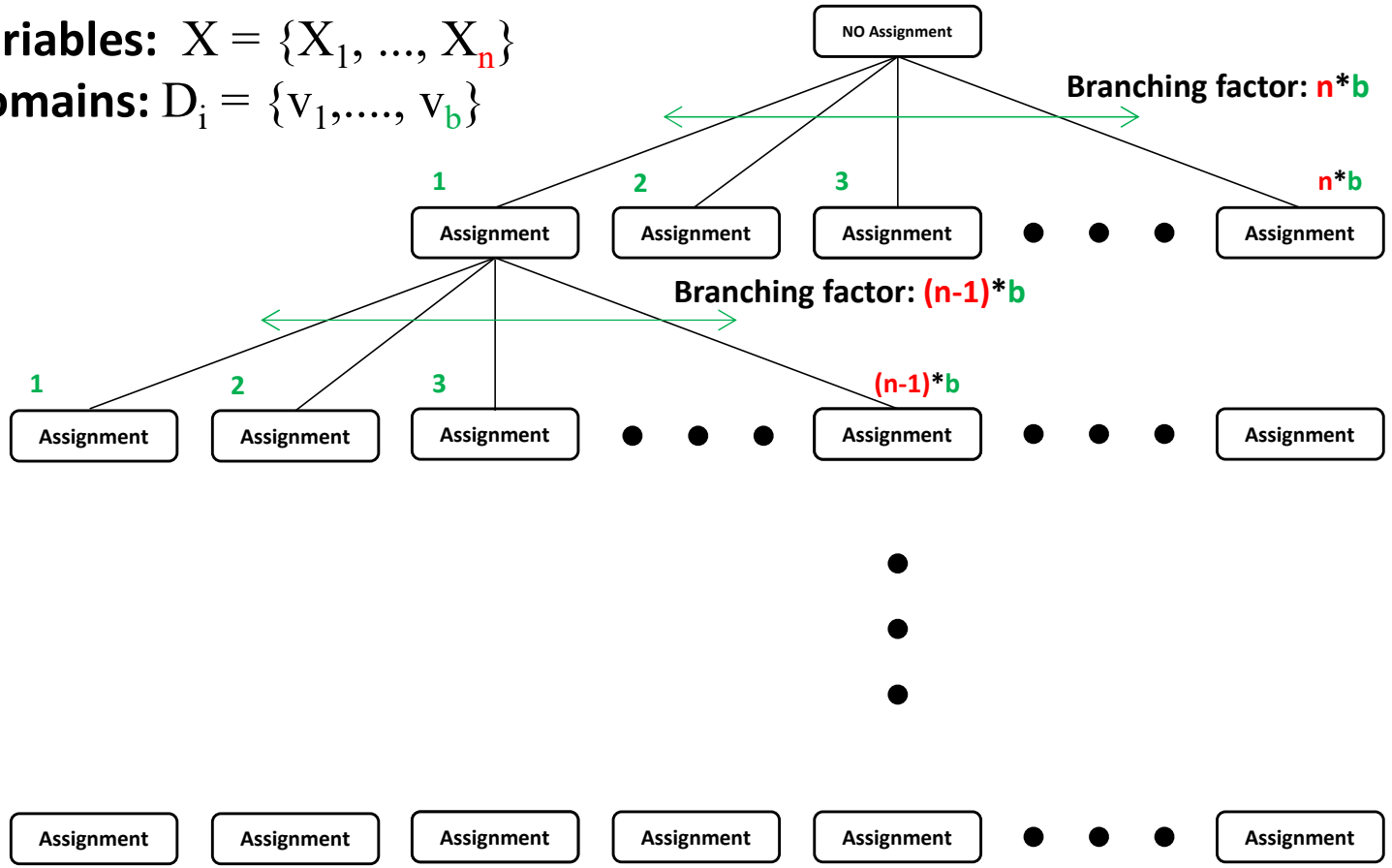


CSP Search Tree: Size

CSP Problem:

Variables: $X = \{X_1, ..., X_n\}$

Domains: $D_i = \{v_1, ..., v_b\}$



$N_0 = 0$

$N_1 = n*b$

$N_2 = n*b*(n-1)*b = n*(n-1)*b^2$

$N_n = n! * b^n$

Total number of leafnodes / states: $n! * b^n$

(ignores COMMUTATIVITY of CSP assignments:

assigning $X_1 = m$ and then $X_2 = n$ SAME as assigning $X_2 = n$ and then $X_1 = m$)

In reality: there is only b^n complete assignments

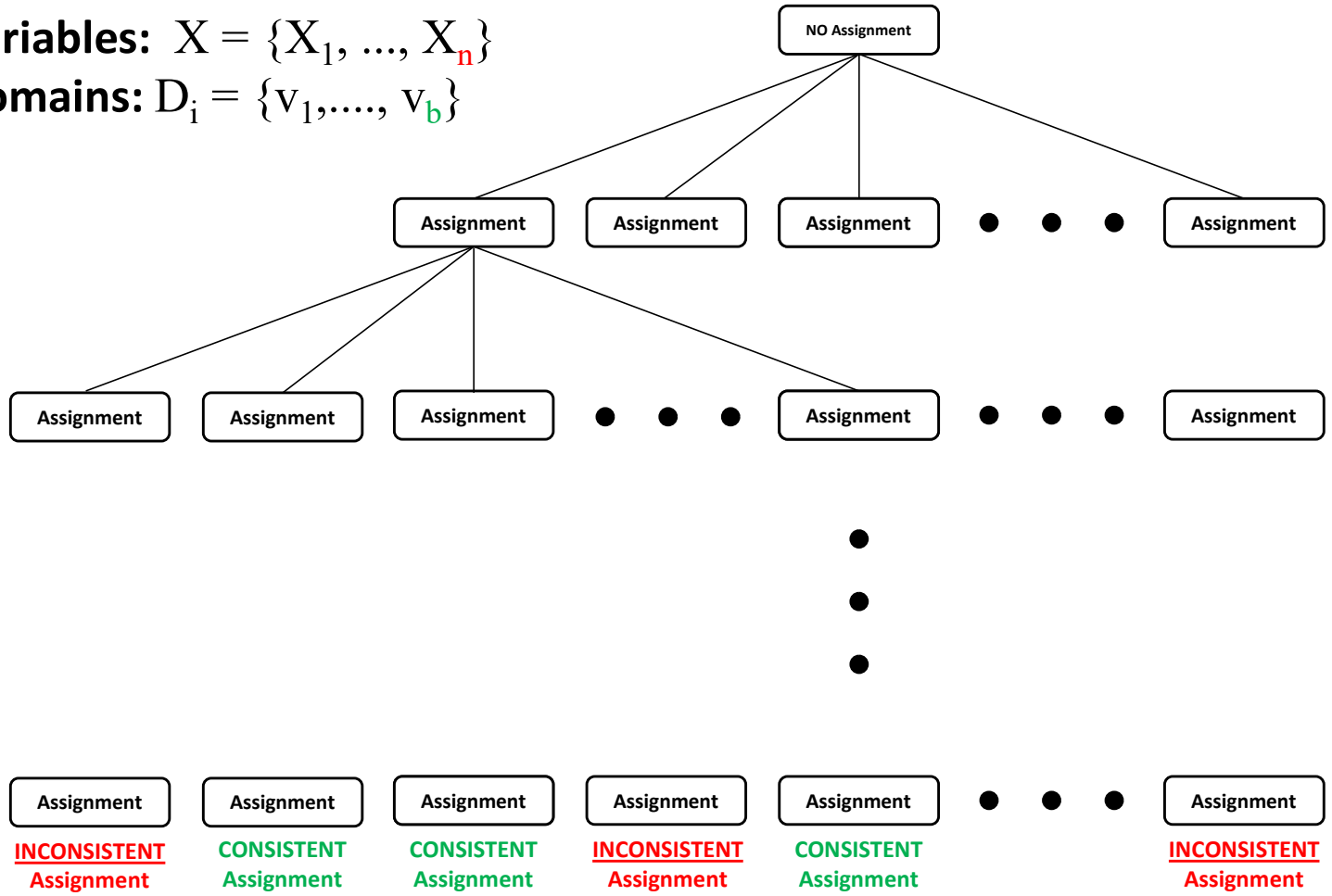
Can We Do Better?

CSP Search Tree: Solutions

CSP Problem:

Variables: $X = \{X_1, ..., X_n\}$

Domains: $D_i = \{v_1, ..., v_b\}$

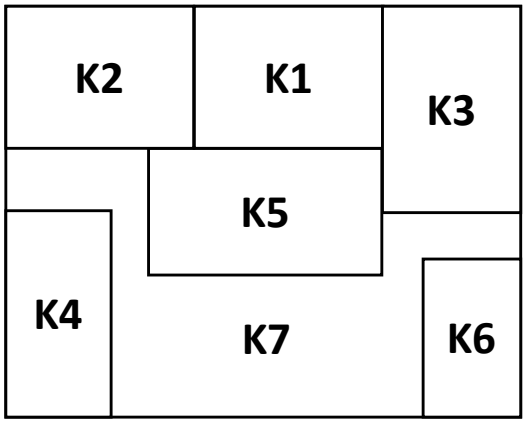


Some nodes / states will be CONSISTENT, while others will be INCONSISTENT.

Depth first search could possibly visit them all → WASTEFUL.

CSP Example: Map Coloring

Problem:



Variables:

X = {K1, K2, K3, K4, K5, K6, K7}

Variable Domains:

D_{K1} = {RED, BLUE, GREEN}
D_{K2} = {RED, BLUE, GREEN}
D_{K3} = {RED, BLUE, GREEN}
D_{K4} = {RED, BLUE, GREEN}
D_{K5} = {RED, BLUE, GREEN}
D_{K6} = {RED, BLUE, GREEN}
D_{K7} = {RED, BLUE, GREEN}

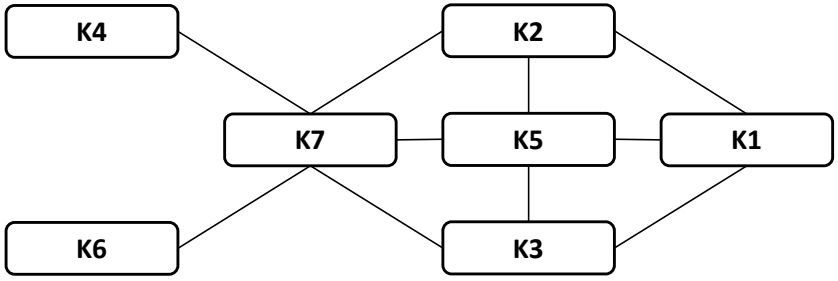
Color this map in a way that no two neighbors have same color

Constraints (Rules):

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

CONSTRAINTS = C = {K1 ≠ K2, K1 ≠ K3, K1 ≠ K5, K2 ≠ K5, K2 ≠ K7, K3 ≠ K5, K3 ≠ K7, K4 ≠ K7, K5 ≠ K7, K6 ≠ K7}

Constraint Graph:

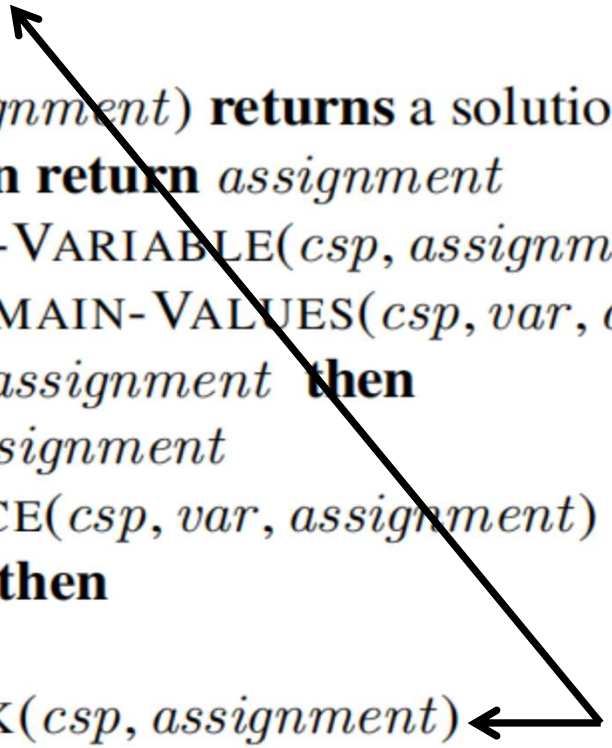


CSP Backtracking: Pseudocode

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

CSP Backtracking: Pseudocode

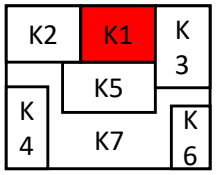
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```



RECURSION

Assignment:
K1: RED
K2: ???
K3: ???
K4: ???
K5: ???
K6: ???
K7: ???

Initial (NO assignment) state not shown



Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: RED

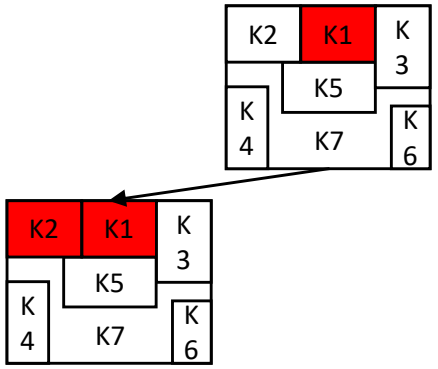
K3: ???

K4: ???

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: RED

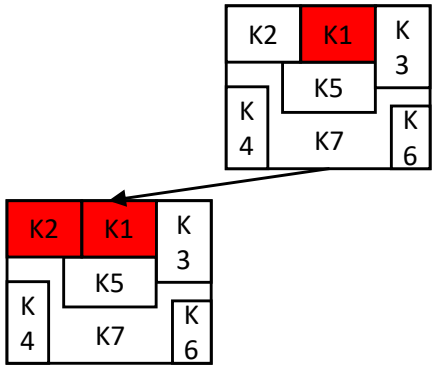
K3: ???

K4: ???

K5: ???

K6: ???

K7: ???



Violates: Rule 1

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: ???

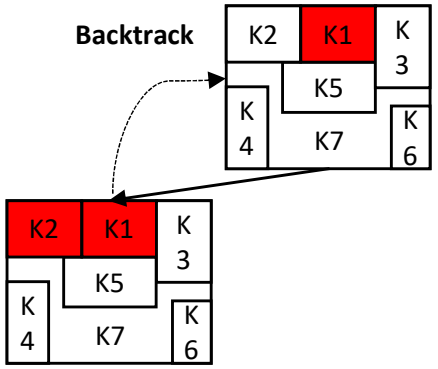
K3: ???

K4: ???

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

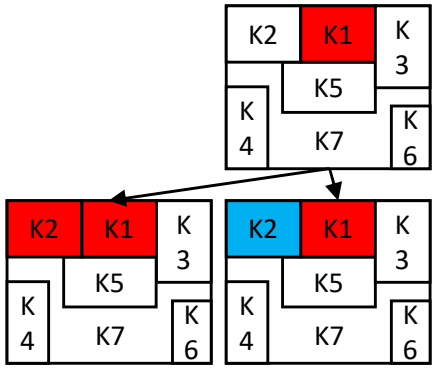
K3: ???

K4: ???

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

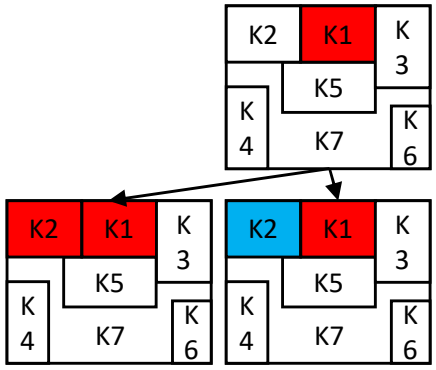
K3: ???

K4: ???

K5: ???

K6: ???

K7: ???



CONSISTENT PARTIAL Assignment

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

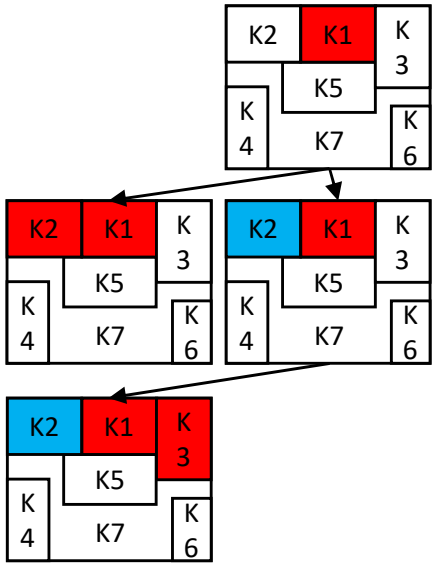
K3: RED

K4: ???

K5: ???

K6: ???

K7: ???



Violates:
Rule 2

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

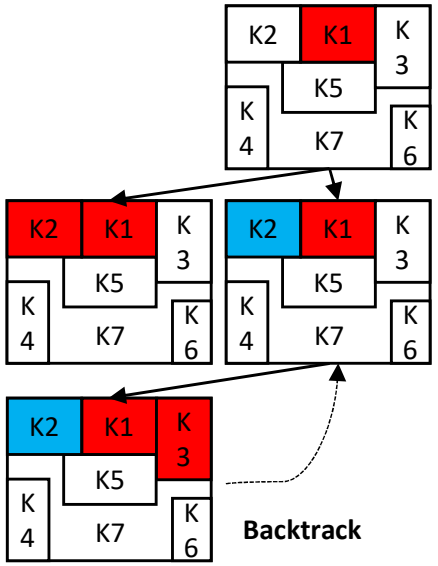
K3: ???

K4: ???

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

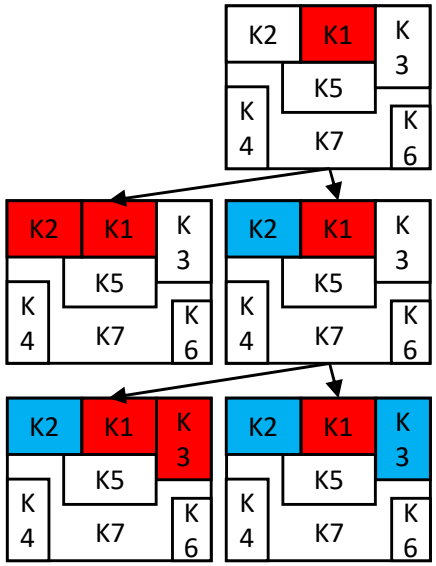
K3: BLUE

K4: ???

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

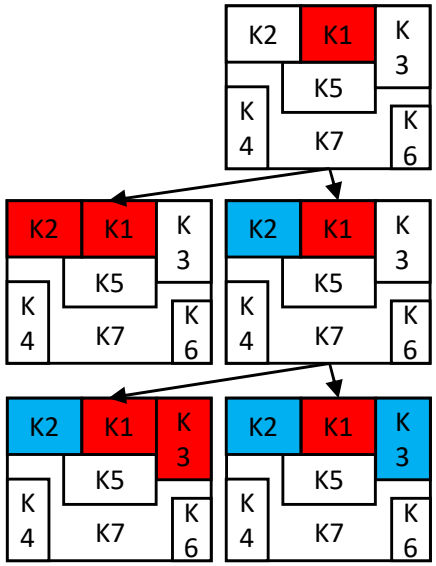
K3: BLUE

K4: ???

K5: ???

K6: ???

K7: ???



CONSISTENT PARTIAL Assignment

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

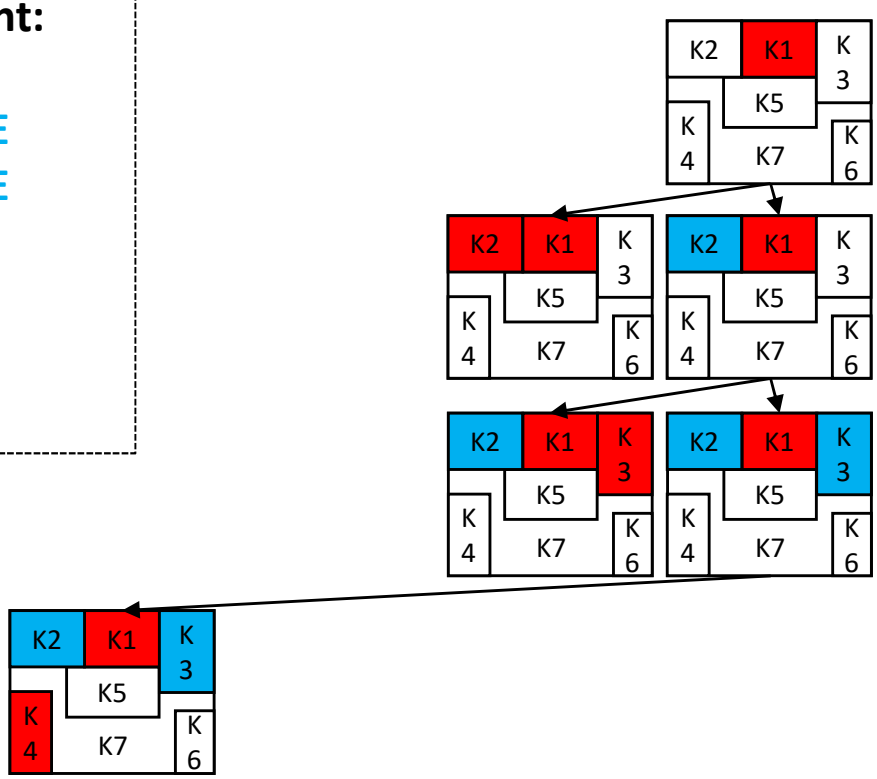
K3: BLUE

K4: RED

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

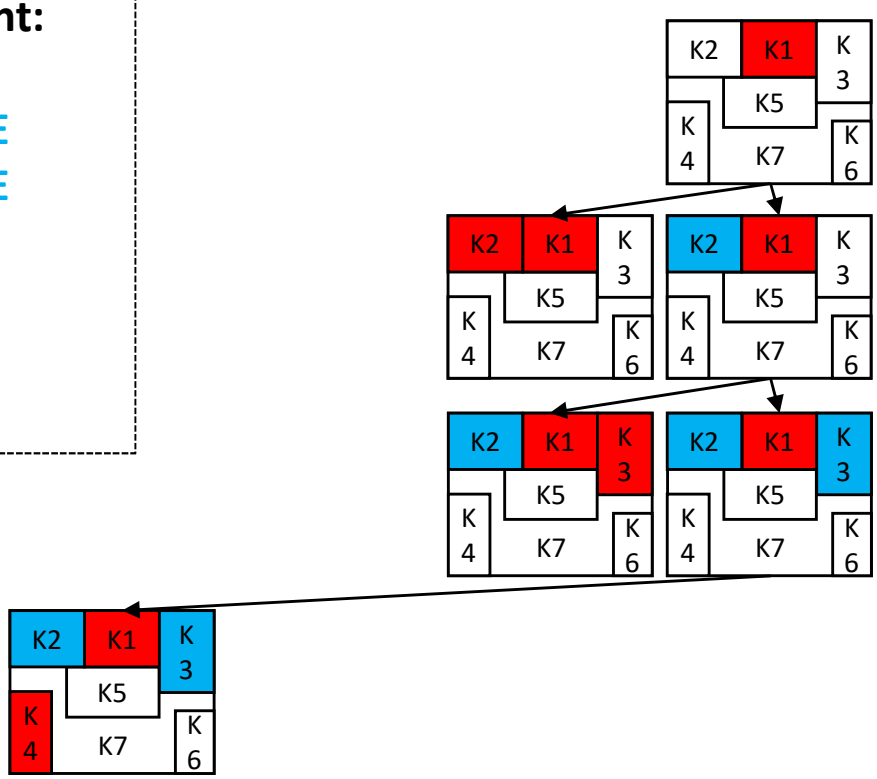
K3: BLUE

K4: RED

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

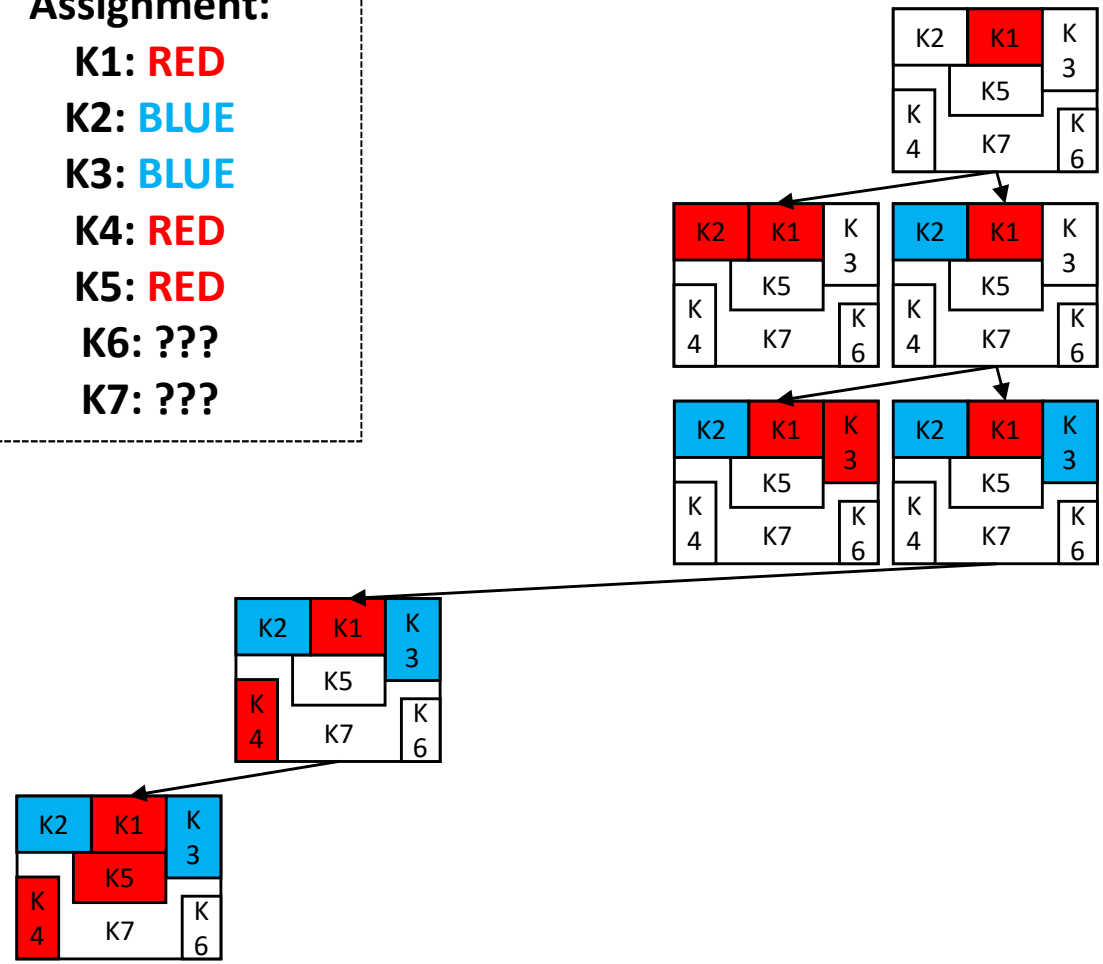
K3: BLUE

K4: RED

K5: RED

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

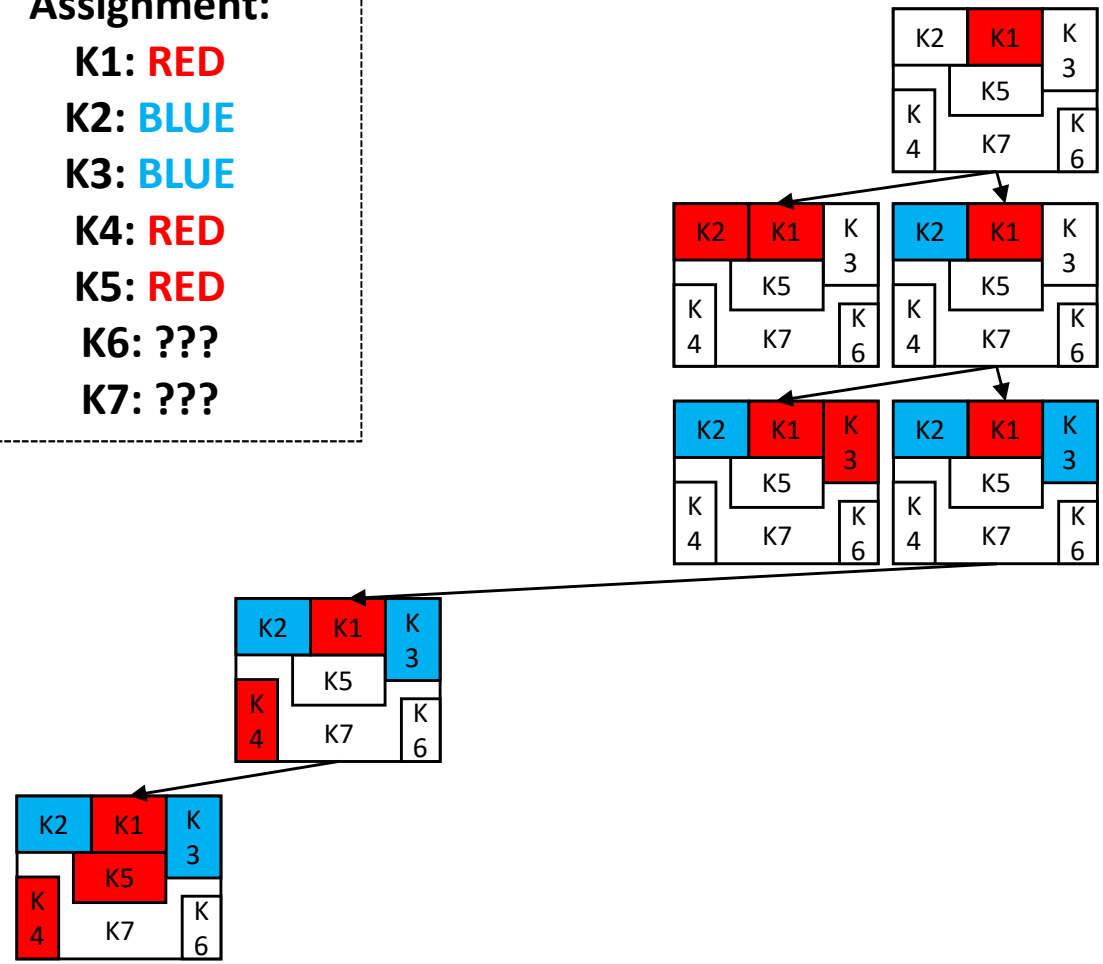
K3: BLUE

K4: RED

K5: RED

K6: ???

K7: ???



Violates:
Rule 3

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

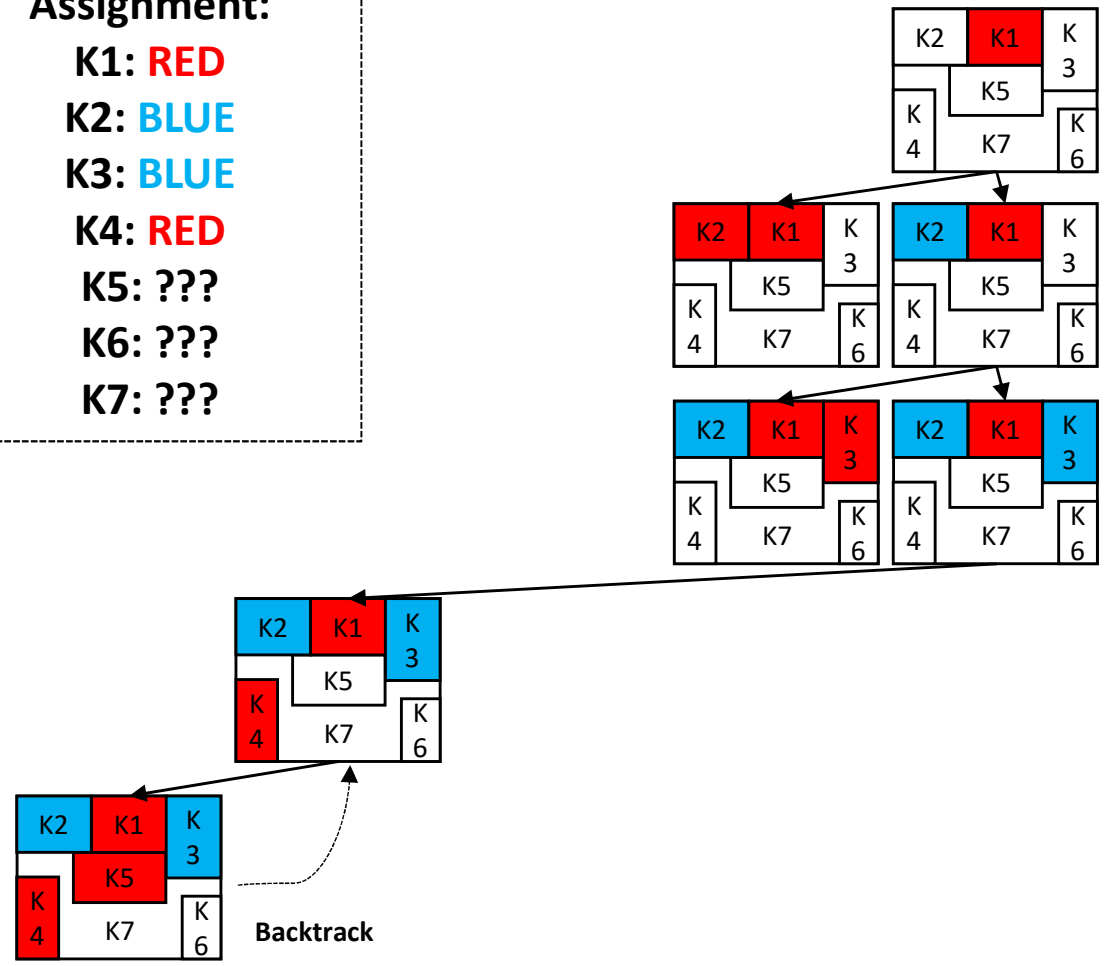
K3: BLUE

K4: RED

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

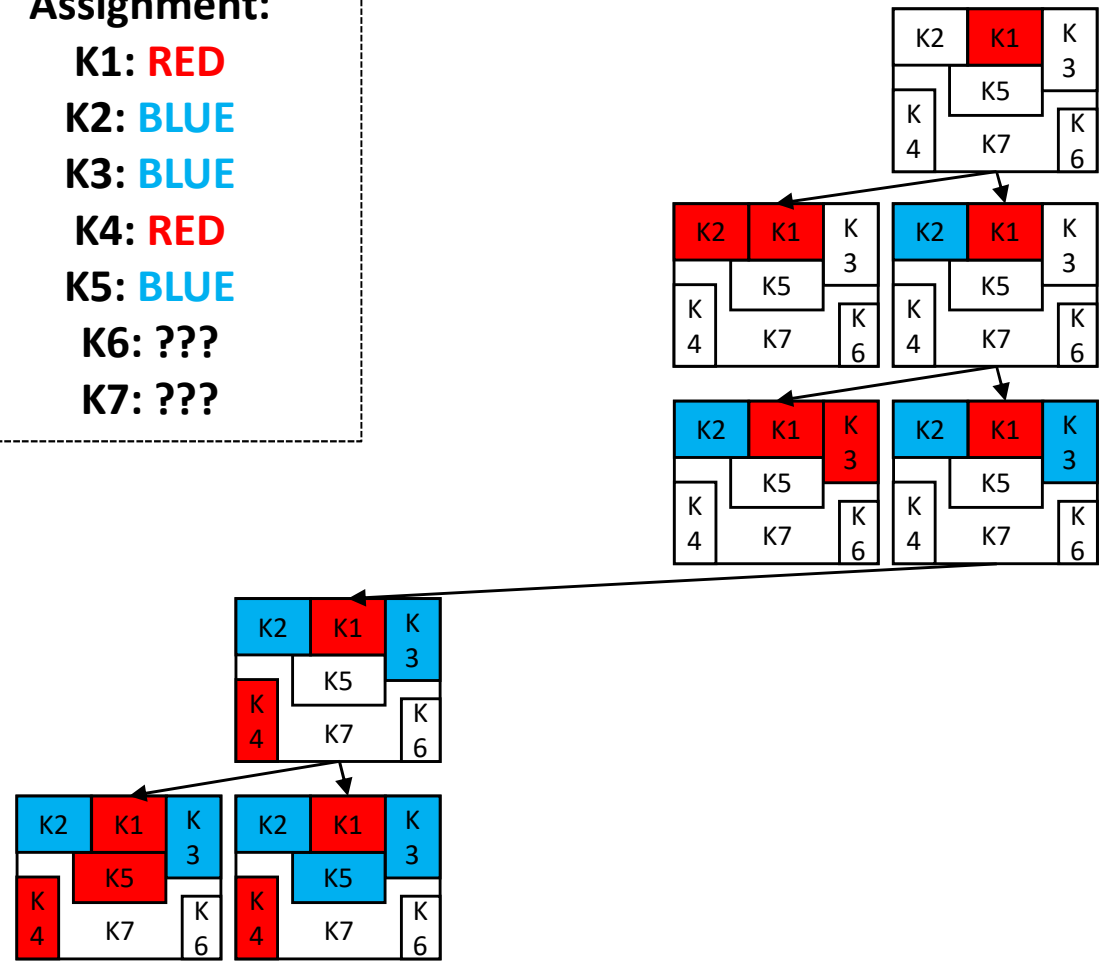
K3: BLUE

K4: RED

K5: BLUE

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

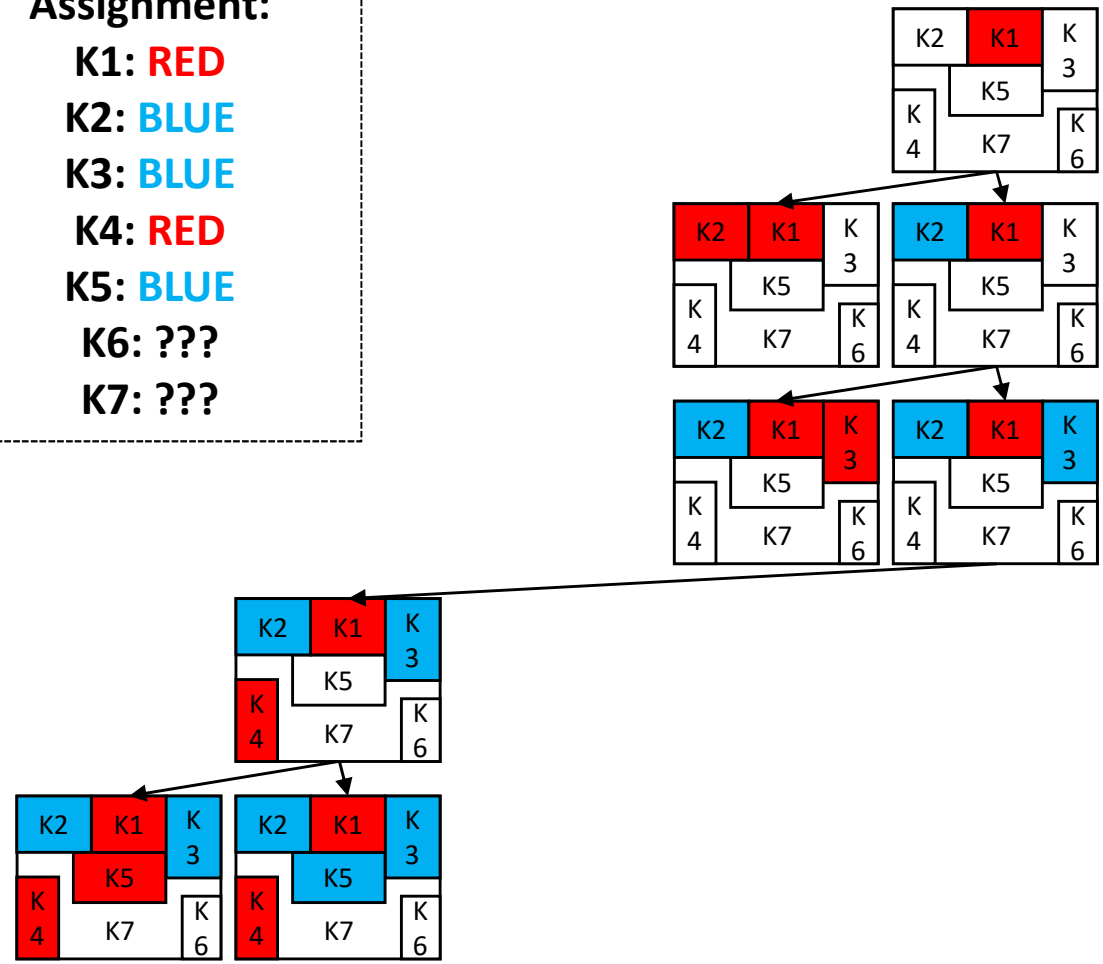
K3: BLUE

K4: RED

K5: BLUE

K6: ???

K7: ???



Violates:
Rule 4
Rule 6

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

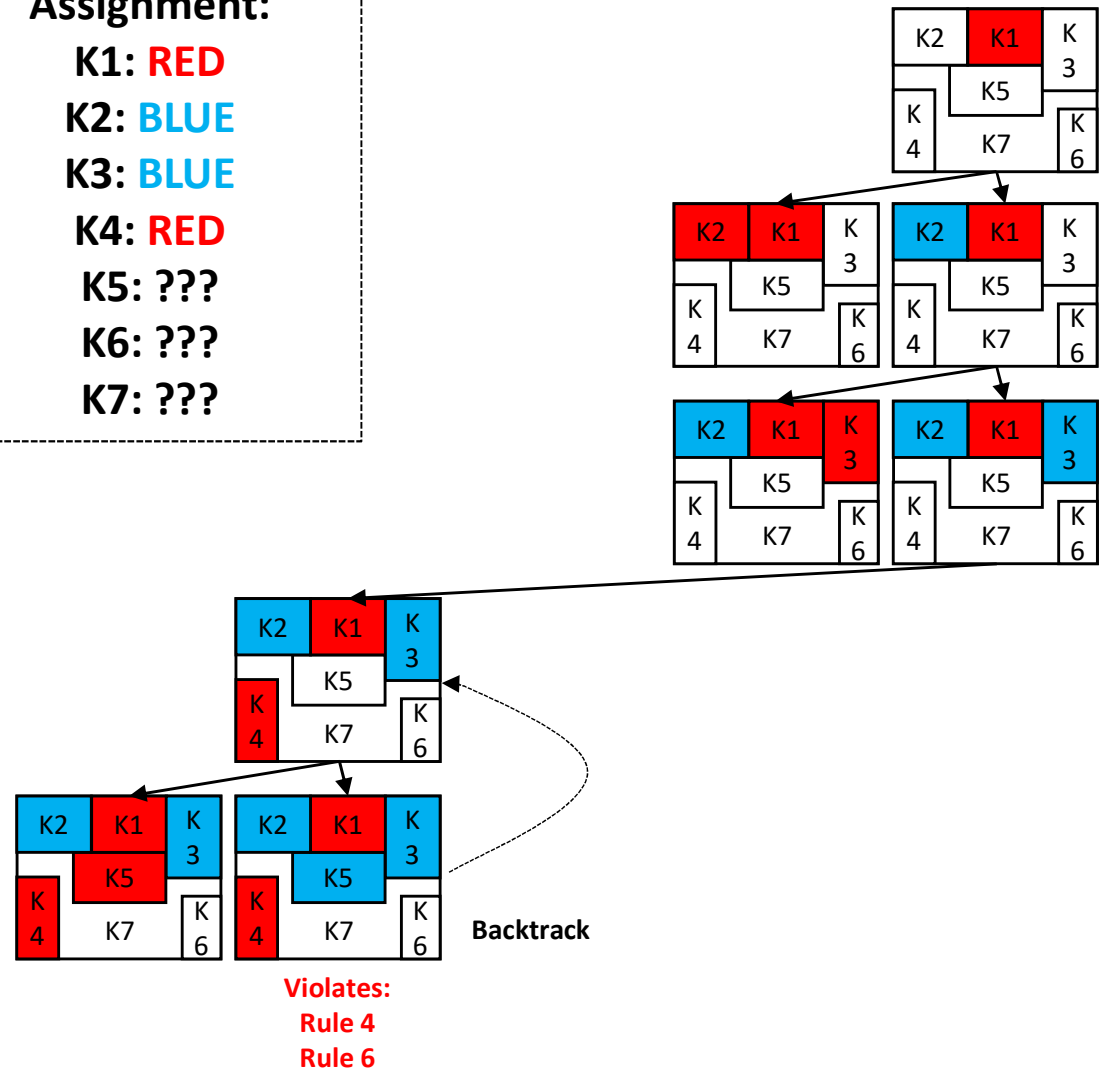
K3: BLUE

K4: RED

K5: ???

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

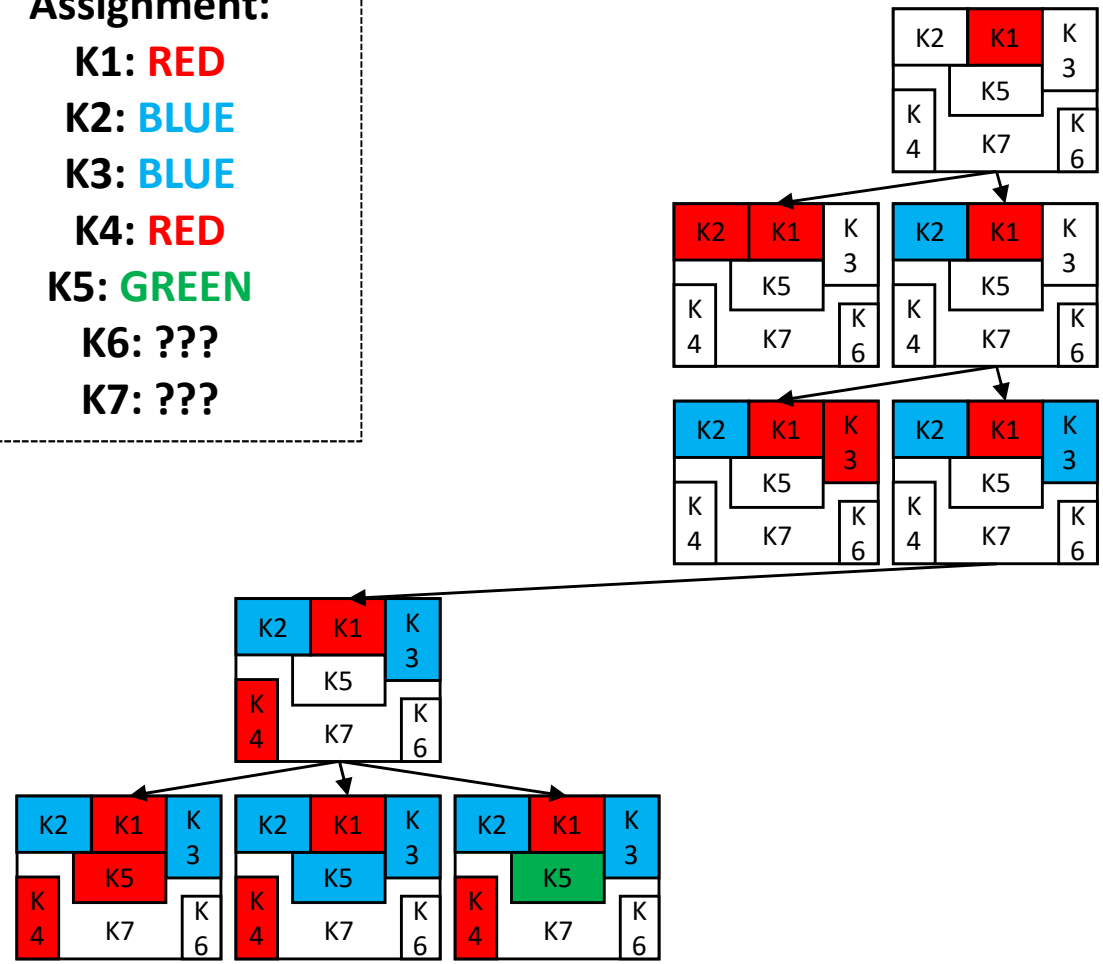
K3: BLUE

K4: RED

K5: GREEN

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

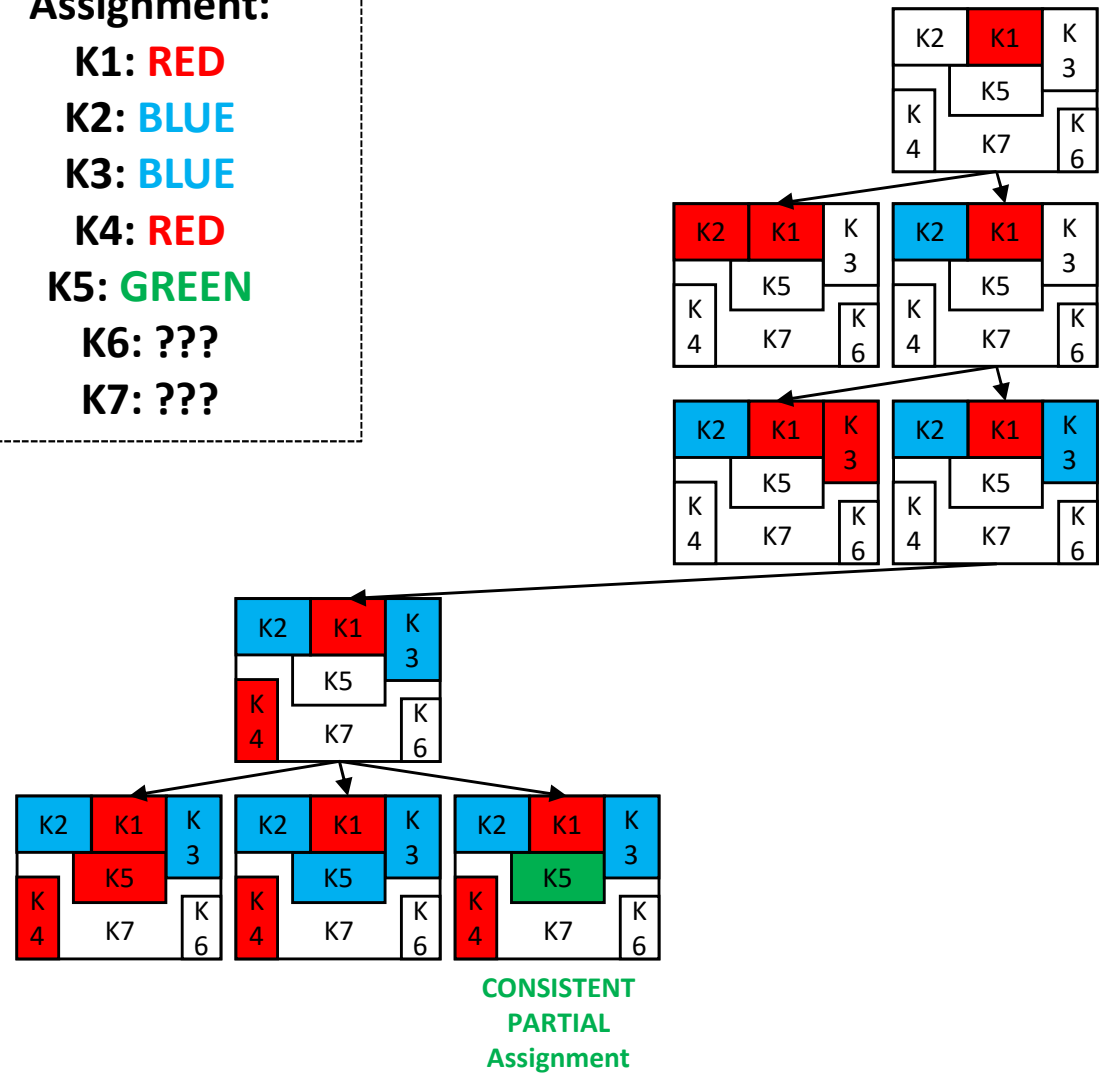
K3: BLUE

K4: RED

K5: GREEN

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

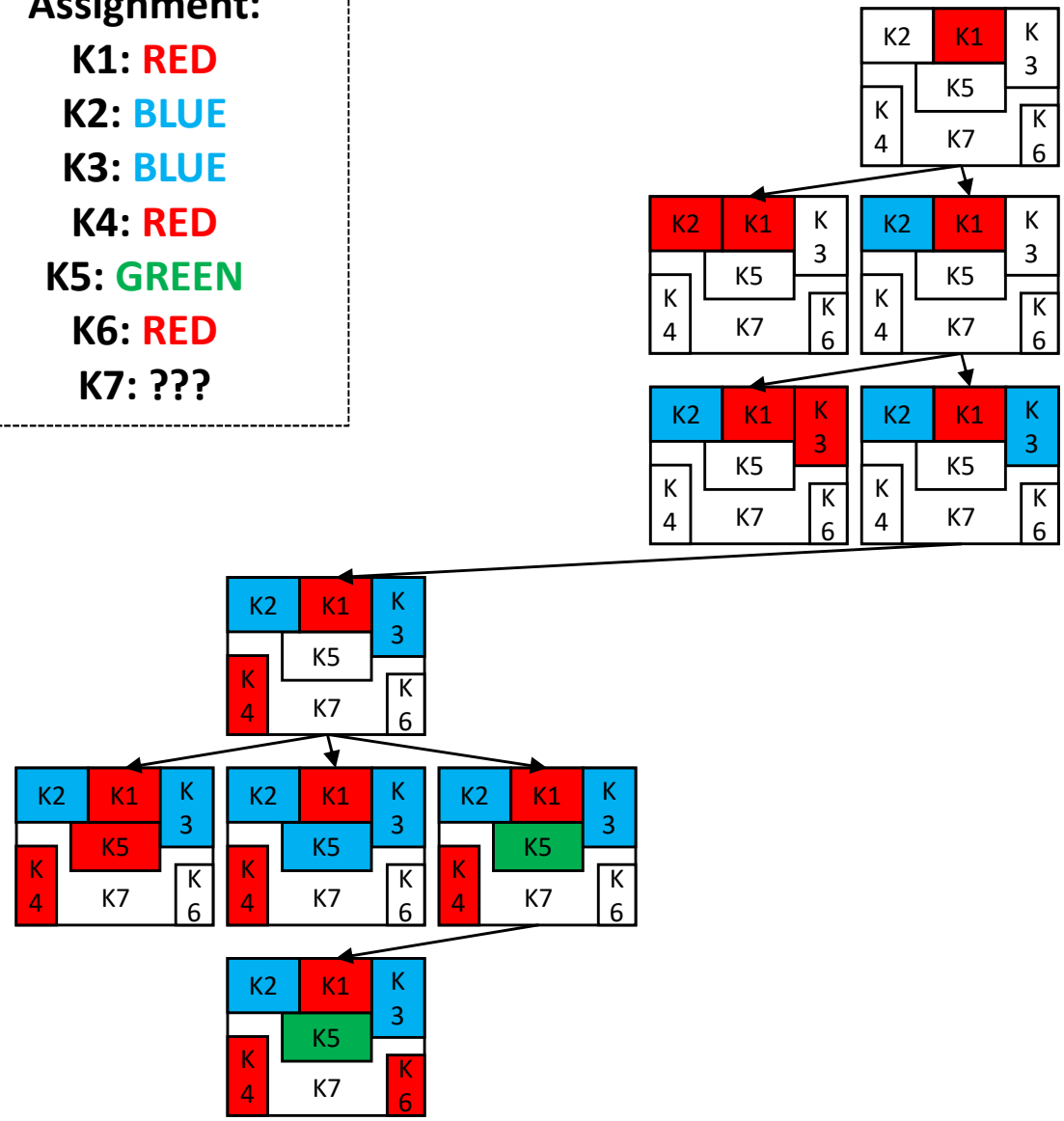
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

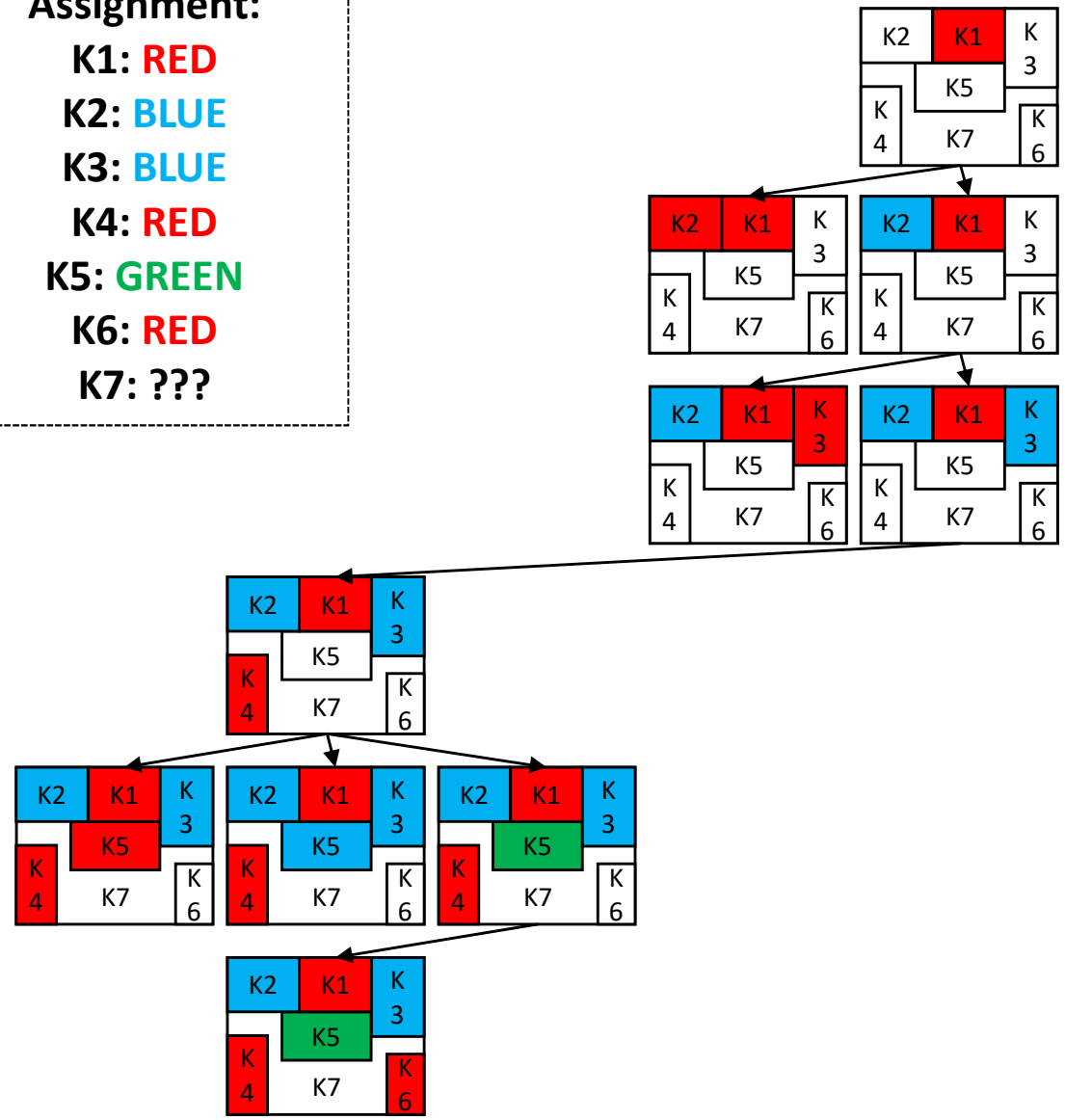
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: ???



CONSISTENT PARTIAL Assignment

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED

K2: BLUE

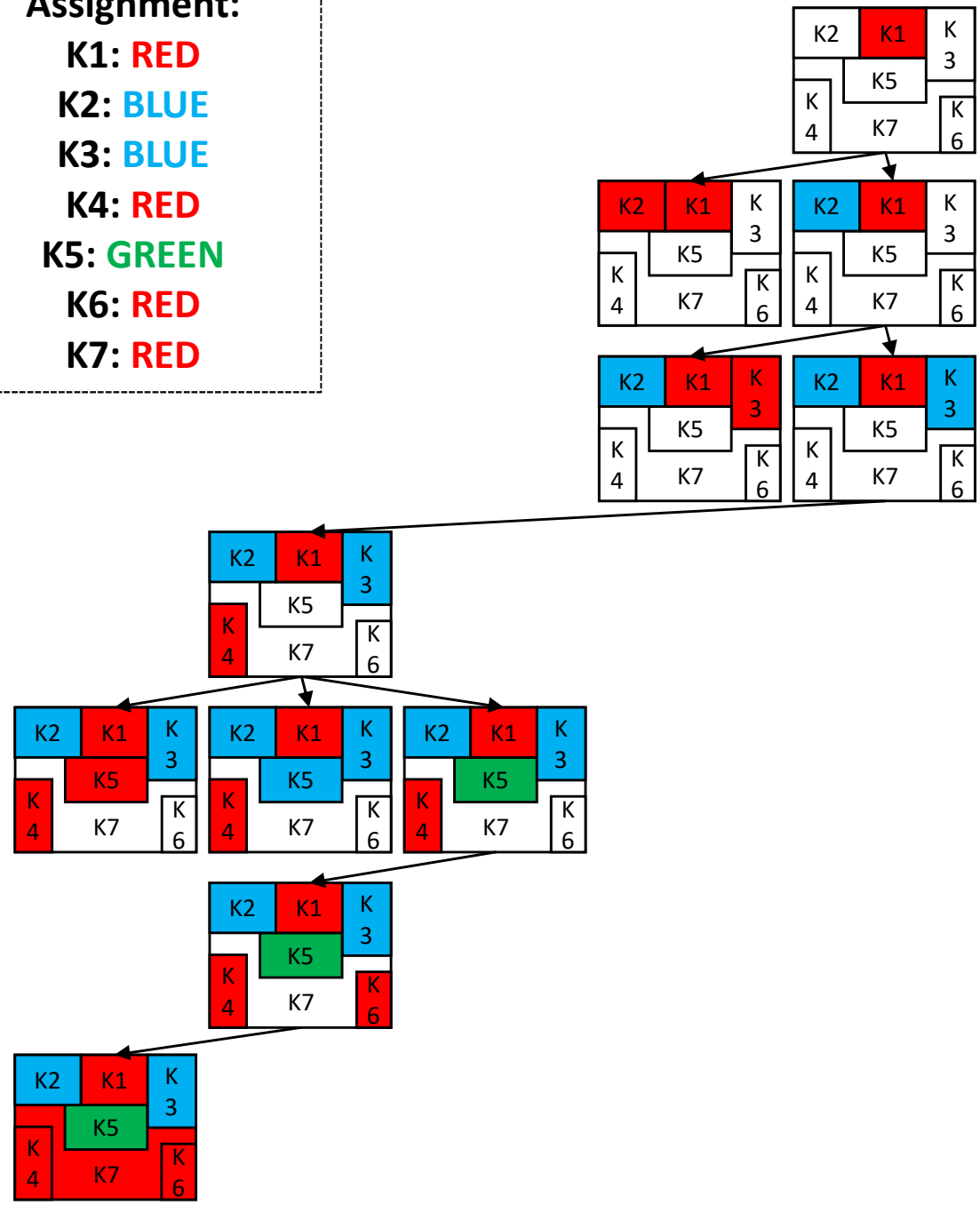
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: RED



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

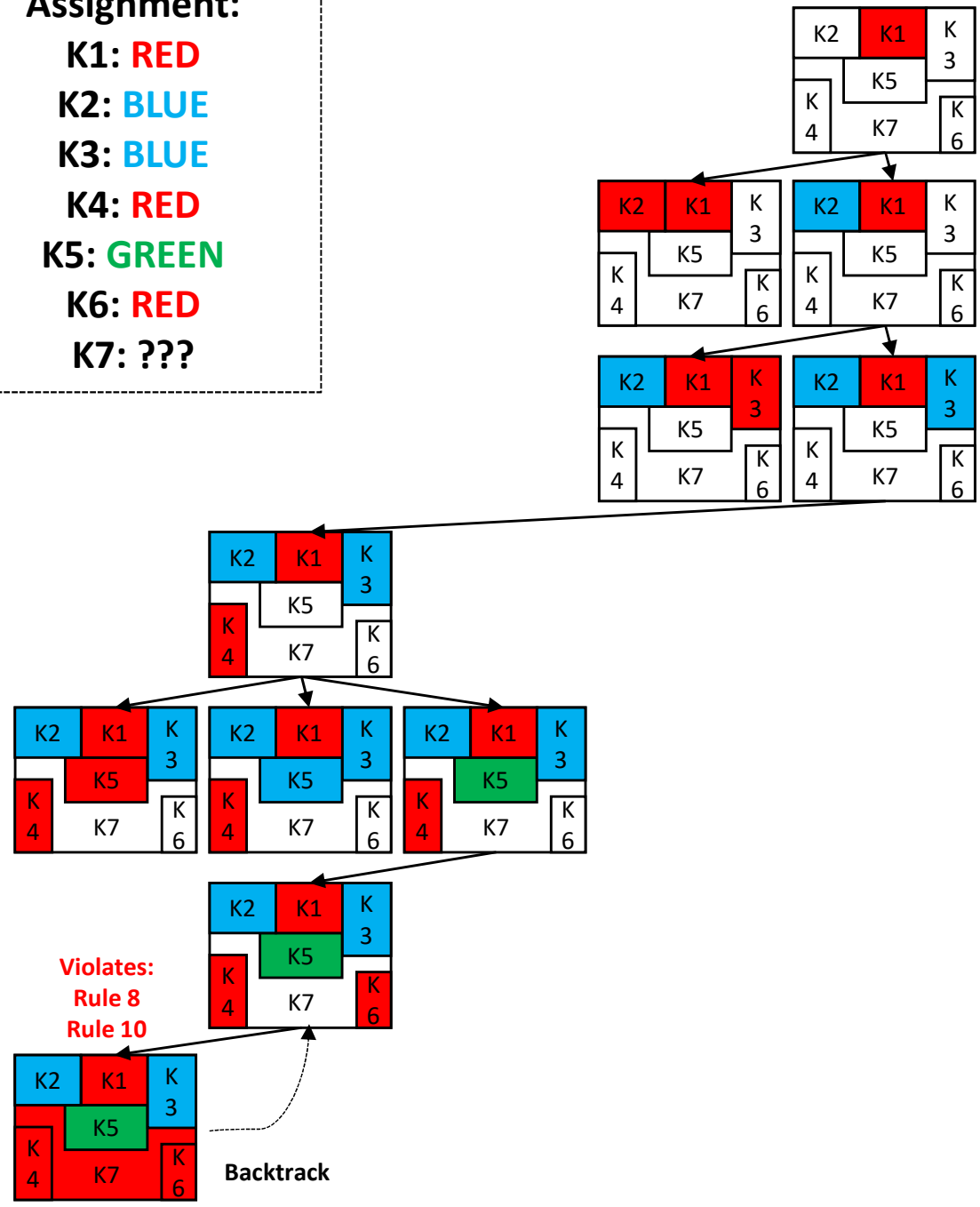
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

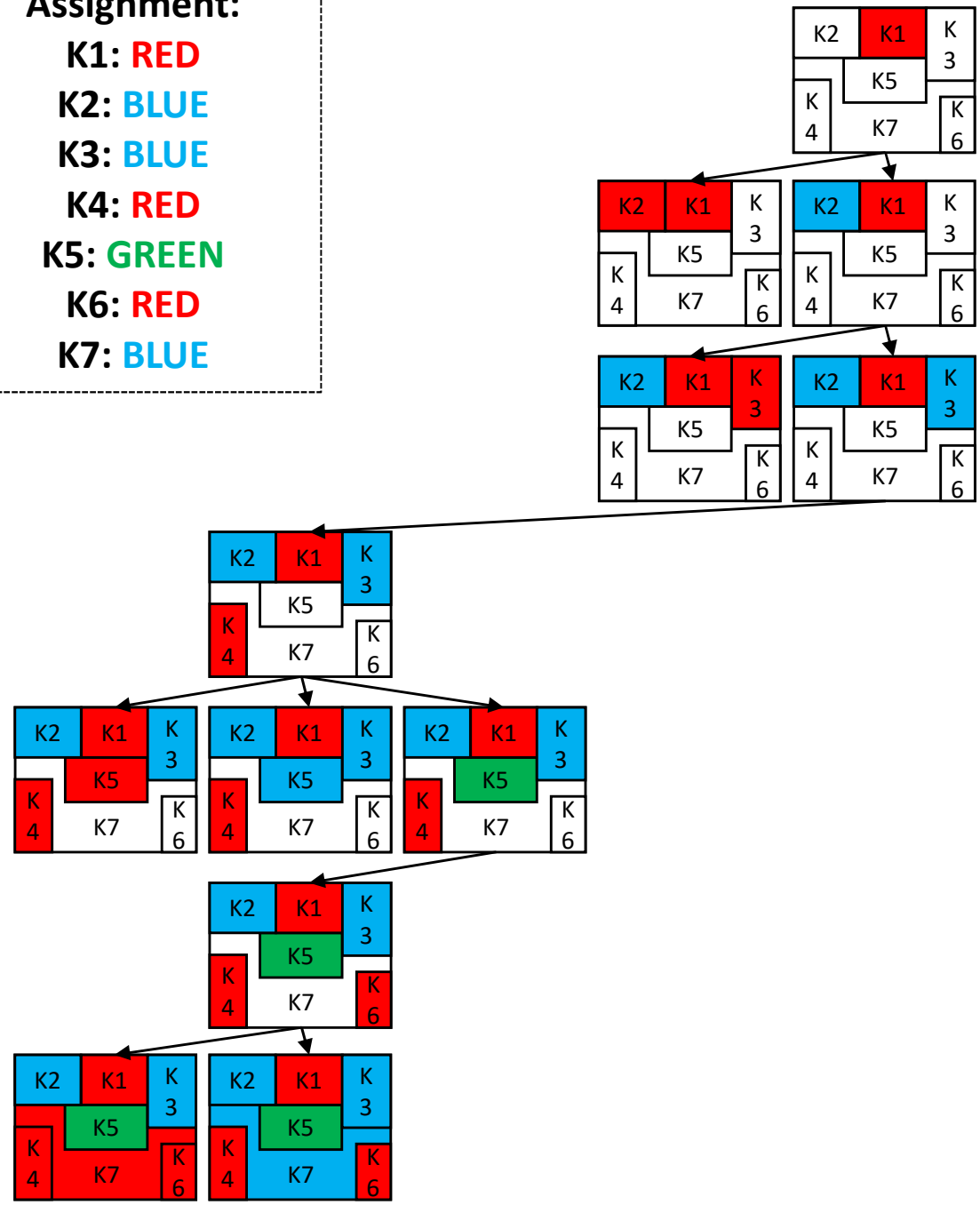
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: BLUE



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

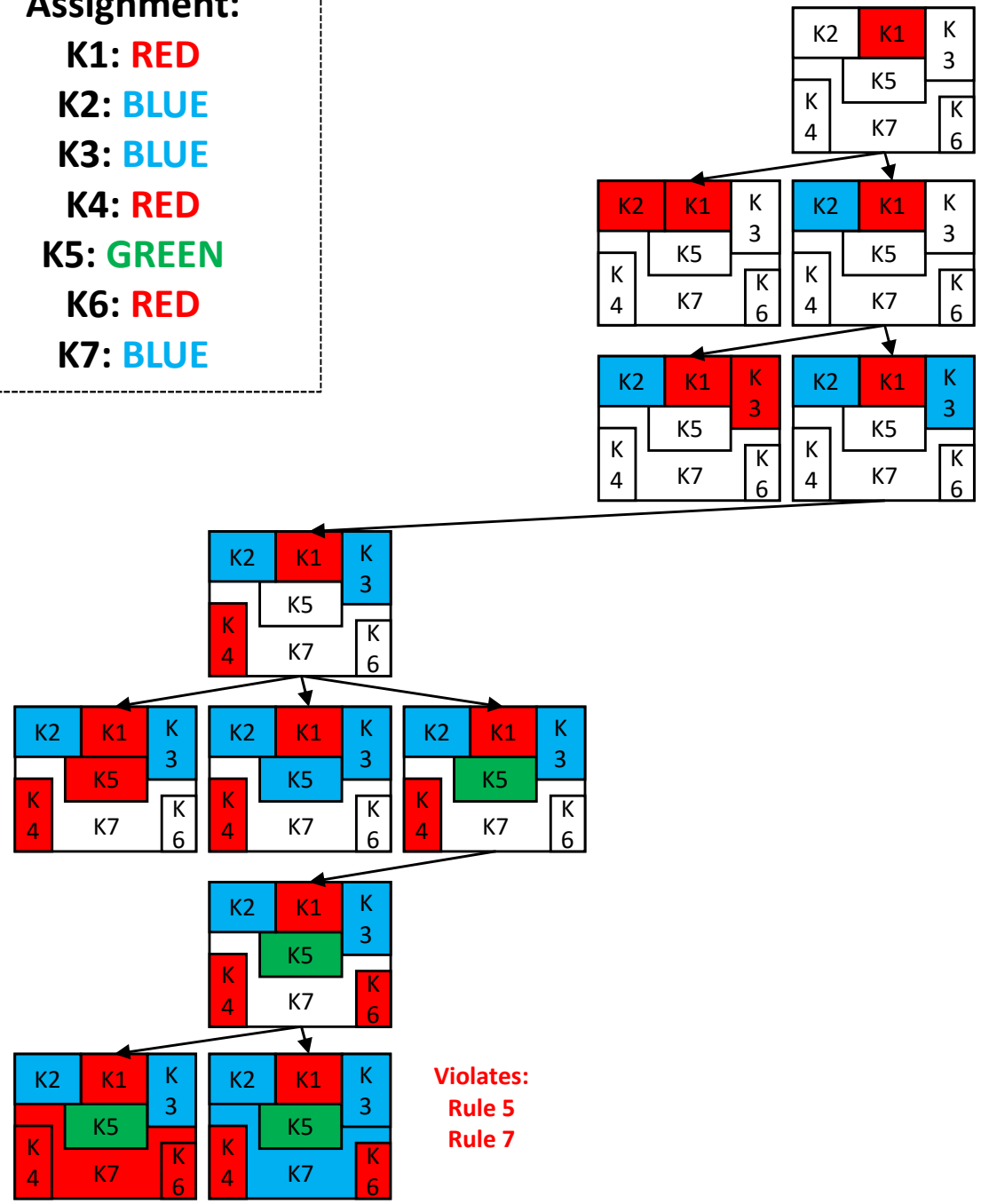
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: BLUE



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

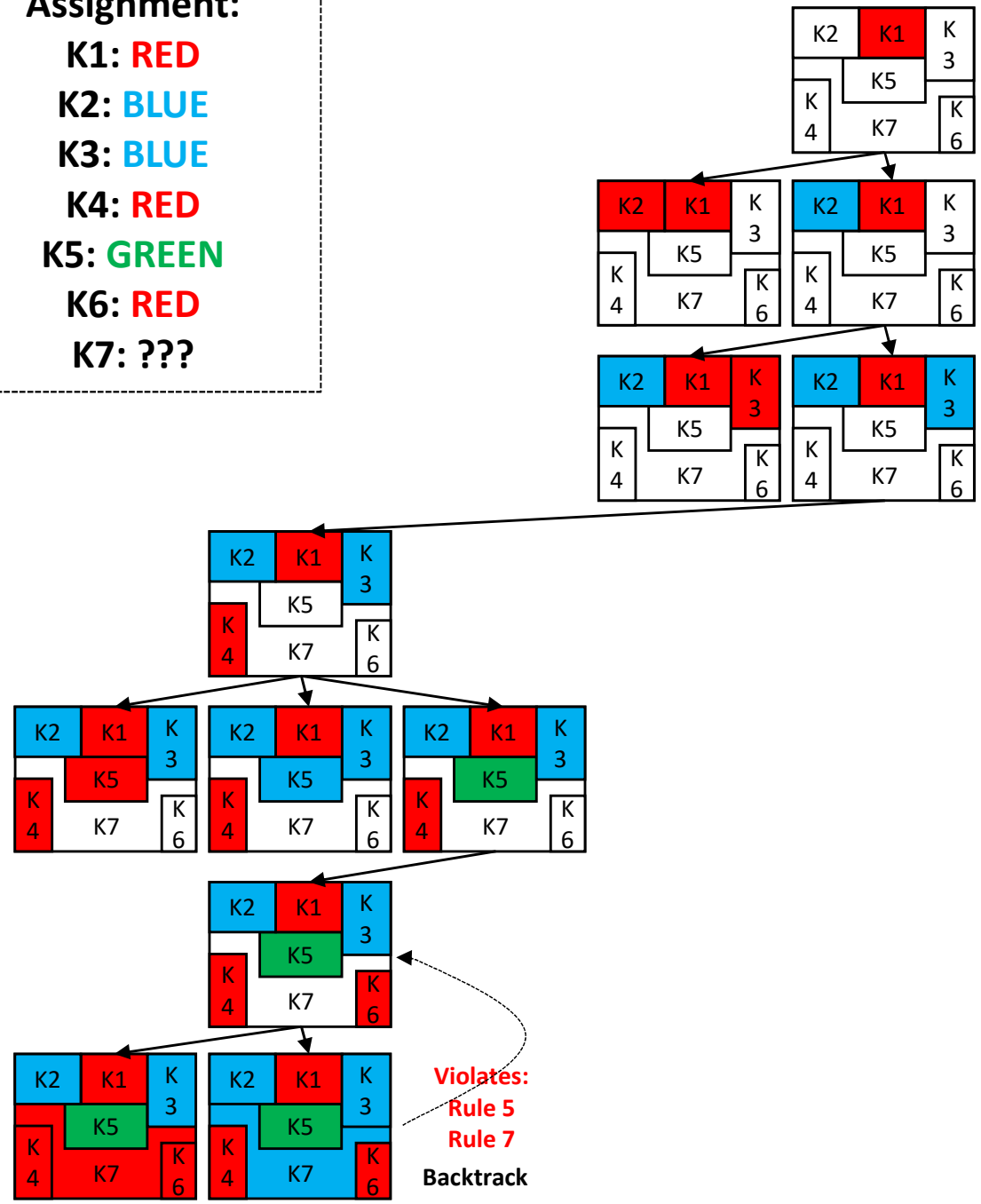
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

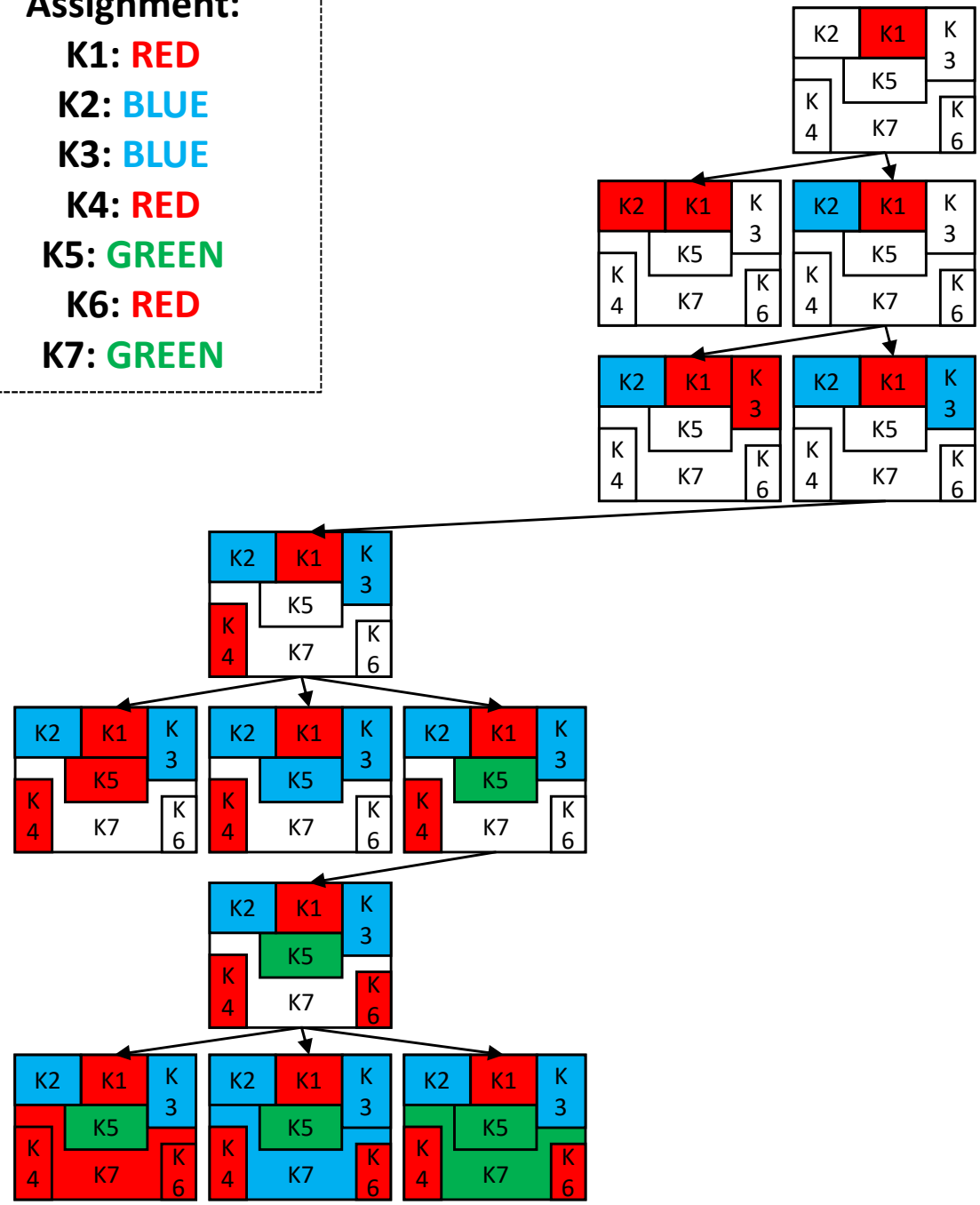
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: GREEN



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

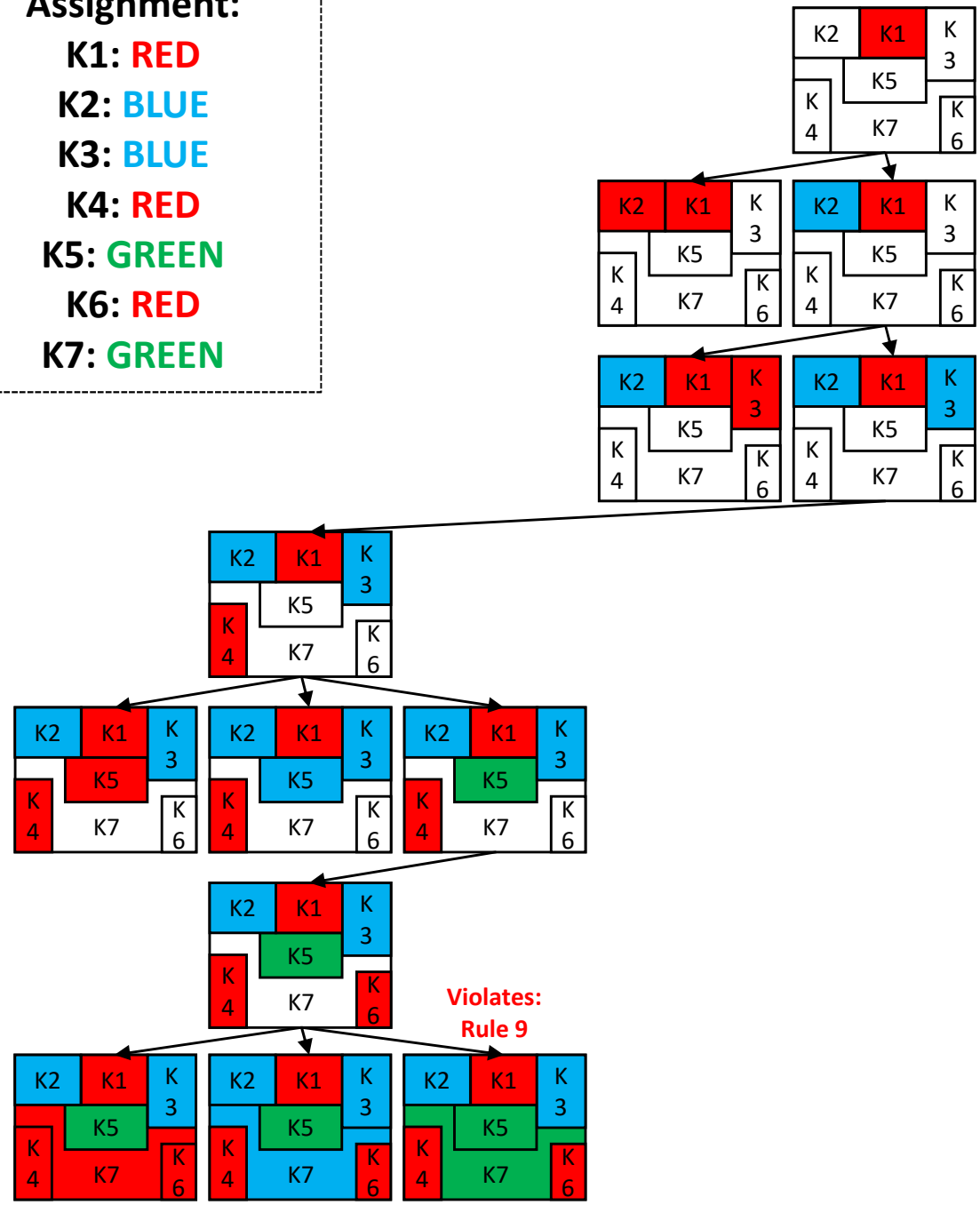
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: GREEN



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

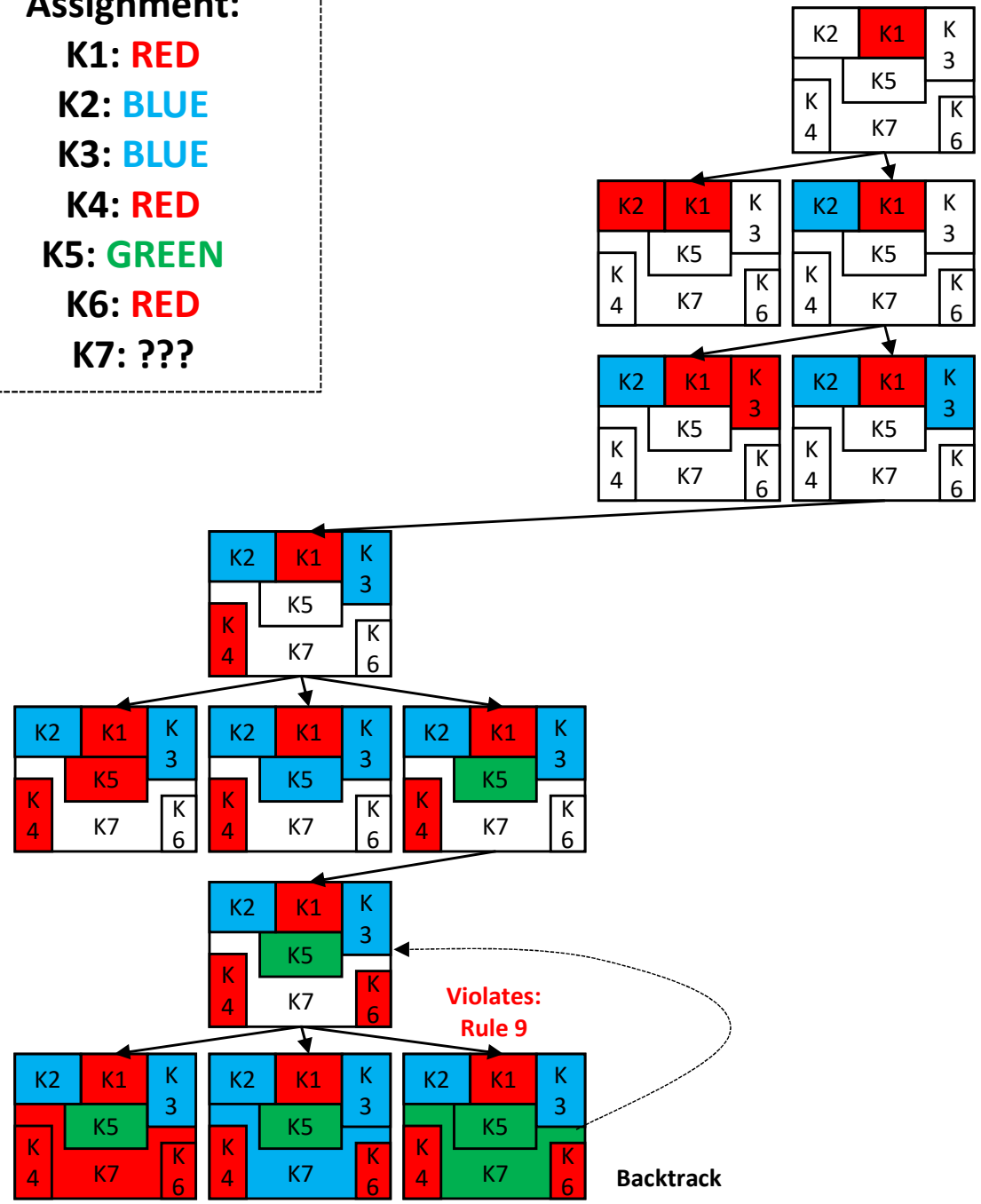
K3: BLUE

K4: RED

K5: GREEN

K6: RED

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

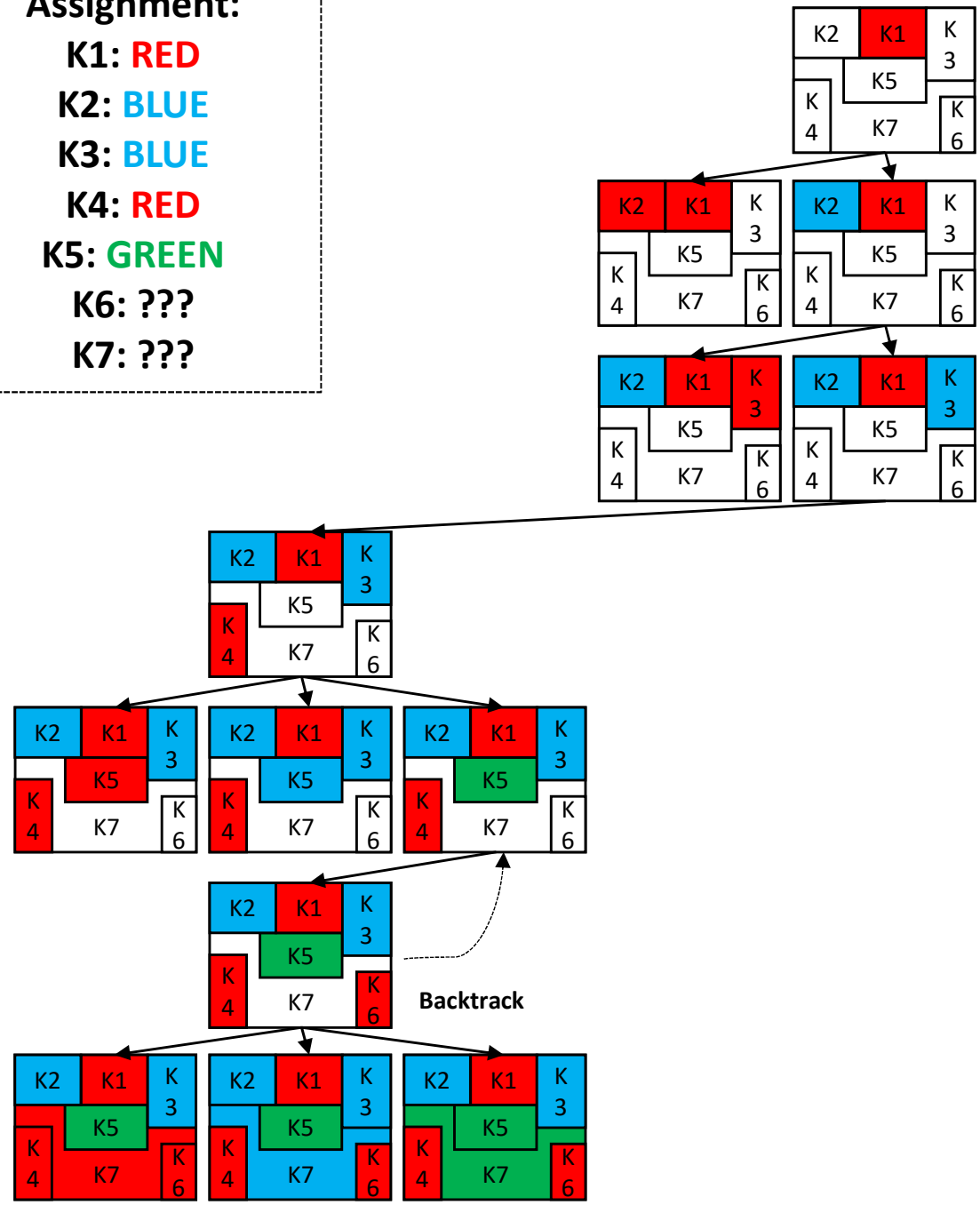
K3: BLUE

K4: RED

K5: GREEN

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

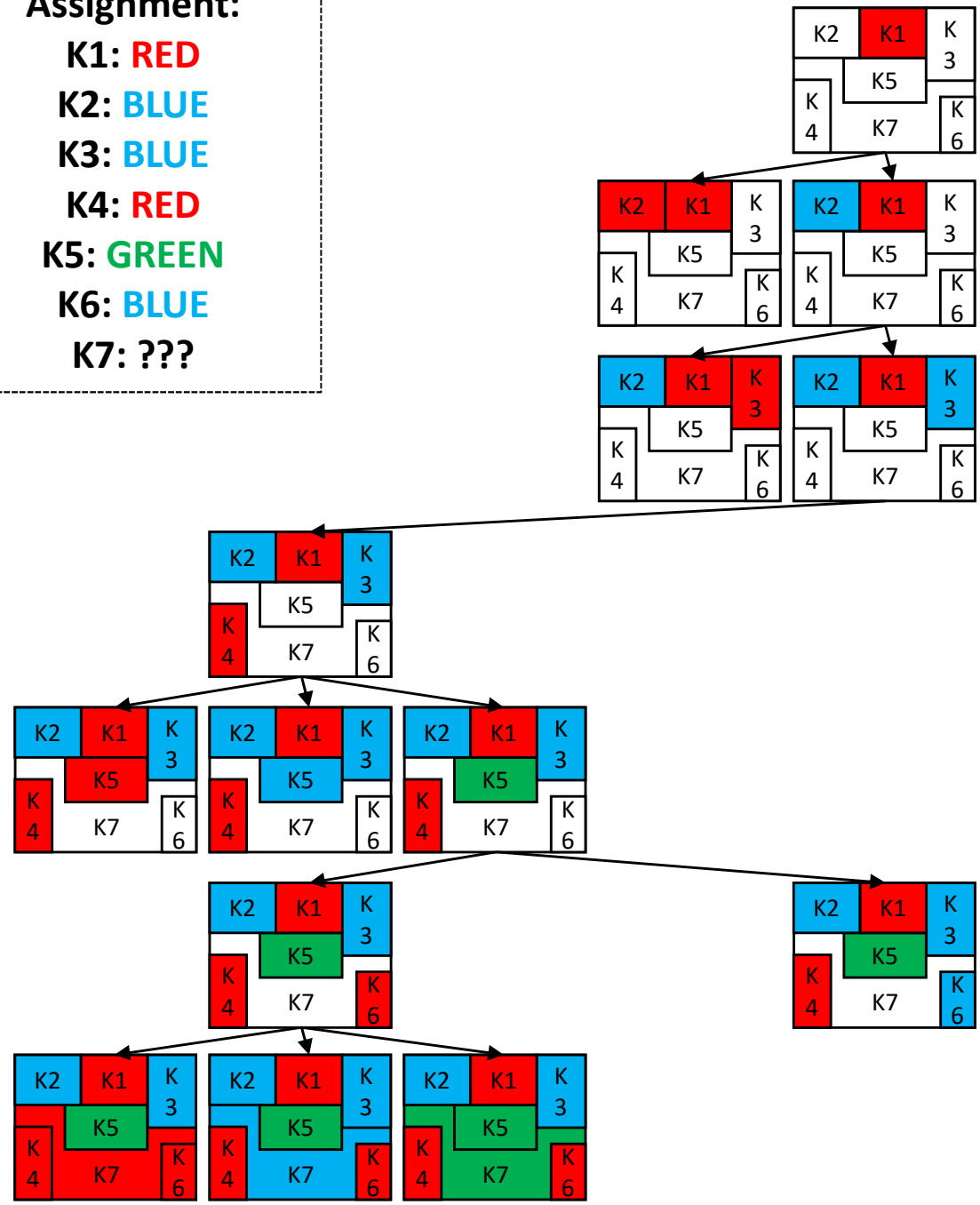
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

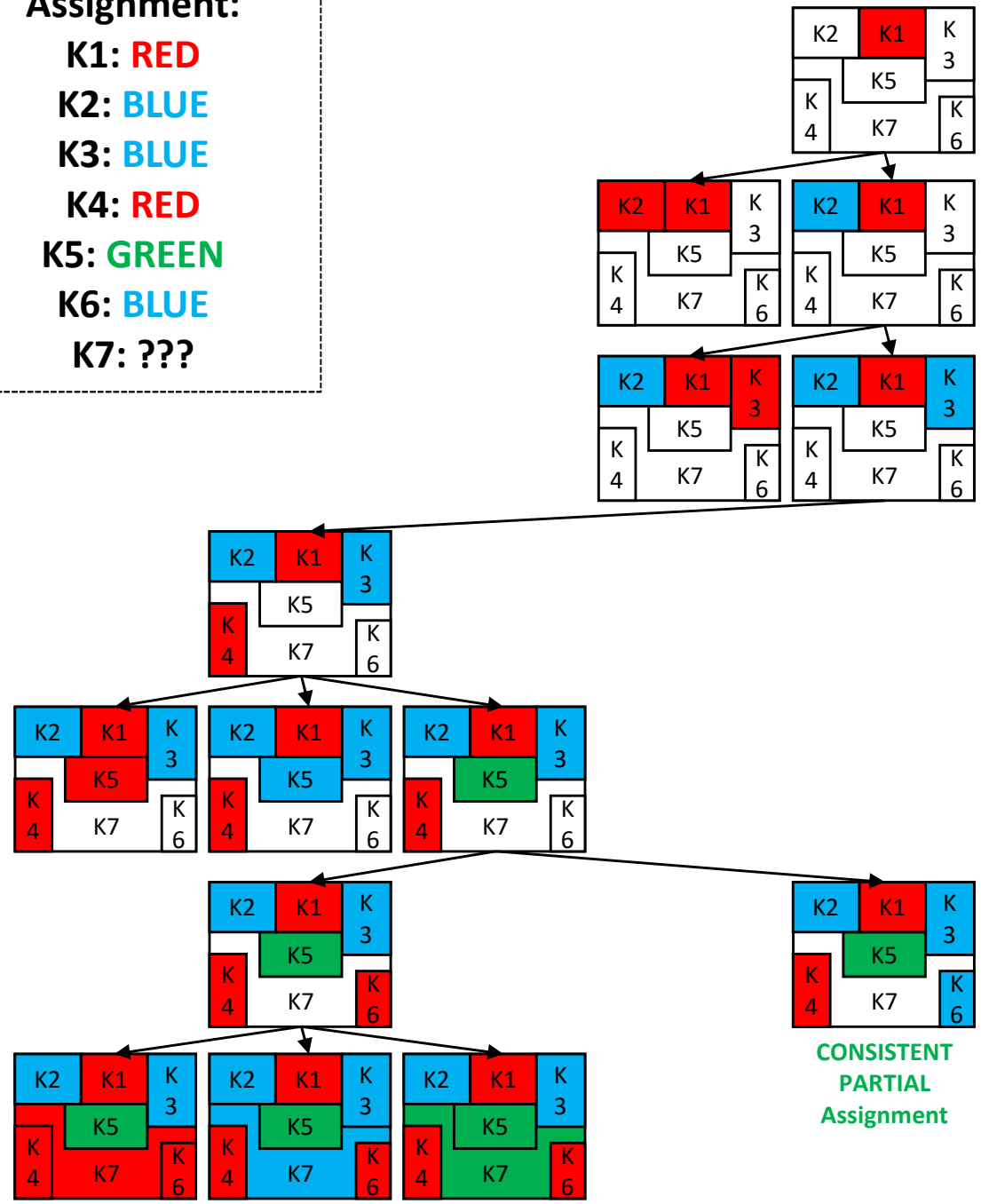
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

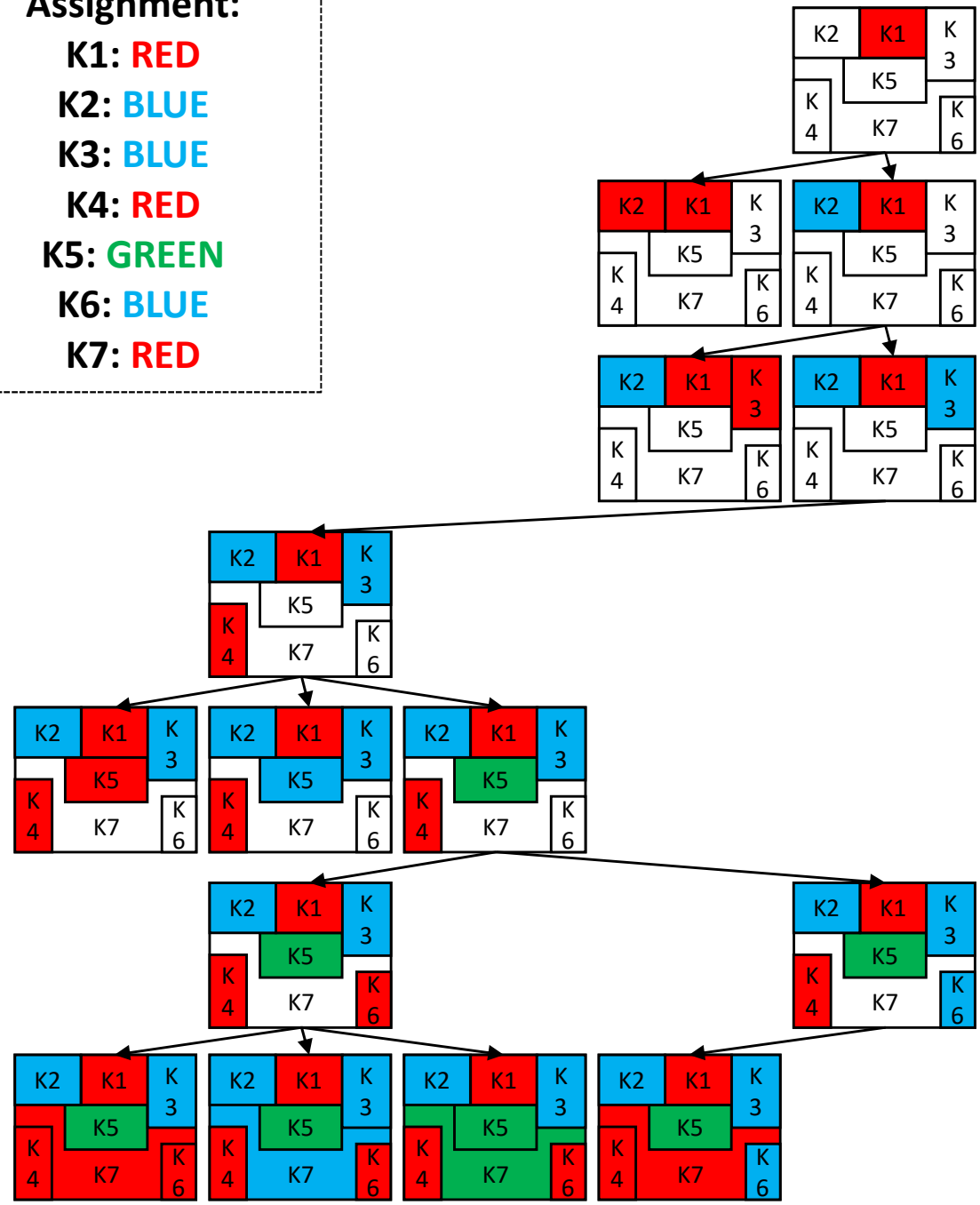
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: RED



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

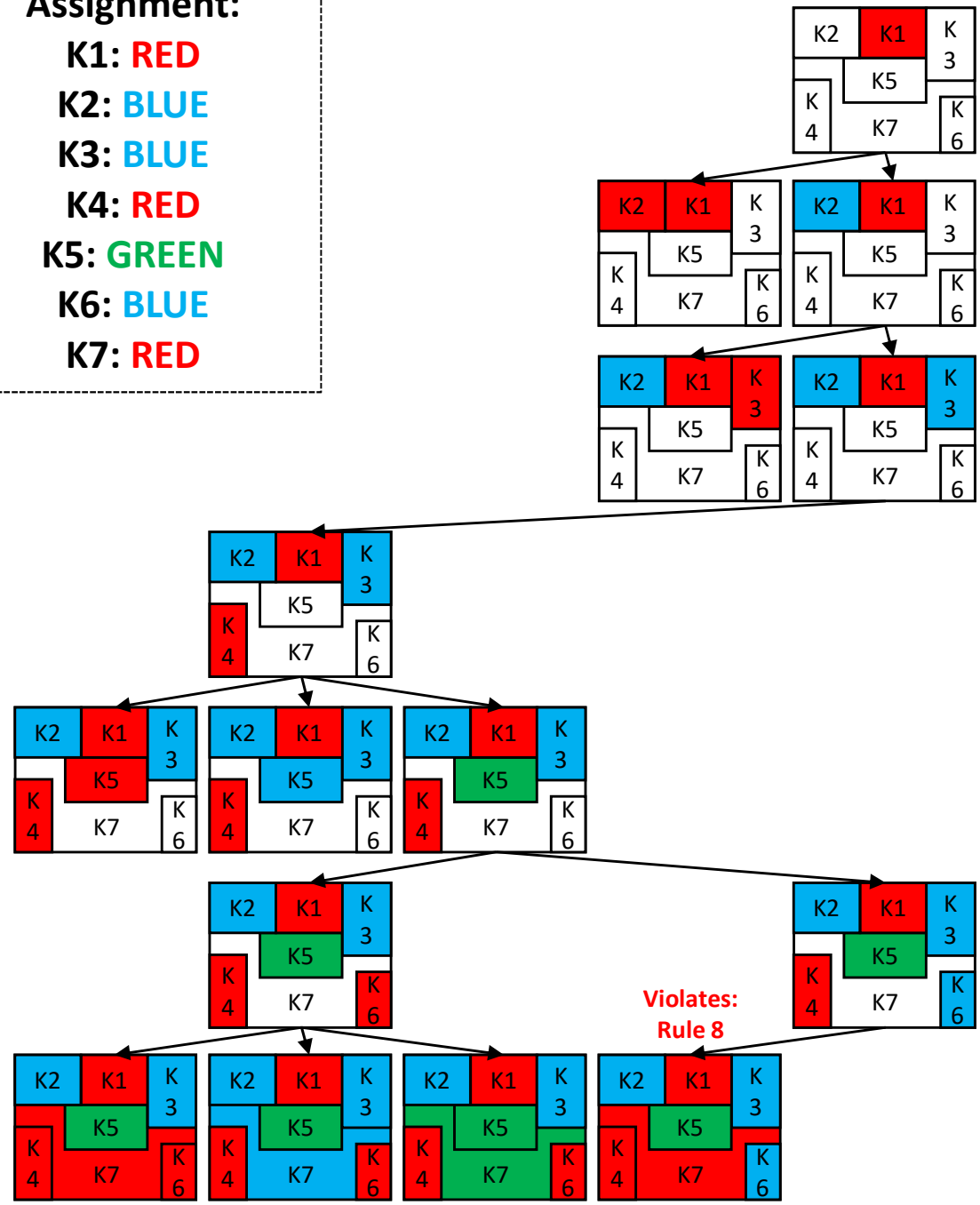
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: RED



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

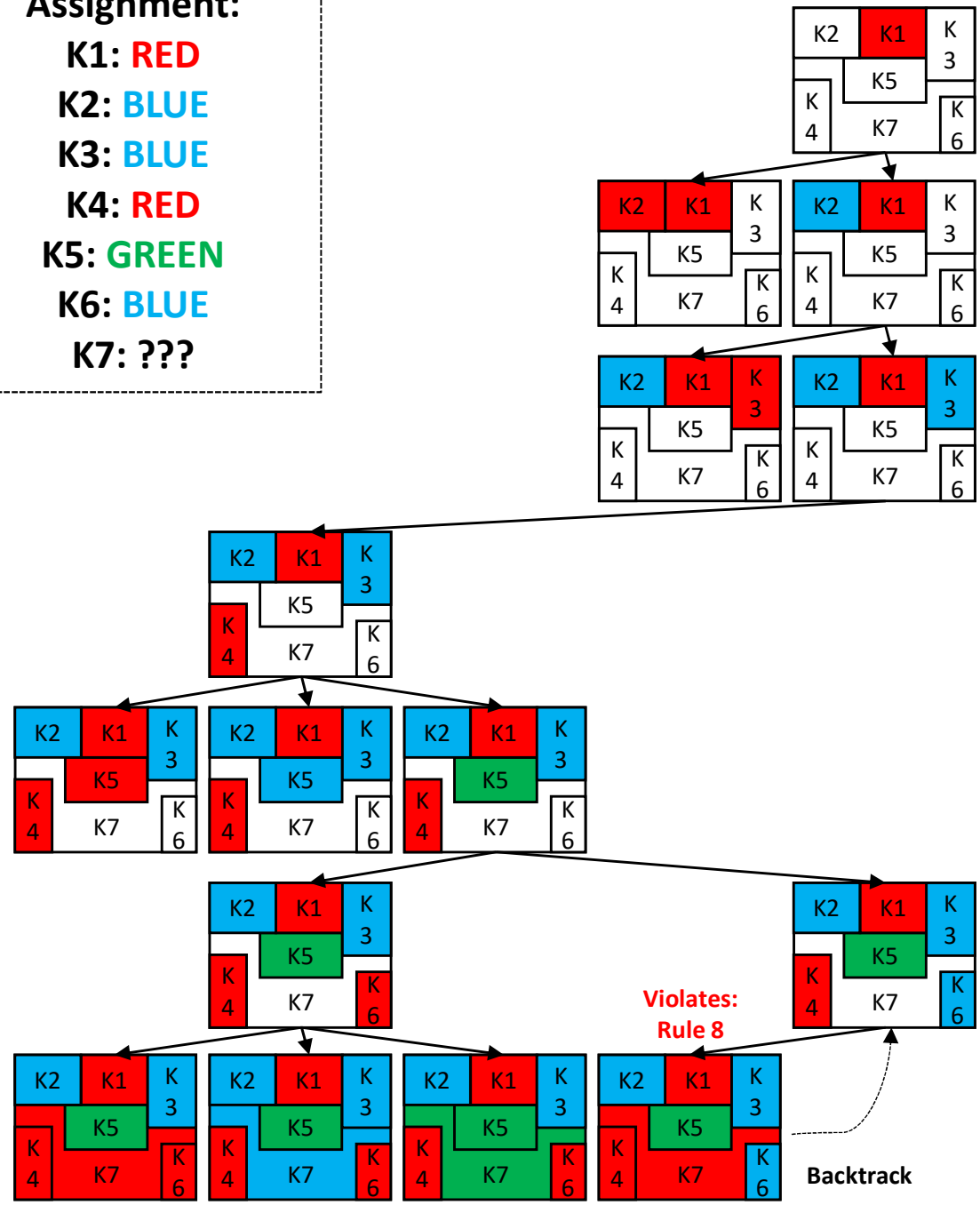
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

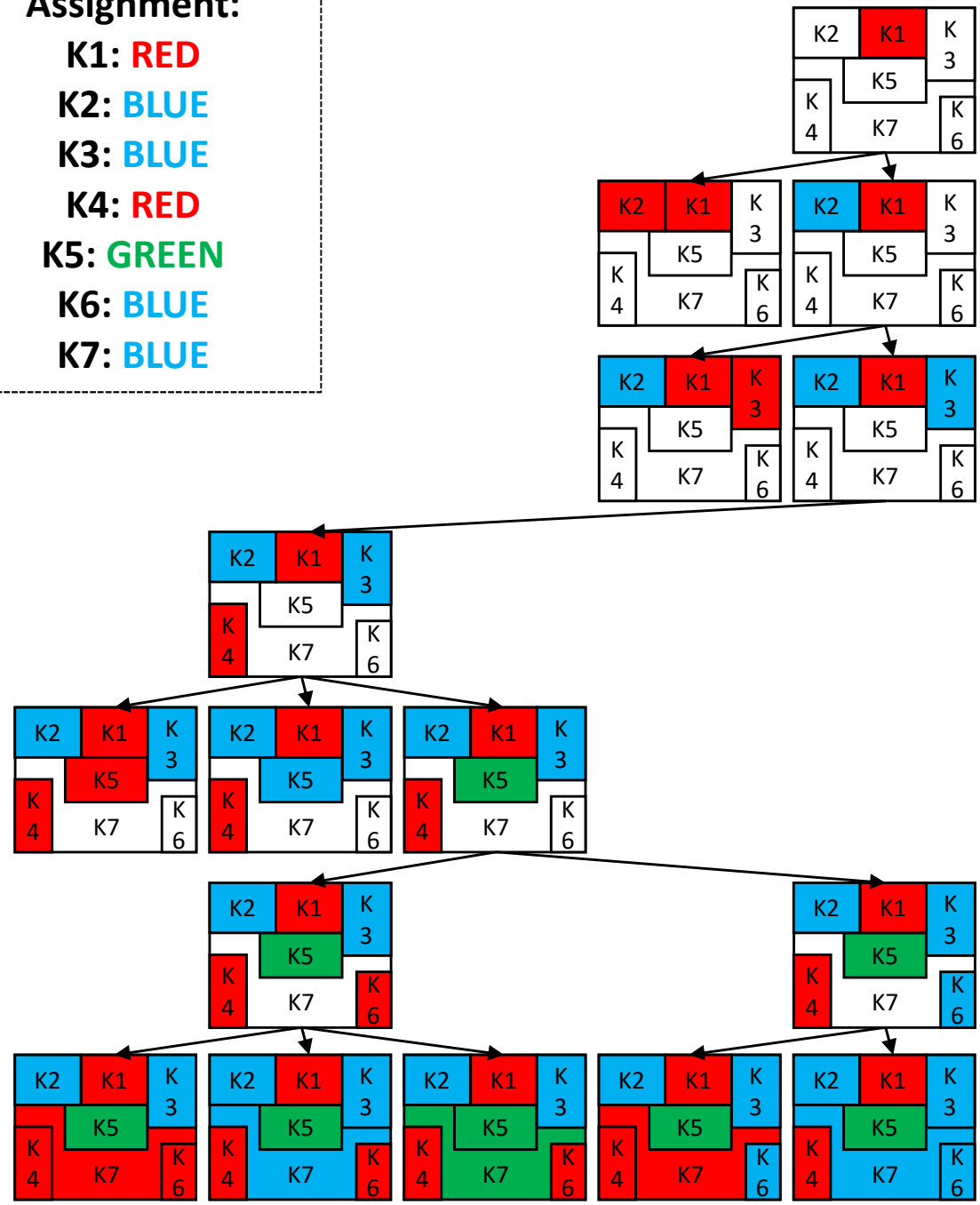
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: BLUE



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

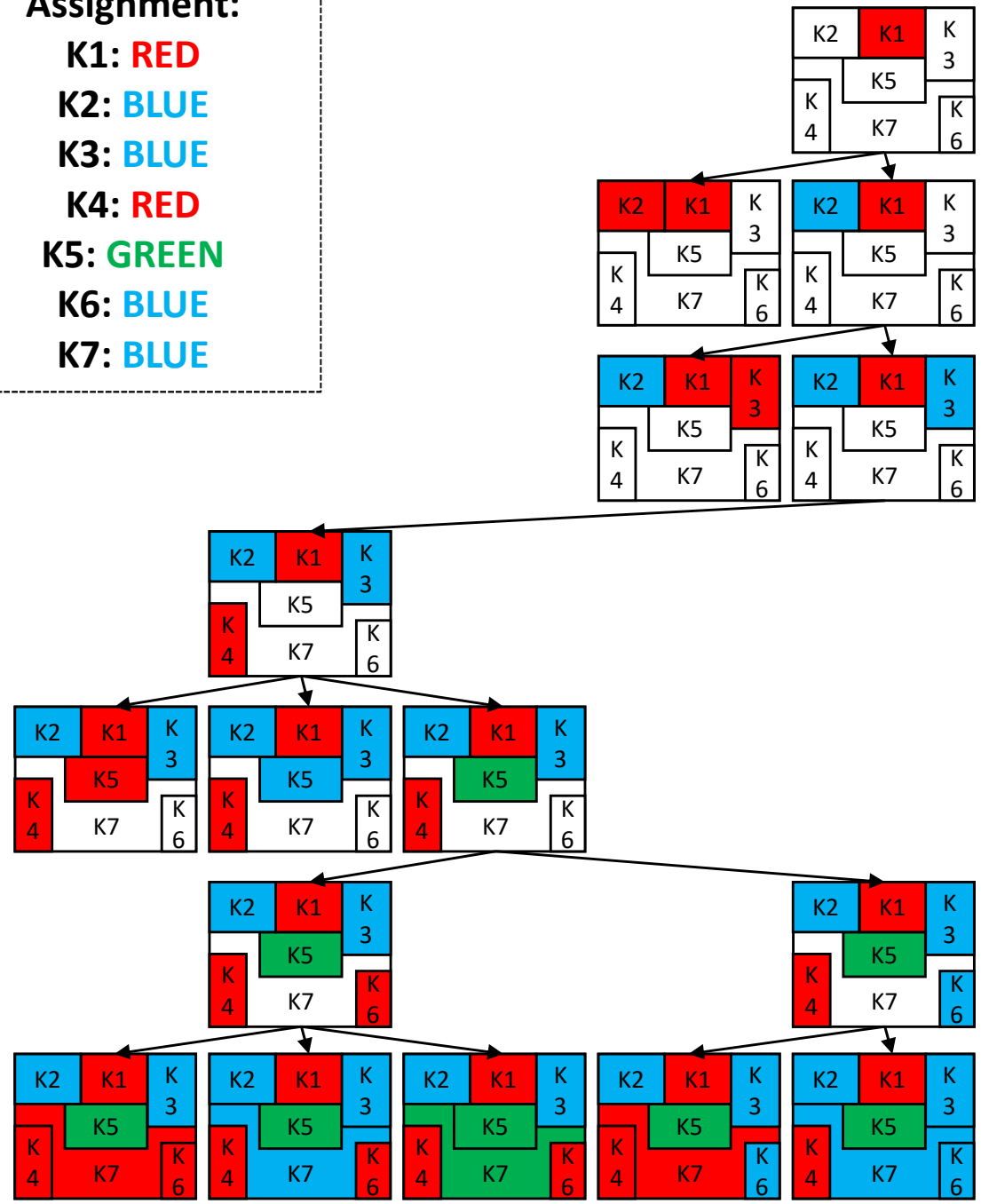
Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED
K2: BLUE
K3: BLUE
K4: RED
K5: GREEN
K6: BLUE
K7: BLUE



Violates:
Rule 5
Rule 7
Rule 10

Constraints:

Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

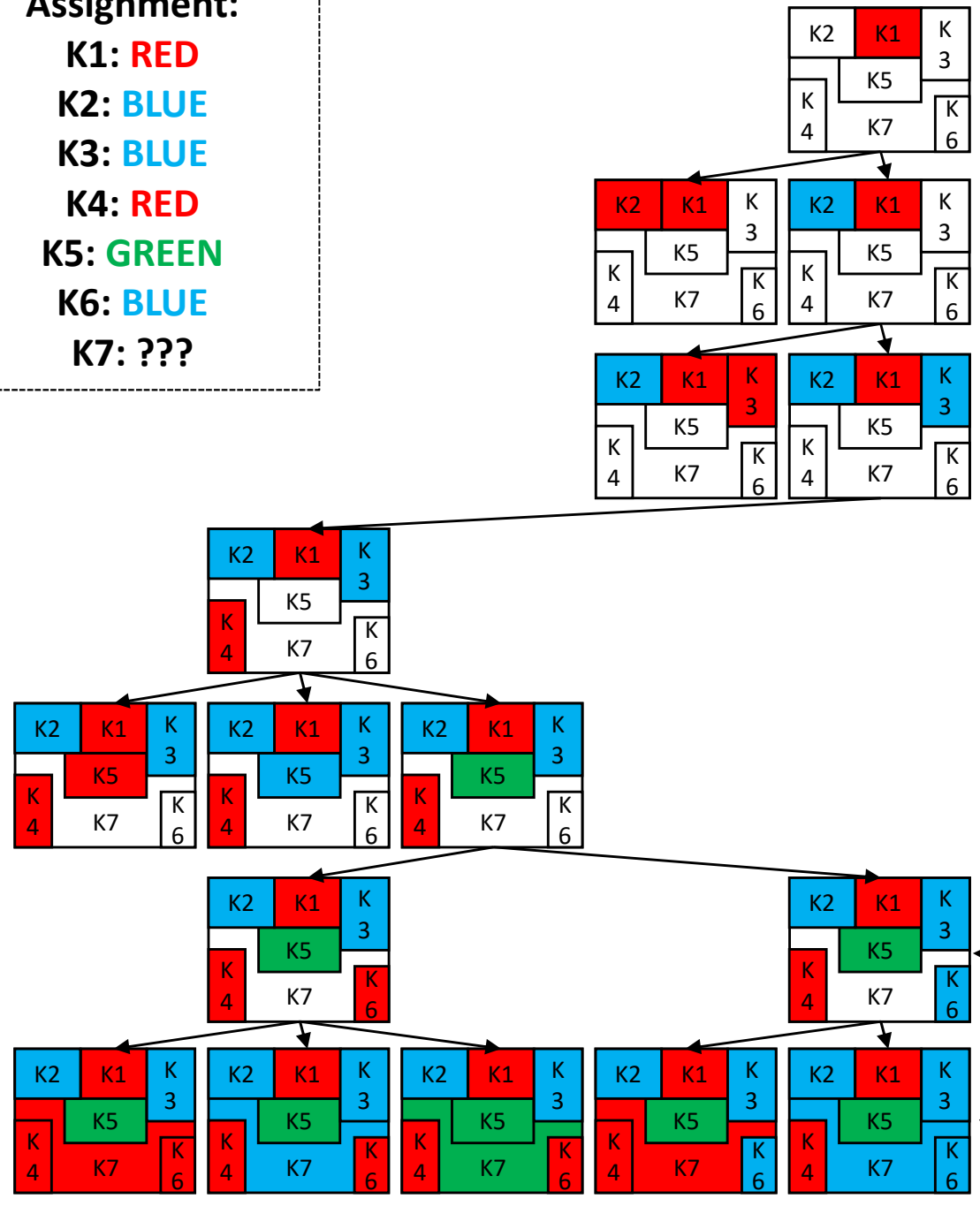
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

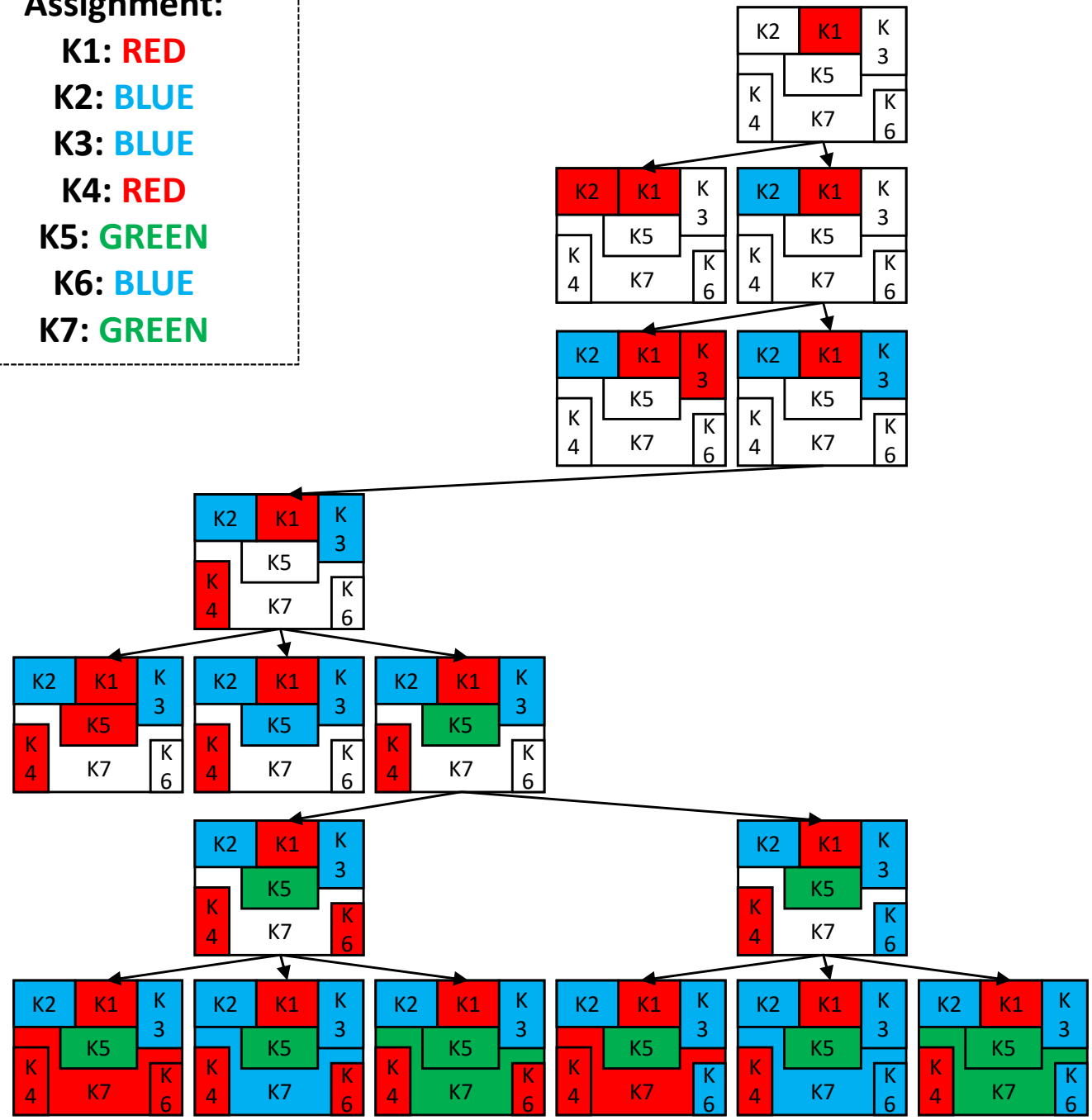
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: GREEN



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

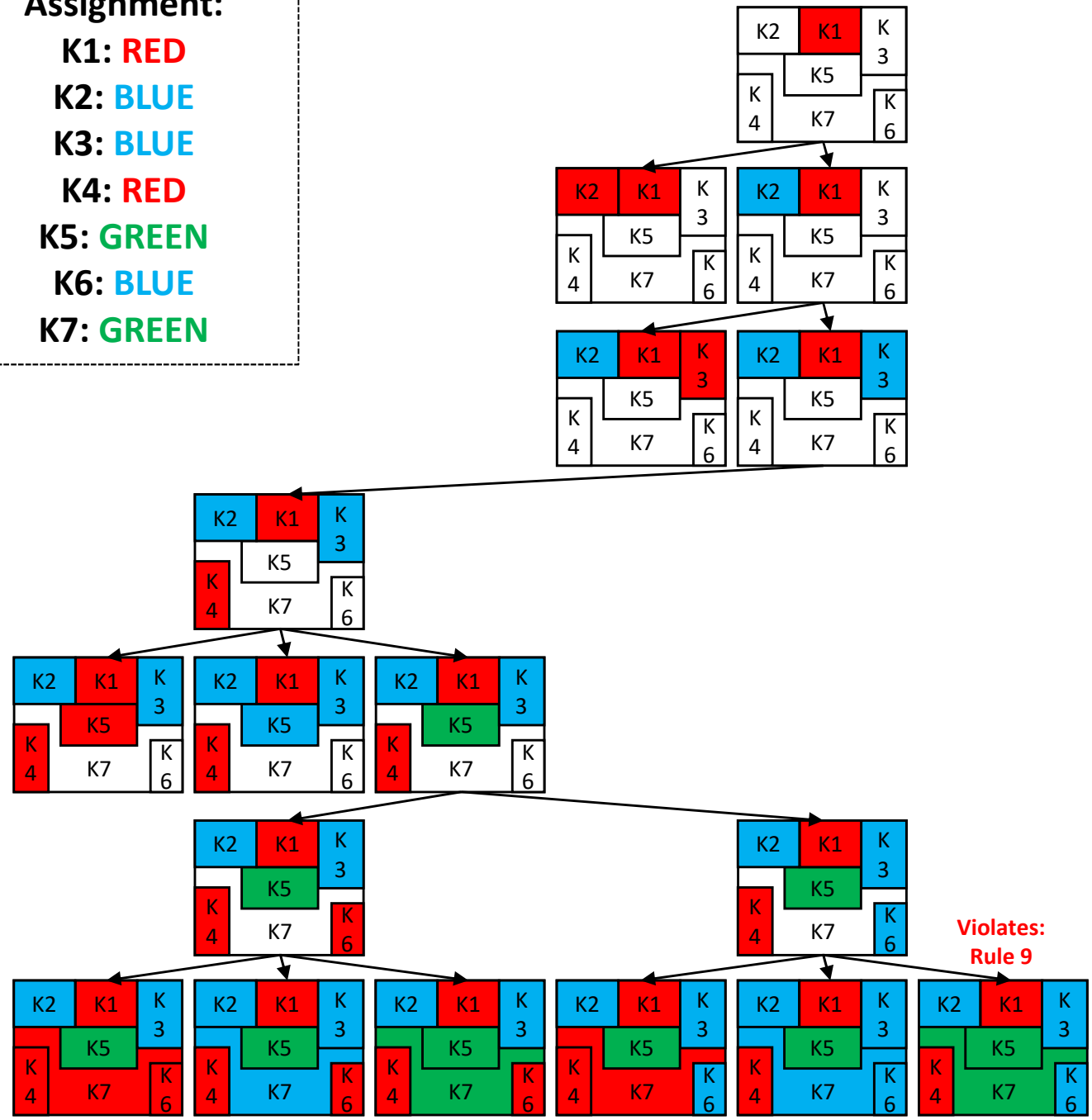
K3: BLUE

K4: RED

K5: GREEN

K6: BLUE

K7: GREEN



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

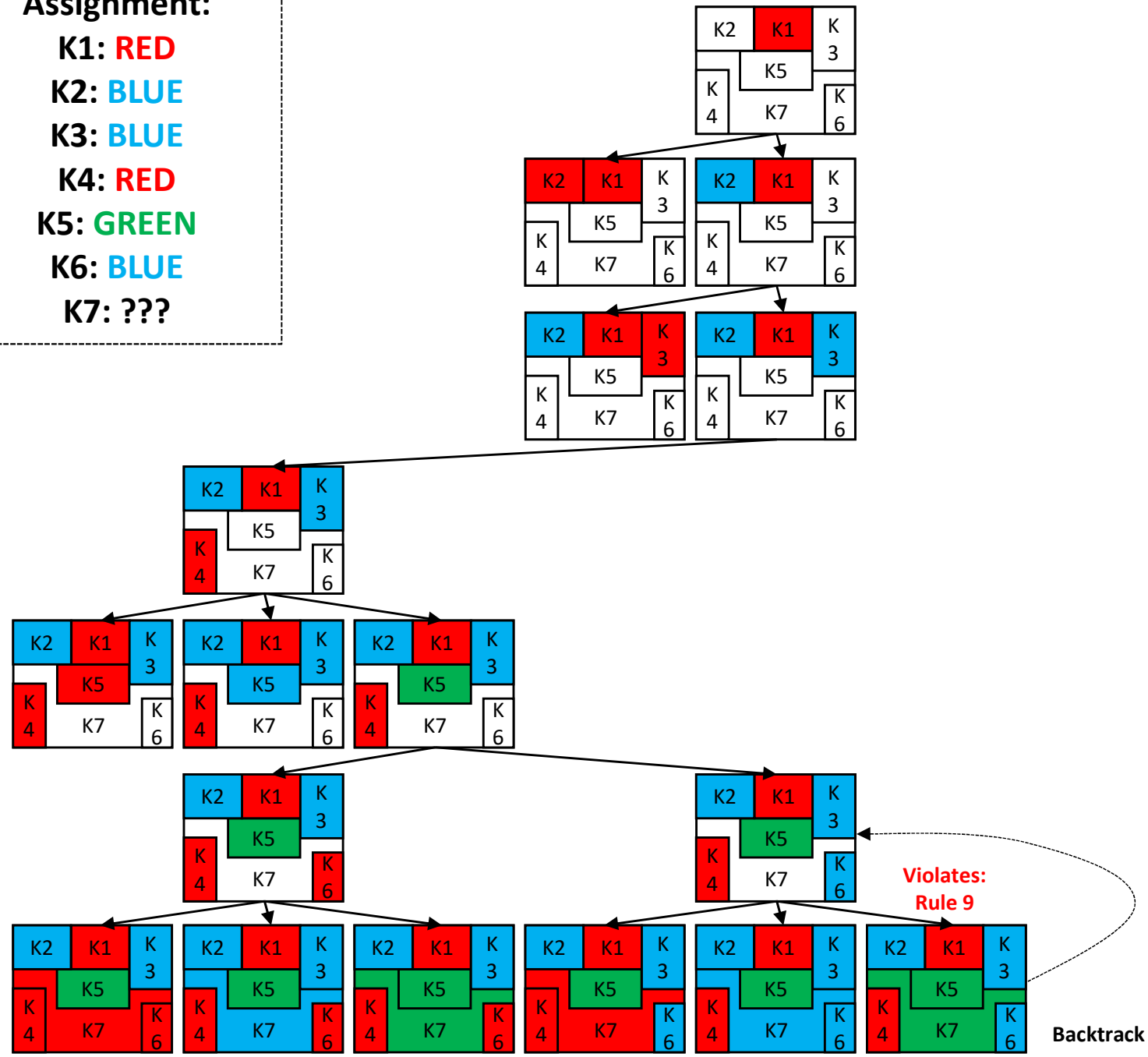
Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: RED
K5: GREEN
K6: BLUE
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

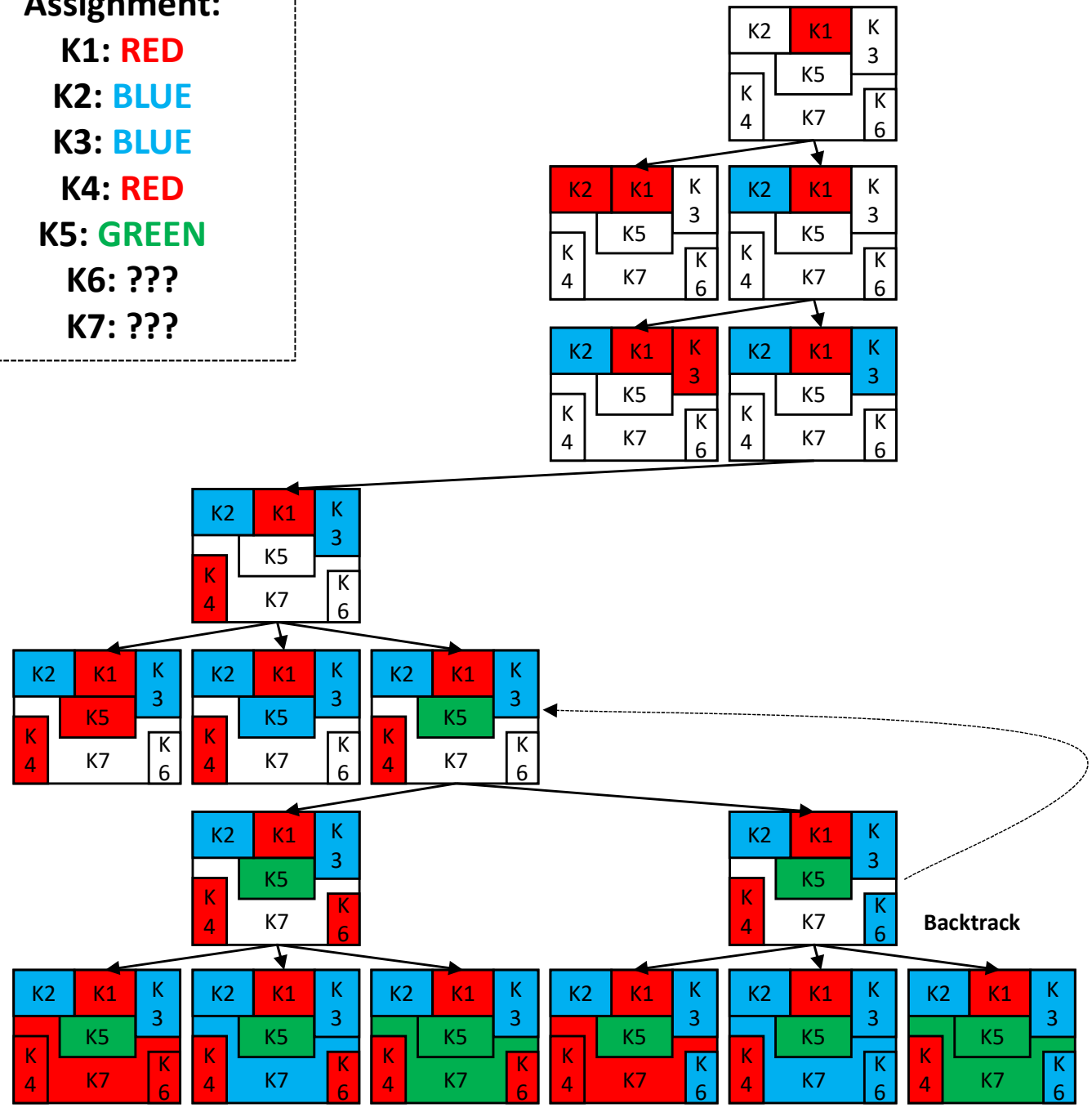
K3: BLUE

K4: RED

K5: GREEN

K6: ???

K7: ???



Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

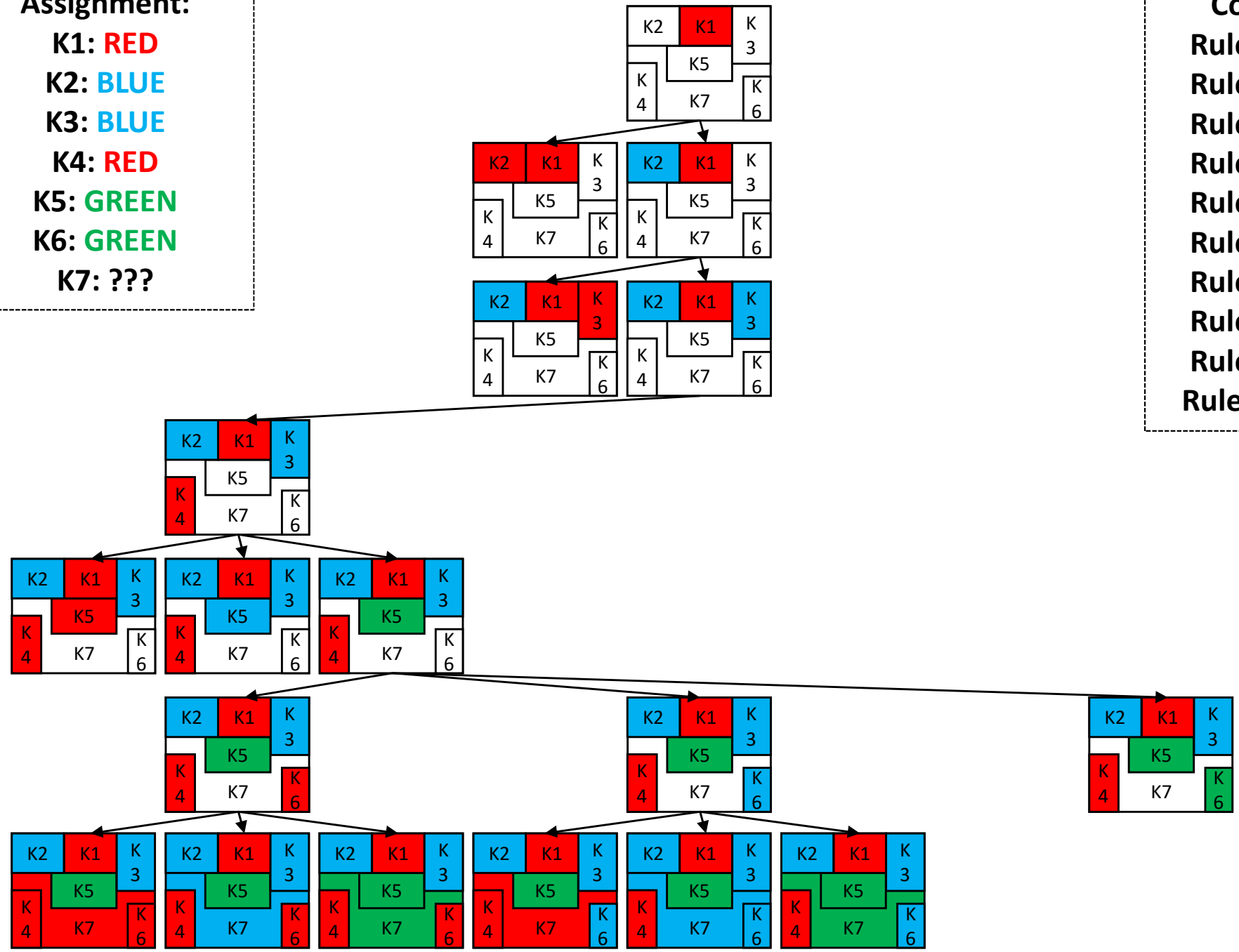
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

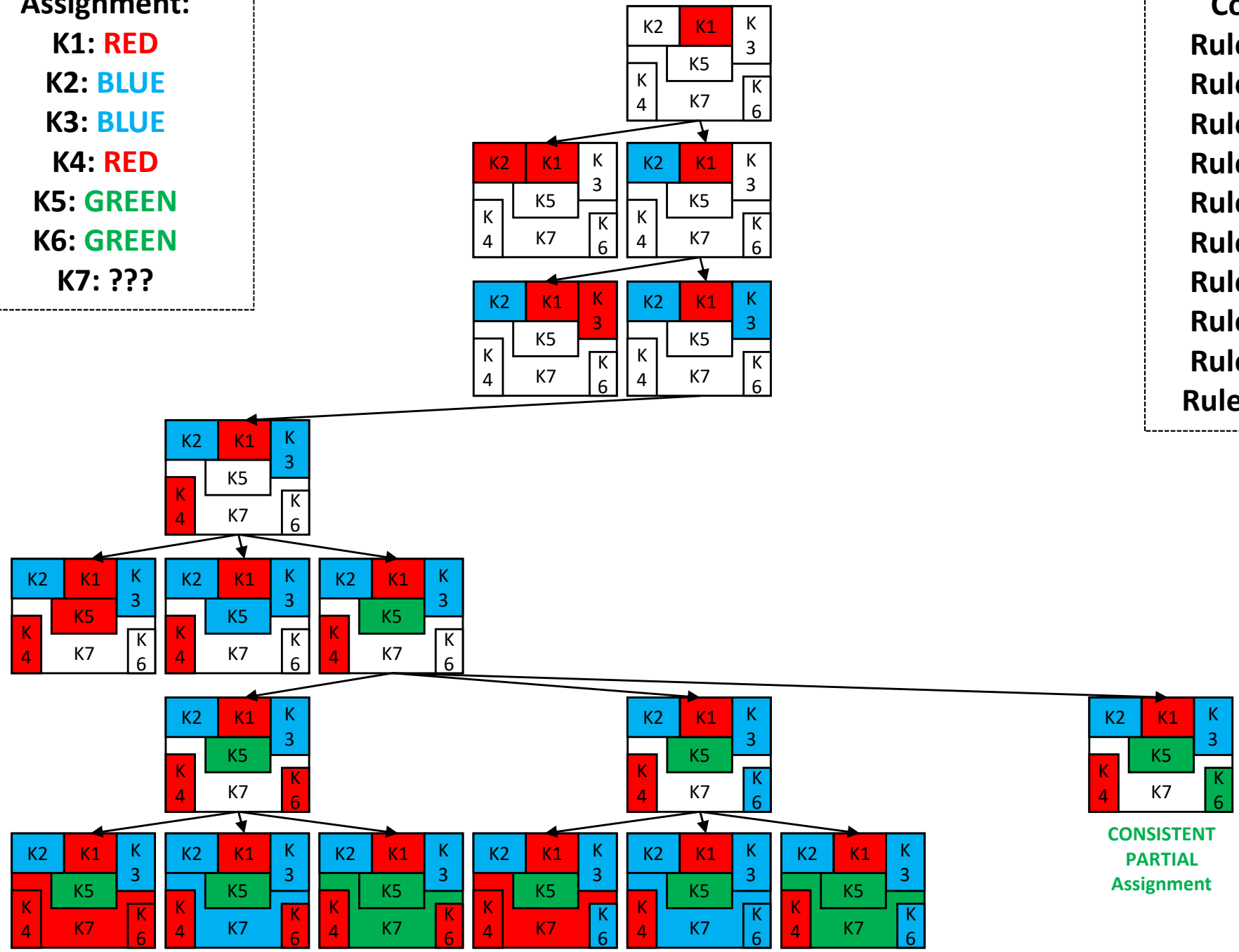
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: RED

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

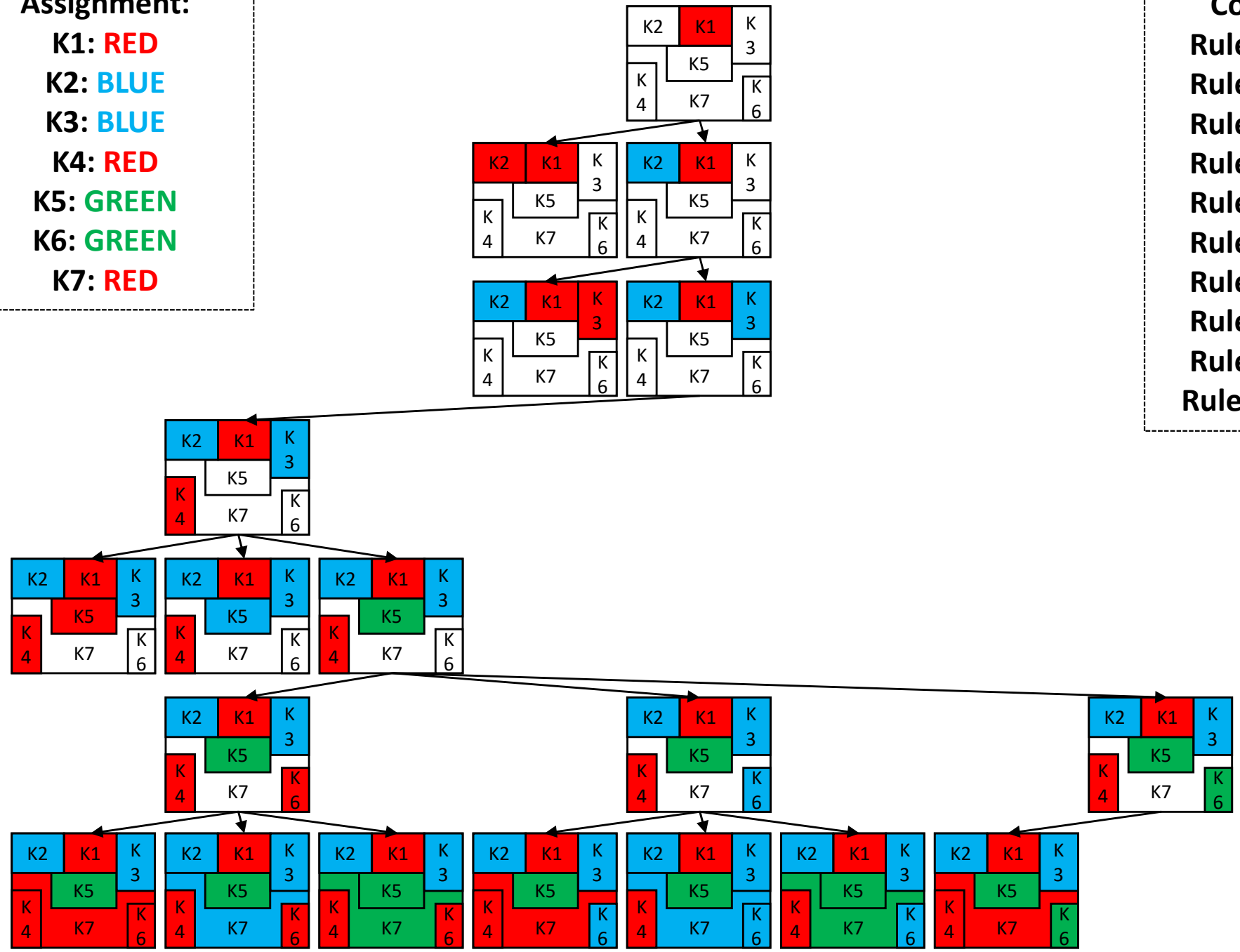
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: RED

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

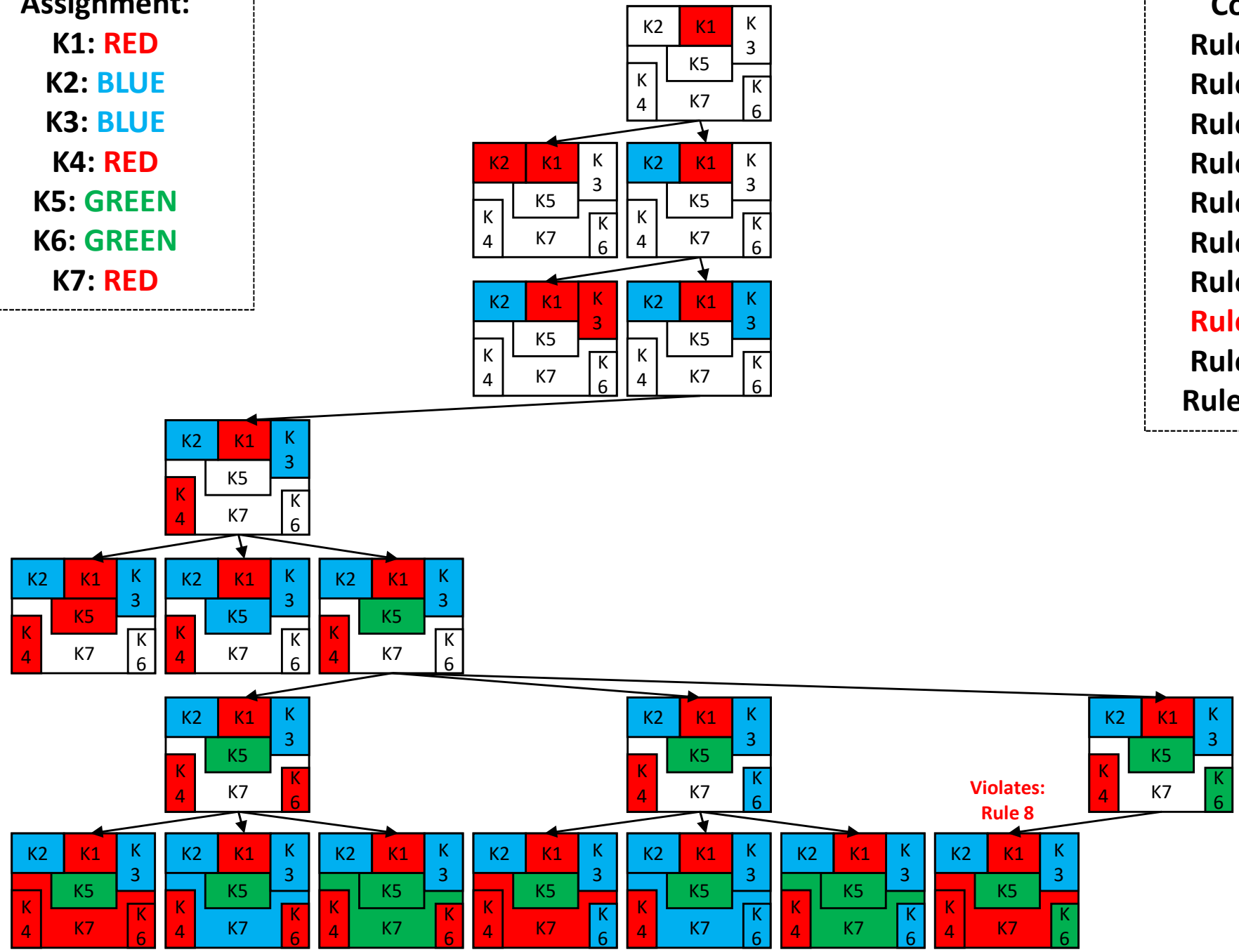
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

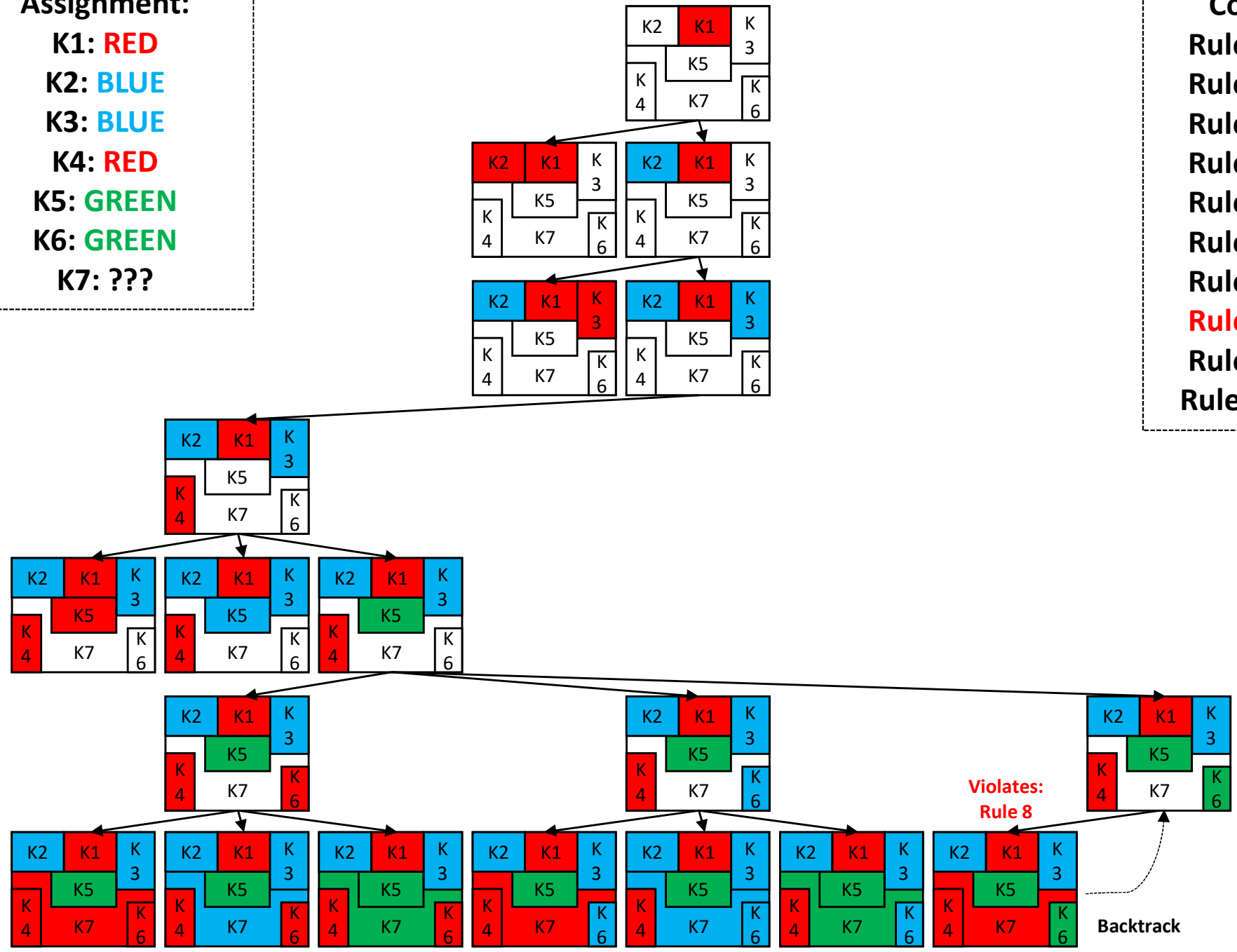
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: BLUE

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

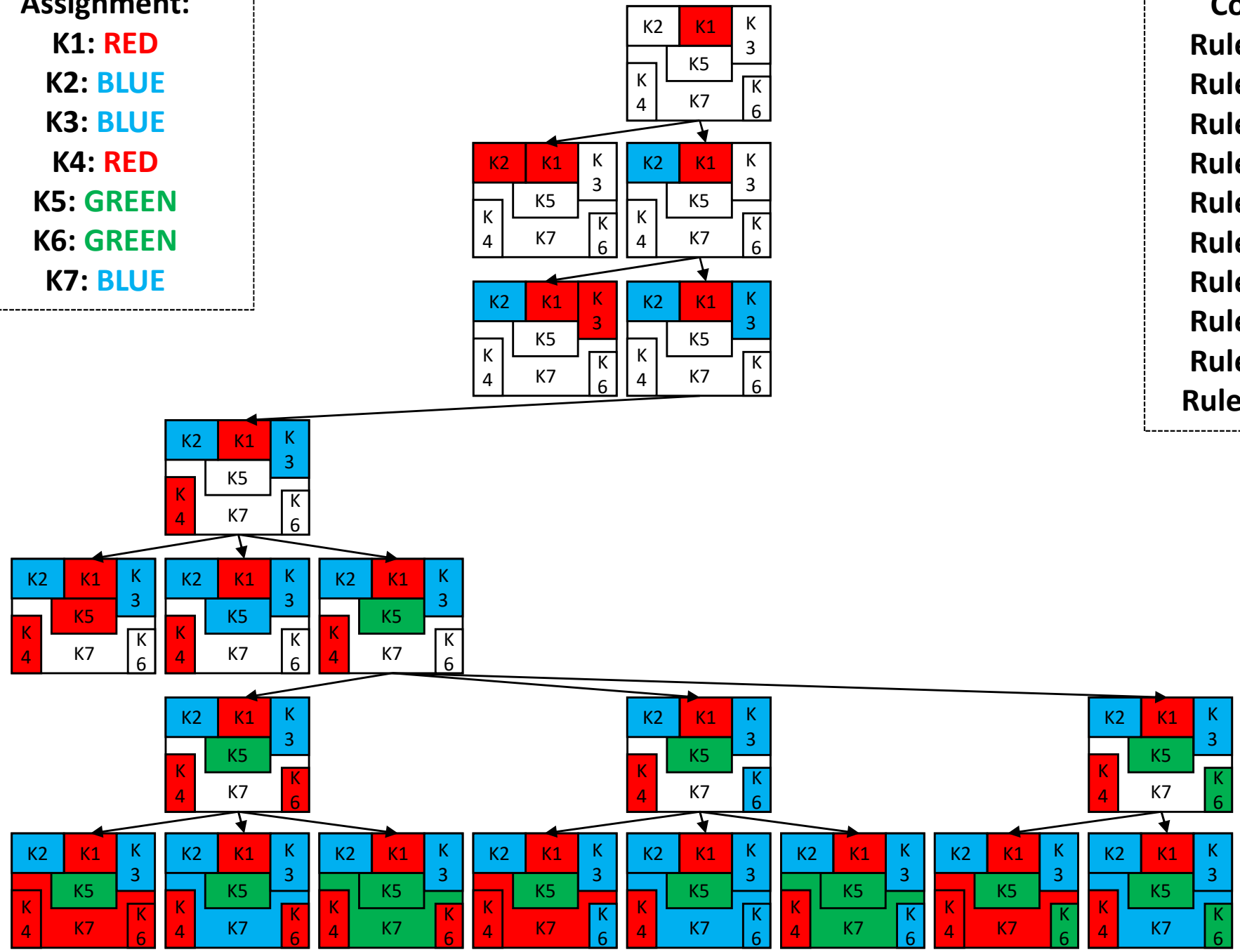
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: BLUE

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

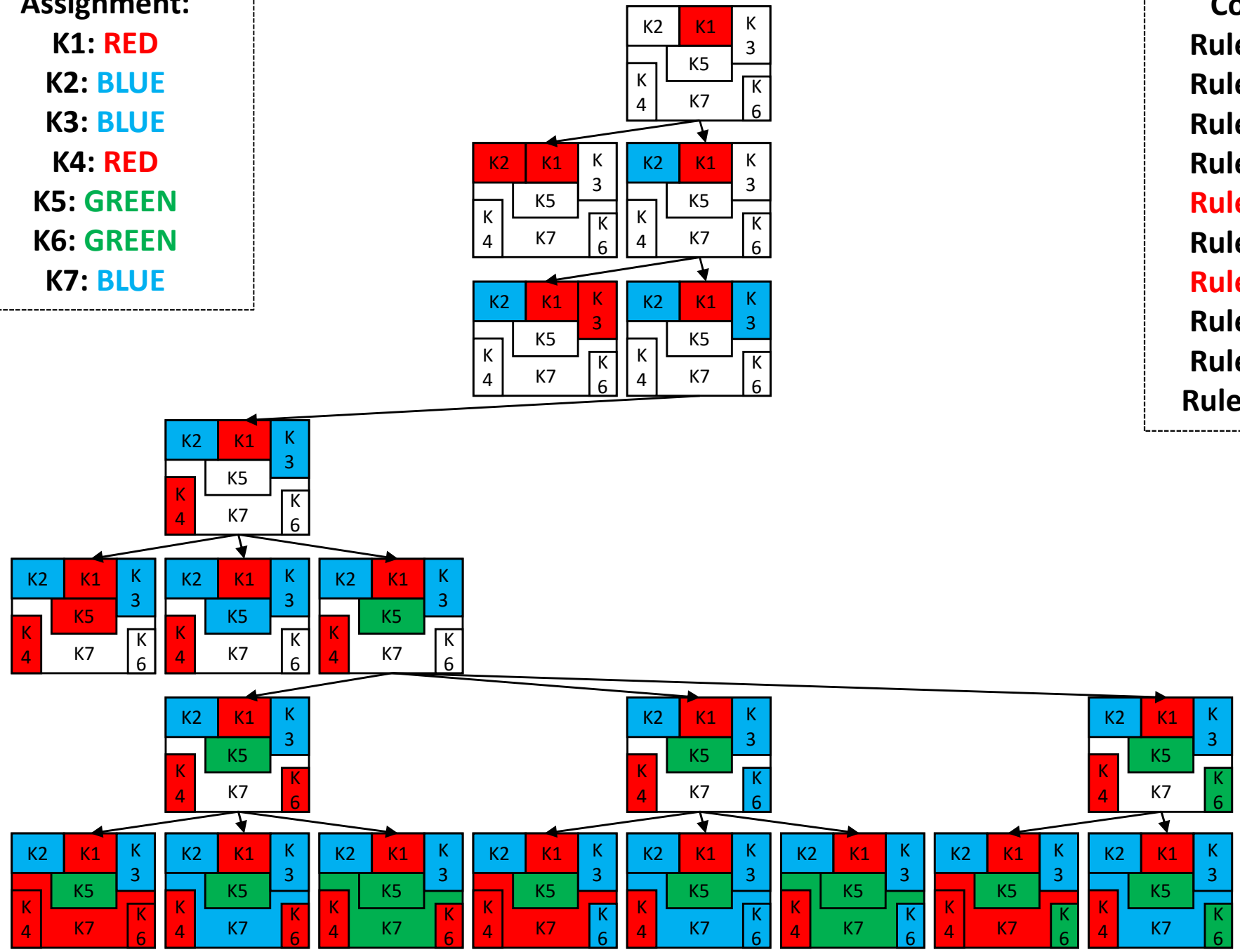
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Violates:
Rule 7
Rule 10

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

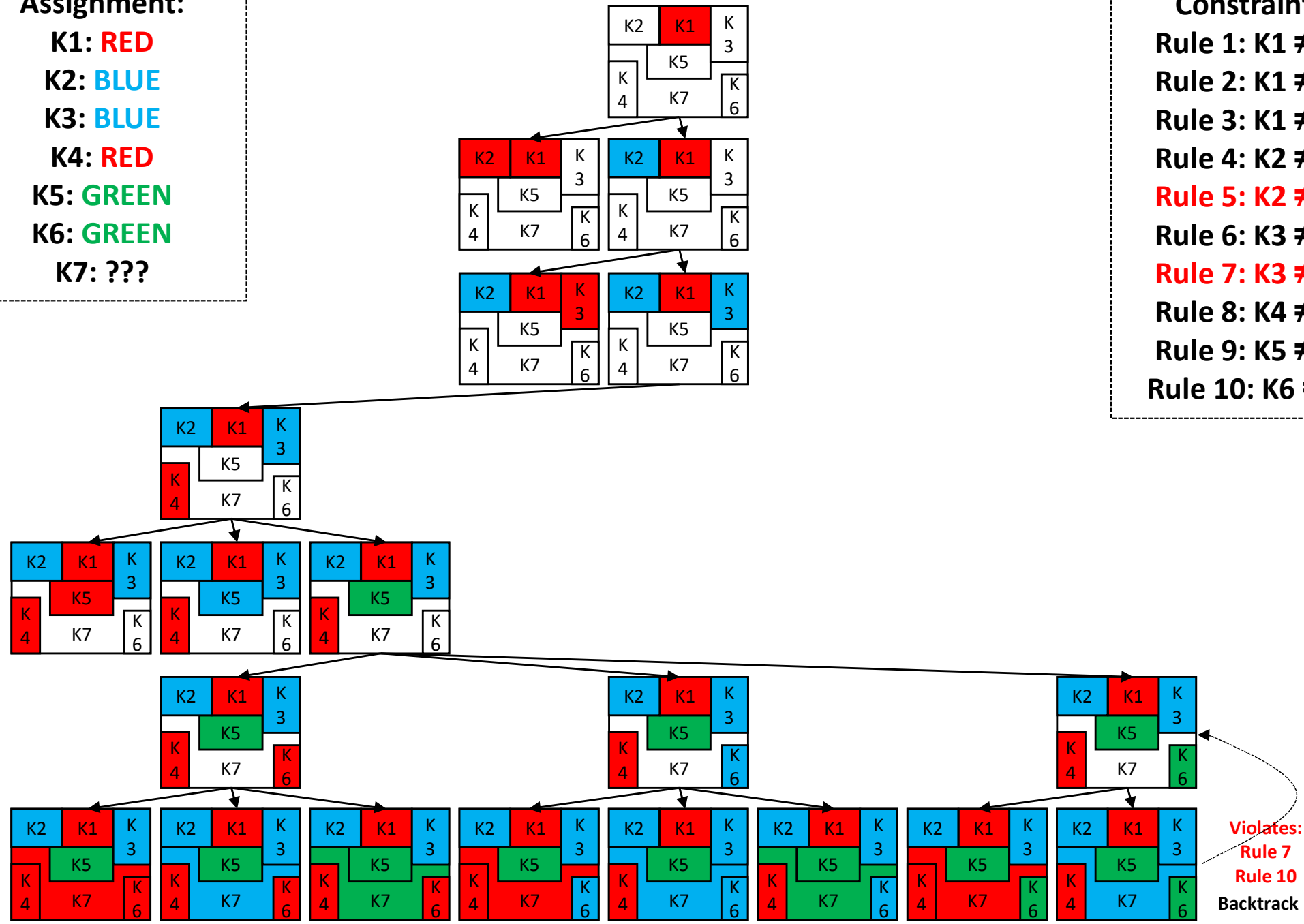
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: GREEN

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

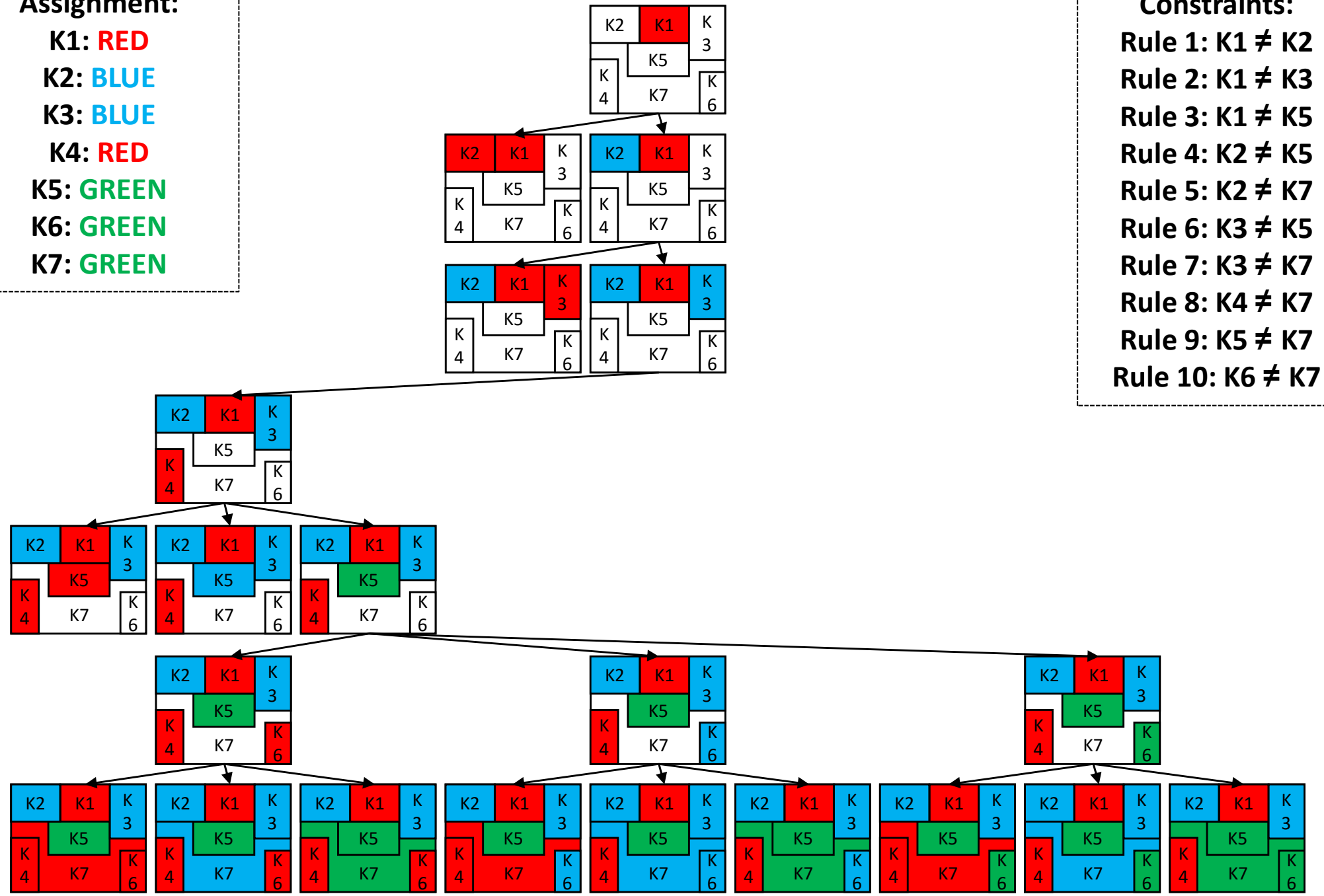
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



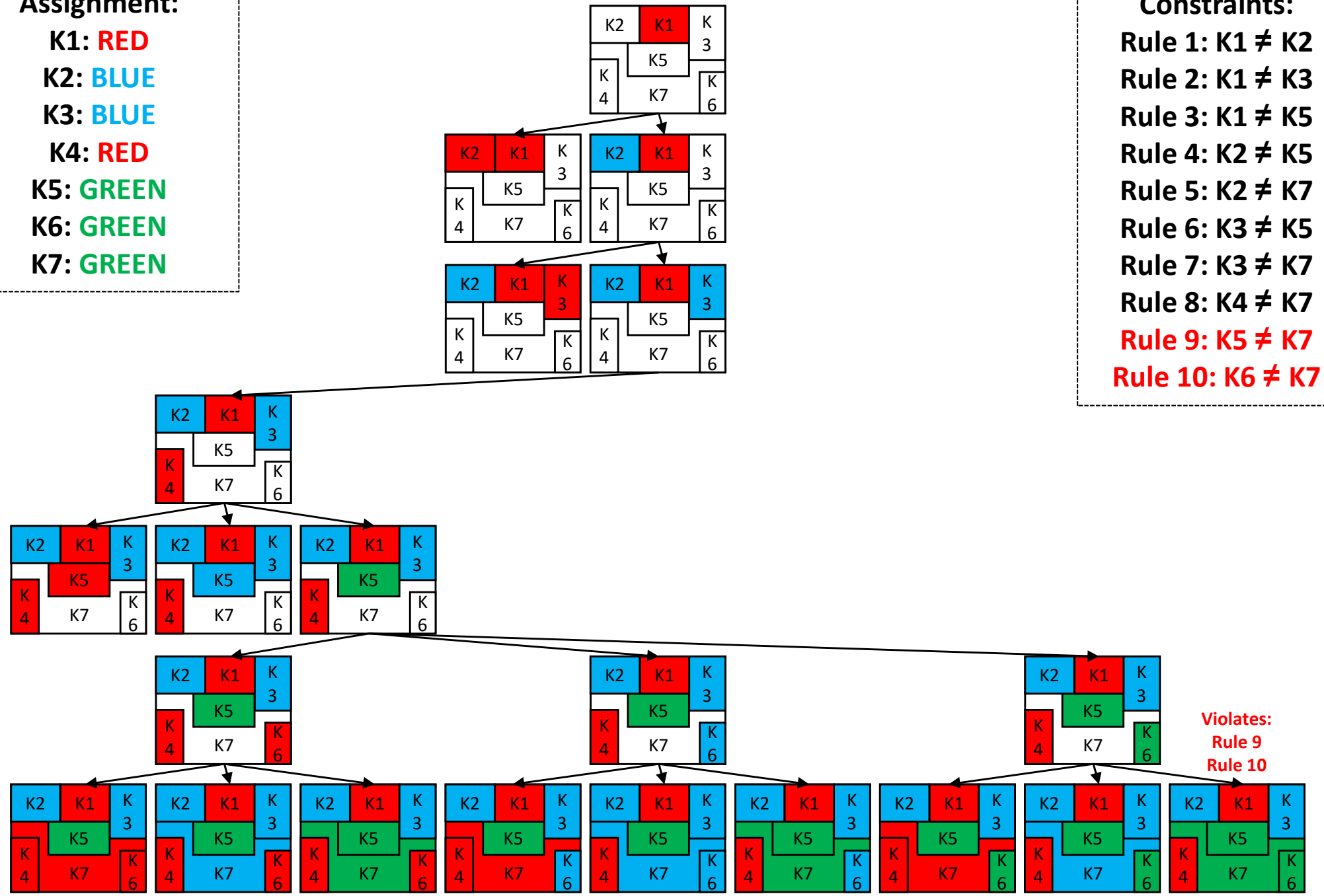
Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED
K2: BLUE
K3: BLUE
K4: RED
K5: GREEN
K6: GREEN
K7: GREEN

Constraints:

Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: GREEN

K6: GREEN

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

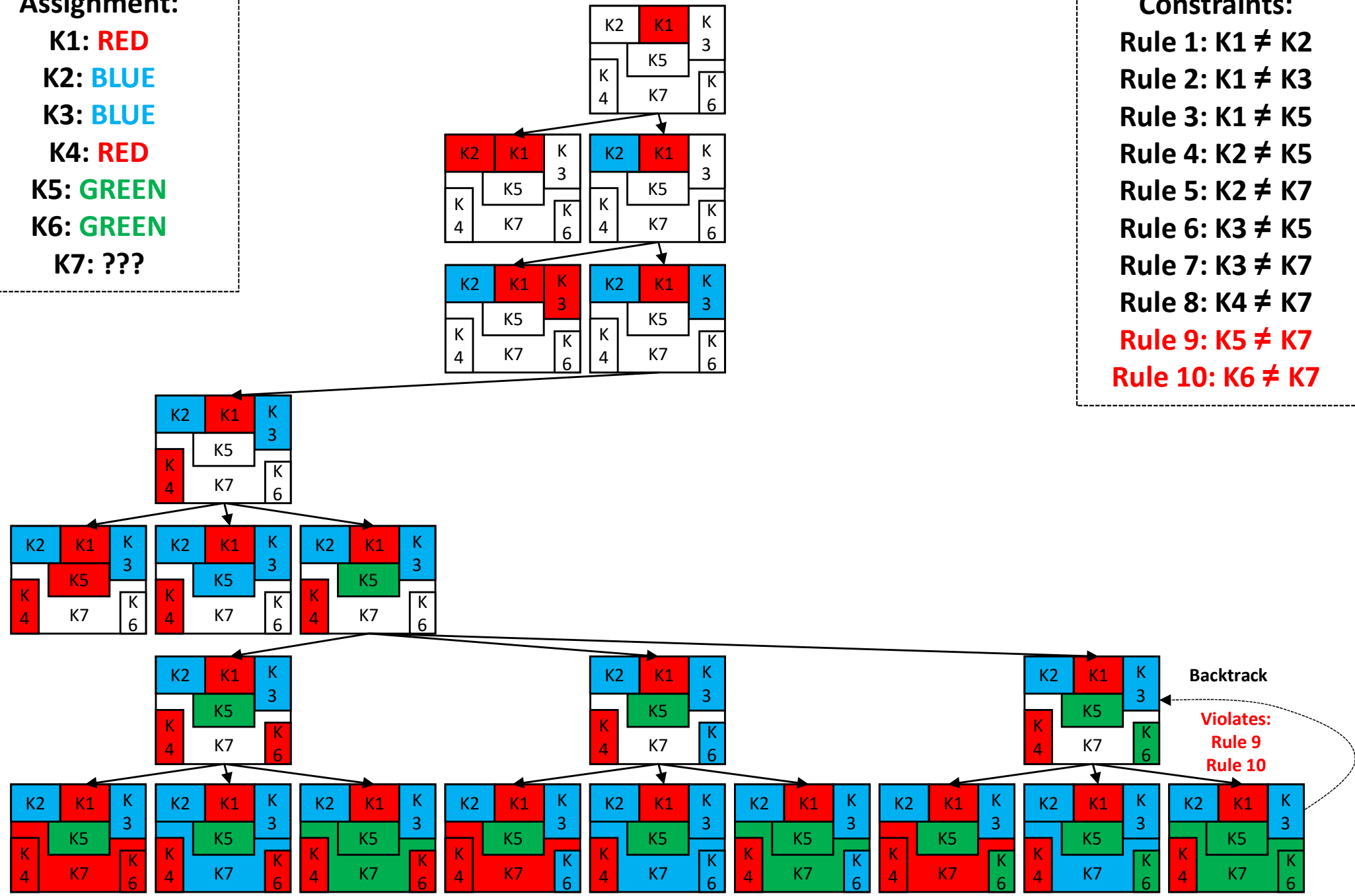
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

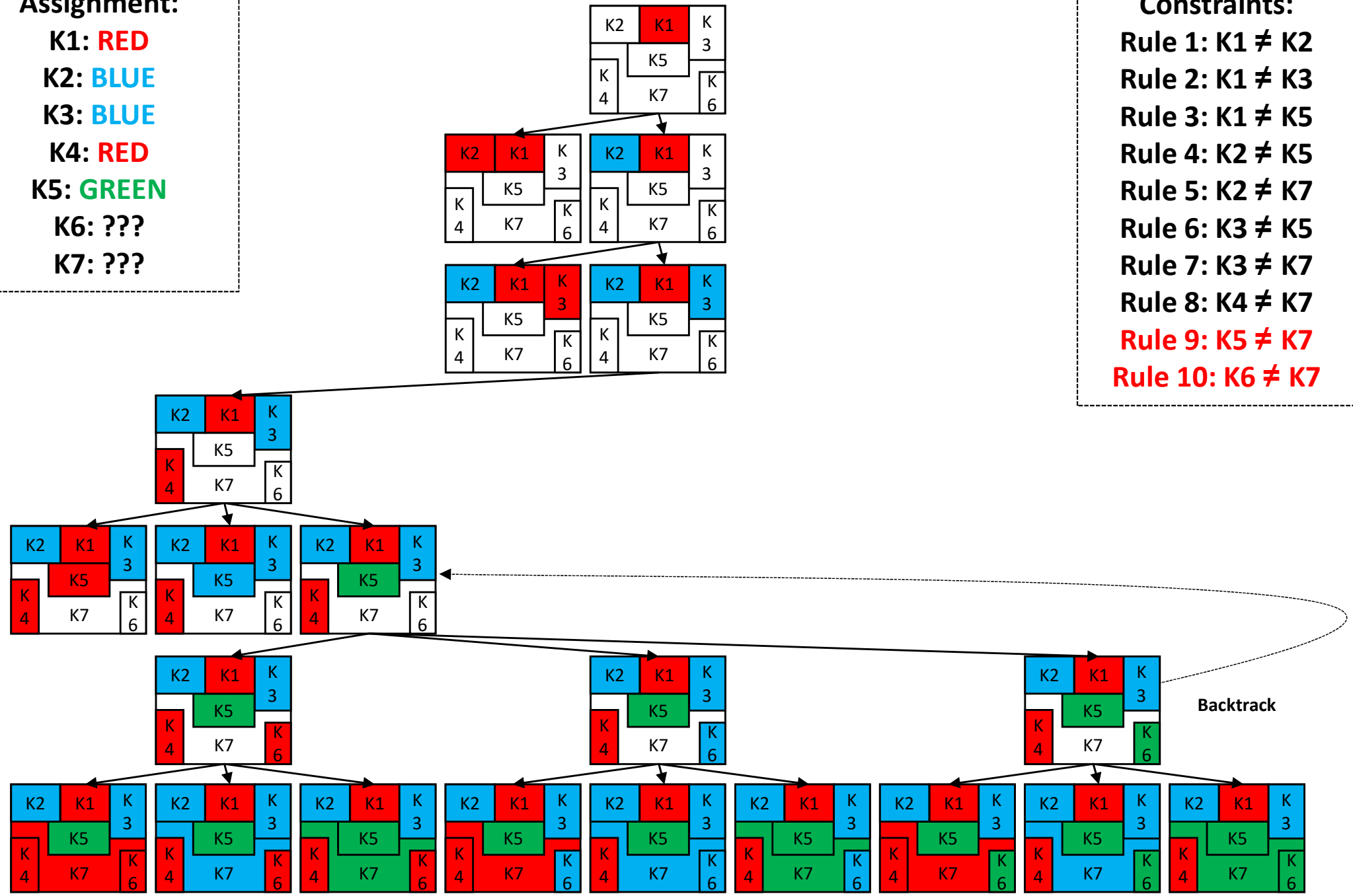
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: RED
K5: GREEN
K6: ???
K7: ???

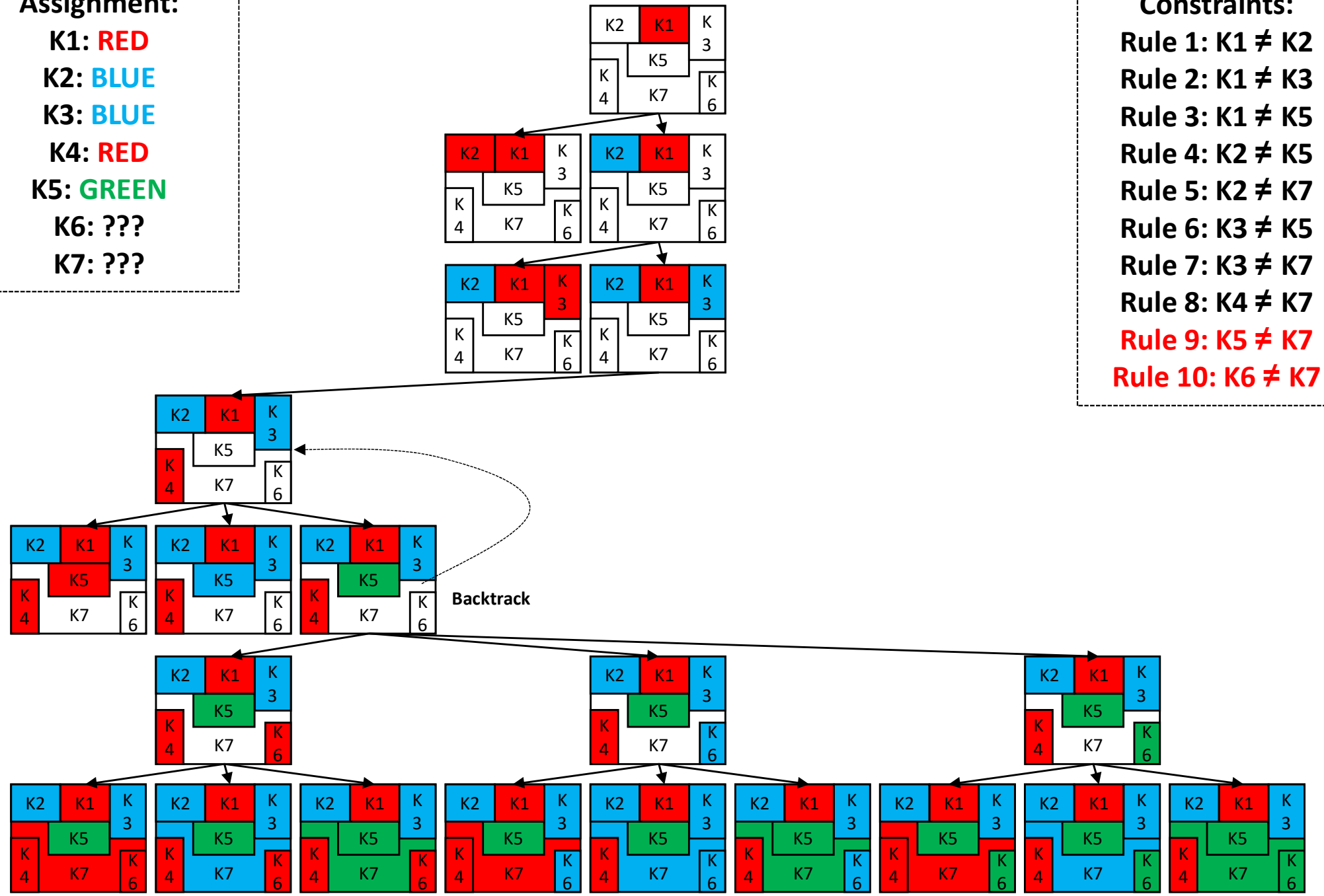
Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: RED
K5: GREEN
K6: ???
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: RED

K5: ???

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

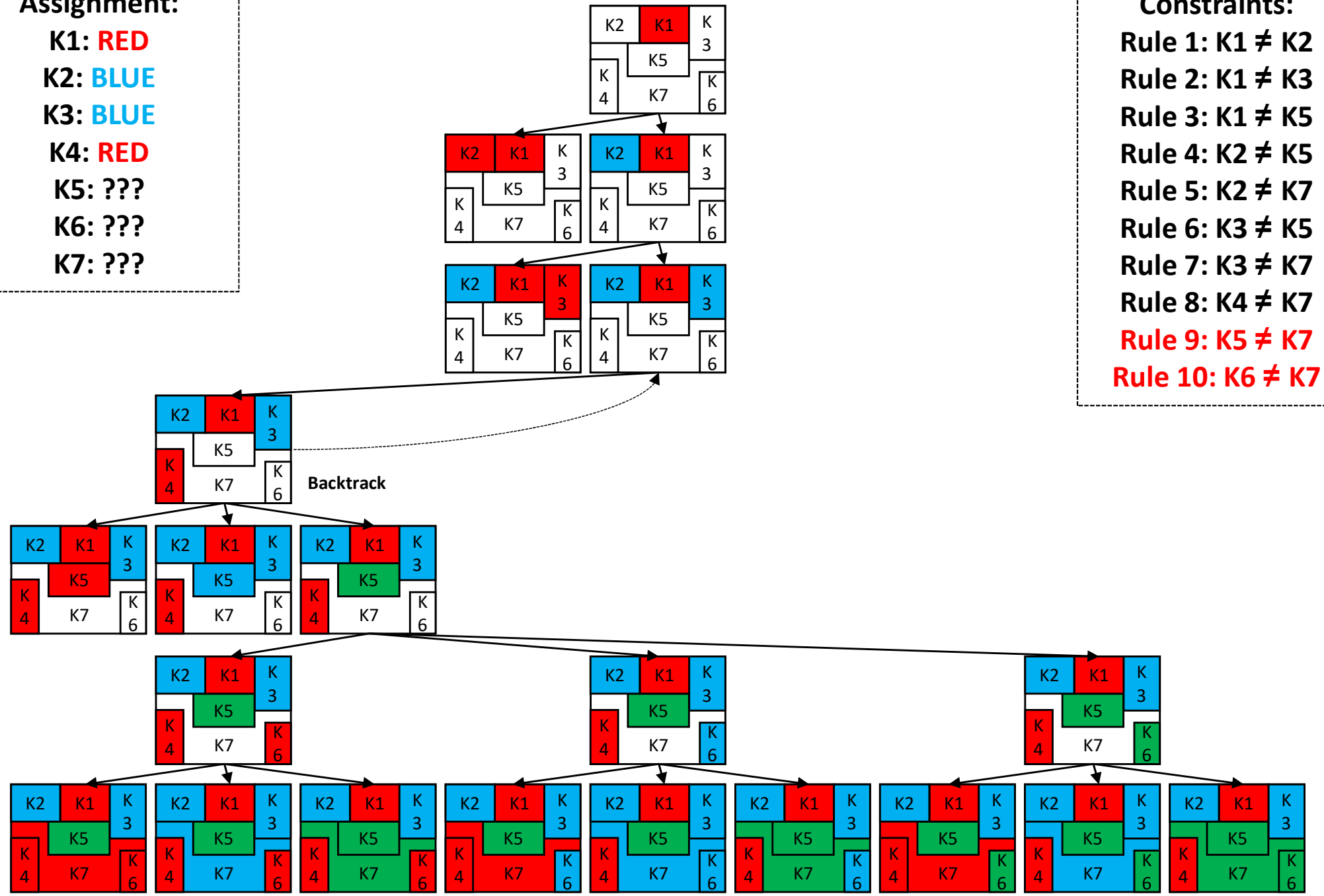
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: ???

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

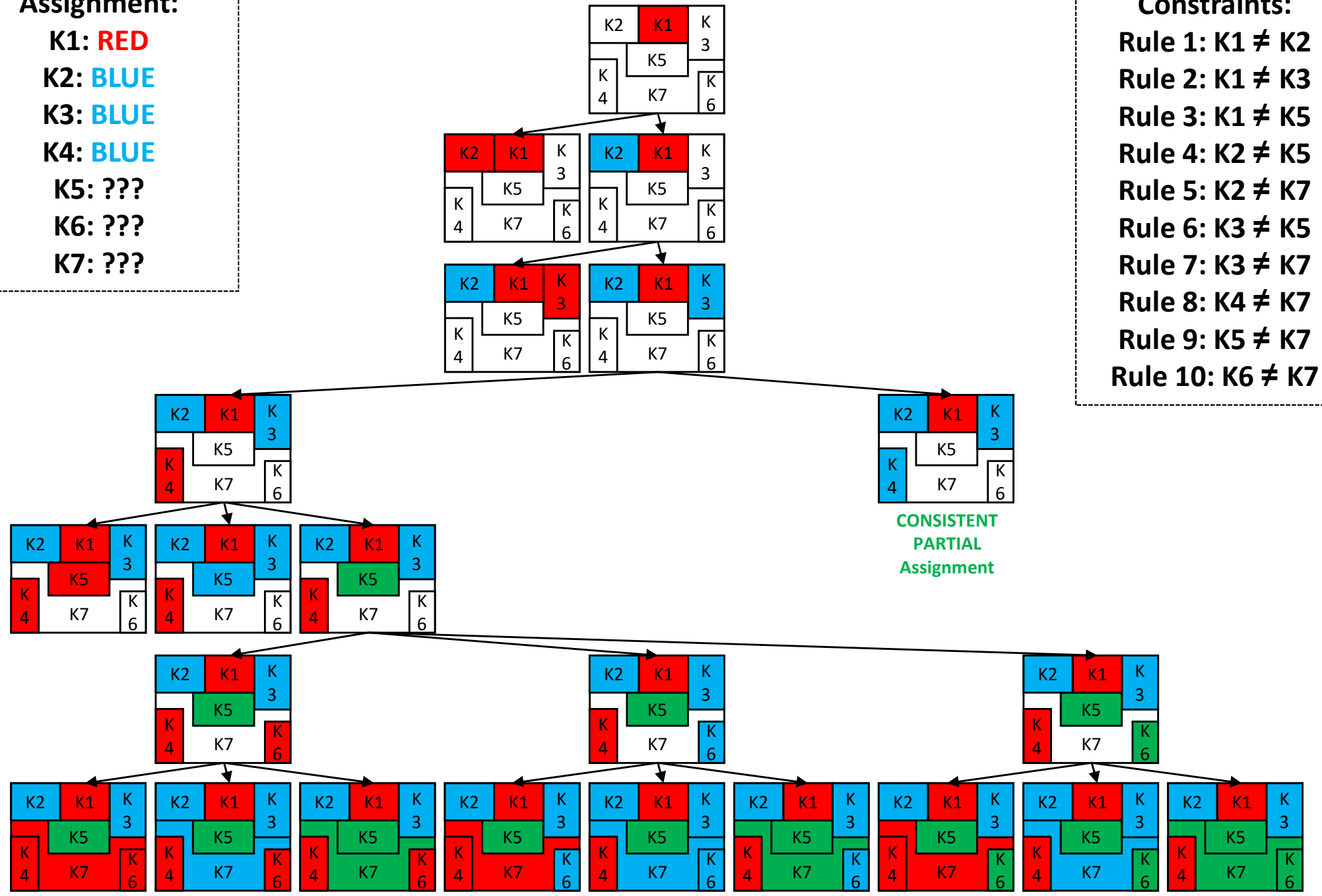
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: RED

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

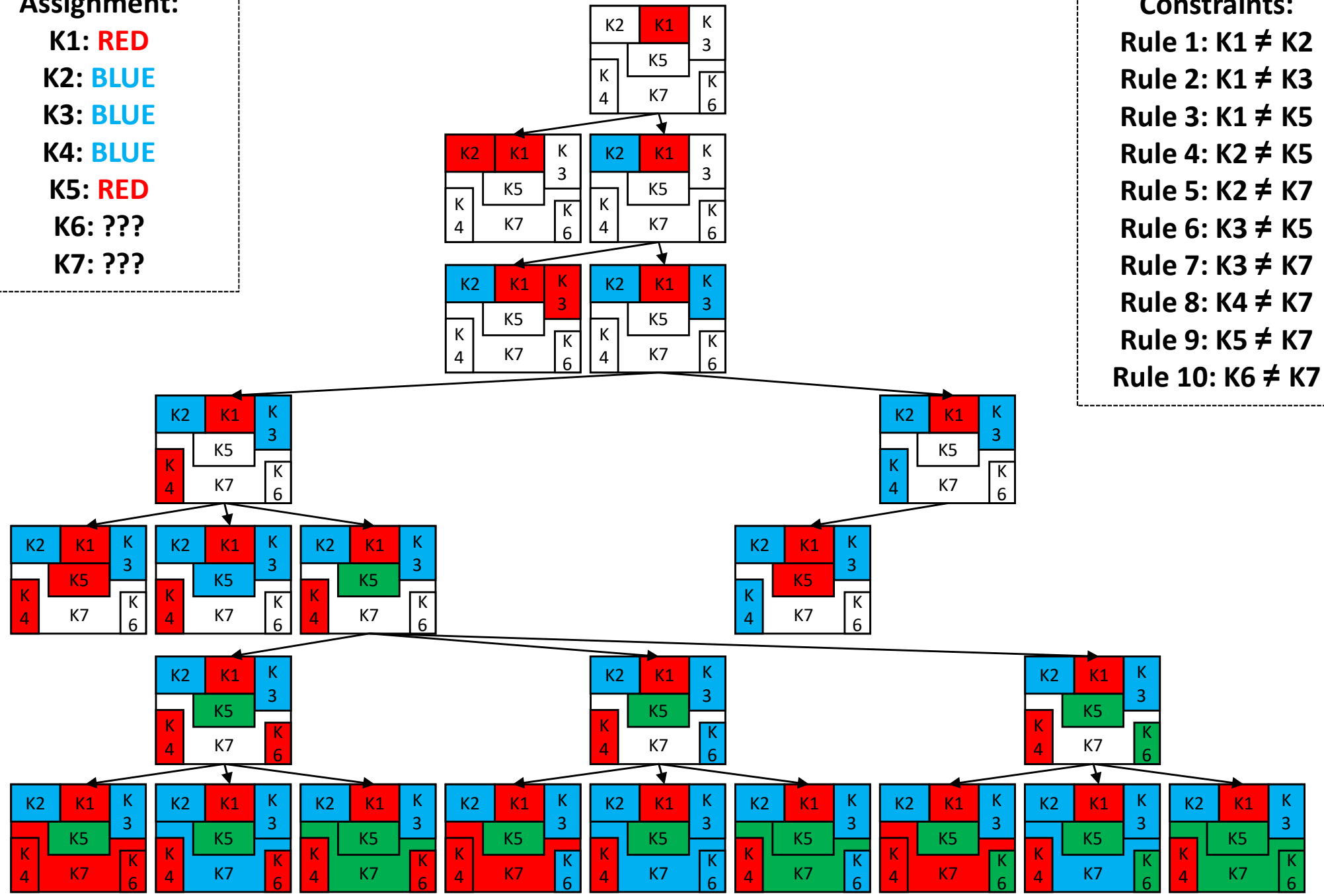
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: RED

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

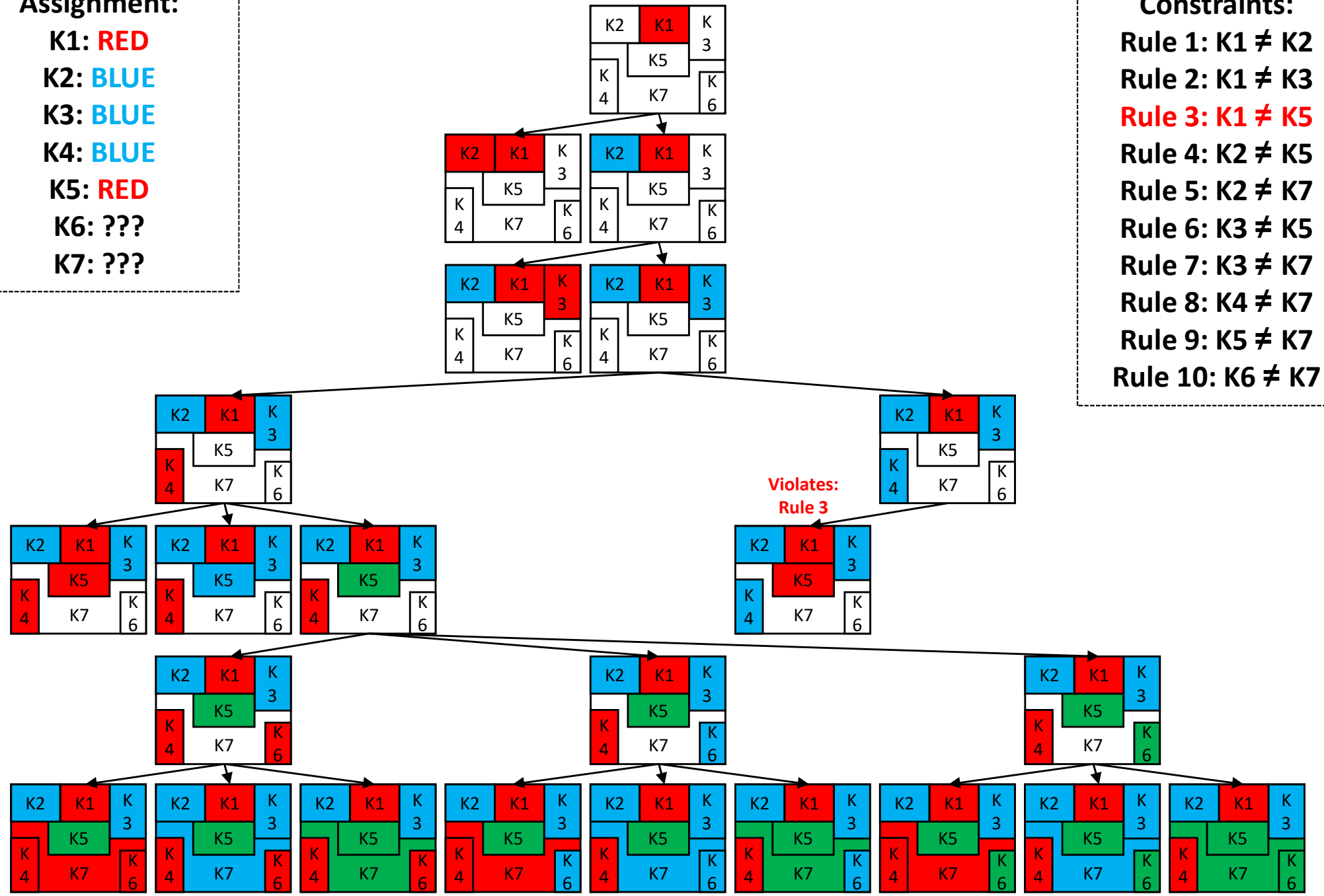
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: ???

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

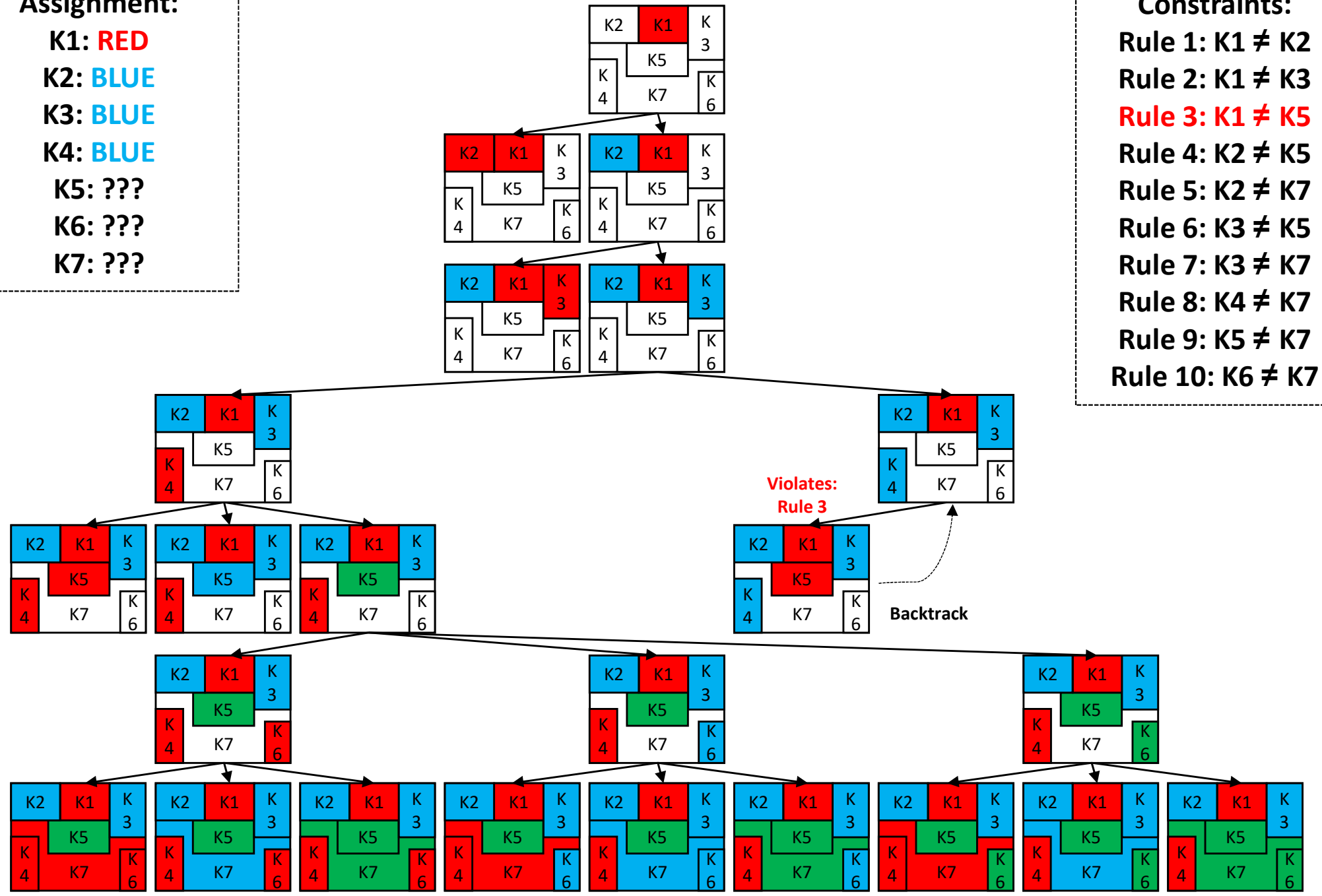
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: BLUE

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

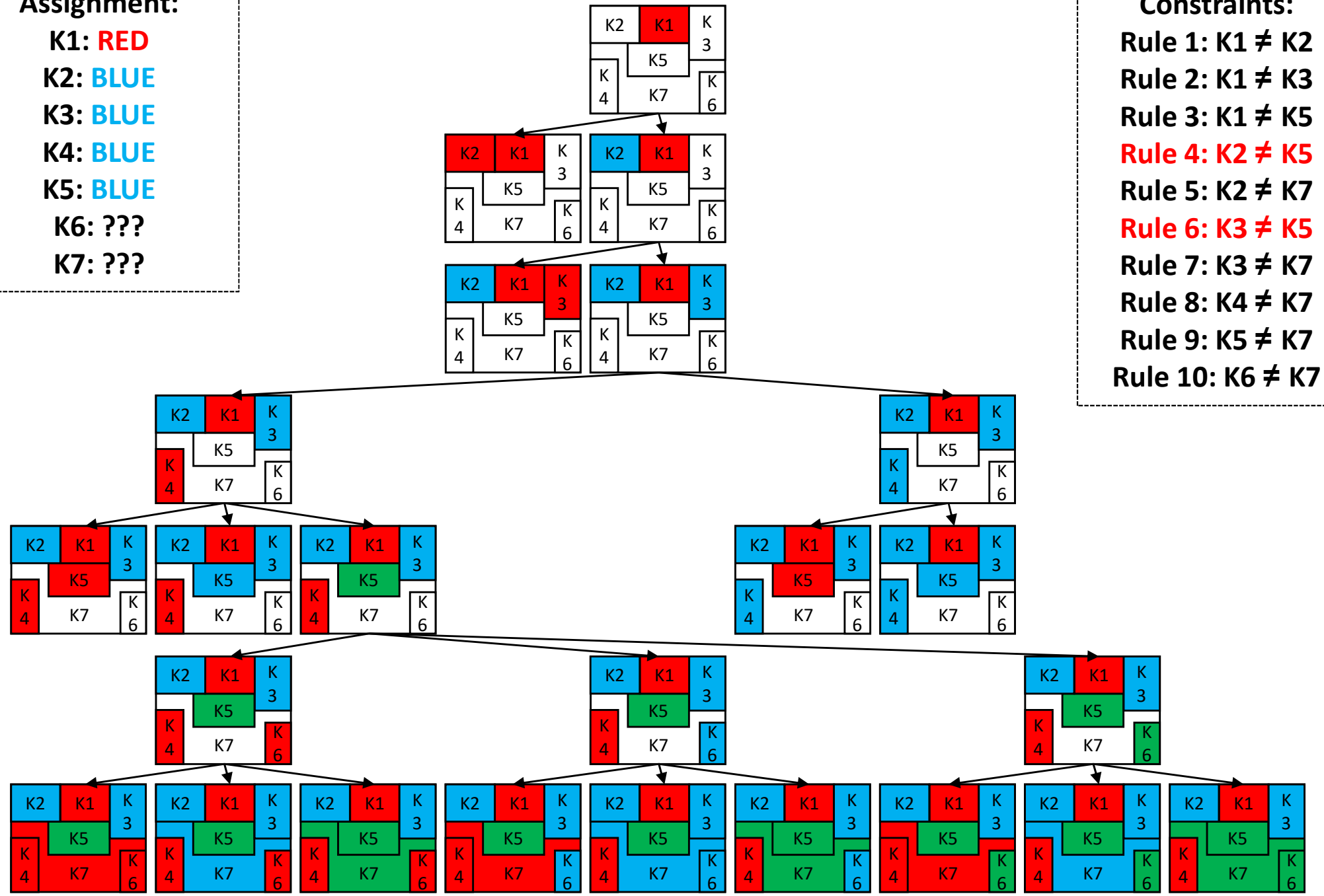
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: BLUE

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

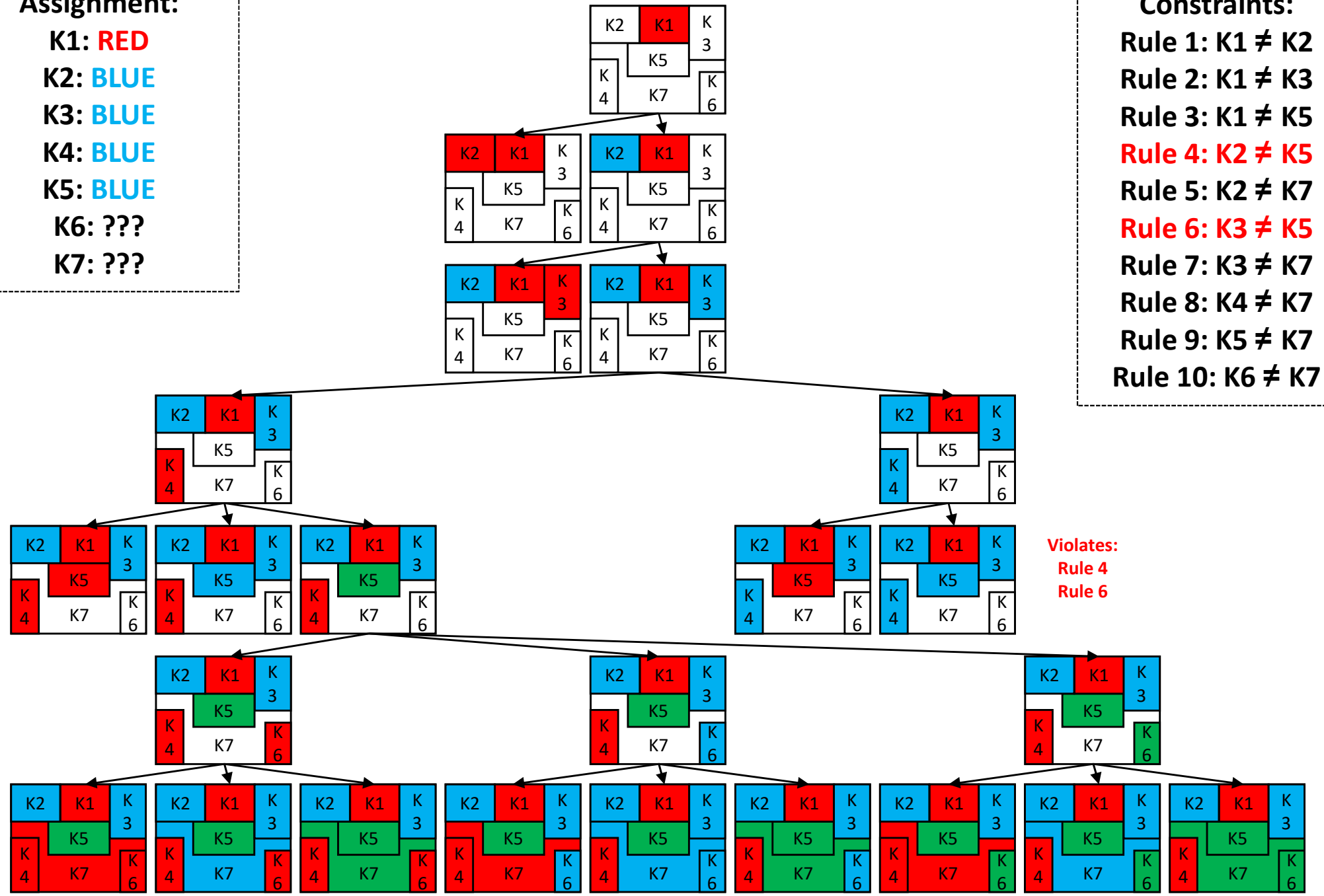
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: ???

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

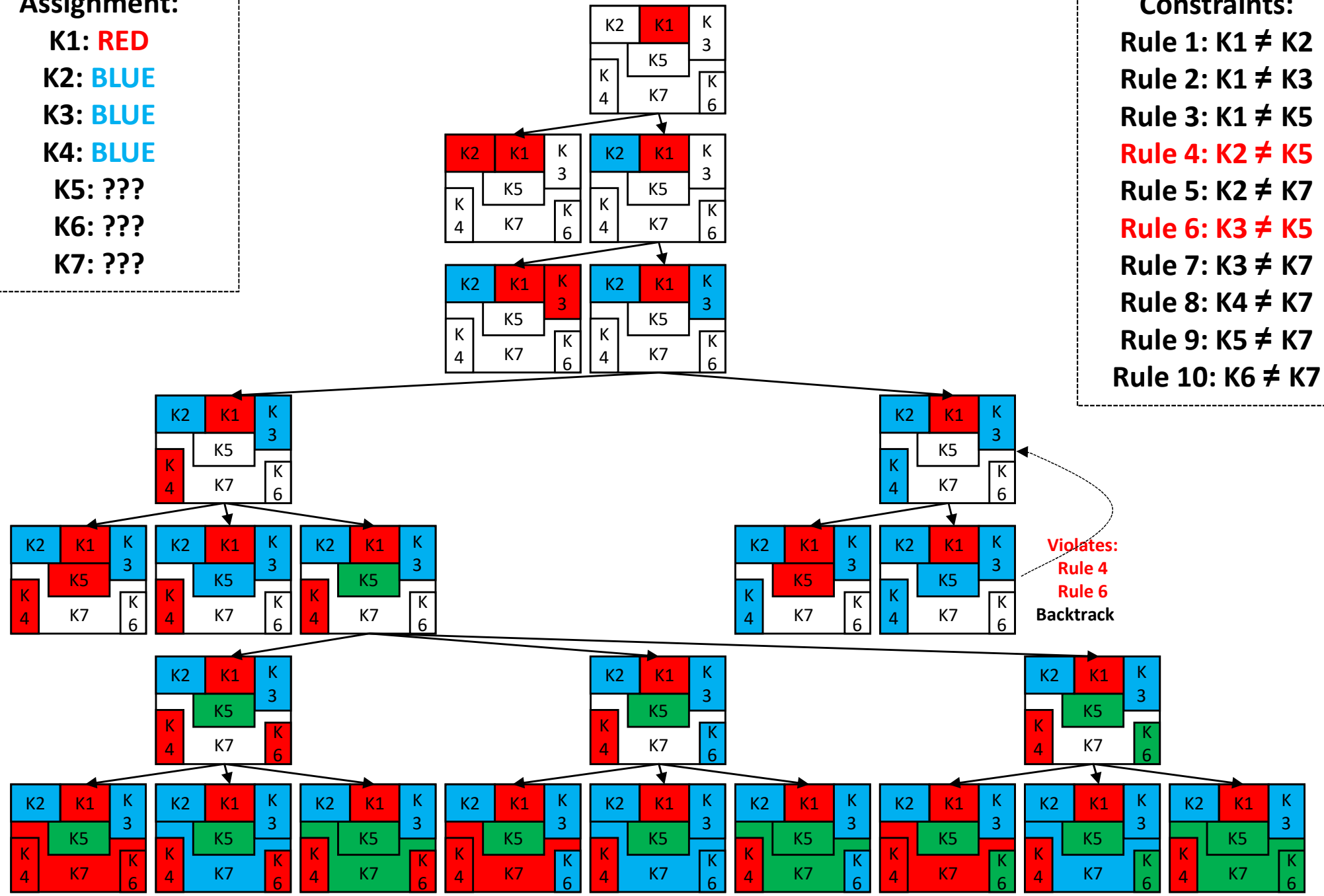
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED

K2: BLUE

K3: BLUE

K4: BLUE

K5: GREEN

K6: ???

K7: ???

Constraints:

Rule 1: K1 ≠ K2

Rule 2: K1 ≠ K3

Rule 3: K1 ≠ K5

Rule 4: K2 ≠ K5

Rule 5: K2 ≠ K7

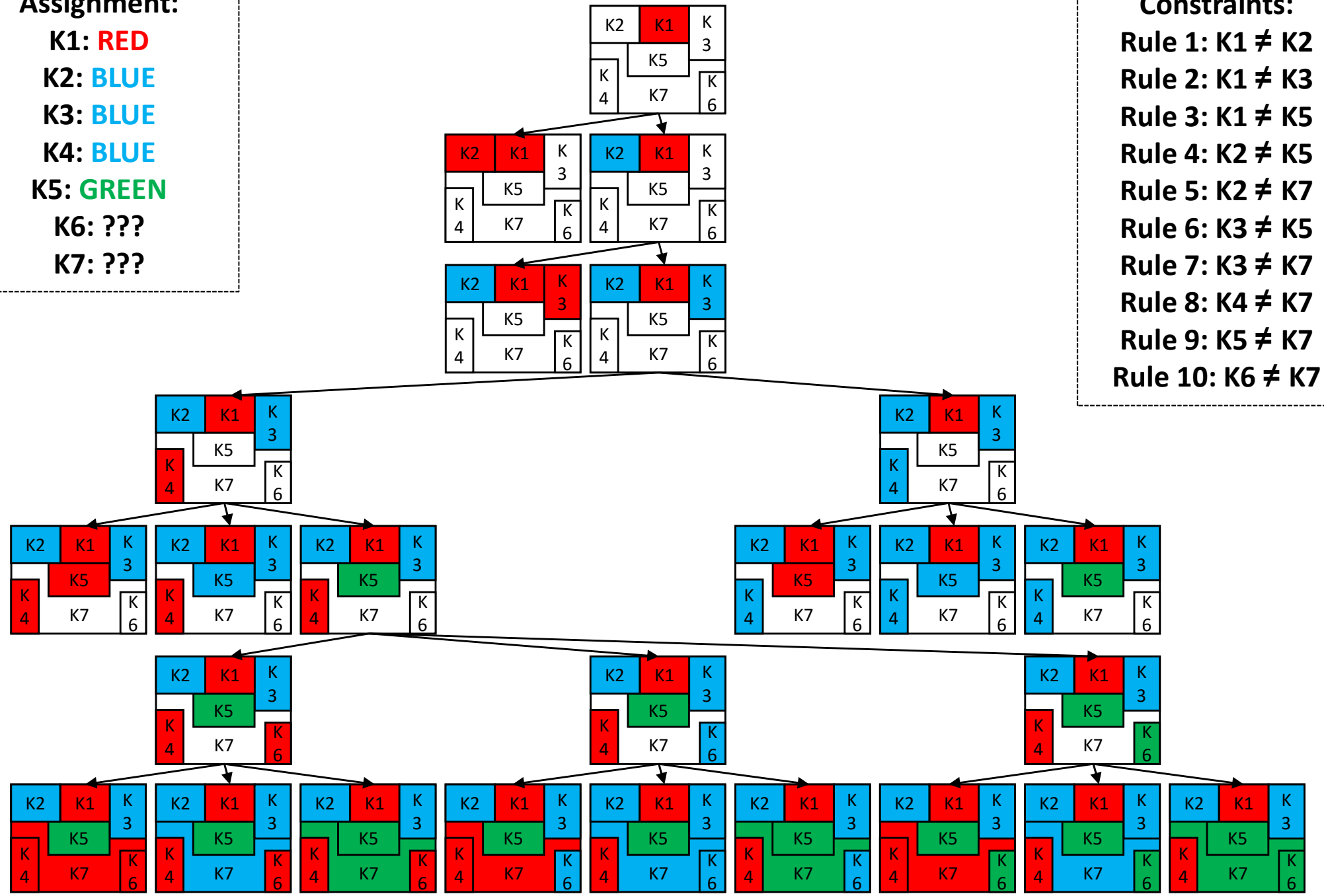
Rule 6: K3 ≠ K5

Rule 7: K3 ≠ K7

Rule 8: K4 ≠ K7

Rule 9: K5 ≠ K7

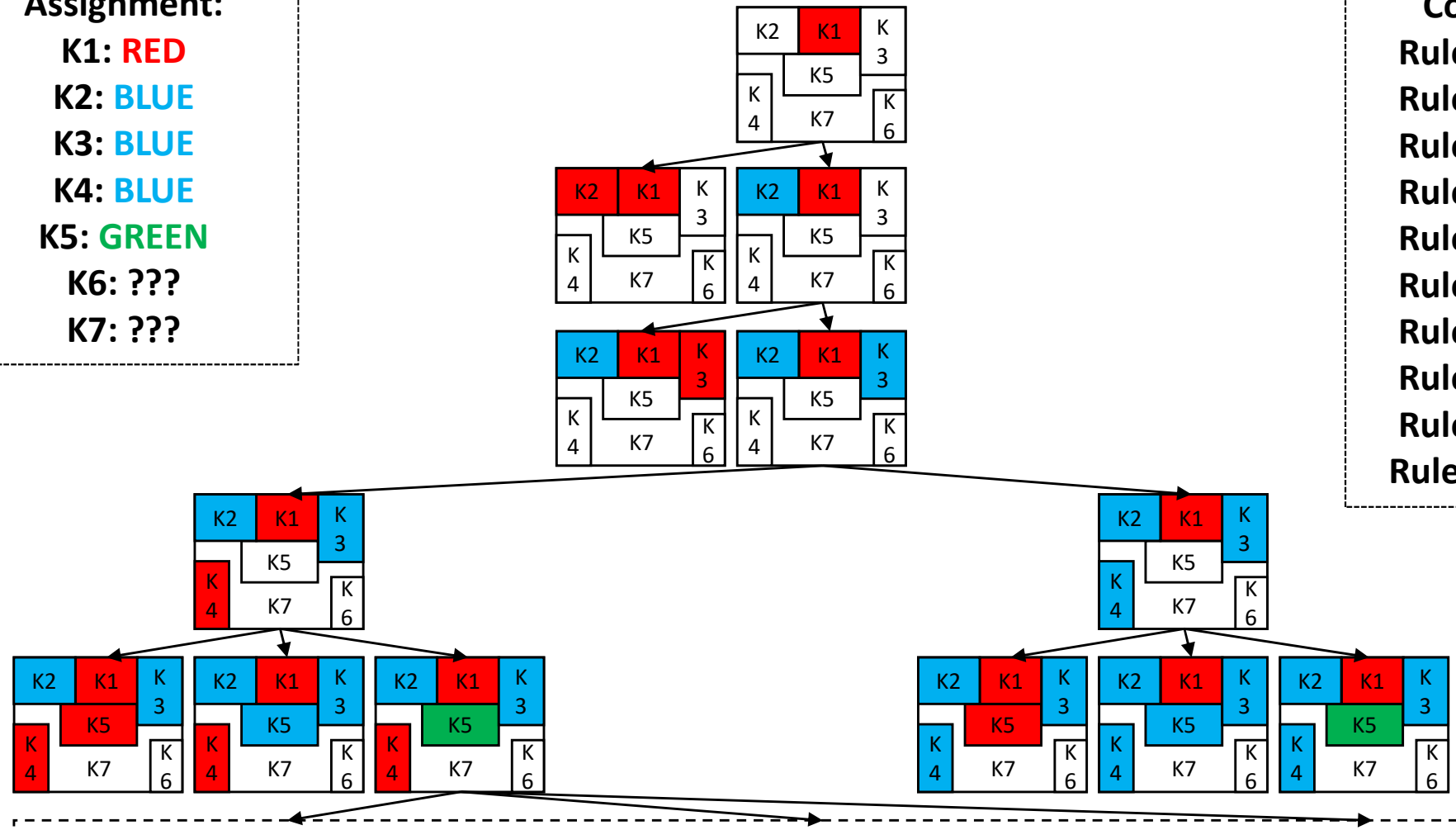
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: ???
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

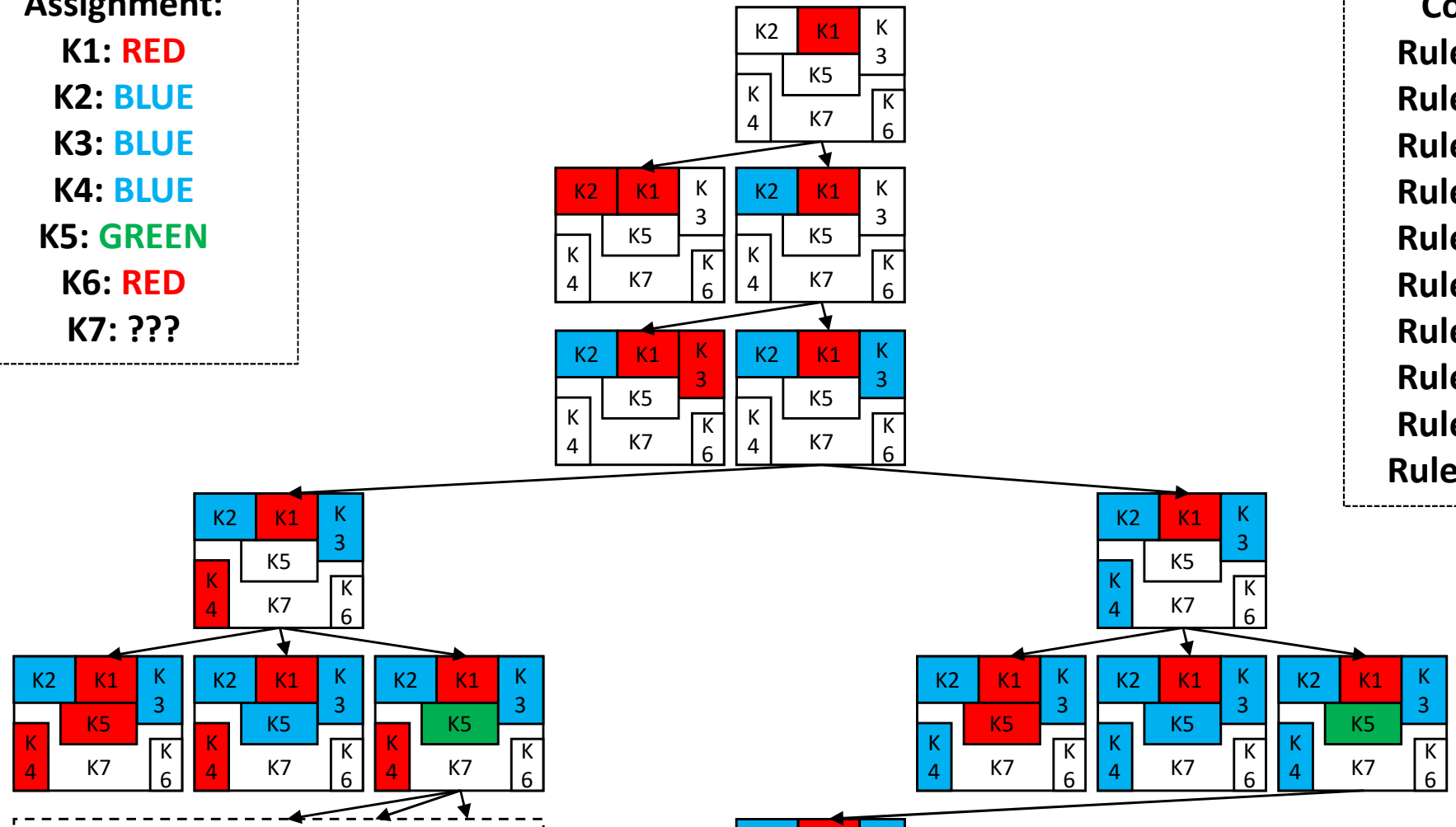


Visited / dead ends
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

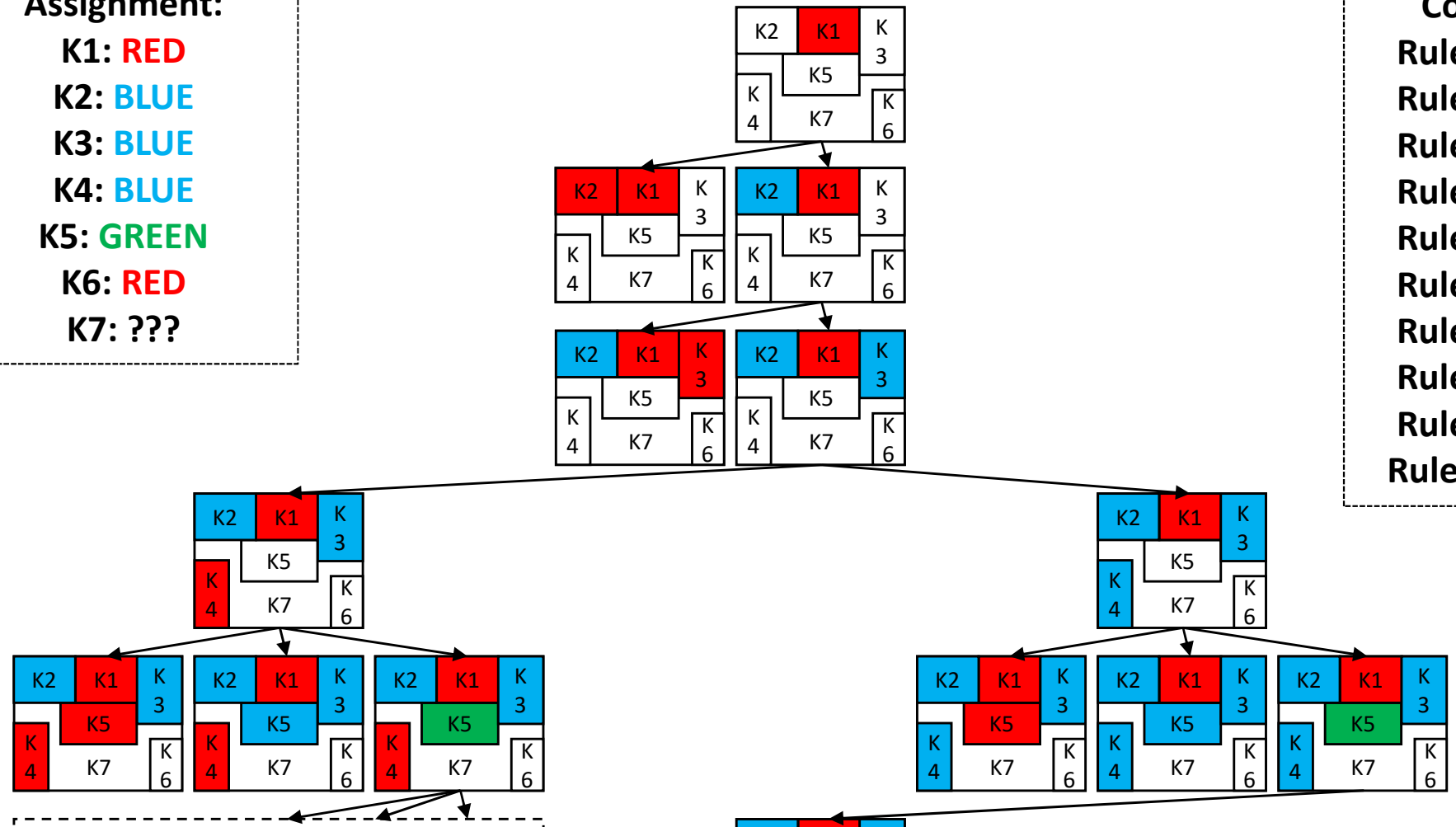


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Visited / dead ends:
Complete, but inconsistent assignments

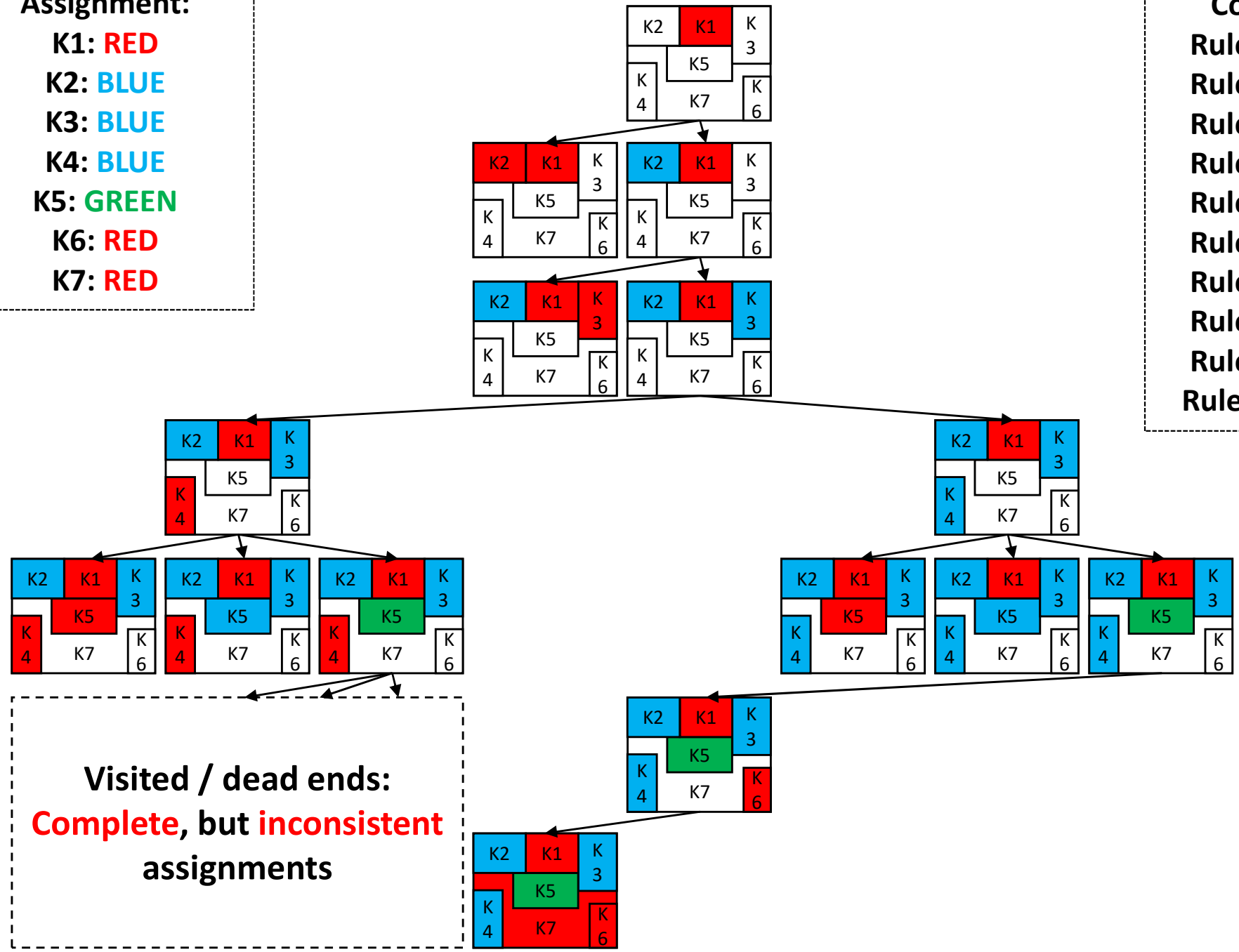
Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:

K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: RED

Constraints:

Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

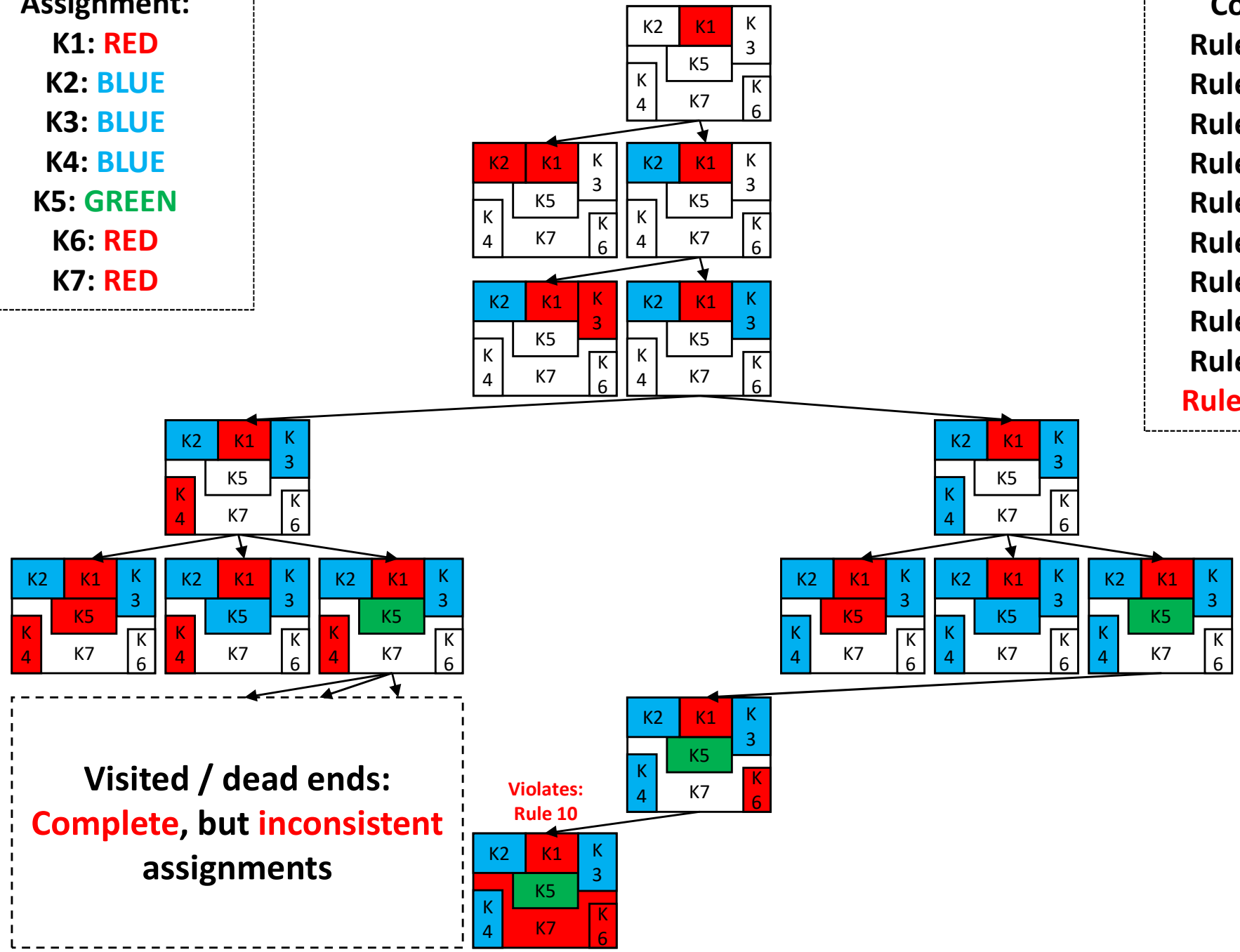


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: RED

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

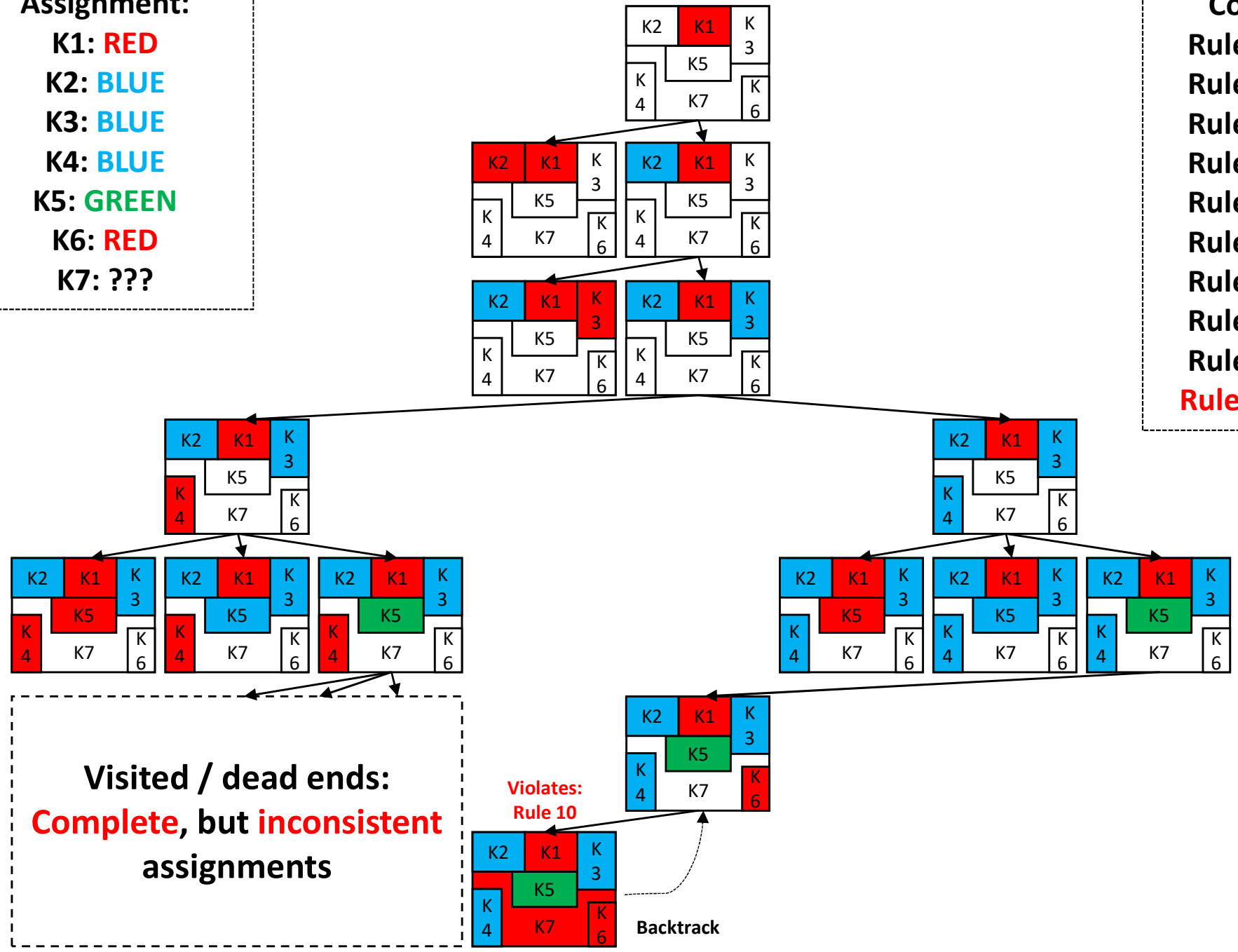


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Visited / dead ends:
Complete, but inconsistent assignments

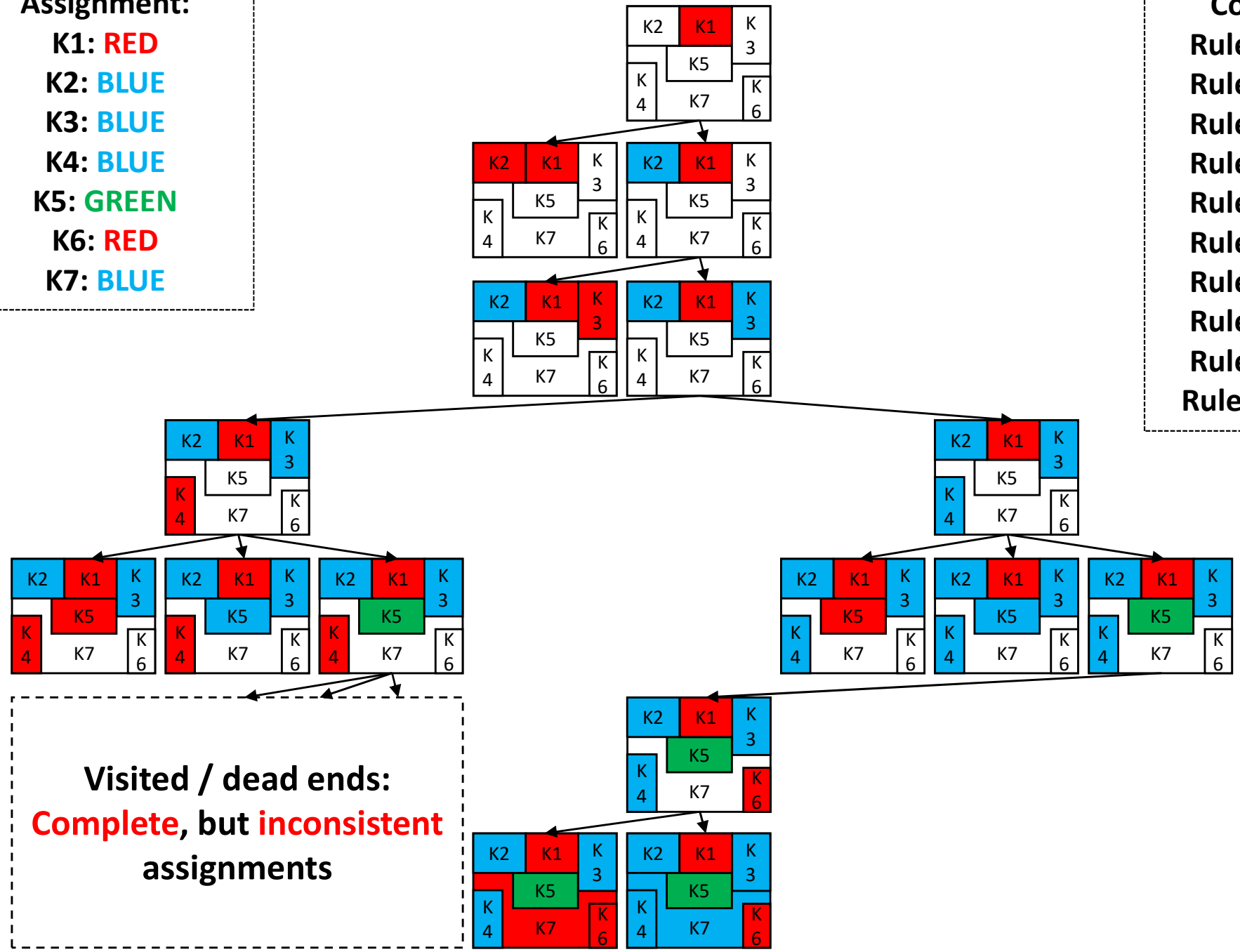
Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: BLUE

Constraints:

Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

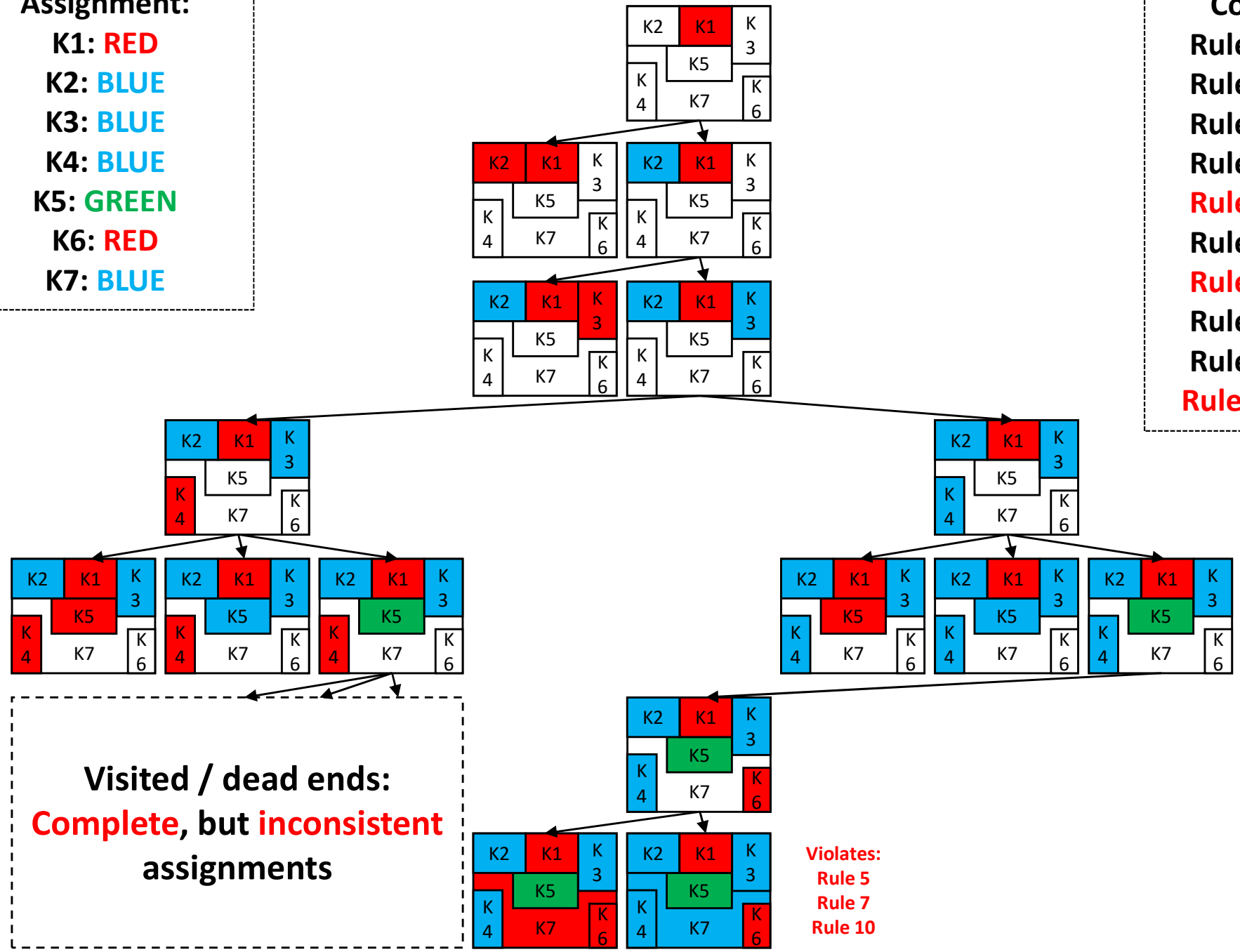


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: BLUE

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



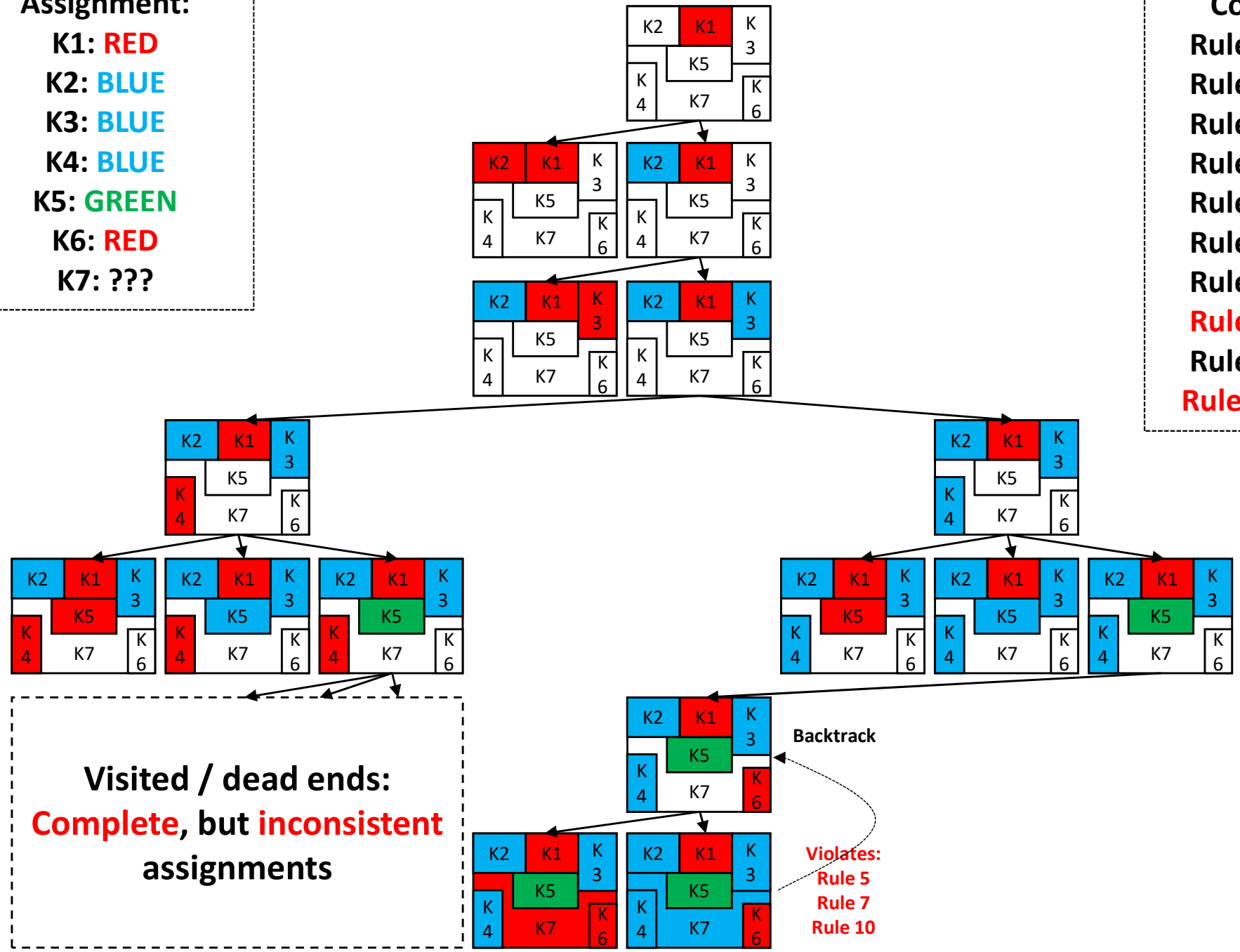
Visited / dead ends:
Complete, but inconsistent assignments

Violates:
Rule 5
Rule 7
Rule 10

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

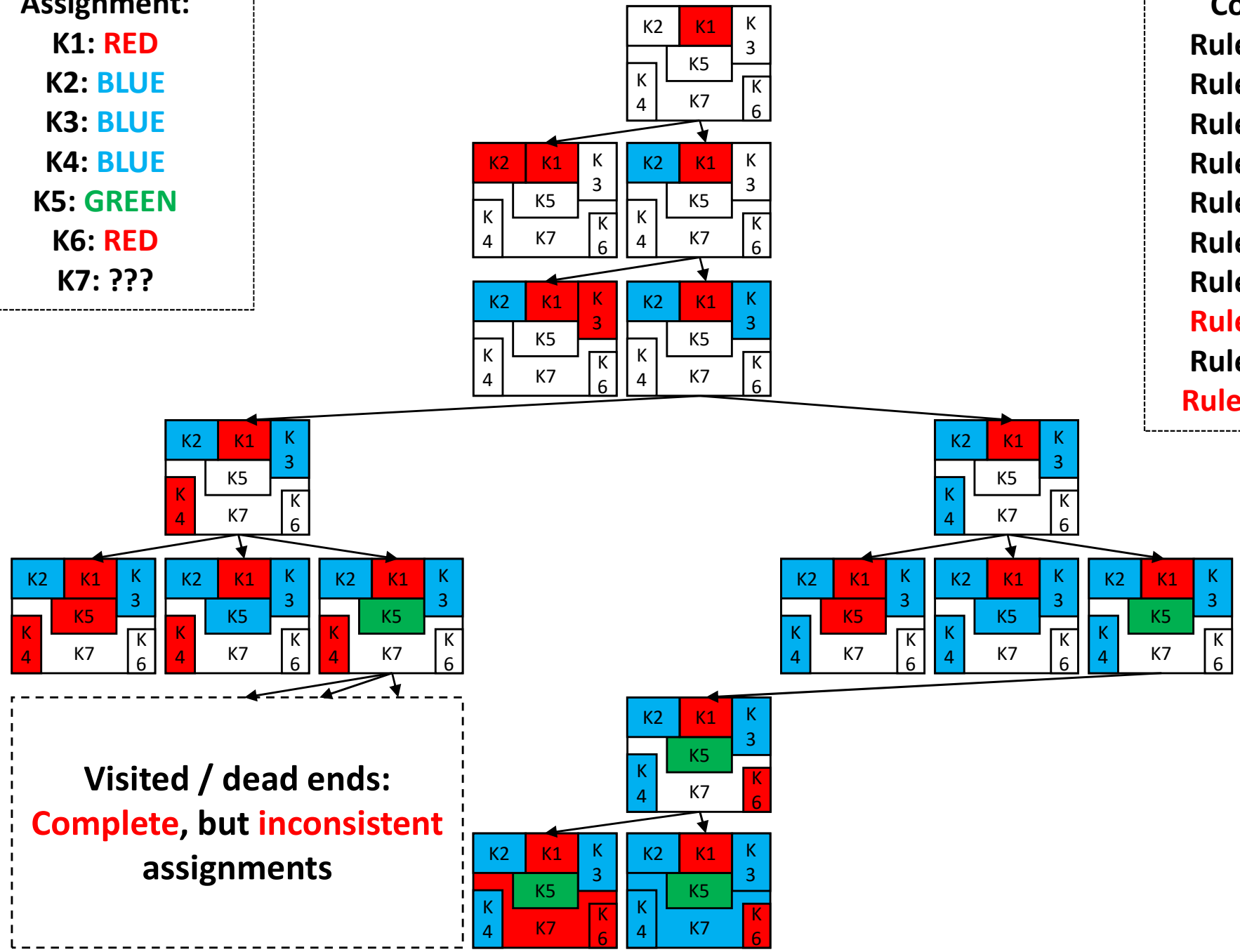


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

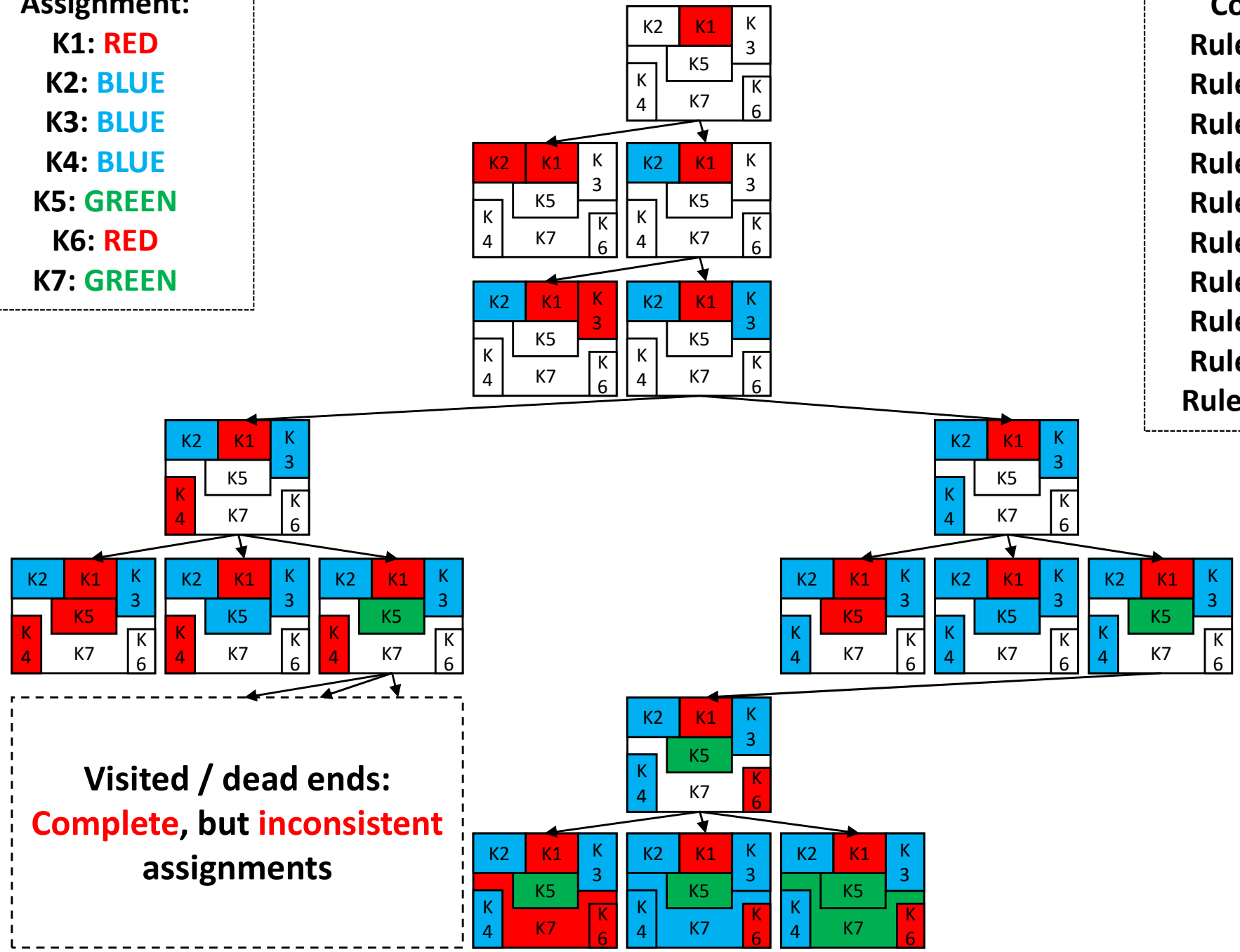


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: GREEN

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Visited / dead ends:
Complete, but inconsistent assignments

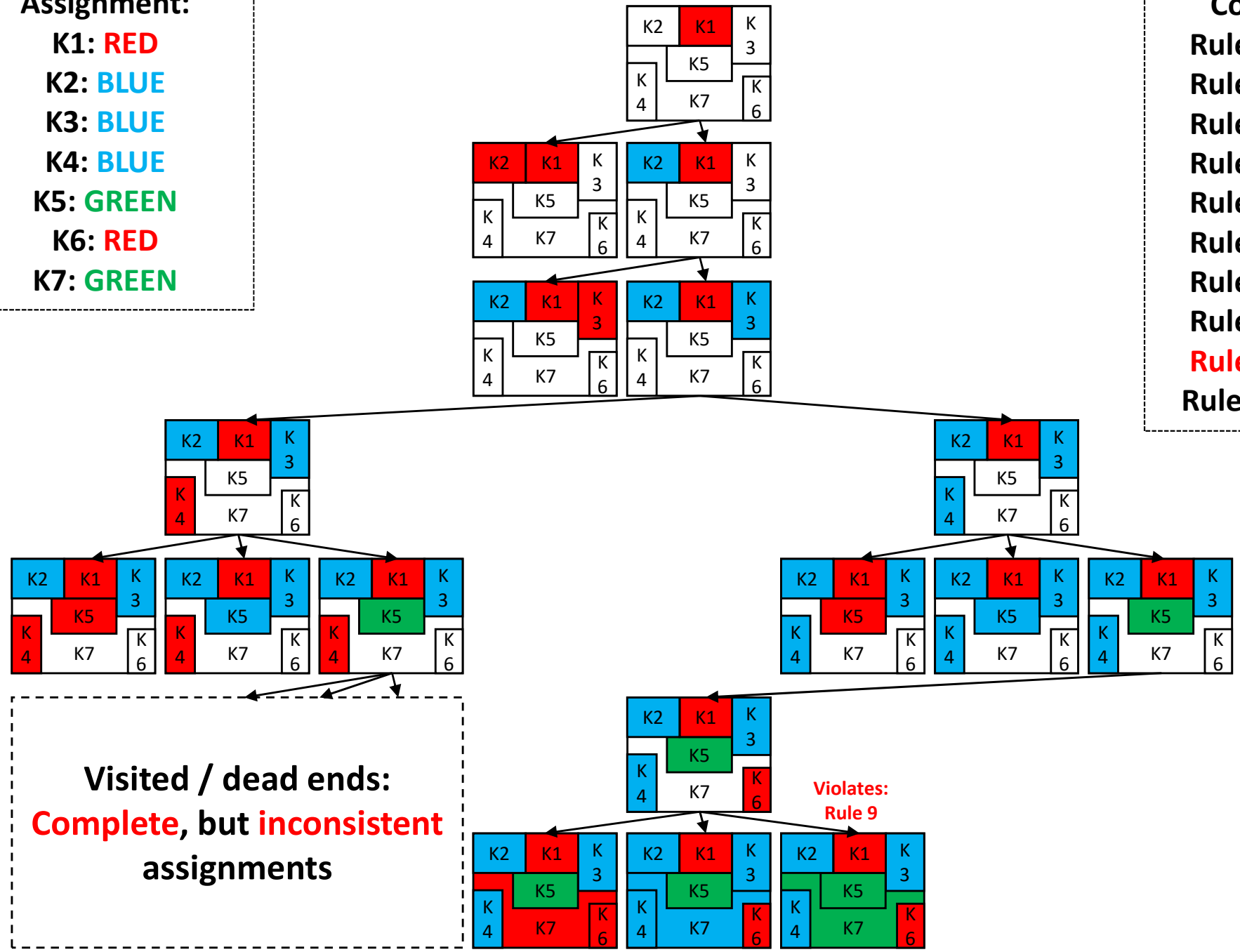
Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:

K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: GREEN

Constraints:

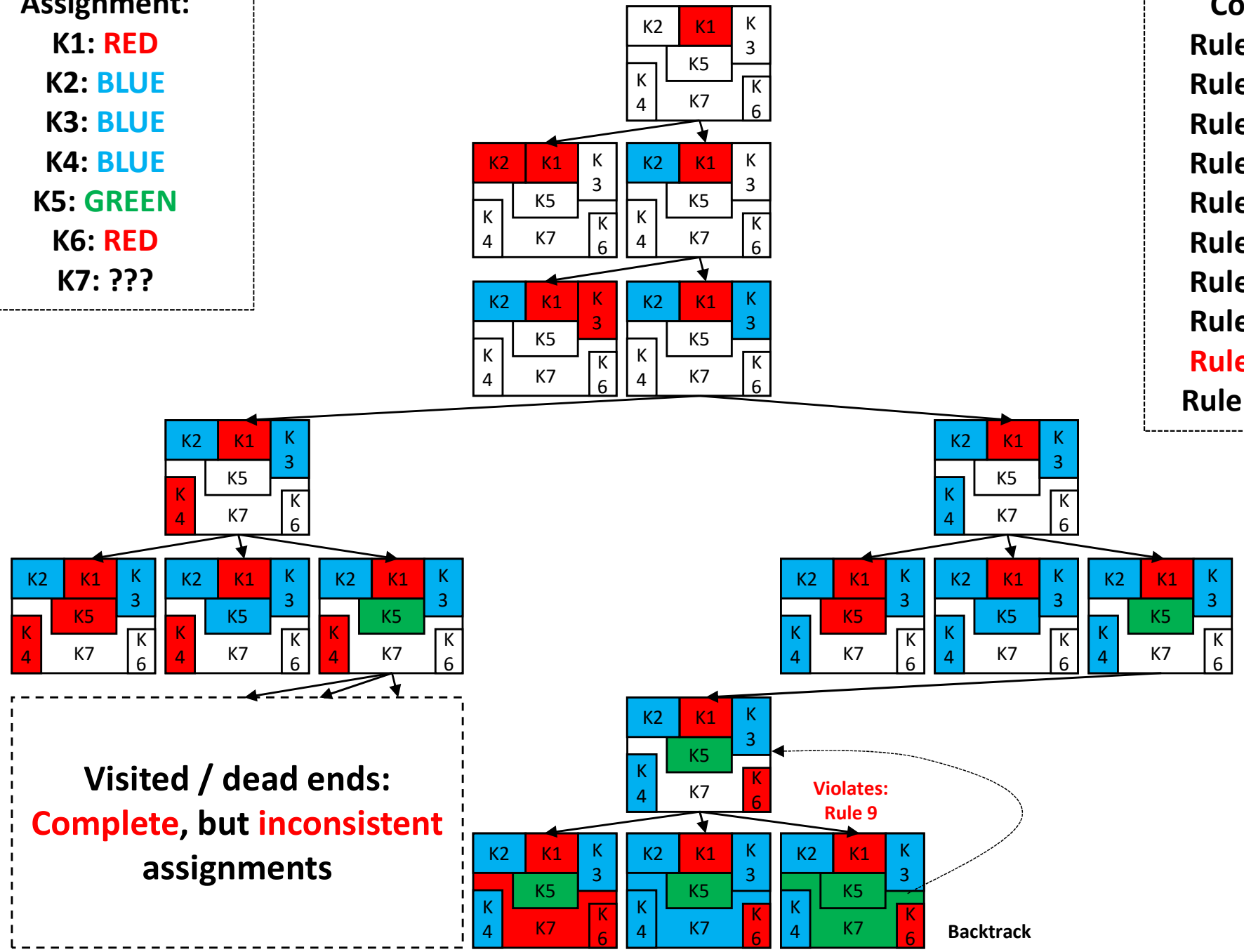
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Visited / dead ends:
Complete, but inconsistent assignments

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: RED
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

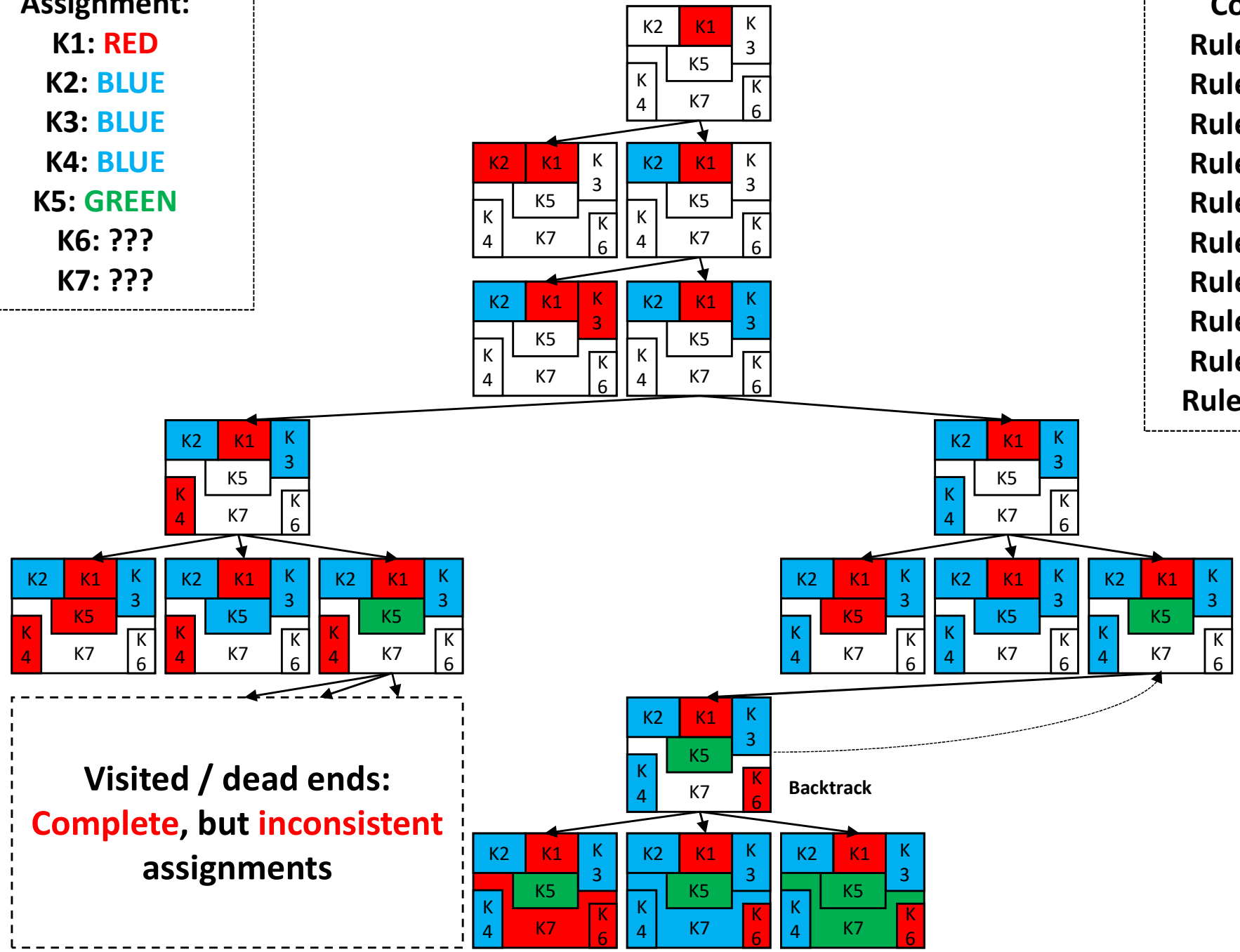


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: ???
K7: ???

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7

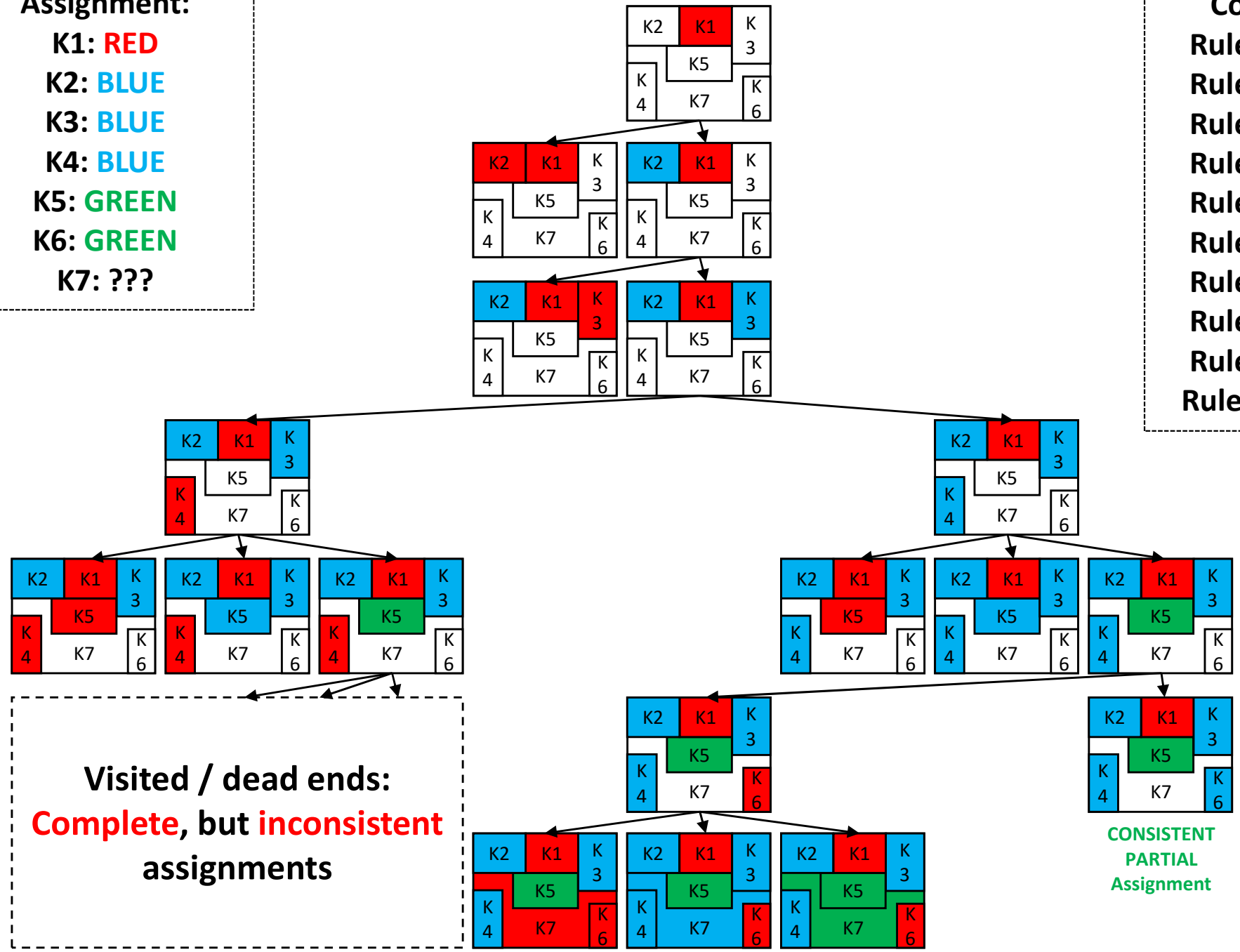


Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order: RED, BLUE, GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: GREEN
K7: ???

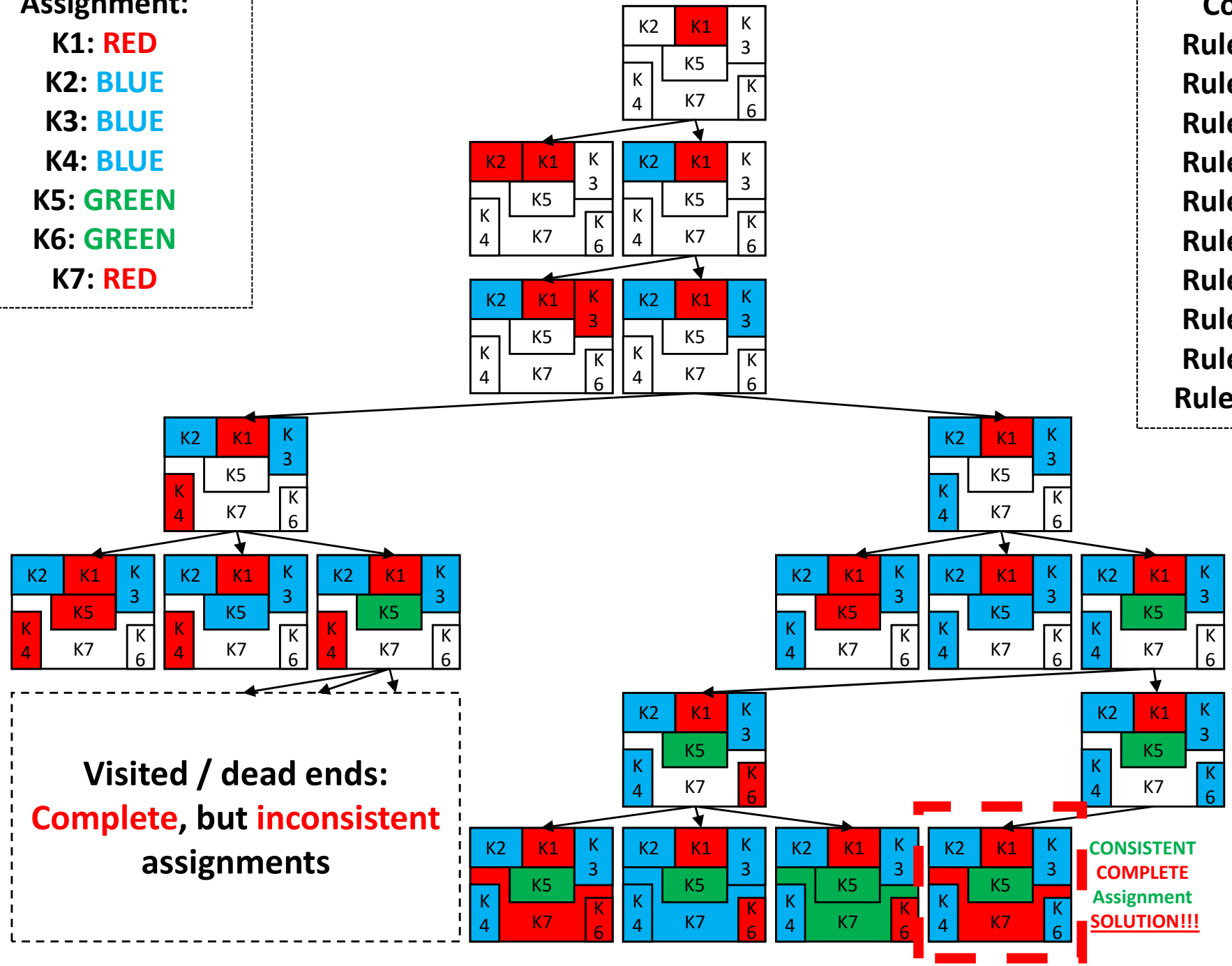
Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

Assignment:
K1: RED
K2: BLUE
K3: BLUE
K4: BLUE
K5: GREEN
K6: GREEN
K7: RED

Constraints:
Rule 1: K1 ≠ K2
Rule 2: K1 ≠ K3
Rule 3: K1 ≠ K5
Rule 4: K2 ≠ K5
Rule 5: K2 ≠ K7
Rule 6: K3 ≠ K5
Rule 7: K3 ≠ K7
Rule 8: K4 ≠ K7
Rule 9: K5 ≠ K7
Rule 10: K6 ≠ K7



Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** **RED**, **BLUE**, **GREEN**

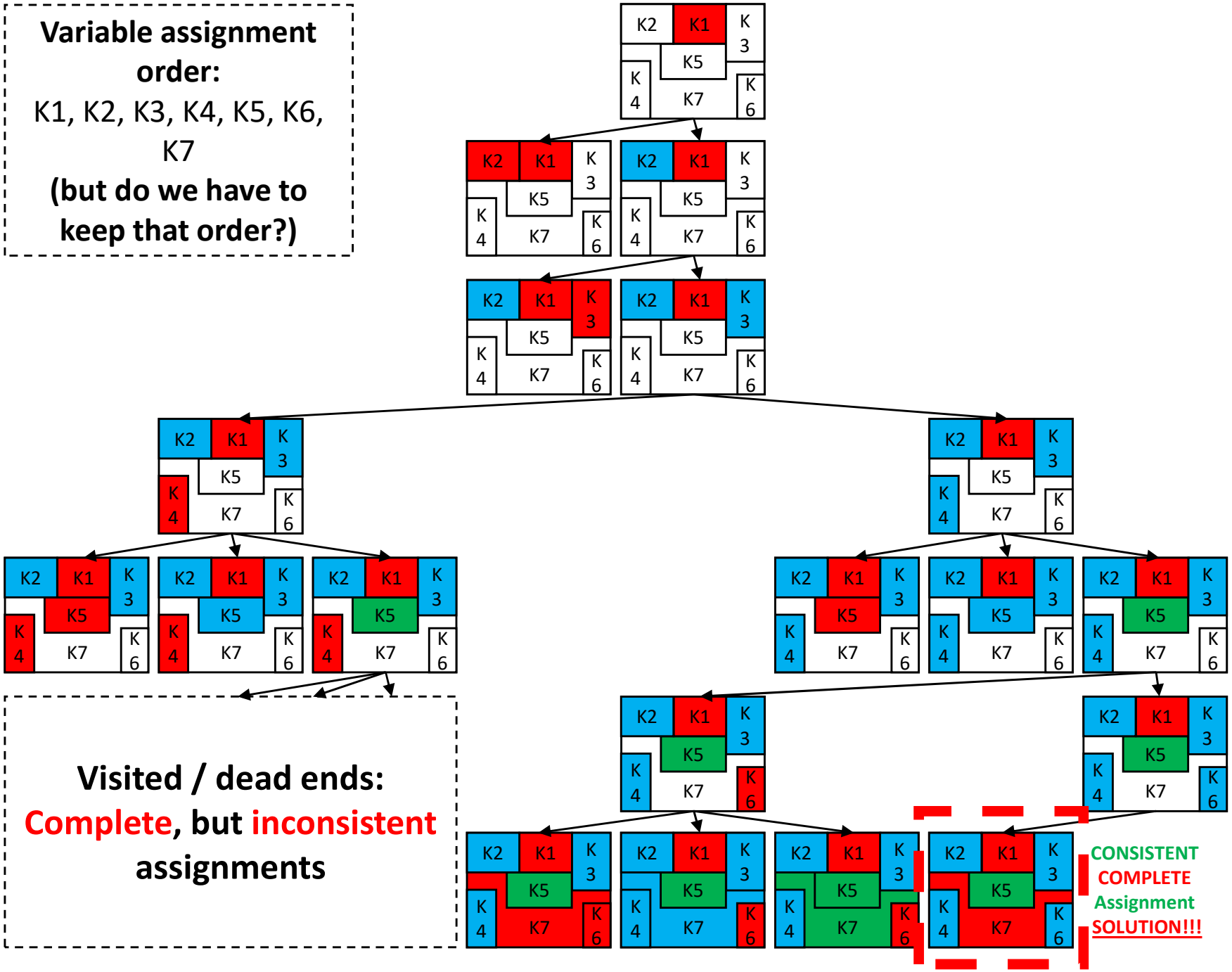
Can We Do Better?

CSP Backtracking: Pseudocode

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

Which variable should we choose to assign a value to next? Does it matter?

Variable assignment order:
K1, K2, K3, K4, K5, K6, K7
(but do we have to keep that order?)



Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7

Variable Ordering: Alternatives

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

You can modify this function to change the variable ordering and potentially improve performance

Variable Ordering: Alternatives

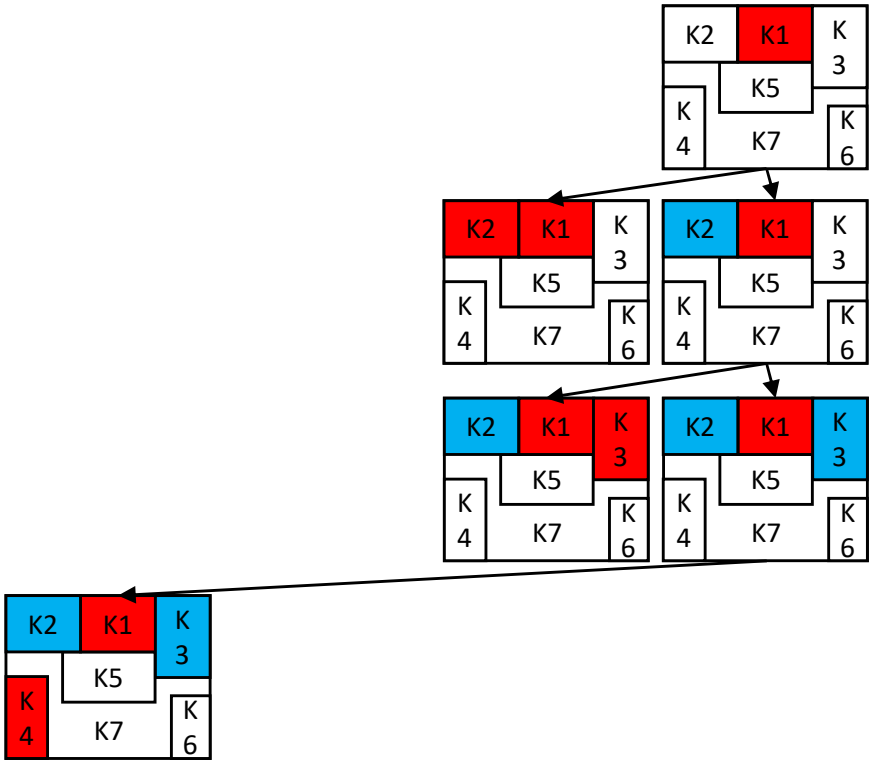
CSP Backtracking algorithm can use a number of variable ordering strategies:

- **Static: choose the variables in order (we did that)**
- **Random: order variables in random sequence**
- **Minimum-remaining-values (MRV) heuristic:**
  - **choose the variable with the "fewest" legal values**
- **Degree heuristic:**
  - **choose the variable involved in the largest amount of constraints on other unassigned variables**
  - **choose the variable with highest node degree on a constraint graph**

Variable Ordering: MRV Heuristic

As CSP Backtracking algorithm progresses, the number of possible value assignments for each variable will shrink (due to constraints):

- **MRV uses "fail-first" heuristics (also called "most constrained variable" heuristics)**
- **MRV picks a variable with lowest value assignment options "left"**
  - **expecting to limit exploration depth**
  - **likely to find a failure assignment faster**
- **Usually better than static and random orderings on average**



K1 = ???

K2 = ???

K3 = ???

K4 = ???

Which variable to explore next (ignore the EXPECTED sequence on the right)?
Available options:

K5: {GREEN}

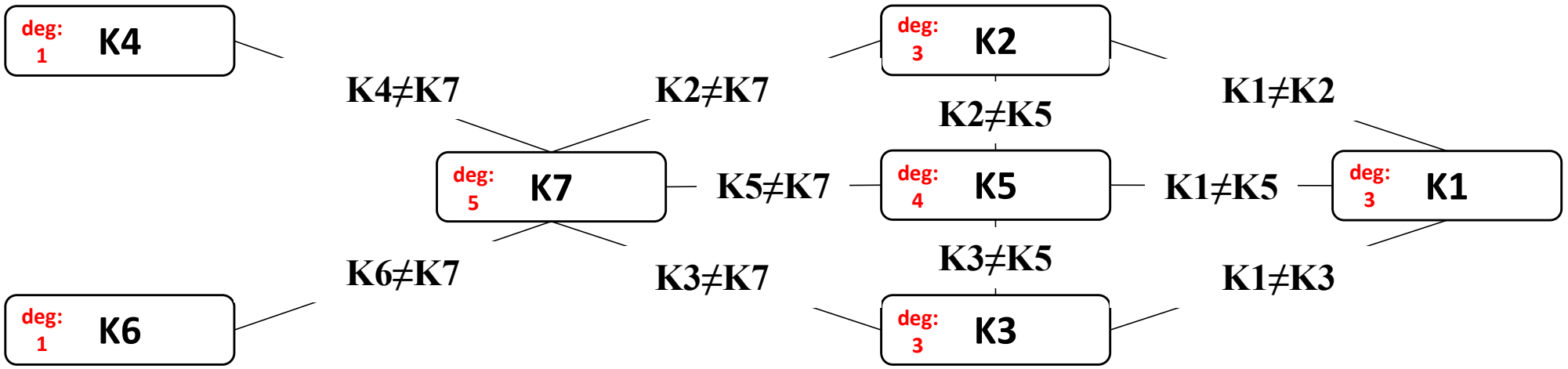
K6: {RED, BLUE, GREEN}

K7: {GREEN}

MRV should pick K5 or K7 ("fail first" variable).
Tie needs to be resolved.

Variable Ordering: Degree Heuristics

Consider the following constraint graph representation of the problem we analyzed:



- degree heuristics is considered less effective than MRV
- degree heuristics can be used as a tie-breaker (two variables with the same "potential" according to MRV)
- attempts to reduce the branching factor on future choices

Value Ordering: Alternatives

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

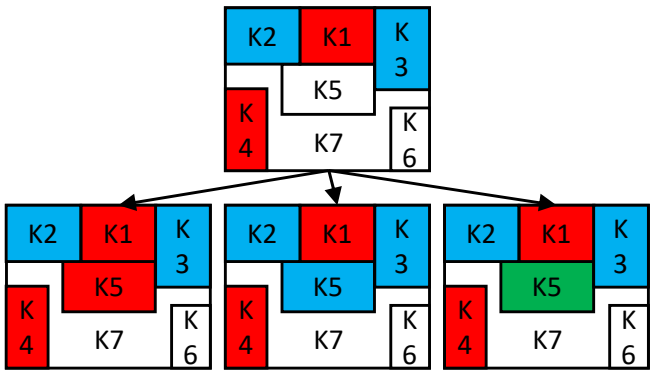
You can modify this order to change the value assignment ordering and potentially improve performance

Least-Constraining-Value Heuristics

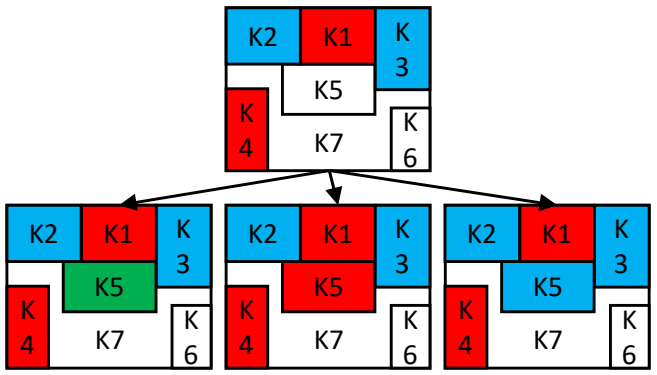
We picked (SELECT-UNASSIGNED-VARIABLE) the next variable to assign a value to and we have a number of values to choose from. What next?

- use the least-constraining-value heuristic
  - picks a value that rules out the fewest choices for neighboring variables in the constraing graph (increase flexibility for FUTURE assignments)
  - ORDER-DOMAIN-VALUES is the function that orders values here

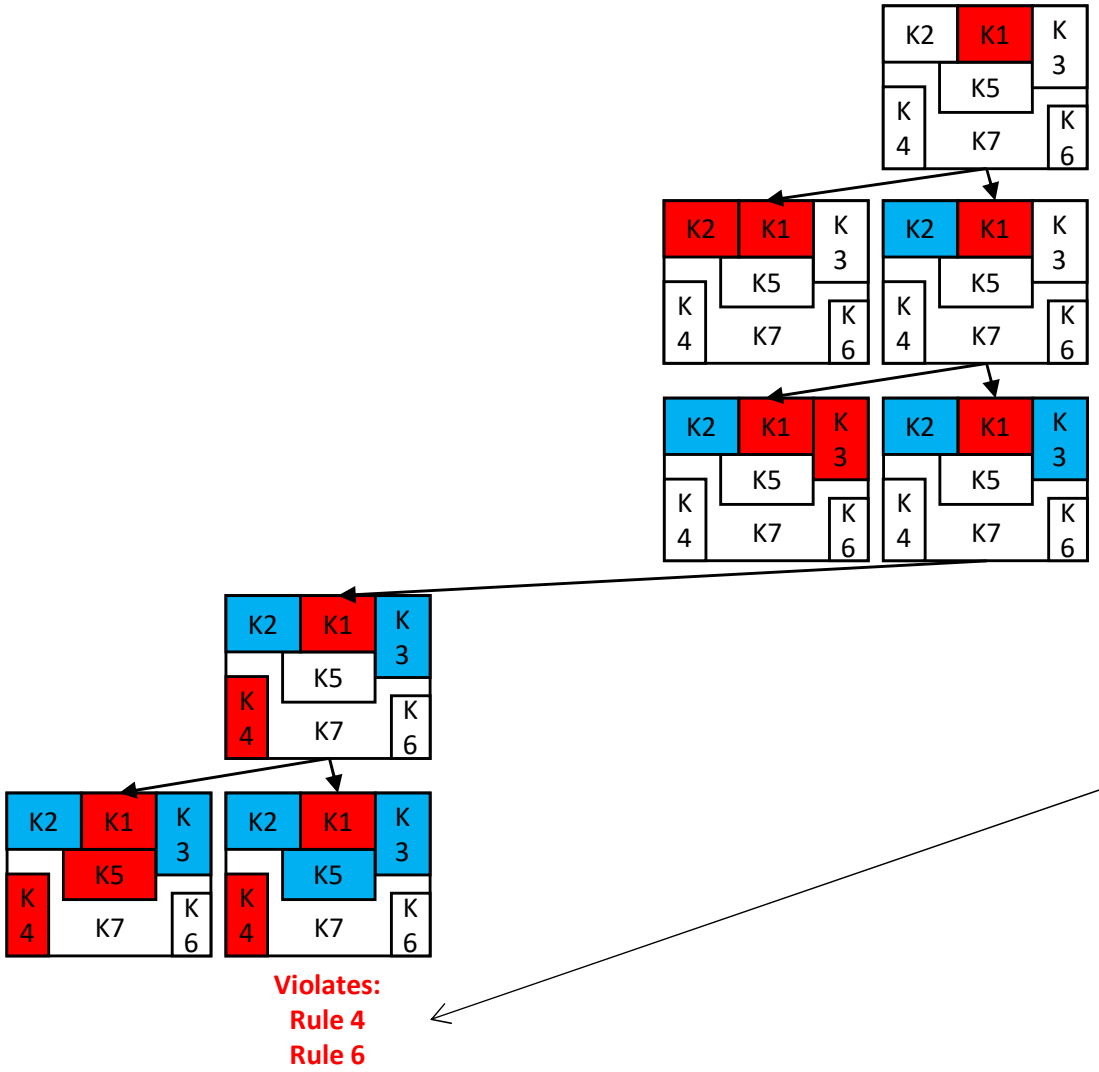
Next variable to assign a value to: K5. Three options available: {RED, BLUE, GREEN}. How to order them?



STATIC value ordering:
poor, best option is last



LEAST-CONSTRAINING-VALUE value ordering:
better, best option is first



INCONSISTENCY of this PARTIAL assignment was established at the node level:

node consistency

Can we detect / "predict" INCONSISTENCIES earlier and prune useless branches?

YES!

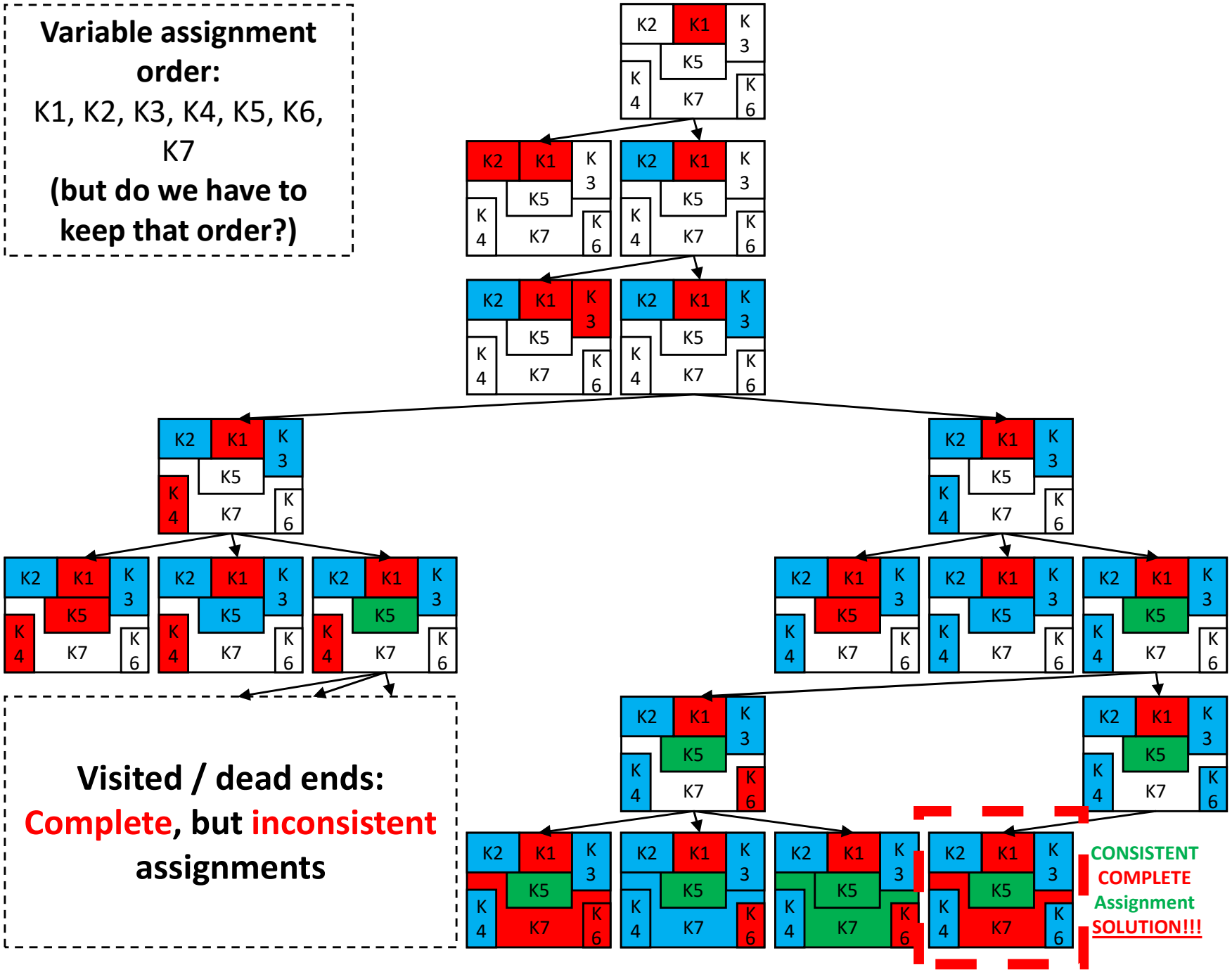
Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

CSP Backtracking: Pseudocode

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

Which variable should we choose to assign a value to next? Does it matter?

Variable assignment order:
K1, K2, K3, K4, K5, K6, K7
(but do we have to keep that order?)



Visited / dead ends:
Complete, but inconsistent assignments

Variable assignment order: K1, K2, K3, K4, K5, K6, K7

Variable Ordering: Alternatives

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

You can modify this function to change the variable ordering and potentially improve performance

Variable Ordering: Alternatives

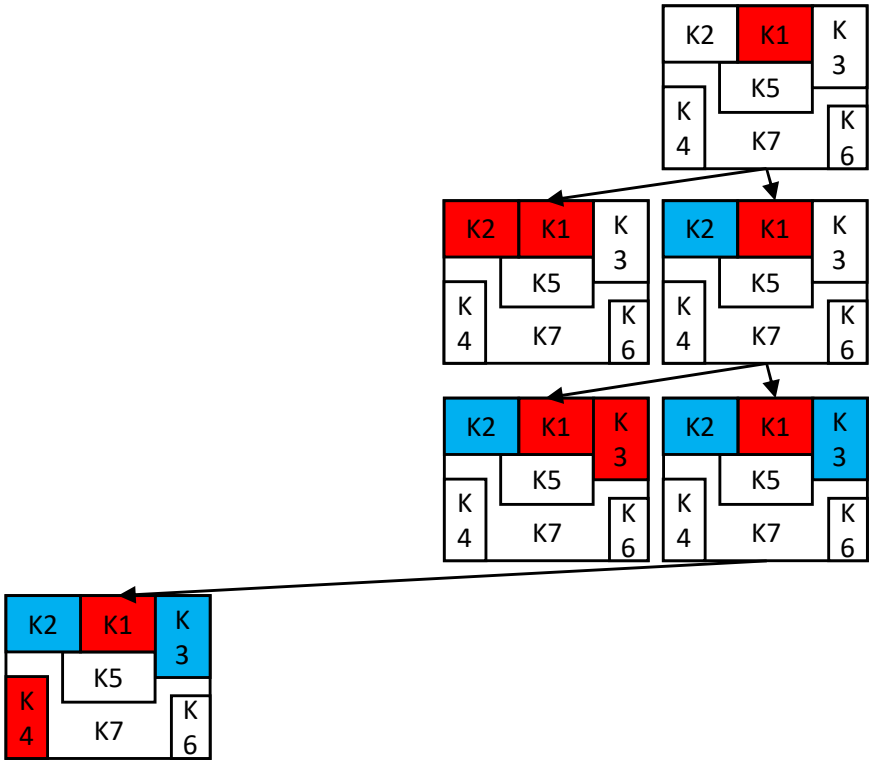
CSP Backtracking algorithm can use a number of variable ordering strategies:

- **Static: choose the variables in order (we did that)**
- **Random: order variables in random sequence**
- **Minimum-remaining-values (MRV) heuristic:**
  - **choose the variable with the "fewest" legal values**
- **Degree heuristic:**
  - **choose the variable involved in the largest amount of constraints on other unassigned variables**
  - **choose the variable with highest node degree on a constraint graph**

Variable Ordering: MRV Heuristic

As CSP Backtracking algorithm progresses, the number of possible value assignments for each variable will shrink (due to constraints):

- **MRV uses "fail-first" heuristics (also called "most constrained variable" heuristics)**
- **MRV picks a variable with lowest value assignment options "left"**
  - **expecting to limit exploration depth**
  - **likely to find a failure assignment faster**
- **Usually better than static and random orderings on average**



K1 = ???

K2 = ???

K3 = ???

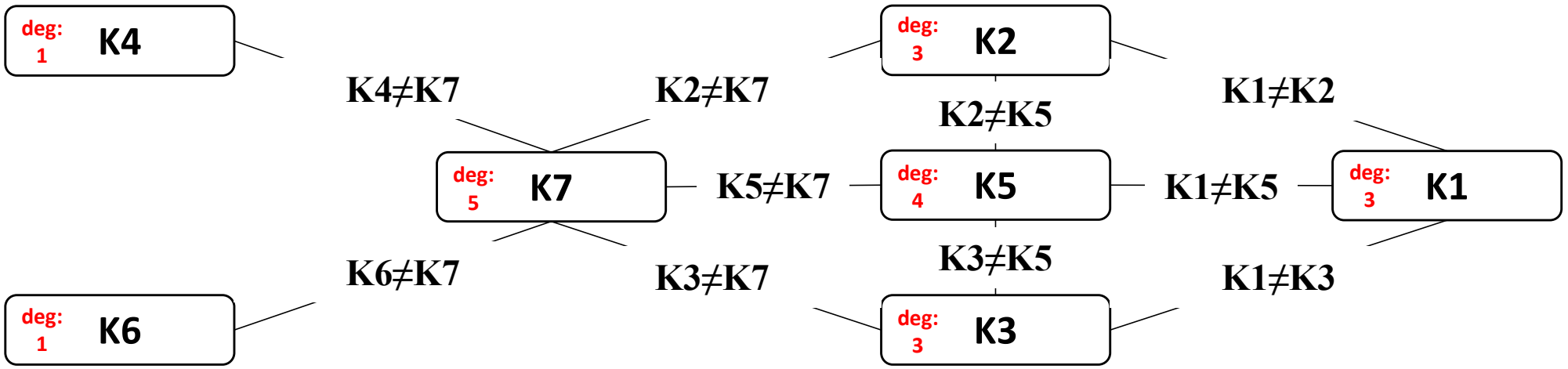
K4 = ???

Which variable to explore next (ignore the EXPECTED sequence on the right)?
Available options:
K5: {GREEN}
K6: {RED, BLUE, GREEN}
K7: {GREEN}

MRV should pick K5 or K7 ("fail first" variable).
Tie needs to be resolved.

Variable Ordering: Degree Heuristics

Consider the following constraint graph representation of the problem we analyzed:



- degree heuristics is considered less effective than MRV
- degree heuristics can be used as a tie-breaker (two variables with the same "potential" according to MRV)
- attempts to reduce the branching factor on future choices

Value Ordering: Alternatives

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

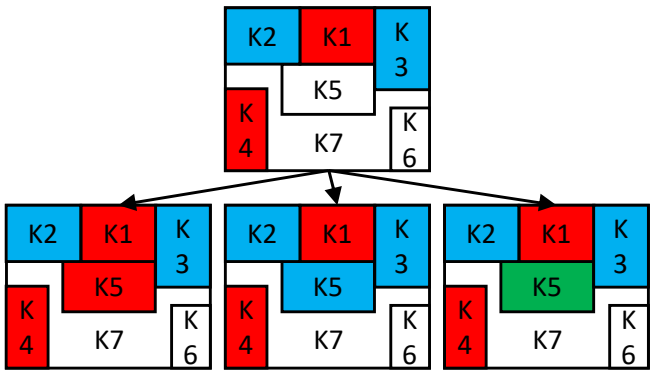
You can modify this order to change the value assignment ordering and potentially improve performance

Least-Constraining-Value Heuristics

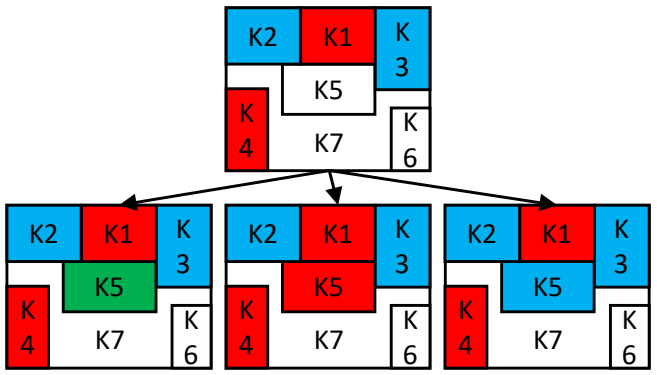
We picked (SELECT-UNASSIGNED-VARIABLE) the next variable to assign a value to and we have a number of values to choose from. What next?

- use the least-constraining-value heuristic
  - picks a value that rules out the fewest choices for neighboring variables in the constraing graph (increase flexibility for FUTURE assignments)
  - ORDER-DOMAIN-VALUES is the function that orders values here

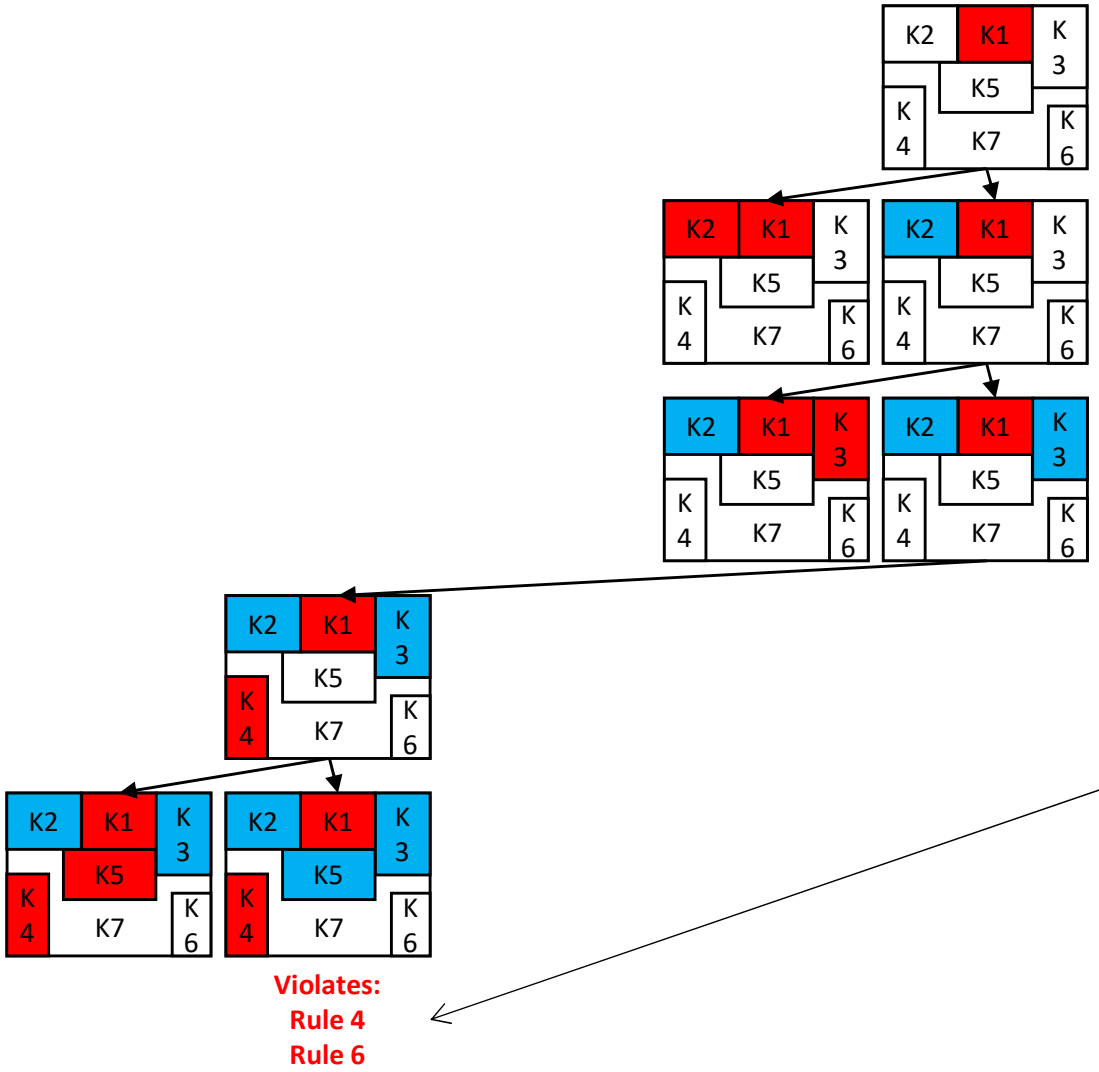
Next variable to assign a value to: K5. Three options available: {RED, BLUE, GREEN}. How to order them?



STATIC value ordering:
poor, best option is last



LEAST-CONSTRAINING-VALUE value ordering:
better, best option is first



INCONSISTENCY of this PARTIAL assignment was established at the node level:

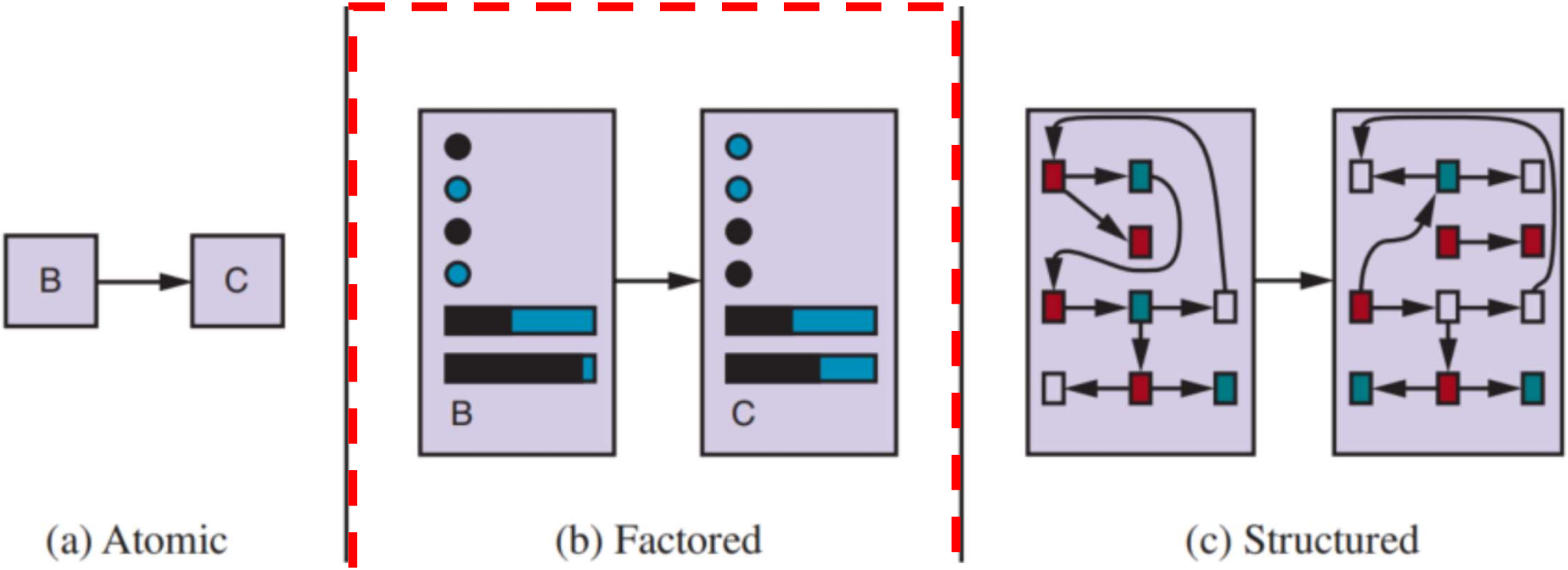
node consistency

Can we detect / "predict" INCONSISTENCIES earlier and prune useless branches?

YES!

Variable assignment order: K1, K2, K3, K4, K5, K6, K7 | **Value assignment order:** RED, BLUE, GREEN

How CSP Can Reduce Work



Next move?

- **Expand the node and visit succesors**

Next move?

- **Expand the node (assign value to a variable) and visit successors**
- **Infer where to go from current assignment and constraints (constraint propagation)**

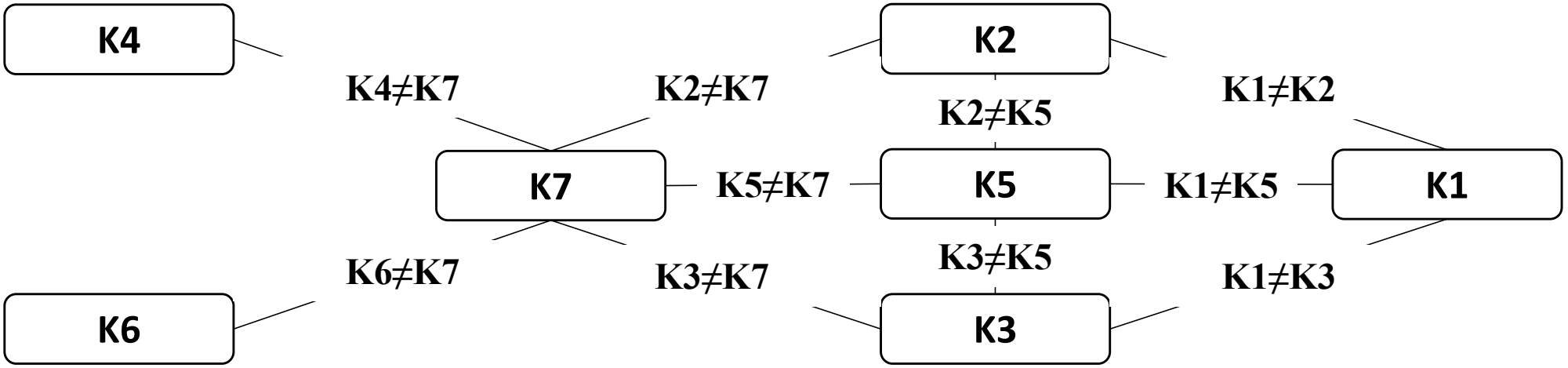
CSP: More Pruning with Inference

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

With the information available to you, you can INFER that a particular branch is going to be INCONSISTENT

Inference in CSP

- **Simplifying the problem:**
  - **preprocessing / pre-check or part of the search**
  - **it can reduce the problem OR even solve it**
- **Inference with Constraint Propagation:**
  - **use constraint graph to enforce consistency locally**

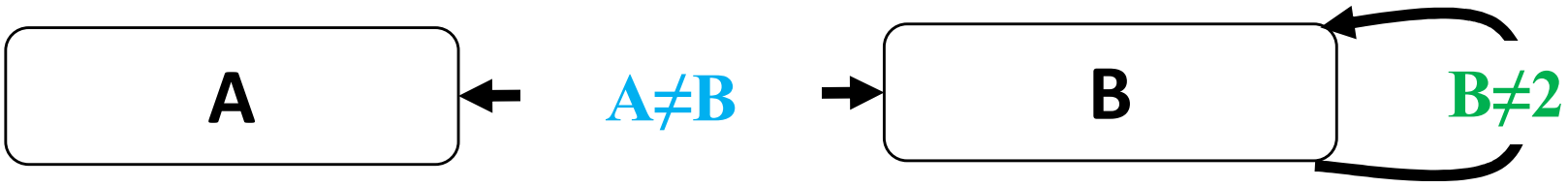


Local Consistency

- The idea:
  - remove inconsistent values from variable domains as we go as they would make certain assignments inconsistent later anyway
- Types:
  - Node consistency
  - Arc consistency (or edge consistency)
  - Path consistency

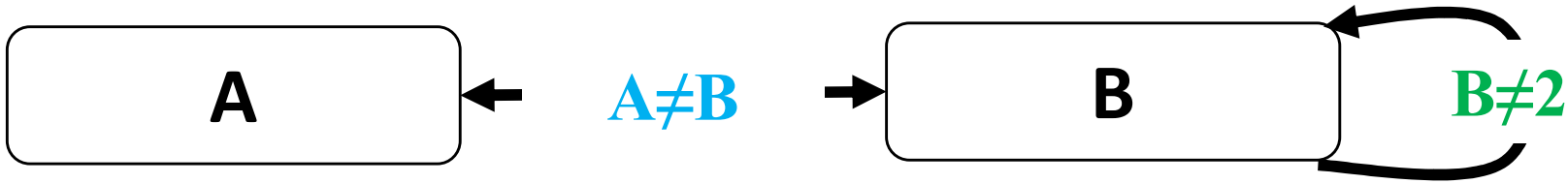
Node Consistency

- **Consider the following CSP example:**
  - **variables:** $X = \{A, B\}$
  - **domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{2, 3, 4\}$
  - **constraints:** $C = \{A \neq B, B \neq 2\}$
    - **one binary and one unary constraint**
  - **constraint graph:**



Node Consistency

- The idea:
  - a single variable is node-consistent (in a constraint graph) if all the values in its domain satisfy variable unary constraints
- (Constraint) graph is node-consistent if every variable in the graph is node-consistent

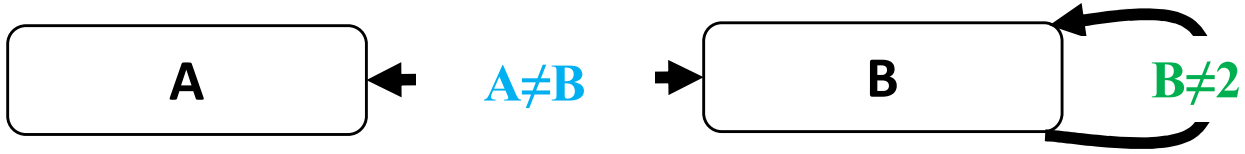


Variable B is NOT node-consistent because in $D_B = \{2,3,4\}$ value 2 does not satisfy unary B≠2

- Approach: remove unary constraints by reducing variable domain

Node Consistency

- **Unary constraints can easily be removed to reduce the problem:**
  - **BEFORE (unary constraint removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{2, 3, 4\}$



Constraint graph is NOT node-consistent because of variable B

  - **AFTER (unary constraint removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{3, 4\}$

Constraint graph is node-consistent

Arc (Edge) Consistency

- The idea:
  - a single variable is arc-consistent (in a constraint graph) if all the values in its domains satisfy ALL its binary constraints
- (Constraint) graph is arc-consistent if every variable in the graph is arc-consistent

Variables A and B are NOT arc-consistent because in $D_A = \{1,2,3\}$ and $D_B = \{3,4\}$ value 3 clashes

- Approach: reducing variable domains to remove clashes

Arc (Edge) Consistency

- **Values that clash can be removed from variable domains to reduce the problem:**
  - **BEFORE (clashing value(s) removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{3, 4\}$

Constraint graph is NOT arc-consistent because of value 3 clashing in both domains

  - **AFTER (clashing value(s) removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{4\}$ or
    - $D_A = \{0, 1\}$
    - $D_B = \{3, 4\}$ (depends on: which variable we start with)

Constraint graph is arc-consistent

AC-3 Algorithm: Pseudocode

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

```
queue ← a queue of arcs, initially all the arcs in csp
while queue is not empty do
  (X_i, X_j) ← POP(queue)
  if REVISE(csp, X_i, X_j) then
    if size of D_i = 0 then return false
    for each X_k in X_i.NEIGHBORS - {X_j} do
      add (X_k, X_i) to queue
return true
```

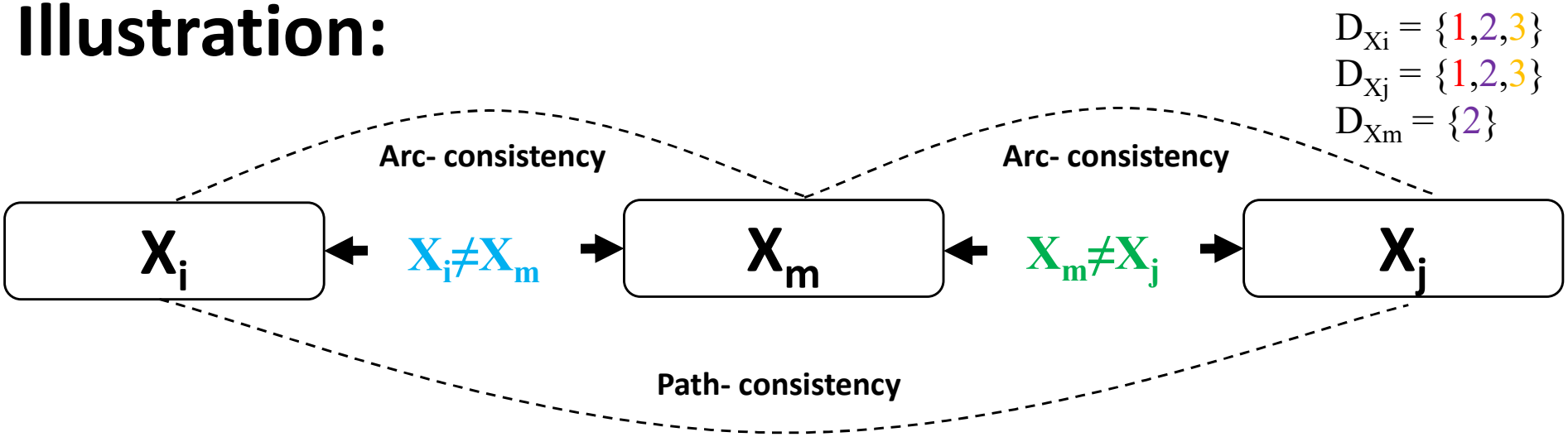
Note: treat a constraint graph edge as two directional edges: constraint $X_i \neq X_j$ corresponds to edges (X_i, X_j) and (X_j, X_i)

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i

```
revised ← false
for each x in D_i do
  if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
    delete x from D_i
    revised ← true
return revised
```

Path Consistency

- **The idea:**
  - **two variable set $\{X_i, X_j\}$ is path-consistent (in a constraint graph) with respect to a third variable X_m if for EVERY assignment $\{X_i = a, X_j = b\}$ there is an assignment to X_m (between X_i and X_j) that satisfies constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.**
- **Illustration:**



Searching with Inference

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

Apply local consistency checks and report failure if you know that following given path is going to dead end

Searching with Inference

Two key ideas:

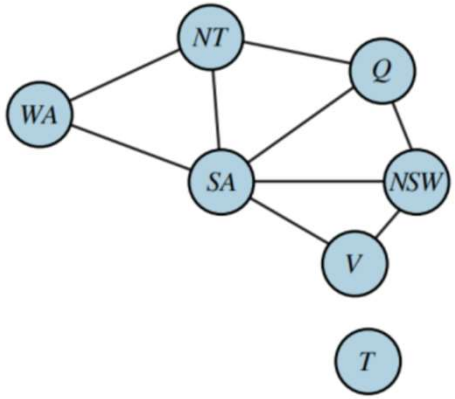
- **Forward checking**
- **Maintaining Arc Consistency**

Forward Checking

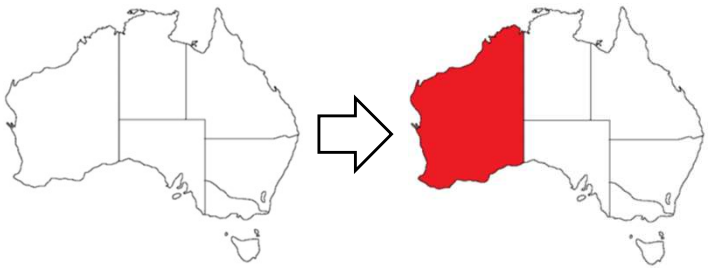
Idea:

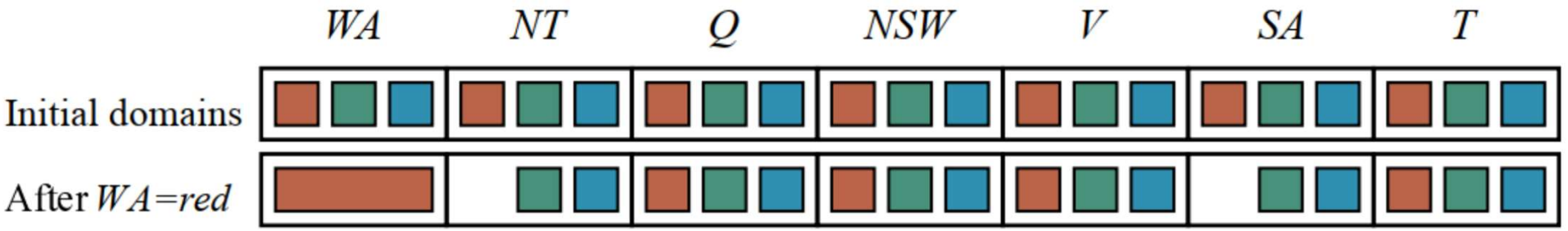
After some value a is assigned to variable X, examine every unassigned variable Y connected to X by a constraint and delete values from Y's domain that are inconsistent with a

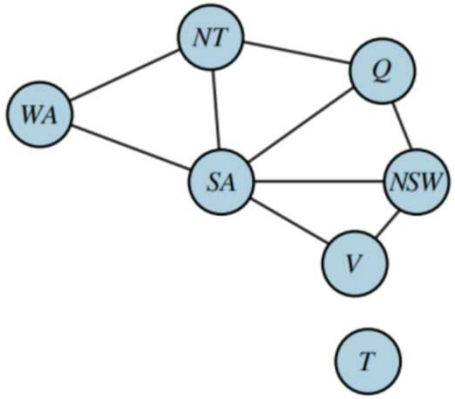
Forward Checking: Map of Australia



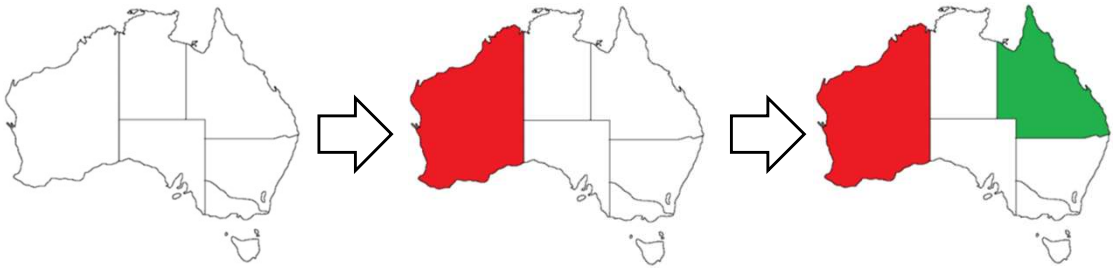
Forward Checking: Map of Australia

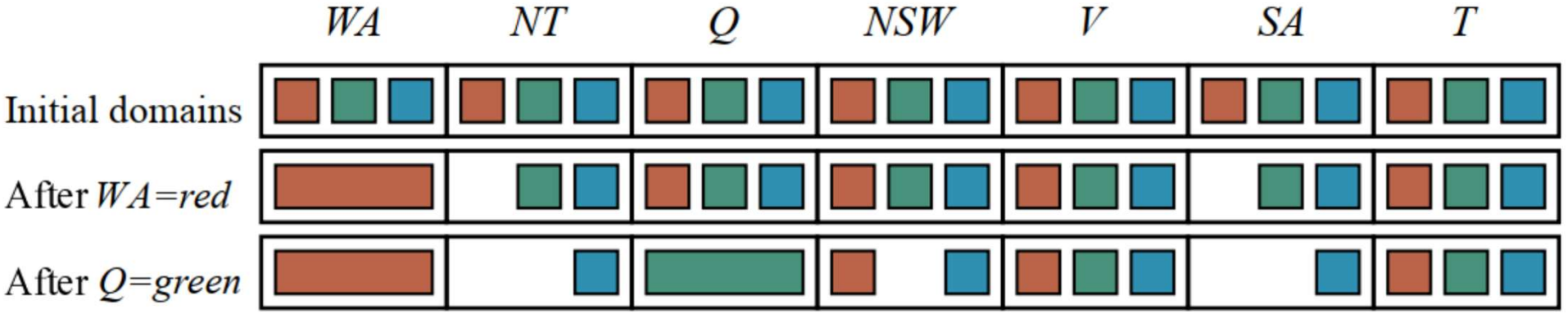


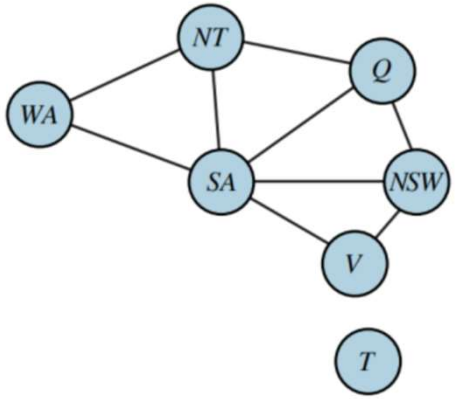




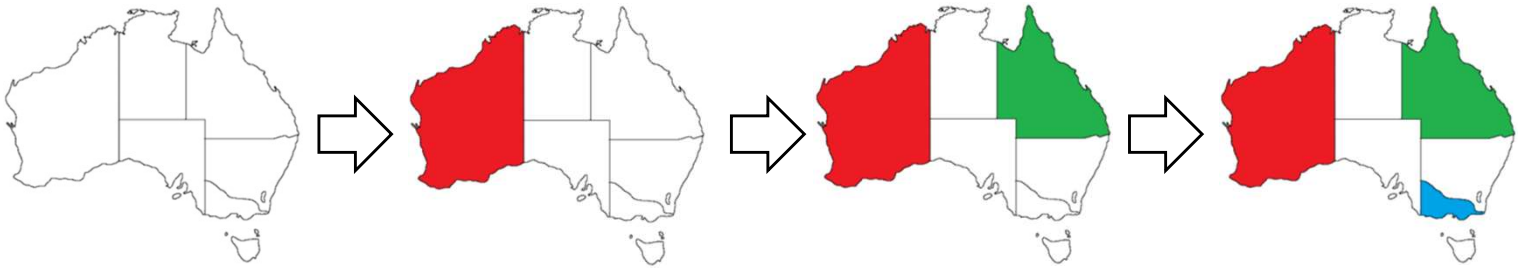
Forward Checking: Map of Australia

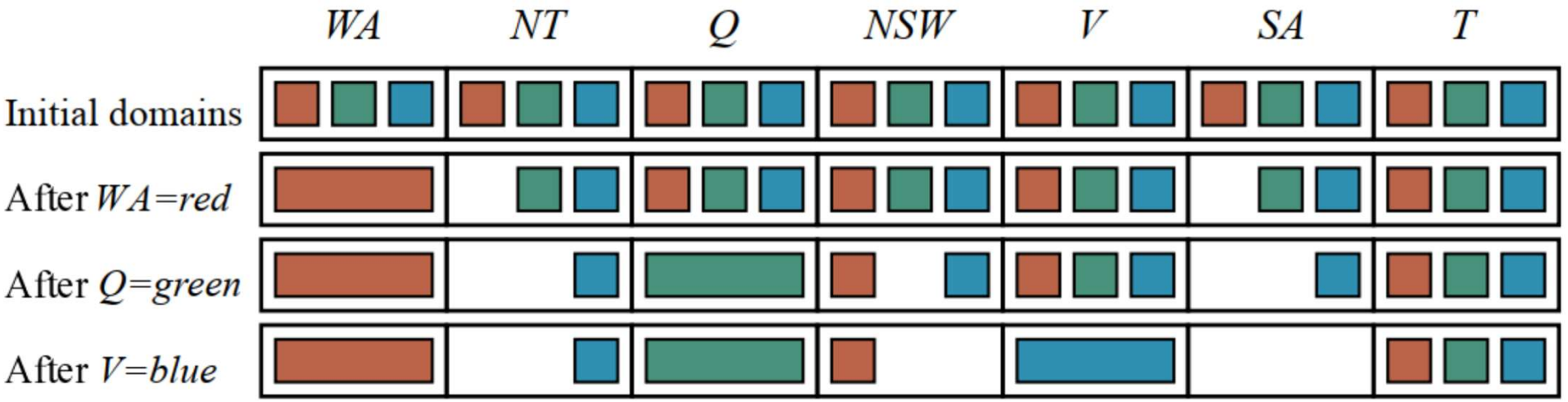


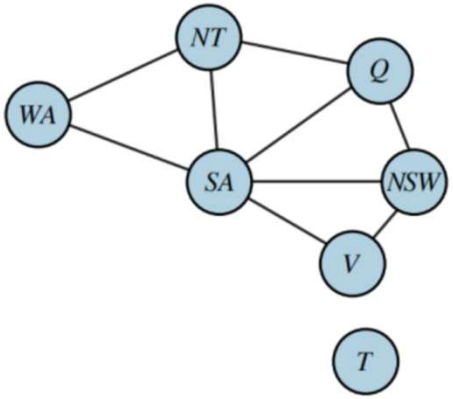




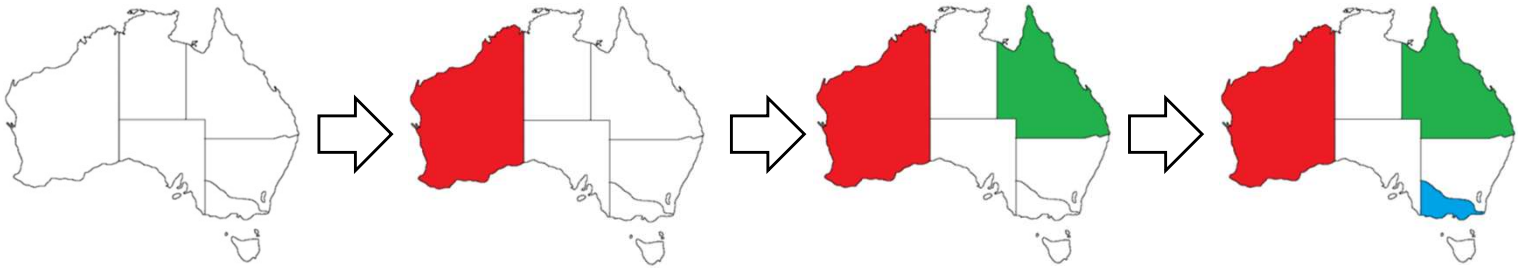
Forward Checking: Map of Australia

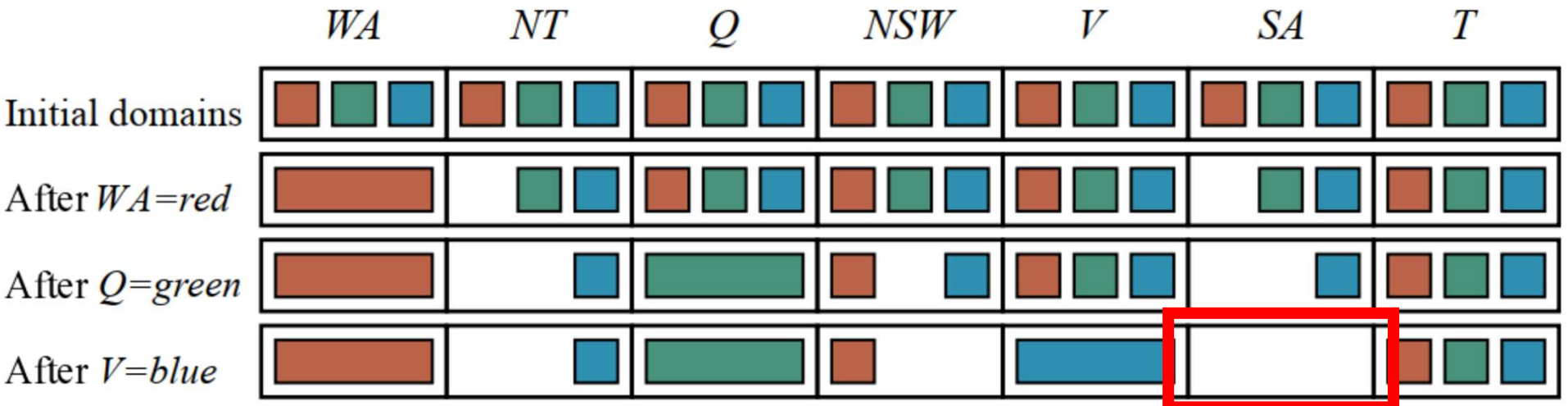


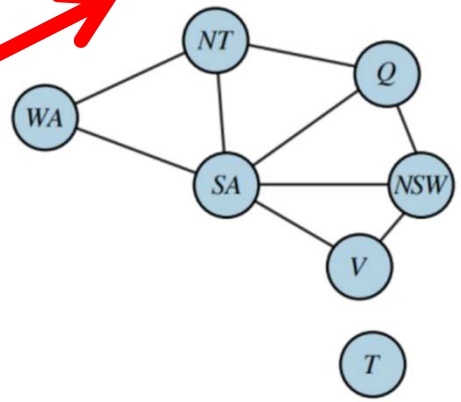




Forward Checking: Map of Australia

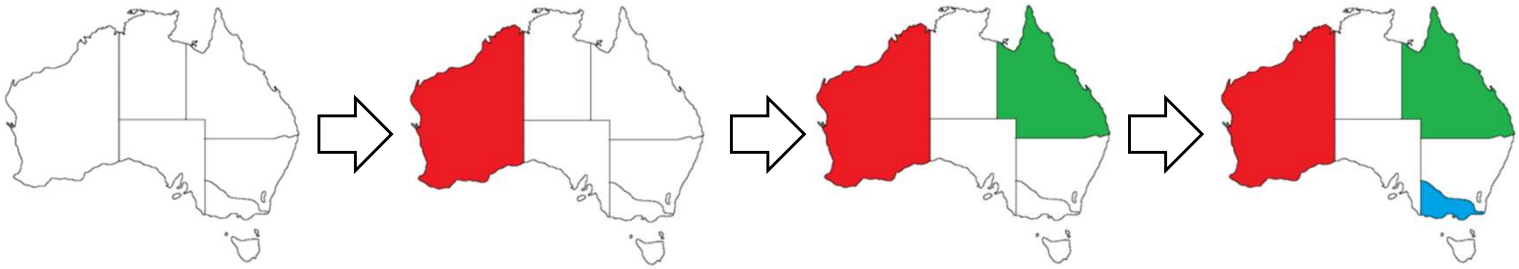


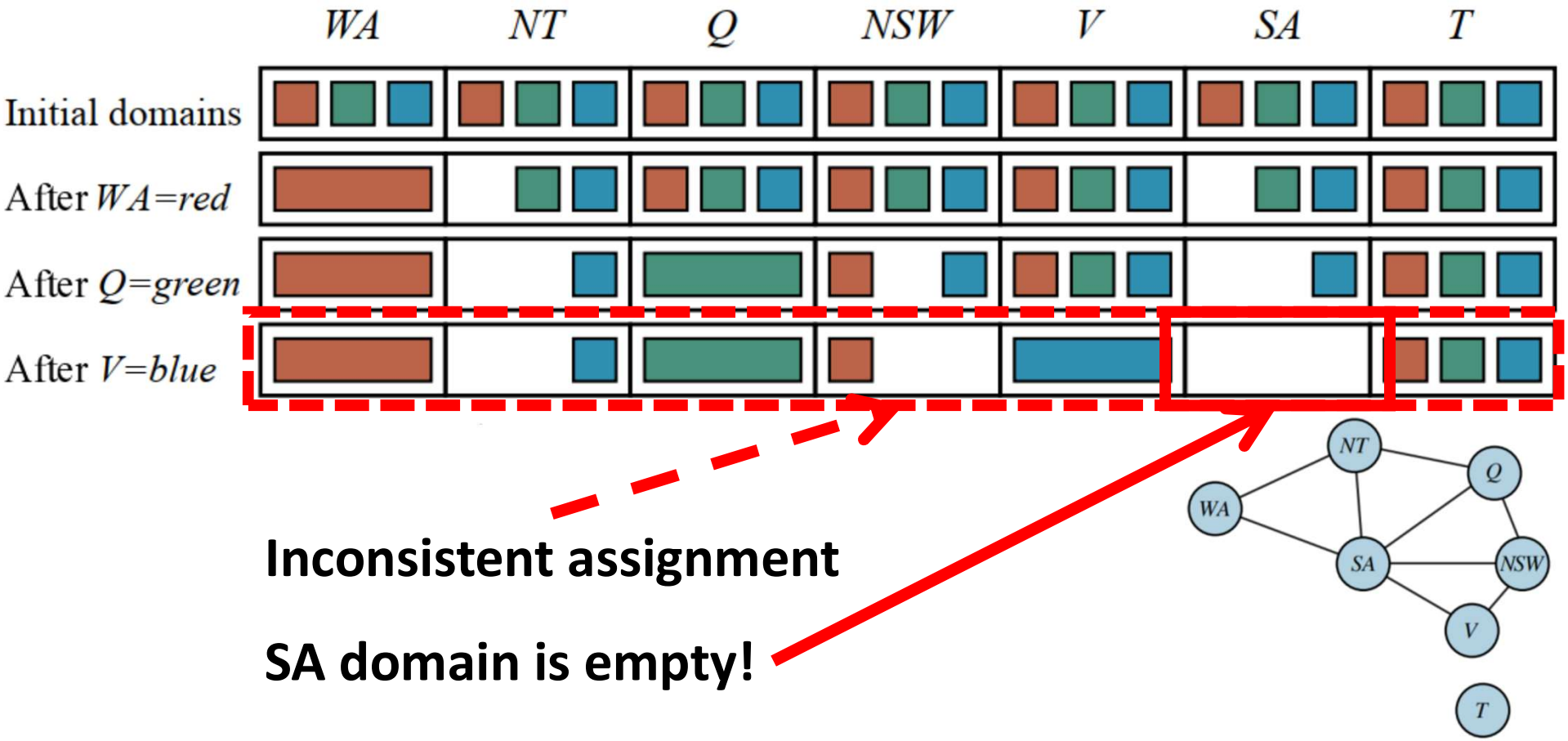




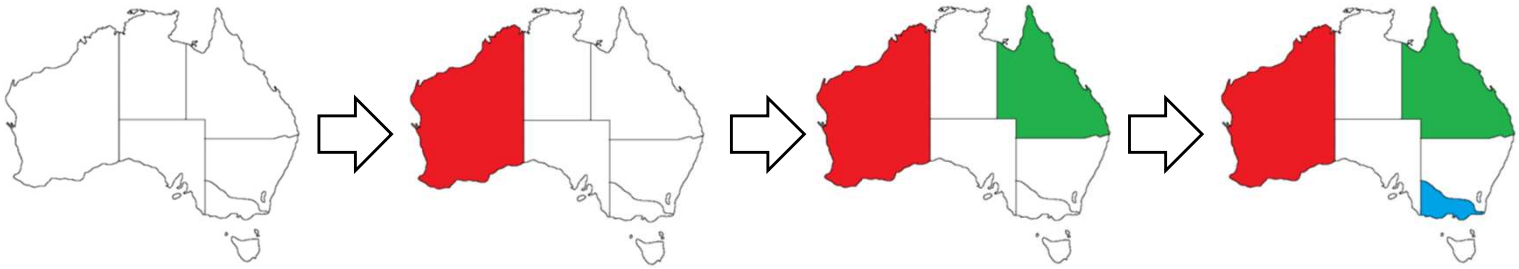
SA domain is empty!

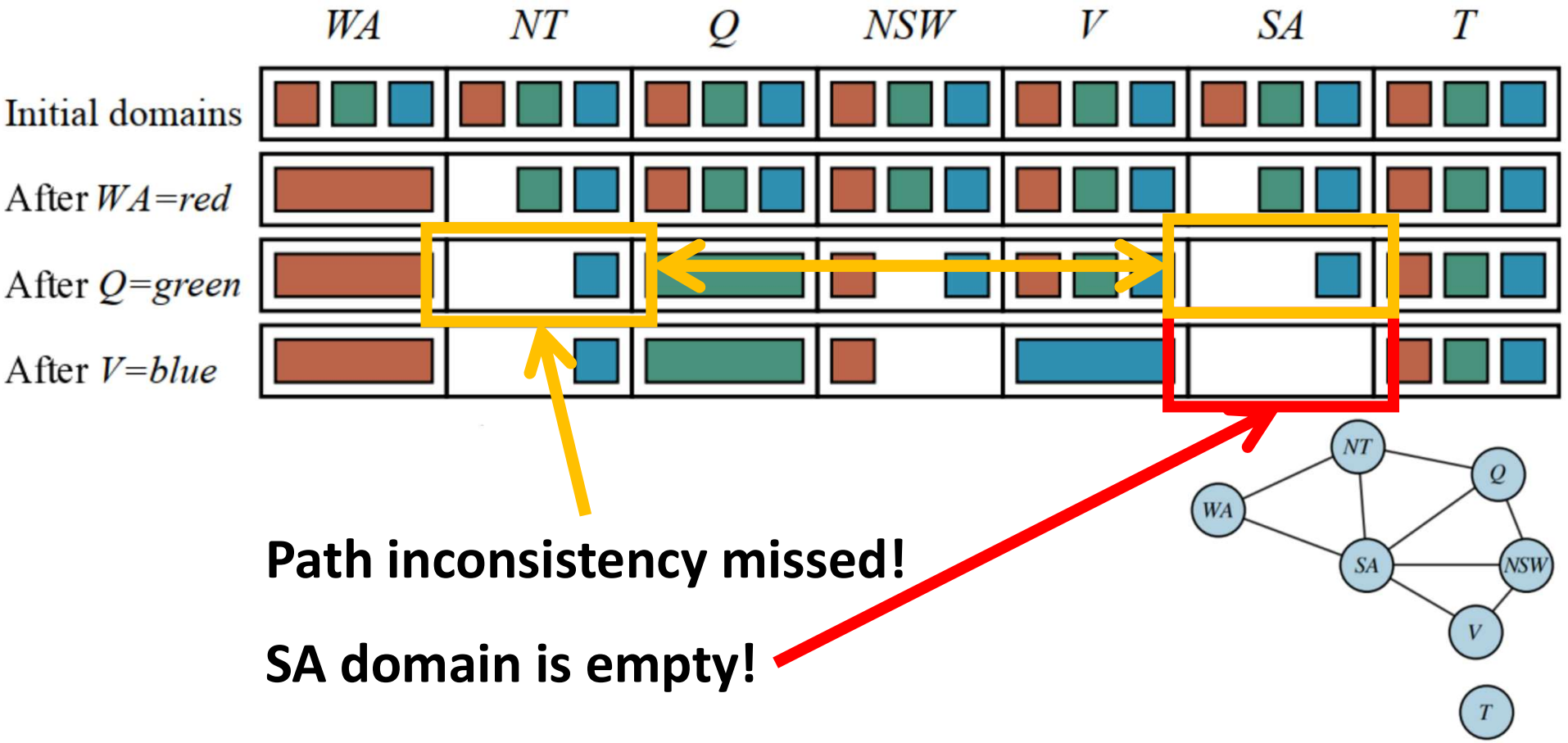
Forward Checking: Map of Australia





Forward Checking: Map of Australia





AC-3 Algorithm: Pseudocode

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

```
queue ← a queue of arcs, initially all the arcs in csp
while queue is not empty do
  (X_i, X_j) ← POP(queue)
  if REVISE(csp, X_i, X_j) then
    if size of D_i = 0 then return false
    for each X_k in X_i.NEIGHBORS - {X_j} do
      add (X_k, X_i) to queue
return true
```

Note: treat a constraint graph edge as two directional edges: constraint $X_i \neq X_j$ corresponds to edges (X_i, X_j) and (X_j, X_i)

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i

```
revised ← false
for each x in D_i do
  if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
    delete x from D_i
    revised ← true
return revised
```

Maintaing Arc-Consistency Algorithm

Idea:

After some value is assigned to variable X_i, infer by calling AC3 algorithm, but with a reduced number of edges / arcs for its queue:

- **only (X_i, X_j) arcs for all X_j variables that:**
  - **are constrained by X_i (neighbors of X_i on the constraint graph)**
  - **have no value assigned**

MAC Algorithm Call to AC3

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

queue ← a queue of arcs, ~~initially all the arcs in~~ *csp*

while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow$ POP(*queue*)
  if REVISE(*csp*, X_i, X_j) **then**
    if size of $D_i = 0$ **then return** *false*
    for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
      add (X_k, X_i) to *queue*
return *true*

only (Xi, Xj) arcs for all Xj variables that:

- are constrained by X_i (neighbors of X_i on the constraint graph)
- have no value assigned

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i

revised ← *false*
for each x **in** D_i **do**
  if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
    delete x from D_i
    revised ← *true*
return *revised*

Intelligent Backtracking

- **Chronological Backtracking:**
  - **Backpropagation used it**
- **Backjumping:**
  - **maintains a conflict set for a node X: a set of assignments that are in conflict with some X domain value**
  - **backtracks to a variable assignment level where a conflict (it ruled out some potential value of X earlier)**
  - **Forward checking can help construct conflict set**

Some CSP Challenges

- What if not all constraints can be satisfied?
  - Hard vs. soft constraints vs. preferences
  - Degree of constraint satisfaction concept
  - Cost of violating constraints
- What if constraints are of different forms?
  - Symbolic constraints
  - Logical constraints
  - Temporal constraints
  - Mixed constraints

Search Problems: Summary

- Initial problem analysis:
  - can it be represented with a state space?
  - what is the most useful state representation?
  - where, in the search tree, solution is expected? BFS or DFS?
- Do problem solutions need to be optimal?
- Do you care about time or space performance? Or both?
- Does your problem representation match known search algorithms?
  - Yes? Use it. No? See if you can make some simplifying assumptions and ask that question again
- Use all available knowledge about the problem to come with handy heuristics and use them to prune search tree