Programmation des jeux

Aziz KHAMJANE

Plan

- Introduction
- Définitions
- L'algorithme MINIMAX
- MinMax avec profondeur limitée
- L'élagage $\alpha - \beta$

Introduction

- Dans les jeux, le monde n’est pas statique, mais dynamique. Le joueur adverse peut modifier l’environnement.
- Plus spécifiquement, un environnement multi-agent (l’autre joueur est un agent non contrôlable et compétitif).

- Q : Comment peut-on résoudre ce problème ?

Supposer que l’adversaire joue de façon rationnelle…

Introduction

Dans un jeu, les joueurs peuvent être :

- Coopératifs.
  - Ils veulent atteindre le même but.
- En compétition direct (avec adversaires).
  - Un gain pour les uns est une perte pour les autres.
  - Cas particulier : les jeux à somme nulle (zero-sum games).
    - Jeux d'échecs, de dame, tic-tac-toe, Connect5, etc.

C'est le type de jeux qui nous intéresse aujourd'hui.

Hypothèses

Théorie des jeux (Game Theory) = sujet large.

Pour l'instant, nous aborderons les :

- Jeux à deux adversaires.
- Jeux discrets à tour de rôle.
- Jeux à somme nulle.
- Jeux avec observation totale.
- Jeux déterministes (sans hasard ou incertitude).

Définition d'un jeu

- Un jeu peut être formellement défini comme un problème de recherche, avec :
  - S0 : l'état initial, qui spécifie l'état du jeu au début de la partie
  - Player(s) : définit quel joueur doit jouer dans l'état s
  - Action(s) : retourne l'ensemble d'actions possibles dans l'état s
  - Result(s, a) : fonction de transition, qui définit quel est le résultat de l'action a dans un état s
  - Terminal-Test(s) : test de terminaison. Vrai si le jeu est fini dans l'état s, faux sinon. Les états dans lesquels le jeu est terminé sont appelés états terminaux.
  - Utility(s, p) : une fonction d'utilité qui associe une valeur numérique à chaque état terminal s pour un joueur p.

Exemple : jeu d’échecs

S0 = l’état initial

- Player={blanc, noire}
- Action(s) : les actions possibles à l’état s en respectant les règles du jeu d’échecs.
- Result(s, a) : fonction de transition, qui définit quel est le résultat de l’action a dans un état s.
- Terminal-Test(s) : échec et mat.
- Utility(s, p) : 1 si blanc gagne, -1 si noire gagne et 0 si match nul.

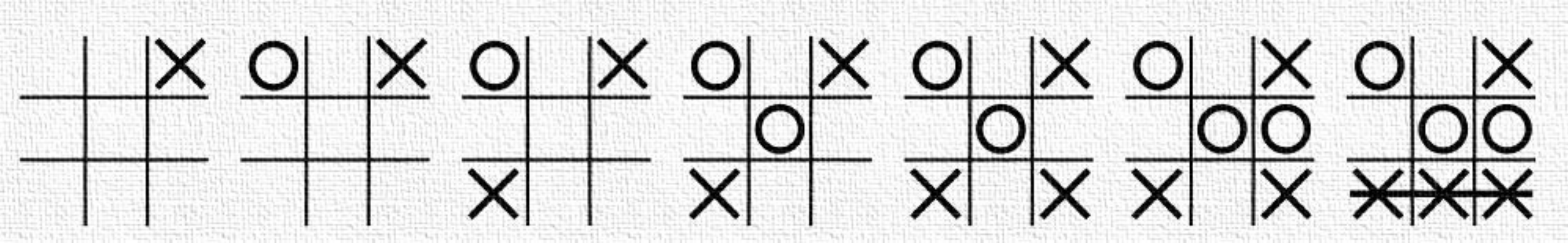
Jeux à somme nulle

- Un jeu a somme nulle est un jeu pour lequel la somme des utilités de tous les joueurs est la même pour toutes les issues possible du jeu.
- Jeu à somme constante serait plus approprié
- Par exemple :
  - 0 + 1 =1
  - 1 + 0 =1
  - 1 /2 + 1 /2=1

L'algorithme Minimax

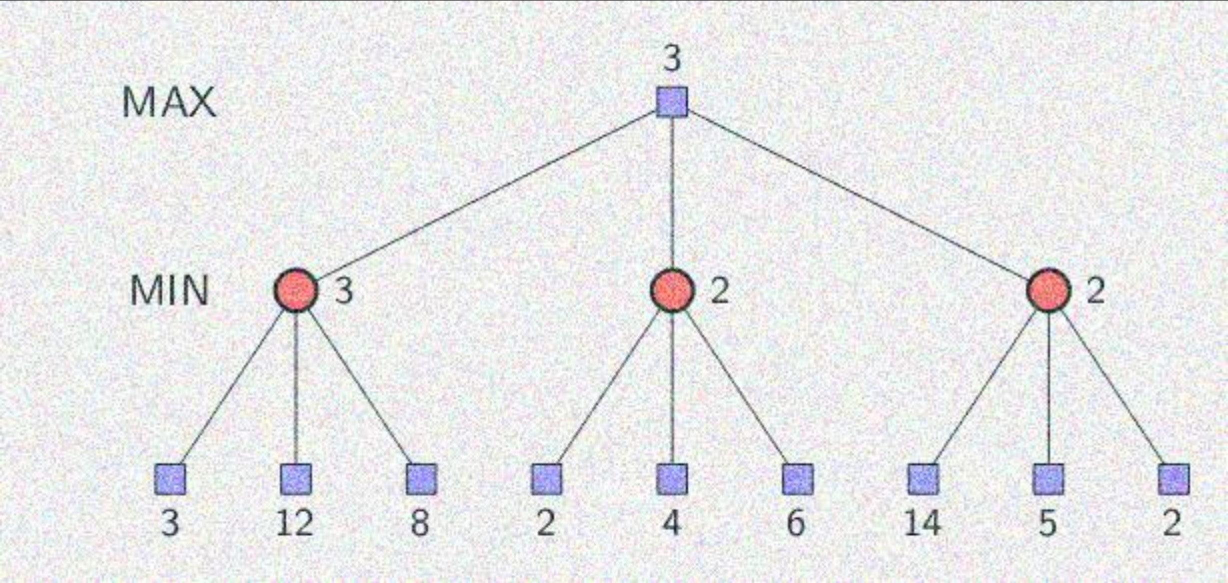
- L'algorithme Minimax s'applique sur des jeux :
  - ✓ à deux joueurs, appelés Max et Min. Par convention, Max joue en premier
  - ✓ à somme nulle.

tic-tac-toe



Algorithme de recherche Minimax

- Idée : choisir le coup qui mène vers l'état qui a la meilleure valeur minimax = meilleure valeur possible contre le meilleur jeu de l'adversaire.
- Exemple d'un jeu à deux coups :



Algorithme de recherche Minimax

1) Évaluer chaque nœud terminal
2) propager ces valeurs aux nœuds non-terminaux
   - La valeur min (adversaire) aux nœuds du joueur MIN
   - La valeur max (joueur) aux nœuds du joueur MAX

Algorithme: On visite l'arbre de jeu pour faire remonter à la racine une valeur (appelée « valeur du jeu ») qui est calculée récursivement de la façon suivante :

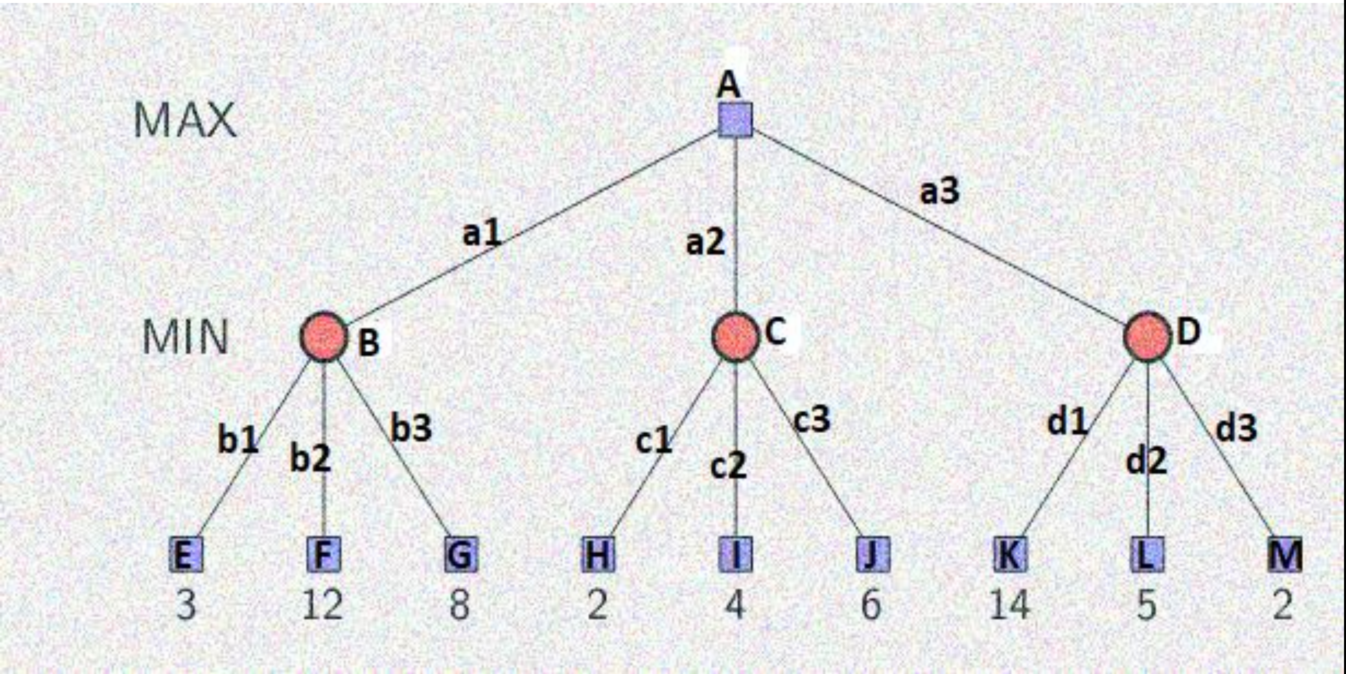
– $minimax(p) = f(p)$ si p est une feuille de l’arbre où f est une fonction d’évaluation de la position du jeu.

– $minimax(p) = MAX(minimax(O1), \ldots, minimax(On))$ si p est un nœud du Joueur MAX où $O1 \ldots On$ sont les fils du nœud p.

– $minimax(p) = MIN(minimax(O1), \ldots, minimax(On))$ si p est un nœud Joueur MIN où $O1 \ldots On$ sont les fils du nœud p.

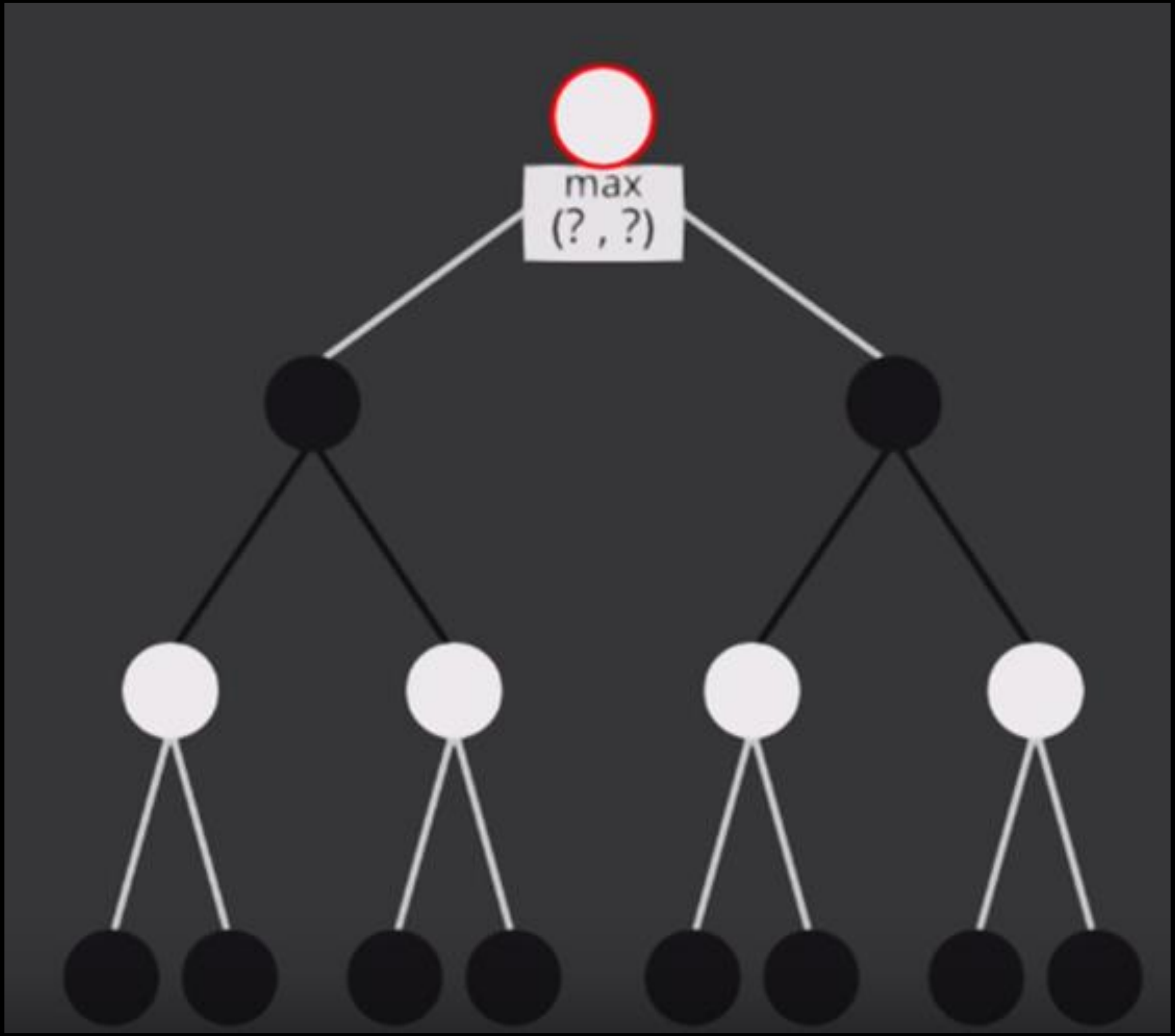
Algorithme de recherche Minimax

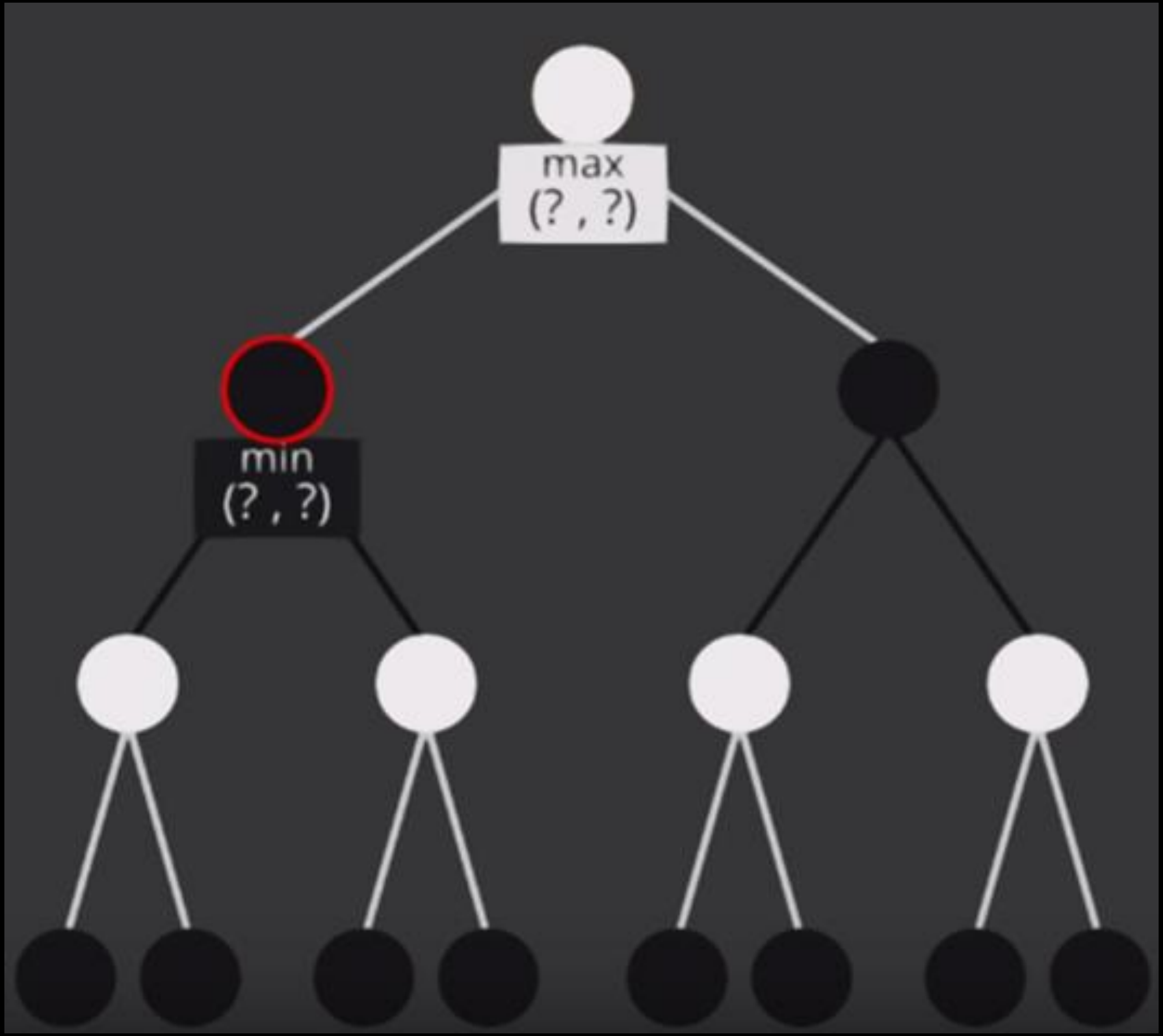
$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

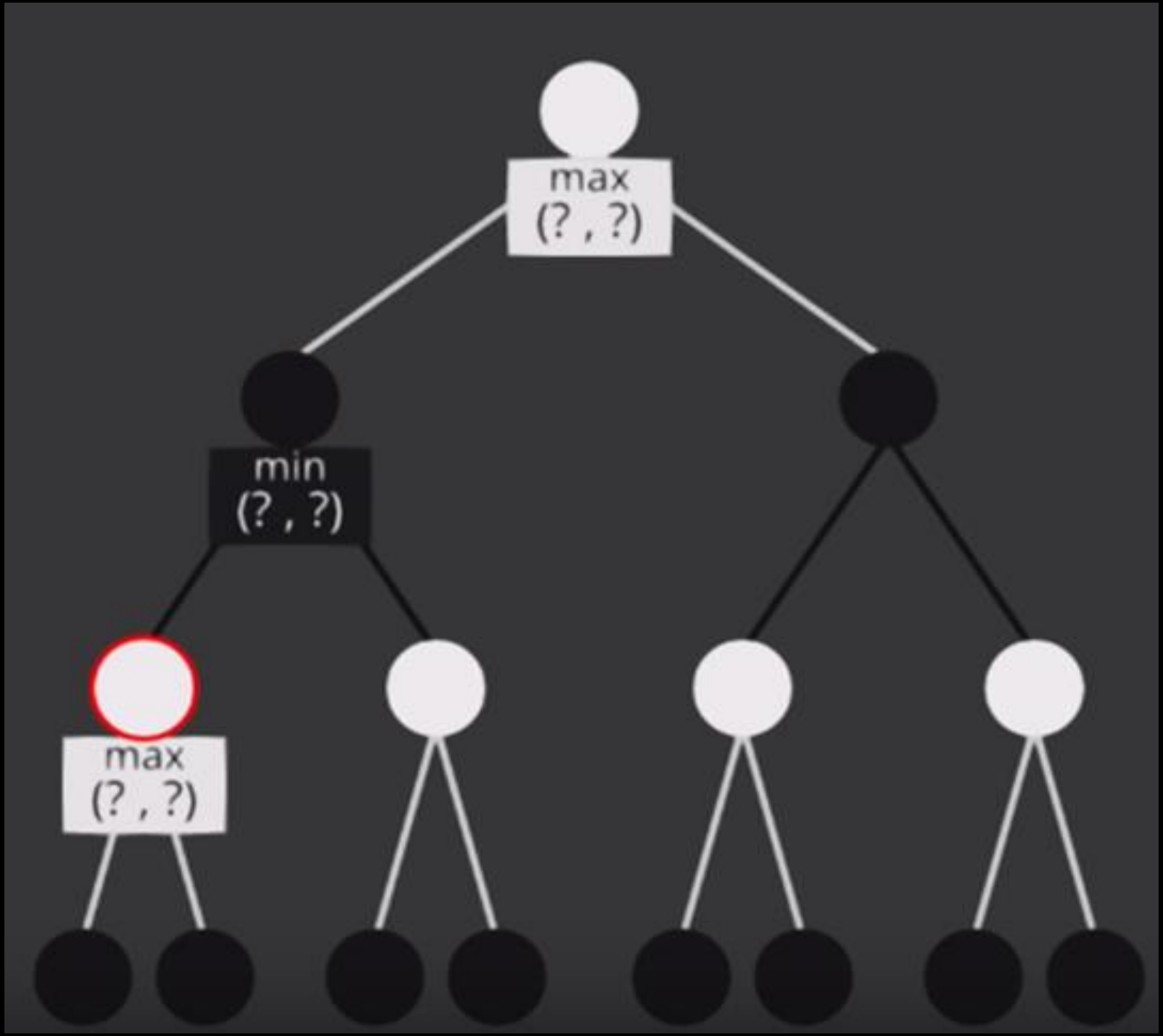


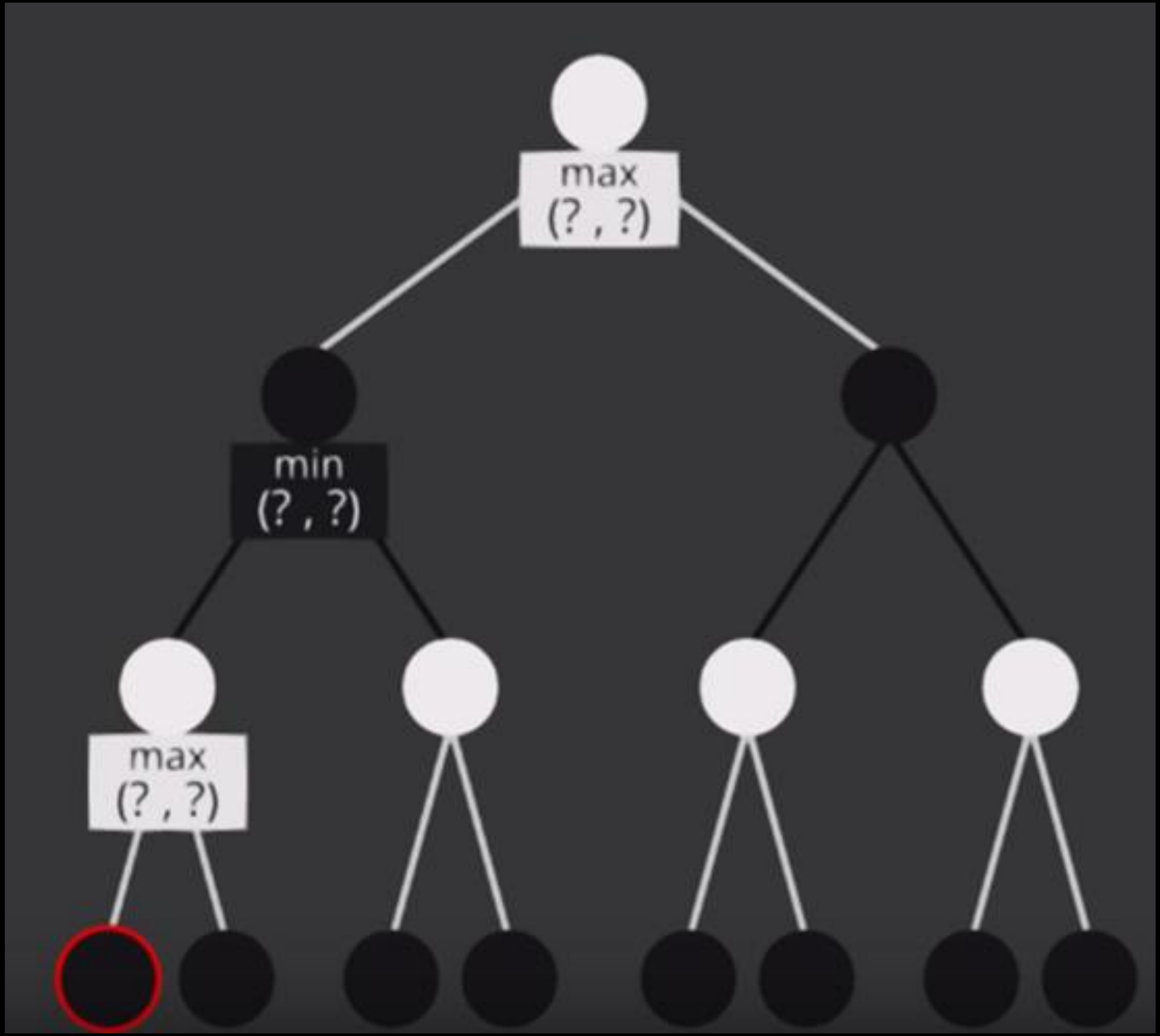
Algorithme de recherche Minimax

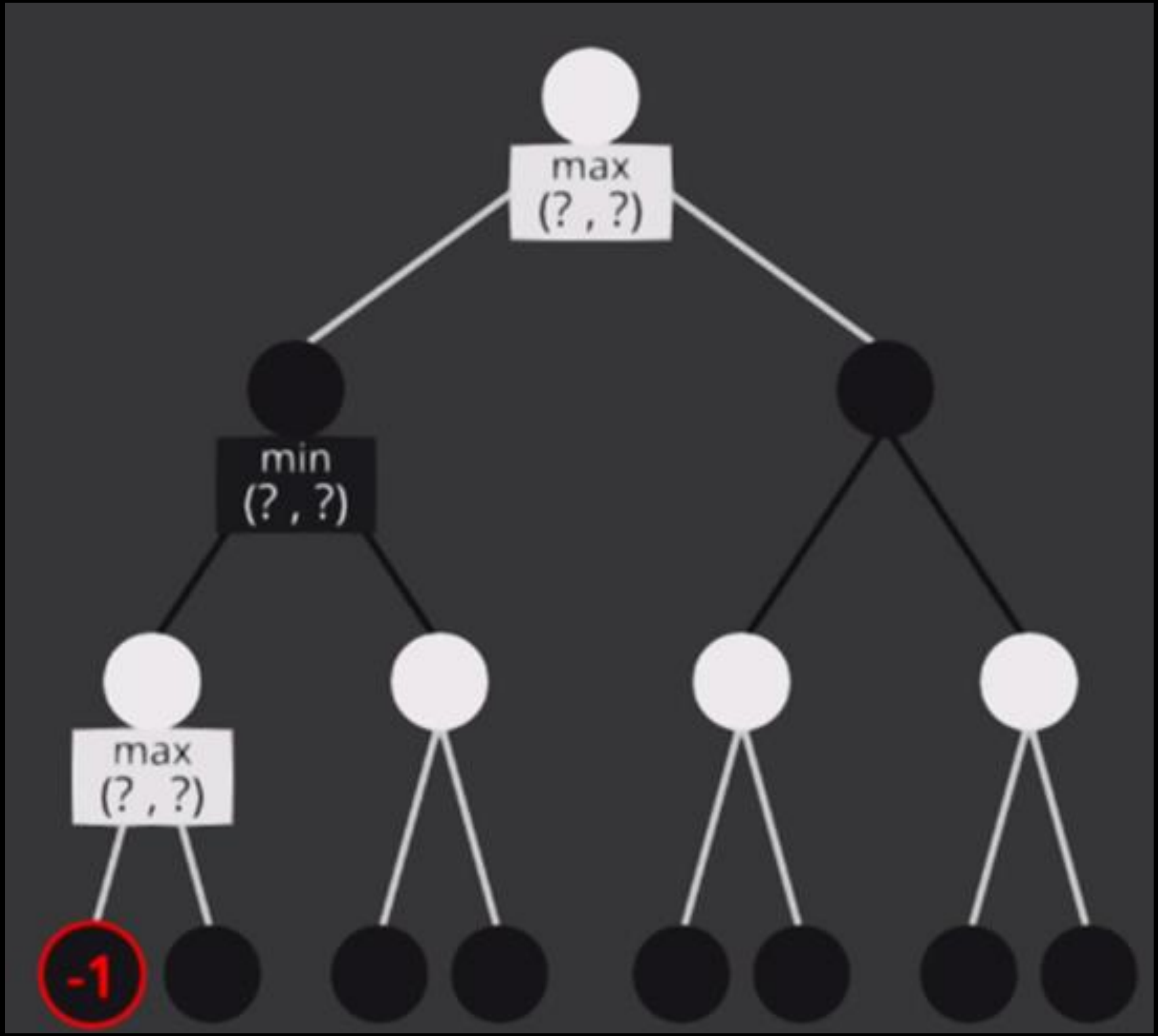
```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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

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

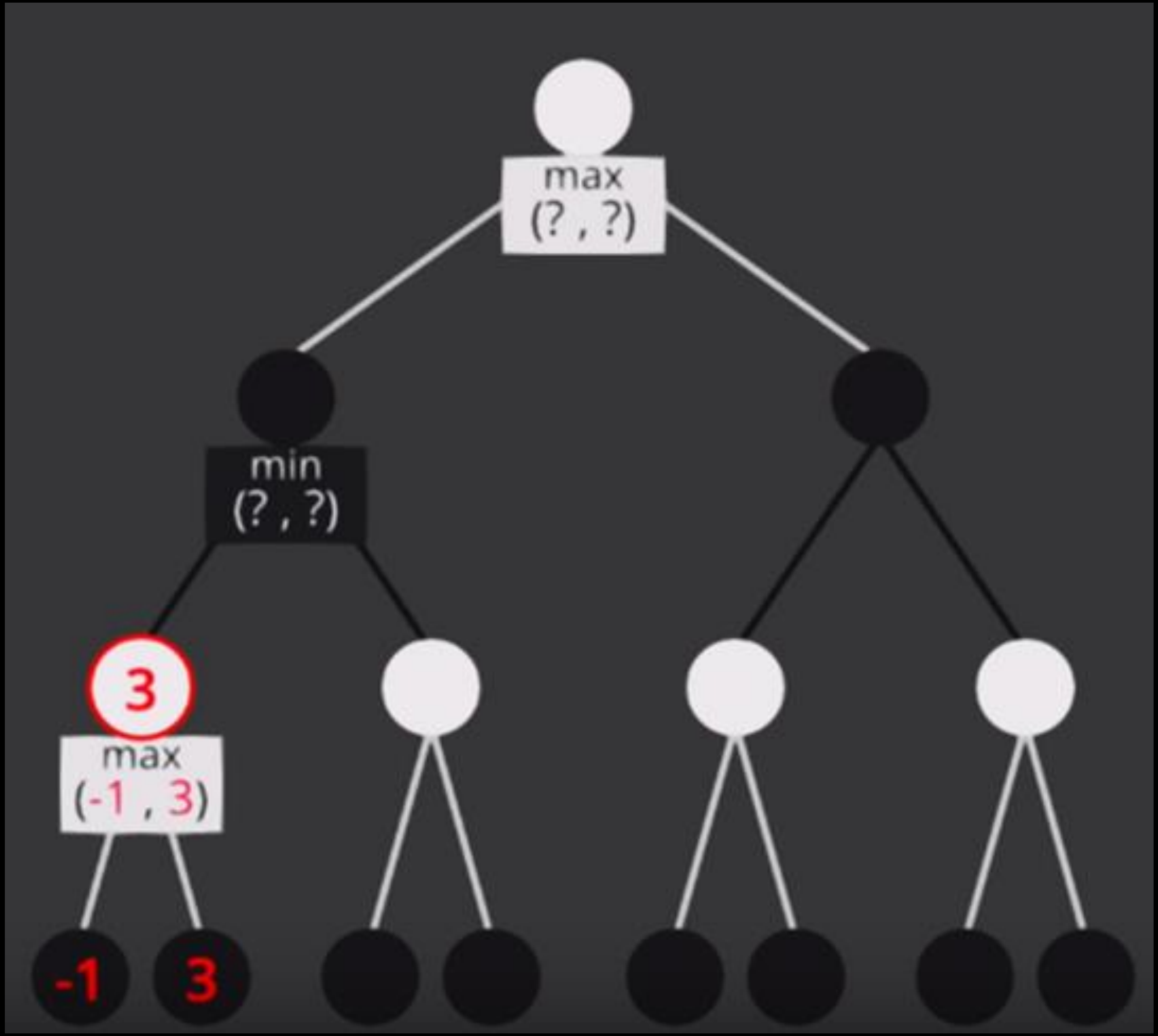

max
(? , ?)

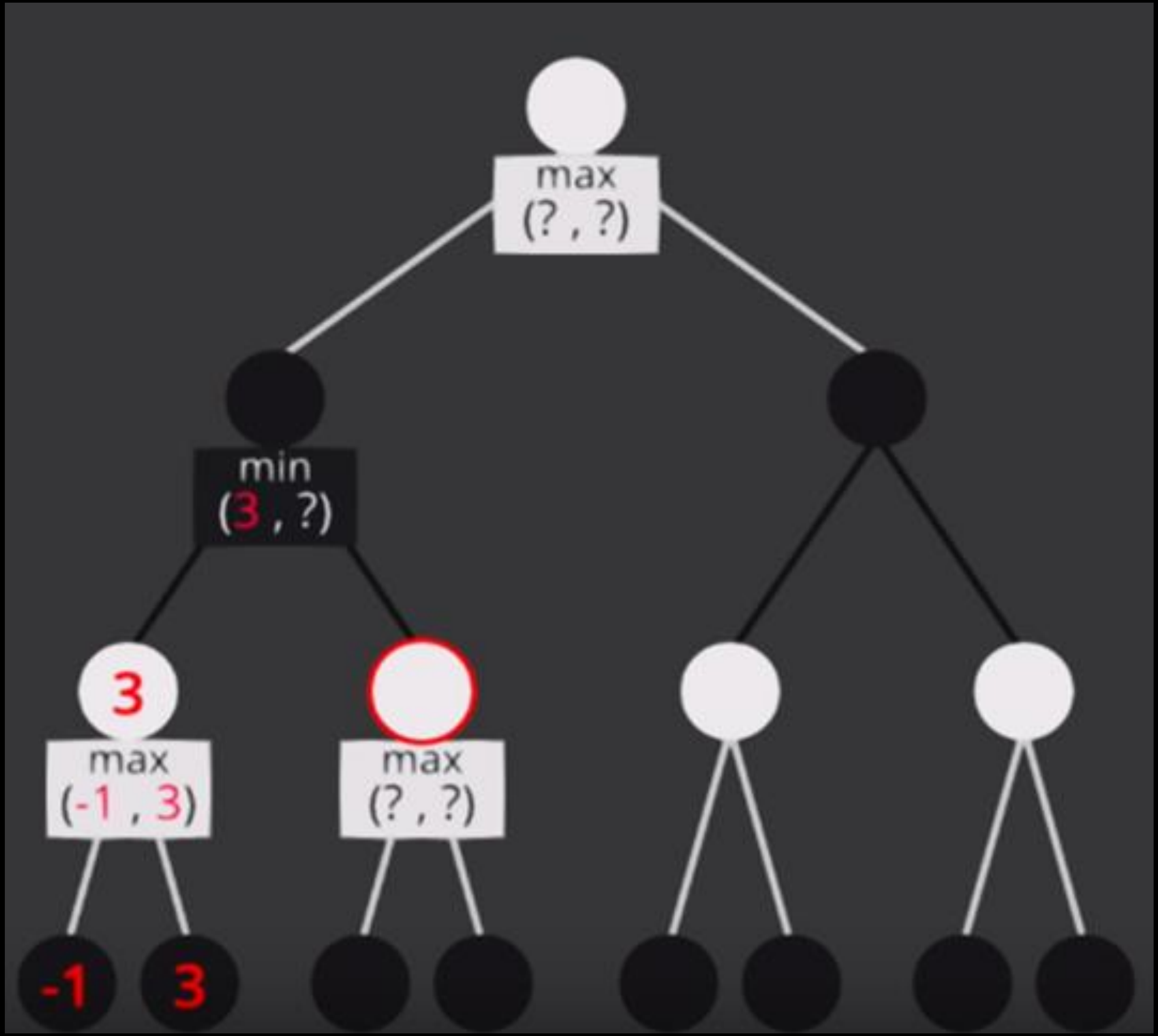

max
(? , ?)
min
(? , ?)

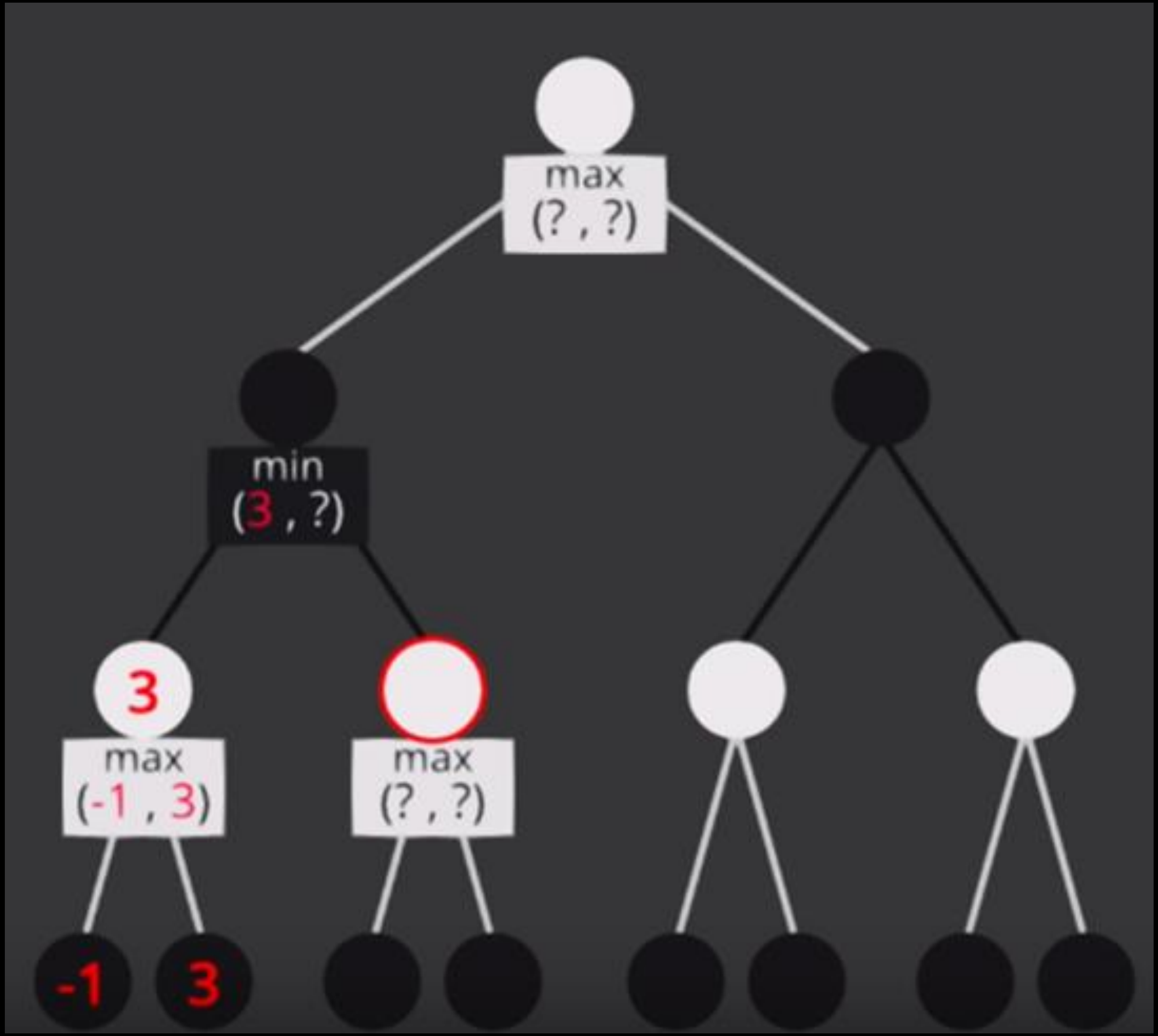

max
(? , ?)
min
(? , ?)
max
(? , ?)


max
(? , ?)
min
(? , ?)
max
(? , ?)

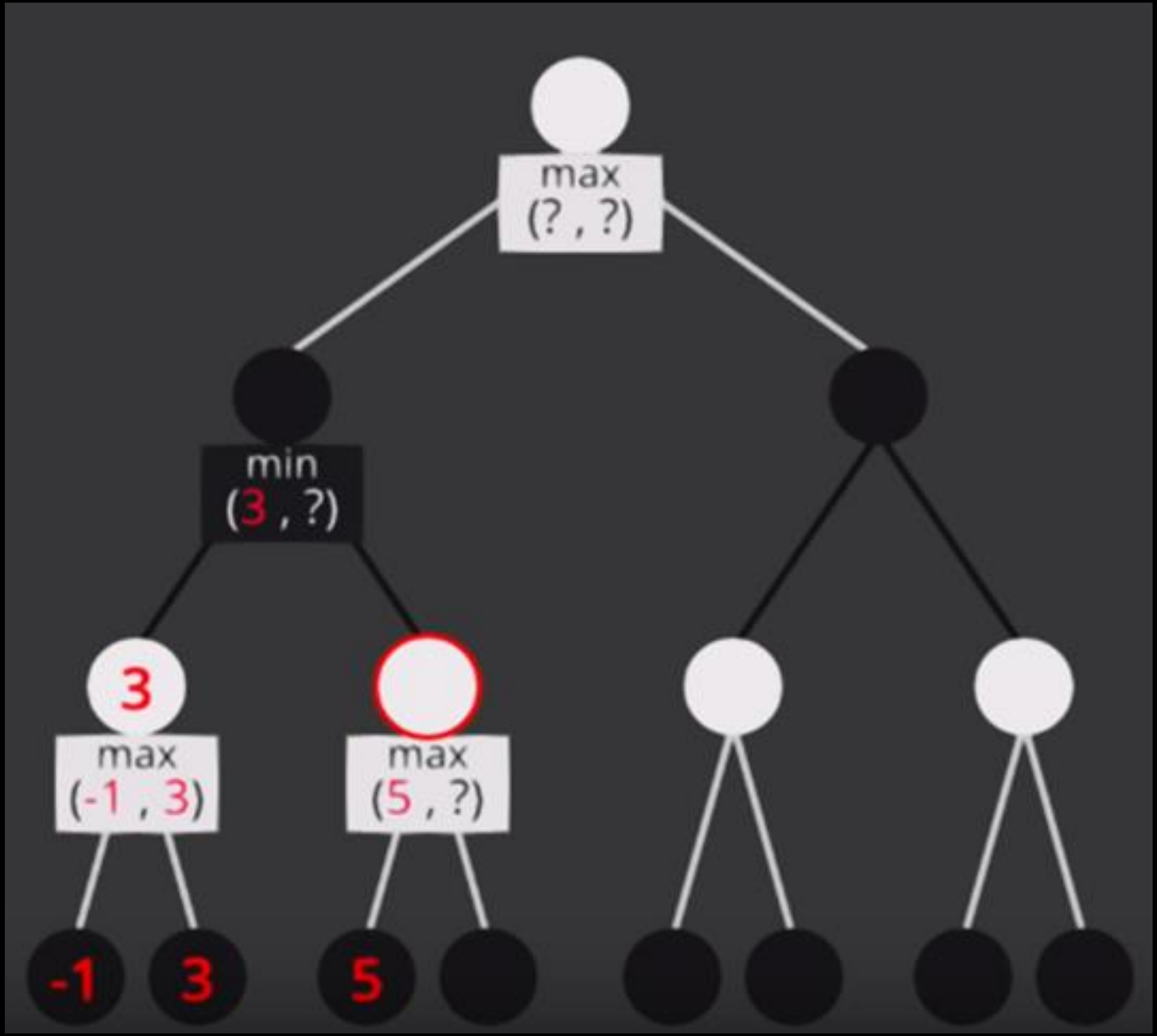

max
(? , ?)
min
(? , ?)
max
(? , ?)
-1

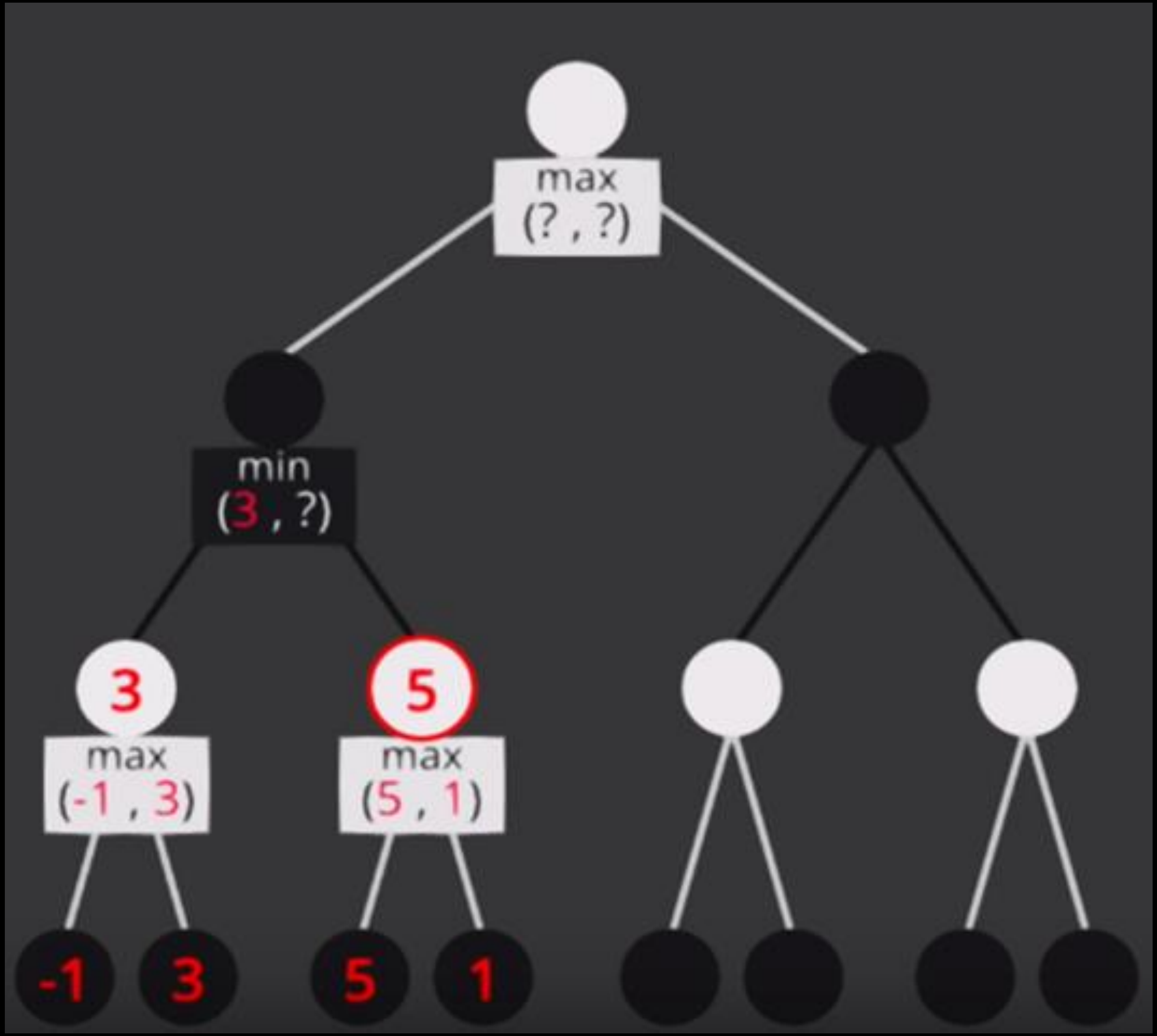

max
(? , ?)
min
(? , ?)
3
max
(-1 , 3)
-1
3

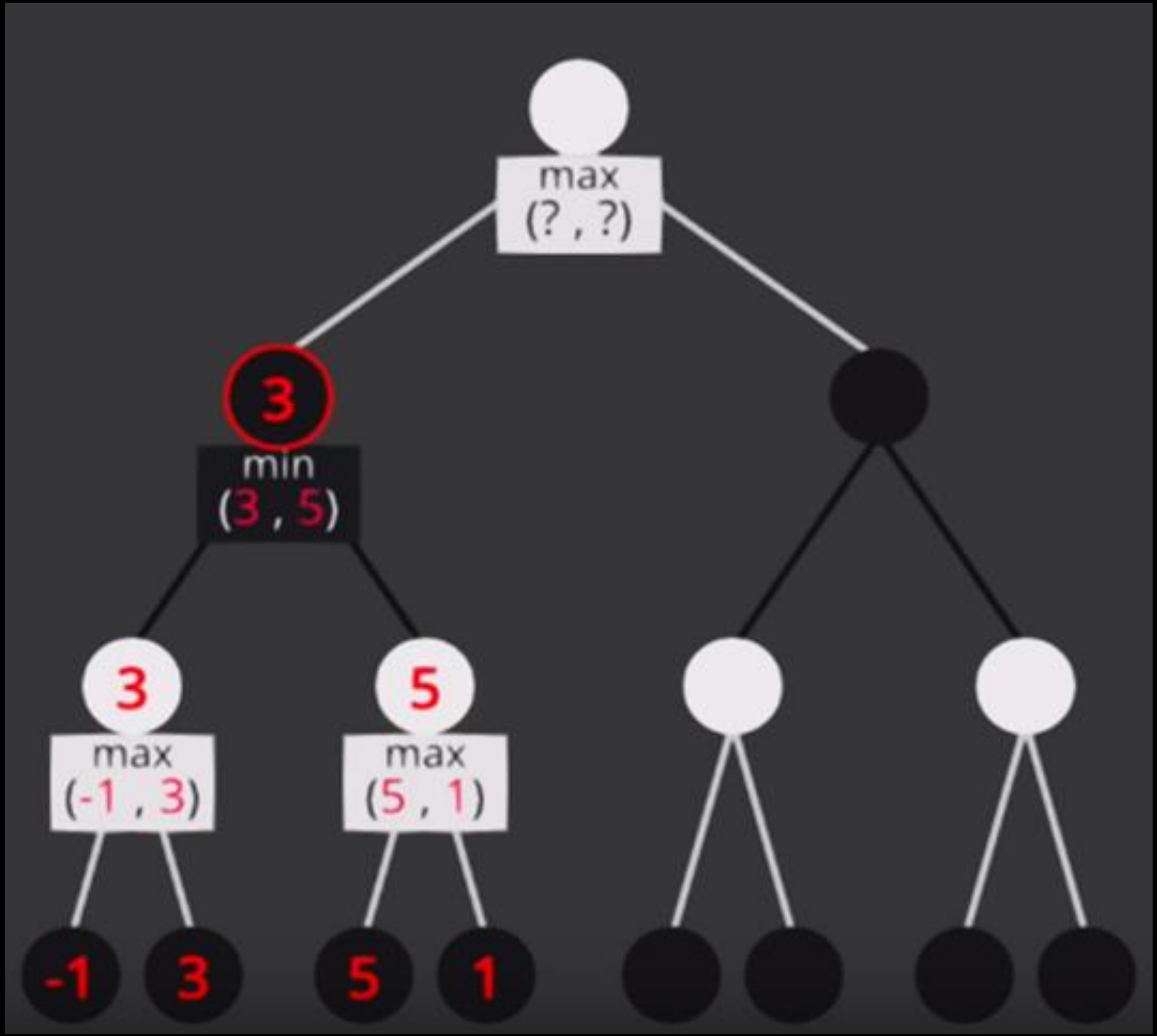

max
(? , ?)
min
(3 , ?)
3
max
(-1 , 3)
max
(? , ?)
-1
3

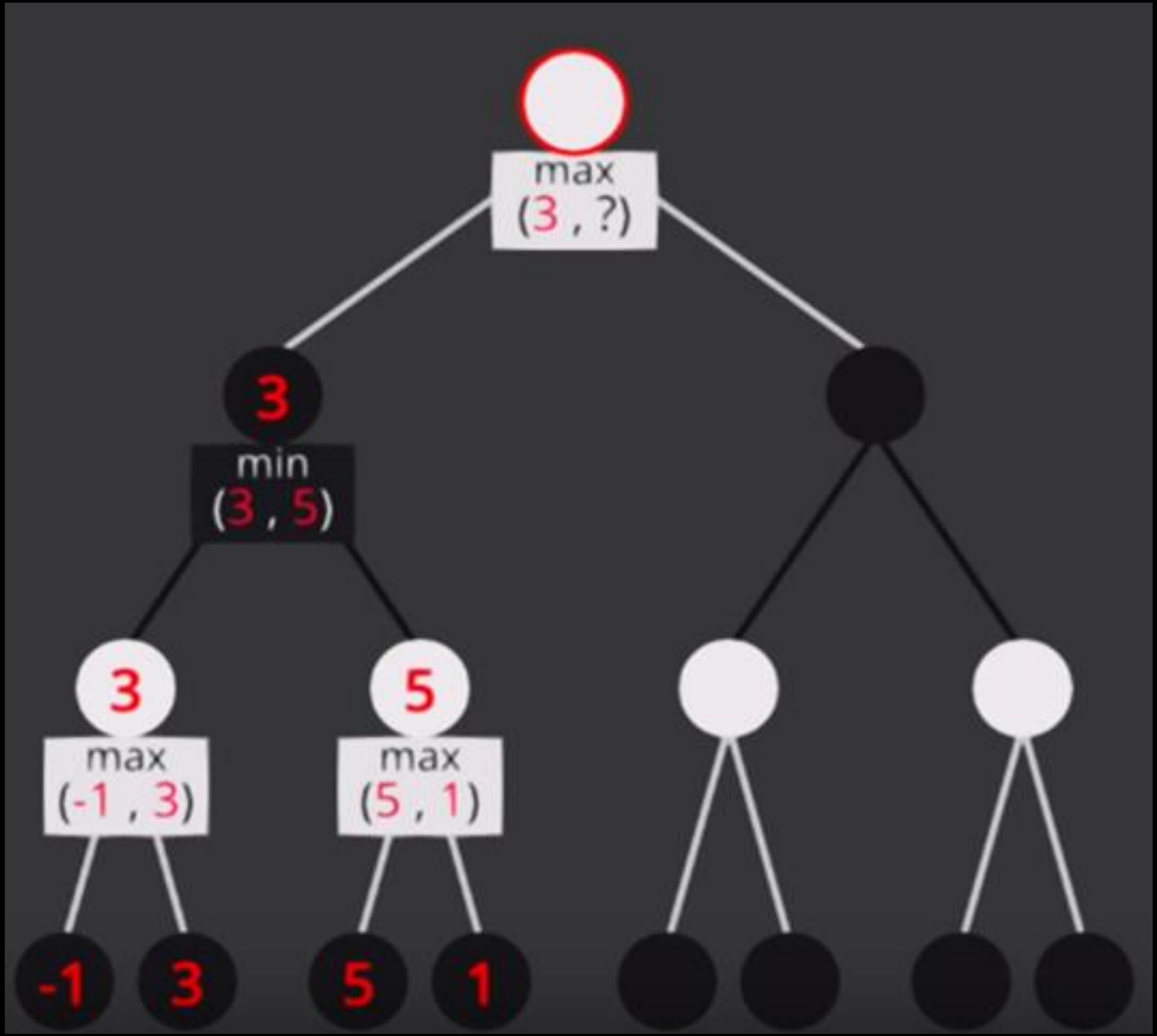


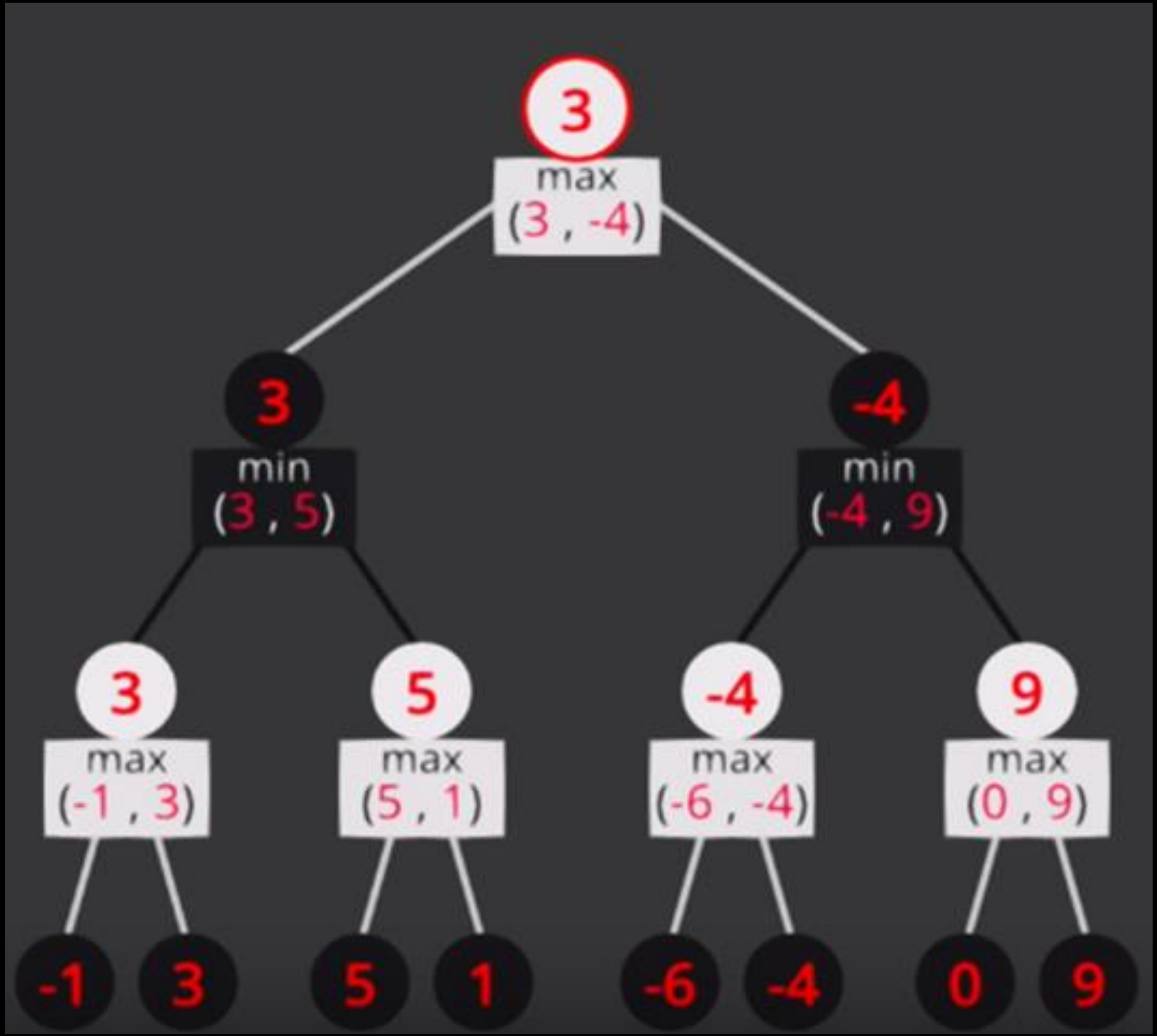
max
(? , ?)
min
(3 , ?)
3
max
(-1 , 3)
max
(? , ?)
-1
3


max
(? , ?)
min
(3 , ?)
3
max
(-1 , 3)
max
(5 , ?)
-1
3
5


max
(? , ?)
min
(3 , ?)
3
max
(-1 , 3)
5
max
(5 , 1)
-1
3
5
1

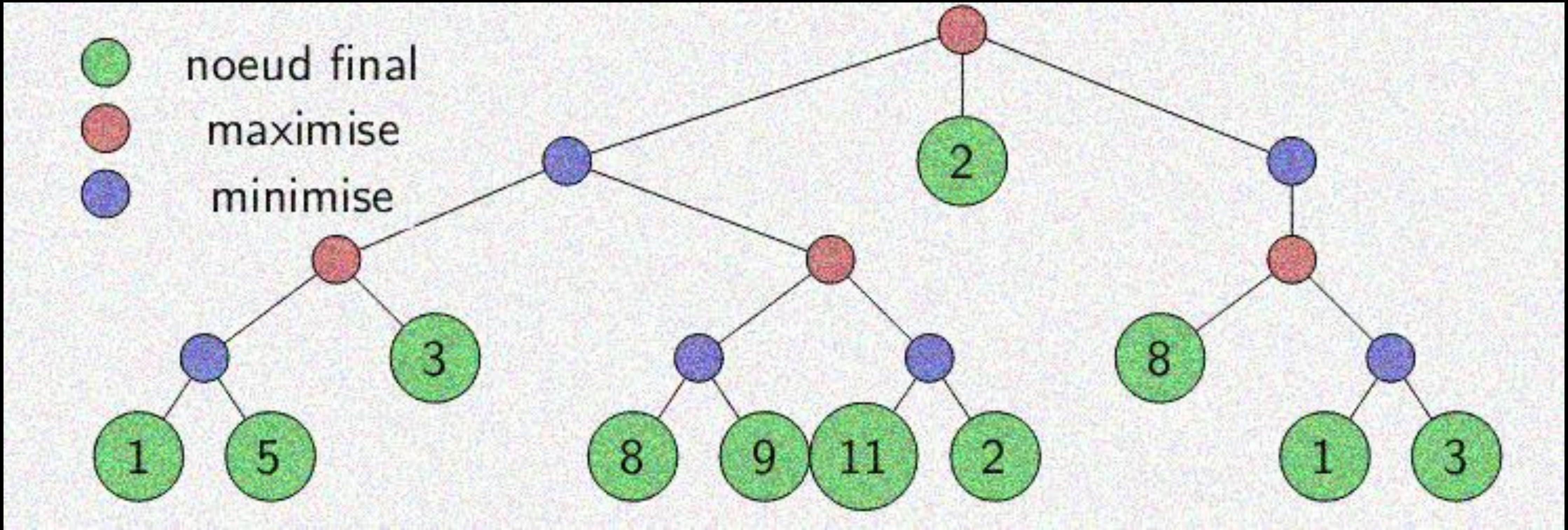

max
(? , ?)
3
min
(3 , 5)
3
max
(-1 , 3)
5
max
(5 , 1)
-1
3
5
1


max
(3 , ?)
3
min
(3 , 5)
3
max
(-1 , 3)
5
max
(5 , 1)
-1
3
5
1

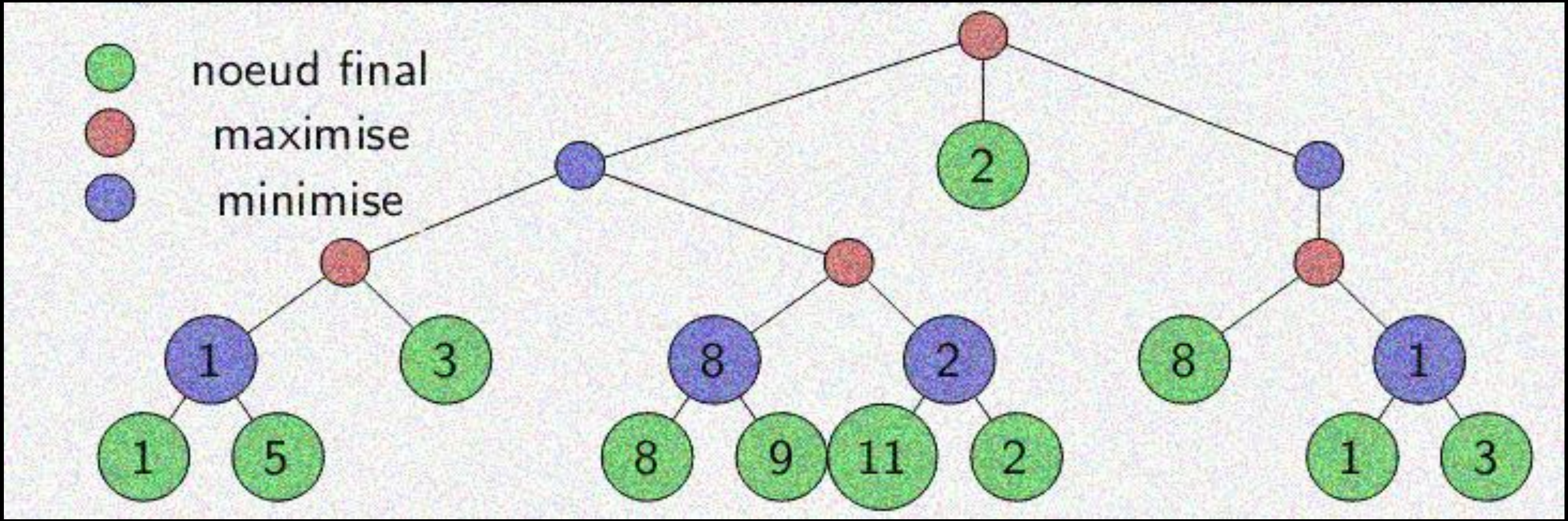


3
max
(3 , -4)
3
min
(3 , 5)
-4
min
(-4 , 9)
3
max
(-1 , 3)
5
max
(5 , 1)
-4
max
(-6 , -4)
9
max
(0 , 9)
-1
3
5
1
-6
-4
0
9

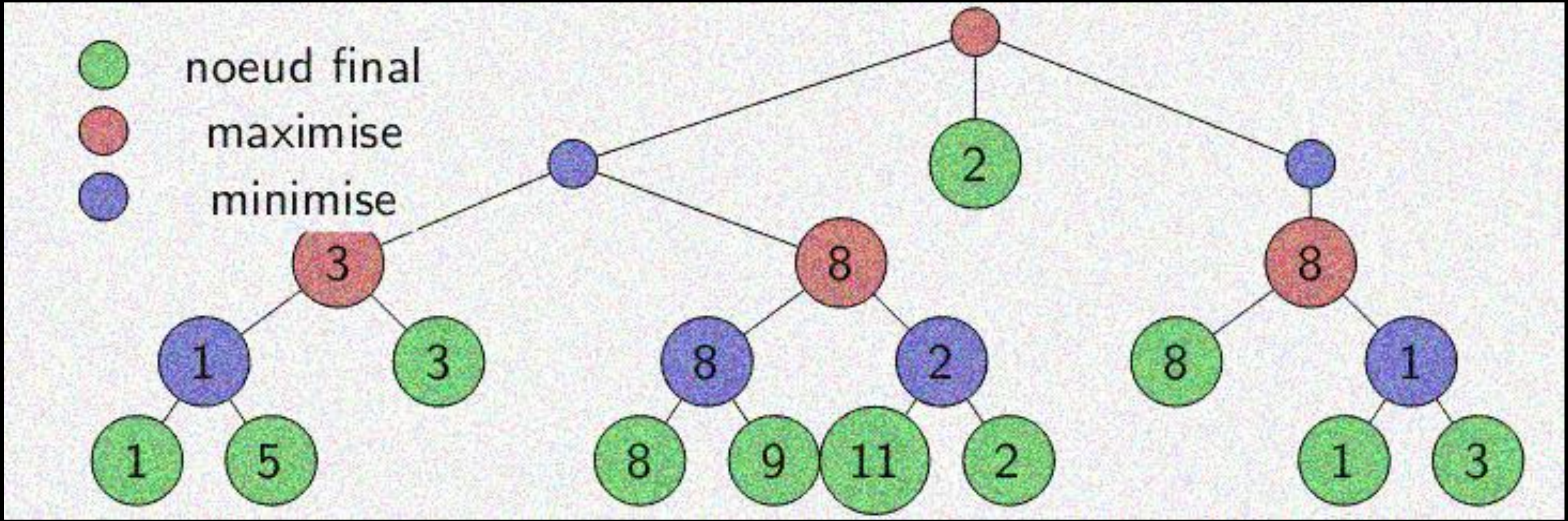
Exemple 2



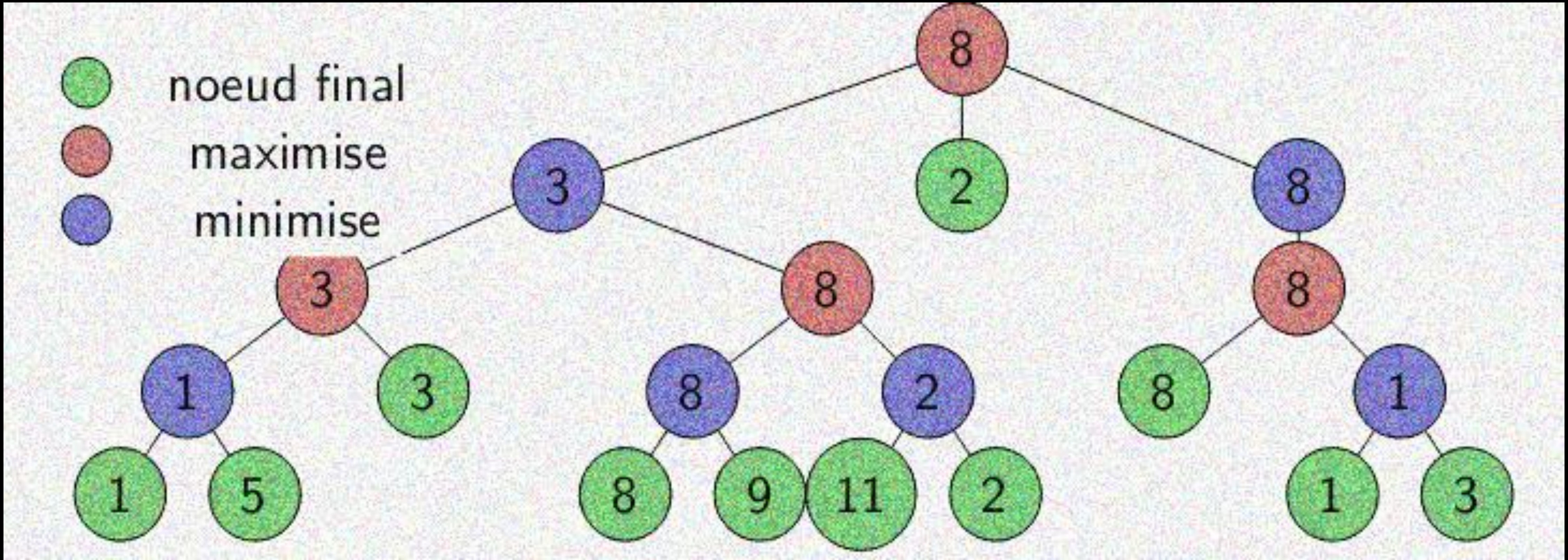
Exemple 2



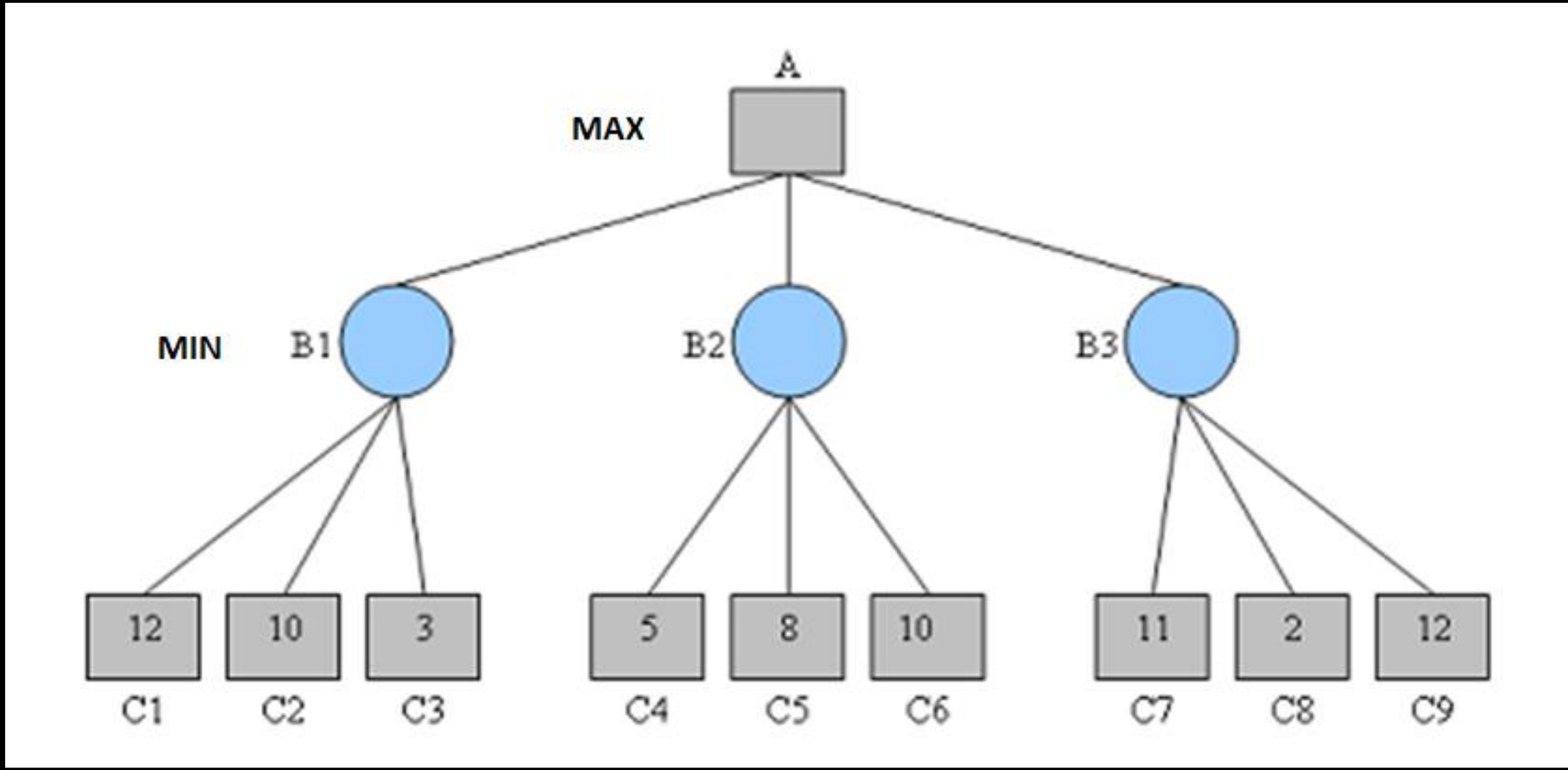
Exemple 2



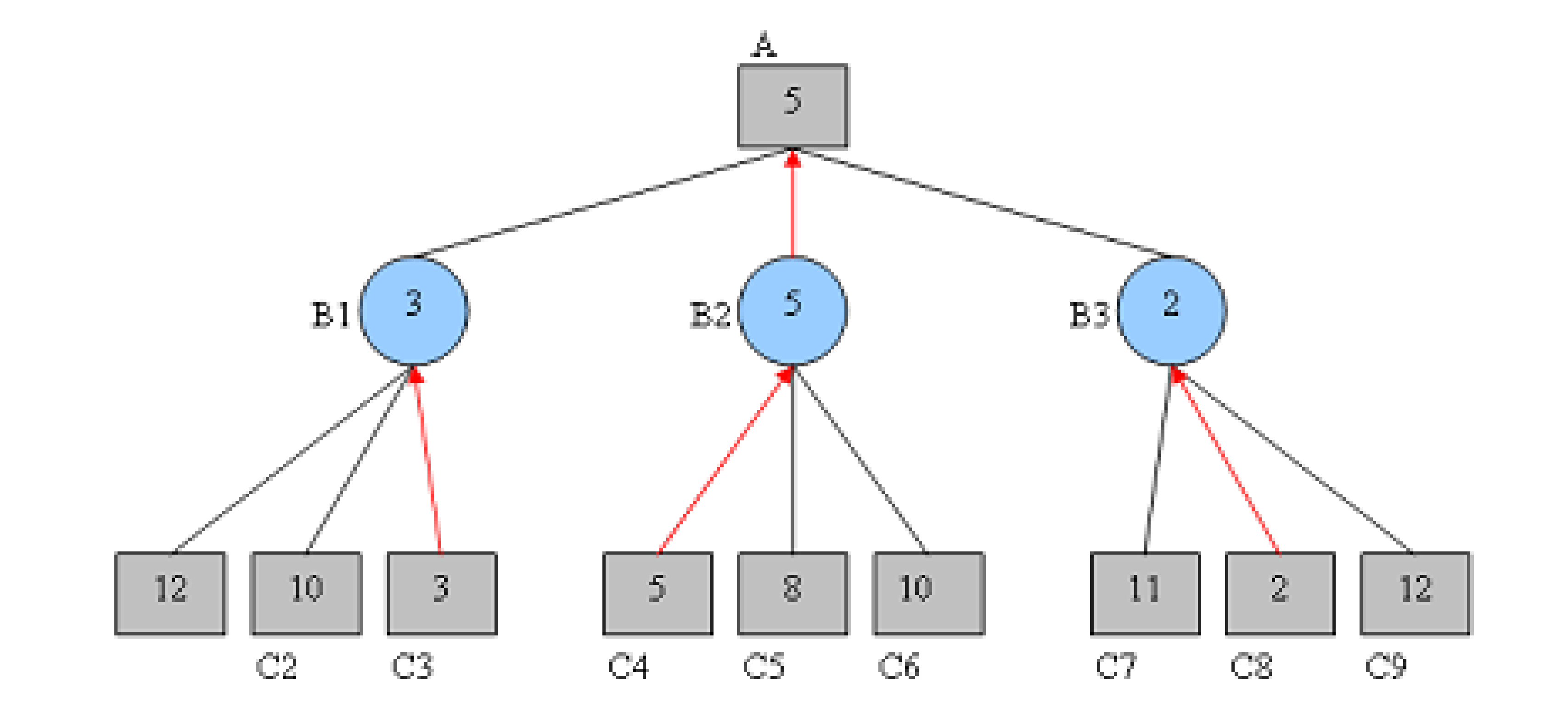
Exemple 2



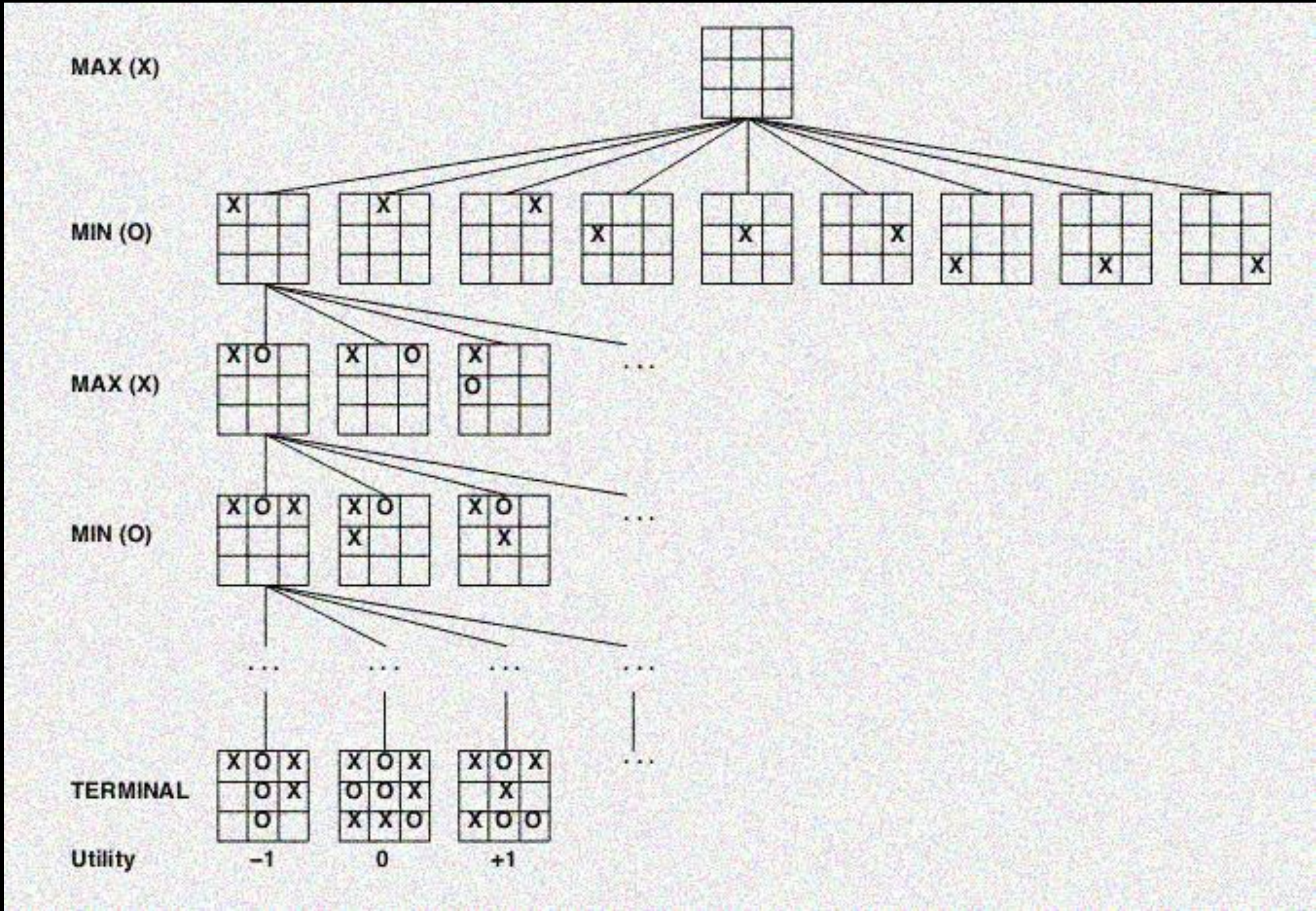
Exemple 3



Exemple 3



tic-tac-toe



Algorithme de recherche Minimax

- Complet, si l'arbre est fini
- Optimal si l'adversaire est optimal

Si b est le nombre maximal d'actions possibles et m est la profondeur maximale de l'arbre

- Complexité en temps : $O(b^m)$
- Complexité en espace : $O(bm)$
  - Pour les échecs par exemple : b ~ 35, m ~ 100 $\Rightarrow$ solution exacte impossible.

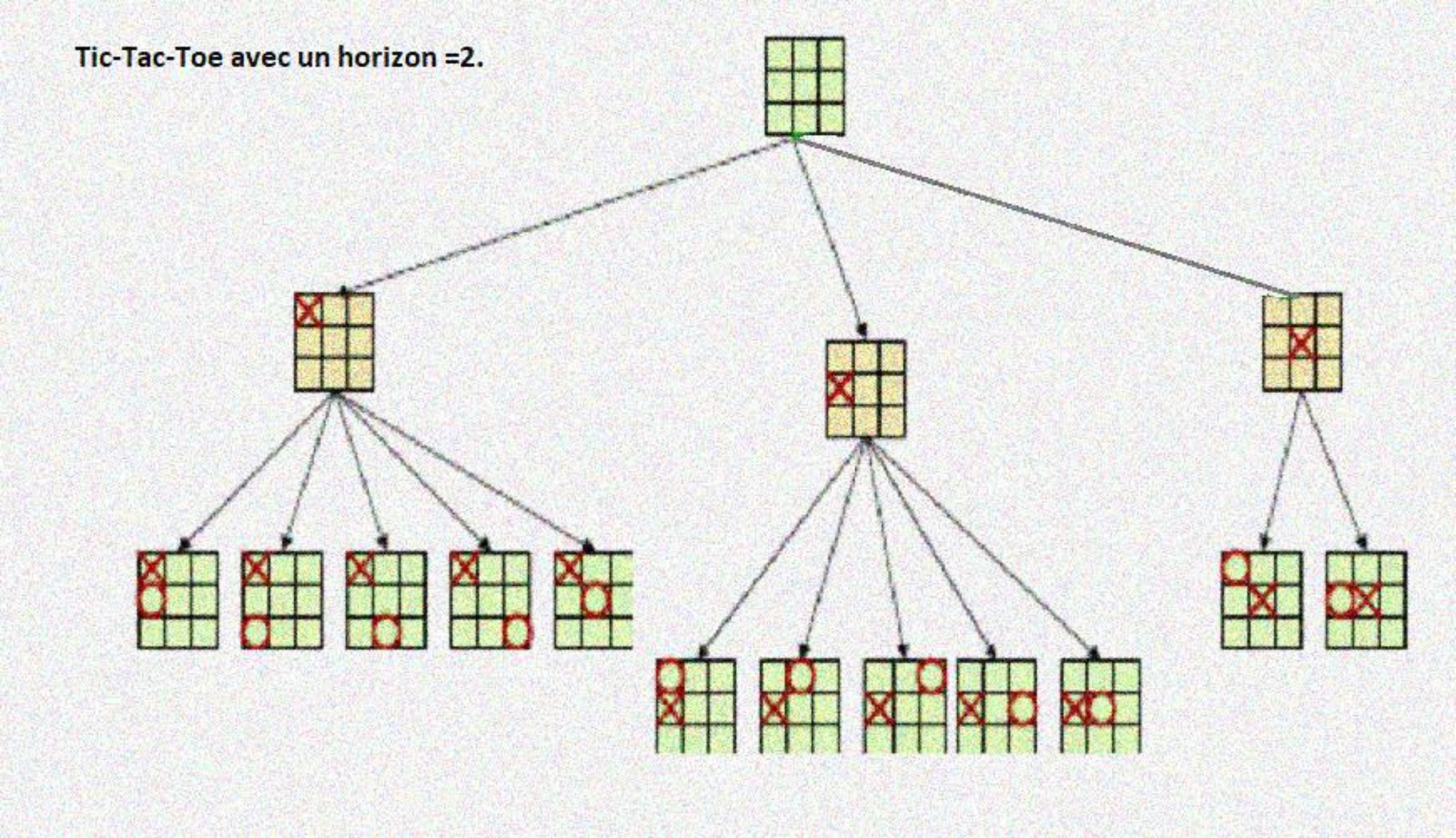
Comment accélérer la recherche ?

- Deux approches
  - La première introduit une approximation
  - la deuxième maintient l'exactitude de la solution
- Couper la recherche et remplacer l'utilité par une fonction d'évaluation heuristique
  - ✓ **Idée** : faire une recherche la plus profonde possible en fonction du temps à notre disposition et tenter de prédire le résultat de la parte si on n'arrive pas à la fin.
- Élagage alpha-bêta (alpha-beta pruning)
  - ✓ **Idée** : identifier des chemins dans l'arbre qui sont explorés inutilement.

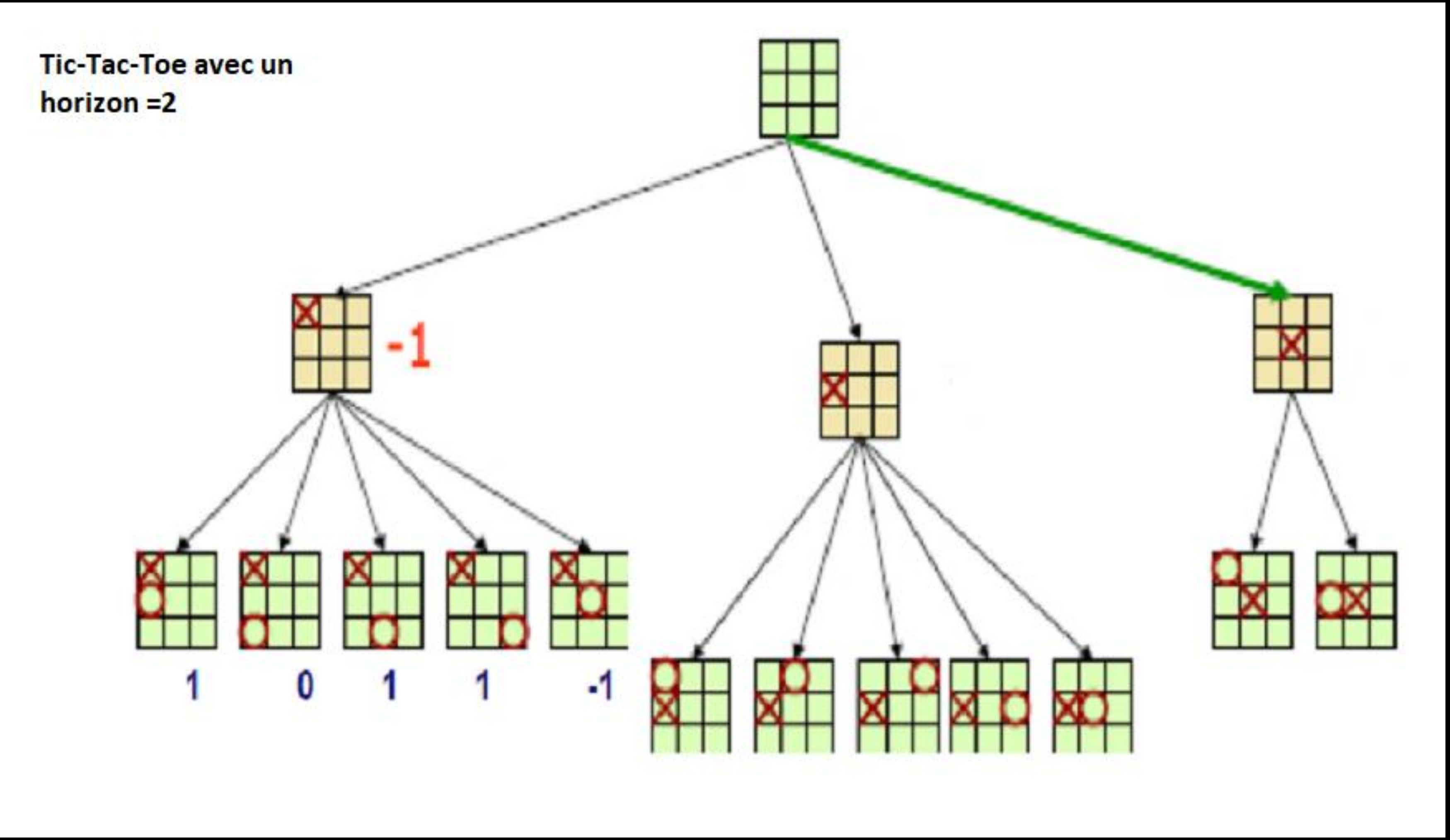
MiniMax avec profondeur limitée

- Principe
  - Etendre l'arbre de jeu jusqu'à une profondeur N à partir du nœud courant.
  - Calculer la valeur de la fonction d'évaluation pour chaque nœud feuille, pas forcément terminal.
  - Propager ces valeurs.

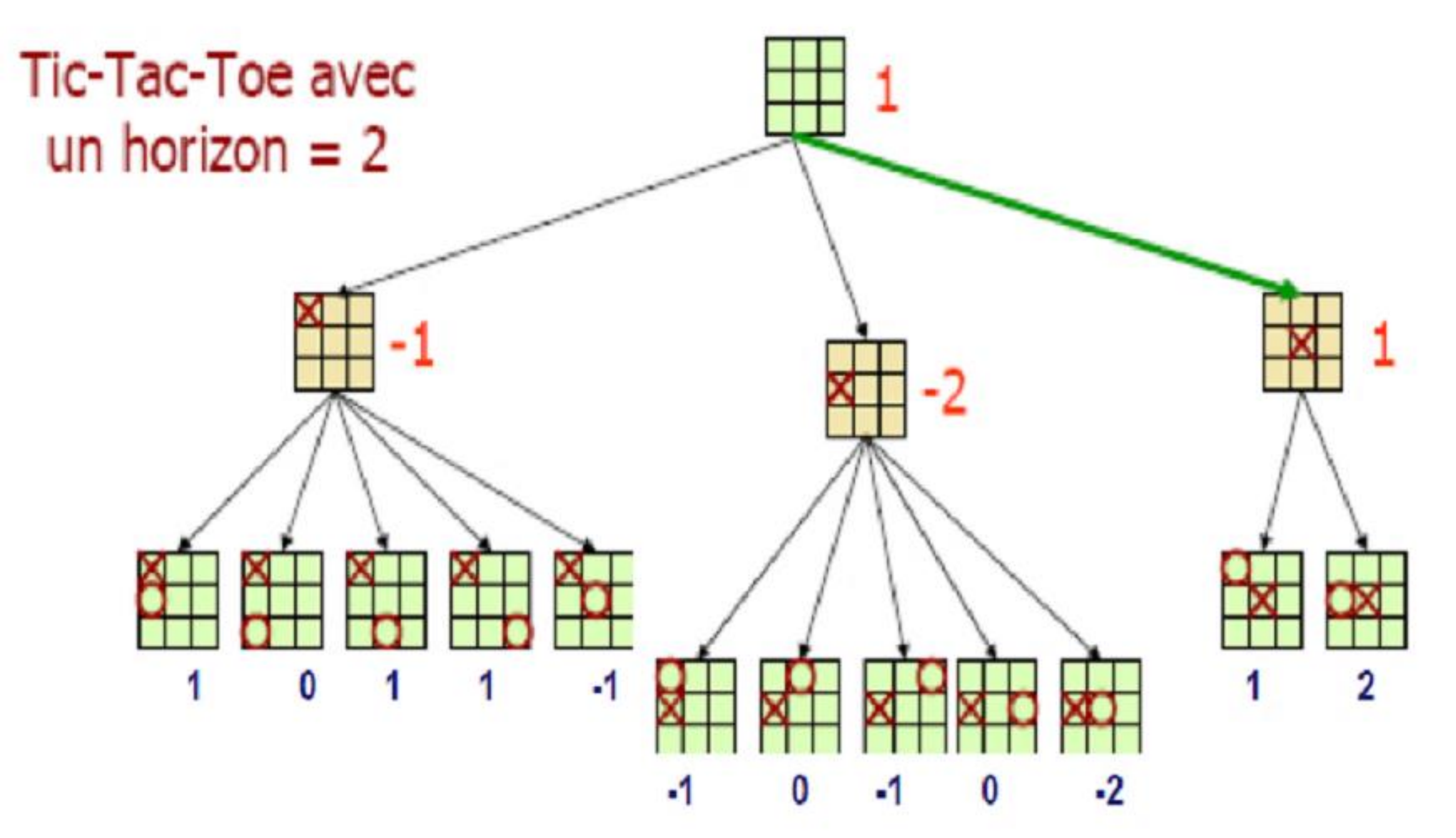
Exemple :MinMax avec profondeur limitée



Exemple :MiniMax avec profondeur limitée



Exemple :MiniMax avec profondeur limitée



L’élagage $\alpha - \beta$

Principe :

- Etendre l'arbre de jeu jusqu'à une profondeur h par une recherche en profondeur.
- Ne plus générer les successeurs d'un nœud dès qu'il est évident que ce nœud ne sera pas choisi, compte tenu des nœuds déjà examinés

L’élagage $\alpha - \beta$

Principe :

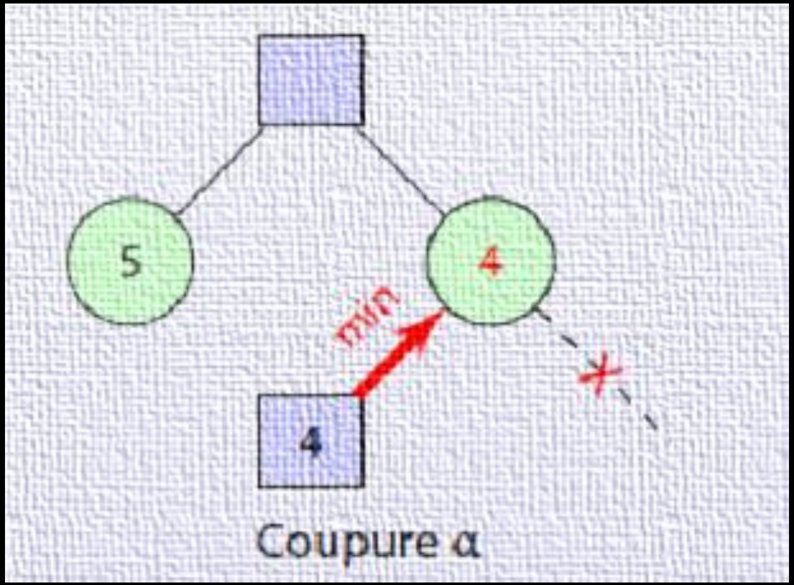
Valeurs initiales : $\alpha = -\infty \; et \; \beta = +\infty$

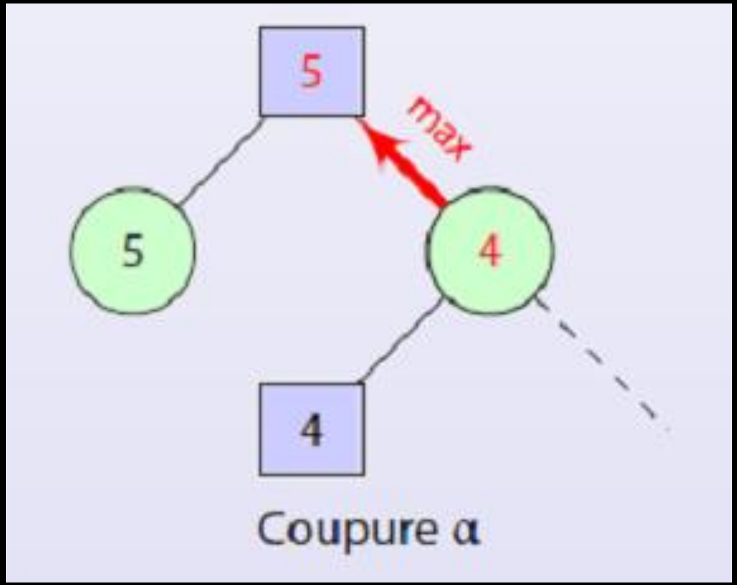
- ❑ Chaque nœud MAX conserve la trace d’une valeur α, qui est la valeur de son meilleur successeur trouvé jusqu'à présent.
- ❑ Chaque nœud MIN conserve la trace d’une valeur β, qui est la valeur de son pire successeur trouvé jusqu'à présent.

L’élagage $\alpha - \beta$

Deux règles :

1. Coupure α

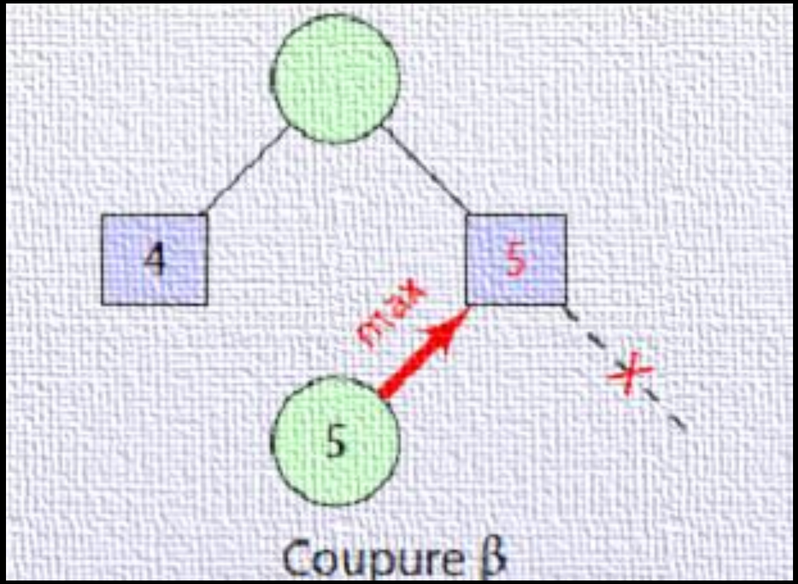


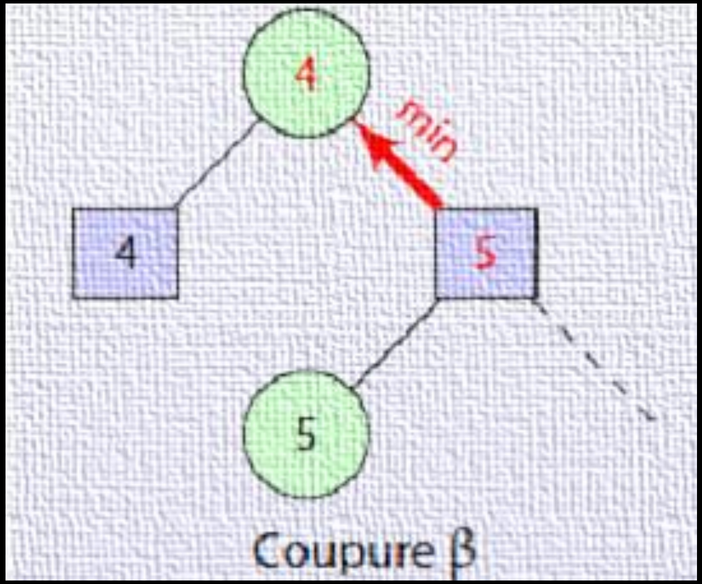


L'élagage $\alpha - \beta$

Deux règles :

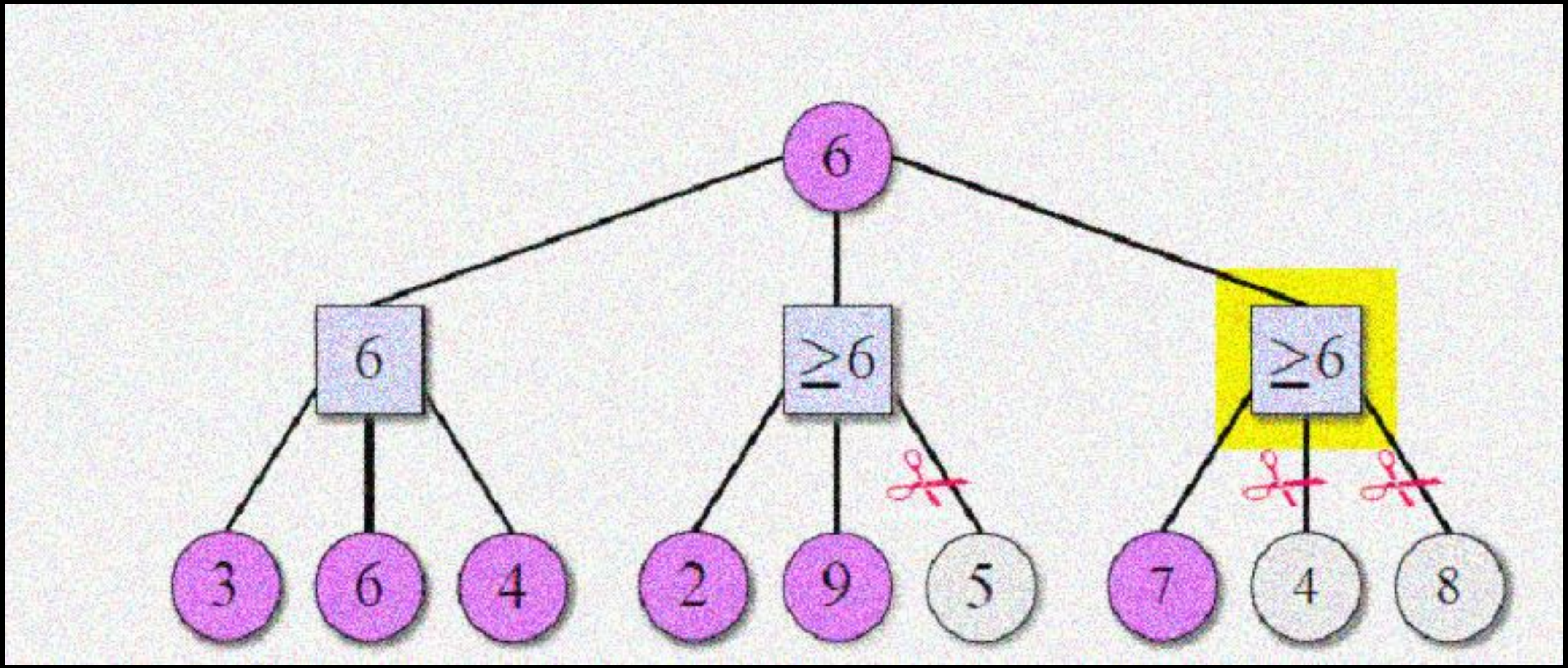
1. Coupure β



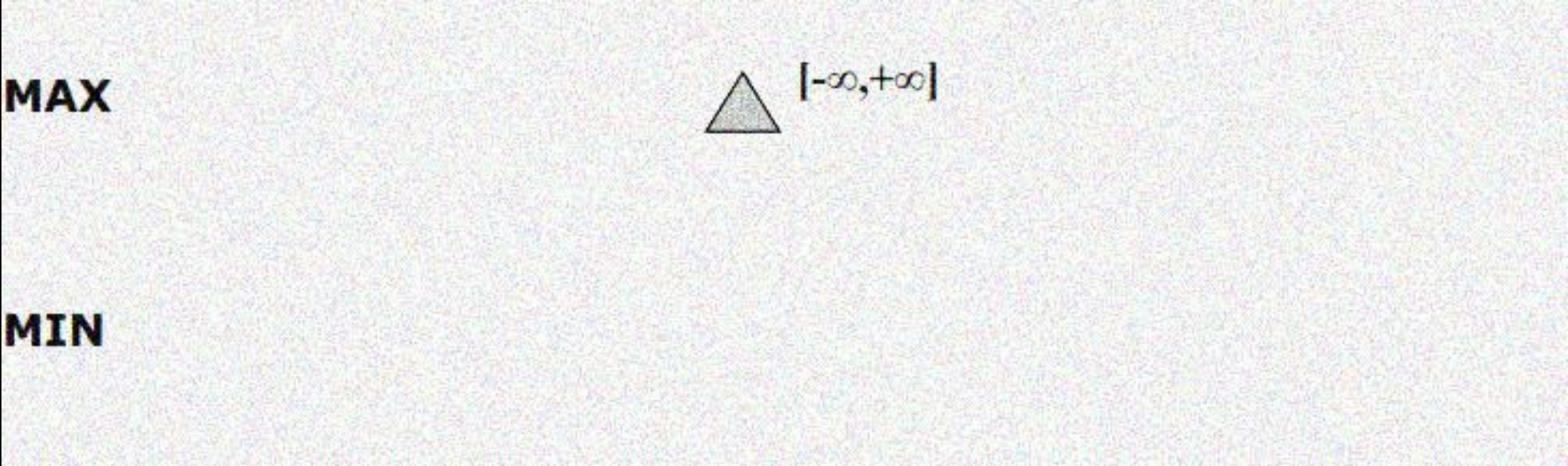


L’élagage $\alpha - \beta$

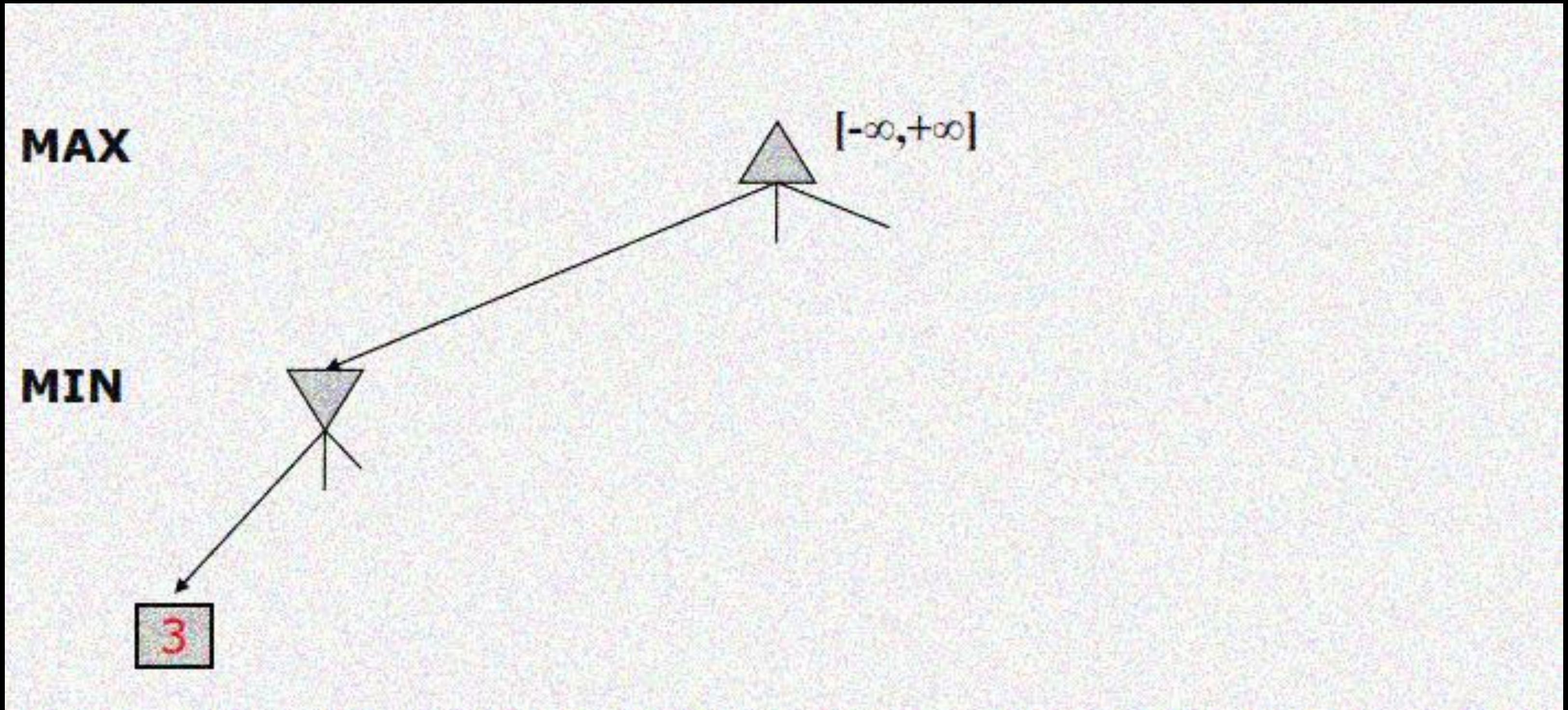
Exemple



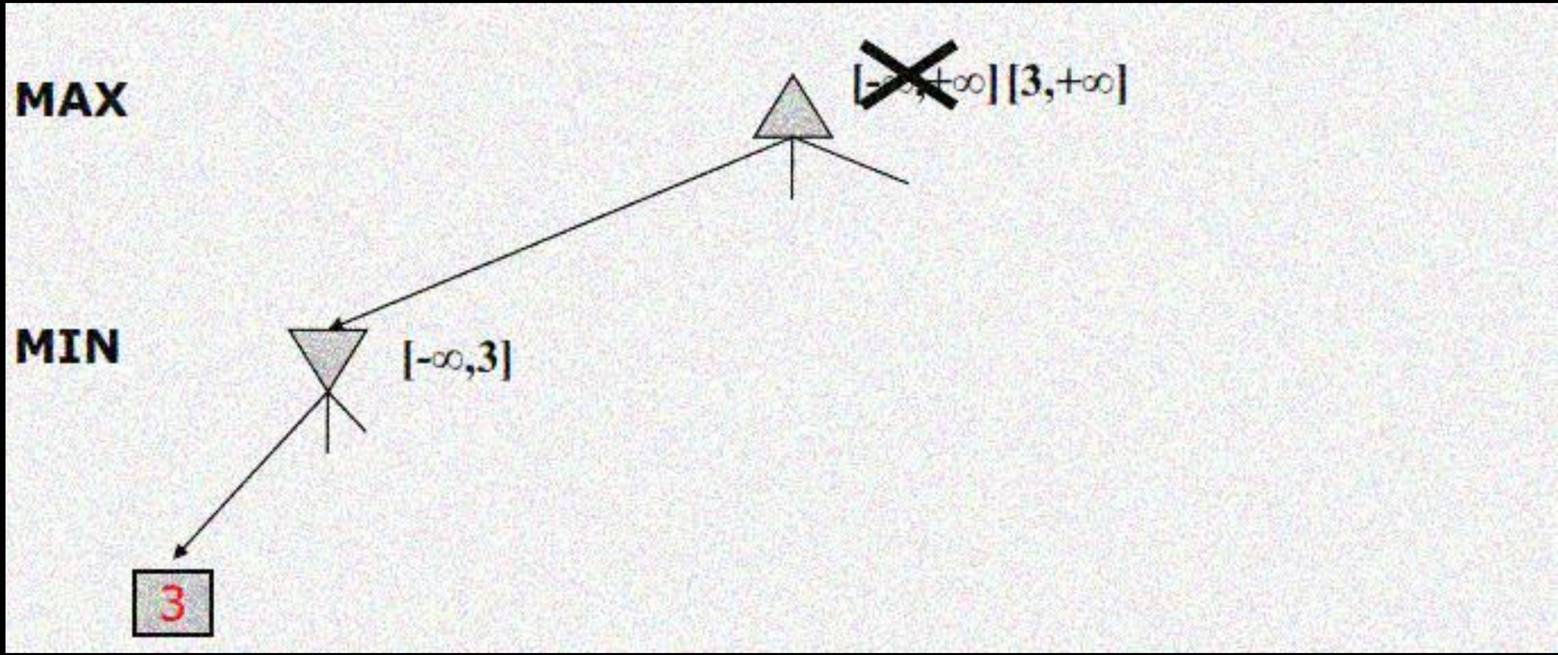
Exemple d'élagage $\alpha - \beta$



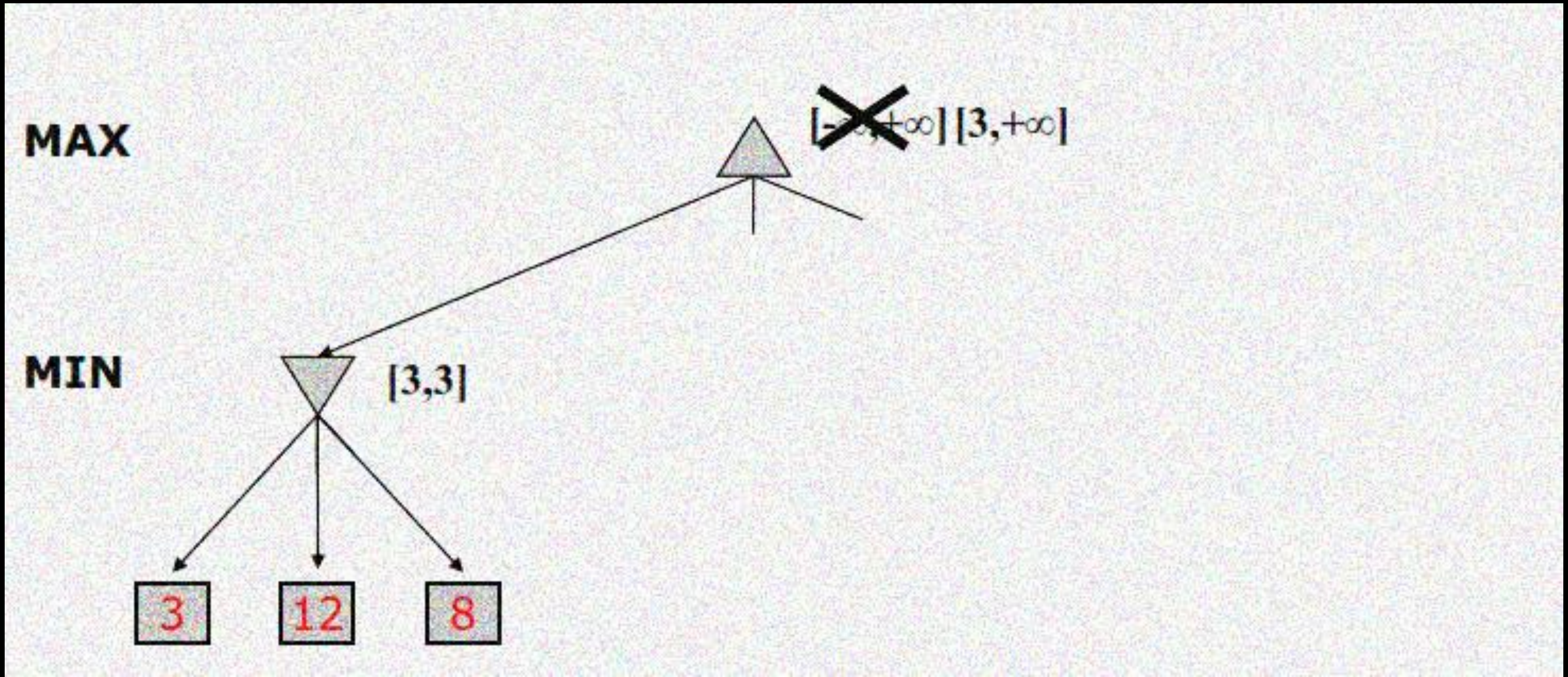
Exemple d'élagage $\alpha - \beta$



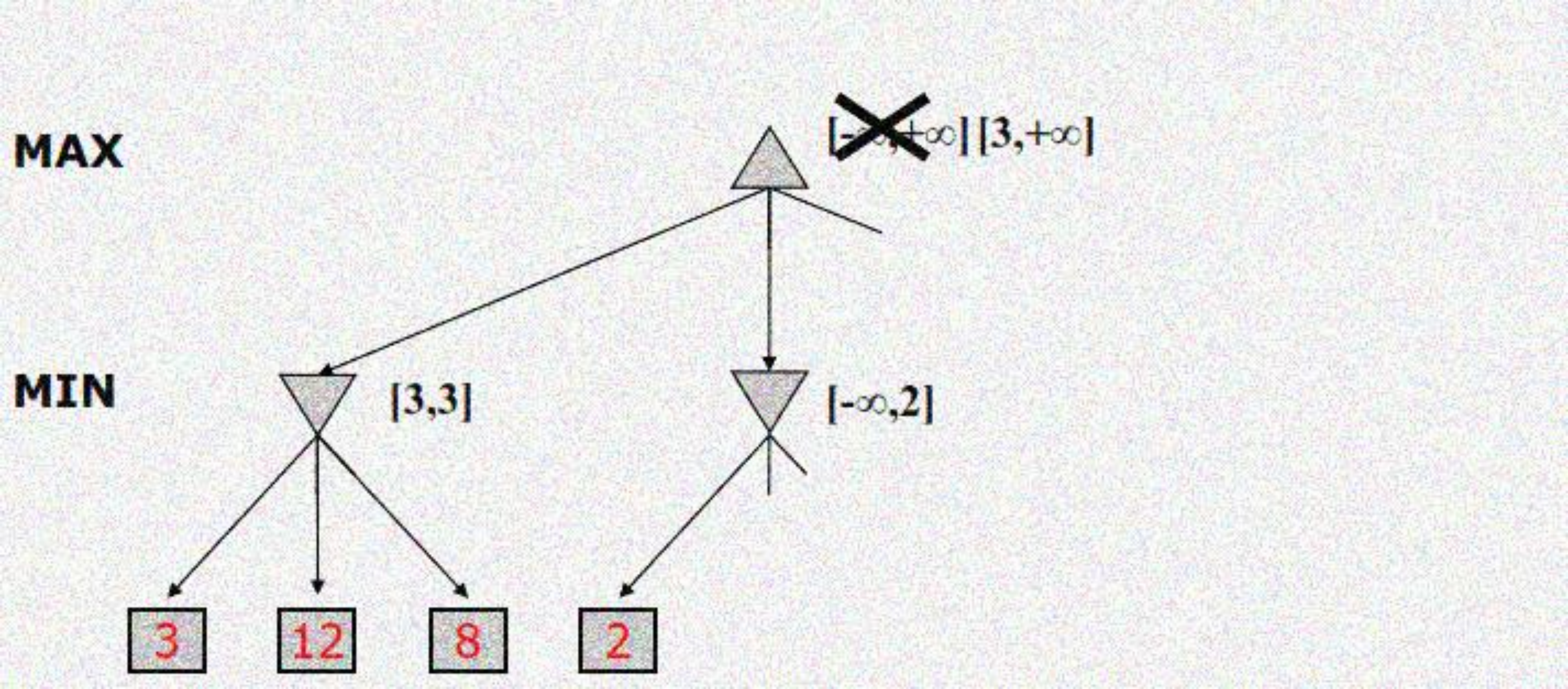
Exemple d'élagage $\alpha - \beta$



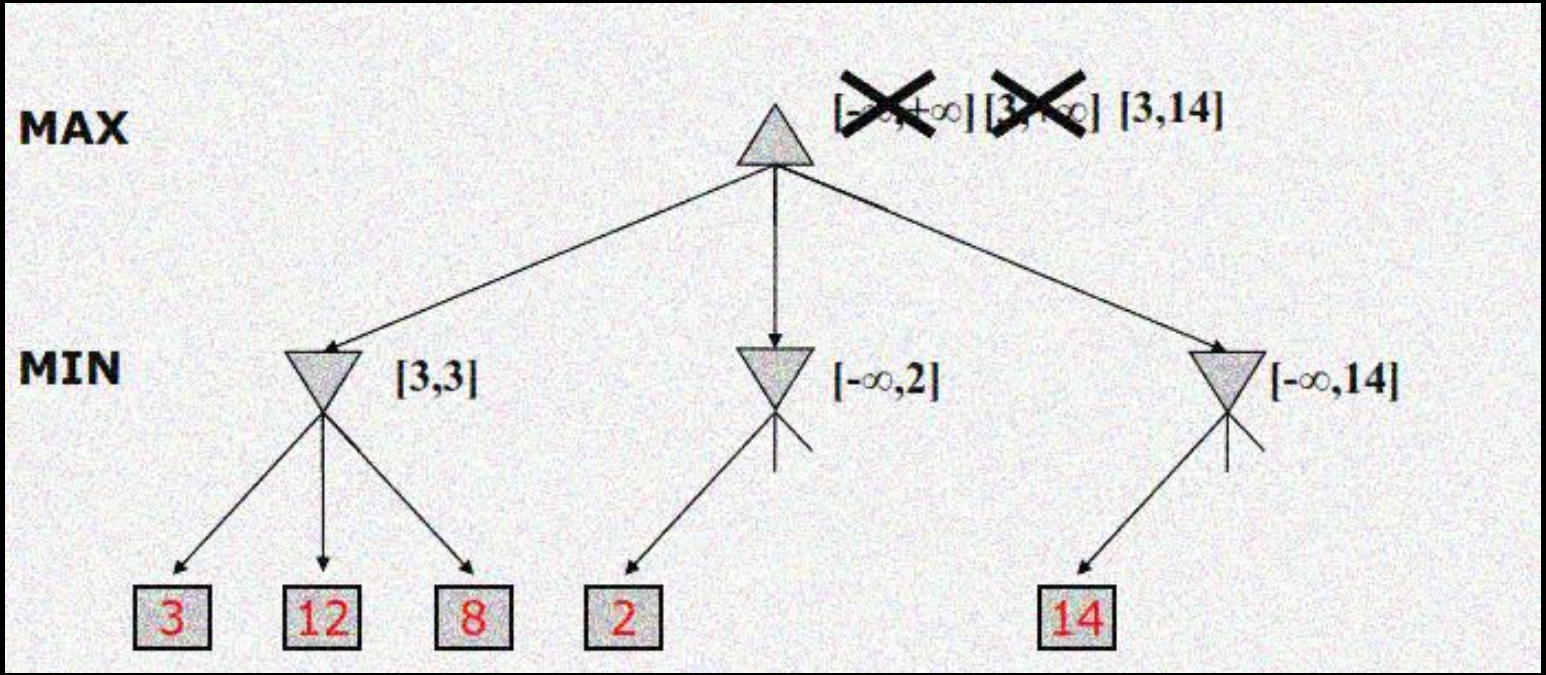
Exemple d'élagage $\alpha - \beta$



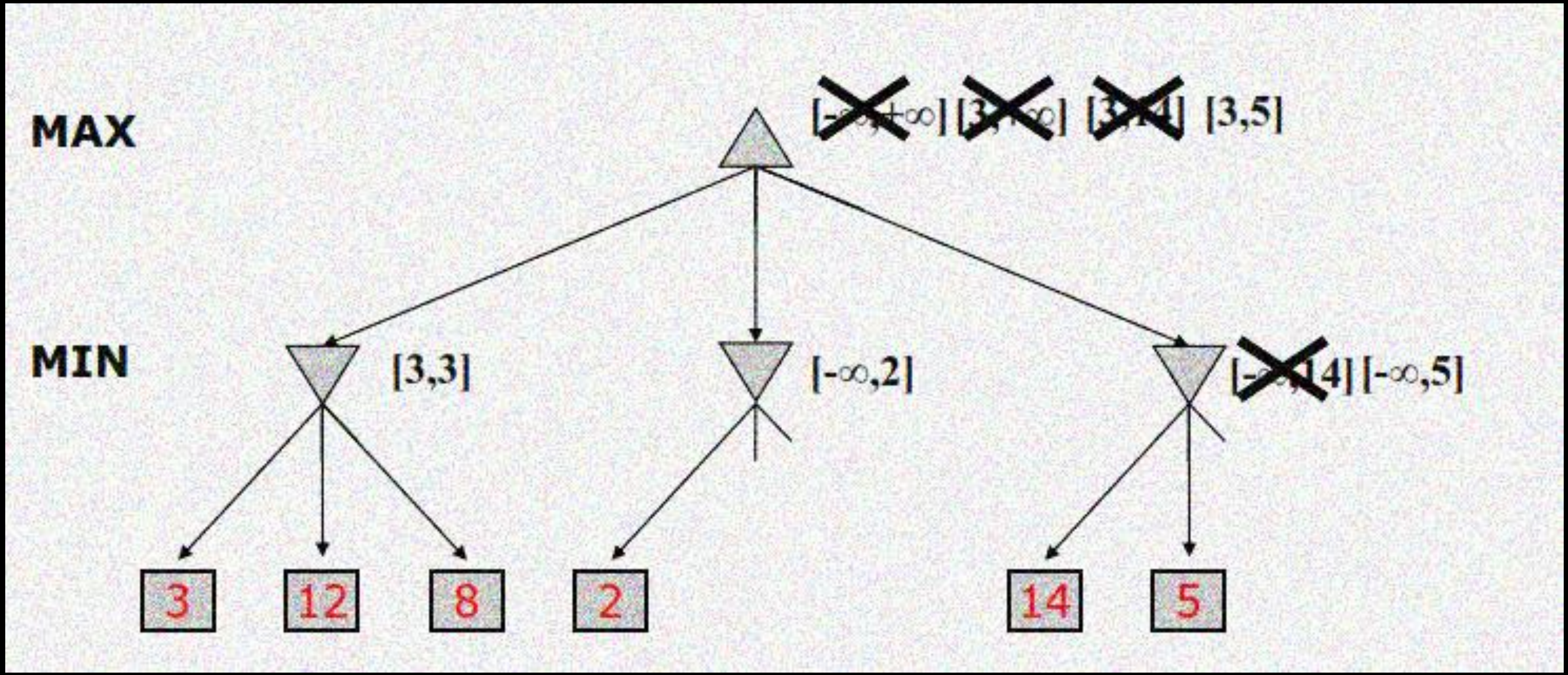
Exemple d'élagage $\alpha - \beta$



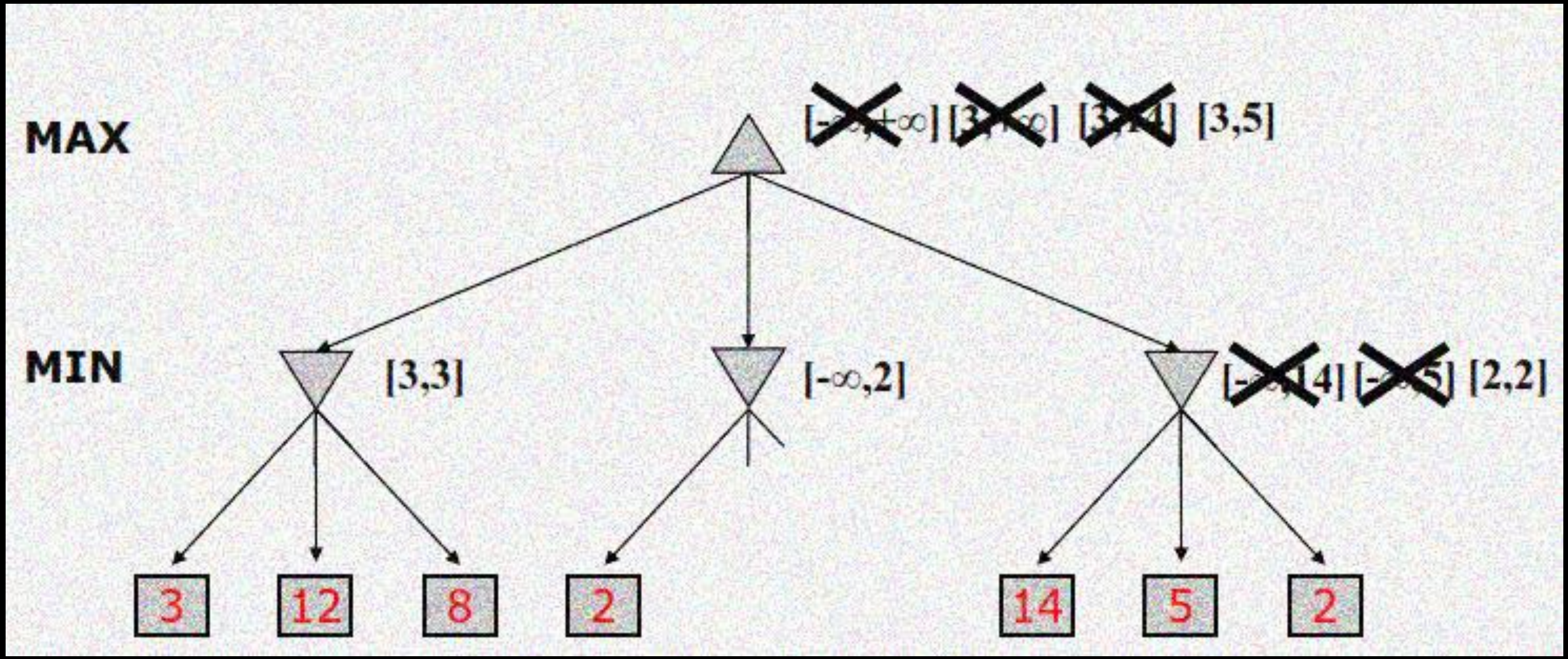
Exemple d'élagage $\alpha - \beta$



Exemple d'élagage $\alpha - \beta$



Exemple d'élagage $\alpha - \beta$



$$
\begin{aligned}
\text{MINIMAX}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}
$$

élagage $\alpha - \beta$

- α est la meilleure valeur (la plus grande) pour MAX trouvée jusqu‘à présent en dehors du chemin actuel
  - Si V est pire que α, MAX va l‘éviter → élaguer la branche
- β est définie similairement pour MIN : β est la meilleur valeur (la plus petite) pour MIN jusqu'a présent.
  - Si V est plus grand que β, MIN va l‘éviter → élaguer la branche.

Algorithme $\alpha - \beta$

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

---

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

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

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  same as MAX-VALUE but with roles of α, β reversed

Propriétés de $\alpha - \beta$

- L'élagage n'affecte pas le résultat final
- Un bon choix améliore l'efficacité de l'élagage
- Avec un "choix parfait" la complexité en temps est $O(b^{m/2})$
  - double la profondeur de recherche par rapport à Minimax
  - Mais trouver la solution exacte avec une complexité de l'ordre de 35^{50}, pour les échecs, est toujours impossible.