

Présentation

Un regard approfondi sur les moteurs du jeu d'échecs

Khalil Bitar



Sommaire

- Classement ELO :
 - Approximation d'ELO
 - Calcul du classement d'un joueur
- Moteur d'échecs traditionnel :
 - Algorithme Minimax
- Moteur d'échecs moderne :
 - Recherche arborescente de Monte Carlo

Approximation d'ELO

Hypothèses posées par ELO :

Le niveau de chaque joueur est une variable aléatoire continue X_{joueur} suivant la loi uniforme où

1. $E(X_{\text{joueur}})$ est inconnue et signifie le niveau moyen du joueur
2. $V(X_{\text{joueur}})$ est la même pour chaque joueur

Approximation d'ELO

Approximation de la fonction de repartition de

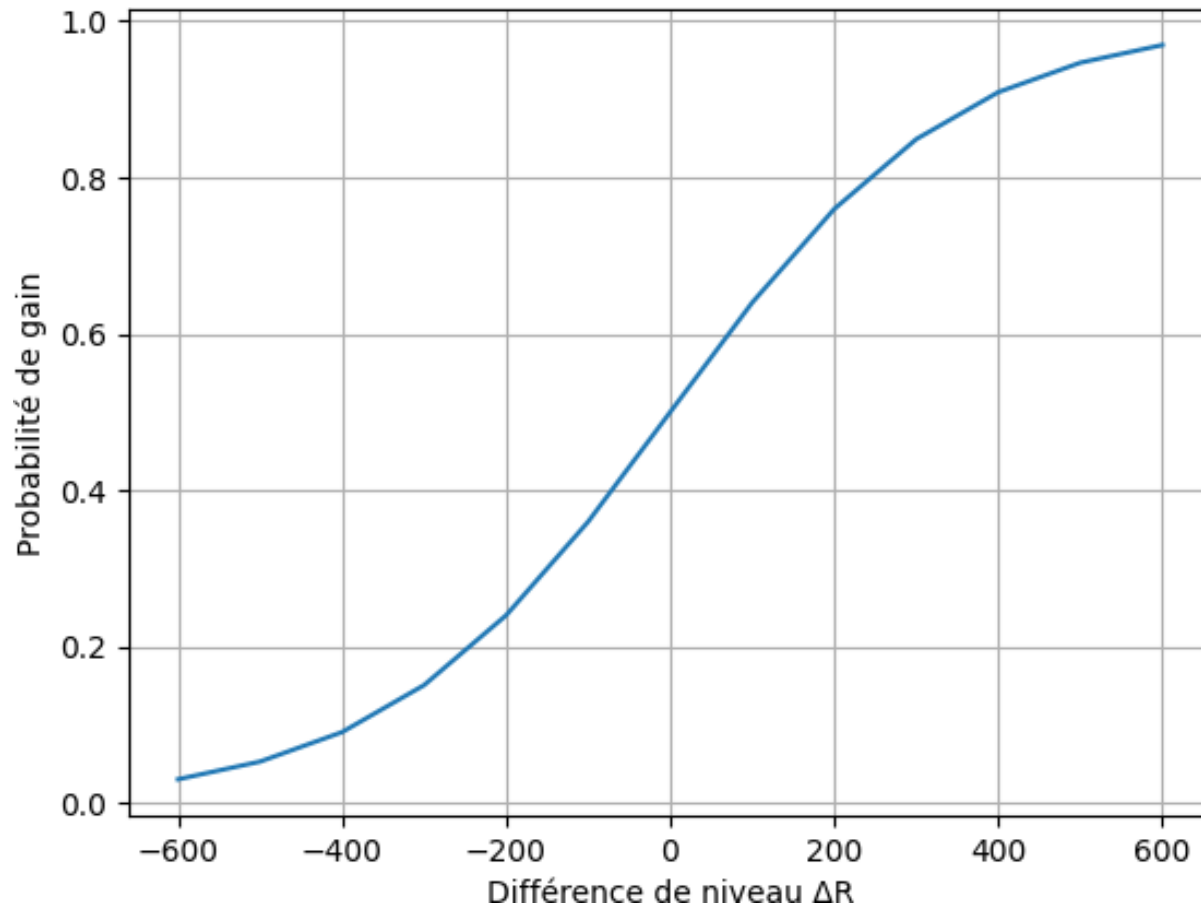
X_{joueur} : la fonction logistique

$$E = \frac{1}{1 + 10^{-(R_B - R_N)/400}}$$

R_B : niveau du joueur avec les pièces blanches

R_N : niveau du joueur avec les pièces noires

Modélisation de la fonction logistique E



Calcul du classement ELO d'un joueur

$$R_{post} = R_{pre} + K. (W_a - E(R_{pre} - R_{adversaire}))$$

- R_{post} , R_{pre} : le classement du joueur avant et après le jeu
- K : coefficient basé sur le classement du joueur avant le jeu
 - W_a : le résultat du jeu

Tous ces coefficients sont connues, c'est le calcul de E qui varie, or la méthode précédente de ELO est la méthode la plus utilisée

Préambule

- $\text{Score_blanc} = \sum_i P_i(\text{blanc})$
- $\text{Score_noir} = \sum_i P_i(\text{noir})$
- où P_i le poids de la pièce i
- La valeur V_1 d'une position pour le joueur 1 est définie en générale $V_1 = \Delta_{0 \rightarrow 1} \text{score}_i$



$P_{roi} = 900$



$P_{reine} = 90$



$P_{fou} = 30$



$P_{cavalier} = 30$



$P_{tour} = 50$



$P_{pion} = 10$

Notations générales

On note :

- s un état (position) quelconque
- a une action possible (movement) dans un état
- A_s l'ensemble des actions possible dans un état s
- S_0 l'état initial (la position qu'on veut évaluer)

Notations générales

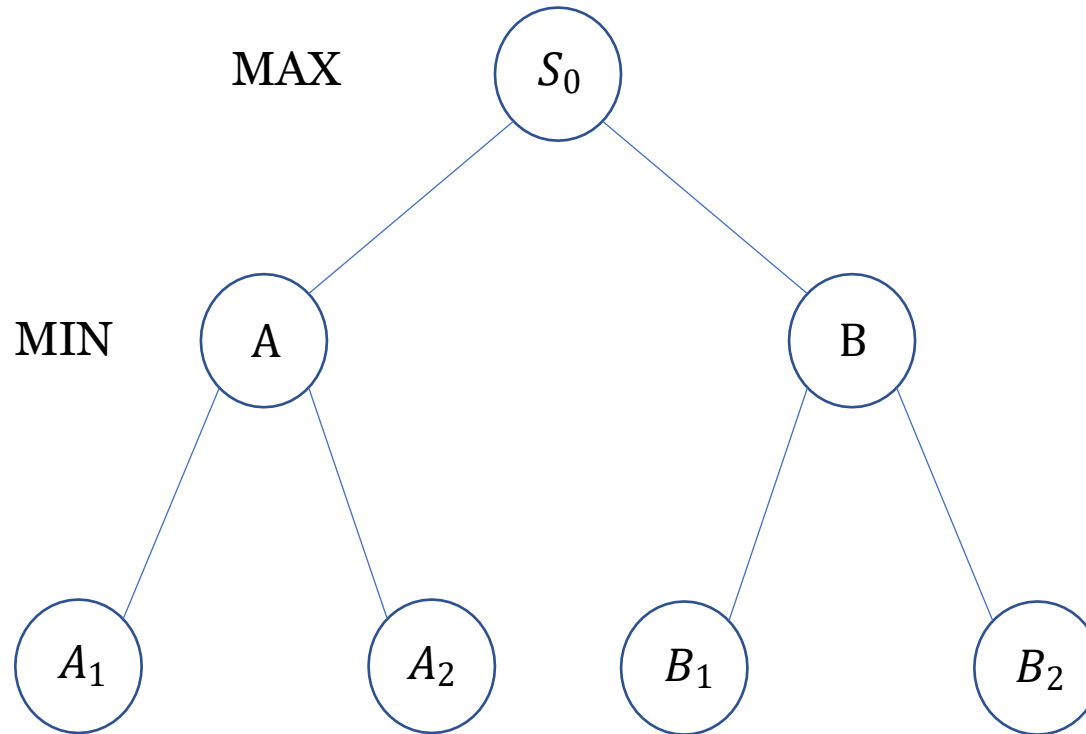
On introduit les fonctions suivantes :

- $R_i(s)$: l'avantage du joueur i dans la position s
 - $P_a(s_1, s_2)$: la probabilité que l'action a dans l'état s_1 nous donne l'état s_2
- $\text{joueur}(s)$: indique quel joueur doit jouer dans l'état s
 - $\text{actions}(s)$: les actions possibles dans l'état s
- $\text{résultat}(s, a)$: donne l'état s' , résultat de l'action a dans l'état s
 - $\text{terminal}(s)$: vérifie si s est un état terminal

Algorithme de Minimax

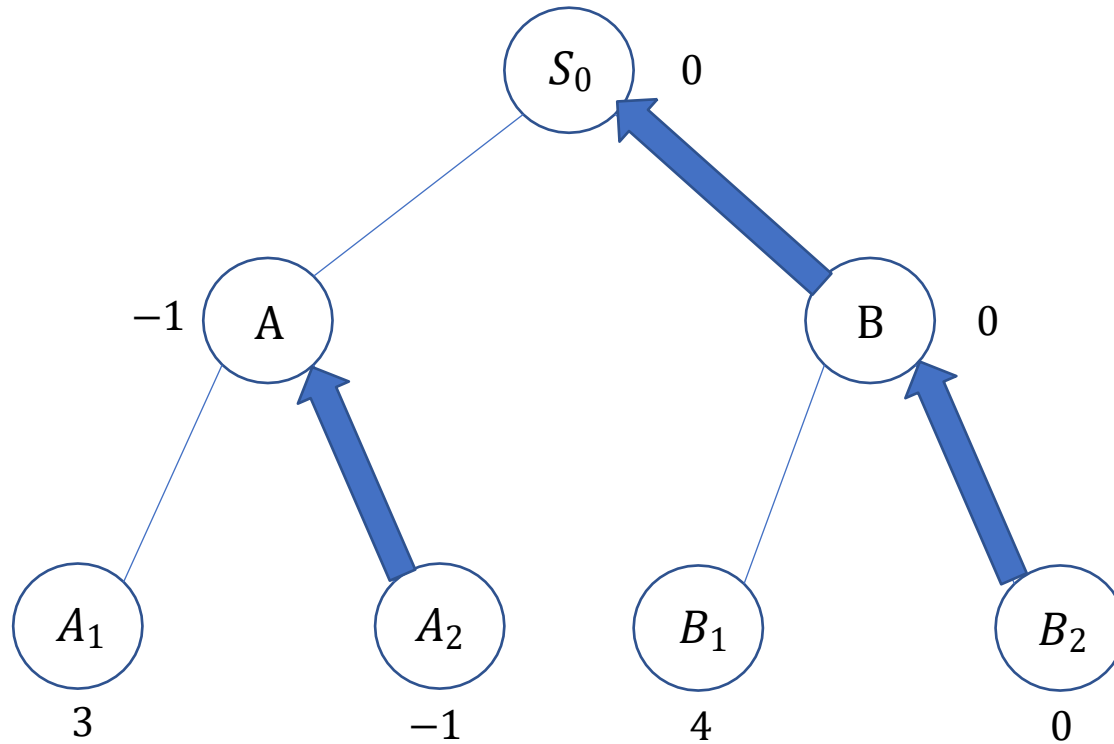
- On considère les 2 joueurs MAX (pour lequel on veut évaluer la position) et MIN (l'adversaire)
 - MAX cherche à maximiser son gain, MIN cherche à minimiser sa perte, c'est le principe fondamentale de l'Algorithme de Minimax
 - Les fonctions actions(s) et résultat(s,a) définissent un arbre de jeu
 - La fonction minimax(s) est récursive

Arbre du jeu (cas simple) et fonction minimax(s)



$$\text{minimax}(s) = \begin{cases} R_{MAX}(s) & \text{si } \text{terminal}(s) = \text{True} \\ \max_{a \in A_s}(\text{minimax}(\text{résultat}(s, a))) & \text{si } \text{joueur}(s) = MAX \\ \min_{a \in A_s}(\text{minimax}(\text{résultat}(s, a))) & \text{si } \text{joueur}(s) = MIN \end{cases}$$

Exemple



Algorithme de MiniMax

```
function miniMax(vertex,maxDepth, maxiPlayer)
  if maxDepth = 0 or vertex is a leaf node then
    | return value of vertex
  end
  if maxiPlayer then
    | score =  $-\infty$ 
    | while every child of vertex do
    |   | score=max(score, miniMax(child,
    |   | maxDepth -1, FALSE))
    | end
    | return score
  end
  else
    | score =  $+\infty$ 
    | while every child of vertex do
    |   | score=min(score, miniMax(child, maxDepth
    |   | -1, TRUE))
    | end
    | return score
  end
end
```

Limitations de Minimax

- Soit m la profondeur de l'exploration et b la moyenne de nombre de mouvement possible dans chaque position. Pour les échecs :
 - ❖ $m \approx 35$
 - ❖ $50 \leq b \leq 100$
- La complexité est $O(b^m)$
- Afin d'améliorer cette complexité, on utilise l'élagage alpha-bêta

Recherche Arborescente de Monte Carlo

LES CONCEPTS SUIVANTES SONT
ADAPTÉS PARTICULIÈREMENT AUX
ÉCHECS

$$\text{UCB1}(S_i) = \underbrace{v_i}_{\text{Terme d'exploitation}} + \underbrace{K \sqrt{\frac{\ln(N)}{n_i}}}_{\text{Terme d'exploration}}$$

v_i : la moyenne de la valeur de l'état S_i

K : constante d'exploration, on peut la modifier
basés sur le nombre de fois qu'on veut explorer les
nœuds

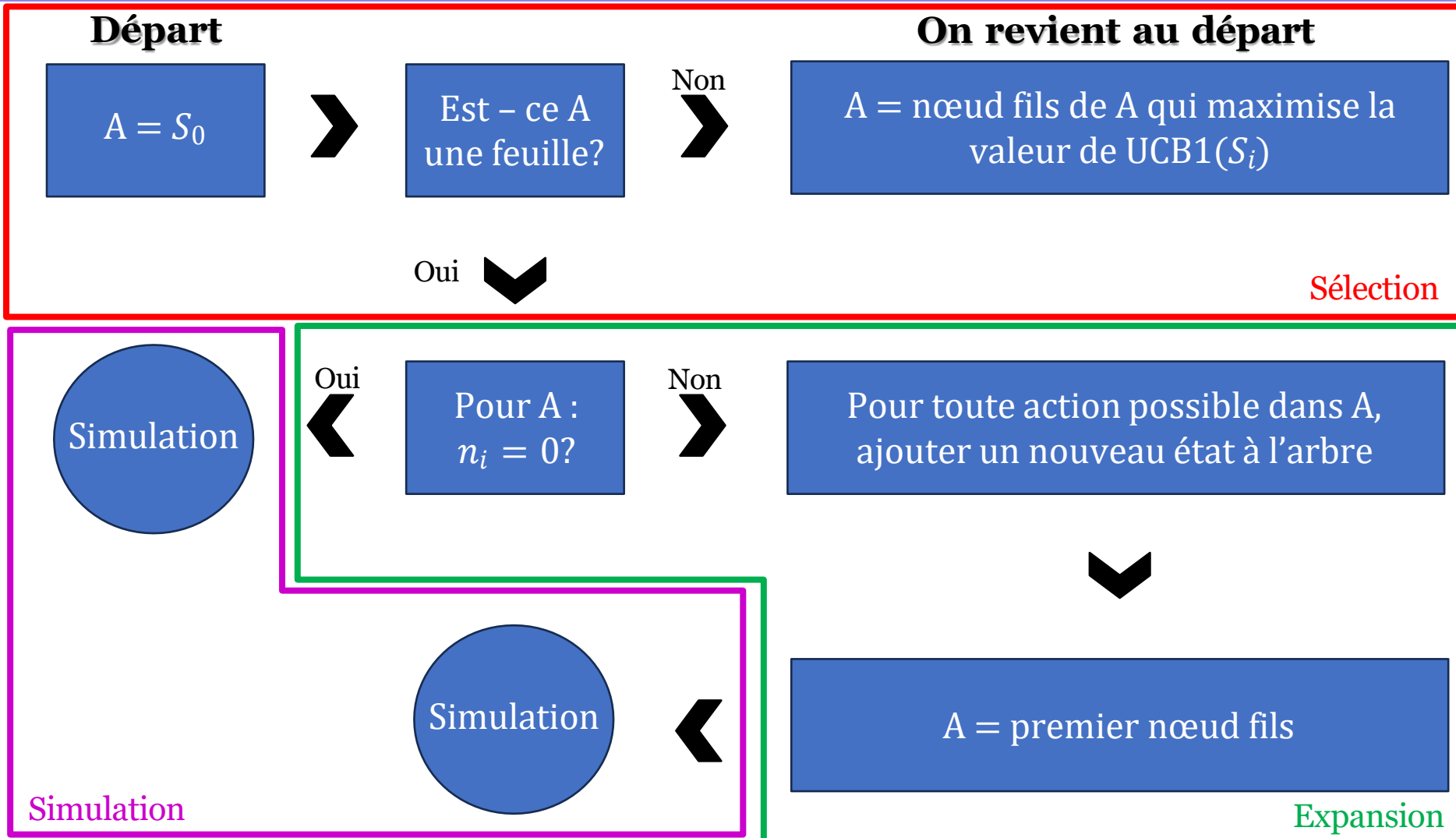
N : nombre de visite du parent

n_i : nombre de visite du successeur S_i

Principe de la Recherche arborescente de Monte Carlo

1. Selection : commence toujours à partir du nœud racine et, à chaque niveau, sélectionne le nœud suivant selon la politique de sélection
2. Expansion : ajouter un nœud fils à l'arbre
3. Simulation : effectue une simulation aléatoire complète du jeu/problème, c'est-à-dire atteint un état terminal et récupère les gains
4. Rétropropagation : propage les gains vers tous les nœuds le long du chemin

Politique de l'arbre



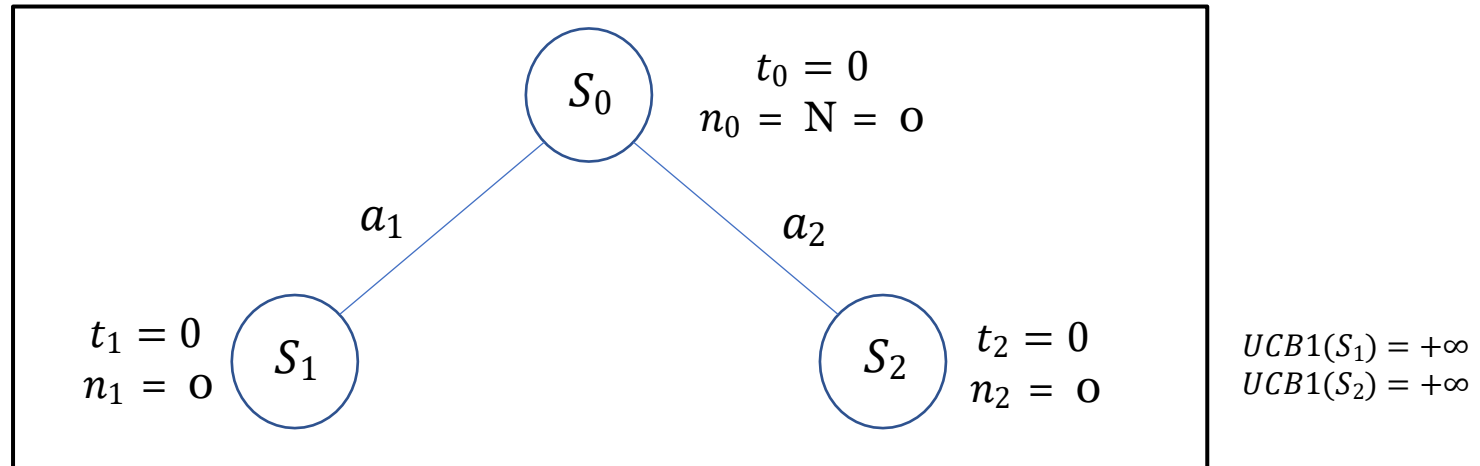
Pratique (1/4)

On considère l'arbre de jeu suivante, et on suppose qu'on obtient pas une base de données

On prend $K = 2$

On notera t_i la valeur totale d'une position S_i
 n_i le nombre de visite de S_i

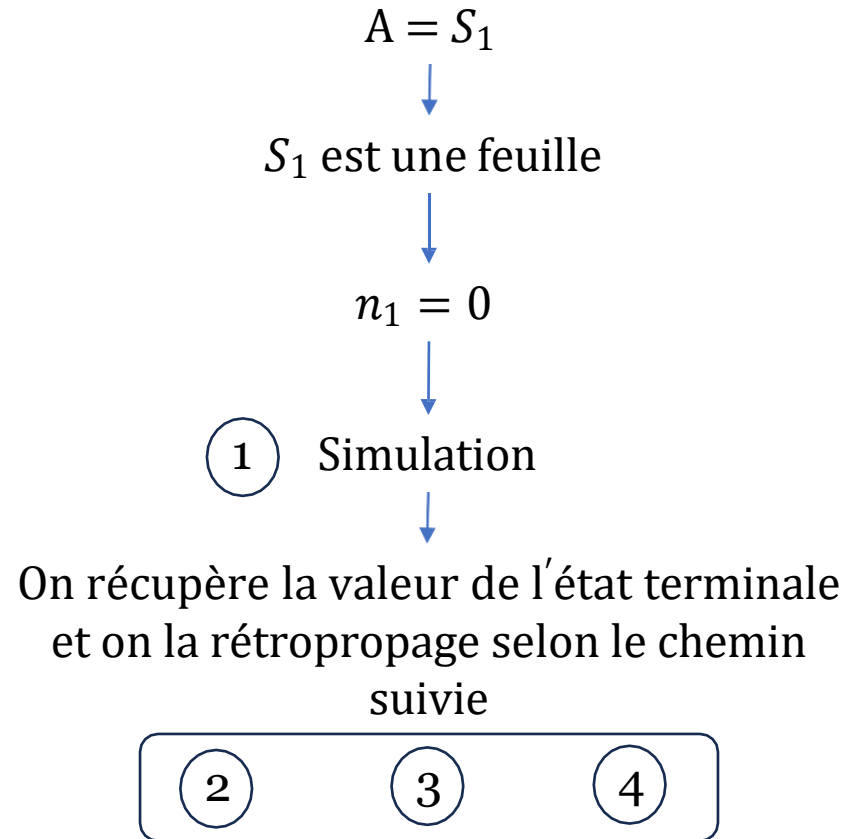
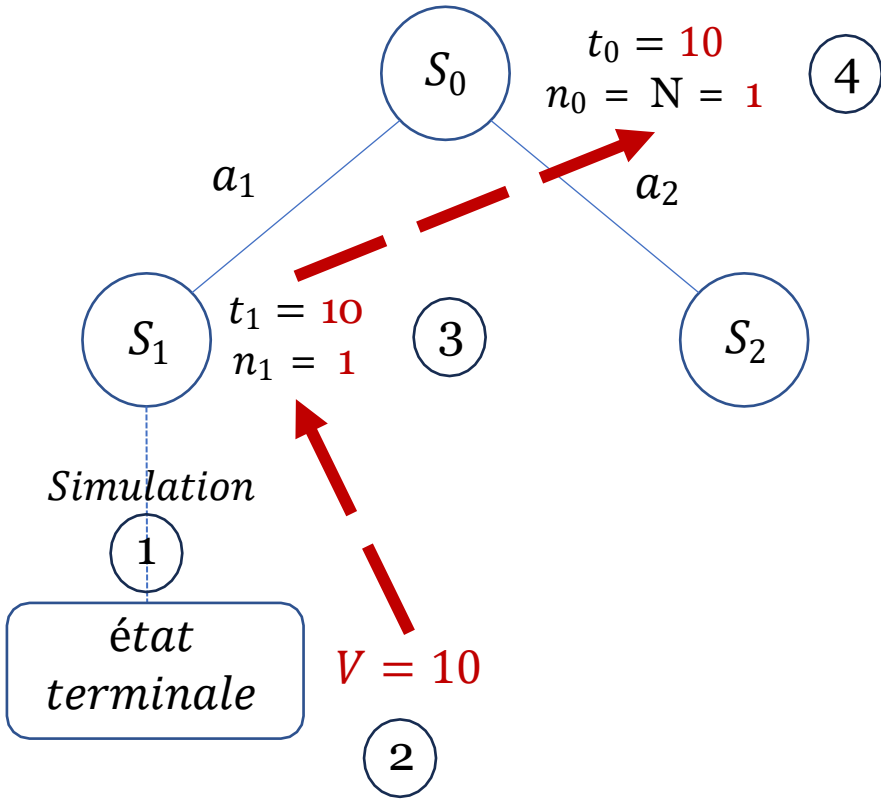
Etat initial de
l'arbre



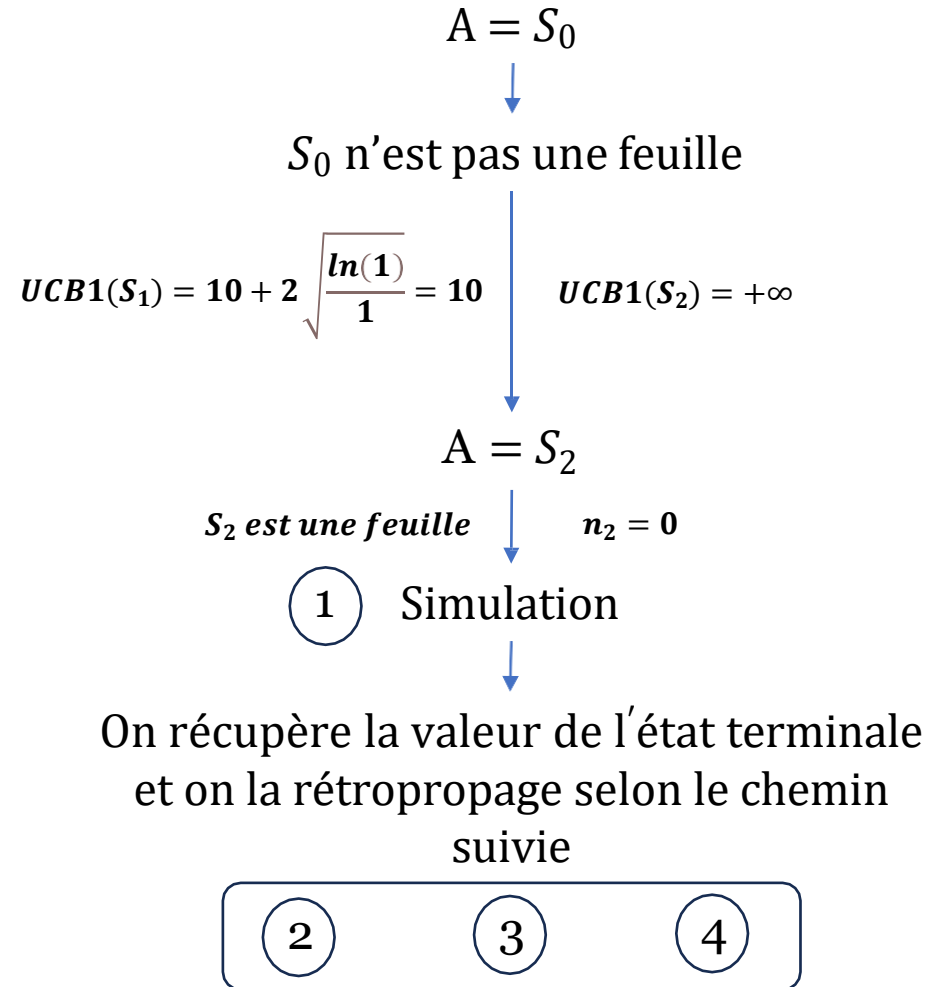
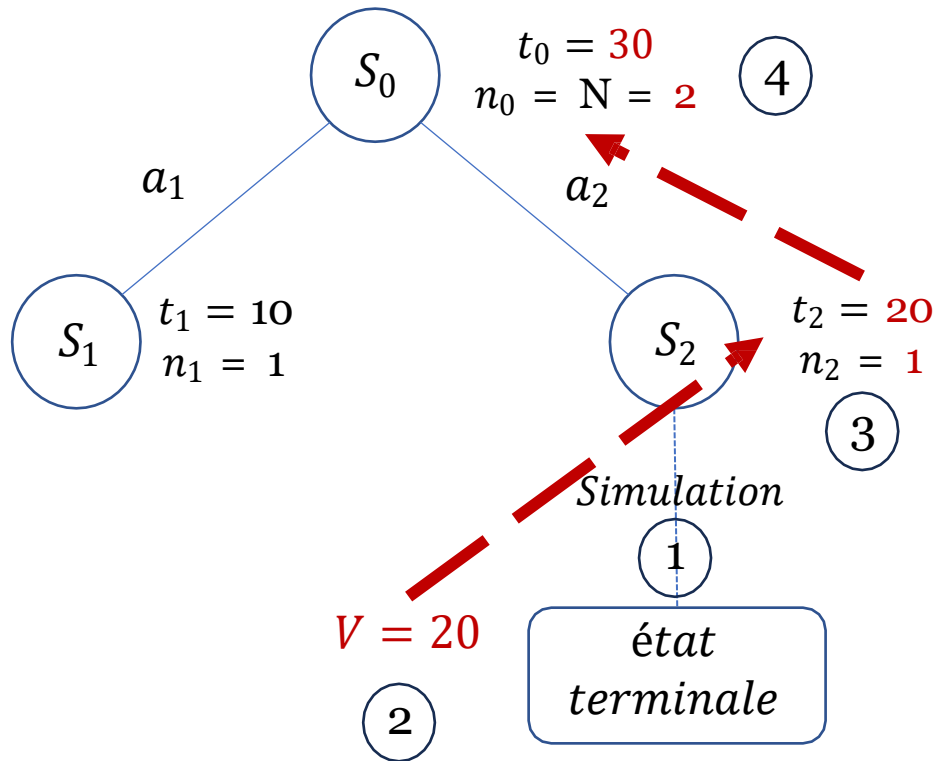
RQ : Si les valeurs de UCB1 sont égaux, alors la sélection se fait de gauche à droite

$A = S_0 \longrightarrow S_0$ n'est pas une feuille $\longrightarrow A = S_1$

Pratique (2/4)



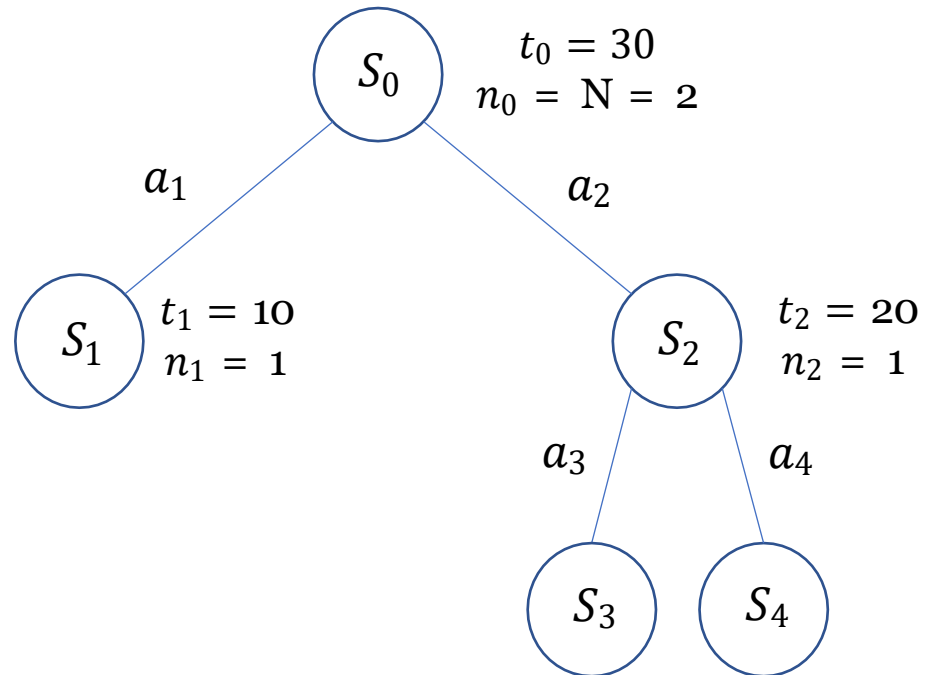
Pratique (3/4)



Pratique (4/4)

$A = S_0$
↓
 S_0 n'est pas une feuille
↓
 $A = S_2$
↓
 $n_2 \neq 0$
↓
On ajoute les états S_3 et S_4 résultats des actions a_3 et a_4
↓
 $A = S_3$
↓
Simulation

$$UCB1(S_1) = 10 + 2 \sqrt{\frac{\ln(2)}{1}} = 11,665 < UCB1(S_2) = 20 + 2 \sqrt{\frac{\ln(2)}{1}} = 21,665$$



Conclusion

Différence de classement :

Etant donné plus de temps, et dans les memes conditions, MCTS tend a performer mieux que l'algorithme MiniMax

- Pour un instant court de délibération [$t \leq 0,5s$], on préfère Minimax
- Sinon, il est mieux d'utiliser MCTS

