Intelligence artificielle

ENSIIE

Semestre 4 en avance – 2024/25

Algorithmes de jeux

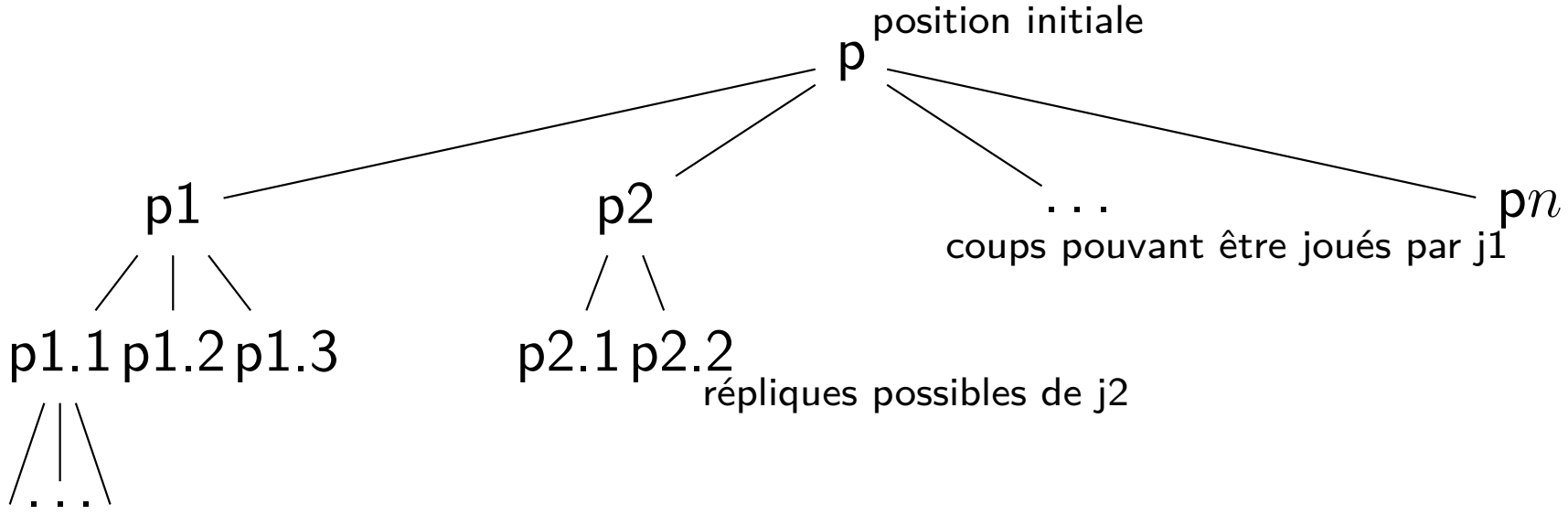
Jeux

Dans ce cours, essentiellement :

- jeux à deux joueurs
- séquentiels
- symétriques
- à somme nulle (si l'un gagne, l'autre perd)
- chaque adversaire a une connaissance de l'ensemble du jeu et de ce que chacun peut faire
- sans aléa

Arbre et/ou

Les différentes parties possibles peuvent être représentée par un arbre



Feuilles : état terminal dans lequel j1 a gagné, fait nul ou perdu.

Procédé minimax

Parcours de l'arbre, ou d'une partie

- Différents coups jusqu'à une profondeur limitée

Évaluation des feuilles

- Fonction d'évaluation qu'on cherche à maximiser
- Évalue une situation pour j1 (la machine)

Remontée des valeurs : principe min ou max selon le joueur

- Une position où j1 joue = la valeur max de ses fils
- Une position où j2 joue = la valeur min de ses fils

(Suppose une stratégie identique pour les 2 joueurs)

Exemple de fonction d'évaluation

Au morpion :

- arbre complet
- gagné = 1, perdu = -1, nul = 0

Aux échecs :

- arbre incomplet !
- $$\sum_{\text{type de pièce } t} w_t \times d_t(a)$$
  avec w_t un poids pour les pièces de type t
  $d_t(a)$ différence entre le nombre de pièces du type t dans la configuration a

Algorithme minimax

```
fonction minimax(nœud)
   si nœud est une feuille alors retourner la valeur de nœud
   sinon si nœud est de type Min alors
           v = +∞
           pour tout fils de nœud faire
               v = min(v, minimax(fils))
   sinon
           v = -∞
           pour tout fils de nœud faire
               v = max(v, minimax(fils))
   retourner v
```

Algorithme minimax

```
fonction minimax_prof_bornée(nœud, p)
   si p > profondeur_max alors retourner évaluation du nœud
   sinon
   si nœud est une feuille alors retourner la valeur de nœud
   sinon si nœud est de type Min alors
           v = +∞
           pour tout fils de nœud faire
               v = min(v, minimax(fils, p + 1))
   sinon
           v = -∞
           pour tout fils de nœud faire
               v = max(v, minimax(fils, p + 1))
   retourner v
```

Caractéristiques de minimax

- Complet si l'arbre est fini
- Complexité en temps : $O(b^m)$
- Complexité en espace : $O(bm)$
- Optimal si l'adversaire est optimal

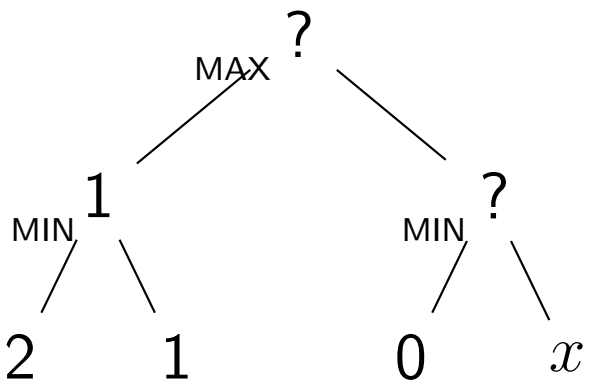
Efficacité de minimax

Aux échecs, $b \simeq 35$
Dans un temps donné, si on veut explorer $b^m = 10^6$ nœuds, on peut donc aller au plus à la profondeur $m = 4$

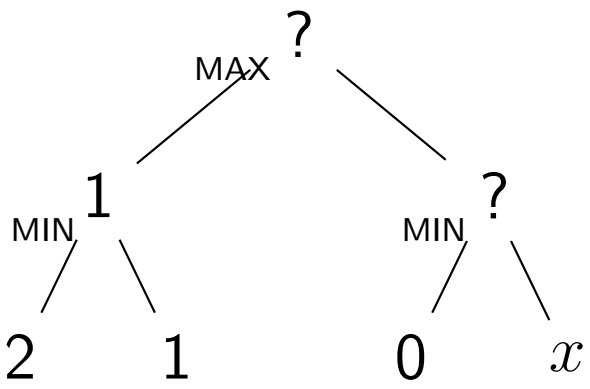
Idée : ne pas calculer de nœuds inutilement

- dont on sait déjà qu'ils seront inférieurs au max ou supérieur au min

Idée

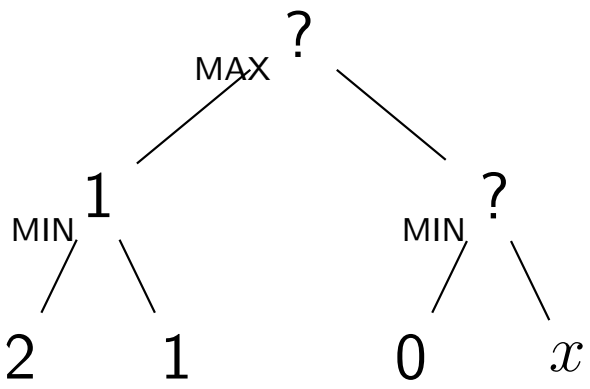


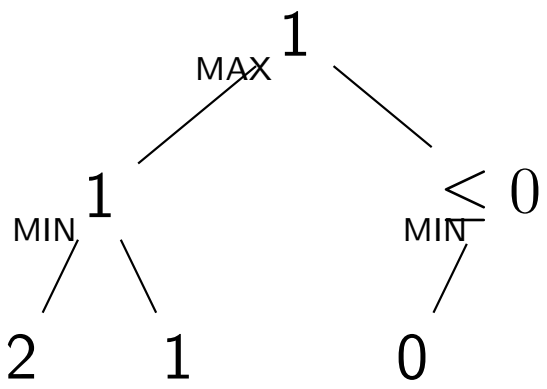
Idée



Quelle que soit la valeur de x, on sait que vaudra 1 à la racine

Idée





Quelle que soit la valeur de x, on sait que vaudra 1 à la racine

Une fois le 0 calculé, on peut couper la branche du x

Élagage $\alpha\beta$

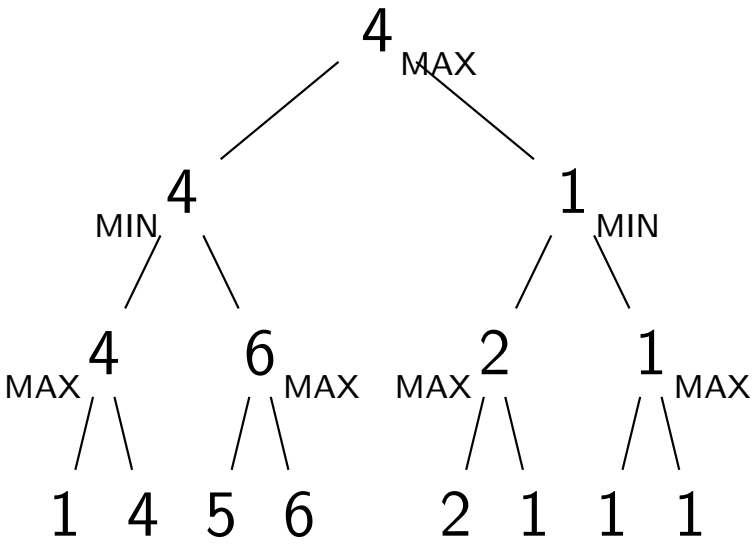
Intuition :
On parcours le nœuds avec une fenêtre $\alpha \ldots \beta$

- on est sûr que MAX renverra au moins α
- on est sûr que MIN renverra au plus β

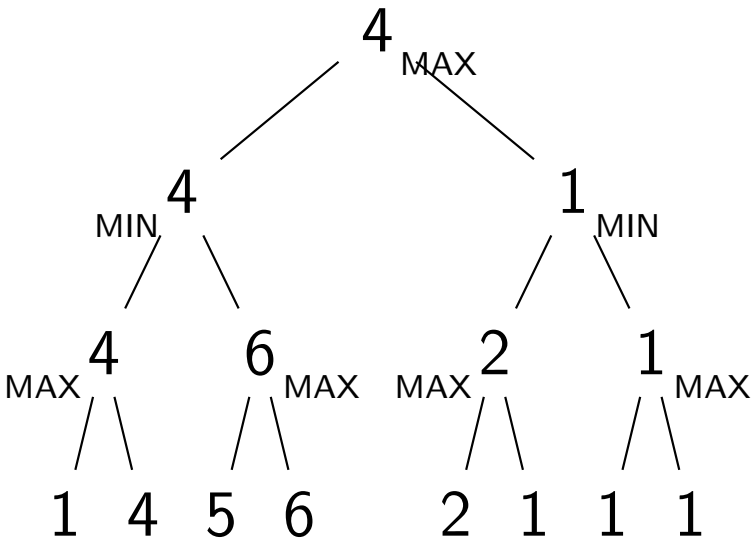
Le calcul de la valeur d'un nœud permet d'affiner la fenêtre pour le calcul de ses frères

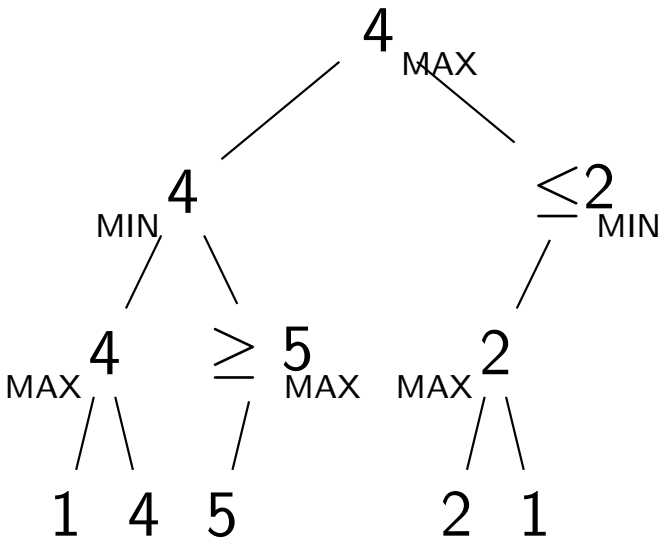
Pas besoin de calculer si fenêtre vide

Exemple



Exemple


4 MAX
4 MIN
1 MIN
4 MAX
6 MAX
2 MAX
1 MAX
1 4 5 6 2 1 1 1


4 MAX
4 MIN
≤2 MIN
4 MAX
≥ 5 MAX
2 MAX
1 4 5 2 1

Algorithme $\alpha\beta$

```
fonction alphabeta(nœud, α, β)
   si nœud est une feuille alors retourner la valeur de nœud
   sinon si nœud est de type Min alors
           v = +∞
           pour tout fils de nœud faire
               v = min(v, alphabeta(fils, α, β))
               si v ≤ α alors  /* coupure alpha */
                   retourner v
               β = min(β, v)
    sinon
           v = -∞
           pour tout fils de nœud faire
               v = max(v, alphabeta(fils, α, β))
               si v ≥ β alors /* coupure beta */
                   retourner v
               α = max(α, v)
    retourner v
```

Recherche Arborescente Monte Carlo

Dans minimax, on explore tout l'arbre jusqu'à une profondeur donnée

Mais peut-être que certains sous-arbres ne sont pas intéressants

Monte Carlo tree search (MCTS)

- Construire l'arbre au fur et à mesure
- En explorant en priorité les nœuds prometteurs (mais pas que)
- En faisant des simulations aléatoires pour évaluer les nœuds

Coulom R (2006) Efficient selectivity and backup operators in Monte Carlo Tree Search. In : International Conference on Computers and Games. Springer, pp 72–83

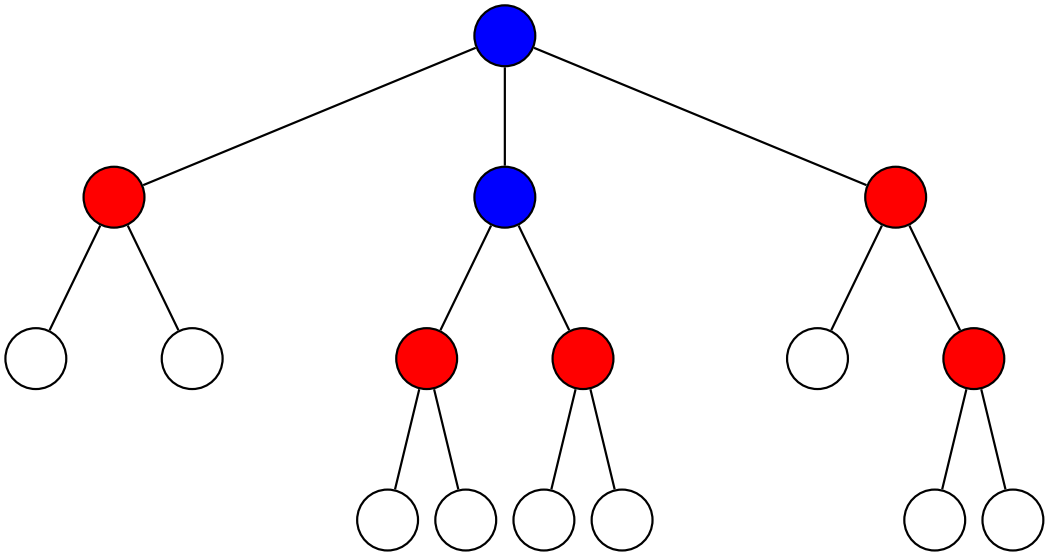
Nœuds visités, complètement explorés

Nœud visité : ●

- une simulation aléatoire a été effectuée pour l'évaluer
- ses successeurs ont été ajoutés à l'arbre

Nœud complétement exploré : ●

- tous ses fils ont été visités



Information dans les nœuds

Dans un nœud s :

- n_s nombre de fois où s ou un de ses descendants a été visité
- w_s évaluation de s (nombre de simulations de s ou un de ses descendants où on a gagné)

Une fois l'arbre suffisamment exploré,
depuis la racine, on cherche à choisir le fils où $\frac{w_s}{n_s}$ est maximal

4 phases

- Sélection
  - choix du prochain nœud non complètement exploré à traiter
- Expansion
  - choix de l'action à traiter, extension de l'arbre suivant cette action
- Simulation
  - partie aléatoire pour évaluer l'état atteint
- Rétropropagation
  - remonte l'évaluation jusqu'à la racine

Sélection

```
s ← racine
tant que n est complètement exploré
  s ← meilleur_successeur(s)
```

Meilleur successeur : compromis entre exploitation (nœud le plus prometteur) et exploration (partie de l'arbre peu explorée)

Upper Confidence Bounds applied for Trees (UCT) :
On maximise

$$\frac{w_i}{n_i} + C\sqrt{\frac{\ln(n_s)}{n_i}}$$

pour i fils du nœud s
où C est une constante, par défaut $\sqrt{2}$

Expansion

Le nœud sélectionné s n'est pas complètement exploré

- on choisit une action possible depuis s, qui mène à un fils f non encore visité
- on ajoute les successeurs de f
- on lance la simulation sur f

Simulation

Depuis f

- on fait des coups au hasard jusqu'à arriver à un état terminal

On évalue w_f comme 1 si on a gagné, 0 si on a perdu

n_f vaut maintenant 1

La simulation n'est pas conservée

Rétropropagation

On remonte l'information obtenue pour f dans tous ses ascendants jusqu'à la racine

```
w ← w_f
s ← f
tant que s n'est pas la racine
  s ← père de s
  n_s ← n_s + 1
  w_s ← w_s + w
```

Algorithme global

```
tant qu'il reste des ressources (temps, mémoire)
  sélection
  expansion
  simulation
  rétropropagation
choisir le successeur de la racine le plus prometteur
```

Analyse

Avantages :

- peut s'adapter aux ressources disponibles
- ne se perd pas dans des sous-arbres mauvais
- spécialement en cas de fort branchement
- pas besoin d'heuristique

Inconvénients :

- complexité en mémoire élevée (on garde l'arbre, $O(b^m)$)
- pas de garantie qu'on ne puisse pas en fait faire mieux en passant par une branche peu explorée

Extensions

- Autres fonctions de choix pour la sélection
- Utilisation de *machine learning*, en particulier de réseaux de neurones profonds, à la place des simulations AlphaGo