

INF8175 - Intelligence artificielle

Méthodes et algorithmes

Module 2: Recherche adversarielle



POLYTECHNIQUE
MONTRÉAL

Quentin Cappart

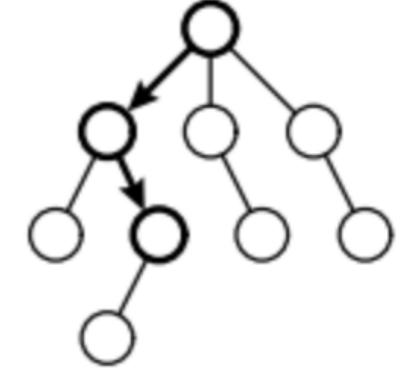
Contenu du cours

Raisonnement par recherche (essais-erreurs avec de l'intuition)

Module 1: Stratégies de recherche

Module 2: Recherche en présence d'adversaires

Module 3: Recherche locale



Raisonnement logique

Module 4: Programmation par contraintes

Module 5: Agents logiques

ΣΣ ΣΣΣΣ Stench		Breeze	PIT
	Breeze ΣΣ ΣΣΣΣ Stench Gold	PIT	Breeze
ΣΣ ΣΣΣΣ Stench		Breeze	
START	Breeze	PIT	Breeze

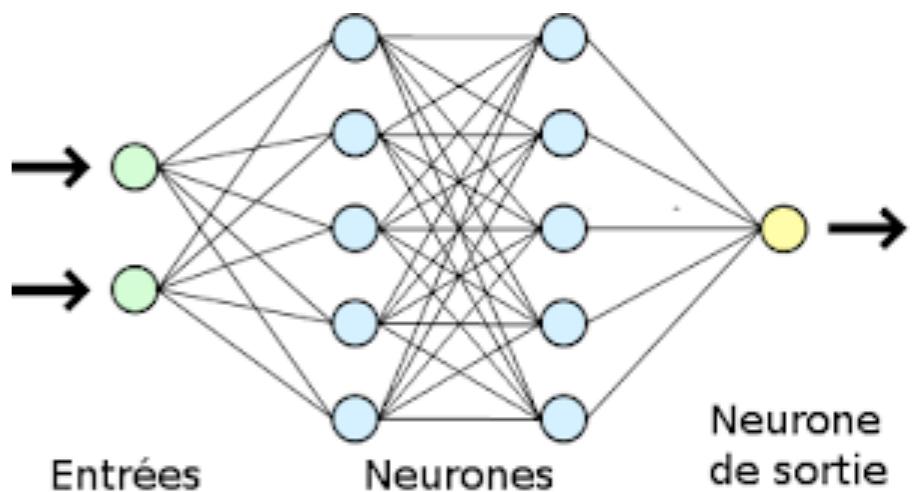
Raisonnement par apprentissage

Module 6: Apprentissage supervisé

Module 7: Réseaux de neurones et apprentissage profond

Module 8: Apprentissage non-supervisé

Module 9: Apprentissage par renforcement



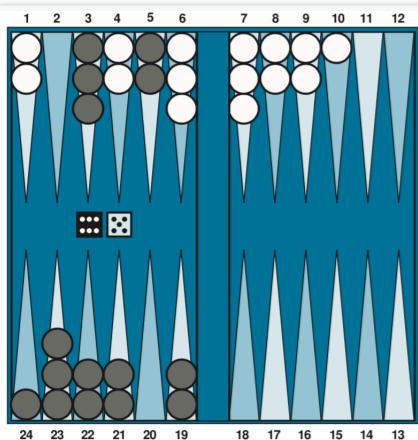
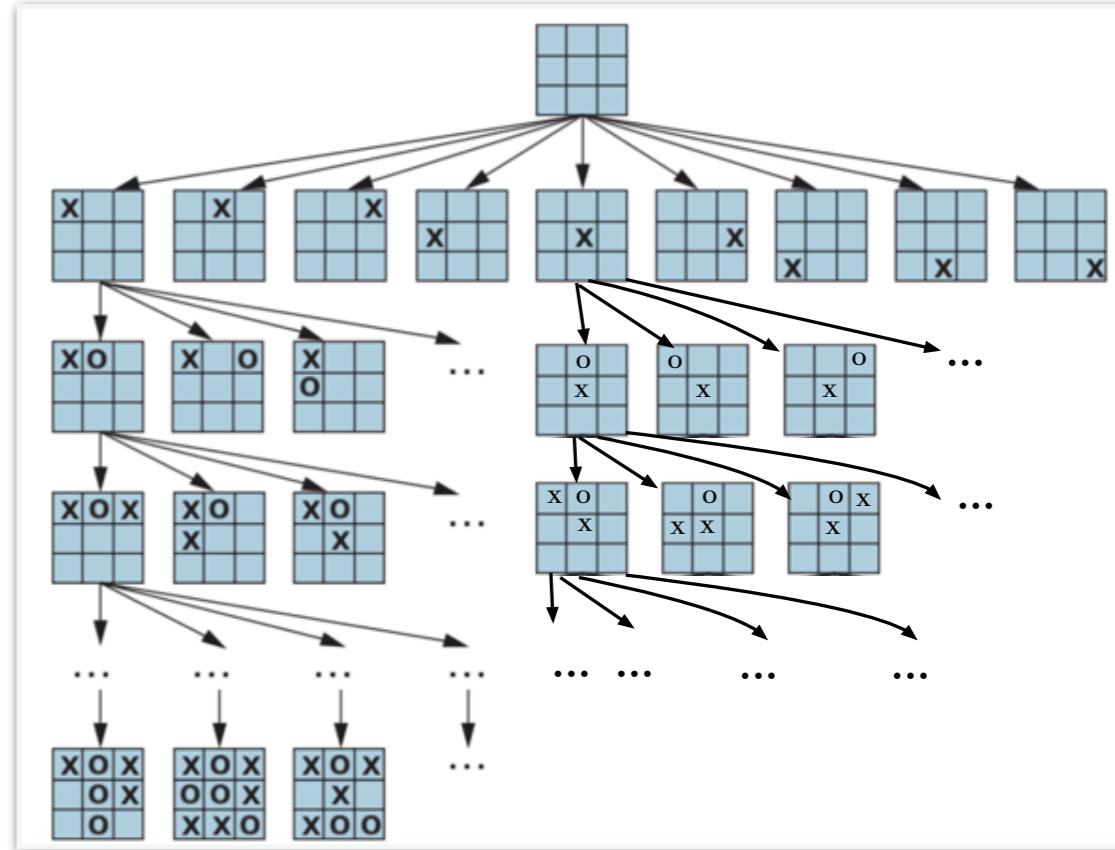
Considérations pratiques et sociétales

Module 10: Utilisation en industrie, éthique, et philosophie

Table des matières

Recherche adversarielle

1. Motivation de la recherche adversarielle
2. Catégorisation d'environnements compétitifs et des jeux
3. Stratégie de recherche minimax
4. Mécanisme du *alpha-beta pruning*
5. Conception de fonctions d'évaluation (heuristiques)
6. Mécanismes supplémentaires d'amélioration
7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
9. Exemples et historique d'agents dédiés aux jeux



Motivation

Limitation des stratégies de recherche vues

Nos agents actuels sont conçus pour réaliser la meilleure séquence d'actions afin d'atteindre un objectif

Hypothèse sous-jacente: l'agent est seul à agir dans son environnement, et rien n'interfère avec lui

Avantage: permet de modéliser un grand nombre de situations et de problèmes

Inconvénient: l'hypothèse est trop restrictive pour des situations impliquant d'autres agents

Environnement compétitif

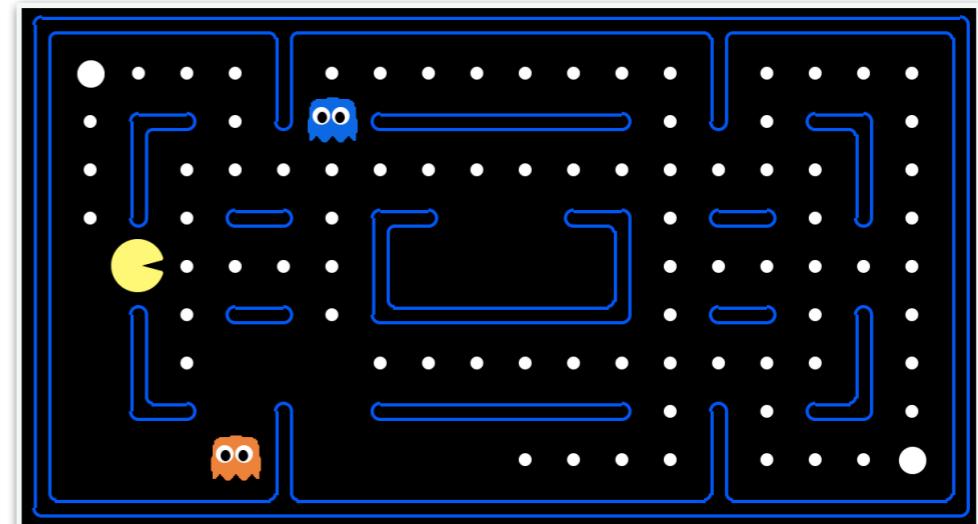
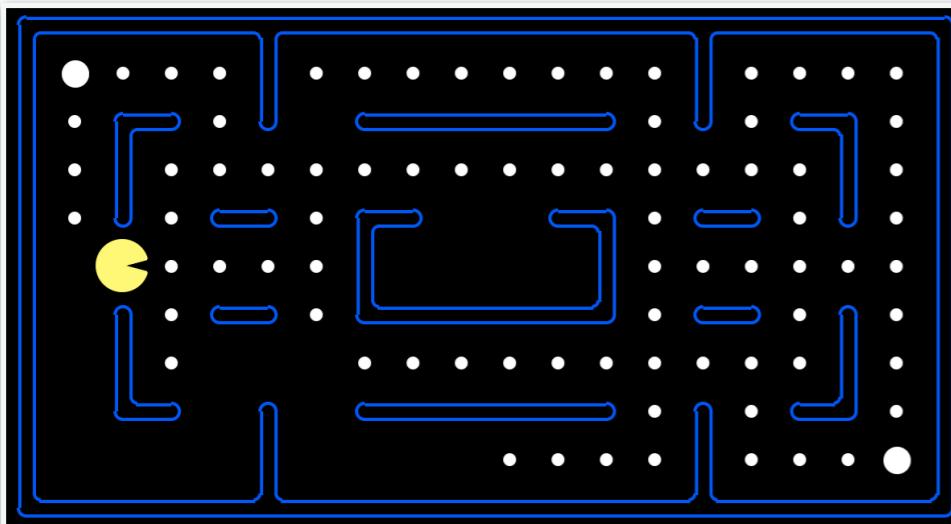
Environnement dans lequel plusieurs agents évoluent afin de réaliser un objectif qui leur est propre

Caractéristiques: engendre généralement de la **compétition** ou de la **coopération** entre les agents



Recherche adversarielle

Stratégie de recherche qui tient compte du fait que l'agent doit réaliser son objectif dans un environnement compétitif, contre un ou des adversaires



Environnements compétitifs réels



Exemple 1: achat d'un bien avec plusieurs acheteurs

Vous êtes plusieurs à vouloir acheter un bien

Objectif de l'acheteur: acheter le bien au prix le plus bas

Objectif du vendeur: vendre son bien au plus haut prix

Exemple: achat d'une maison, vente aux enchères, appel d'offre



Exemple 2: fixer le prix d'un produit

Plusieurs entreprises ont mis en vente un produit similaire

Objectif de chaque entreprise: avoir une bonne part de marché

Action possible pour une entreprise: baisser le prix, mais pas trop !



Exemple 3: négociations

Vous avez un objectif en tête, mais votre interlocuteur aussi !

Chacun d'entre vous veut tirer un avantage de l'accord

Implique généralement des objectifs communs, et individuels

Exemple: négociation de salaires, marchandise, etc.

La plupart des environnements réels sont compétitifs

Environnement compétitif populaire: les jeux

Les jeux constituent l'environnement compétitif par excellence pour développer des nouvelles stratégies

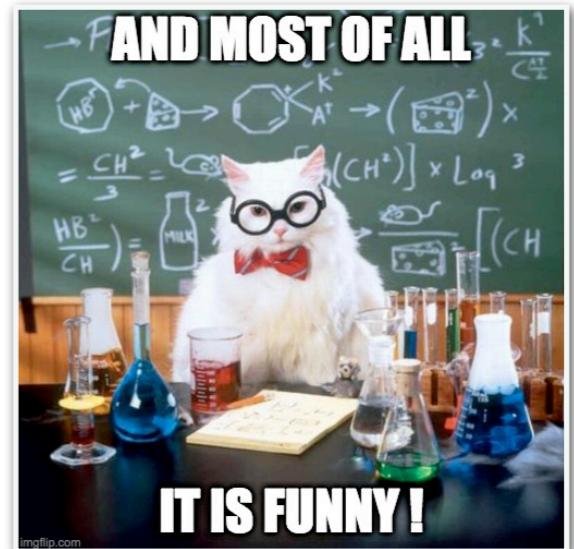
Raison 1: environnement contrôlable et facilement représentable

Raison 2: règles bien définies et sans ambiguïté

Raison 3: facilité pour comparer et évaluer les agents

Raison 4: évite les aléas des environnements réels

Raison 5: différents niveaux de complexité en fonction du jeu choisi



Différentes catégories de jeux

Il existe une multitude de jeux, définis par différentes caractéristiques

Axes principaux: aléatoire, nombre de joueurs, information disponible, *somme du jeu*, temps-réel, etc.

Présence d'aléatoire

Jeu déterministe: aucun aléatoire n'est impliqué

Jeu stochastique: une part d'aléatoire est présente



Nombre de joueurs

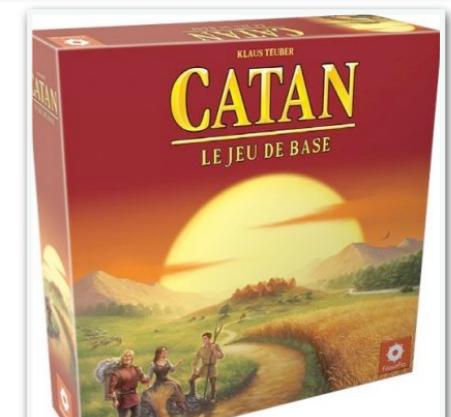
Catégorie: un seul ou plusieurs adversaires

Plusieurs adversaires: les stratégies des joueurs sont parfois alignées (coopération)

Dynamique du jeu: donne lieu à des alliances, trahisons, conflits d'intérêts, etc.

Ses stratégies peuvent fluctuer au cours de la partie en fonction de notre intérêt

Exemple: à un moment du jeu, on se met ensemble contre le joueur le plus fort



Informations disponibles

Information parfaite: on connaît parfaitement l'environnement

Information incomplète: certaines données de l'environnement sont inconnues

Certaines informations ne peuvent être disponibles qu'à un joueur spécifique



Différentes catégories de jeux

Somme du jeu

Jeu à somme nulle: ce qui est bon pour un joueur est mauvais pour l'autre (compétition pure)

Formellement: la somme des gains d'un joueurs avec les pertes des autres vaut 0

Exemple: aux échecs, une victoire (1) d'un joueur engendre une défaite de l'autre joueur (-1)

Jeu à somme non nulle: ce qui est bon pour un joueur n'est pas forcément mauvais pour l'autre

Même à deux joueurs, la coopération ou l'indifférence est possible



Dynamique du jeu

Tour par tour: chaque joueur effectue leurs actions tour par tour

Limitation de temps: le temps de décision pour chaque joueur est limité (exemple: tournoi d'échecs)

Temps réel: chaque joueur effectue ses actions en temps réel



Jeux physiques

Caractéristique: implique généralement des aspects mécanique et de robotique

Difficulté: l'environnement plus difficile à contrôler (qu'est-ce qu'une action ?)



Conclusion

Il existe une multitude de type de jeux différents

Chaque jeu possède certaines caractéristiques, à des degrés différents

Catégorie de référence



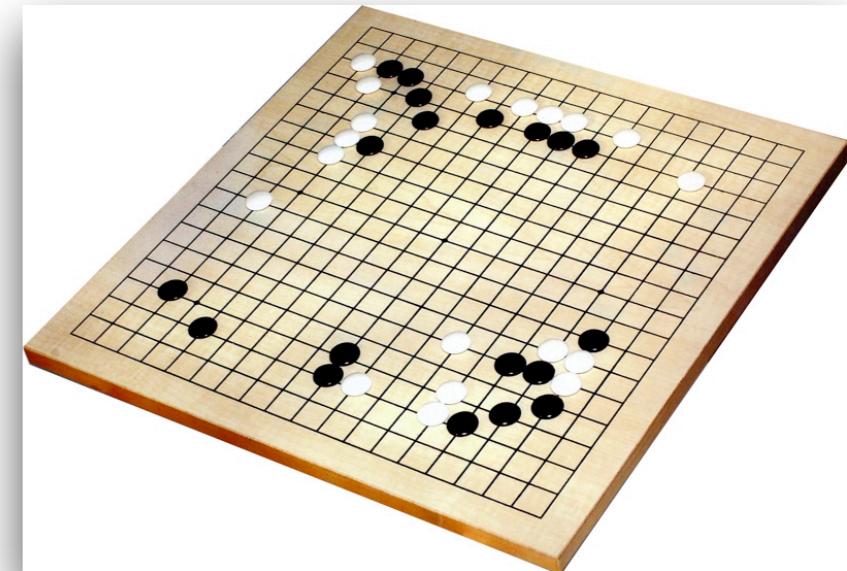
Quelle catégorie de jeux va t-on considérer pour la conception de nos algorithmes ?

Cheminement scientifique: Les chercheurs en IA ont commencé à s'intéresser à une catégorie de jeux

Catégorie de référence: deux joueurs, déterministe, tour par tour, à information parfaite, et à somme nulle

Raison: il s'agit de la catégorie conceptuellement la plus simple et servant de base pour les autres variantes

Par contre, elle offre déjà un défi intéressant de part le nombre immense de possibilités :-)



Dans un premier temps: c'est à cette catégorie de jeux que nous nous intéresserons principalement

Dans un deuxième temps: on verra ensuite comment étendre nos algorithmes à d'autres catégories

Ce domaine est très vaste, et nous ne pourront malheureusement pas tout couvrir

Formalisation standard d'un jeu



Jeux déterministe, tour par tour, à information parfaite, à somme nulle

Un jeu de cette famille peut être formellement défini par les éléments suivants:

S : un ensemble d'états (contenant un état initial (s_0) et un/des états terminaux)

A : un ensemble d'actions

P : l'ensemble des joueurs de la partie (2 dans notre cadre)

turn : $(S \rightarrow P)$: le joueur devant jouer à l'état s

actions : $(S \rightarrow 2^A)$: l'ensemble des actions permises à l'état s

transition : $(S \times A \rightarrow S)$: retourne le nouvel état émanant d'une action à un état précédent

isTerminal : $(S \rightarrow \{0,1\})$: indique si un état est terminal (le jeu est terminé)

utility : $(S \times P \rightarrow \mathbb{R})$: score obtenu pour un joueur à un état terminal spécifique

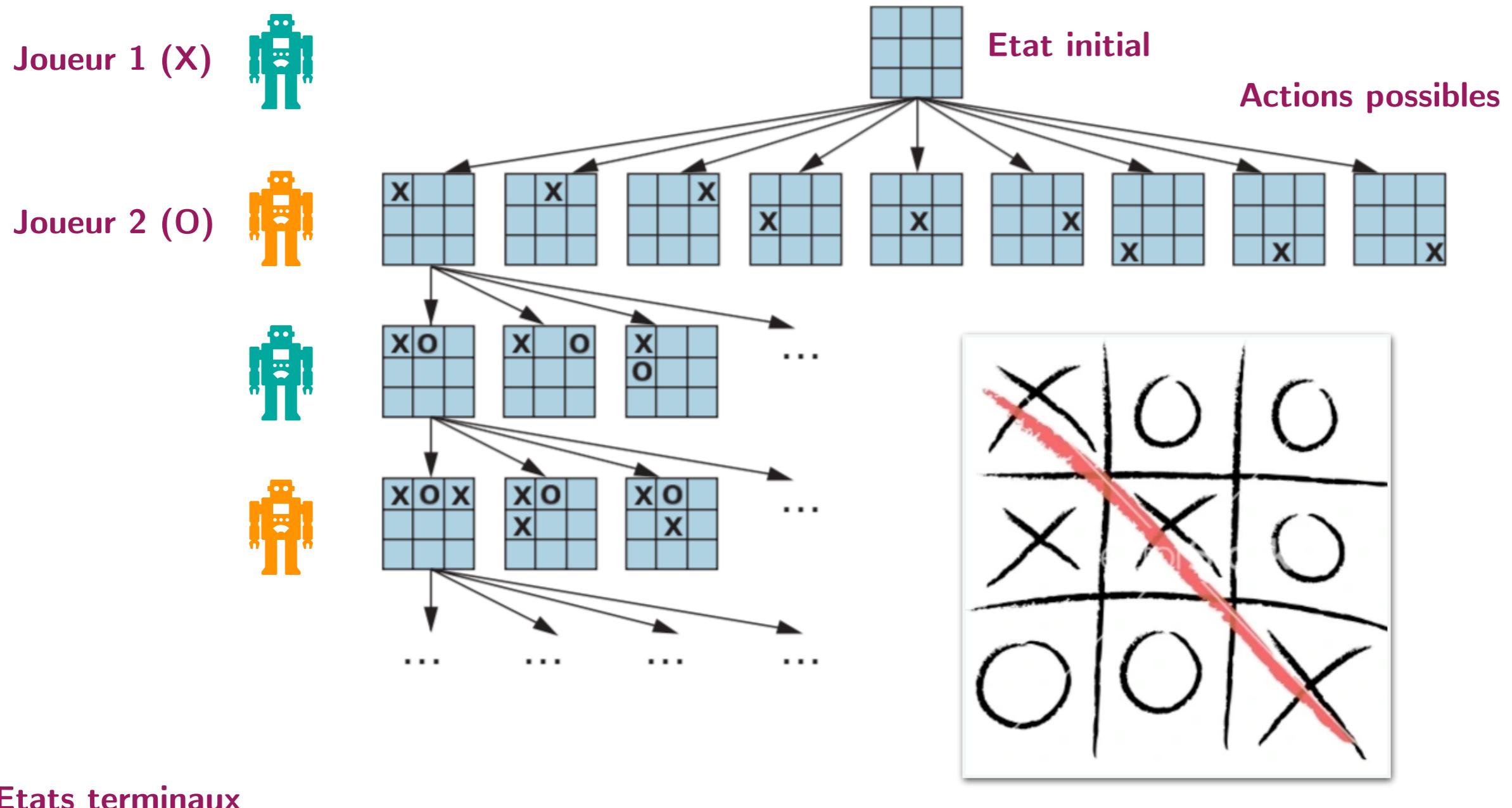
En réalité, c'est très similaire à la définition d'un problème de recherche (module 1)

Similarité 1: l'ensemble des états, des actions, et la fonction de transition forment un graphe des états

Similarité 2: la dynamique du jeu peut ainsi également se représenter comme un arbre de recherche

Differences: les autres éléments rajoutent le concept de joueurs multiples, avec chacun leur score

Exemple: Tic-tac-toe



Score du premier joueur -1 0 1 

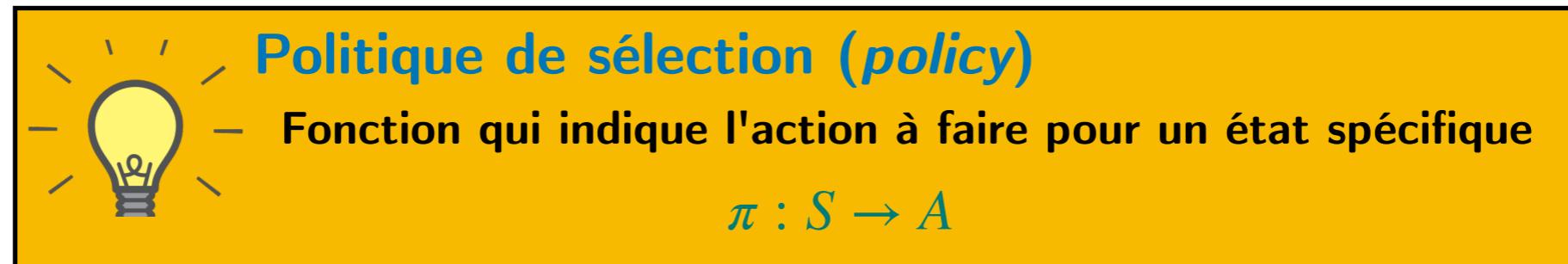
Comme le jeu est à somme nulle, le score pour le joueur 2 est l'opposé de celui du joueur 1

1 0 -1 

Solution pour un joueur

Une solution pour un joueur revient à savoir quoi jouer pour chaque état

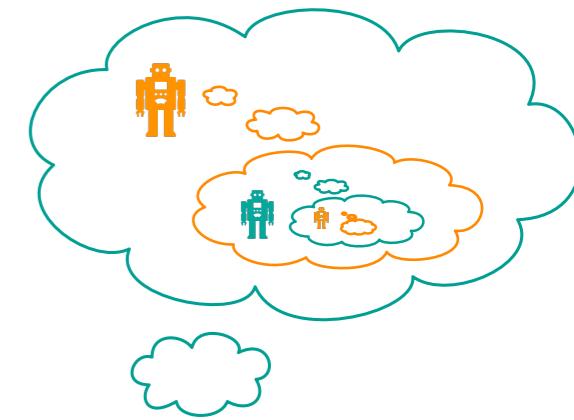
Terminologie: on parle également d'une **politique de sélection**



Attention: notez la différence avec le module 1 où une solution est une séquence d'action

A l'idéal: on souhaite une politique qui va nous amener à gagner le jeu

Difficulté: on ne sait pas ce que l'adversaire va jouer !



Répondre à cette question amène à définir la notion de solution optimale



Objectif du premier joueur: maximiser son score (fonction d'utilité une fois un état terminal atteint)

Objectif du deuxième joueur: minimiser le score du premier joueur (jeu à somme nulle)

Hypothèse: on suppose que l'adversaire à un comportement rationnel (optimal)

Autrement dit: il va jouer au mieux pour réaliser son objectif

Conséquence: cela revient à prévoir tous les coups à l'avance dans le jeu

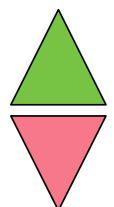
Intuition d'une stratégie de recherche optimale

Regardons ce comportement sur un arbre de recherche

Le premier joueur veut maximiser son score (joueur MAX)

Le deuxième joueur veut minimiser le score du premier joueur (joueur MIN)

Idée: est de considérer que l'adversaire (et nous-même) allons jouer au mieux pour réaliser notre objectif

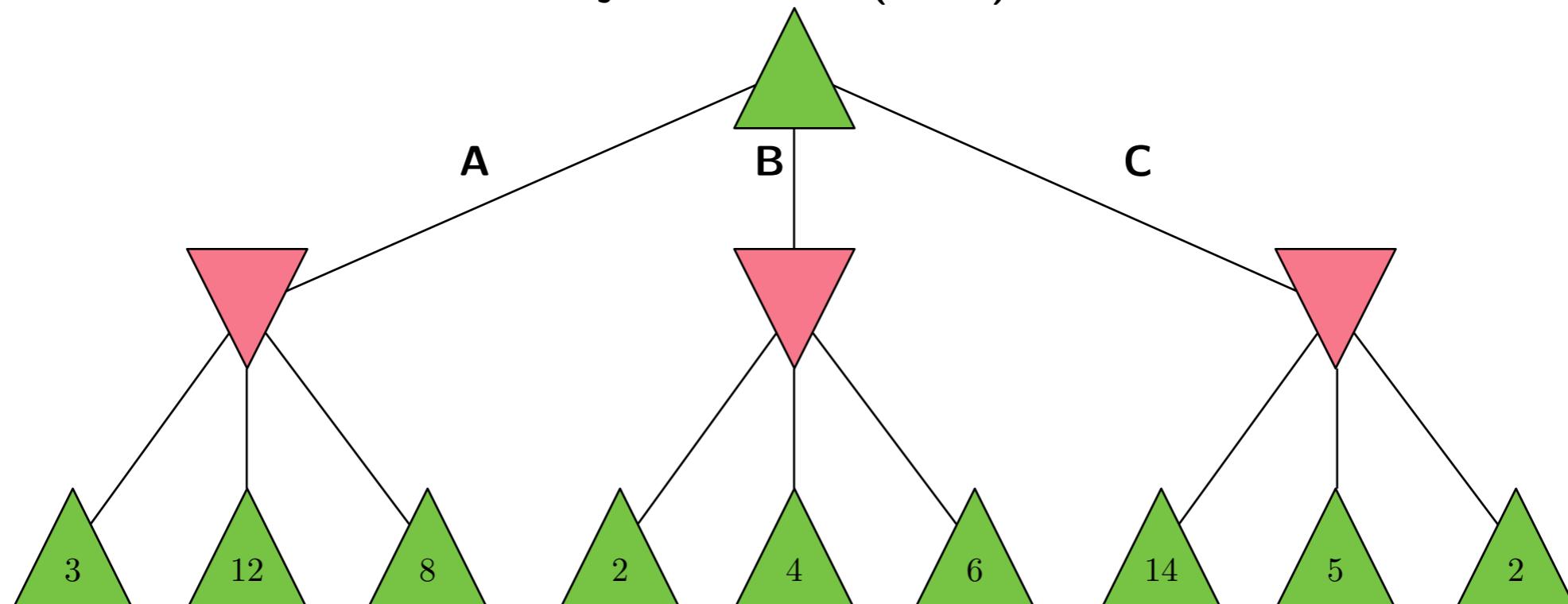


: noeud correspondant au tour du joueur MAX (pointe vers le haut pour maximiser)

: noeud correspondant au tour du joueur MIN (pointe vers le bas pour minimiser)

Noeud racine: indique le joueur qui doit jouer (3 actions possibles)

Noeuds terminaux: contiennent le score du joueur actuel (MAX)



Quelle est la meilleure action à réaliser pour le joueur actuel (MAX) ?

Intuition d'une stratégie de recherche optimale

Première idée

Principe: le meilleur score atteignable pour MAX est 14 via l'action C

Action: le joueur MAX décide de réaliser l'action C

?

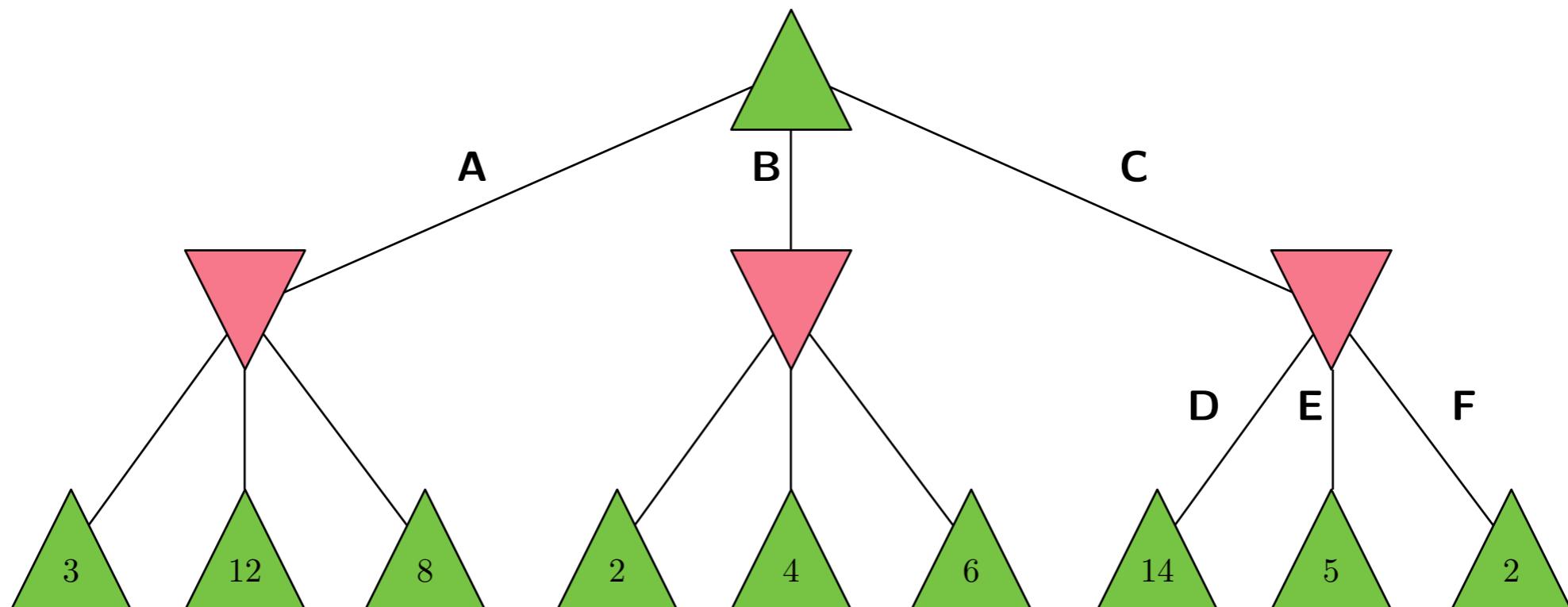
Que pensez vous de ce raisonnement ?

Faille: on ne tient pas compte de ce que va faire le joueur MIN



Comportement optimal: le joueur MIN réagira en choisissant l'action F

Résultat: cela amène à un score de 2 pour MAX



Intuition d'une stratégie de recherche optimale

Deuxième idée

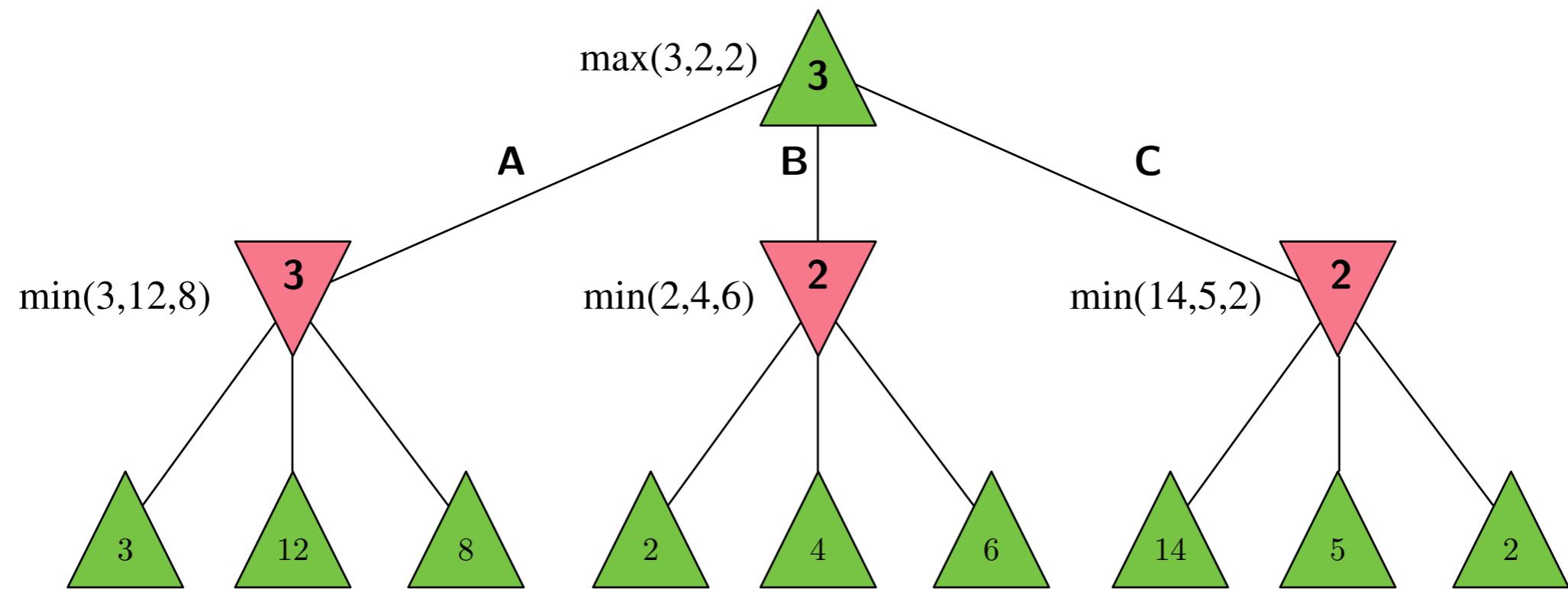
Principe: pour chaque état, calculer le score pouvant être obtenu

Hypothèse: les deux joueurs ont un comportement optimal (ils sont rationnel)

Le score d'un état MIN reprend le score minimum de ses successeurs

Le score d'un état MAX reprend le score maximum de ses successeurs

Calcul du score: de manière récursive depuis les états terminaux



L'action optimale pour MAX est A, ce qui lui amènera à un score de 3 (si MIN fait un coût optimal)

Valeur Minimax

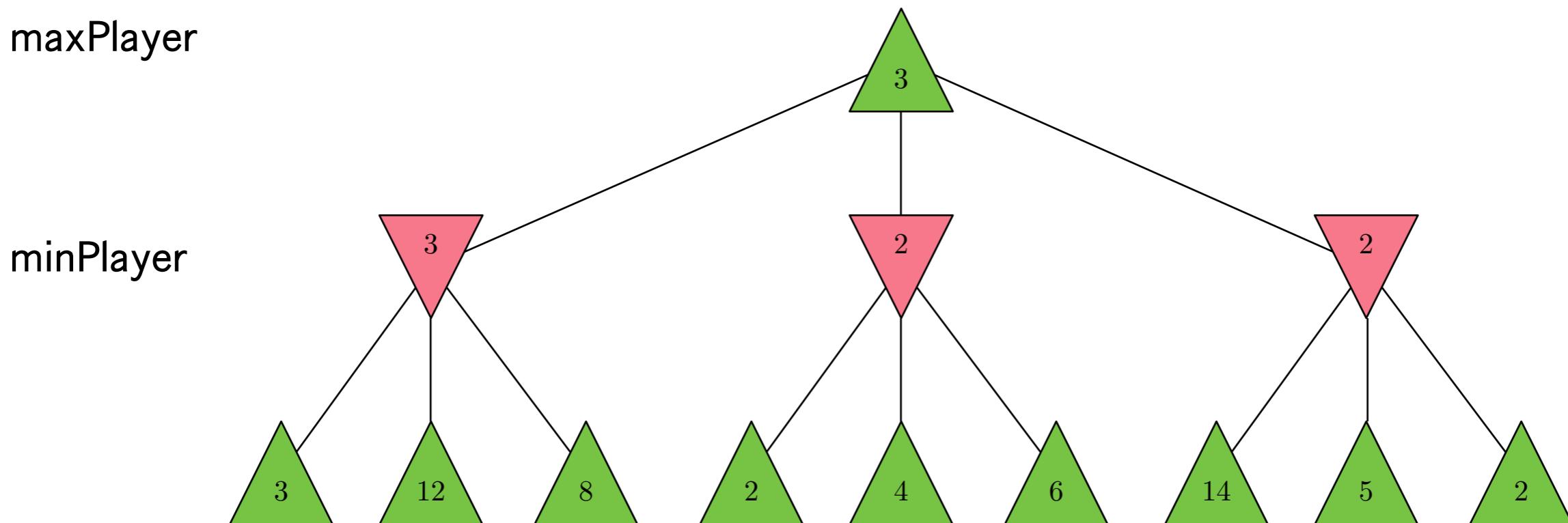


Valeur Minimax

Valeur qui indique le score final (pour le joueur MAX) qui sera obtenu à partir d'un état, si les deux joueurs jouent de façon optimale

$$\text{minimax}(s) = \begin{cases} \text{utility}(s, \text{maxPlayer}) & \text{if } \text{isTerminal}(s) \\ \max_{a \in \text{actions}(s)} \text{minimax}(\text{transition}(s, a)) & \text{if } \text{turn}(s) = \text{maxPlayer} \\ \min_{a \in \text{actions}(s)} \text{minimax}(\text{transition}(s, a)) & \text{if } \text{turn}(s) = \text{minPlayer} \end{cases}$$

maxPlayer

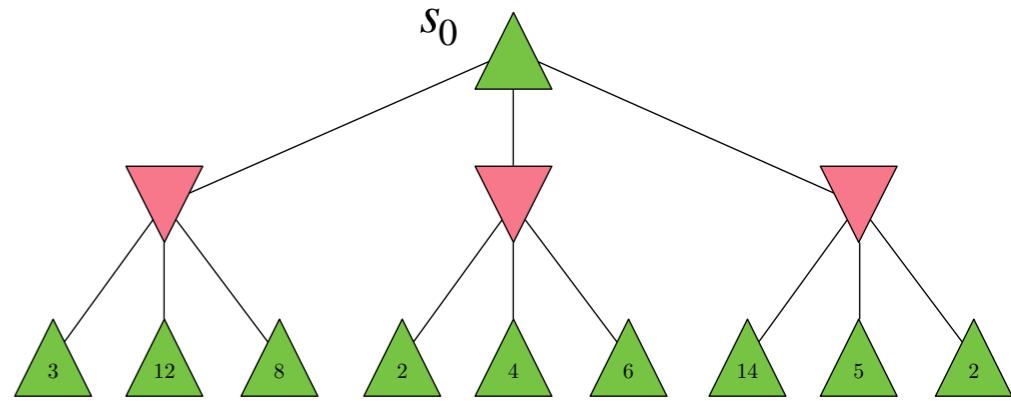


Stratégie de recherche minimax

Formalisons ce fonctionnement par un algorithme

```
minimaxSearch( $s_0$ ) :  
   $\langle v, m \rangle = \text{maxValue}(s_0)$   
  return  $\langle v, m \rangle$ 
```

On lance la recherche au noeud racine
On se met du point de vue du joueur MAX
On retourne la meilleure action (m) et sa valeur (v)



```
maxValue( $s$ ) :  
  if isTerminal( $s$ ) :  
    return  $\langle \text{utility}(s, \text{maxPlayer}), \perp \rangle$   
   $v^* = -\infty$   
   $m^* = \perp$   
  for each  $a \in \text{actions}(s)$   
     $s' = \text{transition}(s, a)$   
     $\langle v, \_ \rangle = \text{minValue}(s')$   
    if  $v > v^*$  :  
       $v^* = v$   
       $m^* = a$   
  return  $\langle v^*, m^* \rangle$ 
```

Score pour les états MAX

Terminal: on retourne le score
Le score est initialisé à la plus petite valeur
Aucune action n'est choisie à ce stade
On analyse toutes les actions

On prend le score des états successeurs (MIN)
S'il est meilleur (plus grand) que celui actuel...
On le retient, ainsi que l'action

On retourne la meilleure action et sa valeur

```
minValue( $s$ ) :  
  if isTerminal( $s$ ) :  
    return  $\langle \text{utility}(s, \text{maxPlayer}), \perp \rangle$   
   $v^* = \infty$   
   $m^* = \perp$   
  for each  $a \in \text{actions}(s)$   
     $s' = \text{transition}(s, a)$   
     $\langle v, \_ \rangle = \text{maxValue}(s')$   
    if  $v < v^*$  :  
       $v^* = v$   
       $m^* = a$   
  return  $\langle v^*, m^* \rangle$ 
```

Valeur obtenue à la racine: reflète le résultat obtenu si les deux joueurs sont optimaux

L'action choisie est celle permettant ce scénario

Illustration de l'algorithme minimax

Exécution de l'algorithme

On cherche à calculer la valeur de l'état a

Etape 1 : $a = \max(b, c, d)$

Etape 2 : $a = \max(\min(3, 12, 8), c, d)$

Etape 3 : $a = \max(3, c, d)$

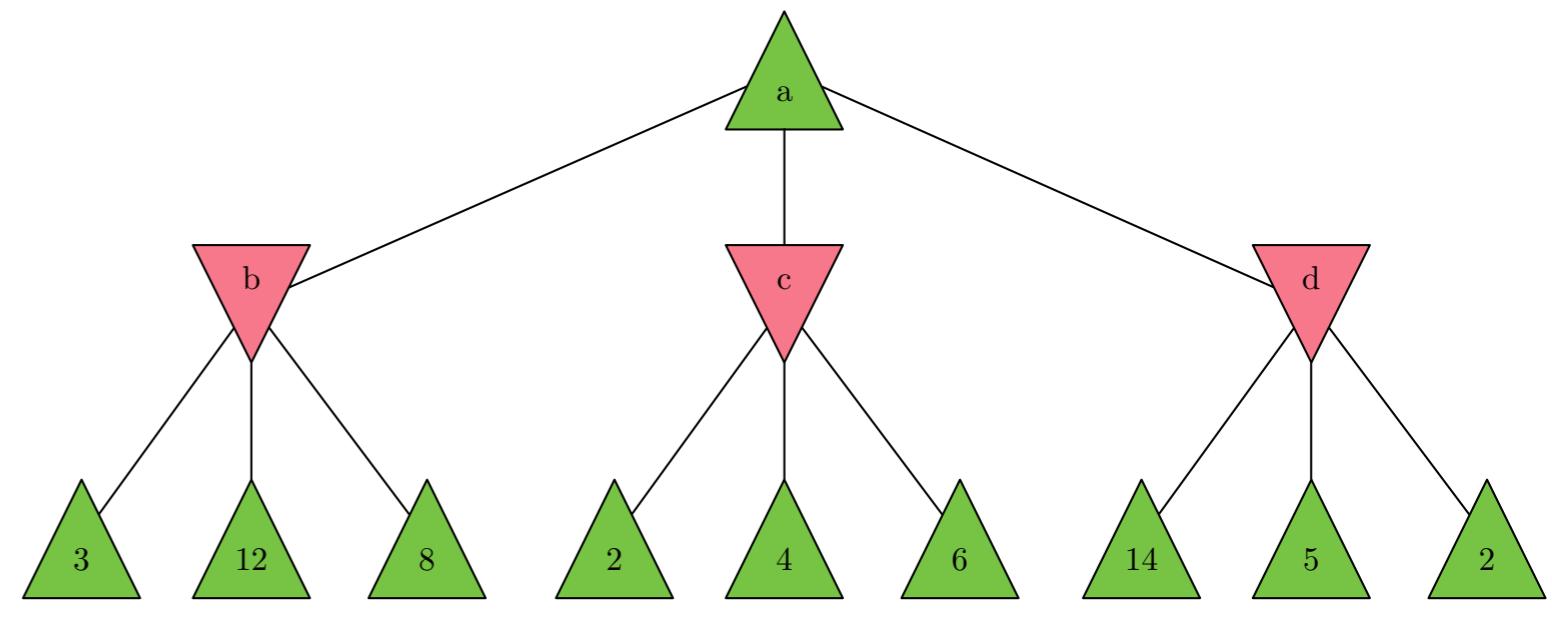
Etape 4 : $a = \max(3, \min(2, 4, 6), d)$

Etape 5 : $a = \max(3, 2, d)$

Etape 6 : $a = \max(3, 2, \min(14, 5, 2))$

Etape 7 : $a = \max(3, 2, 2)$

Etape 8 : $a = 3$



?

Quel type de recherche cet algorithme exécute ?

Efficacité de l'algorithme

Analyse: l'algorithme revient à faire une **recherche en profondeur exhaustive**

Complet et optimal: en supposant que le jeu ne soit pas infini

Complexité temporelle: $\mathcal{O}(b^m)$ (très coûteux) - requiert d'explorer TOUT l'arbre

Complexité spatiale: $\mathcal{O}(bm)$ (très efficace)

Mauvaise nouvelle: l'algorithme impraticable pour la plupart des jeux

Bonne nouvelle: il sert néanmoins de base pour construire des algorithmes plus efficaces

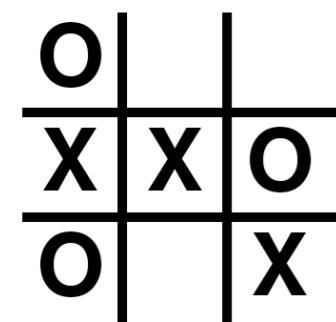
Limitation de la recherche minimax

Il est intéressant d'avoir une idée du nombre d'états possibles dans un jeu

Intérêt: cela va nous donner une idée de la difficulté du jeu

Observation générale: le nombre de coups disponibles varie souvent au cours d'une partie

Regardons quelques exemples de jeu...



Exemple: tic-tac-toe

$$b = 4$$

$$m = 9$$

Taille de l'arbre: $4^9 = 262144$

Possible pour minimax



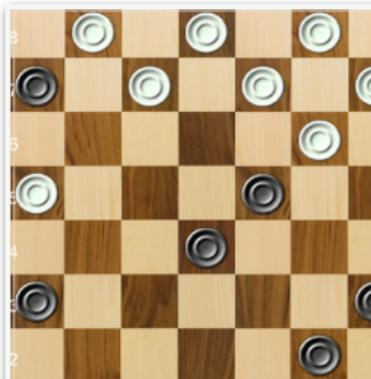
Exemple: jeu d'échecs

$$b = 35$$

$$m = 80$$

Taille de l'arbre: $35^{80} \approx 10^{123}$

Trop coûteux pour minimax



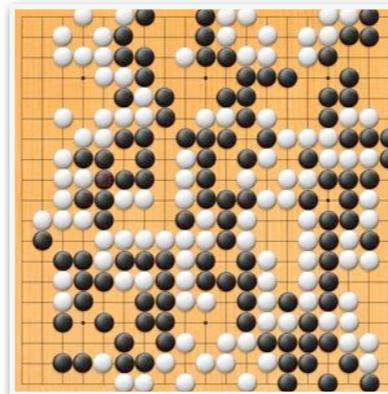
Exemple: jeu de dames

$$b = 2.8$$

$$m = 70$$

Taille de l'arbre: $2.8^{70} \approx 10^{31}$

Trop coûteux pour minimax



Exemple: jeu de Go

$$b = 250$$

$$m = 150$$

Taille de l'arbre: $250^{150} \approx 10^{359}$

Trop coûteux pour minimax

Cependant, on est arrivé à avoir des performances surhumaines pour ces jeu (champions du monde battus)

Pistes d'amélioration

On est dans une situation où notre Minimax n'est pas capable de résoudre des jeux non triviaux



Avez vous des idées d'amélioration pour notre recherche minimax ?

- (1) Eviter d'effectuer du calcul inutile (ne pas explorer certaines parties de l'arbre)
- (2) Ne réfléchir qu'un certain nombre de coups à l'avance (réduire la profondeur de la recherche)
- (3) Ne considérer que les actions les plus prometteuses (réduire le facteur de branchement)
- (4) Intégrer d'autres mécanismes à la recherche (mémoire, connaissance experte, apprentissage, etc.)

Ces améliorations peuvent amener à des agents de très bonne qualité, sur base d'un minimax

- (5) Utiliser une autre stratégie de recherche (*Monte-Carlo tree search - MCTS*)

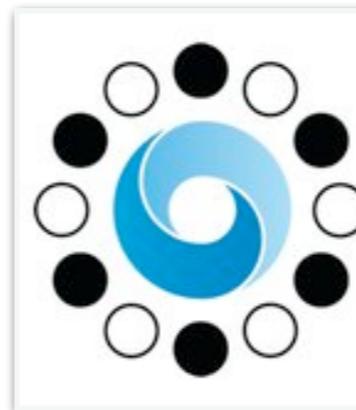


Stockfish

Un des meilleurs logiciels aux échecs

Recherche minimax à sa base

<https://stockfishchess.org/>



AlphaGo (et améliorations dérivées)

Meilleur logiciel au jeu de Go

MCTS comme composant essentiel

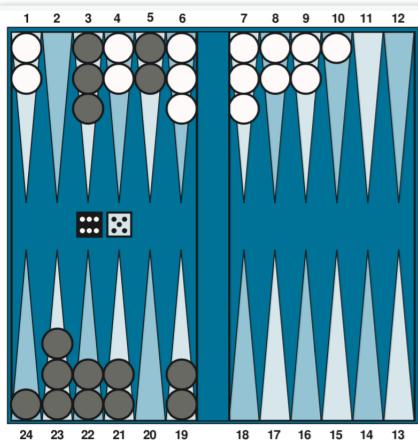
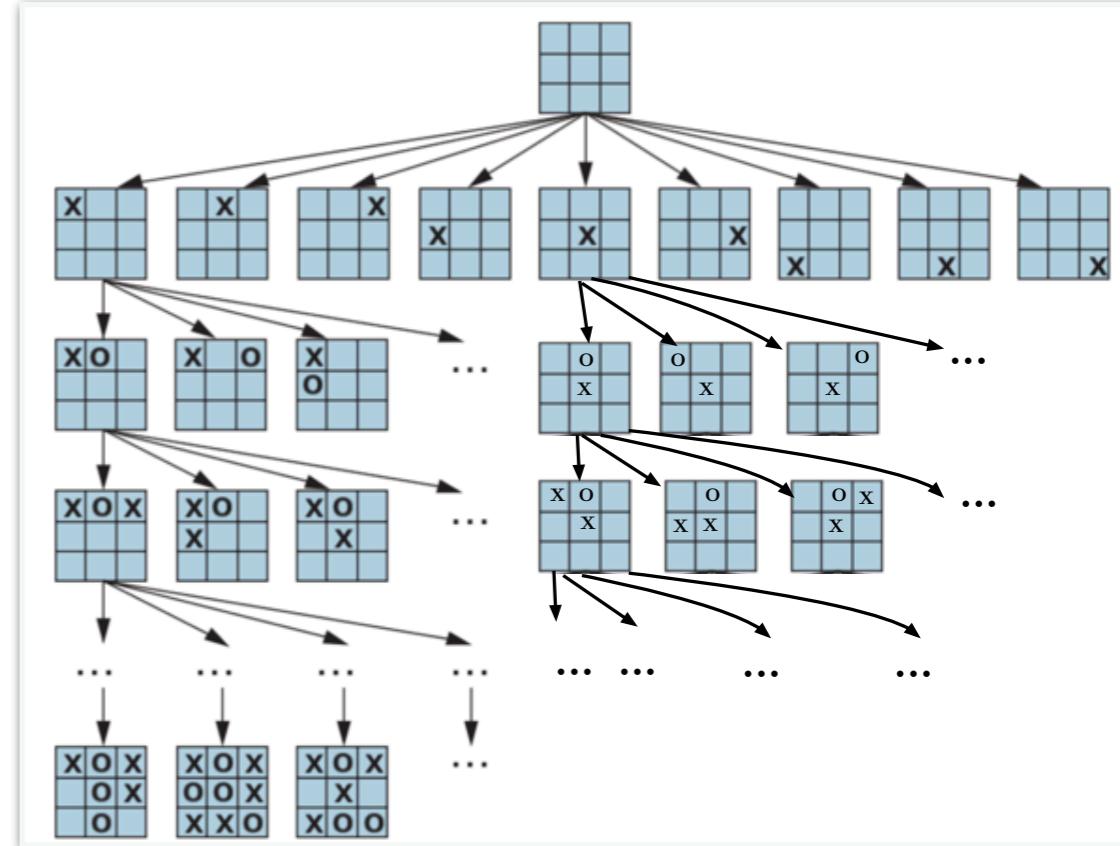
...avec de l'apprentissage profond

Une grande partie du reste de ce module est dédiée à explorer ces différentes pistes

Table des matières

Recherche adversarielle

- ✓ 1. Motivation de la recherche adversarielle
- ✓ 2. Catégorisation d'environnements compétitifs et des jeux
- ✓ 3. Stratégie de recherche minimax
- 4. Mécanisme du *alpha-beta pruning*
- 5. Conception de fonctions d'évaluation (heuristiques)
- 6. Mécanismes supplémentaires d'amélioration
- 7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
- 8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
- 9. Exemples et historique d'agents dédiés aux jeux



Source d'inefficacité dans la recherche Minimax

Ok! commençons par identifier les sources d'inefficacités dans le minimax

Principe minimax: la valeur de tous les états terminaux est propagée vers le noeud racine

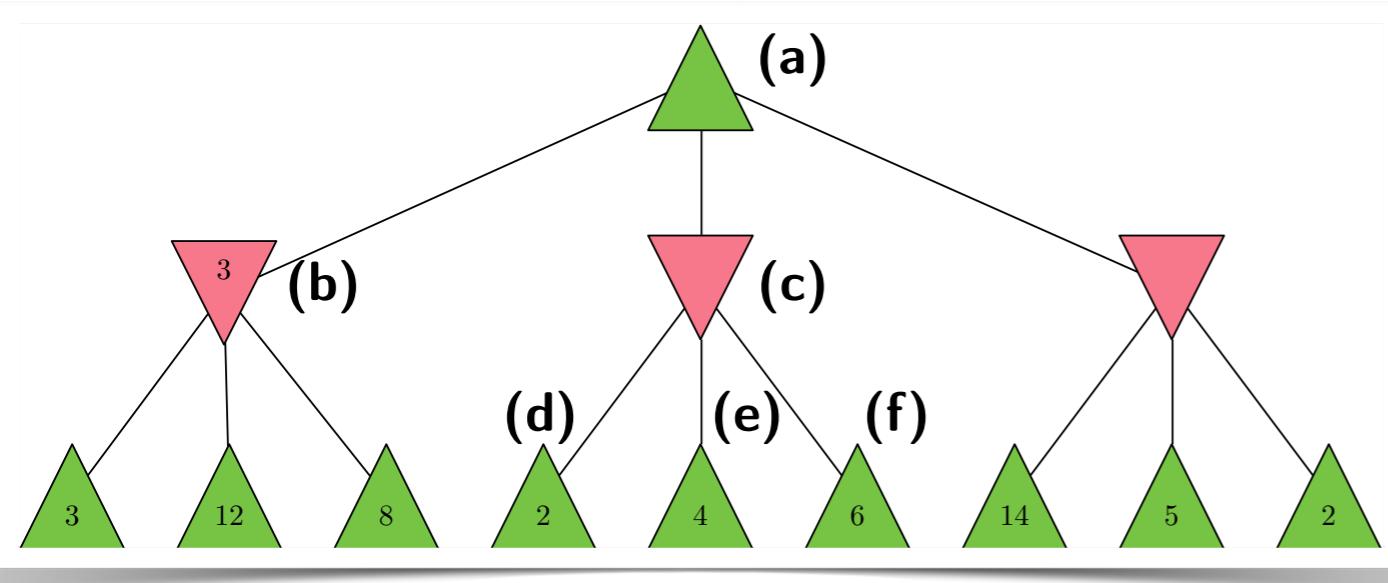
Résultat: le noeud racine contient ainsi le score pouvant être atteint et l'action en découlant

Observation: tous les noeuds de l'arbre sont explorés pour réaliser ce calcul

? Est-ce nécessaire de parcourir tout l'arbre pour calculer le score du noeud racine ?

Autrement dit: peut-on faire un résultat équivalent en explorant moins de noeuds ?

Exemple: considérons la situation où la valeur minimax de (b) a déjà été calculée (noeuds fils explorés)



Etape 1: le noeud (a) est MAX

Déduction: sa valeur sera AU MOINS 3

Etape 2: le noeud (c) est MIN

Observation: la valeur du noeud (d) est 2

Déduction: (c) sera AU PLUS 2

Etape 3: élagage de (e) et de (f) (opération de *pruning*)

Observation: quelque soit la valeur des autres fils de (c), ils n'auront aucun impact sur la valeur de (a)

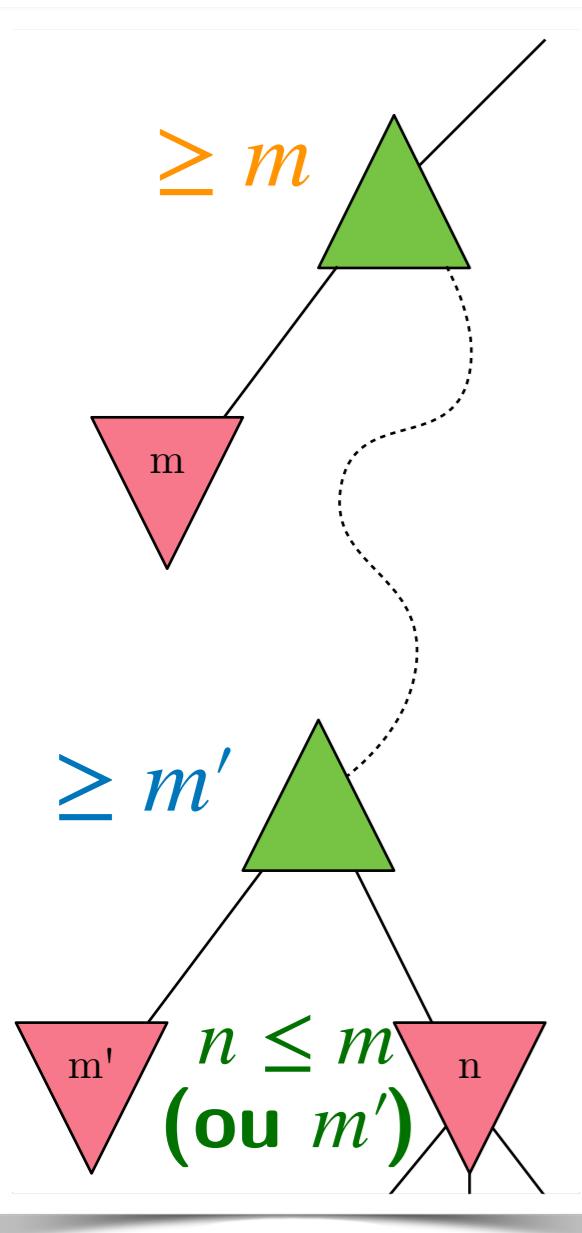
Déduction: on peut ainsi les élaguer (couper ces branches de l'arbre) et éviter du travail inutile

Alpha-beta pruning (élagage alpha-beta)



Alpha-beta pruning (élagage alpha-beta)

Amélioration de l'algorithme minimax visant à supprimer les branches de l'arbre qui n'ont aucun impact sur le calcul des valeurs minimax



Considérons un noeud n de l'arbre (MIN dans ce cas)

- (1) Soit m un autre noeud MIN plus haut que n dans l'arbre (déjà calculé)
- (2) Soit m' un autre noeud MIN au même niveau que n dans l'arbre (déjà calculé)
- (3) Le joueur MAX a une action potentielle de se déplacer au noeud n

? En quelles circonstances ne voudra t-il jamais se déplacer au noeud n ?
- (4) Lorsque se déplacer sur m ou m' est un meilleur choix
- (5) On peut obtenir cette information en explorant certains descendants de n
- (6) On explore les descendants jusqu'à être sur que n est moins bon que m ou m'
- (7) Lorsqu'on a cette certitude, on peut supprimer les branches de n non explorées

Ce raisonnement peut être fait pour les joueurs MAX et MIN symétriquement

En détectant ces situations, on pourra ainsi élaguer une partie des possibilités

Alpha-beta pruning (élagage alpha-beta)



Comment implémenter ce mécanisme efficacement ?

Idée: maintenir deux valeurs supplémentaires lors de la recherche (**bornes alpha-beta**)

α : valeur du meilleur choix trouvé actuellement pour MAX sur le chemin d'un noeud jusqu'à la racine

Interprétation : score minimum que le joueur MAX est assuré d'obtenir (borne inférieure sur le score)

β : valeur du meilleur choix trouvé actuellement pour MIN sur le chemin d'un noeud jusqu'à la racine

Interprétation : score maximum que le joueur MAX peut obtenir (borne supérieure sur le score)

Principe: ces bornes sont propagées de successeurs en successeurs lors de la recherche

Objectif: repérer quelles parties de l'arbre n'auront aucun impact sur les valeurs minimax



Comment utiliser ces bornes ?

Cas 1: règle d'élagage pour un noeud MIN

Objectif du noeud: est avoir la plus petite valeur possible

Elagage: on supprime les branches d'un noeud MIN si sa valeur actuelle est inférieure à **alpha**

Signification: ce noeud n'a aucune chance d'être meilleur qu'un autre déjà trouvé pour MAX

Cas 2: règle d'élagage pour un noeud MAX

Objectif du noeud: est avoir la plus grande valeur possible

Elagage: on supprime les branches d'un noeud MAX si sa valeur actuelle est supérieure à **beta**

Signification: ce noeud n'a aucune chance d'être meilleur qu'un autre déjà trouvé pour MIN

Alpha-beta pruning (élagage alpha-beta)

```
alphaBetaSearch( $s_0$ ) :       $\alpha$      $\beta$   
 $\langle v, m \rangle = \text{maxValue}(s_0, -\infty, +\infty)$   
return  $\langle v, m \rangle$ 
```

```
maxValue( $s, \alpha, \beta$ ) :  
if isTerminal( $s$ ) :  
    return  $\langle \text{utility}(s, \text{maxPlayer}), \perp \rangle$   
 $v^* = -\infty$   
 $m^* = \perp$   
for each  $a \in \text{actions}(s)$   
     $s' = \text{transition}(s, a)$   
     $\langle v, \_ \rangle = \text{minValue}(s', \alpha, \beta)$   
    if  $v > v^*$  :  
         $v^* = v$   
         $m^* = a$   
         $\alpha = \max(\alpha, v^*)$   
        if  $v^* \geq \beta$  : return  $\langle v^*, m^* \rangle$   
return  $\langle v^*, m^* \rangle$ 
```

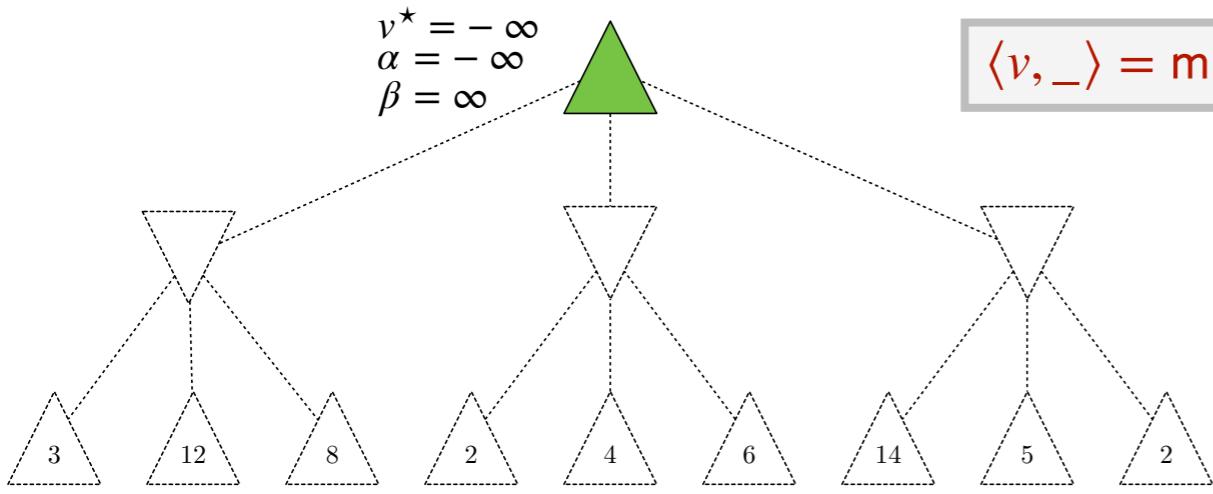
Initialisation des bornes à $+\infty$ (aucune information ne peut être déduite à ce stade)

```
minValue( $s, \alpha, \beta$ ) :  
if isTerminal( $s$ ) :  
    return  $\langle \text{utility}(s, \text{maxPlayer}), \perp \rangle$   
 $v^* = \infty$   
 $m^* = \perp$   
for each  $a \in \text{actions}(s)$   
     $s' = \text{transition}(s, a)$   
     $\langle v, \_ \rangle = \text{maxValue}(s', \alpha, \beta)$   
    if  $v < v^*$  :  
         $v^* = v$   
         $m^* = a$   
         $\beta = \min(\beta, v^*)$   
        if  $v^* \leq \alpha$  : return  $\langle v^*, m^* \rangle$   
return  $\langle v^*, m^* \rangle$ 
```

Raisonnement similaire pour MIN

On a ainsi un algorithme concret intégrant le mécanisme d'élagage alpha-beta !

Illustration de l'algorithme



$\langle v, _ \rangle = minValue(s', \alpha, \beta)$

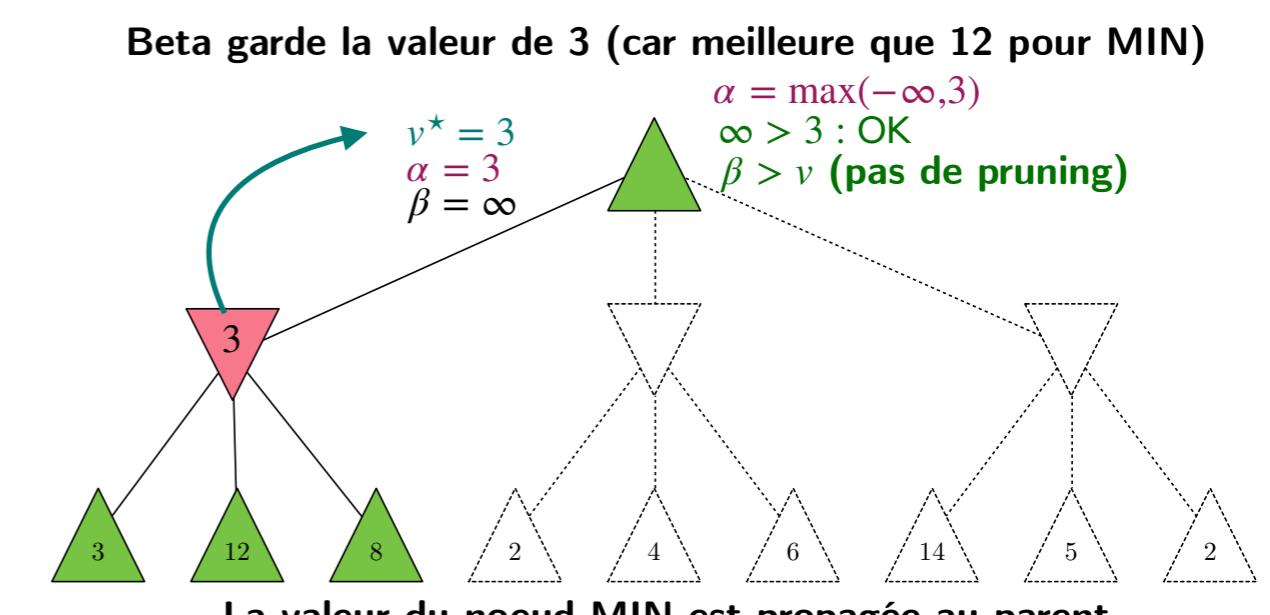
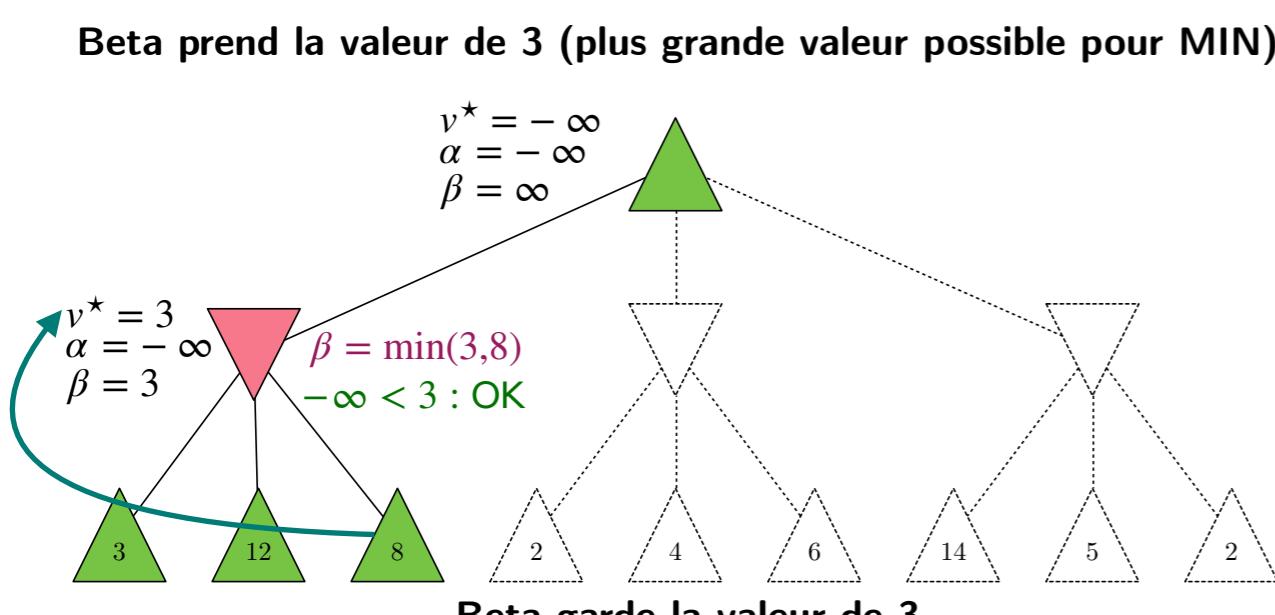
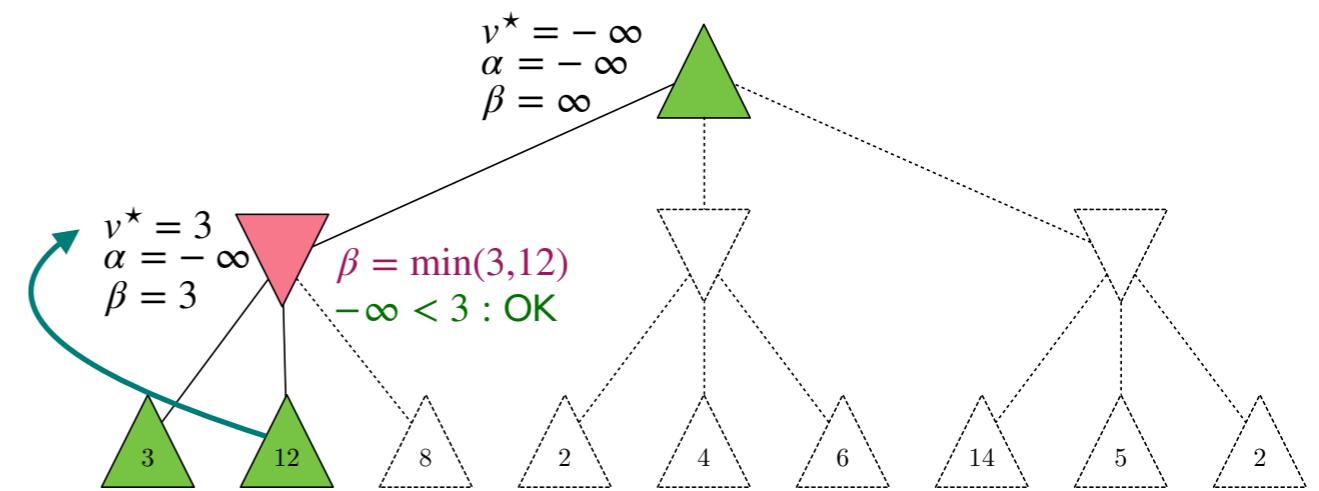
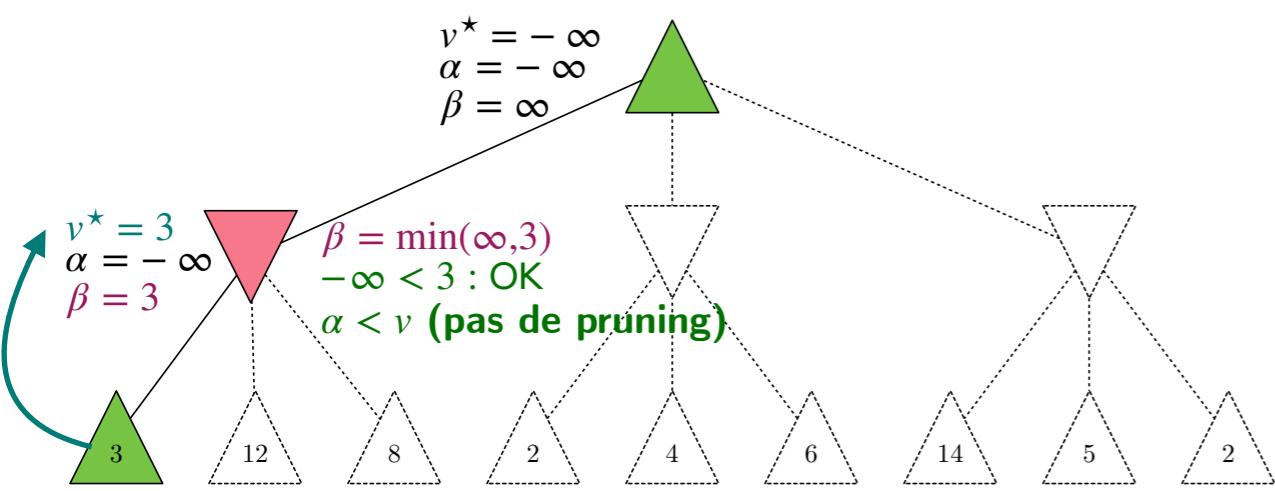
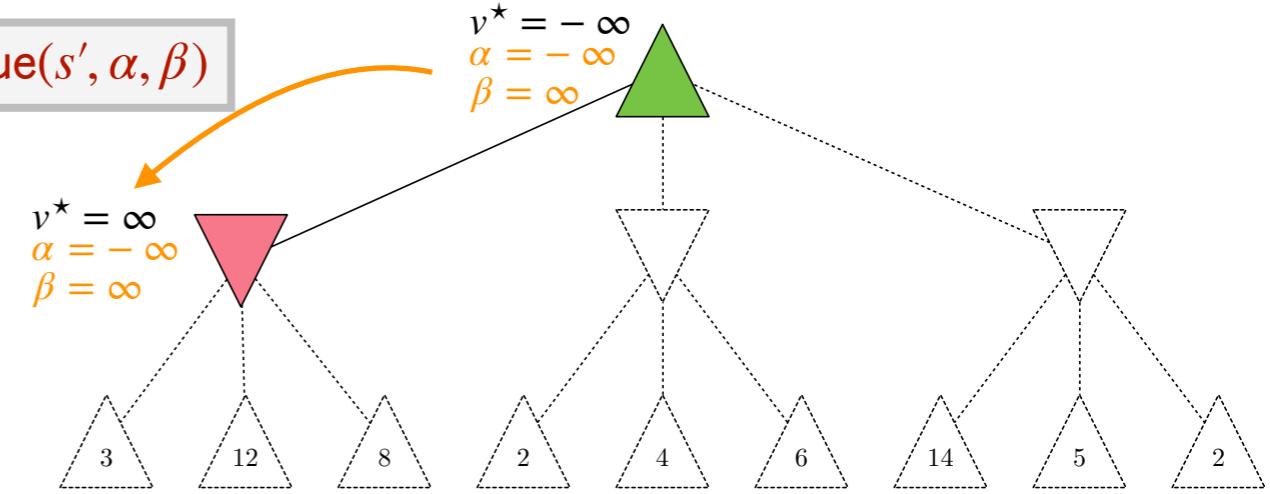
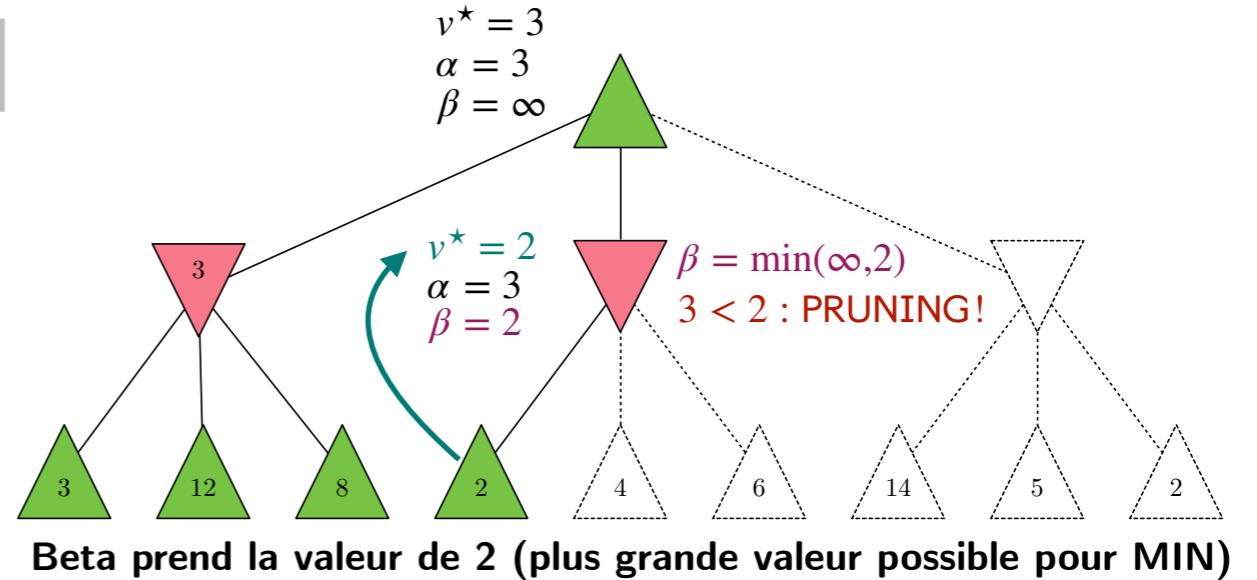
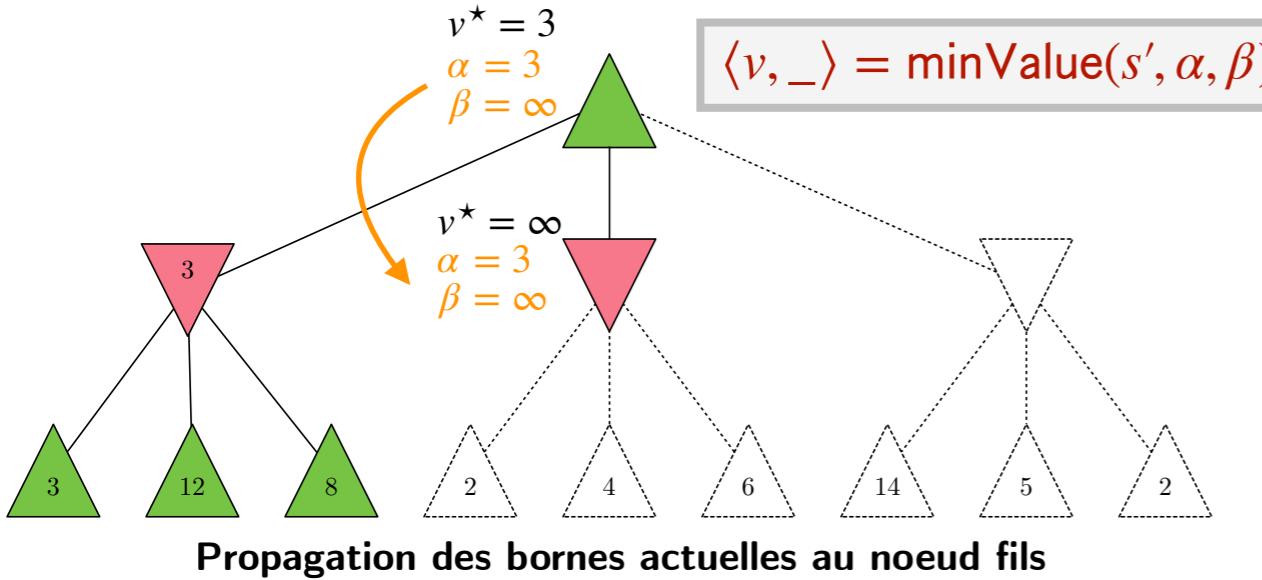


Illustration de l'algorithme



Observation: on arrive à un noeud où alpha est plus grand que beta

Raison 1: on a déjà un chemin où le joueur MAX pourra AU MOINS obtenir un score de 3 (valeur alpha)

Raison 2: a ce noeud, le joueur MAX pourra AU PLUS obtenir un score de 2 (valeur assurée pour MIN)

Elagage: Il n'y a aucun intérêt de continuer l'expansion de ce noeud

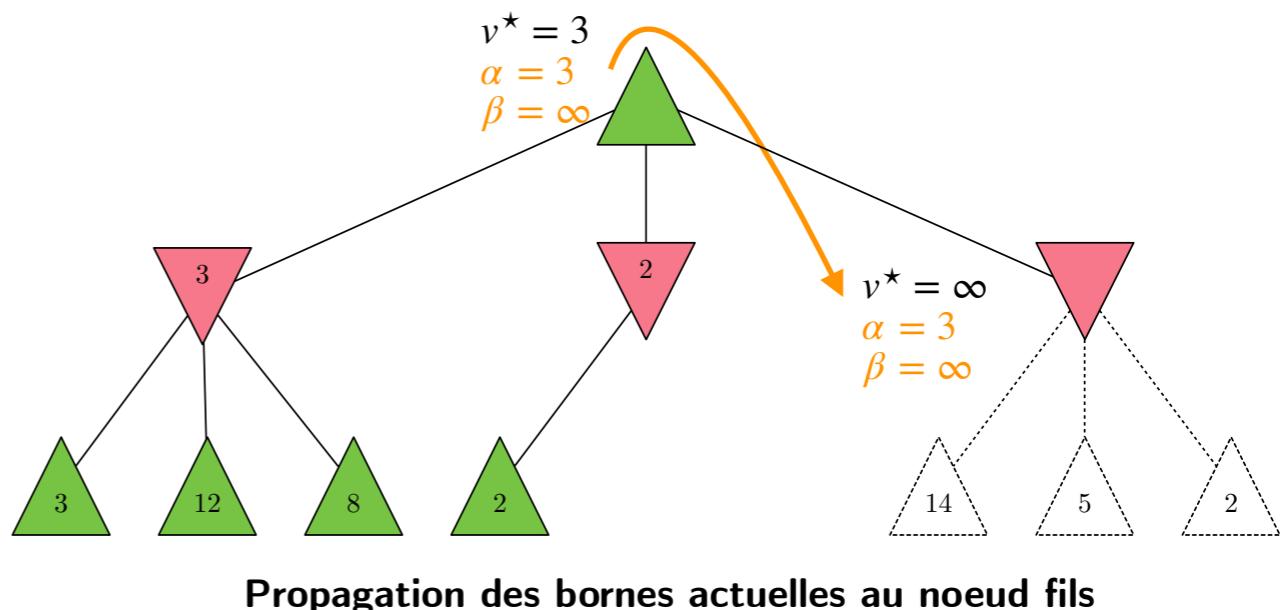
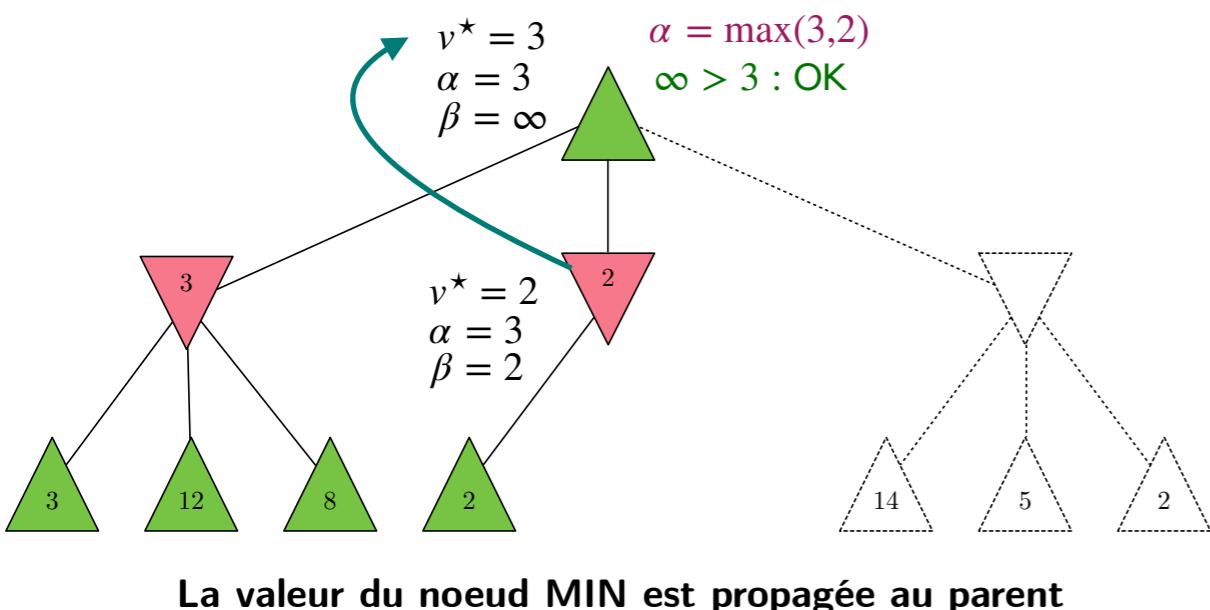
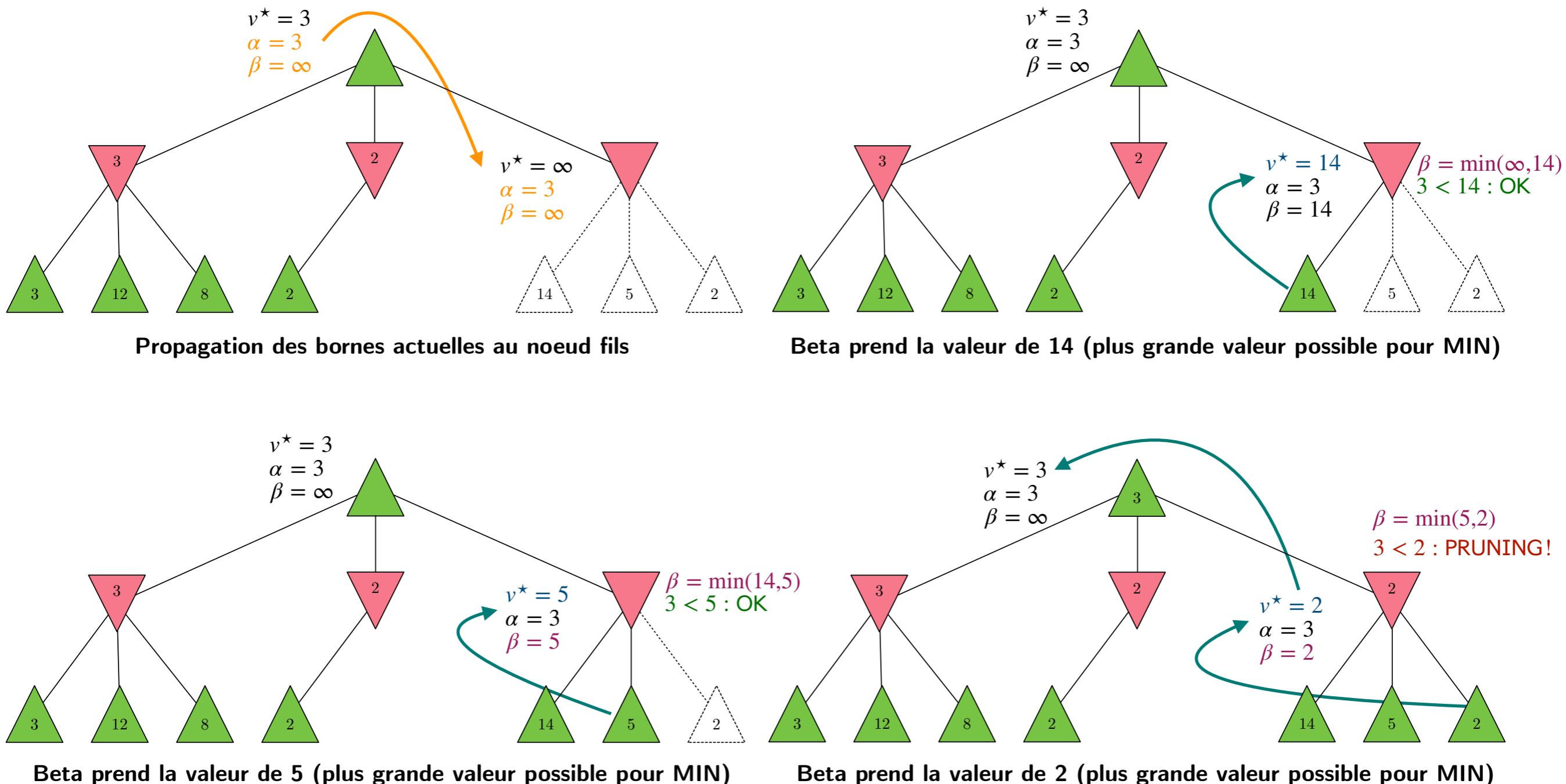
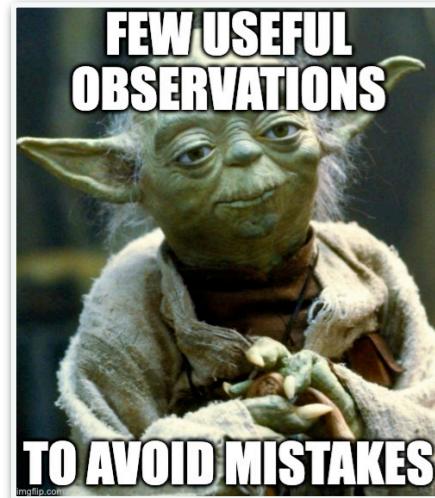


Illustration de l'algorithme



L'algorithme a fini son exécution, et on a évité l'exploration de deux noeuds

Propriétés de l'algorithme



Observation 1: les valeurs minimax (v) sont toujours propagées vers le noeud parent

Observation 2: les bornes alpha-beta sont toujours propagées vers les noeuds fils

Observation 3: les bornes beta sont mises à jour uniquement par les noeuds MIN

Observation 4: les bornes alpha sont mises à jour uniquement par les noeuds MAX

Signal d'élagage: dès qu'on est à un noeud ou alpha est plus grand que beta



Exactitude de l'élagage alpha-beta

La valeur minimax obtenue à la racine sera la même que celle obtenue par la recherche minimax

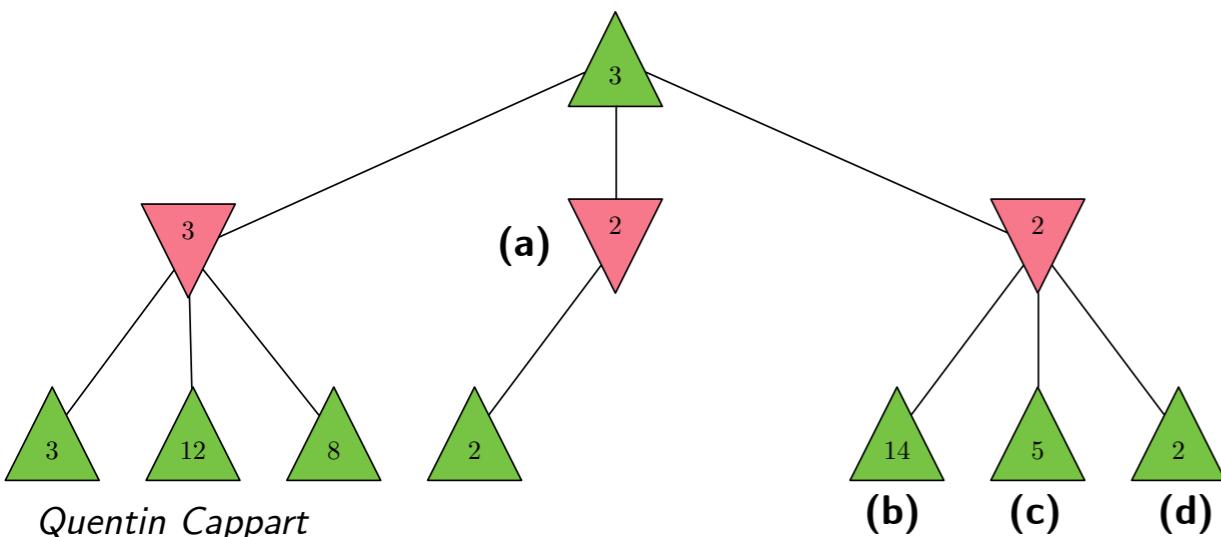
Autrement dit: cet algorithme n'effectue aucune approximation !

Bonne nouvelle: au niveau calculatoire, on ne rajoute aucune opération coûteuse

En pratique: des sous-arbres complets peuvent être prunés, et non seulement des noeuds feuilles



Est-il possible de pruner plus de branches dans notre exemple ?



Oui: en étendant le noeud (d) avant (b) et (c)

Résultat: on aurait directement obtenu le signal de pruning

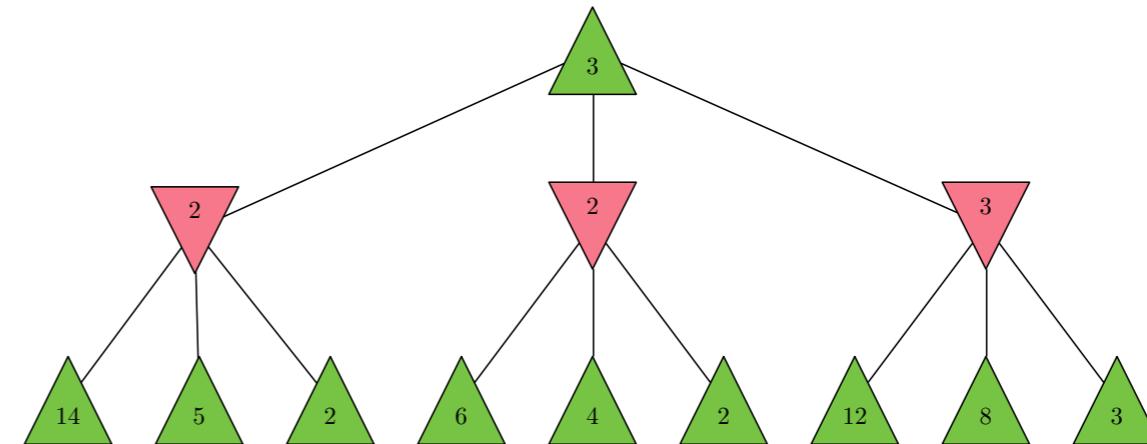
Elagage supplémentaire: les noeuds (c) et (b)

Ordre de considération des actions

L'efficacité du pruning alpha-beta dépend fortement de l'ordre dans lequel les actions sont étendues

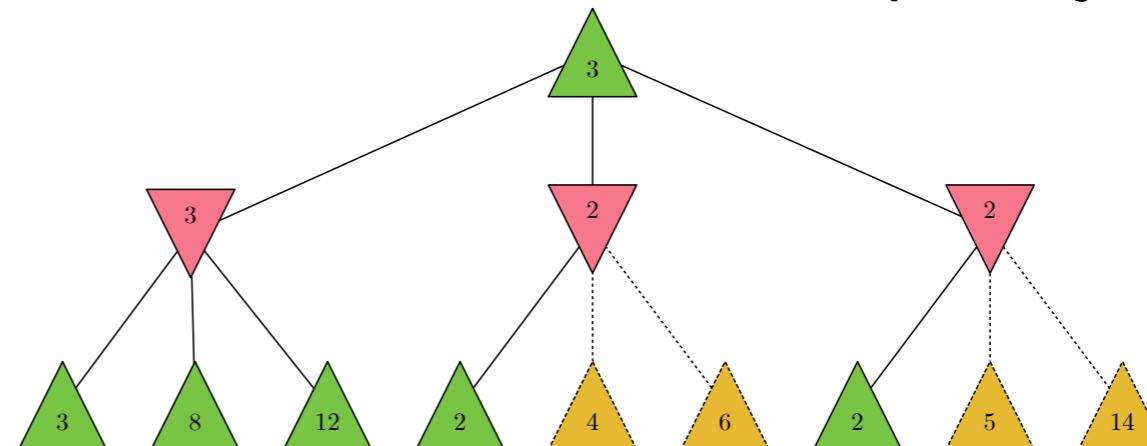
```
maxValue( $s, \alpha, \beta$ ) :  
    if isTerminal( $s$ ) :  
        return ⟨utility( $s$ , maxPlayer), ⊥⟩  
     $v^* = -\infty$   
     $m^* = \perp$   
    for each  $a \in \text{actions}(s)$  ←—————  
         $s' = \text{transition}(s, a)$   
        ⟨ $v, \_$ ⟩ = minValue( $s', \alpha, \beta$ )  
        if  $v > v^*$  :  
             $v^* = v$   
             $m^* = a$   
             $\alpha = \max(\alpha, v^*)$   
        if  $v^* \geq \beta$  : return ⟨ $v^*, m^*$ ⟩  
    return ⟨ $v^*, m^*$ ⟩
```

Idée 1: étendre d'abord les pires actions possibles pour le joueur actuel



Ordre peu efficace: aucun pruning possible

Idée 2: étendre d'abord les meilleures actions pour le joueur actuel



Ordre efficace: 4 noeuds prunés (environ 30% des noeuds de l'arbre)

Règle d'or pour la sélection

Aux noeuds MAX: étendre les actions ayant le plus haut score

Aux noeuds MIN: étendre les actions ayant le plus petit score



A t-on accès à ces valeurs ?

Sélection d'un ordre des actions

Malheureusement, on n'a pas accès à ces valeurs !

Explication: cela revient à obtenir la valeur minimax (et donc résoudre le problème)

Compromis possible: on peut tenter **d'estimer** à quel point une action est bonne pour un joueur

Intuitivement: c'est ce qu'on fait lorsqu'on joue en considérant d'abord ce qui nous semble le plus favorable



Exemple pour les échecs: considérer d'abord les captures, puis les menaces, puis les déplacements

Utilisation d'une heuristique: estimer les états intermédiaires et les trier en fonction d'une heuristique

? Quel impact l'élagage a t-il sur la complexité temporelle de la recherche minimax simple ?

Nouveau facteur de branchement: $b \rightarrow \sqrt{b}$ (**preuve en lecture complémentaire**)

Recherche minimax simple: $\mathcal{O}(b^m)$

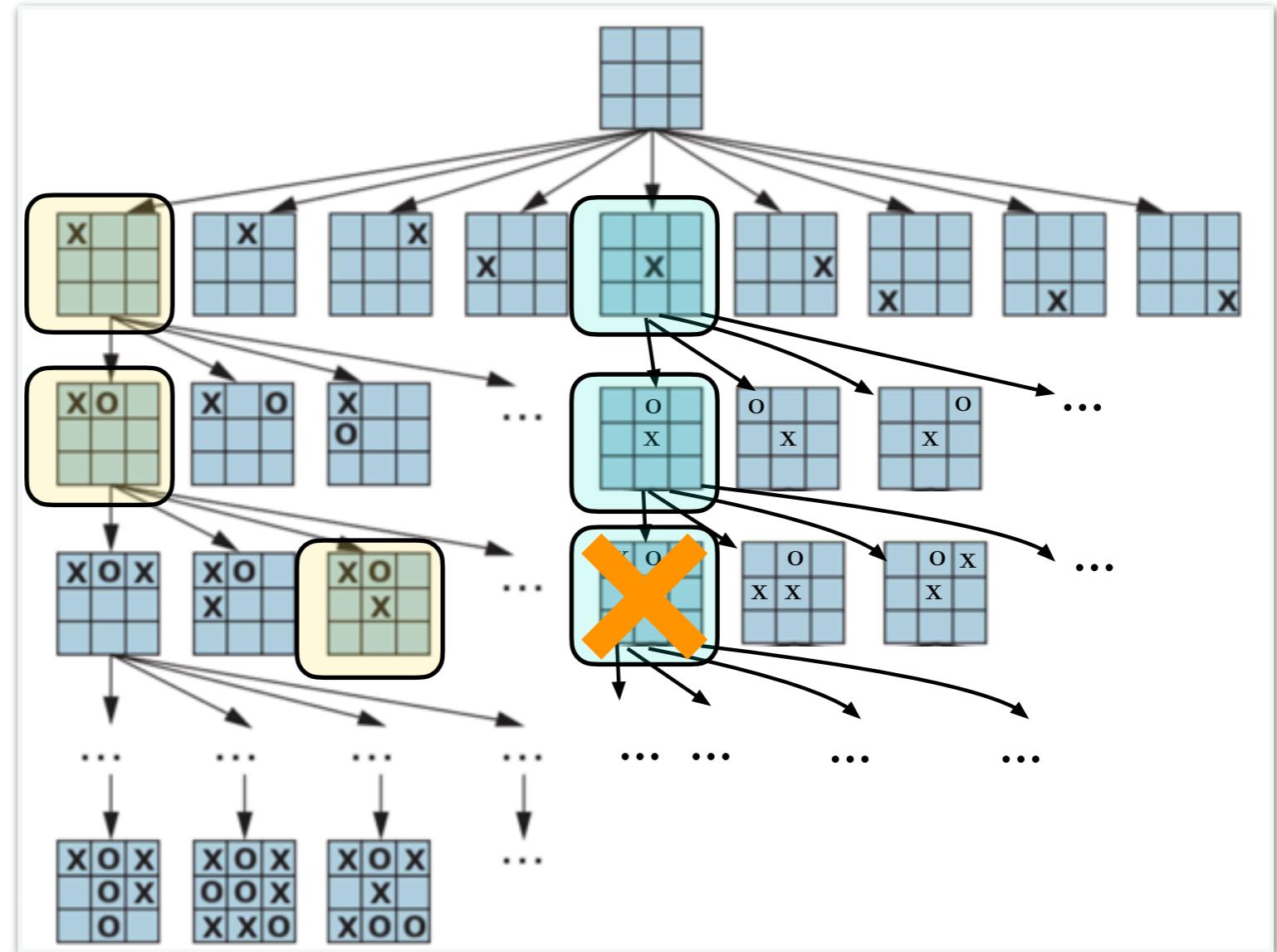
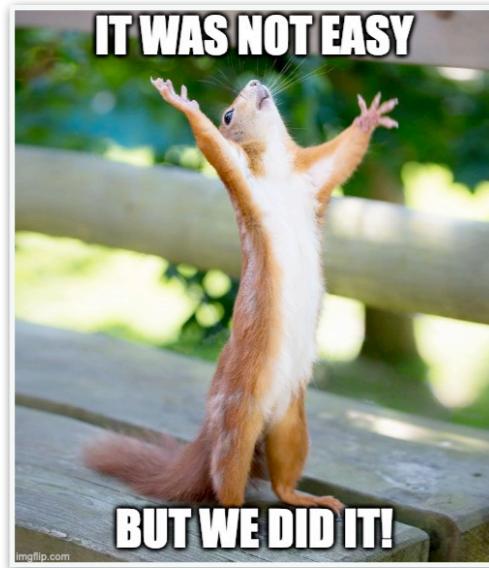
Alpha-beta pruning avec un mauvais ordre: $\mathcal{O}(b^m)$ (**aucune amélioration**)

Alpha-beta pruning avec un ordre optimal: $\mathcal{O}(\sqrt{b}^m) = \mathcal{O}(b^{m/2})$

Alpha-beta pruning avec un ordre aléatoire: $\approx \mathcal{O}(b^{3m/4})$

Bonne nouvelle: le pruning peut rendre possible une exploration deux fois plus profonde !

Chemins redondants et table de transposition



? Voyez-vous du travail inutile dans cet arbre de recherche ?

Observation: un même état est présent plusieurs fois dans l'arbre de recherche

Source n'inefficacité: les sous-arbres d'états identiques seront aussi identiques

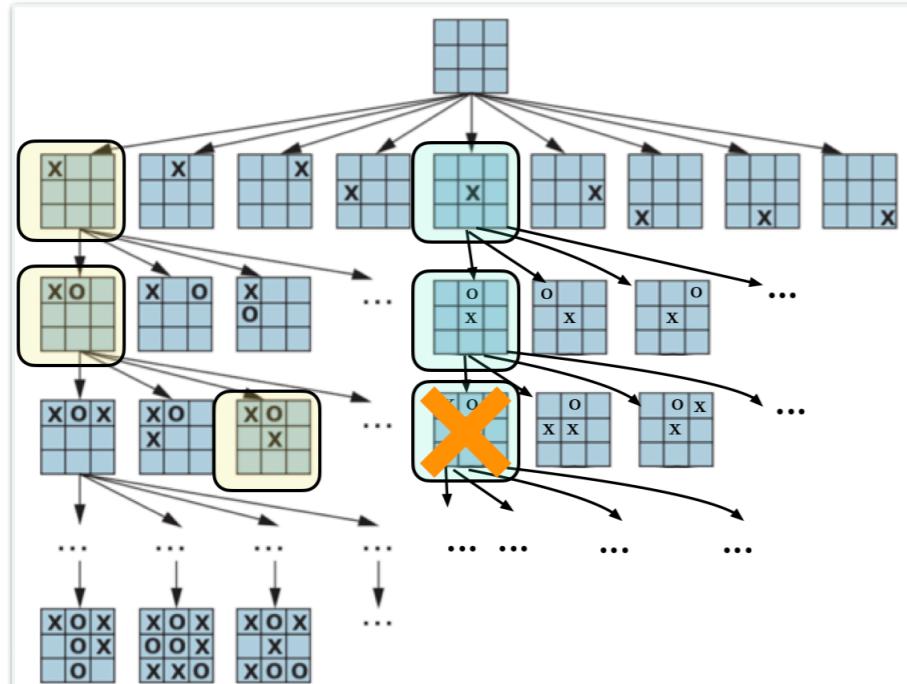
Mauvaise nouvelle: ne pas tenir compte de ce phénomène engendre de grosses pertes de performances

Chemins redondants et table de transposition



Chemin redondant

Séquence d'actions différentes menant à un même état dans l'arbre de recherche



BUT WHAT CAN WE DO?



Table de transposition

Mémoire qui va retenir les états déjà visités lors de la recherche

Principe: avant d'étendre un état, on va vérifier qu'il ne se trouve pas déjà dans la mémoire

Même philosophie que la recherche en graphe retenant les états déjà explorés (module précédent)

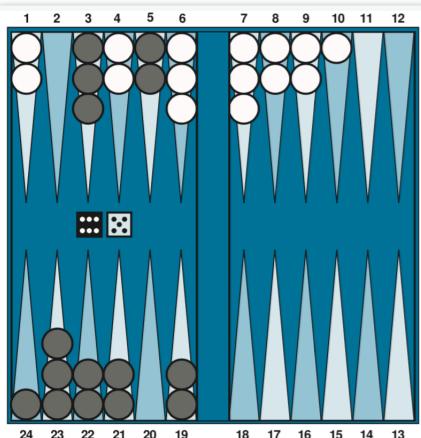
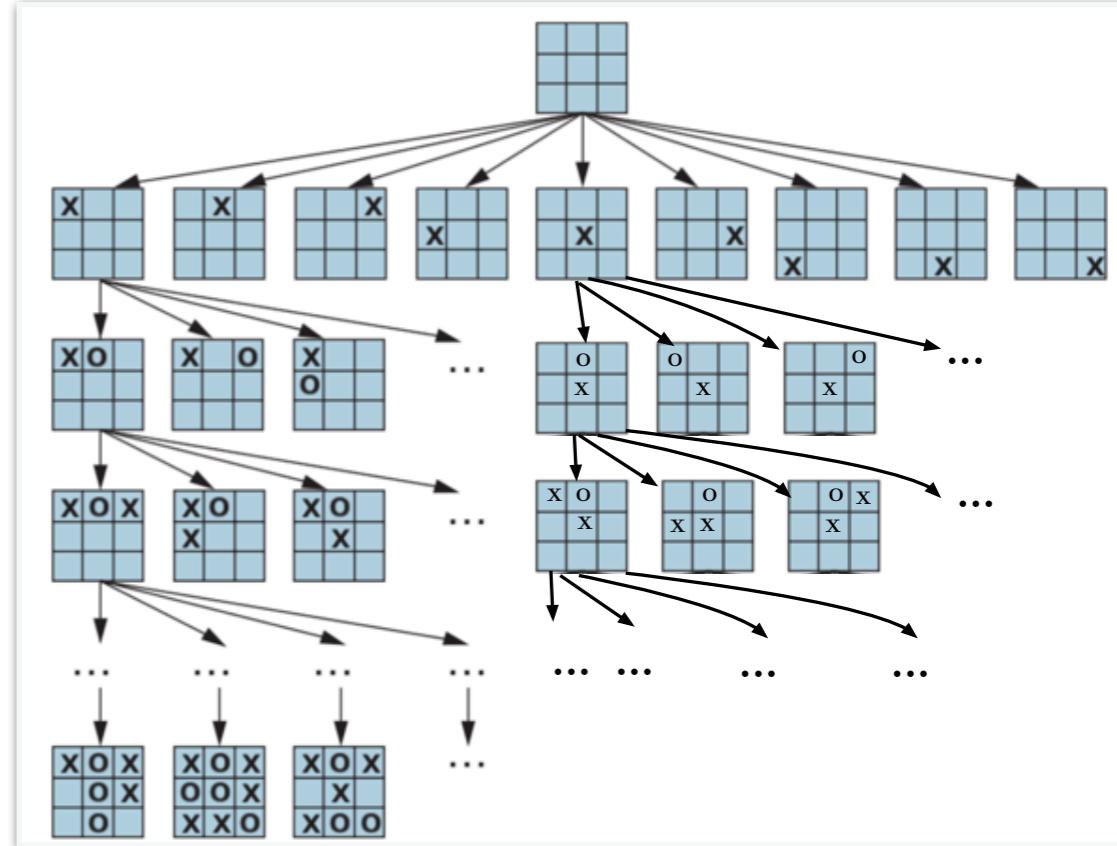
Ce mécanisme est très puissant ! (permet une exploration deux fois plus profondes aux échecs)

Implémentation efficace: avec une fonction de hachage (p.e., *hachage de Zobrist* - lecture complémentaire)

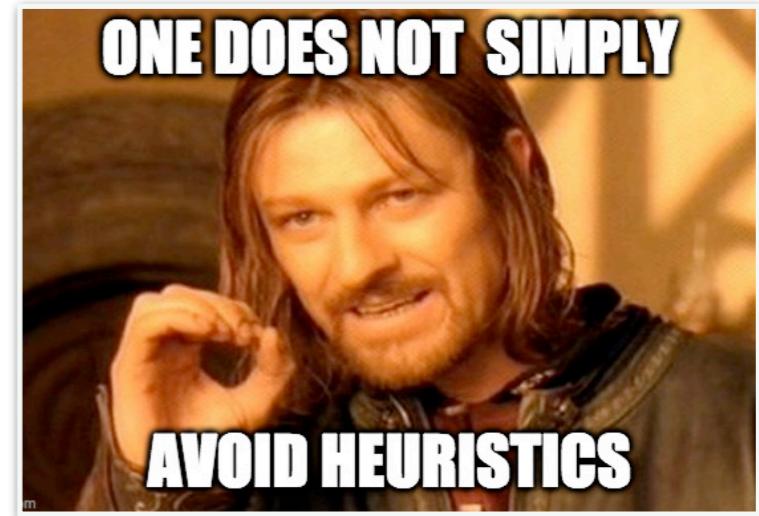
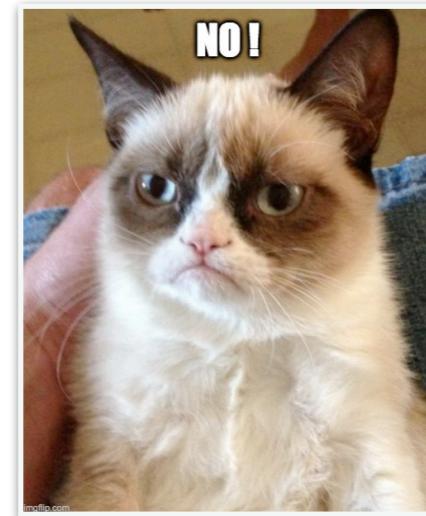
Table des matières

Recherche adversarielle

- ✓ 1. Motivation de la recherche adversarielle
- ✓ 2. Catégorisation d'environnements compétitifs et des jeux
- ✓ 3. Stratégie de recherche minimax
- ✓ 4. Mécanisme du *alpha-beta pruning*
- 5. Conception de fonctions d'évaluation (heuristiques)
- 6. Mécanismes supplémentaires d'amélioration
- 7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
- 8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
- 9. Exemples et historique d'agents dédiés aux jeux



Stratégies pour améliorer la recherche



Mauvaise nouvelle: on est encore très loin de pouvoir être efficace à des jeux complexes (échecs, Go, etc.)

Que peut-on faire ?

Autre mauvaise nouvelle: on doit abandonner l'idée de calculer une stratégie optimale

Plan de secours: utiliser des heuristiques pour estimer la qualité des états intermédiaires

Objectif: réduire la taille de l'espace de recherche

Inconvénient: contrairement à l'alpha-beta pruning, on effectue des approximations

Inconvénient: on est généralement très sensible à la qualité des heuristiques choisies

Plusieurs stratégies sont possibles selon la façon dont les heuristiques sont intégrées à la recherche

Stratégies de type A

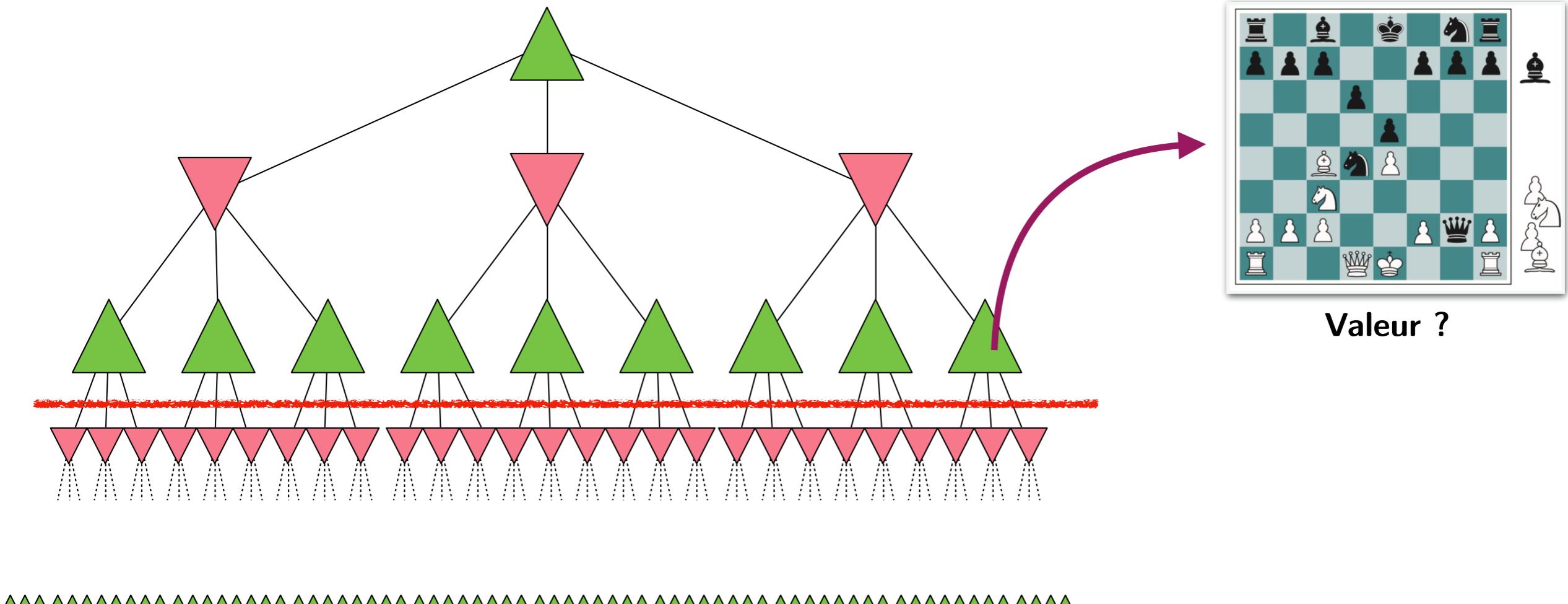


Stratégie de type A

Stratégie visant à effectuer une recherche jusqu'à une certaine profondeur de l'arbre

Principe: fixer une profondeur limite pour une exploration complète

Implication pratique: l'exploration de l'arbre est large mais peu profonde



Comment obtenir la valeur minimax des états de la profondeur non finale ?

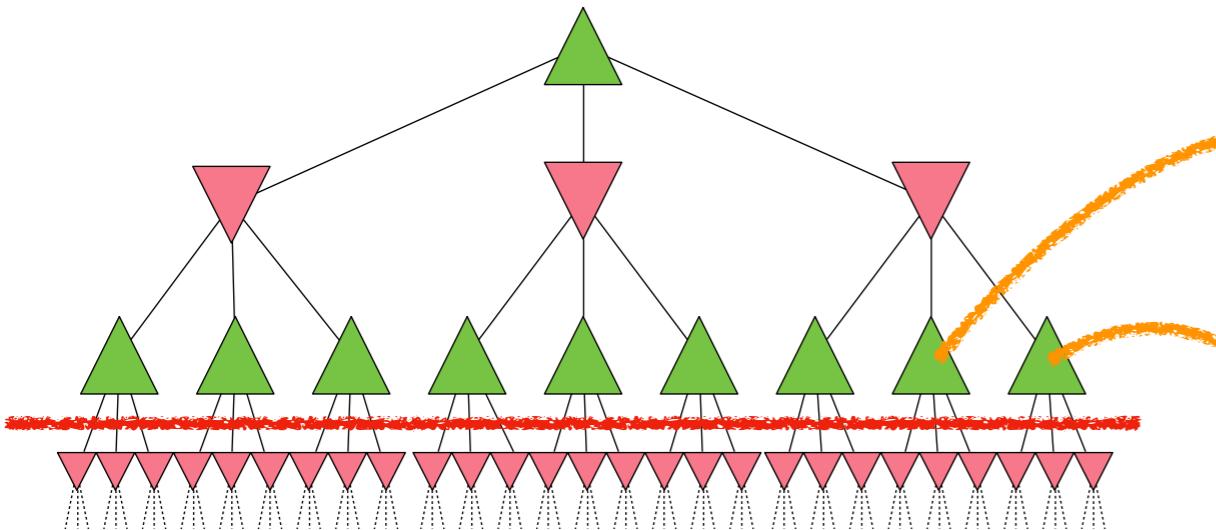
Heuristique Minimax



Heuristique minimax

Principe identique à la valeur minimax, sauf qu'on utilise une fonction heuristique pour évaluer la qualité des états de la profondeur limite

$$\text{heuristicMinimax}(s, d) = \begin{cases} \text{heuristicEvaluation}(s, \text{maxPlayer}) & \text{if } \text{isCutoffDepth}(d) \\ \max_{a \in \text{actions}(s)} \text{heuristicMinimax}(\text{transition}(s, a), d + 1) & \text{if } \text{turn}(s) = \text{maxPlayer} \\ \min_{a \in \text{actions}(s)} \text{heuristicMinimax}(\text{transition}(s, a), d + 1) & \text{if } \text{turn}(s) = \text{minPlayer} \end{cases}$$



HeuristicEvaluation(



HeuristicEvaluation(



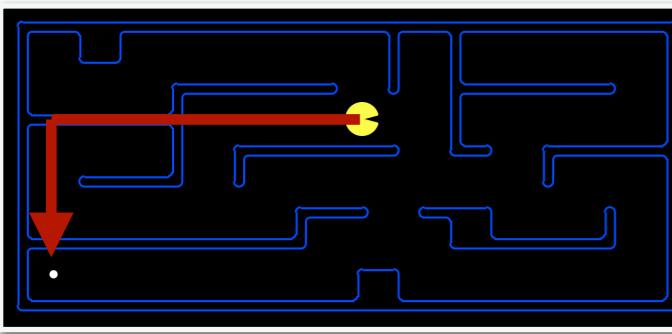
Principe 1: les états de la profondeur limite sont estimés suivant notre heuristique d'évaluation

Principe 2: les valeurs sont propagées jusqu'à la racine de la même façon que par la recherche minimax

Implication pratique: on évite de devoir aller à la fin de l'arbre pour pouvoir calculer l'action à effectuer

En pratique: combinaison possible et souhaitée avec de l'alpha-beta pruning

Fonction heuristique des les jeux



Nature de l'heuristique: la signification de sa valeur

Module précédent: estimation de la distance entre un état et l'état final

Module actuel: estimation du score (*utility*) qui pourra être obtenu via cet état

Pour les états terminaux: $\text{heuristicEvaluation}(s, p) = \text{utility}(s, p)$

Pour les états non terminaux: $\min_{s \in S} \text{utility}(s, p) \leq \text{heuristicEvaluation}(s, p) \leq \max_{s \in S} \text{utility}(s, p)$

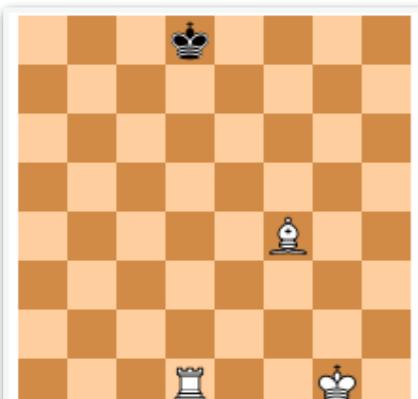
Exemple: jeu d'échecs

Valeurs finales: score de -1 pour une défaite, +1 pour une victoire, 0 pour une égalité (point de vue blanc)



Estimation de l'heuristique: 0.3

Interprétation: état mitigé, avec une estimation d'un léger avantage pour les blancs



Estimation de l'heuristique: 0.95

Interprétation: état montrant un clair avantage pour les blancs

Intuition: on voudra se retrouver dans un état avec une valeur élevée pour nous

Conception d'une heuristique

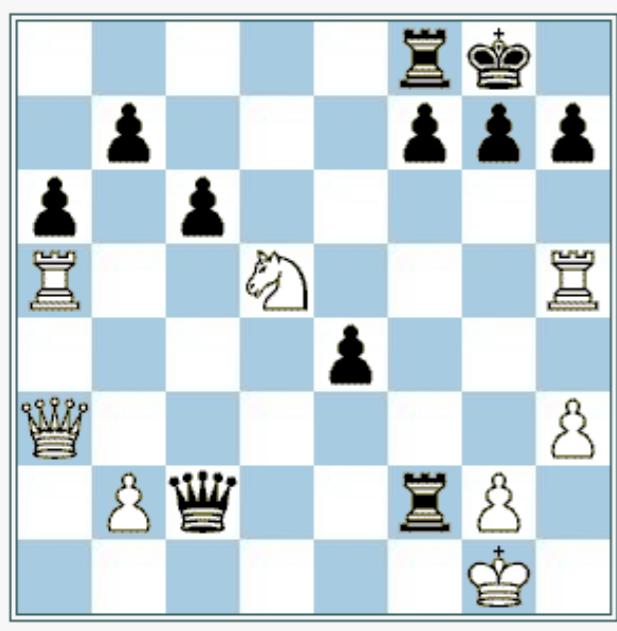
? Quelles sont les caractéristiques essentielles d'une bonne heuristique ?

Précision: fortement corrélée avec le score final atteignable depuis l'état

Efficacité: pas trop coûteuse à calculer (l'objectif est de rendre la recherche plus rapide)

La construction d'une heuristique revient souvent à un processus en deux étapes

- (1) **Caractérisation:** consiste à transformer un état en une série de caractéristiques (*features*)
- (2) **Evaluation:** consiste à évaluer la qualité d'un état, sur base des caractéristiques identifiées



Exemple: jeu d'échecs

- (1) **Caractérisation sur base du matériel**

Joueur blanc: $n_{\text{pions}} = 3, n_{\text{cavaliers}} = 1, \dots$

Joueur noir: $n_{\text{pions}} = 6, n_{\text{cavaliers}} = 0, \dots$

- (2) **Evaluation pour une combinaison linéaire**

$$h(s) = w_1 x_1(s) + w_2 x_2(s) + \dots + w_m x_m(s)$$

Valeur des pièces: $w_{\text{pions}} = 1, w_{\text{cavaliers}} = 3, \dots$

$$h_{\text{materiel}}(s, p) = n_{\text{pions}}(s, p) + 3 \times n_{\text{cavaliers}}(s, p) + 3 \times n_{\text{fous}}(s, p) + 5 \times n_{\text{tours}}(s, p) + \dots$$

$$h_{\text{blanc}}(s) = h_{\text{materiel}}(s, \text{blanc}) - h_{\text{materiel}}(s, \text{noir})$$

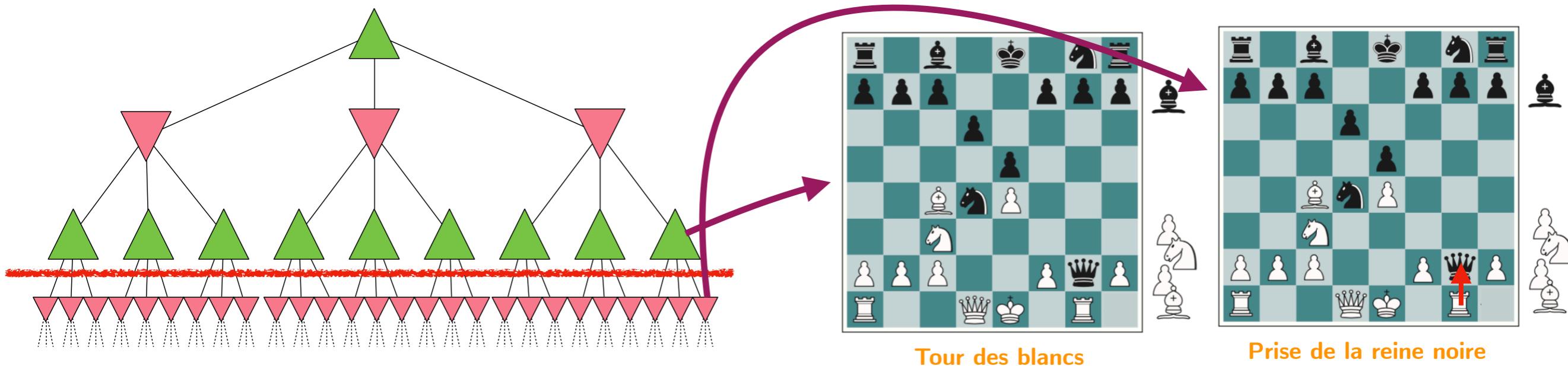
Calcul des poids: soit par connaissance experte, ou apprentissage automatique

Dangers des stratégies de type A



Quels sont les dangers/difficultés d'une stratégie de type A ?

- (1) **Forte dépendance à l'heuristique:** haut risque d'une mauvaise action si l'heuristique est mauvaise
- (2) **Effet de myopie:** un état paraît bon à la profondeur limite, mais ne l'est pas réellement



Exemple: considérons une heuristique basée sur le matériel de chaque joueur

Situation à un noeud limite: avantage pour les noirs (meilleur matériel)



Qu'en pensez vous ?

Difficulté: la reine noire est non protégée et peut être prise gratuitement par la tour du joueur blanc

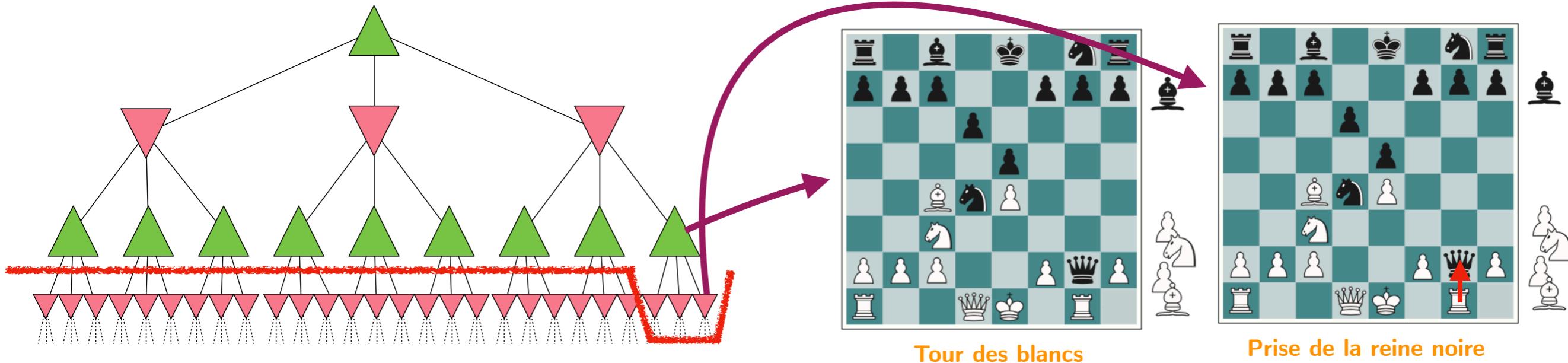
En réalité, l'état est favorable pour les blancs (non visible à cause de la profondeur limite)



Avez-vous une idée pour mitiger cette difficulté ?

Dangers des stratégies de type A

 **Etat sécuritaire (*quiescent state*)**
Position sécuritaire pour couper la recherche (sans action importante en attente)



Amélioration possible: autoriser de couper la recherche que sur les *quiescent states*

$$\text{heuristicMinimax}(s, d) = \begin{cases} \text{heuristicEvaluation}(s, \text{maxPlayer}) & \text{if } \text{OnCutoffOrGreater}(d) \wedge \text{isQuiescent}(s) \\ \max_{a \in \text{actions}(s)} \text{heuristicMinimax}(\text{transition}(s, a), d + 1) & \text{if } \text{turn}(s) = \text{maxPlayer} \\ \min_{a \in \text{actions}(s)} \text{heuristicMinimax}(\text{transition}(s, a), d + 1) & \text{if } \text{turn}(s) = \text{minPlayer} \end{cases}$$

Exemple aux échecs: ne pas couper la recherche lorsqu'une capture importante est en cours

Difficulté: pas toujours aisément de repérer un *quiescent state*

Autre amélioration: arrêter d'étendre un noeud selon certains critères (p.ex., défaite ou victoire assurée)

Stratégies de type B



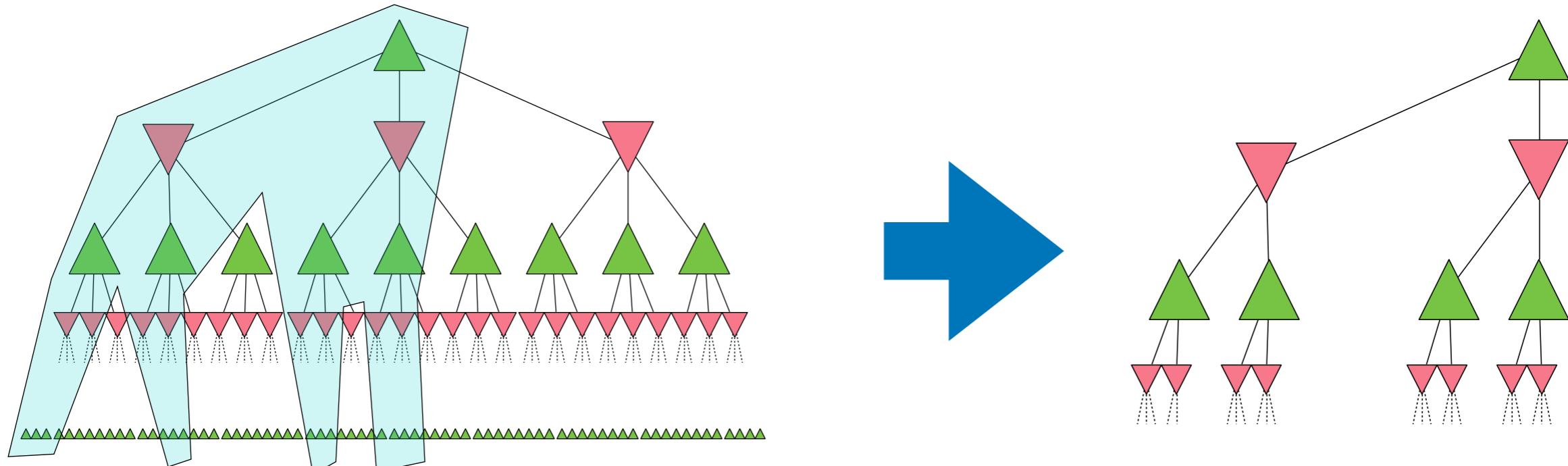
Stratégie de type B

Stratégie visant à étendre en priorité que les actions qui sont les plus prometteuses

Principe: ne pas considérer les actions qui sont mauvaises suite à notre évaluation heuristique

Implication pratique: l'exploration de l'arbre est étroite mais profonde

Idéalement, on souhaite explorer jusqu'en bas de l'arbre, mais pour un exemple restreint de scénarios



Similaire à la façon humaine de raisonner (de par notre faculté limitée à considérer trop de possibilités)

Réalisation simple de cette stratégie: n'étendre que les n meilleures mouvements suivant l'heuristique

Danger: il est possible que la meilleure action soit supprimée par ce mécanisme

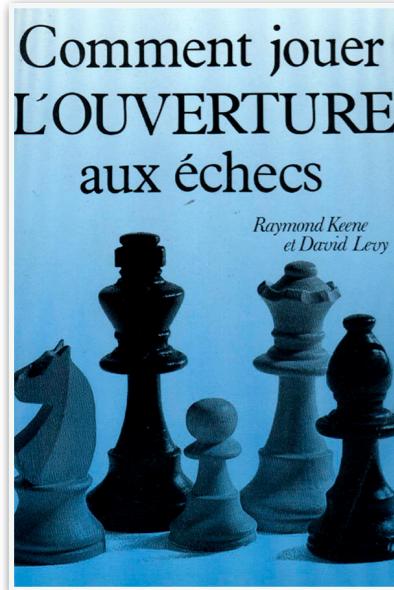
Amélioration: intégrer une profondeur variable (profond pour les bonnes actions, progressivement plus faible)

Débuts et fins de partie



Quelles sont les particularités d'un début et d'une fin de partie dans un jeu ?

Début de partie



Particularité 1: l'arbre de recherche est immense (exploration complète impensable)

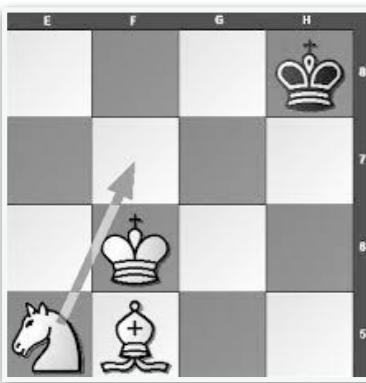
Particularité 2: les conséquences à long terme des actions sont difficiles à prévoir

Conséquence: on finit souvent par réaliser les même actions en début de partie

Idée: suivre une **table d'ouverture** pour les premiers mouvements, au lieu d'explorer l'arbre

Construction de la table: souvent sur base de connaissances expertes ou de statistiques

Fin de partie



Particularité 1: l'exploration complète de l'arbre est possible mais reste coûteuse

Particularité 2: beaucoup de fin de partie peuvent se ressembler (p.e., KBNK)

Particularité 3: les conséquences d'une mauvaise action peuvent être irrémédiables

Idée: suivre une **table de fin de partie** pour les derniers mouvements, au lieu d'explorer

Construction de la table: par l'exécution antérieure d'une stratégie de recherche (éventuellement coûteuse)

Dans les deux cas, on économise du temps d'exécution par du pré-c calcul et de la consommation mémoire

Cas d'étude: Stockfish



Stockfish: logiciel entièrement dédié à jouer aux échecs

Création: en 2004 (développement toujours actif)

Caractéristiques: open-source et code disponible sur Github

Utilisation pratique: outil d'analyse sur les plateformes de jeu en ligne (*Lichess*)

<https://stockfishchess.org/>

Stratégie de recherche

Combinaison des idées précédentes: minimax, alpha-beta, et tables de transposition

Ingrédient principal: heuristique excellente mais très complexe

Mécanisme additionnel: gestion spécifique des fins de partie

Performances

Minimax avec alpha-beta pruning: profondeur de 8 actions

Stockfish: profondeur de plus de 30 actions

Performances largement supérieures aux humains

Souvent la première ou deuxième place aux concours d'IA

```
// Main evaluation begins here
initialize<WHITE>();
initialize<BLACK>();

// Pieces evaluated first (also populates attackedBy, attackedBy2).
// Note that the order of evaluation of the terms is left unspecified.
score += pieces<WHITE, KNIGHT>() - pieces<BLACK, KNIGHT>()
    + pieces<WHITE, BISHOP>() - pieces<BLACK, BISHOP>()
    + pieces<WHITE, ROOK >() - pieces<BLACK, ROOK >()
    + pieces<WHITE, QUEEN >() - pieces<BLACK, QUEEN >();

score += mobility[WHITE] - mobility[BLACK];

// More complex interactions that require fully populated attack bitboards
score += king< WHITE>() - king< BLACK>()
    + passed< WHITE>() - passed< BLACK>();

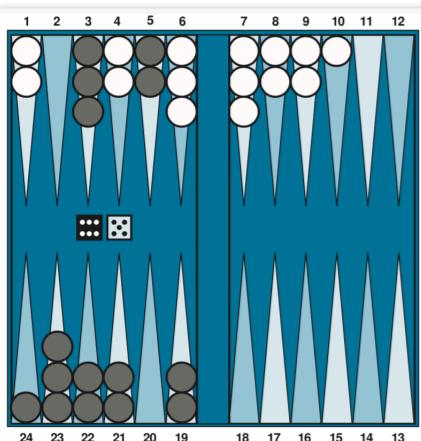
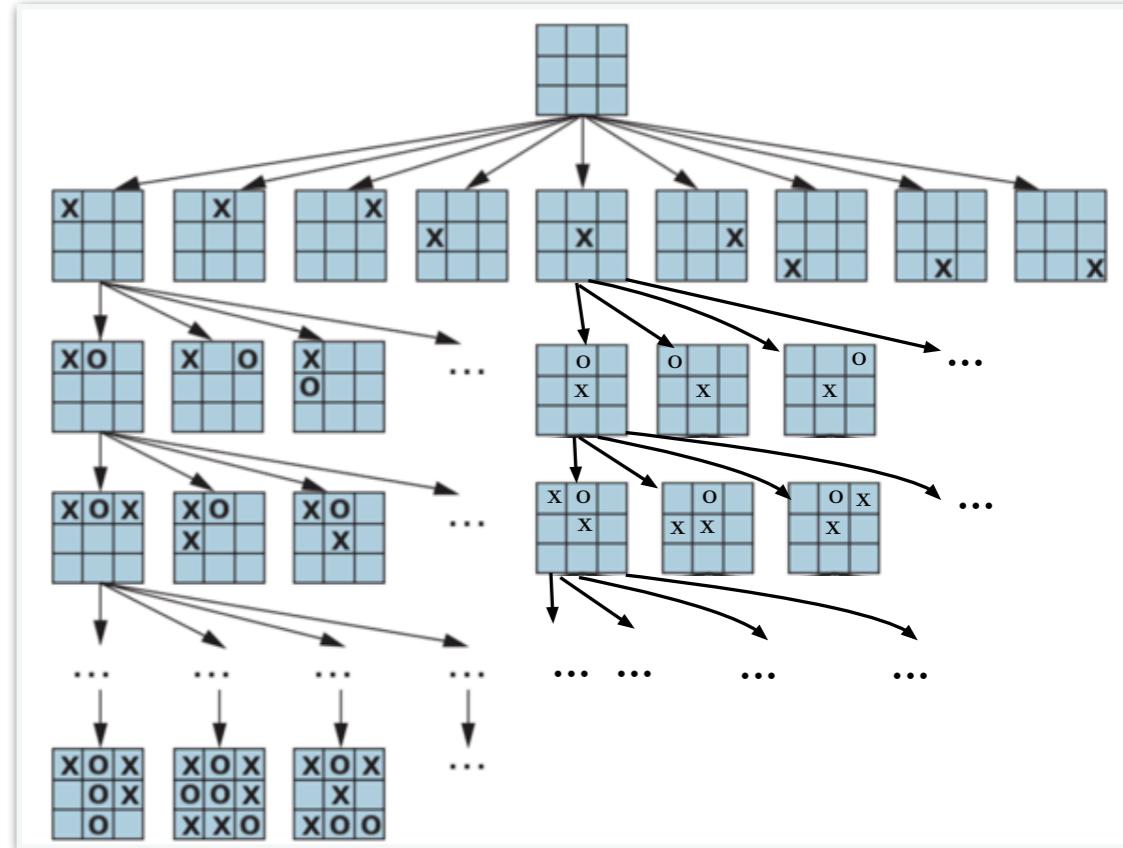
if (lazy_skip(LazyThreshold2))
    goto make_v;

score += threats<WHITE>() - threats<BLACK>()
    + space< WHITE>() - space< BLACK>();
```

Table des matières

Recherche adversarielle

- ✓ 1. Motivation de la recherche adversarielle
- ✓ 2. Catégorisation d'environnements compétitifs et des jeux
- ✓ 3. Stratégie de recherche minimax
- ✓ 4. Mécanisme du *alpha-beta pruning*
- ✓ 5. Conception de fonctions d'évaluation (heuristiques)
- ✓ 6. Mécanismes supplémentaires d'amélioration
- 7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
- 8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
- 9. Exemples et historique d'agents dédiés aux jeux



Motivation d'un autre algorithme de recherche

? Avec les méthodes vues, êtes-vous capables de créer une bonne AI pour n'importe quel jeu ?

En théorie: oui ! (dans un jeu déterministe, tour par tour, etc.)

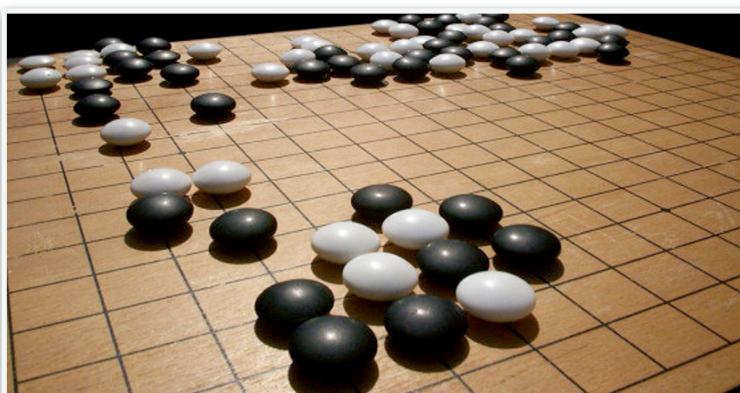
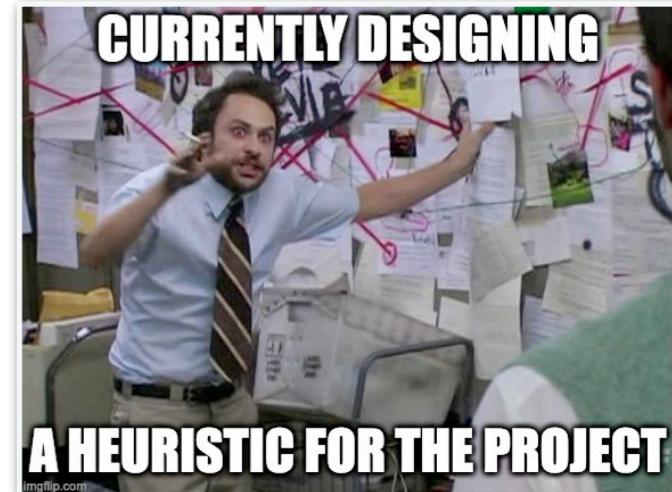
En pratique: on doit être capable de construire une bonne heuristique

Difficulté: les stratégies vues dépendent fortement de la qualité des heuristiques

Conséquence: si l'heuristique est mauvaise, les performances seront médiocres

Difficulté: construire une heuristique comme pour Stockfish demande des connaissances expertes

Conséquence: pour certains jeux, il est très difficile de construire une heuristique convenable



Jeu de Go: le matériel n'est pas une bonne indication d'une victoire

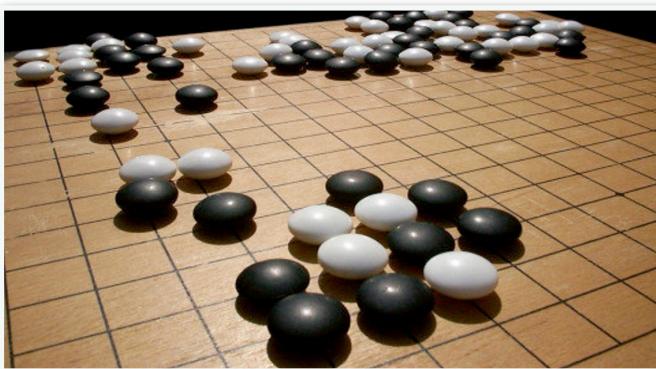
Civilization: règles très complexes



Que faire dans cette situation ?

Arbre de recherche de Monte Carlo - *Monte Carlo Tree Search (MCTS)*

Faiblesses du minimax heuristique avec alpha-beta pruning

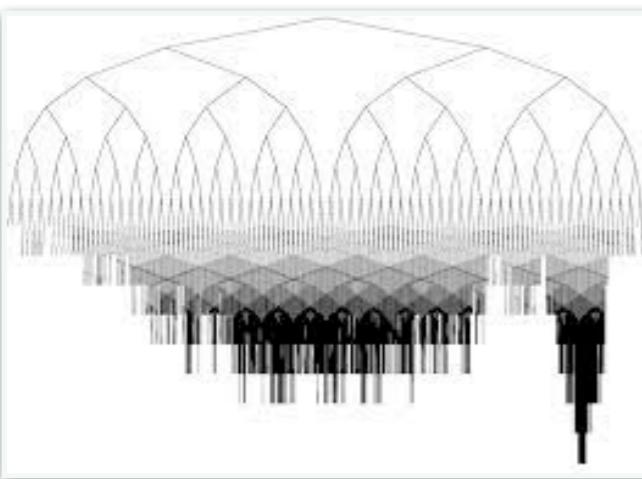


Faiblesse 1: extrêmement dépendant de la qualité de l'heuristique

Faiblesse 2: exploration profonde difficile avec un grand facteur de branchement

Jeu de Go: cas pathologique par excellente ($b = 361$ en début de partie)

Monte Carlo tree search (MCTS)



Stratégie alternative qui n'utilise nativement aucune heuristique

Idée: estimer la qualité des états à l'aide de simulations (aléatoires) de parties

Les états amenant souvent à une victoire seront considérés comme prometteurs

L'arbre est construit en étendant en priorité des états prometteurs

Subtilité: la construction tient aussi compte de l'incertitude que l'on peut avoir sur la qualité d'un état

Avantage: ne demande pas de définir une fonction d'évaluation

Avantage: la stratégie est générique dans le sens où elle peut être exportée facilement de jeu en jeu

Difficulté: la meilleure façon de réaliser les simulations n'est pas clairement définie

Difficulté: de par la nature aléatoire de cette stratégie, on risque de manquer des lignes de jeu importantes

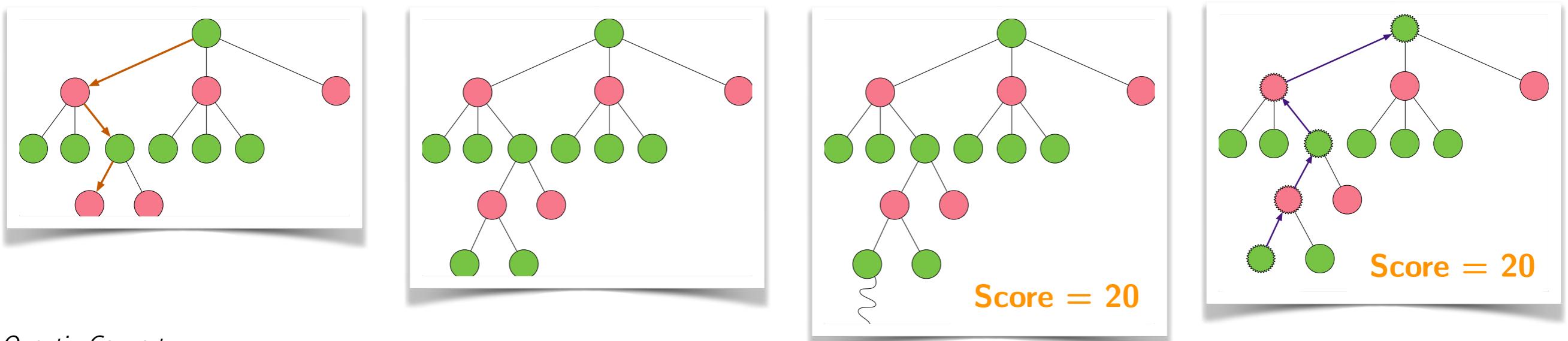
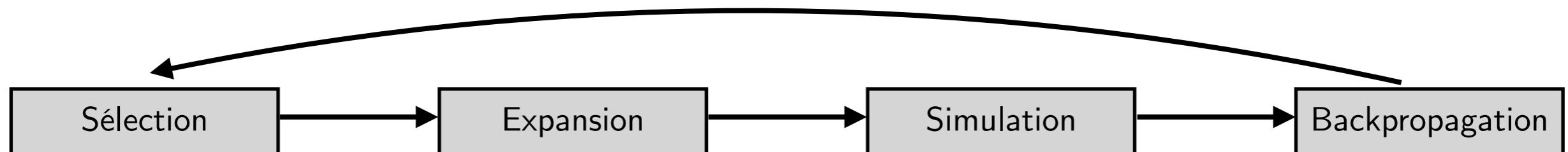
MCTS: Algorithme

Fonctionnement de l'algorithme

Un arbre de recherche est construit successivement via **4 phases fondamentales**

- (1) **Sélection:** à partir de la racine, choisir des actions afin d'arriver à un noeud feuille de l'arbre actuel
- (2) **Expansion:** étendre les noeuds successeurs du noeud feuille précédemment atteint
- (3) **Simulation:** prendre un successeur, simuler une partie à partir de ce noeud, et observer le score obtenu
- (4) **Backpropagation:** propager cette information jusqu'à la racine et mettre à jour certaines valeur

Les étapes (1) à (4) sont répétées jusqu'à ce que le budget temps soit épuisé



MCTS: pseudocode et initialisation



Arbre de recherche de Monte-Carlo (MCTS- *Monte-Carlo tree search*)

Stratégie de recherche basée sur une répétition de ces 4 phases fondamentales

$\text{MCTS}(s_0, \Theta)$:

$t = \text{initTree}(s_0)$

for $i \in 0$ to Θ :

$n_{leaf} = \text{select}(t)$

$n_{child} = \text{expend}(n_{leaf})$

$v = \text{simulate}(n_{child})$

$t = \text{backpropagate}(v, n_{child})$

return $\text{bestAction}(s_0)$



Paramètres: l'état sur lequel exécuter la recherche, et le nombre d'itérations limite

Initialisation de l'arbre de recherche à partir de l'état actuel (position actuelle du plateau)

Itération: tant qu'un nombre d'itérations limite n'est pas atteint, on effectue les 4 phases

Phase de sélection: retourne un noeud feuille en vue d'une extension

Phase d'extension: le noeud feuille est étendu et un de ses fils est retourné

Phase de simulation: le noeud fils est simulé afin d'en estimer sa valeur

Phase de backpropagation: l'information est remontée jusqu'à la racine en modifiant l'arbre

Valeur de retour: on choisit une action à effectuer sur base de l'arbre partiellement construit

Implémentation: fournir une procédure complète pour ces différentes fonctions

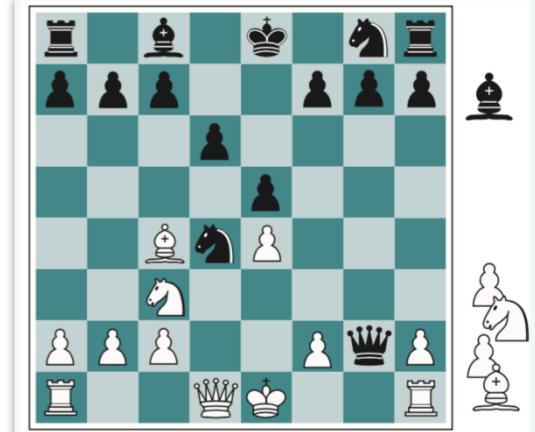
Initialisation: création d'un noeud racine correspondant à l'état actuel du jeu

Principe: deux statistiques seront maintenues pour chaque noeud étendu

$N(n)$: le nombre total de simulations ayant impliqué un noeud n

$U(n)$: Le score total cumulé de toutes les simulations pour un noeud n

Valeur initiale: ces valeurs sont initialisées à **zéro** pour la racine



$$U = 0$$

$$N = 0$$

MCTS: phase de sélection

MCTS(s_0, Θ) :

$t = \text{initTree}(s_0)$

for $i \in 0$ to Θ :

$n_{leaf} = \text{select}(t)$

$n_{child} = \text{expend}(n_{leaf})$

$v = \text{simulate}(n_{child})$

$t = \text{backpropagate}(v, n_{child})$

return $\text{bestAction}(s_0)$

Phase de sélection

Objectif: atteindre un noeud feuille depuis la racine en vue de l'étendre

Exemple: considérons l'arbre suivant obtenu suite à quelques itérations



Comment sélectionner le noeud feuille à étendre ?

C'est ici que les statistiques des noeuds seront utilisées

Première idée: à partir de la racine, choisir chaque fois le noeud ayant obtenu le meilleur score moyen

Intuition: on priorise le scénario qui en moyenne a abouti au meilleur score cumulé lors des simulations

$$\text{averageScore}(n) = \frac{U(n)}{N(n)}$$

$$\text{averageScore}(B) = \frac{12}{30} = 0.4$$

$$\text{averageScore}(C) = \frac{0}{1} = 0$$

$$\text{averageScore}(D) = \frac{5}{10} = 0.5$$

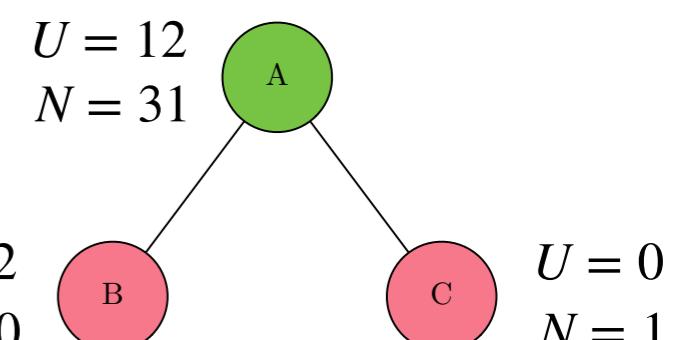
$$\text{averageScore}(E) = \frac{7}{20} = 0.35$$

Au noeud A →

Sélection de B

Au noeud B →

Sélection de D



Résultat: D sera le prochain noeud à étendre

MCTS: phase de sélection - principe UCT



Quelle est la faiblesse de cette idée ?

Principe: on choisit la meilleure action selon l'estimation actuelle

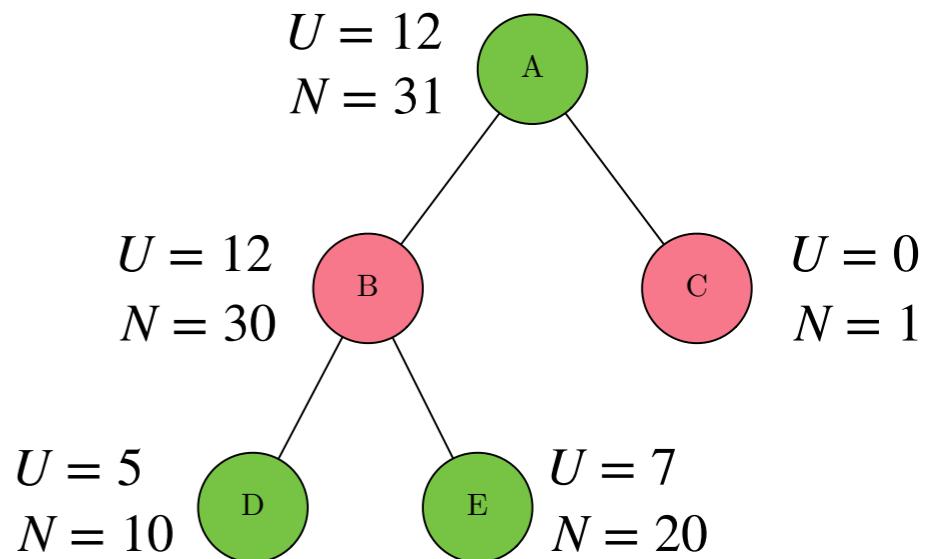
Difficulté: nos estimations sont imparfaites !

Surtout au début de la recherche

Surtout lorsque le nombre de simulation est faible

Exemple: Le noeud C n'a été simulé qu'une seule fois, ainsi l'indication de son score actuel est peu fiable

Correction possible: inciter également à la sélection des noeuds qui ont été peu simulés (haute incertitude)



Règle de sélection UCT (*upper confident bound applied to trees*)

Sélectionne d'un état successeur selon la valeur UCB (*upper confidence bound*)

$$\text{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\ln N(\text{Parent}(n))}{N(n)}}$$

Avec C une constante permettant de balancer le poids donné à l'exploration

Intuition: sélection sur base de la valeur moyenne (terme de gauche) et l'incertitude (terme de droite)

La valeur de C est généralement choisie expérimentalement (valeur théorique de $\sqrt{2}$)

MCTS: phase de sélection - principe UCT

$$\text{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\ln N(\text{Parent}(n))}{N(n)}}$$

Premier terme: favorise l'exploitation d'une bonne action

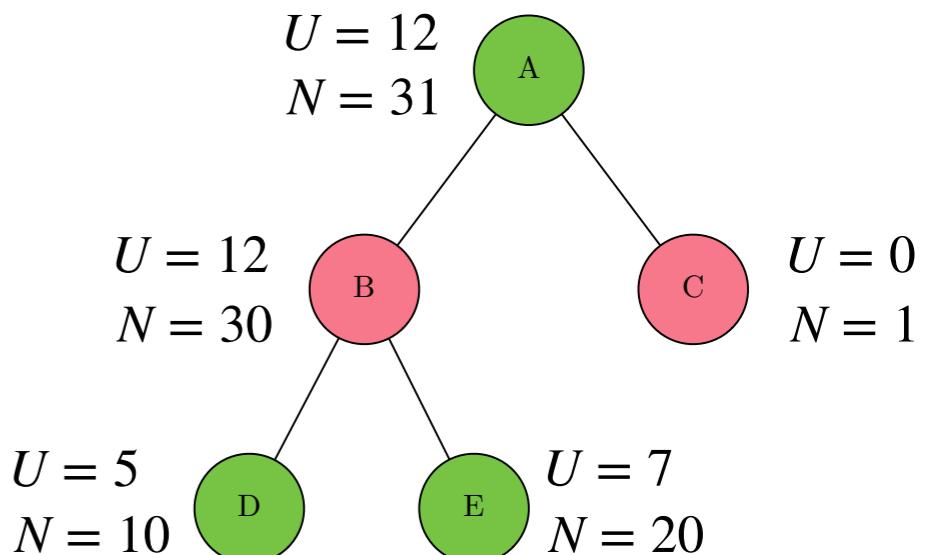
Deuxième terme: favorise l'exploration des noeuds peu simulés

Fonctionnement: plus le nombre de simulation est bas, plus la valeur est grande

$$\text{UCB1}(B) = \frac{12}{30} + 2 \times \sqrt{\frac{\ln 31}{30}} = 1.08$$

$$\text{UCB1}(C) = \frac{0}{1} + 2 \times \sqrt{\frac{\ln 31}{1}} = 3.7$$

Sélection de l'action menant au noeud C



Autres règles de sélection (à ne pas connaître)

$$\text{UCB1tuned}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\ln N(\text{Parent}(n))}{N(n)} \times \min\left(\frac{1}{4}, V(n) + \frac{2 \ln(\text{Parent}(n))}{N(n)}\right)}$$

Avec $V(n)$, la variance d'effectuer l'action menant au noeud n

$$\text{AlphaZeroSelection}(n) = \frac{U(n)}{N(n)} + C \times P(n) \times \sqrt{\frac{N(\text{Parent}(n))}{1 + N(n)}}$$

Avec $P(n)$, une probabilité de sélectionner l'action menant au n (calculée via un réseau de neurones)

MCTS: phase d'expansion

Phase d'expansion

Déclenchement: dès qu'un noeud feuille est atteint lors de la sélection

Objectif: générer les successeurs du noeud feuille

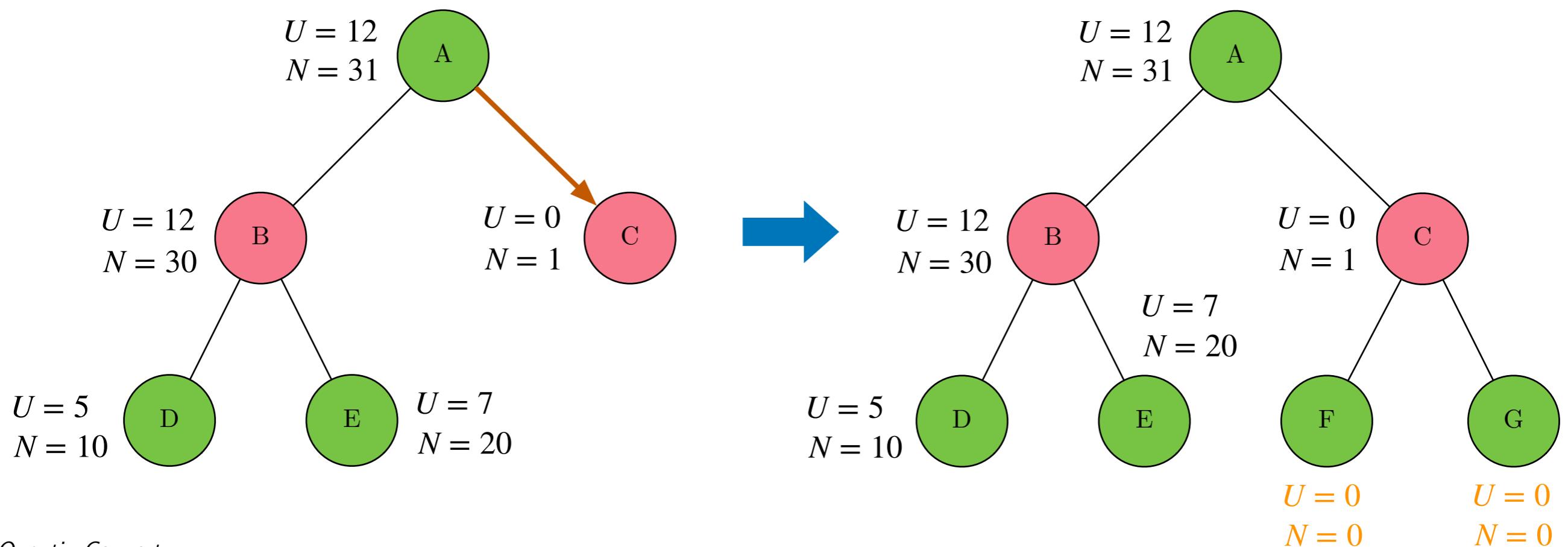
Conséquence: le noeud n'est plus une feuille (sauf si l'état est terminal)

Condition d'expansion: que si le noeud a déjà été simulé

Condition pour l'expansion : $N(n) > 0$

```
MCTS( $s_0, \Theta$ ) :  
     $t = \text{initTree}(s_0)$   
    for  $i \in 0$  to  $\Theta$  :  
         $n_{leaf} = \text{select}(t)$   
         $n_{child} = \text{expend}(n_{leaf})$   
         $v = \text{simulate}(n_{child})$   
         $t = \text{backpropagate}(v, n_{child})$   
    return  $\text{bestAction}(s_0)$ 
```

Initialisation d'un noeud: Les statistiques du noeud sont mises à 0



MCTS: phase de simulation

Phase de simulation (*rollout, playout*)

Déclenchement: dès qu'un noeud feuille atteint (sélection ou expansion)

Principe: simuler le reste d'une partie à partir de cet état

Objectif: obtenir une estimation de la qualité de l'état

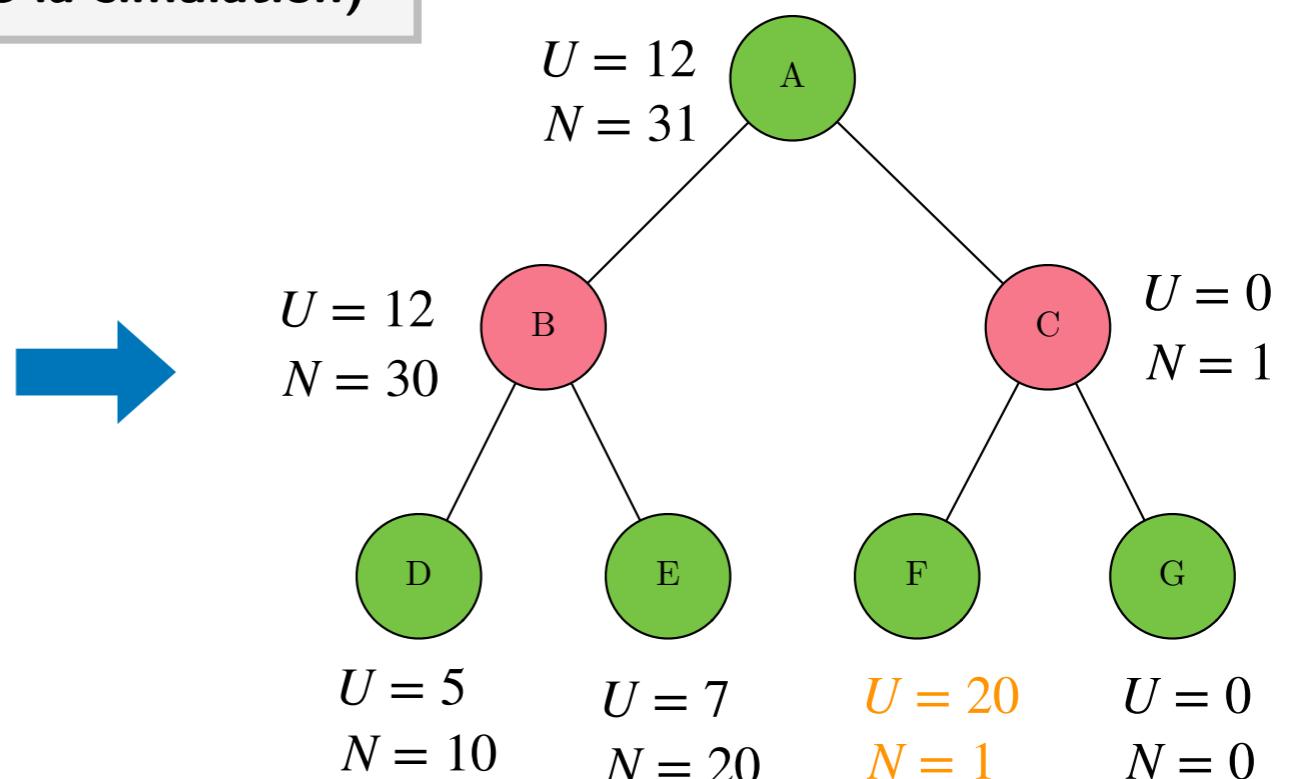
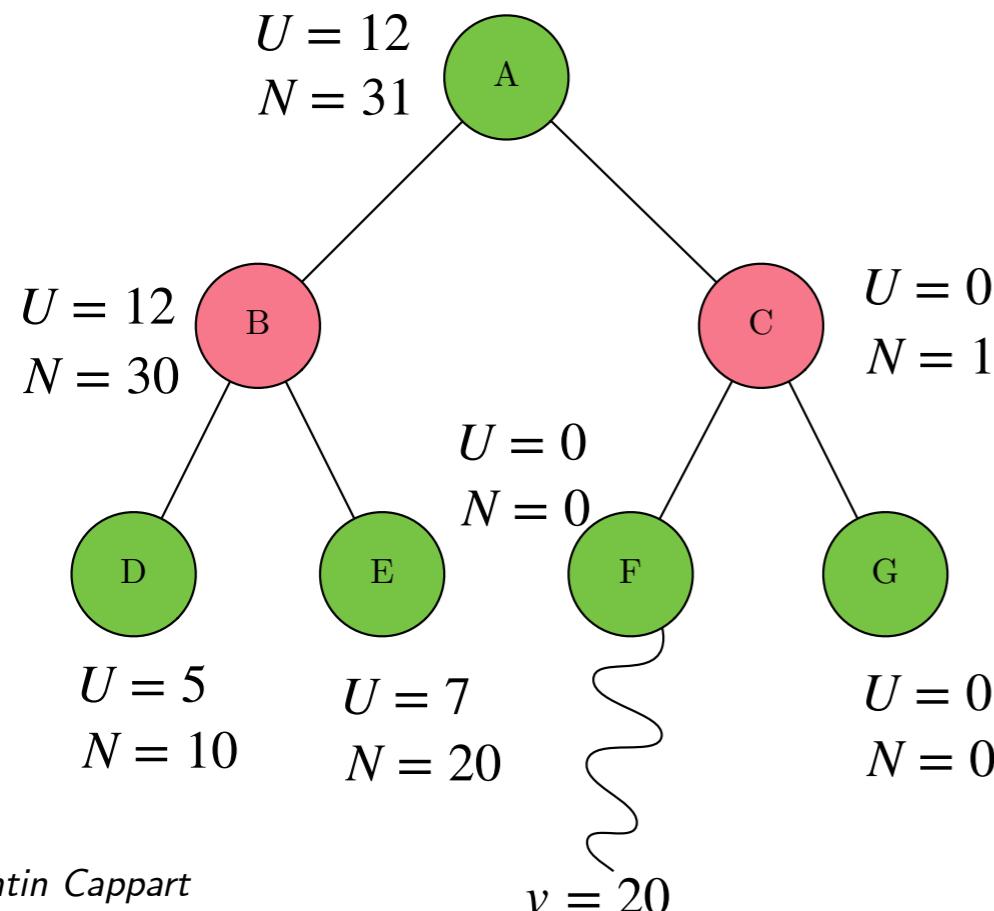
Valeur de retour: Le score final obtenu à l'issue de la simulation

Conséquence: les statistiques du noeud simulé sont mises à jour

$N(n) = 1$ (une seule simulation a été faite pour ce noeud)

$U(n) = v$ (la valeur de ce noeud est le résultat de la simulation)

```
MCTS( $s_0, \Theta$ ) :  
     $t = \text{initTree}(s_0)$   
    for  $i \in 0$  to  $\Theta$  :  
         $n_{leaf} = \text{select}(t)$   
         $n_{child} = \text{expend}(n_{leaf})$   
         $v = \text{simulate}(n_{child})$   
         $t = \text{backpropagate}(v, n_{child})$   
    return bestAction( $s_0$ )
```

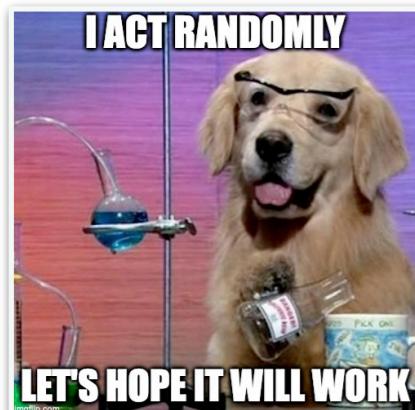


?

Comment réaliser une (bonne) simulation ?

MCTS: phase de simulation

Idée 1: simulation aléatoire



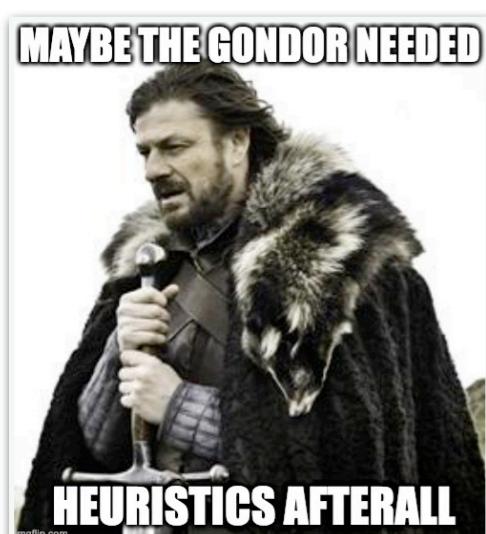
Principe: les deux joueurs choisissent leurs actions aléatoirement lors des simulations

Avantage: peut fonctionner pour des jeux simples (donne une stratégie potable)

Avantage: très simple à mettre en place

Inconvénient: performances moindres pour des jeux plus complexes

Idée 2: intégration d'heuristiques



Principe: utiliser une heuristique pour biaiser les simulations vers des bonnes actions

Exemple: donner une probabilité de sélection plus haute à une capture aux échecs

Avantage: donne de meilleurs résultats en pratique

Avantage: la dépendance à l'heuristique est moindre que pour le cas minimax

Inconvénient: demande quand même d'intégrer des heuristiques, souvent non triviales

Idée 3: principe de l'auto-entraînement (*self-play*)



Principe: faire jouer l'agent contre lui-même lors des simulations

(1) Partir d'un agent aléatoire

(2) mettre à jour l'agent après x itérations, et l'utiliser pour les nouvelles simulations

Intuition: l'agent va rencontrer un adversaire de plus en plus efficace

AlphaGo Zero: 29 millions de parties en self-play

MCTS: phase de backpropagation

Phase de backpropagation

Déclenchement: après chaque simulation

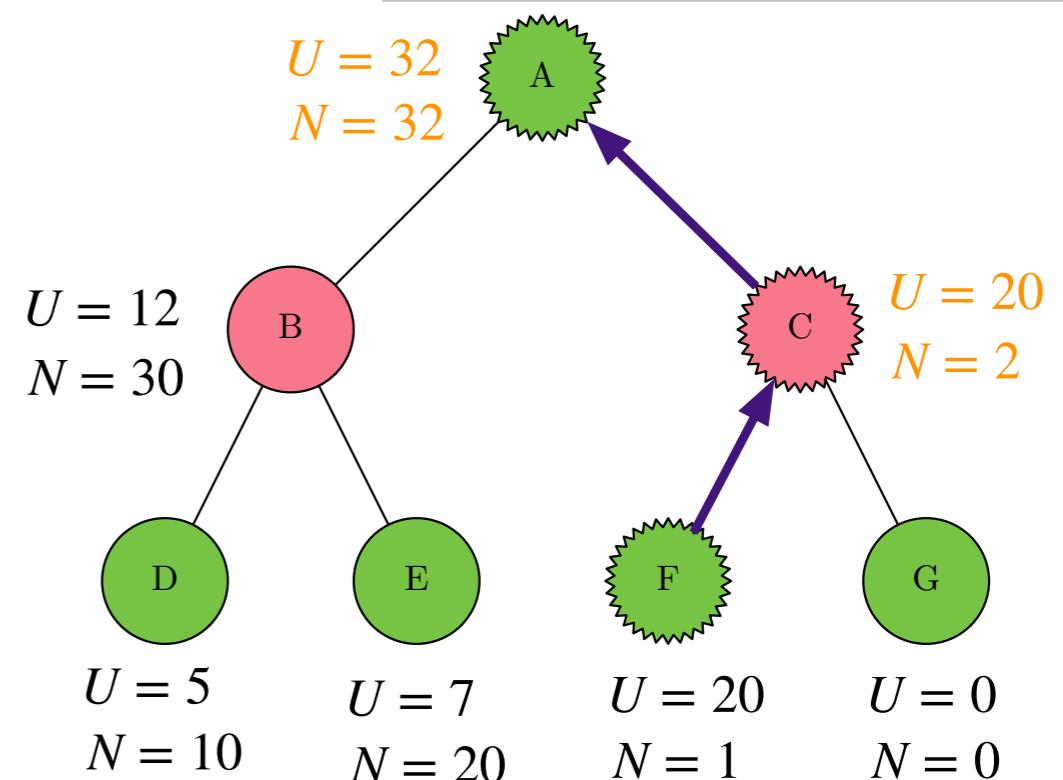
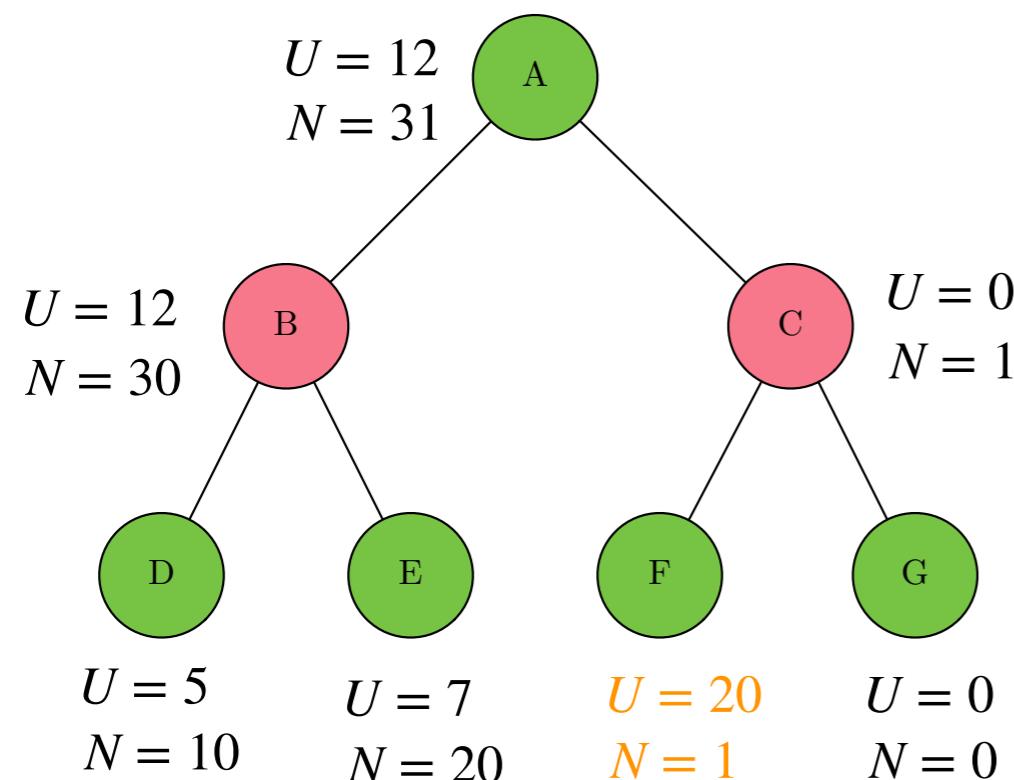
Objectif: mettre à jour les statistiques des noeuds après la simulation

Impact: sur tous les noeuds du chemin jusqu'à la racine

$N(n) = N(n) + 1$ (on a fait une simulation de plus impliquant ce noeud)

$U(n) = U(n) + v$ (le score de ce noeud est mis à jour avec le résultat)

```
MCTS( $s_0, \Theta$ ) :  
     $t = \text{initTree}(s_0)$   
    for  $i \in 0$  to  $\Theta$  :  
         $n_{leaf} = \text{select}(t)$   
         $n_{child} = \text{expend}(n_{leaf})$   
         $v = \text{simulate}(n_{child})$   
         $t = \text{backpropagate}(v, n_{child})$   
    return  $\text{bestAction}(s_0)$ 
```



Prochaine itération: on répète le processus avec le nouvel arbre et les statistiques mises à jour

Critère d'arrêt: après la réalisation de x itérations, ou sur base d'un budget temps alloué

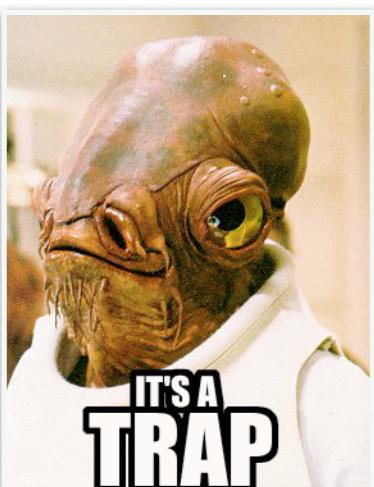
MCTS: choix de l'action à exécuter

Choix de l'action

Objectif principal: déterminer la meilleure action à exécuter



Quelle action choisiriez vous ? celle amenant à *B* ou *C* ?



Choix intuitif: l'action ayant le meilleur score moyen

$$\text{bestAction}(n) = \frac{U(n)}{N(n)} \rightarrow \text{Selection de } C$$

Ce choix est très dangereux !



Quel est le soucis de cette sélection ?

Danger: engendre un haut risque de favoriser les actions incertaines

Exemple: seulement 2 simulations ont été faites pour *C*

Meilleur choix: prendre l'action qui a été simulée le plus de fois

$$\text{bestAction}(n) = N(n) \rightarrow \text{Selection de } B$$

MCTS(s_0, Θ):

$t = \text{initTree}(s_0)$

for $i \in 0$ to Θ :

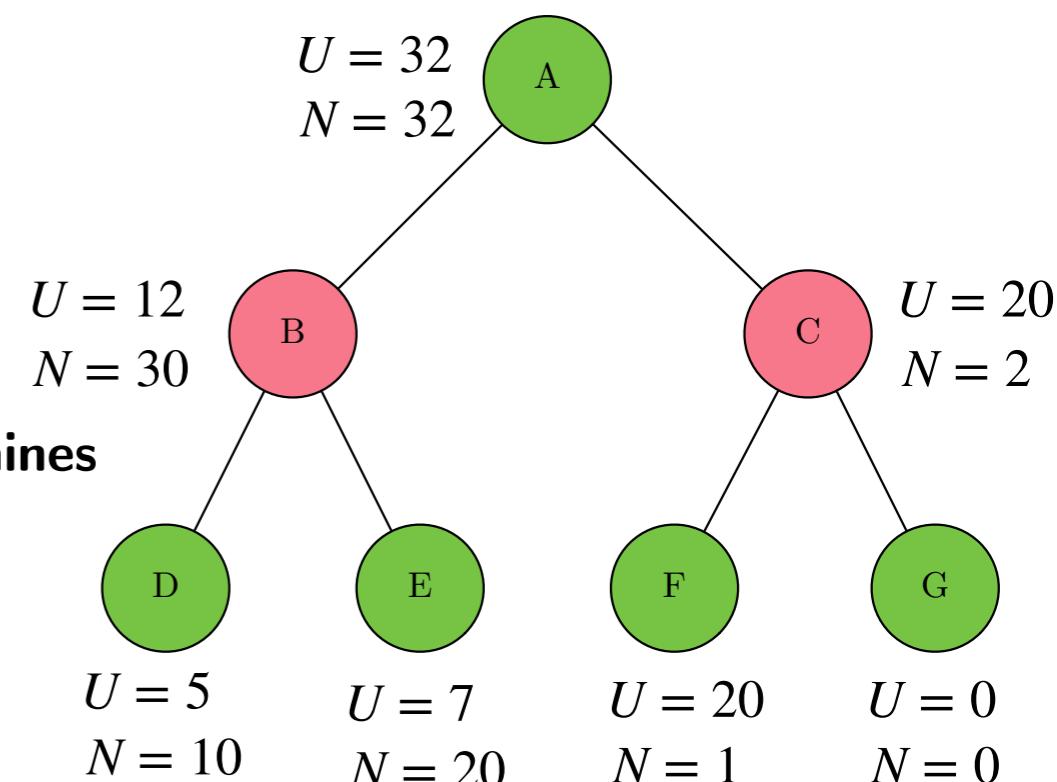
$n_{leaf} = \text{select}(t)$

$n_{child} = \text{expend}(n_{leaf})$

$v = \text{simulate}(n_{child})$

$t = \text{backpropagate}(v, n_{child})$

return **bestAction(s_0)**



Intuition: la sélection UCB assure déjà que les actions les plus simulées sont de bonne qualité

MCTS vs Minimax (avec alpha-beta pruning et heuristiques)

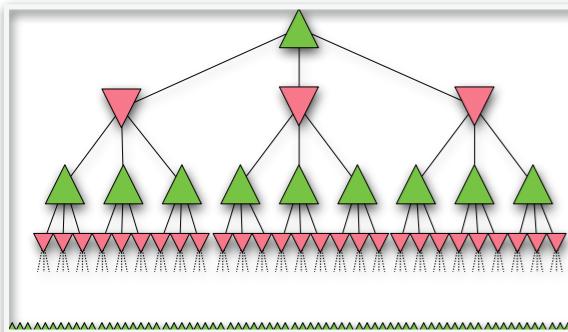


Quand devriez choisir une méthode ou une autre ?

Difficile à répondre, sans réponse claire et définitive

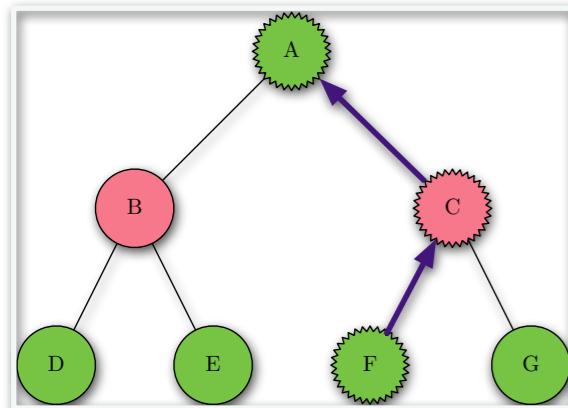
On peut uniquement énoncer certaines observations

Arguments en faveur de Minimax (+ améliorations)



- (1) Lorsque vous avez une excellente heuristique que vous souhaitez utiliser
- (2) Lorsque le facteur de branchement est faible (exploration plus profonde)
- (3) Lorsque des actions critiques sont impliquées (action ayant un gros impact)

Arguments en faveur de MCTS



- (1) Lorsqu'il est très difficile de concevoir une bonne heuristique
- (2) Lorsque le facteur de branchement est grand (difficulté pour minimax)
- (3) Lorsque vous voulez facilement transposer votre agent à un autre jeu

Bonus: se combine assez bien avec de l'apprentissage automatique

Hybridation possible: construire un agent basé sur ces deux stratégies

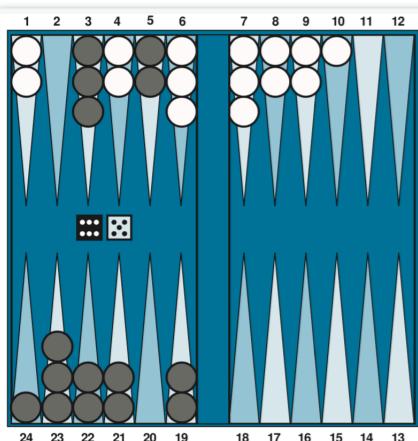
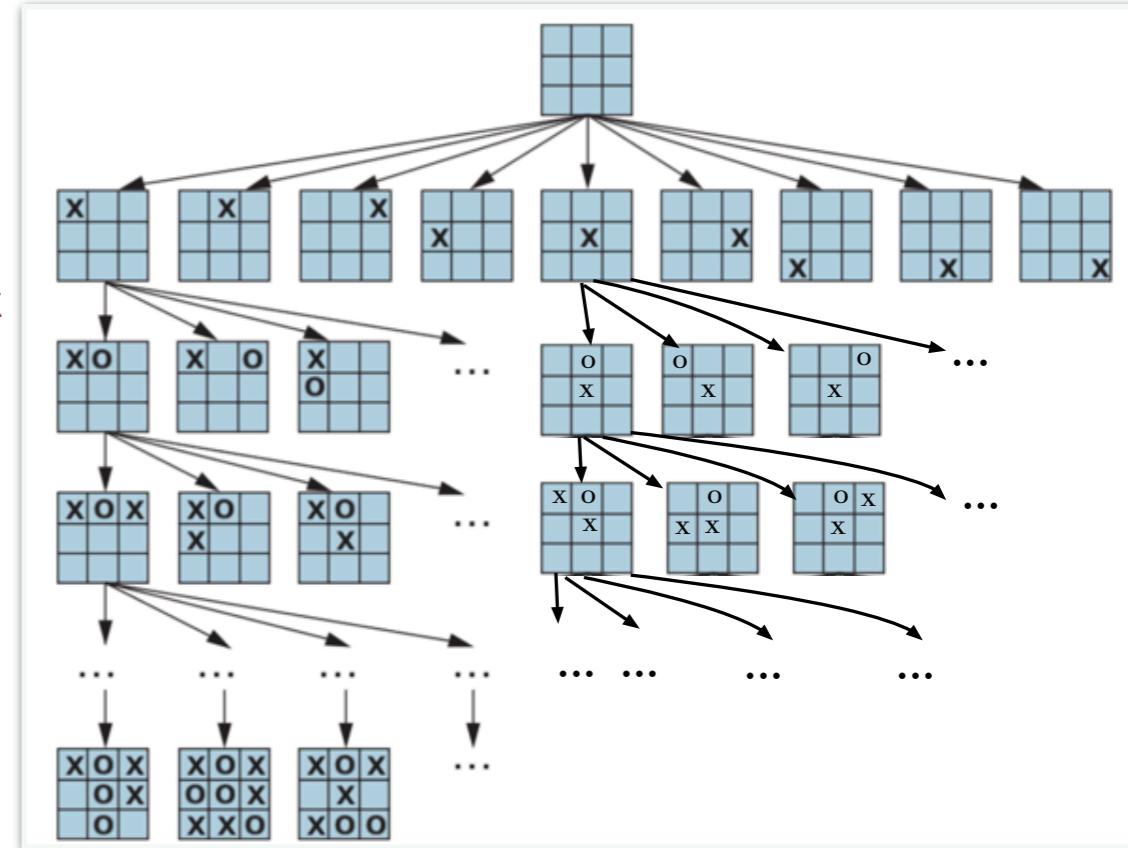
Exemple: utilisation d'une heuristique dans une simulation

Exemple: MCTS en début de partie, et minimax par la suite

Table des matières

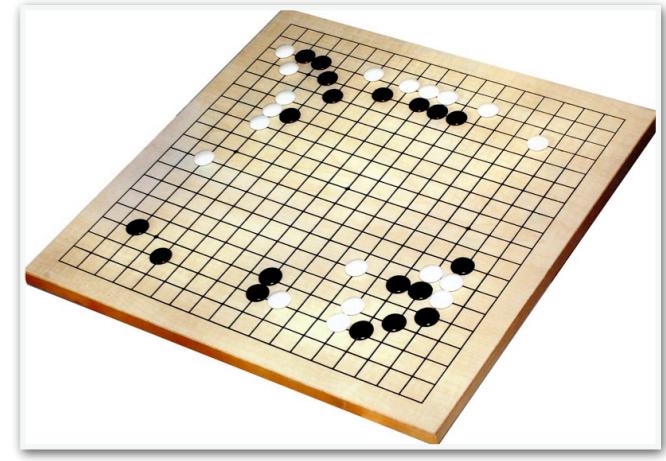
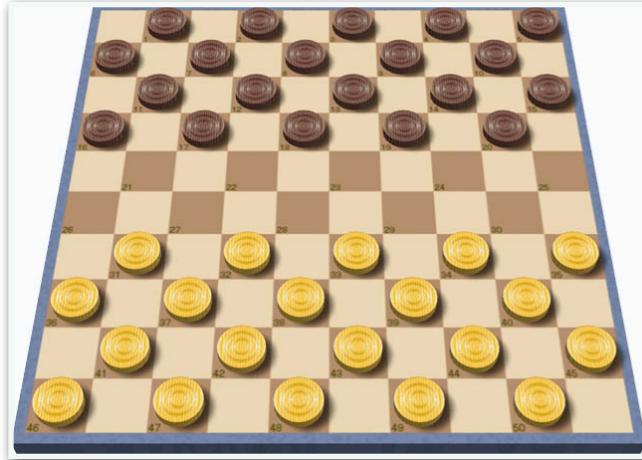
Recherche adversarielle

- ✓ 1. Motivation de la recherche adversarielle
- ✓ 2. Catégorisation d'environnements compétitifs et des jeux
- ✓ 3. Stratégie de recherche minimax
- ✓ 4. Mécanisme du *alpha-beta pruning*
- ✓ 5. Conception de fonctions d'évaluation (heuristiques)
- ✓ 6. Mécanismes supplémentaires d'amélioration
- ✓ 7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
- 8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
- 9. Exemples et historique d'agents dédiés aux jeux



Extension à d'autres types de situations

Cadre actuel: jeux à deux joueurs, déterministe, tour par tour, à information parfaite, et à somme nulle



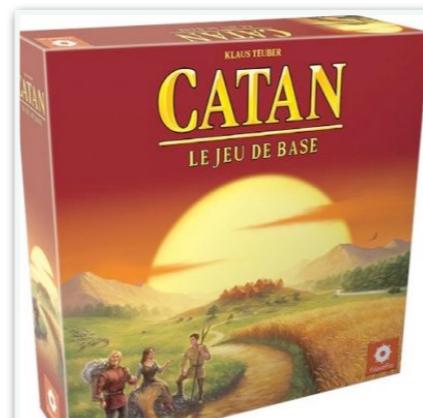
Hypothèse fondamentale: on a supposé que notre adversaire était rationnel (qu'il joue le mieux possible)

Limitation: il existe une multitude d'autres jeux (ou situations réelles) qu'on ne sait pas encore gérer

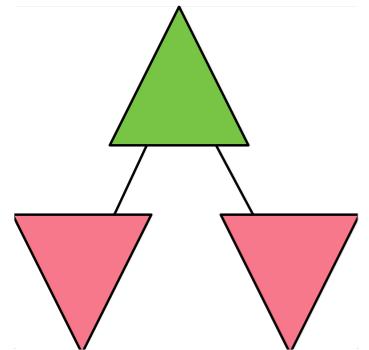
? Que se passe t-il si on sort de ce cadre ?

Autres variantes: plus de deux joueurs, sujets à l'aléatoire, avec des informations cachées, etc.

Relaxation de l'hypothèse fondamentale: l'adversaire n'est pas forcément rationnel



Minimax avec des joueurs non rationnels



Rappel minimax: on a la garantie que l'action choisie est optimale (si la recherche est finie)

Supposition: l'adversaire est rationnel et va effectuer la meilleure action possible

En pratique: on a aucune assurance que l'adversaire joue rationnellement



Est-ce que minimax garde son caractère optimal ?

Bonne nouvelle: on garde l'assurance d'avoir un score au moins aussi bon

Intuition: minimax envisage le pire scénario qui peut se produire pour nous

Mauvaise nouvelle: on risque de manquer un meilleur score

Exemple: arbre à deux niveaux de décisions

(1) **Raisonnement minimax:** amène à exécuter l'action *a*

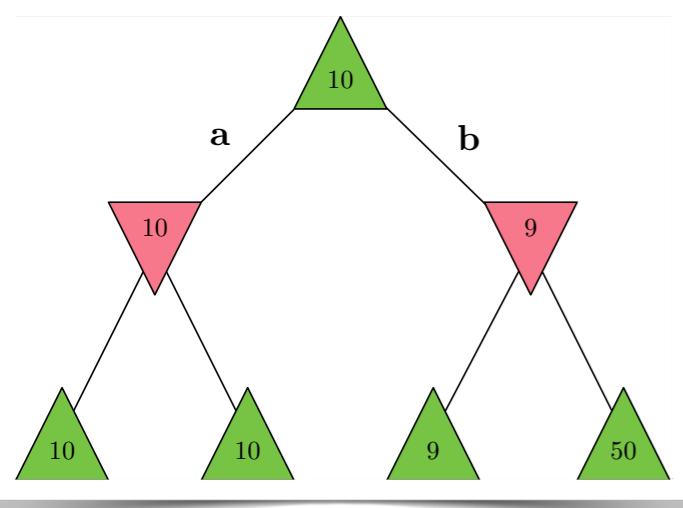
Résultat: le score obtenu sera alors de 10

(2) **En cas d'erreur:** si l'adversaire fait une erreur (action *b*)

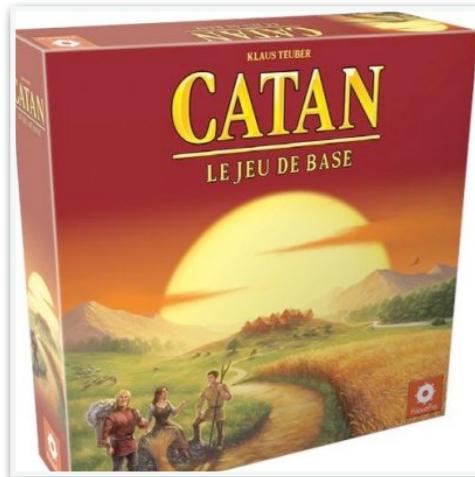
Résultat possible: on aurait pu obtenir un score de 50

Piste d'amélioration: modéliser le comportement non-optimal de l'adversaire

Note: MCTS n'est pas sensible à cela (ne suppose pas un adversaire optimal)



Minimax avec plus de deux joueurs

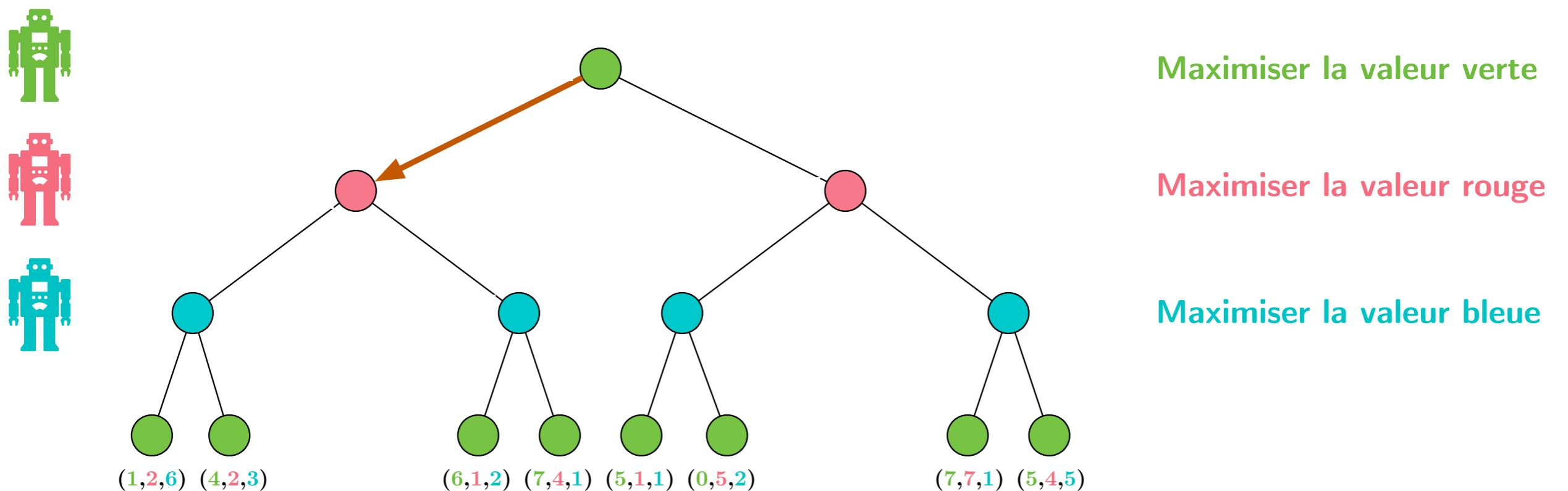


Jeu multi-joueur: jeu qui implique plus de deux joueurs

Conséquence: chaque joueur a sa propre fonction d'utilité qu'il souhaite maximiser

Etat terminal: donne le score atteint pour chaque joueur

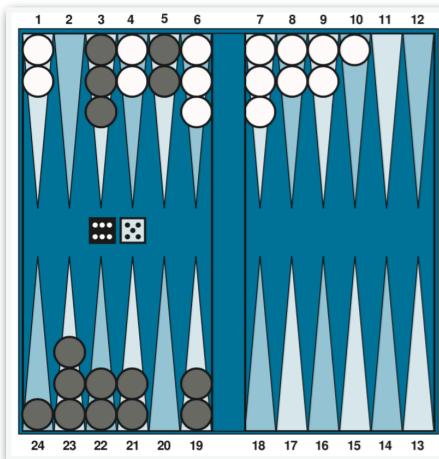
Adaptation du minimax: chaque joueur va faire le meilleur coup possible pour lui



Dynamique du jeu: en fonction des intérêts communs, des alliances peuvent être formées et détruites

Note: le jeu n'est pas forcément à somme nulle

Minimax avec la présence d'aleatoire



Jeu stochastique: jeu qui implique une présence d'aleatoire

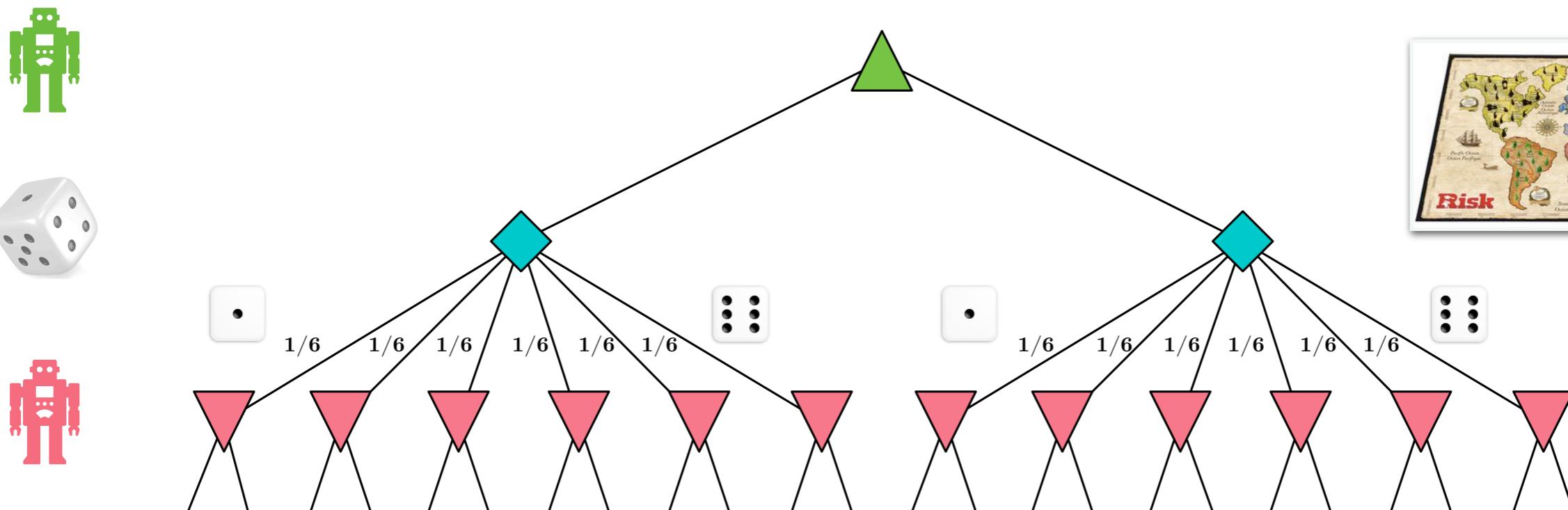
Aléatoire explicite: dynamique propre du jeu (p.e., un lancement de dés)

Adversaires imprévisibles: l'adversaire pas toujours rationnel

Actions imprévisibles: l'action faite peut échouer (tir au but d'un joueur)

Point de considération: rajouter cette notion dans une recherche minimax

Chance node: nouveau type de noeud, représentant les différents scénarios possibles



Niveau 1: le joueur actuel (MAX) a un choix entre deux actions

Niveau 2: un évènement aléatoire se produit (lancer de dé de l'adversaire avec 6 résultats équiprobables)

Niveau 3: l'adversaire (MIN) a ensuite un choix entre deux actions

Valeur Expectiminimax



Valeur Expectiminimax

Valeur qui indique le score final **espéré** (pour le joueur MAX) qui sera obtenu à partir d'un état, si les deux joueurs jouent de façon optimale

$$\text{ExpMinimax}(s) = \begin{cases} \text{utility}(s, \text{maxPlayer}) & \text{if } \text{isTerminal}(s) \\ \max_{a \in \text{actions}(s)} \text{ExpMinimax}(\text{transition}(s, a)) & \text{if } \text{turn}(s) = \text{maxPlayer} \\ \min_{a \in \text{actions}(s)} \text{ExpMinimax}(\text{transition}(s, a)) & \text{if } \text{turn}(s) = \text{minPlayer} \\ \sum_{a \in \text{actions}(s)} P(a) \text{ExpMinimax}(\text{transition}(s, a)) & \text{if } \text{turn}(s) = \text{chanceNode} \end{cases}$$

Note: à un **chance node**, une action correspond à une possibilité d'un tirage aléatoire

Probabilité d'une action (P): chaque action aléatoire a une certaine probabilité d'être obtenue

Valeur minimax: reflète le résultat obtenu en **une situation extrême** (meilleure stratégie des deux joueurs)

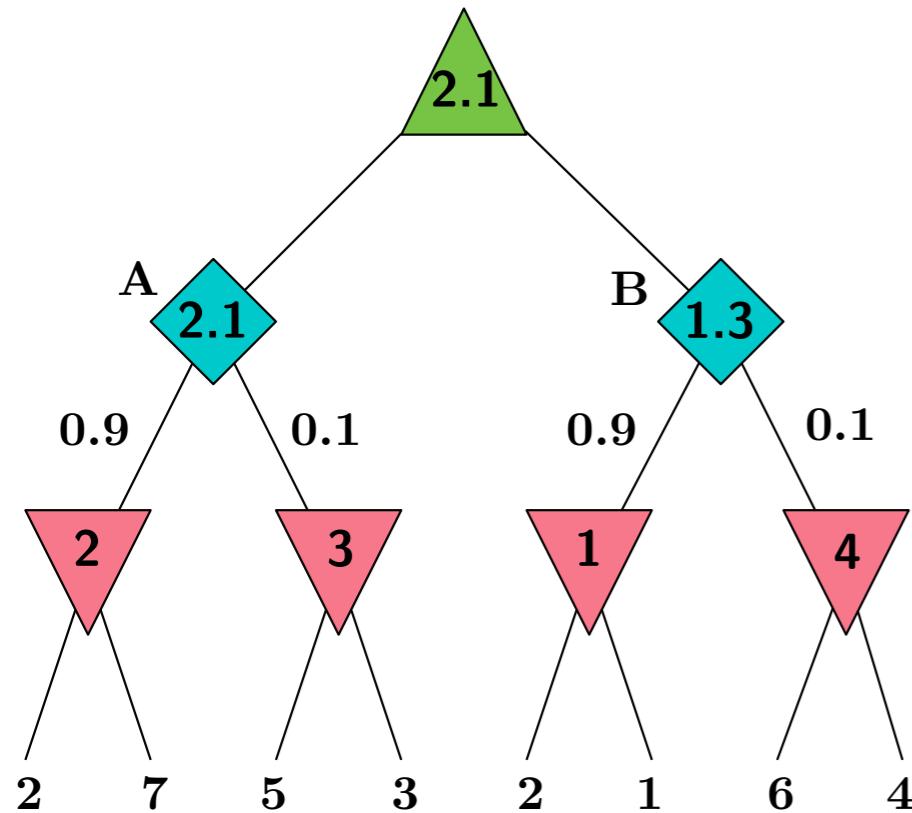
Valeur expectiminimax: tient également compte **du résultat moyen** obtenu lors des **chance nodes**

Interprétation: représente le meilleur résultat moyen pouvant être obtenu si les deux joueurs sont optimaux

Valeur Expectiminimax



Quelle action doit être faite si on veut maximiser son score moyen ?



(1) **Noeuds MIN:** on prend la valeur minimale

(2) **Noeuds aléatoires:** on prend la valeur moyenne (espérance)

$$A : 0.9 \times 2 + 0.1 \times 3 = 2.1$$

$$B : 0.9 \times 1 + 0.1 \times 4 = 1.3$$

(3) **Noeud MAX:** on prend la valeur maximale

Conclusion expectiminimax: on va choisir l'action A

Considérations supplémentaires

Taille de l'arbre: considérablement plus grand car il faut explorer toutes les possibilités

Alpha-beta pruning: peut-on toujours supprimer des branches ? (oui, sous certaines conditions)

Fonction heuristique: comment les concevoir ?

Mauvaise nouvelle: en pratique, l'ajout d'aleatoire complexifie énormément le problème

Jeux partiellement observables

Jeux partiellement observables

Caractéristiques principales: certains éléments du jeu ne sont pas visibles pour les joueurs

Informations cachées pour tous: informations non connues de tous les joueurs (cartes communes au poker)

Informations exclusives: informations disponibles seulement pour certains joueurs (unités au Stratègo)



Starcraft



Poker



Whist



Stratègo

Particularités des stratégies pour ces jeux

Aspect 1: utiliser au mieux les informations qui nous sont disponibles (exploiter une vue locale)

Aspect 2: acquérir de nouvelles informations (avoir une vue globale)

Aspect 3: tenir compte que l'adversaire a également des informations manquantes

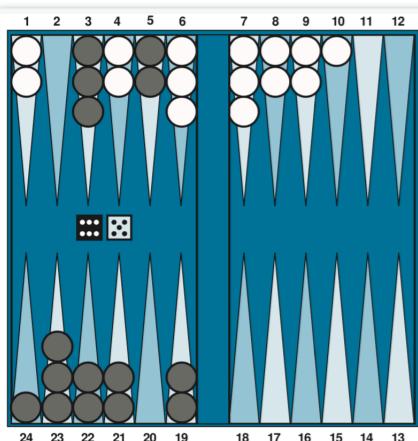
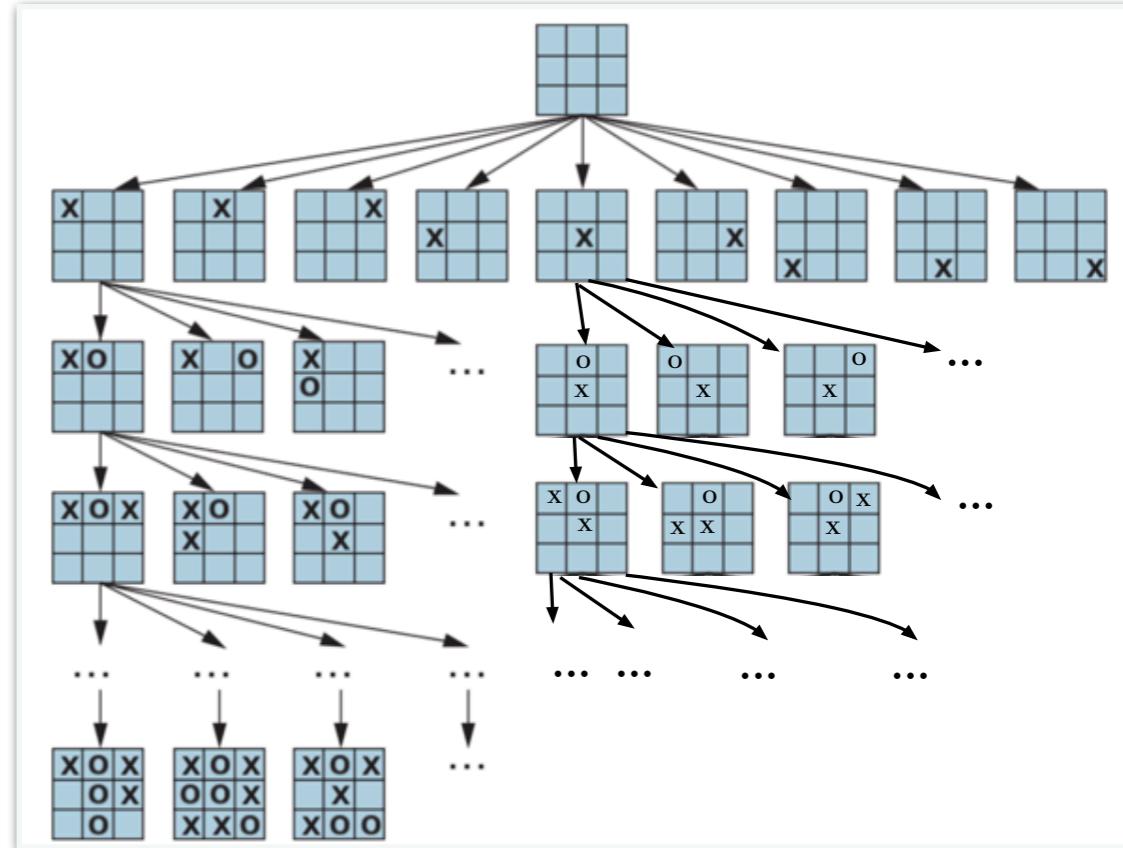
Une bonne stratégie passe généralement par la construction d'une représentation de l'environnement

Mauvaise nouvelle: il s'agit de situations très complexes sortant du cadre de ce cours

Table des matières

Recherche adversarielle

- ✓ 1. Motivation de la recherche adversarielle
- ✓ 2. Catégorisation d'environnements compétitifs et des jeux
- ✓ 3. Stratégie de recherche minimax
- ✓ 4. Mécanisme du *alpha-beta pruning*
- ✓ 5. Conception de fonctions d'évaluation (heuristiques)
- ✓ 6. Mécanismes supplémentaires d'amélioration
- ✓ 7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
- ✓ 8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
- 9. Exemples et historique d'agents dédiés aux jeux



Historique des agents dédiés aux jeux

? Comment caractériser les performances d'un agent intelligent par rapport à la performance humaine ?

Niveau débutant: performance en dessous d'un joueur humain raisonnable (pas trop nul)

Niveau intermédiaire: performance d'un joueur moyen

Niveau expert: capable de battre la majorité des joueurs

Niveau surhumain: capable de battre les meilleurs joueurs au monde

Jeu résolu: possible de prédire le résultat d'une partie si les deux joueurs sont optimaux (**niveau ultime**)

Puissance 4



Etat de l'art: Jeu résolu (1988)

Résultat: le premier joueur peut forcer la victoire s'il joue de façon optimale

Preuve: basée sur des raisonnements logiques

Jeu de dames (checkers)



Etat de l'art: jeu résolu (2007)

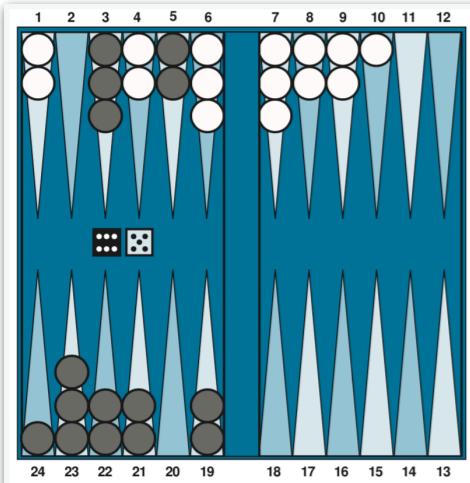
Résultat: amène à une égalité si les deux joueurs sont optimaux

Preuve: basée sur de l'alpha-beta pruning et une base de donnée des fins de partie

Lecture complémentaire disponible sur Moodle

Historique des agents dédiés aux jeux

Backgammon



Difficulté du jeu: fait intervenir de l'aléatoire (lancer de deux dés)

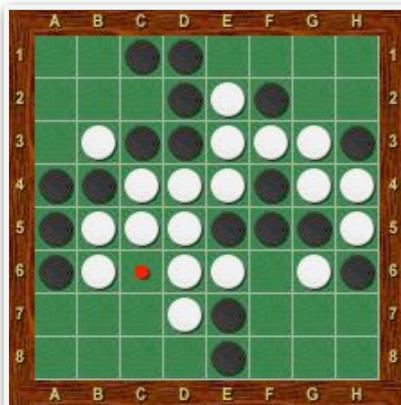
Etat de l'art: performance surhumaine (jeu non résolu)

Moment clé: BKG a battu un champion mondial (1980)

Caractéristiques de l'algorithme: excellente fonction d'évaluation

Postérité: première IA capable de battre un champion pour un jeu complexe

Othello



Difficulté du jeu: il est difficile de créer une bonne fonction d'évaluation

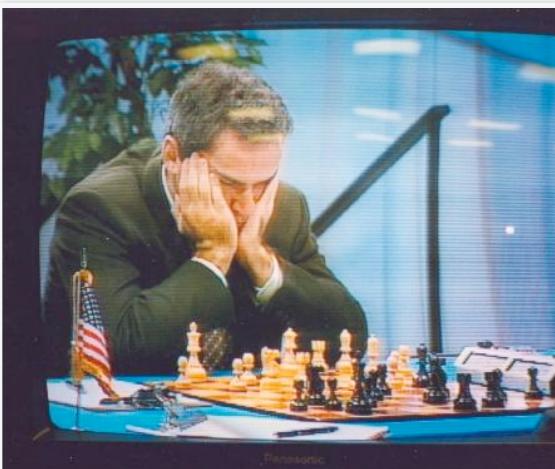
Etat de l'art: performance surhumaine (jeu non résolu)

Moment clé: Logistello a battu un champion mondial (1997)

Heuristique: millions de paramètres calibrés avec une régression linéaire

<https://skatgame.net/mburo/log.html>

Jeu des échecs



Etat de l'art: performance surhumaine (jeu non résolu)

Moment clé: DeepBlue a battu le champion Garry Kasparov (1997)

Caractéristique: minimax intégrant de l'alpha-beta pruning et des heuristiques

Performances: capable d'évaluer 100 millions de positions par seconde

Performances: capable d'explorer 40 coups à l'avance

Historique des agents dédiés aux jeux

Jeu de Go



Etat de l'art: performance surhumaine (jeu non résolu)

Moment clé: AlphaGo a battu le champion Lee Sedol (2015)

Mécanismes utilisés: MCTS, apprentissage supervisé, et par renforcement

Caractéristique: utilisation d'un réseau de neurones (très) profond

Postérité: AlphaGo Zero qui n'utilise plus de connaissances expertes

Poker



Difficulté du jeu: présence d'aléatoire et d'informations cachées

Etat de l'art: performance surhumaine (jeu non résolu)

Moment clé: Libratus a battu des champions (parties à 2 joueurs, 2017)

Moment clé: Pluribus a battu des champions (parties à 6 joueurs, 2019)

Mécanismes utilisés: apprentissage profond, et notions de théorie des jeux

Starcraft 2



Difficulté du jeu: informations cachées et jeu temps-réel

Etat de l'art: performance surhumaine (jeu non résolu)

Moment clé: AlphaStar a battu des champions mondiaux (2019)

Mécanismes utilisés: apprentissage par renforcement profond

Caractéristique: utilisation intensive du mécanisme de *self-play*

Historique des agents dédiés aux jeux

Il y a encore plein d'autres jeux où l'IA atteint des performances surhumaines



Scrabble (2006)



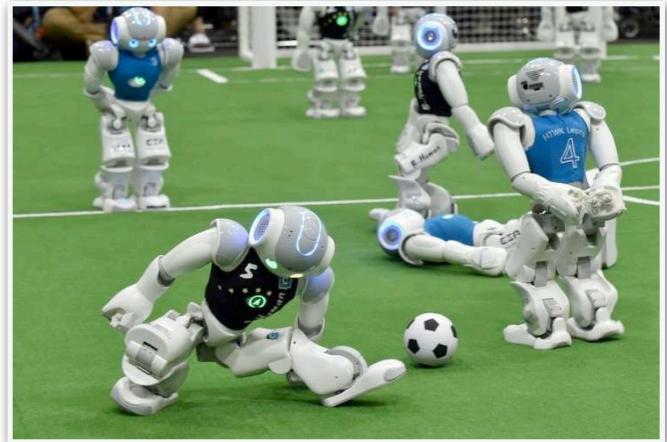
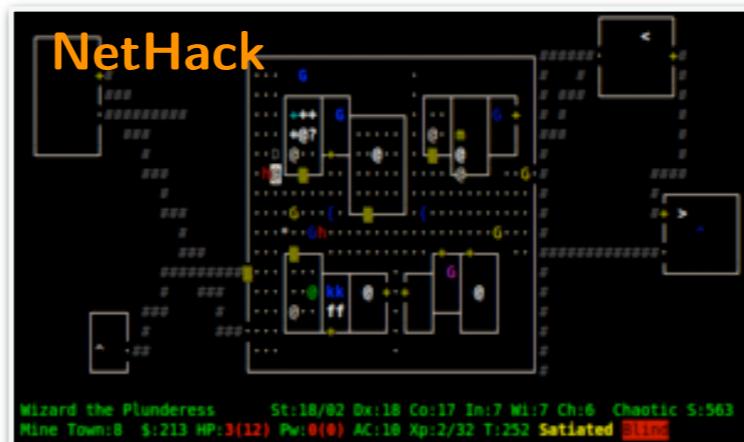
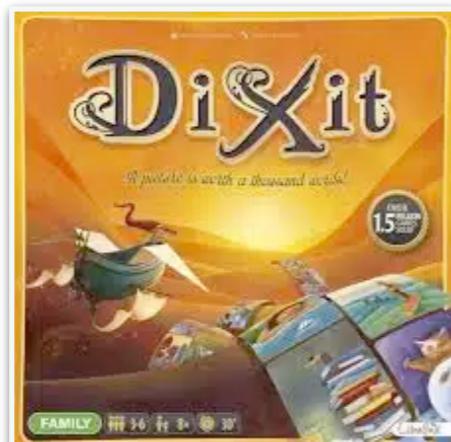
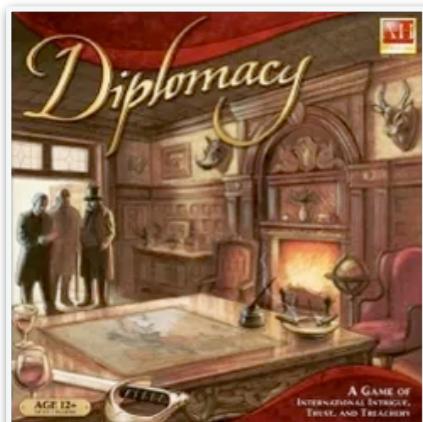
Dota 2 (2018)



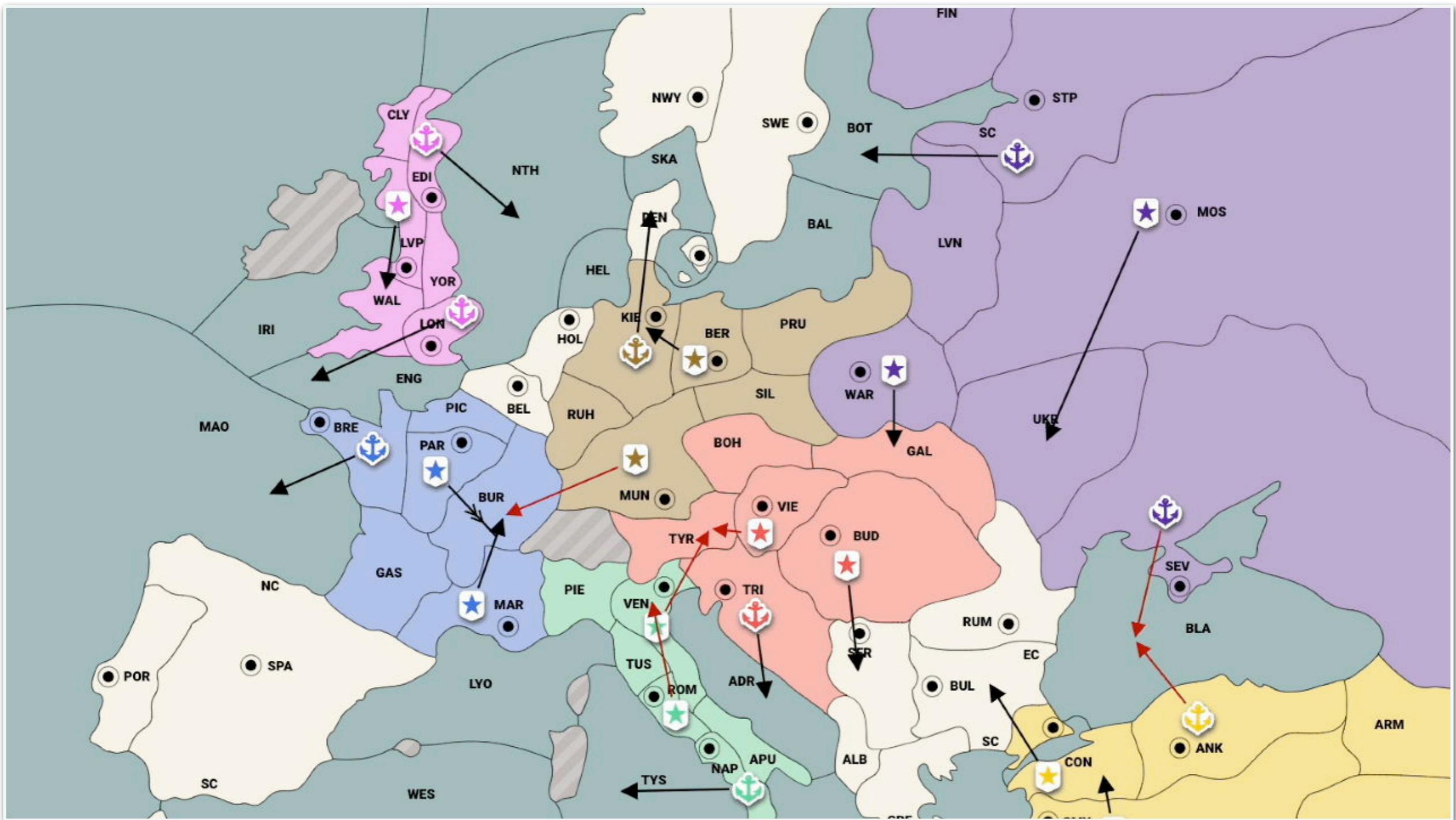
Quake 3 (2019)

? Est-ce qu'il existe encore des jeux où les humains l'emportent ?

- (1) Jeux demandant une riche interaction entre les joueurs (négociation, alliance, trahison, etc.)
- (2) Jeux ayant une composante plus abstraite et faisant appel à l'imagination
- (3) Jeux demandant une grande part d'exploration pour déceler les bonnes actions à exécuter
- (4) Jeux sujets aux considérations du monde réel (physique des mouvements, etc.)



Percée récente - Diplomacy



Diplomacy: jeu de plateau demandant des interactions très fortes avec les autres joueurs

Cicero: agent développé par Meta en 2022 et ayant des performances *supérieure au joueur moyen*

A vous de jouer !

A votre tour d'implémentez le meilleur agent pour Divercité !



I HAVE SOME ADVICES FOR YOU



Début: une stratégie minimax heuristique est le plus simple pour démarrer

- (1) Jouez et analysez vos parties pour identifier les états qui sont favorables
- (2) Analysez les choix faits par votre agent et vérifier leur cohérence
- (2) N'oubliez pas de considérer les cas limites

Important: définir une bonne heuristique est un point de conception crucial

Il est préférable d'aller le plus profond possible dans l'arbre (un seul niveau peut faire une grosse différence)

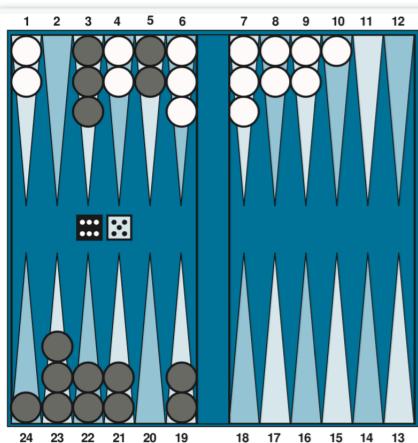
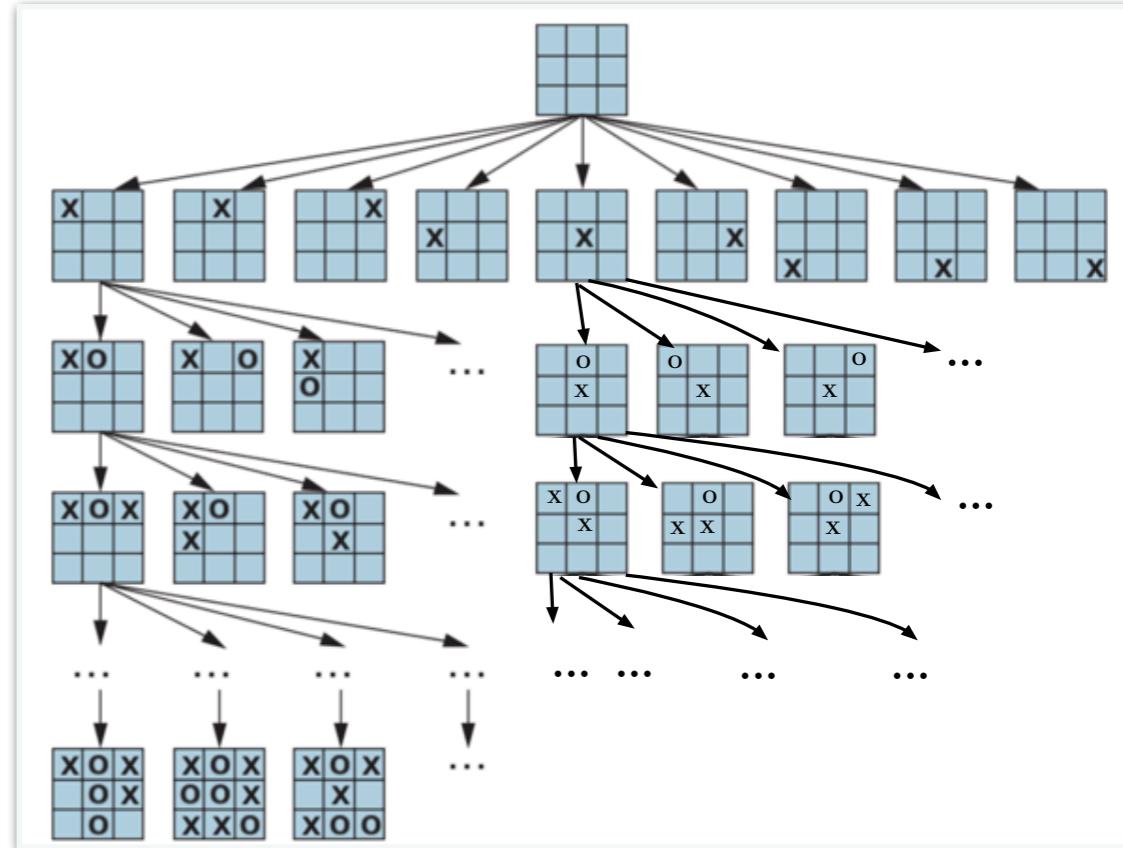
Utilisation d'un budget temps: une stratégie de type IDS est une bonne idée



Table des matières

Recherche adversarielle

- ✓ 1. Motivation de la recherche adversarielle
- ✓ 2. Catégorisation d'environnements compétitifs et des jeux
- ✓ 3. Stratégie de recherche minimax
- ✓ 4. Mécanisme du *alpha-beta pruning*
- ✓ 5. Conception de fonctions d'évaluation (heuristiques)
- ✓ 6. Mécanismes supplémentaires d'amélioration
- ✓ 7. Arbre de recherche de Monte-Carlo (*Monte-Carlo tree search*)
- ✓ 8. Extension à d'autres familles de jeux (plusieurs adversaires, présence d'aléatoire, etc.)
- ✓ 9. Exemples et historique d'agents dédiés aux jeux



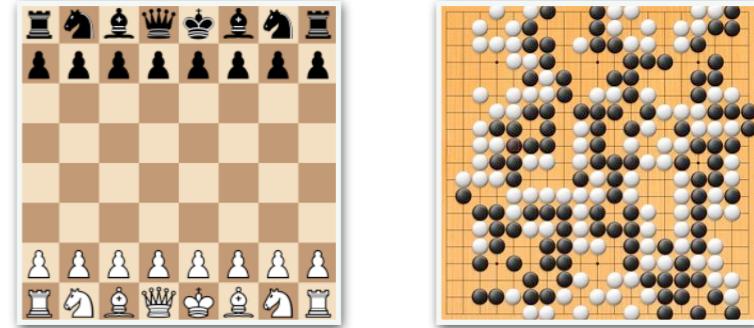
Synthèse des notions vues

Recherche adversarielle

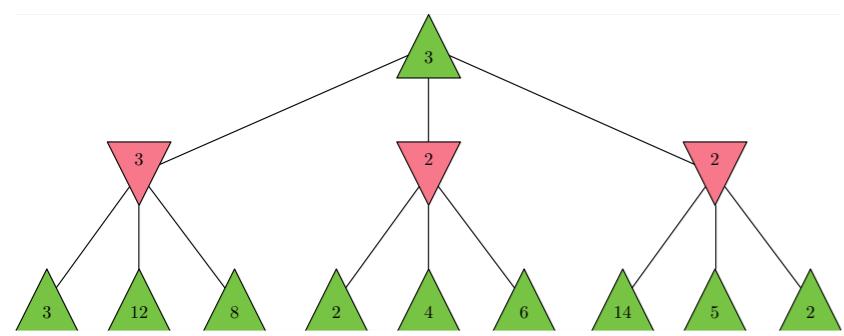
Contexte: l'agent évolue dans un environnement compétitif

Environnement: caractérisé par la présence d'un ou de plusieurs adversaires

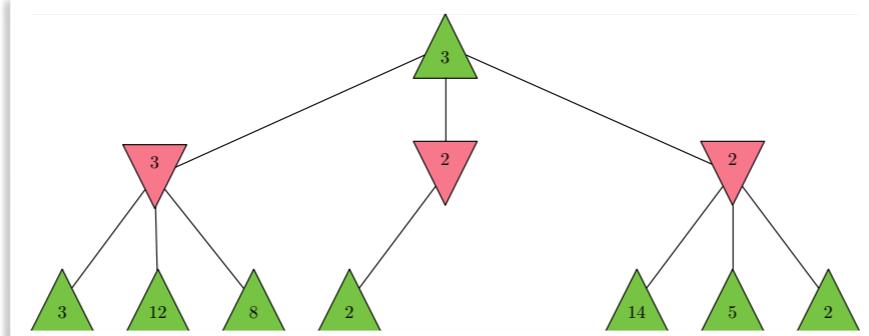
Cadre de référence: deux joueurs, déterministe, tour par tour, à information parfaite, et à somme nulle



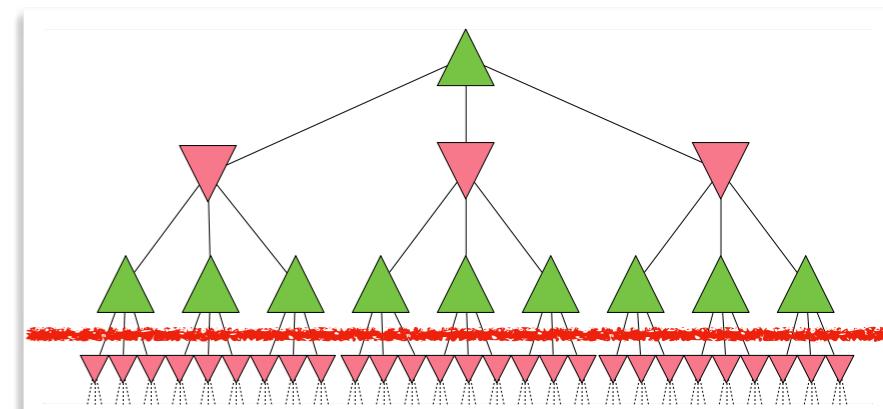
Stratégie de recherche Minimax



Minimax classique



Alpha-beta pruning



Avec des heuristiques

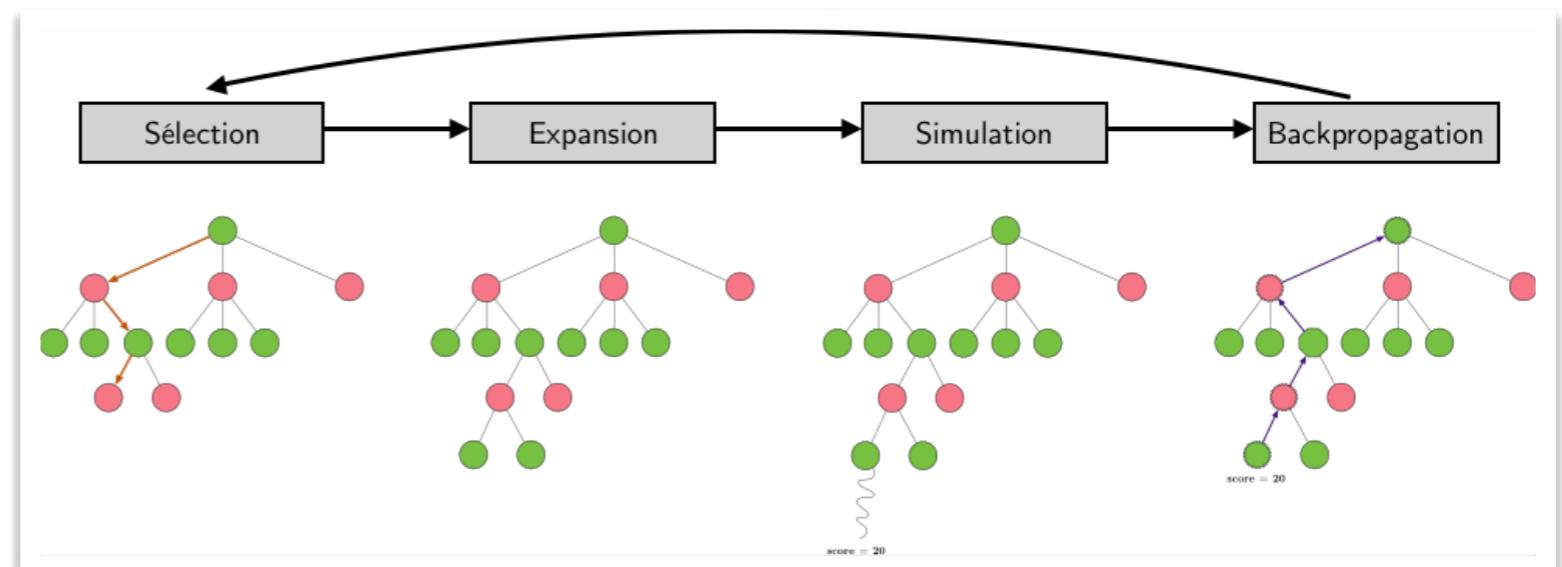
Autres améliorations: tables de transposition, méthodes hybrides, suppression de branches, etc.

Arbre de recherche MC

Basé sur des simulations de parties

Dépendance moindre à une heuristique

Méthode assez générique



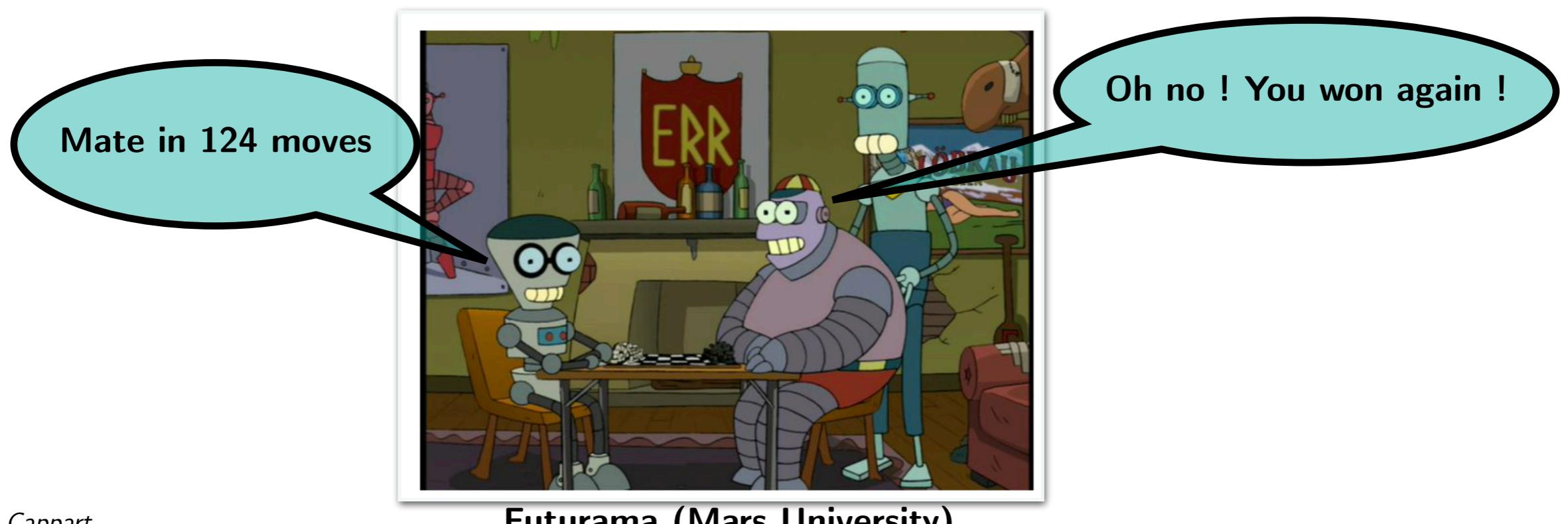
Exemples de questions d'examen

Théorie

1. Expliquer le fonctionnement et les propriétés d'une stratégie de recherche vue
2. Donner les avantages/inconvénients d'une stratégie de recherche par rapport à une autre
3. Expliquer les variations possibles de notre cadre de référence

Pratique

1. Modéliser une situation comme un problème de recherche adversarielle
2. Savoir appliquer correctement un algorithme de recherche sur une situation donnée
3. Savoir construire des heuristiques pertinentes





DALLE: A colored pencil drawing
by Cézanne of a robot playing chess