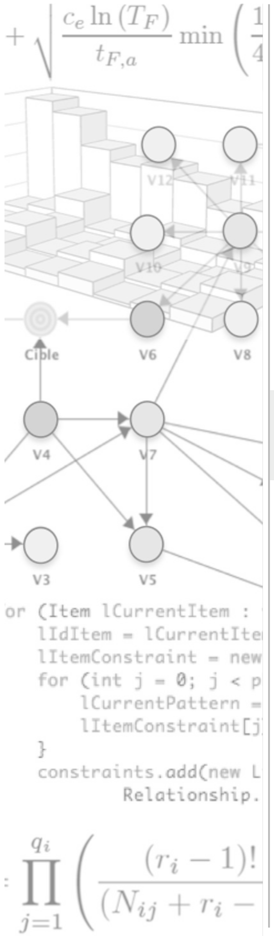
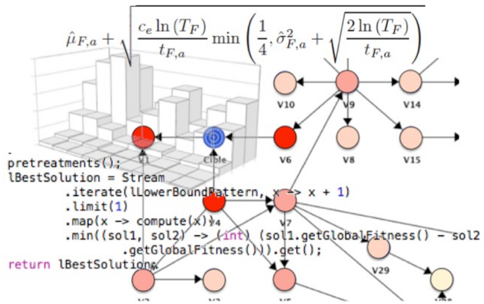


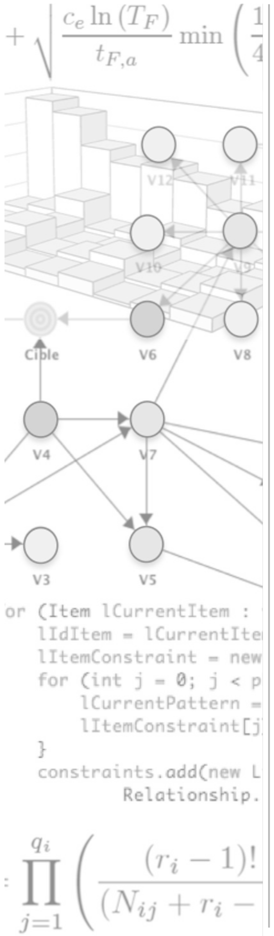
INTELLIGENCE ARTIFICIELLE

Minimax, Alpha-beta, MCTS

Stéphane BONNEVAY

Polytech Lyon
Laboratoire ERIC
stephane.bonnevay@univ-lyon1.fr

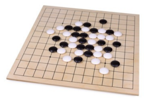




Jeux à deux joueurs

Information complète : chaque joueur connaît ses possibilités d'action, les possibilités d'action de son adversaire, ainsi que les gains résultants des actions

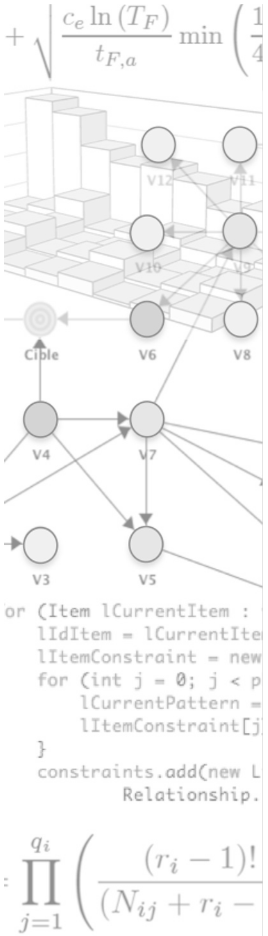




Objectif : implémenter des IA qui joue contre un joueur humain :

- Minimax
- Alpha-beta
- MCTS (Monte Carlo Tree Search)

- AlphaZero → cours « Deep Learning »



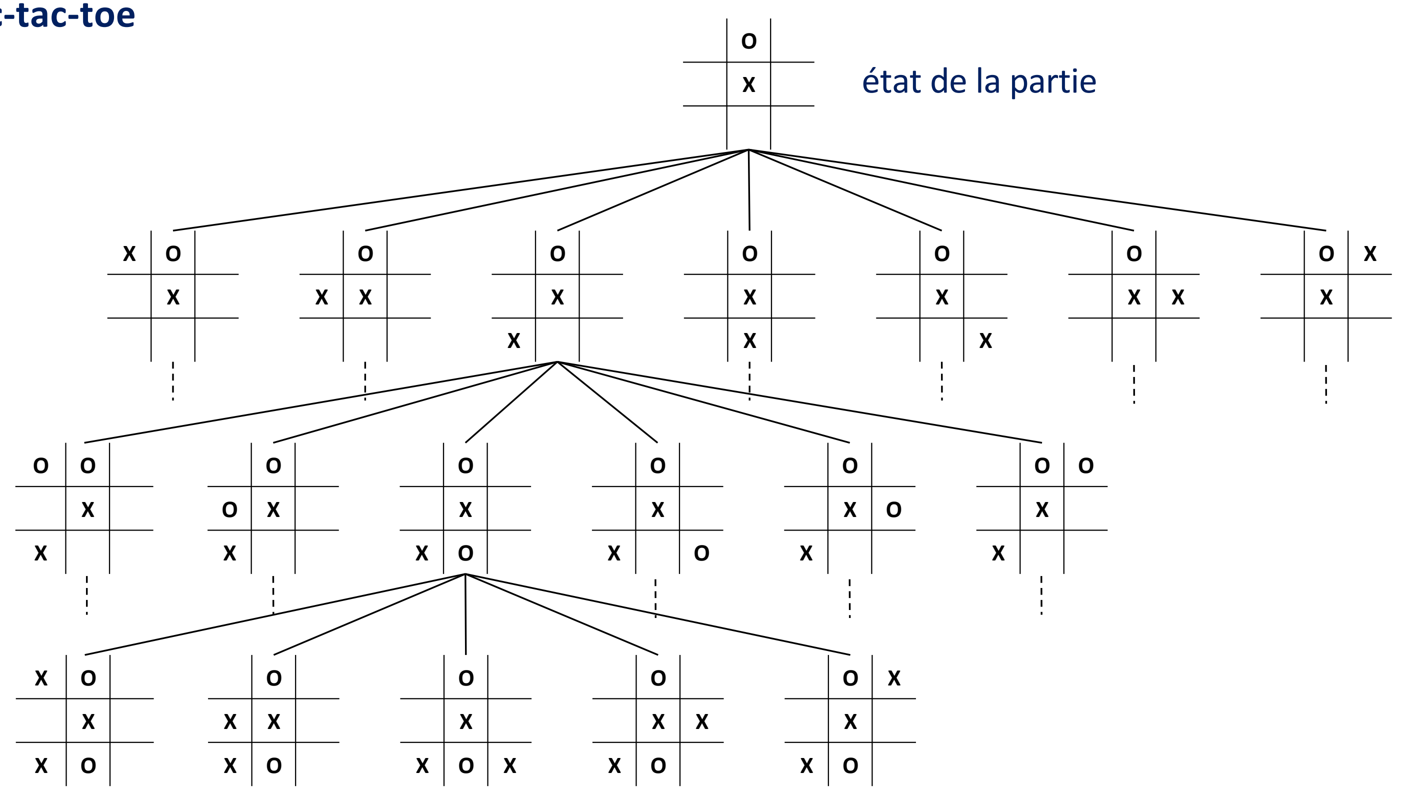
Tic-tac-toe

x joue

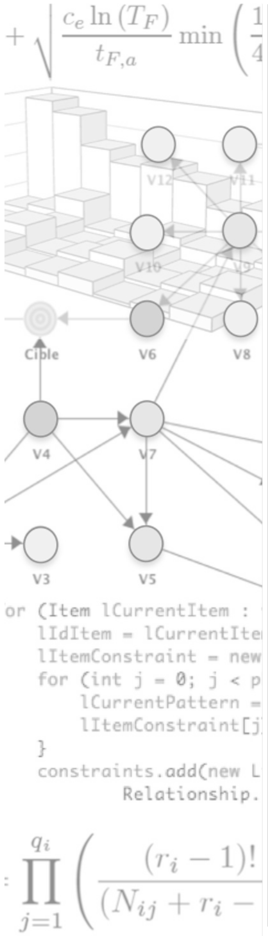
état de la partie

o joue

x joue



Se fixer une profondeur (hauteur de l'arbre) maximum

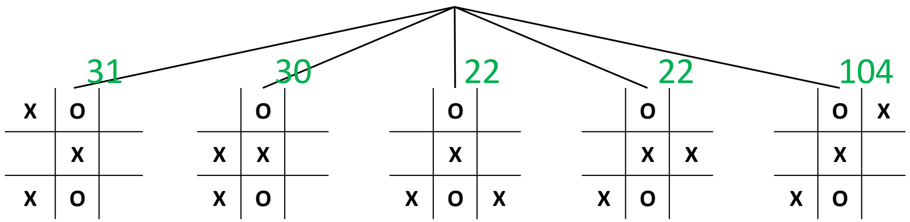


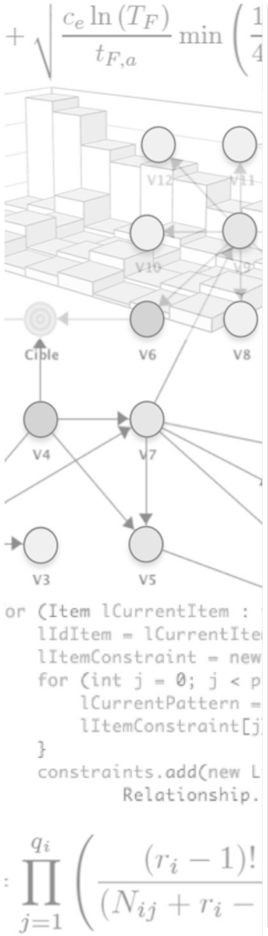
Fonction d’évaluation pour le Tic-tac-toe

Calcul d’un score pour chacune des 3 lignes, 3 colonnes et 2 diagonales :

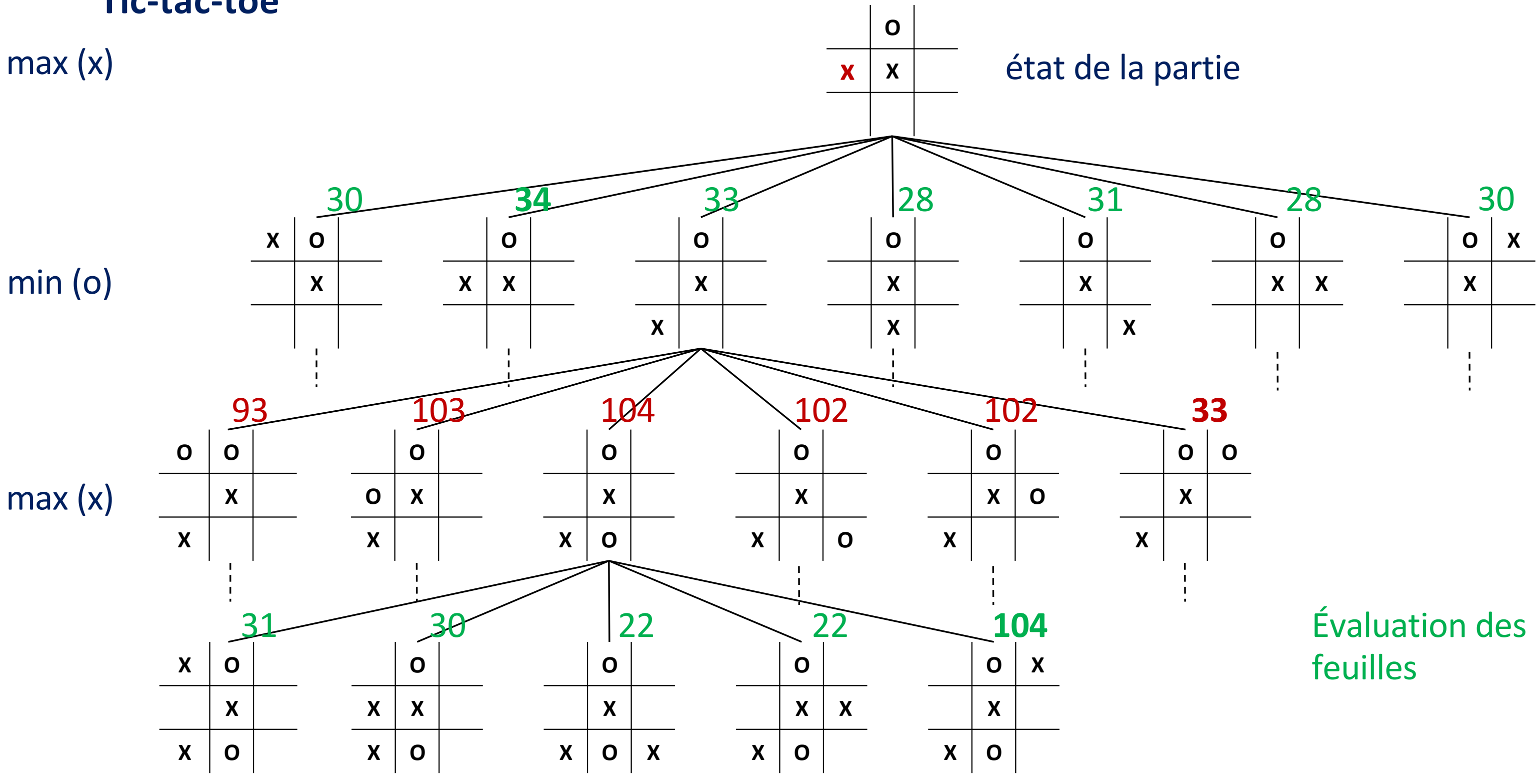
Score = +100	si 3 « x » alignés	(-100 pour « o »)
Score = +10	si 2 « x » alignés et 1 case vide	(-10 pour « o »)
Score = +1	si 1 « x » et 2 cases vides	(-1 pour « o »)

Evaluation = somme des 8 scores

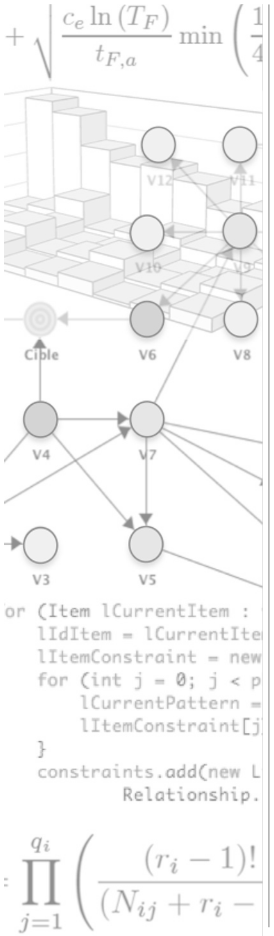




Tic-tac-toe



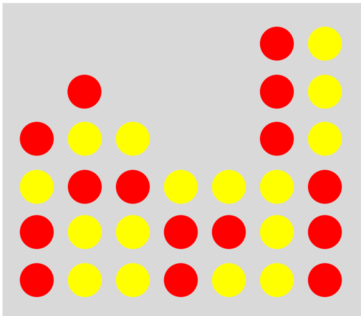
On suppose que tout le monde joue de manière optimale



Fonction d'évaluation

La qualité de l'IA dépend de la profondeur de l'arbre mais beaucoup de la fonction d'évaluation !

Pour le « Puissance 4 », quelle pourrait être la fonction d'évaluation ?

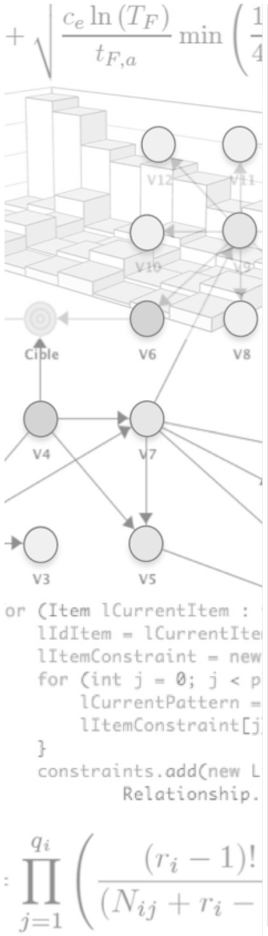


Par exemple :

Pour chaque joueur, faire la somme des points calculés comme suit :

- 4 pions alignés : 1000 points
- 3 pions et 1 case vide alignés : 50 points
- 2 pions et 2 cases vides alignés : 5 points
- 1 pion et 3 cases vides alignés : 1 point

Faire la différence des scores des deux joueurs.



Code

```
function MINIMAX(racine, maxProfondeur)
    eval, action = JOUEURMAX(racine, maxProfondeur)
    return action
end function
```

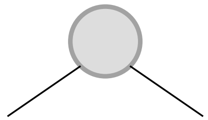
```
function JOUEURMAX(n, p)
    if n est une feuille ou p = 0 then
        return EVAL(n), _null_
    end if
    u = −∞ et a = _null_
    for all f fils de n (obtenu par une action a_f) do
        eval, . = JOUEURMIN(f, p − 1)
        if eval > u then
            u = eval et a = a_f
        end if
    end for
    return u, a
end function
```

```
function JOUEURMIN(n, p)
    if n est une feuille ou p = 0 then
        return EVAL(n), _null_
    end if
    u = +∞ et a = _null_
    for all f fils de n (obtenu par une action a_f) do
        eval, . = JOUEURMAX(f, p − 1)
        if eval < u then
            u = eval et a = a_f
        end if
    end for
    return u, a
end function
```

Alpha-beta = Minimax + élagage (pruning)

Objectif : diminuer la taille de l'arbre de l'algorithme Minimax

$\alpha =$
$\beta =$

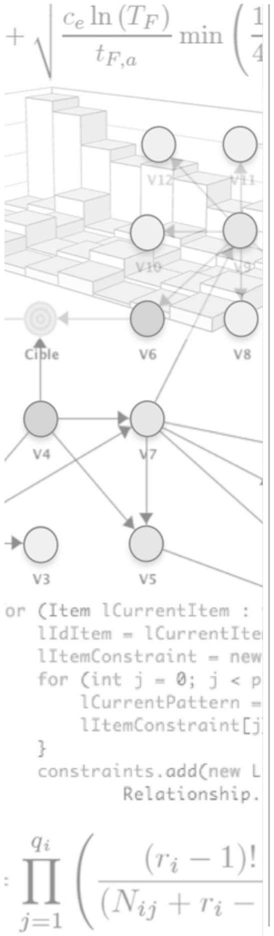


α : meilleure valeur de la branche pour le joueur Max
α n'est mis à jour que sur les nœuds Max

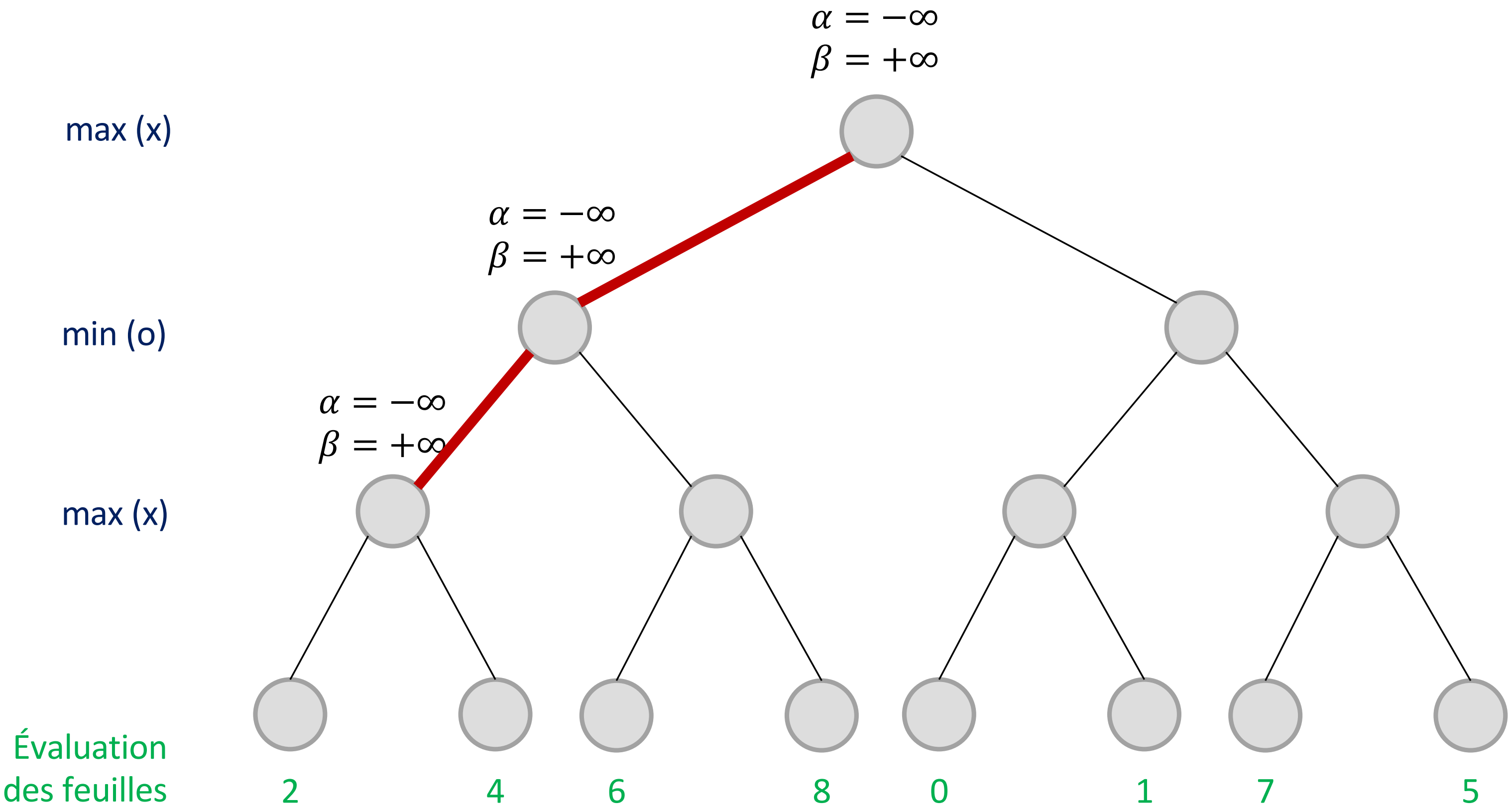
β : meilleure valeur de la branche pour le joueur Min
β n'est mis à jour que sur les nœuds Min

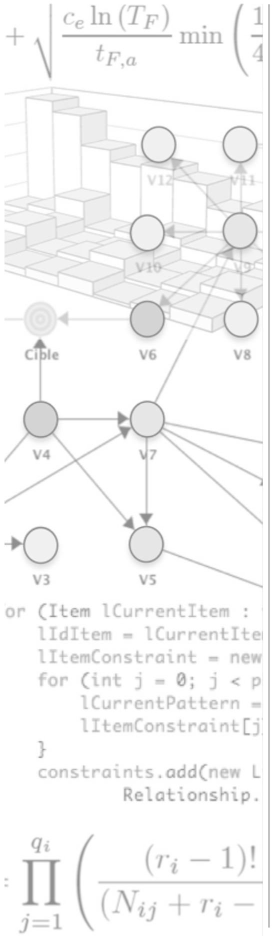
Initialisation : $\alpha = -\infty$
$\beta = +\infty$

<u>Règle</u> : si $\alpha \geq \beta$, on élague la branche

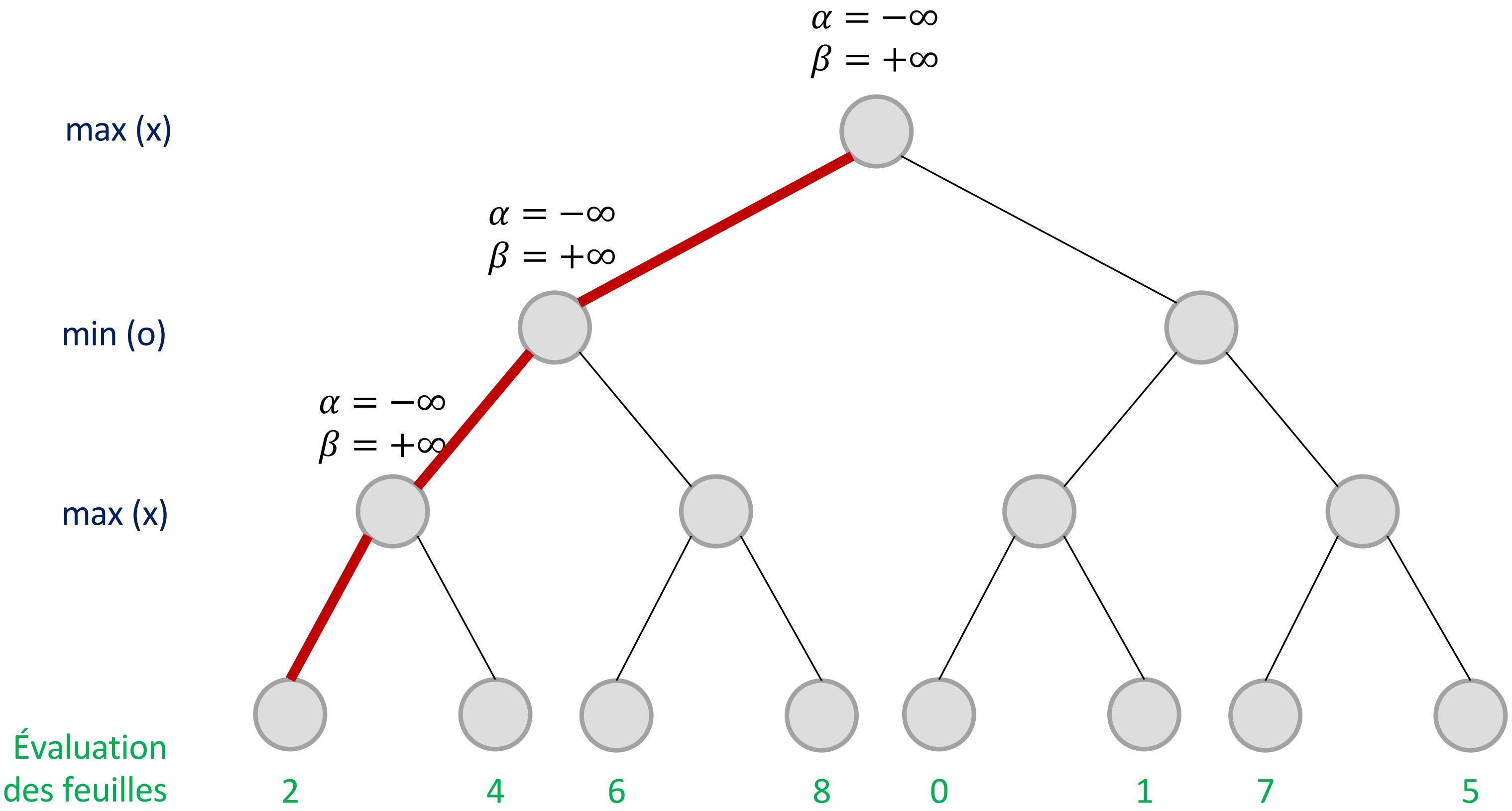


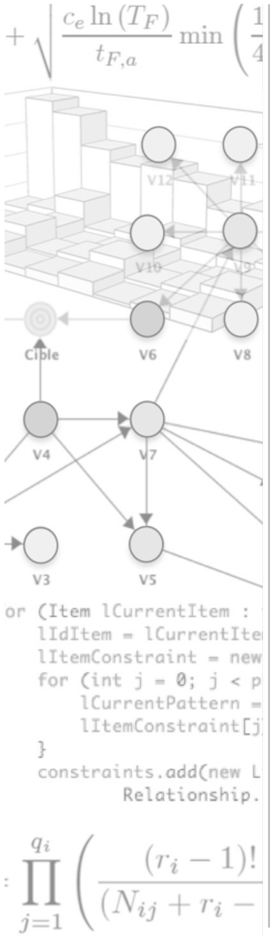
Alpha-beta = Minimax + élagage (pruning)



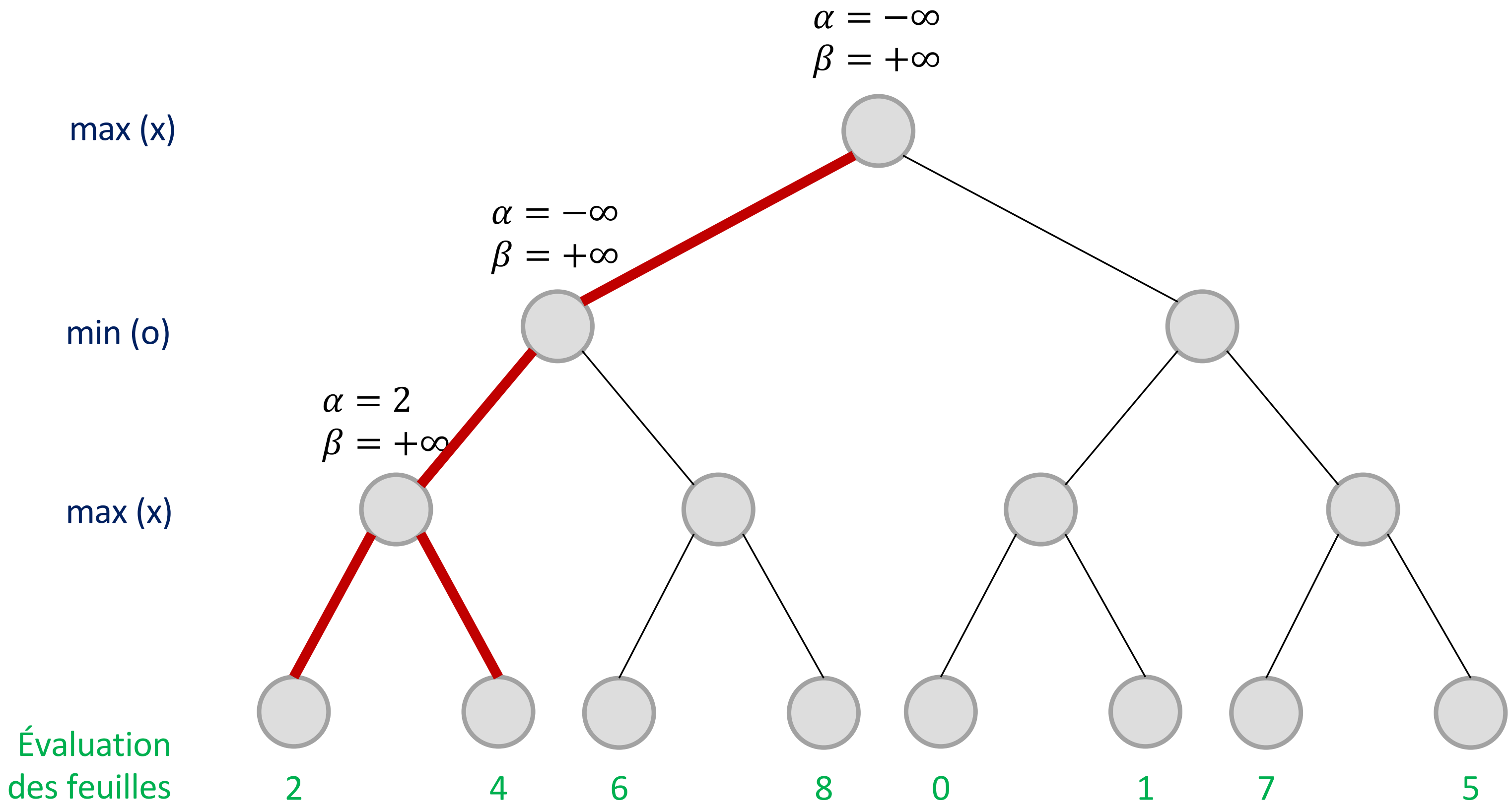


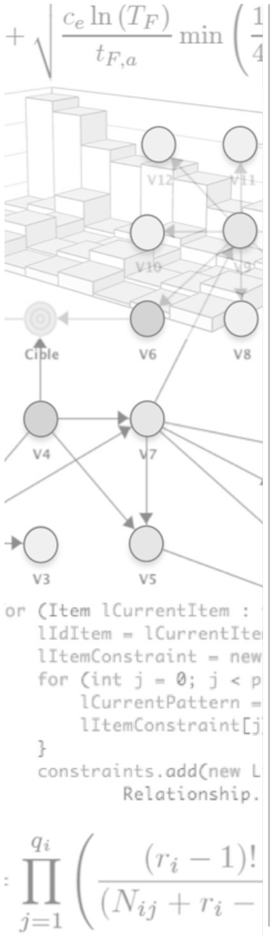
Alpha-beta = Minimax + élagage (pruning)



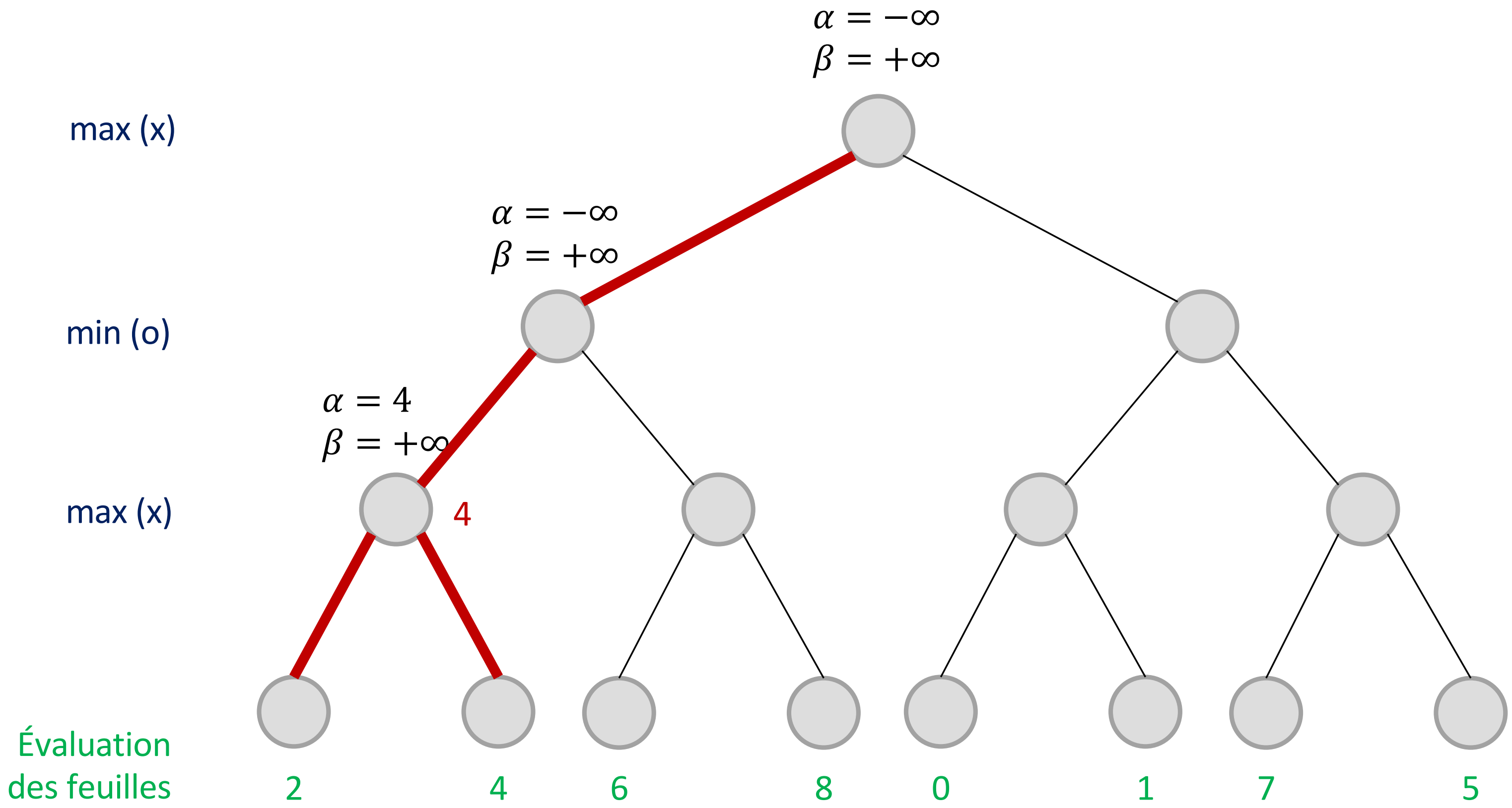


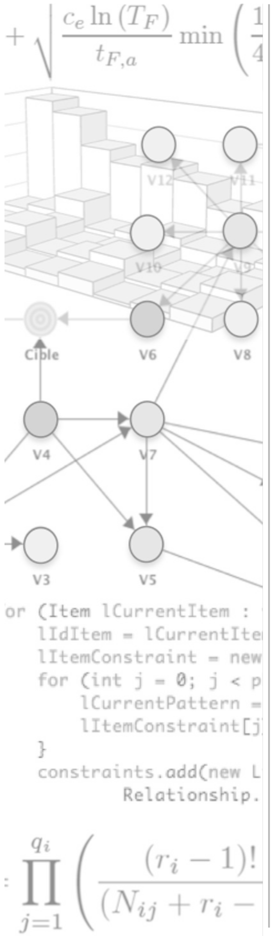
Alpha-beta = Minimax + élagage (pruning)



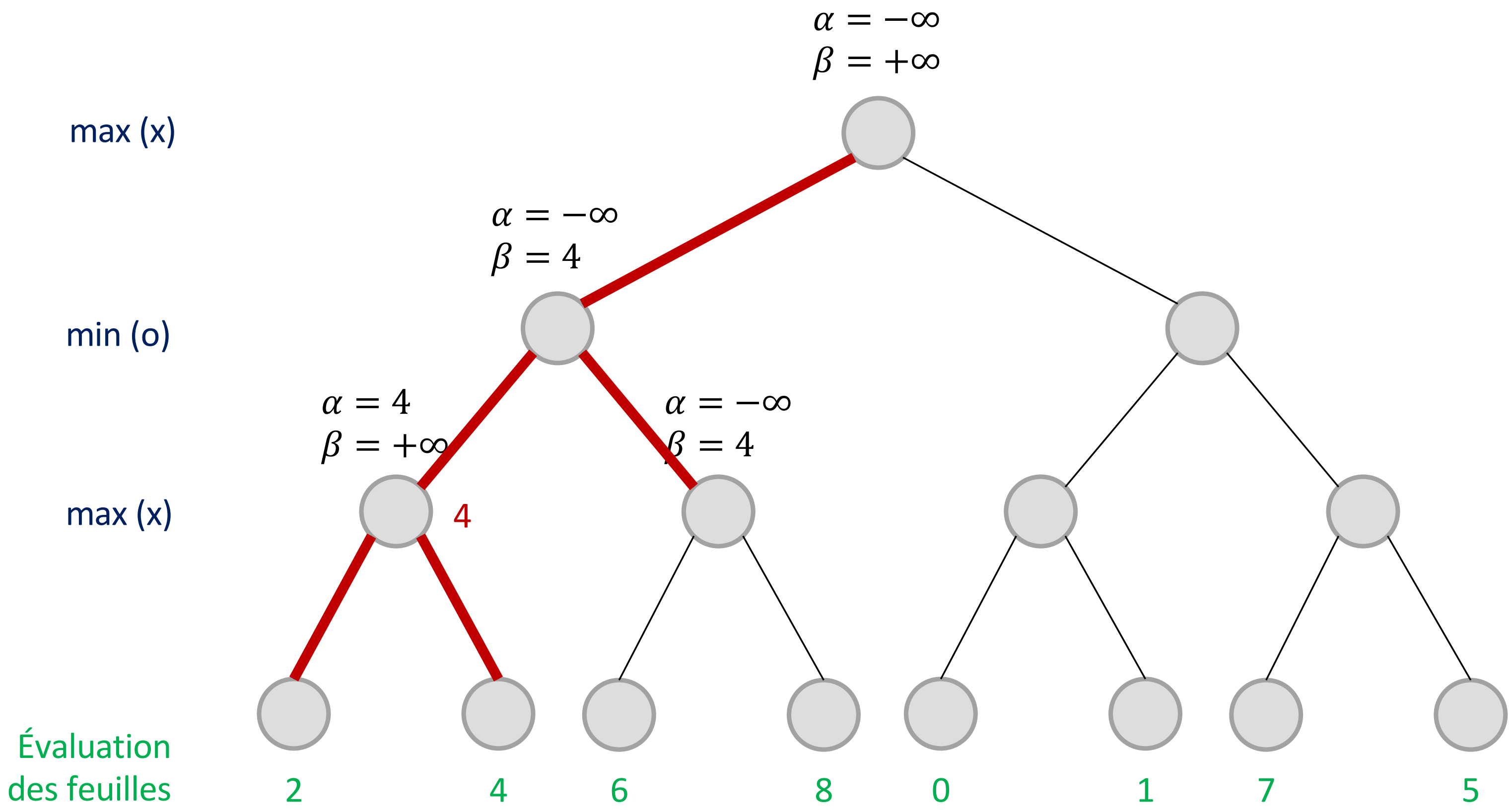


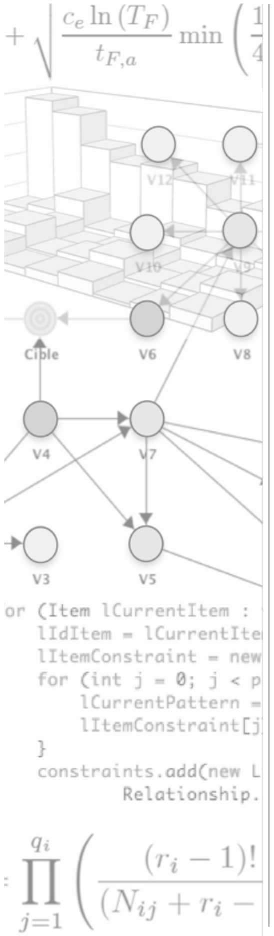
Alpha-beta = Minimax + élagage (pruning)



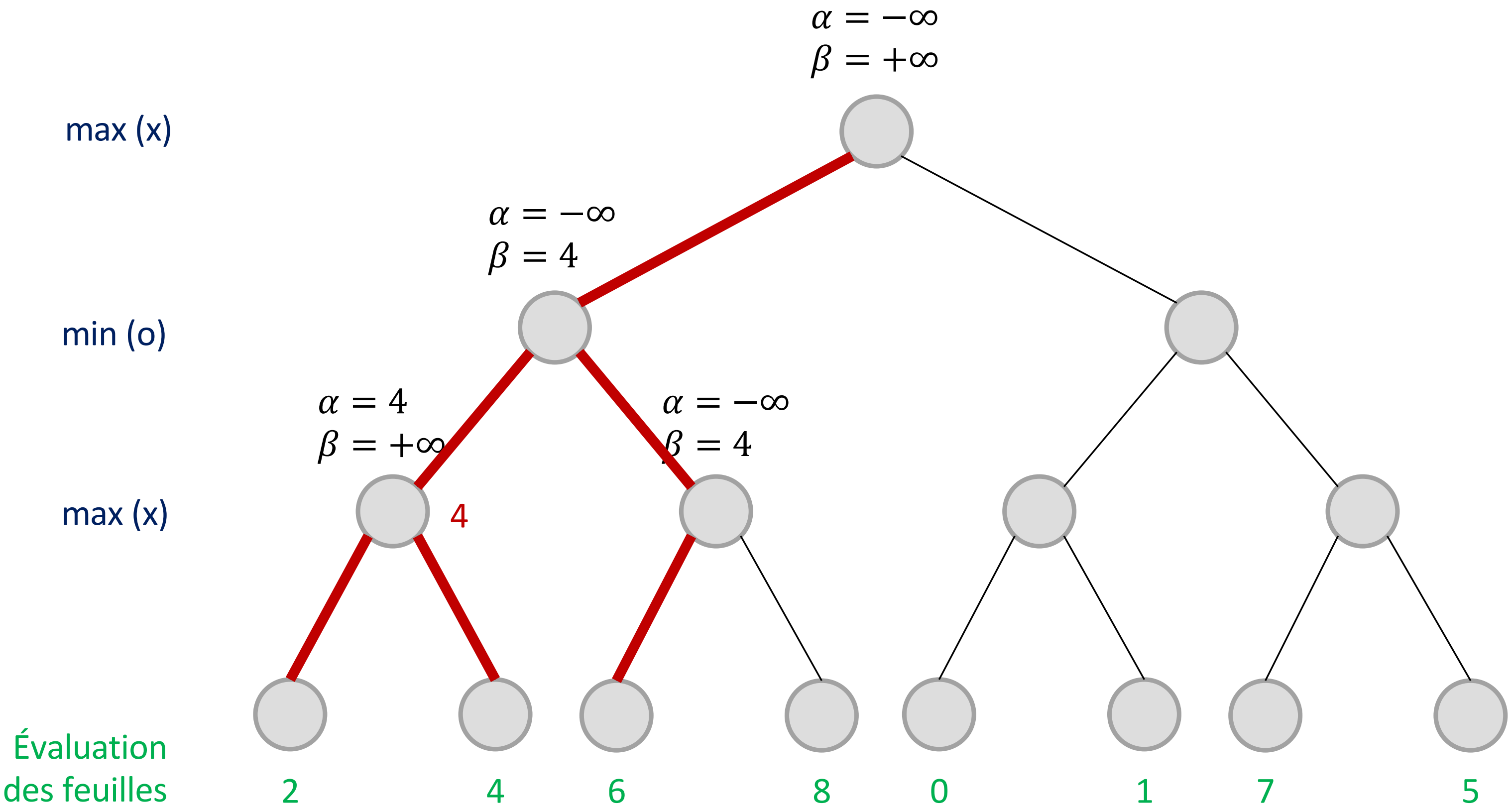


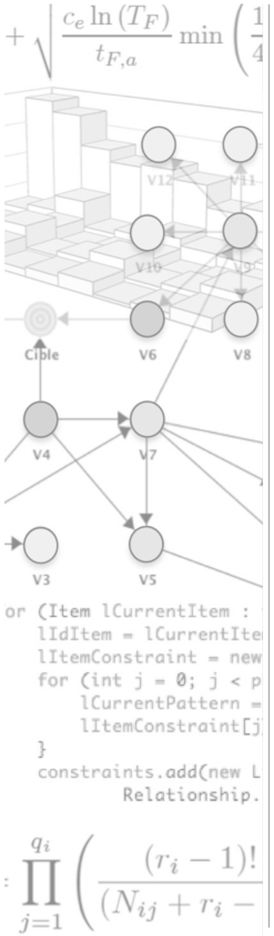
Alpha-beta = Minimax + élagage (pruning)



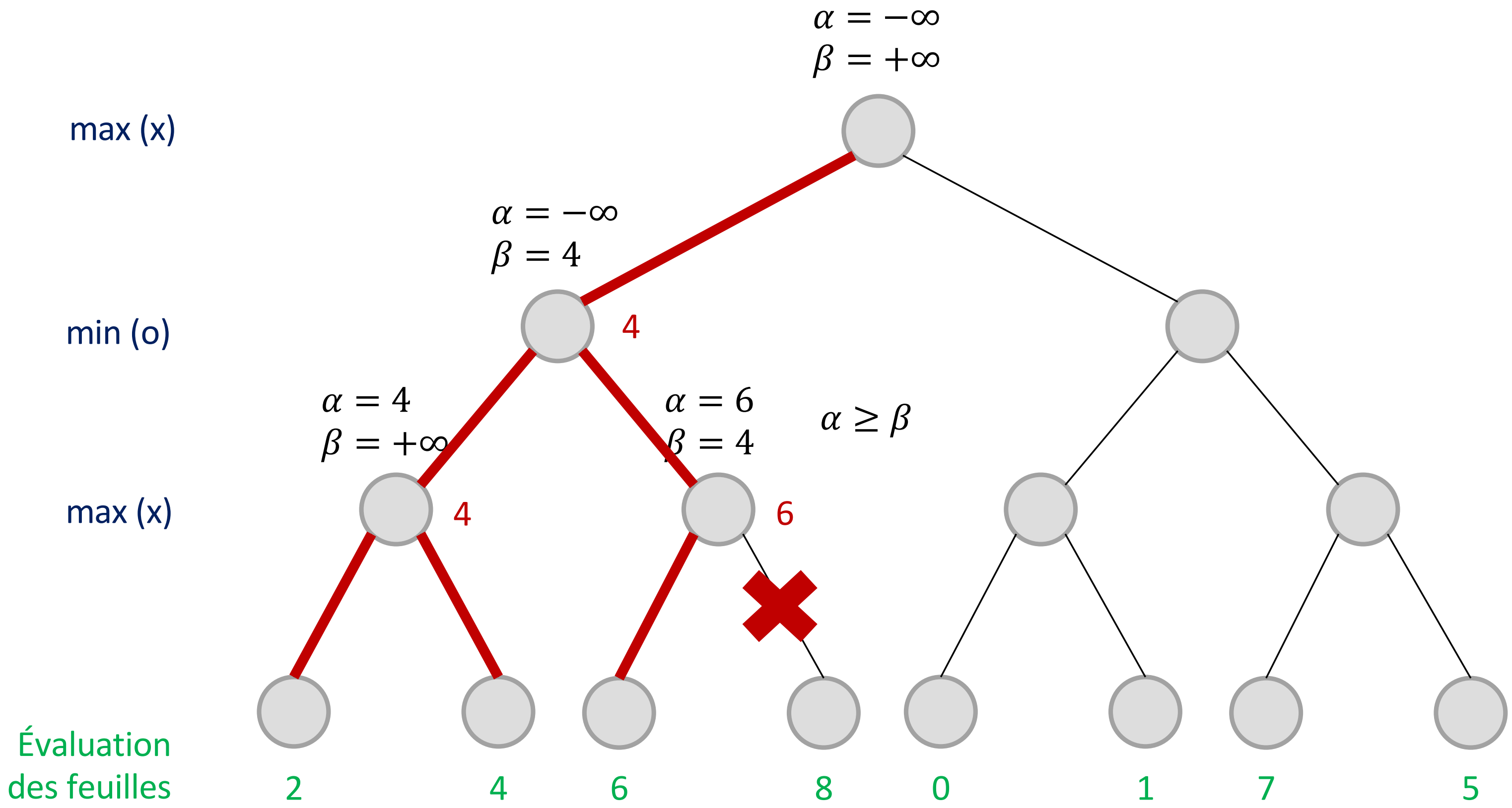


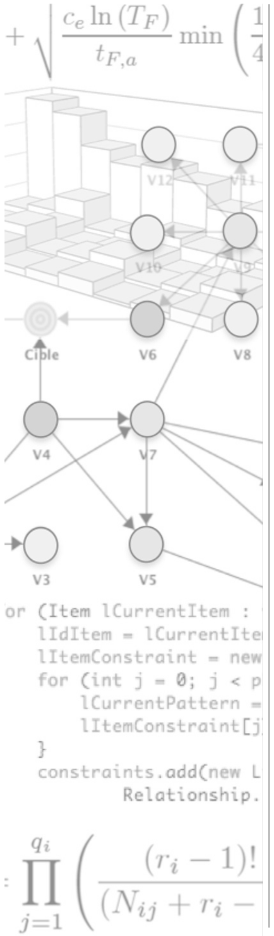
Alpha-beta = Minimax + élagage (pruning)



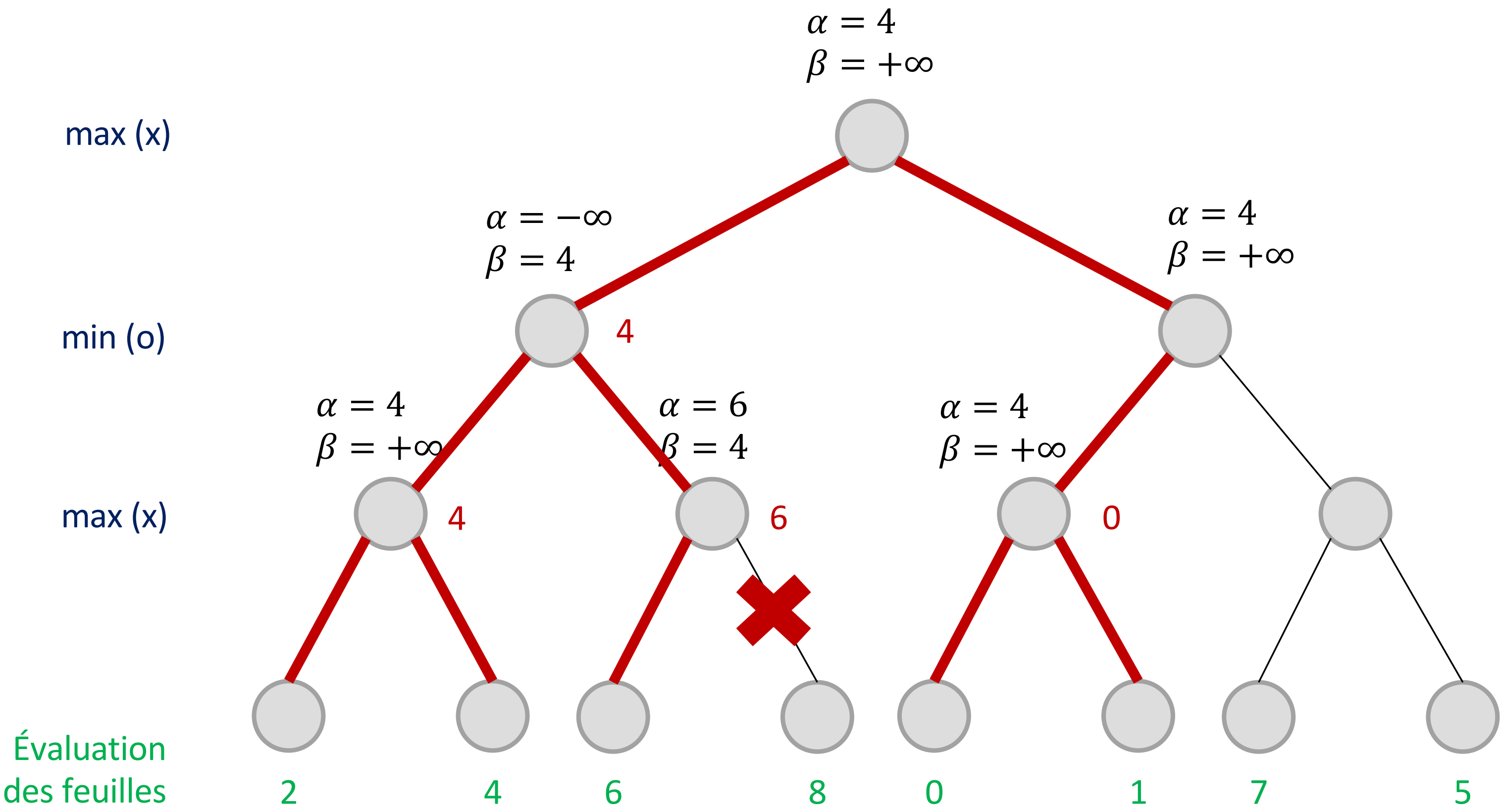


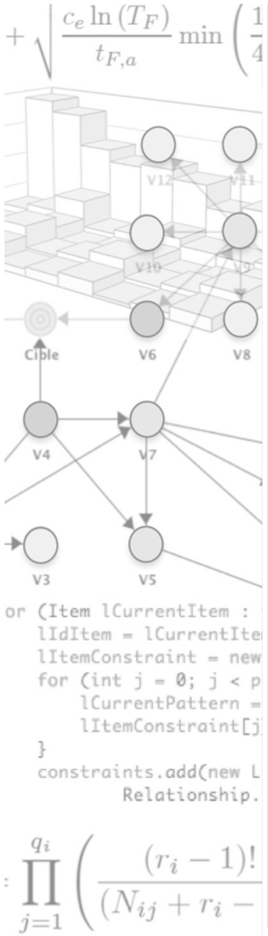
Alpha-beta = Minimax + élagage (pruning)



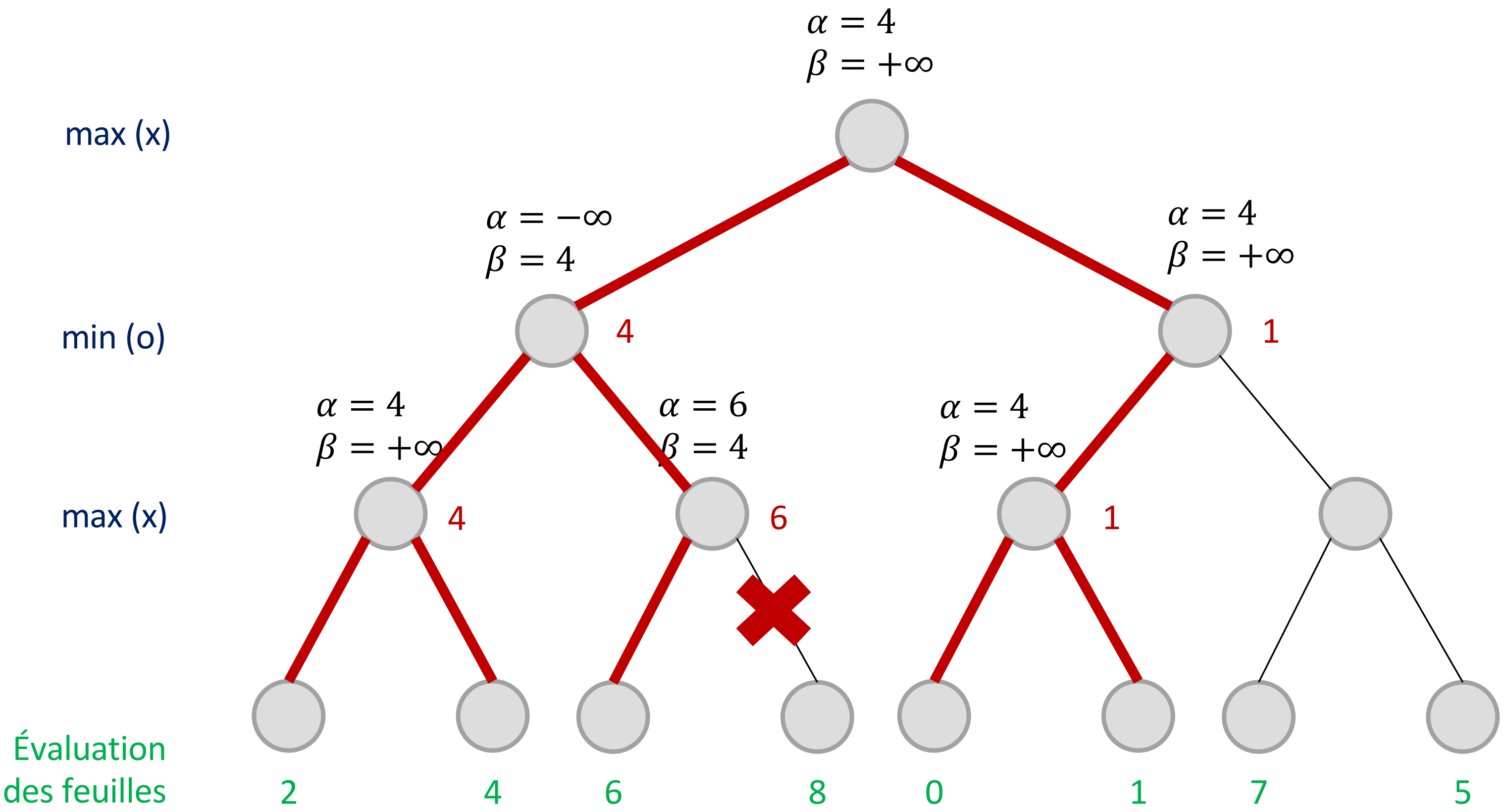


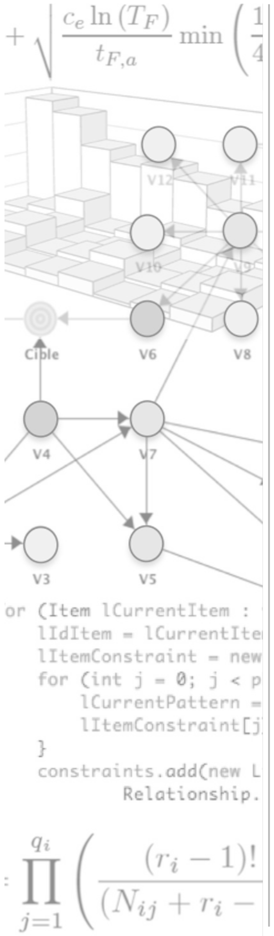
Alpha-beta = Minimax + élagage (pruning)



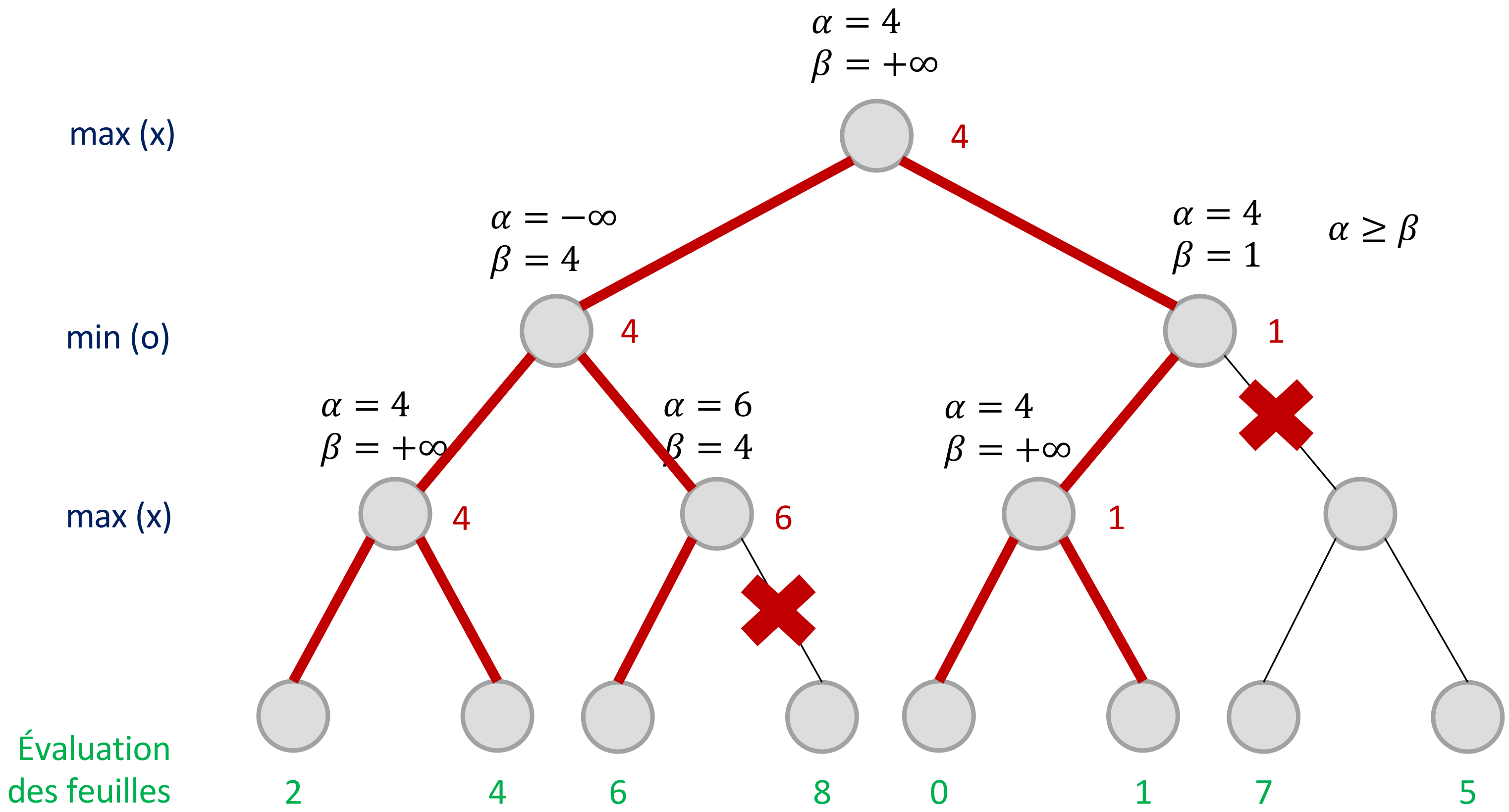


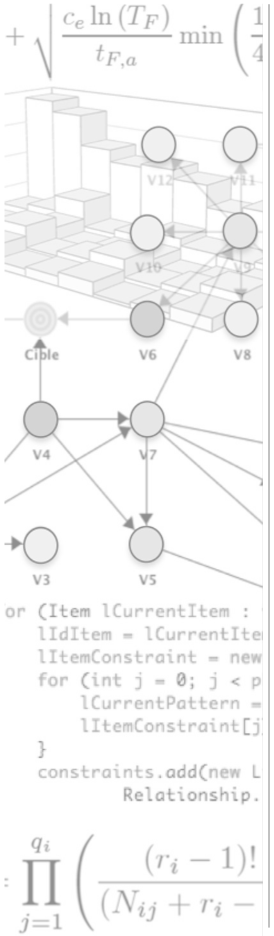
Alpha-beta = Minimax + élagage (pruning)





Alpha-beta = Minimax + élagage (pruning)



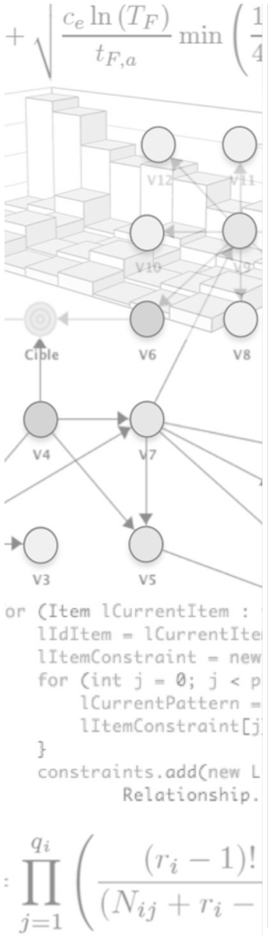


Code

function ALPHA-BETA($racine$, $maxProfondeur$)
    eval, action = JOUEURMAX($racine$, $maxProfondeur$, $-\infty$, $+\infty$)
    return action
end function

function JOUEURMAX(n, p, α, β)
    if n est une feuille *ou* $p = 0$ **then**
        return EVAL(n), _null_
    end if
    $u = -\infty$ et $a =$ _null_
    for all f fils de n (obtenu par une action a_f) **do**
        eval, . = JOUEURMIN($f, p-1$, α, β)
        if $eval > u$ **then**
            $u = eval$ et $a = a_f$
        end if
        if $u \geq \beta$ **then**
            return u, a
        end if
        $\alpha = max(\alpha, u)$
    end for
    return u, a
end function

function JOUEURMIN(n, p, α, β)
    if n est une feuille *ou* $p = 0$ **then**
        return EVAL(n), _null_
    end if
    $u = +\infty$ et $a =$ _null_
    for all f fils de n (obtenu par une action a_f) **do**
        eval, . = JOUEURMAX($f, p-1$, α, β)
        if $eval < u$ **then**
            $u = eval$ et $a = a_f$
        end if
        if $u \leq \alpha$ **then**
            return u, a
        end if
        $\beta = min(\beta, u)$
    end for
    return u, a
end function



Monte Carlo Tree Search

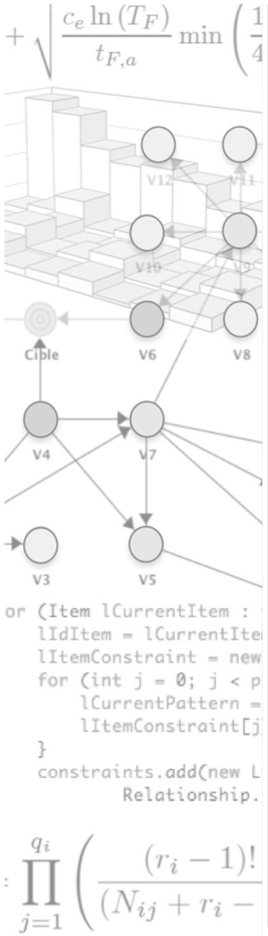
IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1, MARCH 2012 1

A Survey of Monte Carlo Tree Search Methods

Cameron Browne, *Member, IEEE,* Edward Powley, *Member, IEEE,* Daniel Whitehouse, *Member, IEEE,* Simon Lucas, *Senior Member, IEEE,* Peter I. Cowling, *Member, IEEE,* Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton

Abstract—Monte Carlo Tree Search (MCTS) is a recently proposed search method that combines the precision of tree search with the generality of random sampling. It has received considerable interest due to its spectacular success in the difficult problem of computer Go, but has also proved beneficial in a range of other domains. This paper is a survey of the literature to date, intended to provide a snapshot of the state of the art after the first five years of MCTS research. We outline the core algorithm's derivation, impart some structure on the many variations and enhancements that have been proposed, and summarise the results from the key game and non-game domains to which MCTS methods have been applied. A number of open research questions indicate that the field is ripe for future work.

Index Terms—Monte Carlo Tree Search (MCTS), Upper Confidence Bounds (UCB), Upper Confidence Bounds for Trees (UCT), Bandit-based methods, Artificial Intelligence (AI), Game search, Computer Go.

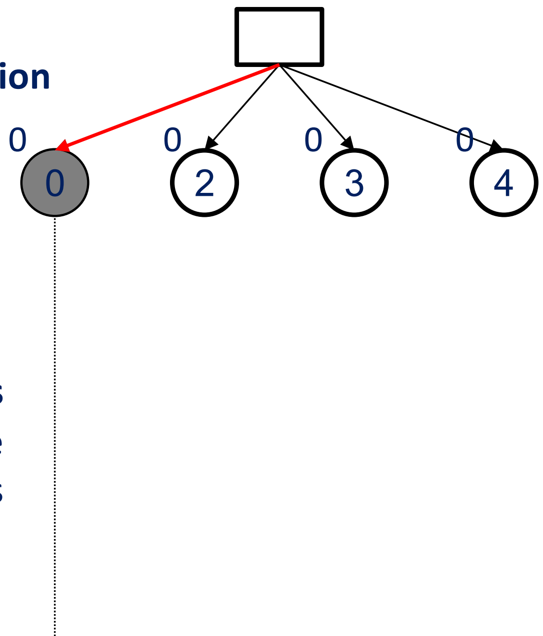


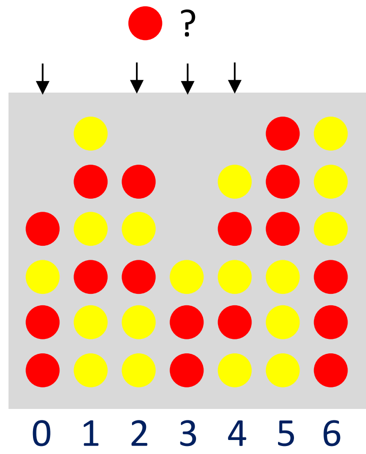
Monte Carlo Tree Search

Selection

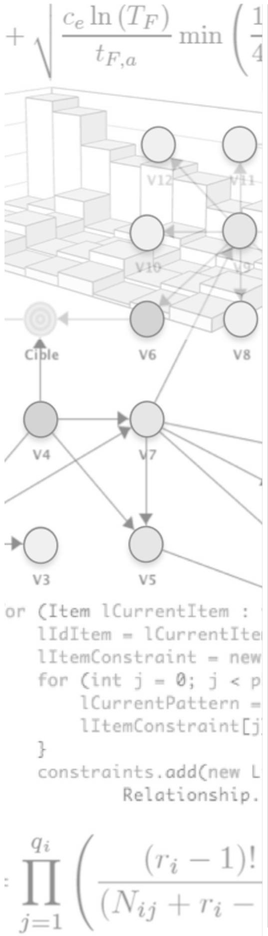
Expansion
= création du nœud

Simulations
= génération aléatoire
de plusieurs parties



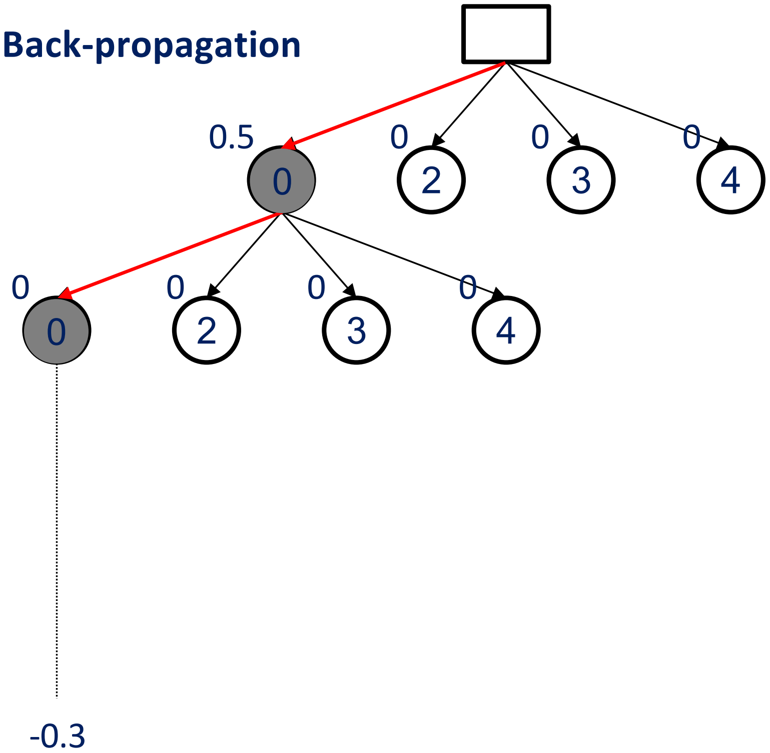


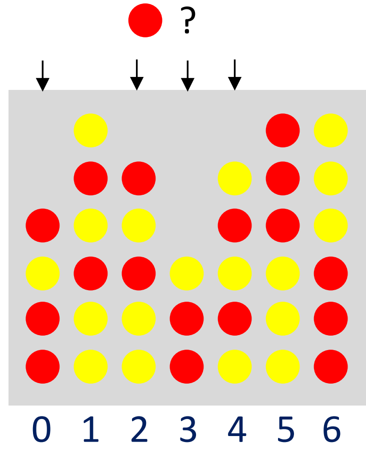
+0.5 en moyenne, le joueur gagne un peu qu'il ne perd



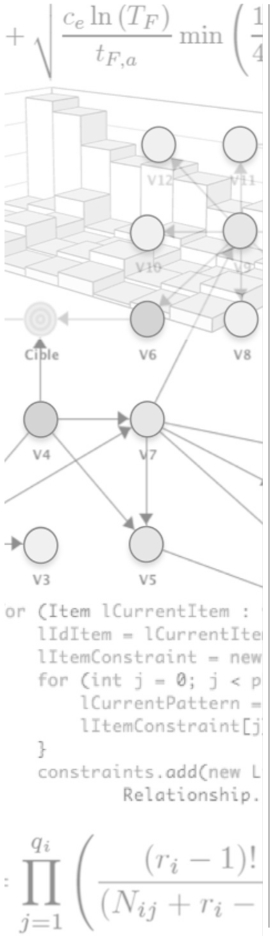
Monte Carlo Tree Search

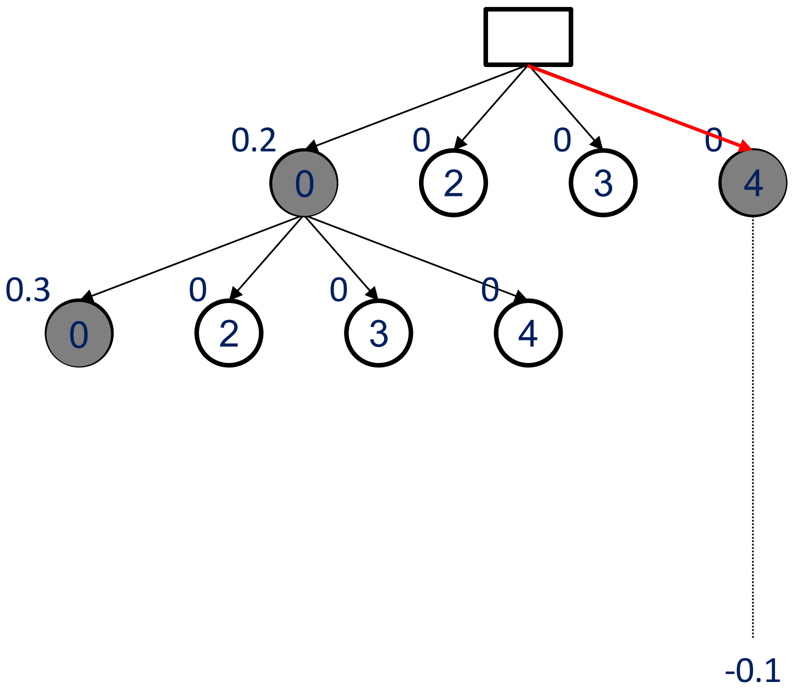
Back-propagation

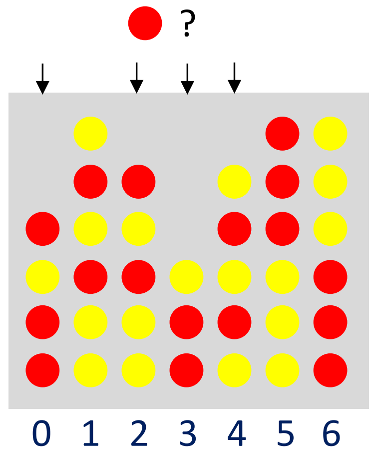




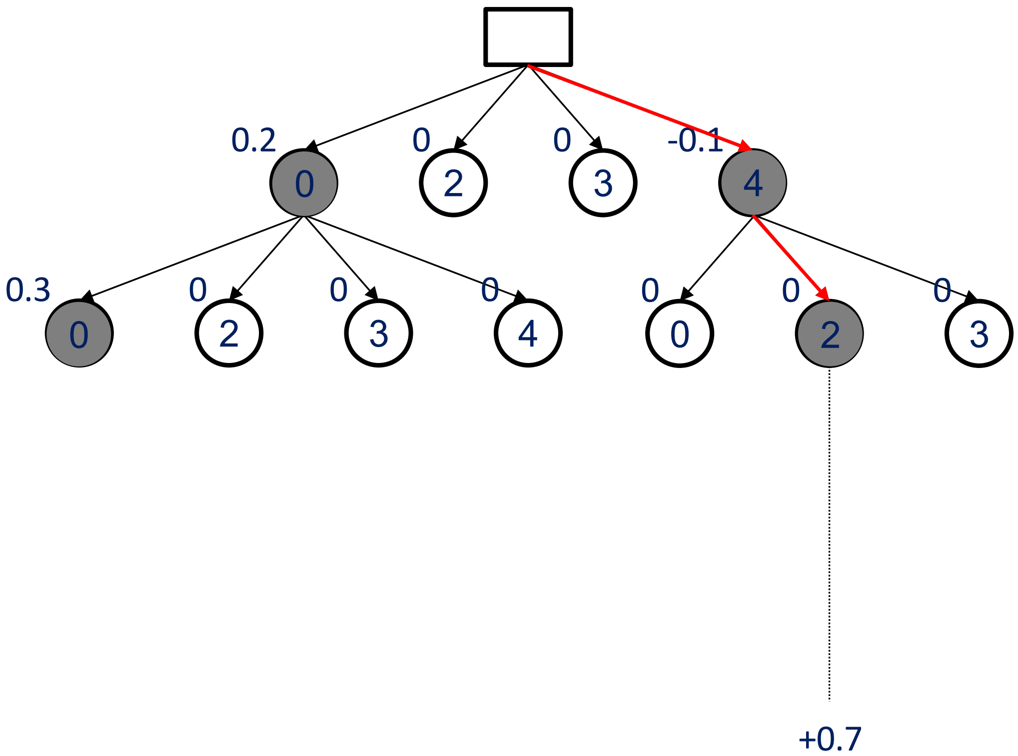
Monte Carlo Tree Search

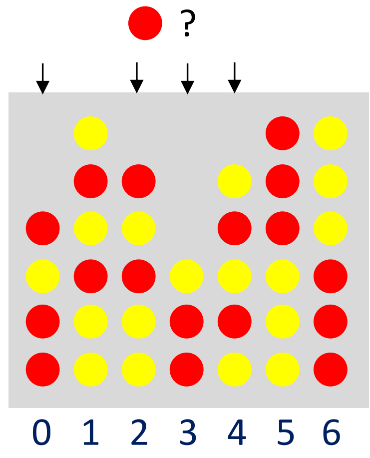


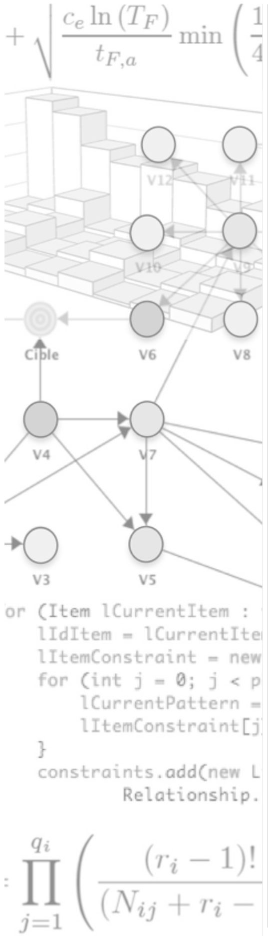




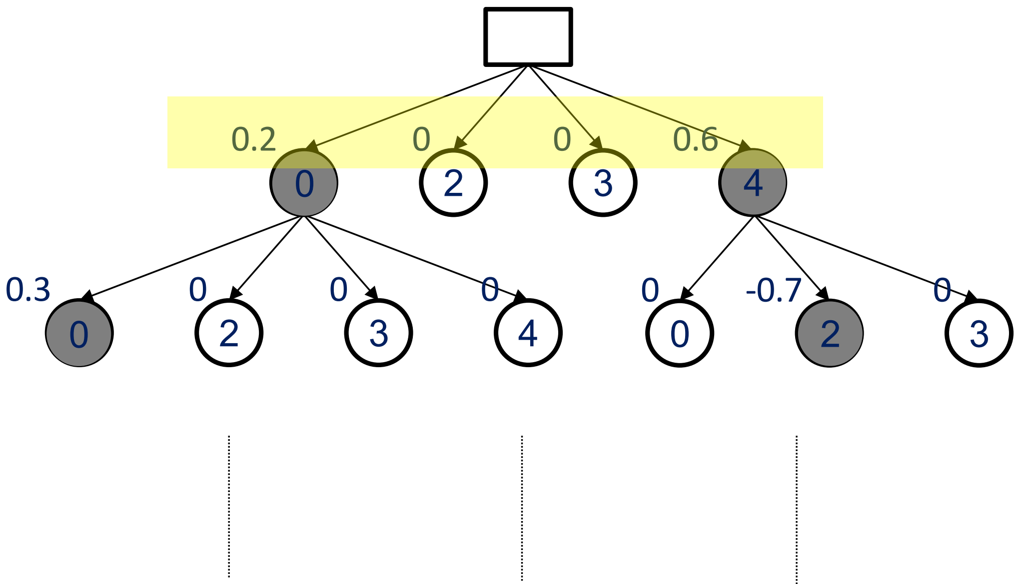
Monte Carlo Tree Search

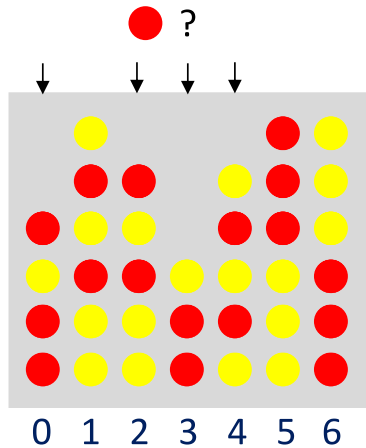






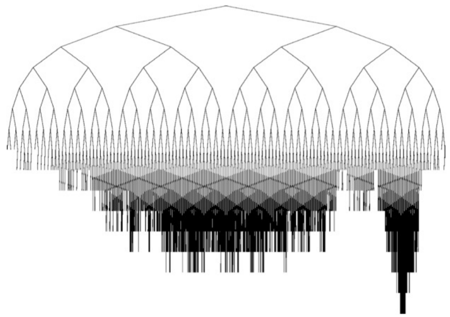
Monte Carlo Tree Search

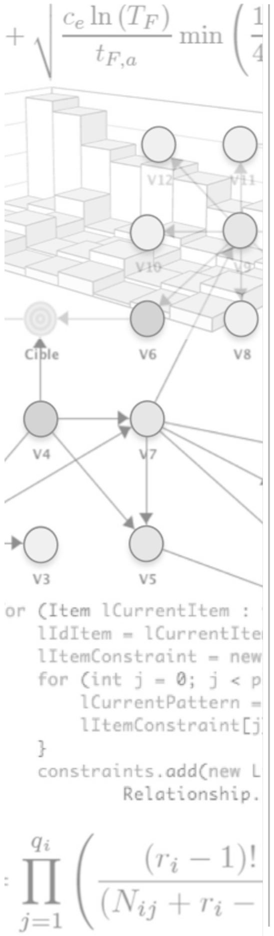




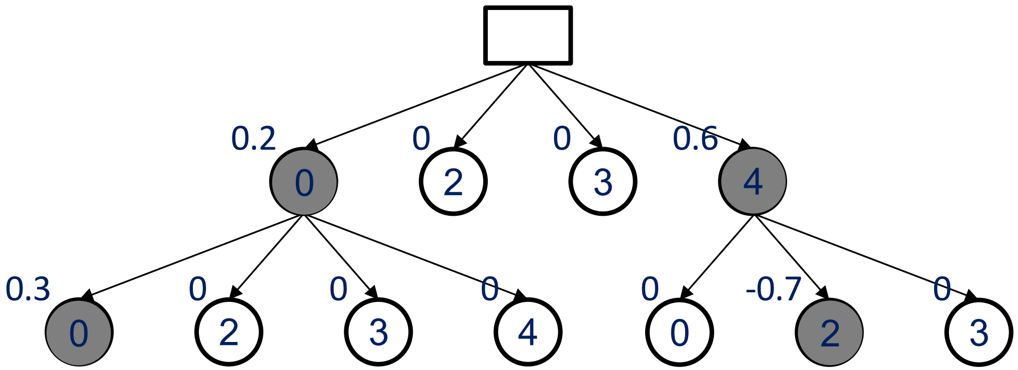
poursuivre le processus …

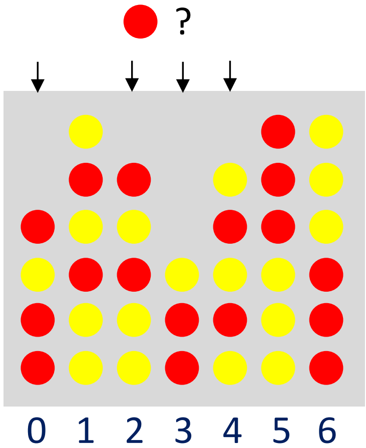
à la fin choisir l’action avec la plus grande valeur





Monte Carlo Tree Search



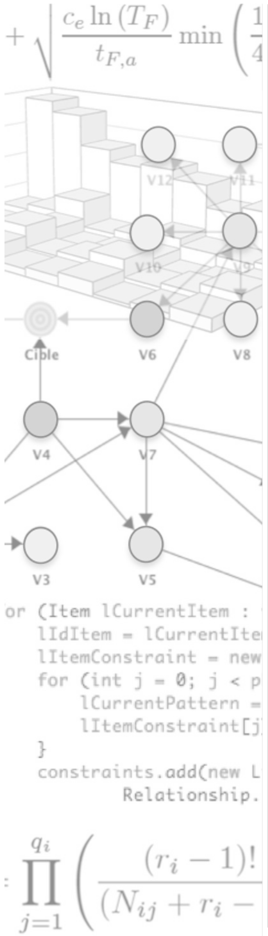


Sélection :

$$\frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

intensification diversification

La valeur de C influence la forme de l'arbre et la qualité des résultats



Code

3.3 Upper Confidence Bounds for Trees (UCT)

This section describes the most popular algorithm in the MCTS family, the *Upper Confidence Bound for Trees* (UCT)

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)

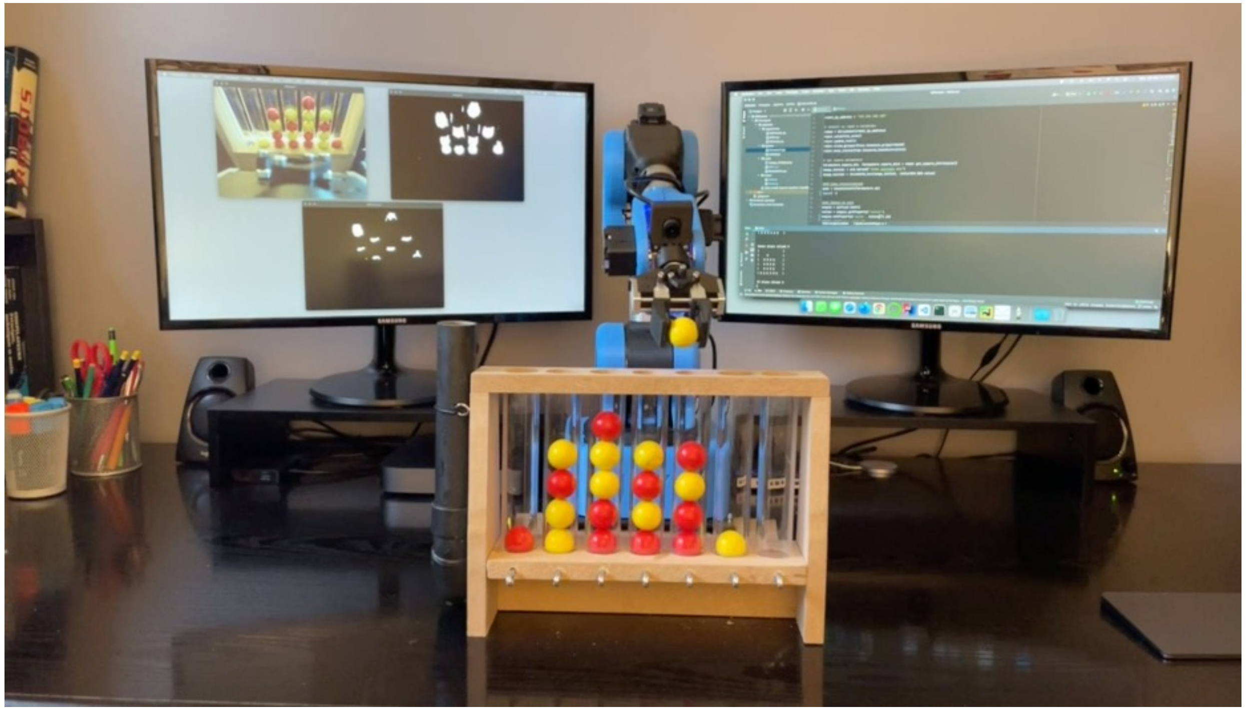
$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

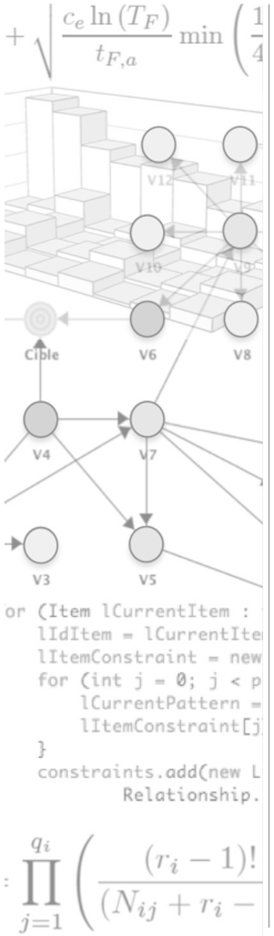
function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta$
    $\Delta \leftarrow -\Delta$
    $v \leftarrow$ parent of v

Ned de NIRYO

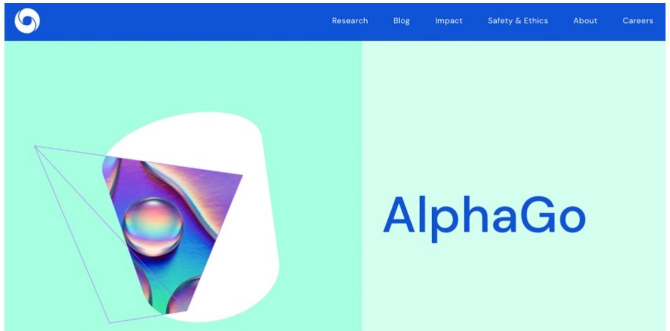
NIRYO





AlphaGo

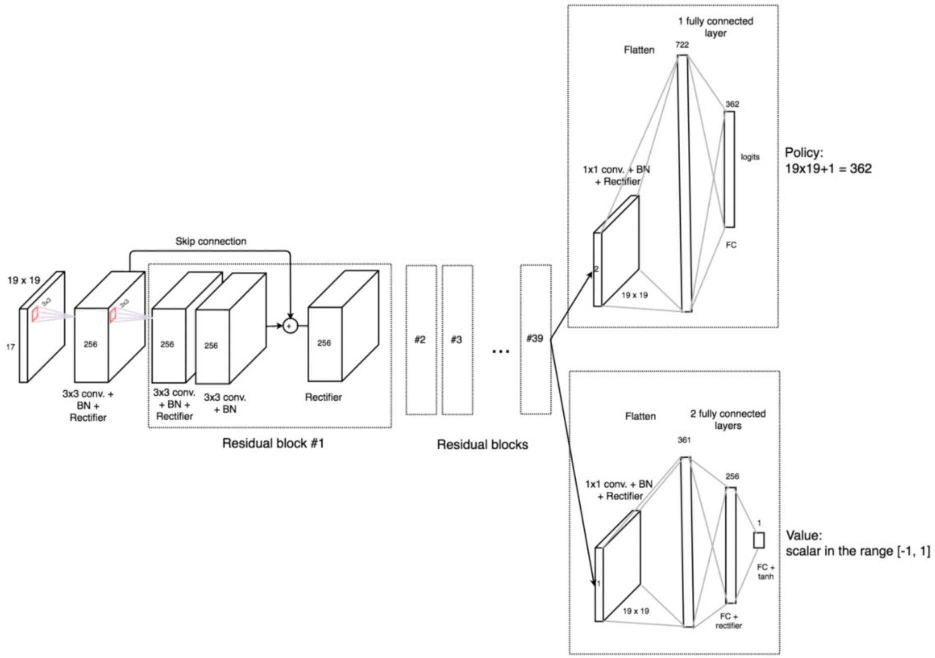
https://www.youtube.com/watch?v=WXuK6gekU1Y



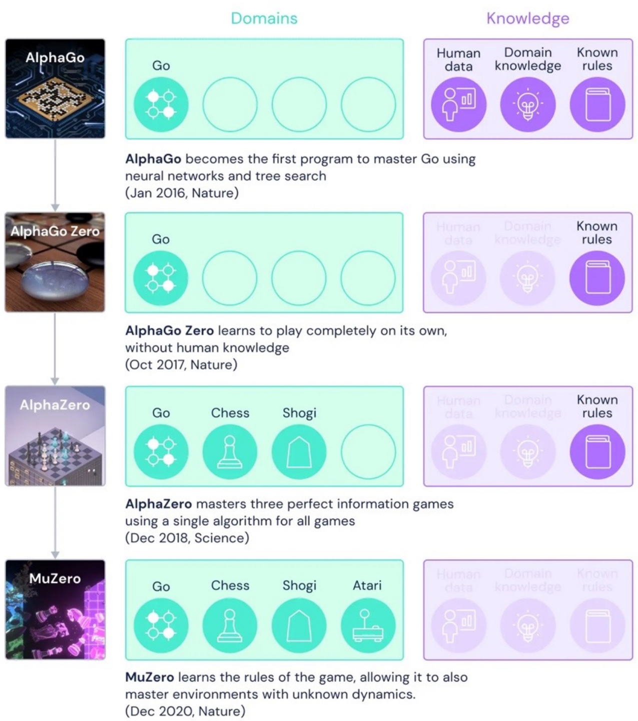
https://www.deepmind.com/research/highlighted-research/alphago

AlphaGo Zero

Idée : faire apprendre la **policy** et la **value function** à un réseau de neurones



AlphaGo, AlphaGo Zero, AlphaZero, MuZero



AlphaZero

- Généralisation de AlphaGo Zero à d'autres jeux que le jeu de Go
- Globalement même architecture que AlphaGo Zero
- Par contre, il faut adapter l'entrée du NN en fonction du jeu
- Pas d'évaluateur : c'est le même NN qui est amélioré au cours du temps