

IFT 615 – Intelligence Artificielle

Planification dans les jeux compétitifs

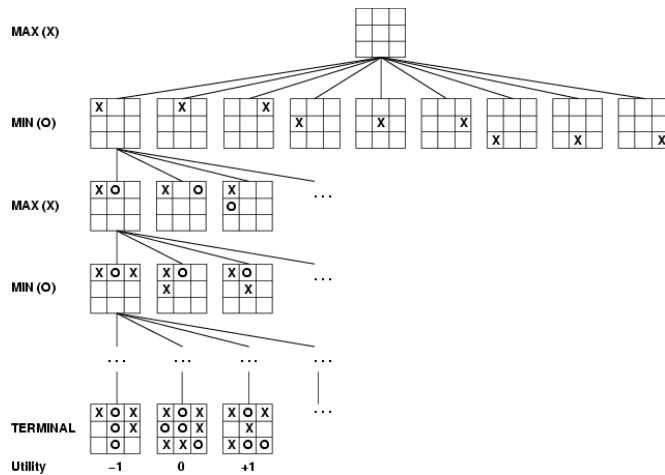
Professeur: Froduald Kabanza

Assistants: D'Jeff Nkashama

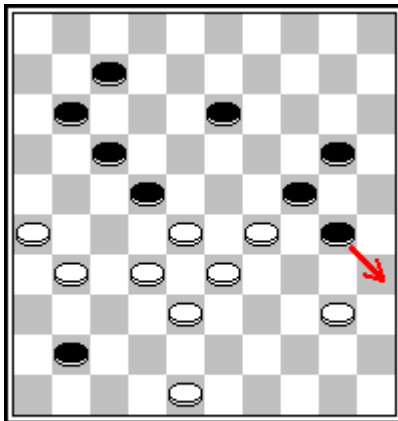
Objectifs

- Comprendre l'approche générale pour développer une IA pour un jeu à tour de rôle
- Comprendre et pouvoir appliquer l'algorithme *minimax*
- Comprendre et pouvoir appliquer l'algorithme d'élagage *alpha-bêta*
- Savoir traiter le cas de décisions imparfaites en temps réel (temps de réflexion limité)
- Comprendre et pouvoir appliquer l'algorithme *expectimax*
- Comprendre l'algorithme *Monte-Carlo-Tree-Search*

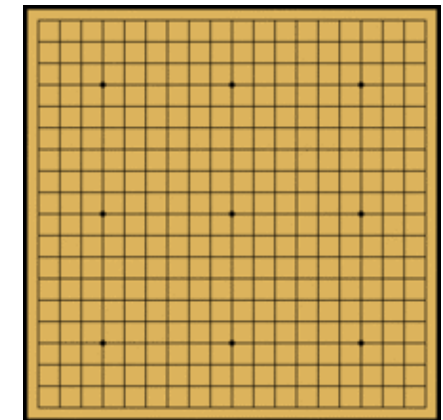
Exemples de jeu à tour de rôle



Source: chess.com



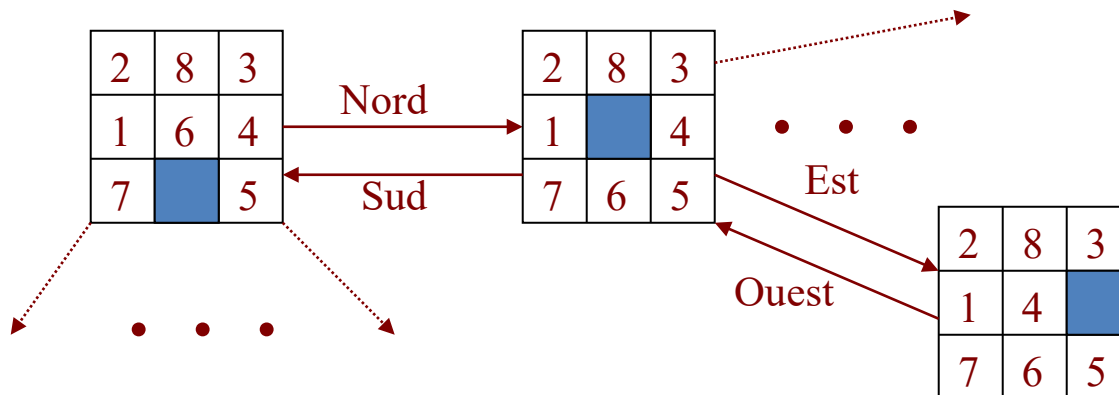
Source: Wikipedia



Source: Wikipedia

Rappel sur A*

- Notion d'état (configuration)
- État initial
- Fonction de transition (successeurs)
- Fonction de but (configuration finale)



2	8	3
1	6	4
7		5



1	2	3
8		4
7	6	5

Vers les jeux avec adversité ...

- Q : Est-il possible d'utiliser A^* pour des jeux entre deux adversaires ?
 - ◆ Q : Comment définir un état pour le jeu d'échecs ?
 - ◆ Q : Quelle est la fonction de but ?
 - ◆ Q : Quelle est la fonction de transition ?
- R : Non. Pas directement.
- Q : Quelle hypothèse est violée dans un jeu à deux adversaires?
- R : Dans les jeux, l'environnement est multi-agents. Le joueur adverse peut modifier l'environnement.
- Q : Comment peut-on résoudre ce problème ?
- R : C'est le sujet d'aujourd'hui !

Particularité des jeux avec adversaires

- Plusieurs acteurs qui modifient l'environnement (les configurations/états du jeu).
- Les coups des adversaires sont “imprévisibles”.
- Le temps de réaction à un coup de l'adversaire est limité.
- Les joueurs peuvent avoir une connaissance totale ou partielle de l'état du jeu.

Relation entre les joueurs

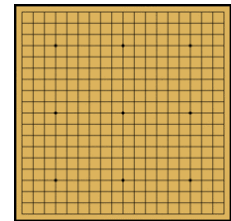
- De façon générale, la **théorie des jeux algorithmique** s'intéresse aux situations où les décisions d'un agent peuvent être influencées par d'autres agents – *décisions multi-agents*.
- Dans un jeu, des joueurs peuvent être :
 - ◆ **Coopératifs**
 - » ils veulent atteindre le même but
 - ◆ Des **adversaires** en compétition
 - » un gain pour les uns est une perte pour les autres
 - » cas particulier : les jeux à somme nulle (*zero-sum games*)
 - jeux d'échecs, de dame, tic-tac-toe, Go, etc.
 - ◆ **Mixte**
 - » il y a tout un spectre entre les jeux purement coopératifs et les jeux avec adversaires (ex. : alliances)

Hypothèses pour ce cours

- Dans cette leçon, nous aborderons les :
 - ◆ jeux à **deux adversaires**
 - ◆ jeux à **tour de rôle**
 - ◆ jeux à **somme nulle**
 - ◆ jeux avec états **complètement observés**
- Dans un premier temps, jeux **déterministes** (sans hasard ou incertitude)
- Dans un deuxième temps, mais de façon simple, nous allons explorer une généralisations à plusieurs joueurs et avec des actions aléatoires (par exemple, jeux dans lesquels on jette un dé pour choisir une action).



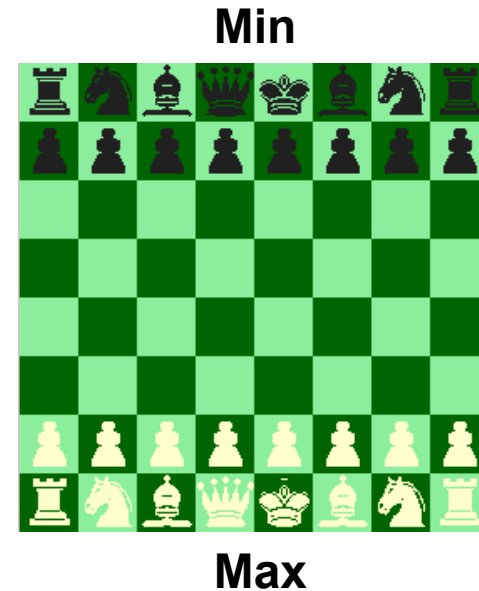
Source: chess.com



Source: Wikipedia

Jeux entre deux adversaires

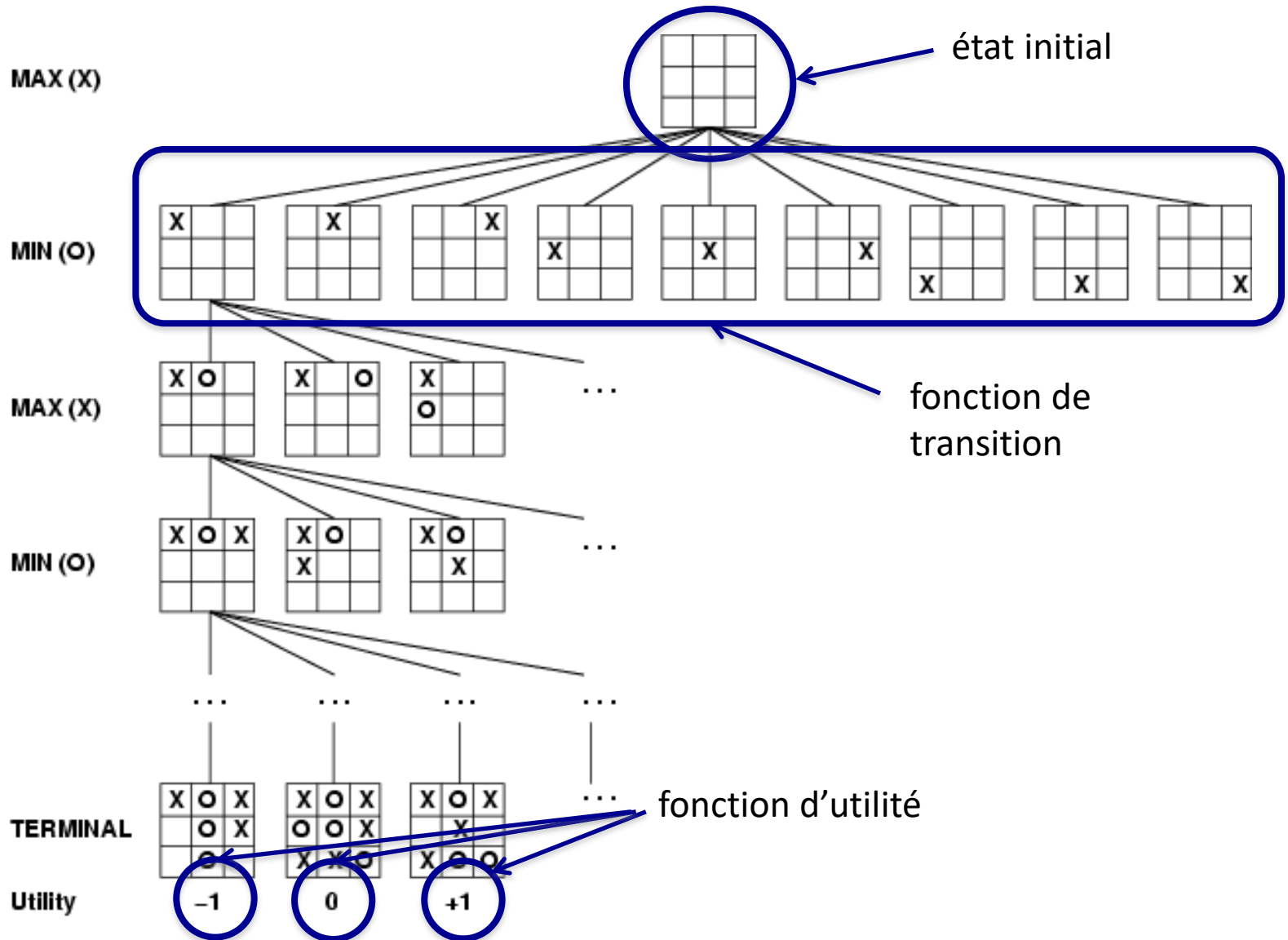
- Noms des joueurs : Max vs. Min
 - ◆ Max est le premier à jouer (notre joueur)
 - ◆ Min est son adversaire
- On va interpréter le résultat d'une partie comme la **distribution d'une récompense**
 - ◆ On peut voir cette récompense comme le résultat d'un pari
 - ◆ Min reçoit l'opposé de ce que Max reçoit



Arbre de recherche

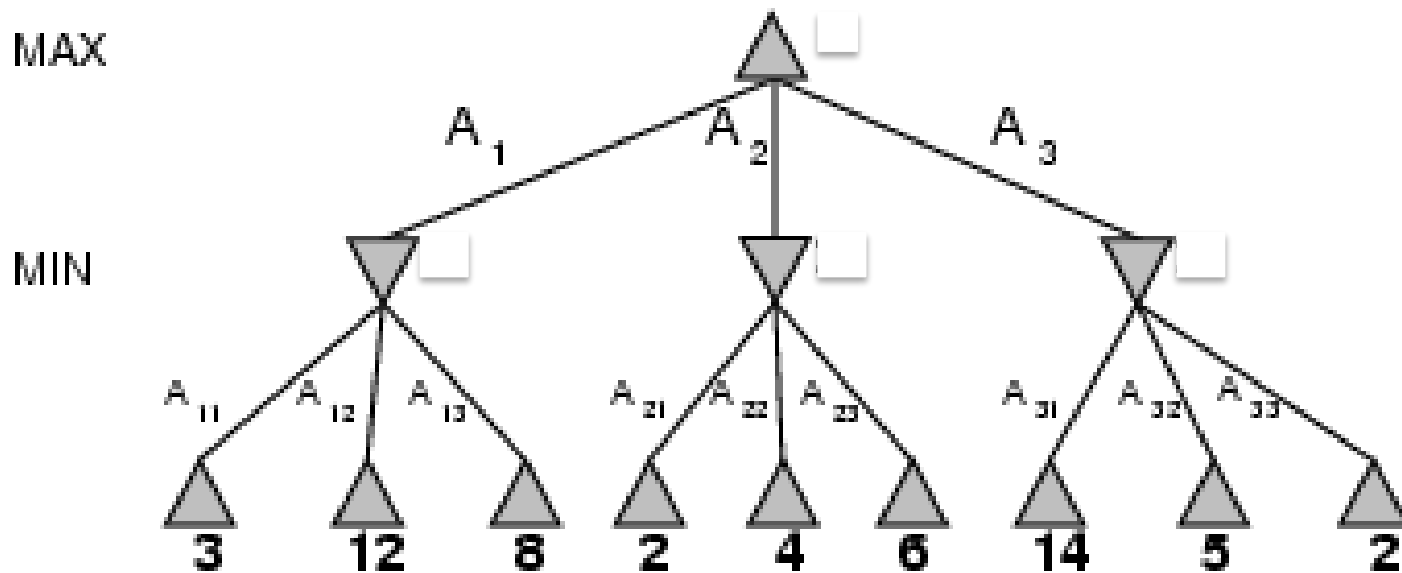
- Similaire à A^* et les processus de décision de Markov, on commence par déterminer la structure de notre espace de recherche
- Un problème de jeu peut être vu comme un problème de recherche dans un arbre :
 - ◆ Un **nœud (état) initial** : configuration initiale du jeu
 - ◆ Une **fonction de transition** :
 - » retournant un **ensemble de paires (action, noeud successeur)**
 - action possible (légale)
 - noeud (état) résultant de l'exécution de cette action
 - ◆ Un **test de terminaison**
 - » indique si le jeu est terminé
 - ◆ Une **fonction d'utilité** pour les états finaux (c'est la récompense reçue)

Arbre de recherche tic-tac-toe



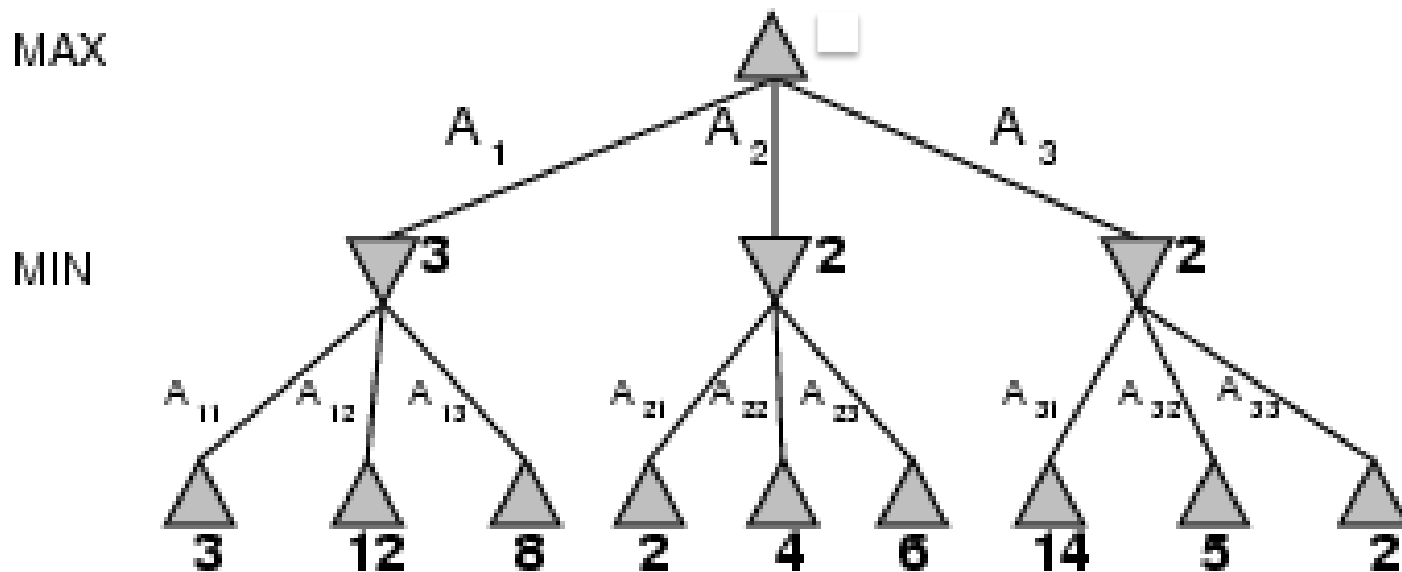
Algorithme *minimax*

- **Idée:** À chaque tour, choisir l'action menant à la plus grande *valeur minimax*.
 - ◆ Cela donne la meilleure action optimale (plus grand gain) contre un joueur optimal.
- Exemple simple:



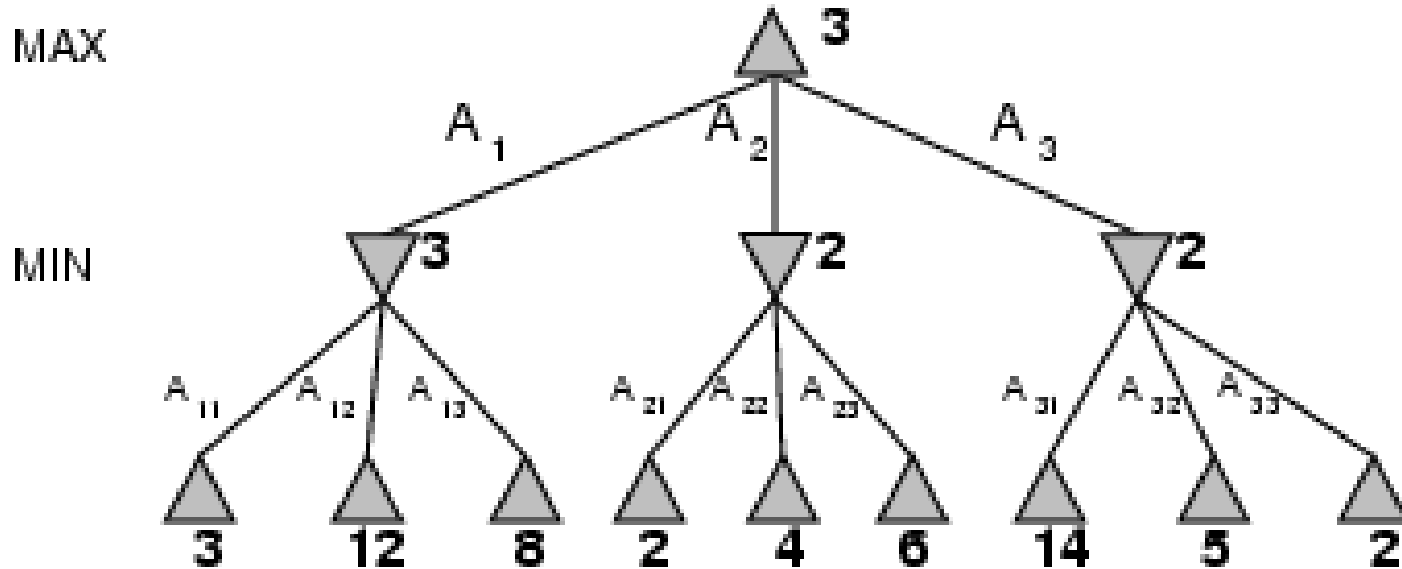
Algorithme *minimax*

- **Idée:** À chaque tour, choisir l'action menant à la plus grande *valeur minimax*.
 - ◆ Cela donne la meilleure action optimale (plus grand gain) contre un joueur optimal.
- Exemple simple:



Algorithme *minimax*

- **Idée:** À chaque tour, choisir l'action menant à la plus grande *valeur minimax*.
 - ◆ Cela donne la meilleure action optimale (plus grand gain) contre un joueur optimal.
- Exemple simple:



Algorithme *minimax*

- **Hypothèse:** MAX et MIN jouent optimalement.
- **Idée:** À chaque tour, choisir l'action menant à la plus grande *valeur minimax*.
 - ◆ Cela donne la meilleure action optimale (plus grand gain) contre un joueur optimal (rationnel).

MINIMAX-VALUE(s) =

UTILITY(s , MAX)

if IS-TERMINAL(s)

$\max_{a \in \text{Actions}(s)} \text{MINIMAX-VALUE}(\text{RESULT}(s, a))$

if TO-MOVE(s)=MAX

$\min_{a \in \text{Actions}(s)} \text{MINIMAX-VALUE}(\text{RESULT}(s, a))$

if TO-MOVE(s)=MIN

RESULT(s, a) est l'état résultant de l'exécution de a dans l'état s (modèle de transition)

- Ces équations donnent la programmation récursive des valeurs jusqu'à la racine de l'arbre.

Algorithme *minimax*

function MINIMAX-SEARCH(*game, state*) *returns an action*

player \leftarrow *game*.TO-MOVE(*state*)

value, move \leftarrow MAX-VALUE(*game, state*)

return *move*

function MAX-VALUE(*game, state*) *returns a (utility, move) pair*

if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state, player*), *null*

v $\leftarrow -\infty$

for each *a* **in** *game*.ACTIONS(*state*) **do**

v2, a2 \leftarrow MIN-VALUE(*game, game*.RESULT(*state, a*))

if *v2* > *v* **then**

v, move \leftarrow *v2, a*

return *v, move*

function MIN-VALUE(*game, state*) *returns a (utility, move) pair*

if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state, player*), *null*

v $\leftarrow +\infty$

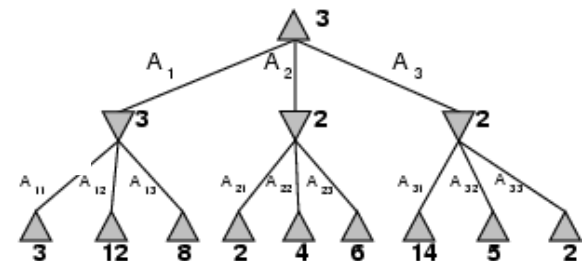
for each *a* **in** *game*.ACTIONS(*state*) **do**

v2, a2 \leftarrow MAX-VALUE(*game, game*.RESULT(*state, a*))

if *v2* < *v* **then**

v, move \leftarrow *v2, a*

return *v, move*



Propriétés de *minimax*

- Complet?
 - ◆ Oui (si l'arbre est fini)
- Optimal?
 - ◆ Oui (contre un adversaire qui joue optimalement)
- Complexité en temps?
 - ◆ $O(b^m)$:
 - » b : le nombre maximum d'actions/coups légaux à chaque étape
 - » m : nombre maximum de coup dans un jeu (profondeur maximale de l'arbre).
- Complexité en espace?
 - ◆ $O(bm)$, parce que l'algorithme effectue une recherche en profondeur.
- Pour le jeu d'échec: $b \approx 35$ et $m \approx 100$ pour un jeu « raisonnable »
 - ◆ Il n'est pas réaliste d'espérer trouver une solution exacte en temps réel.

Comment accélérer la recherche

Combinaison de deux approches:

1. Élagage alpha-beta (*alpha-beta pruning*)

- ◆ **idée** : éviter des chemins dans l'arbre qui sont explorés inutilement
- ◆ maintient l'exactitude de la solution

2. Fonction d'évaluation heuristique

- ◆ **idée** : faire une recherche la plus profonde possible en fonction du temps à notre disposition et tenter de prédire le résultat de la partie si on n'arrive pas à la fin
- ◆ introduit une approximation (c.-à-d., ne garantit pas l'optimalité)
- ◆ Beaucoup plus puissante que l'élagage alpha-beta

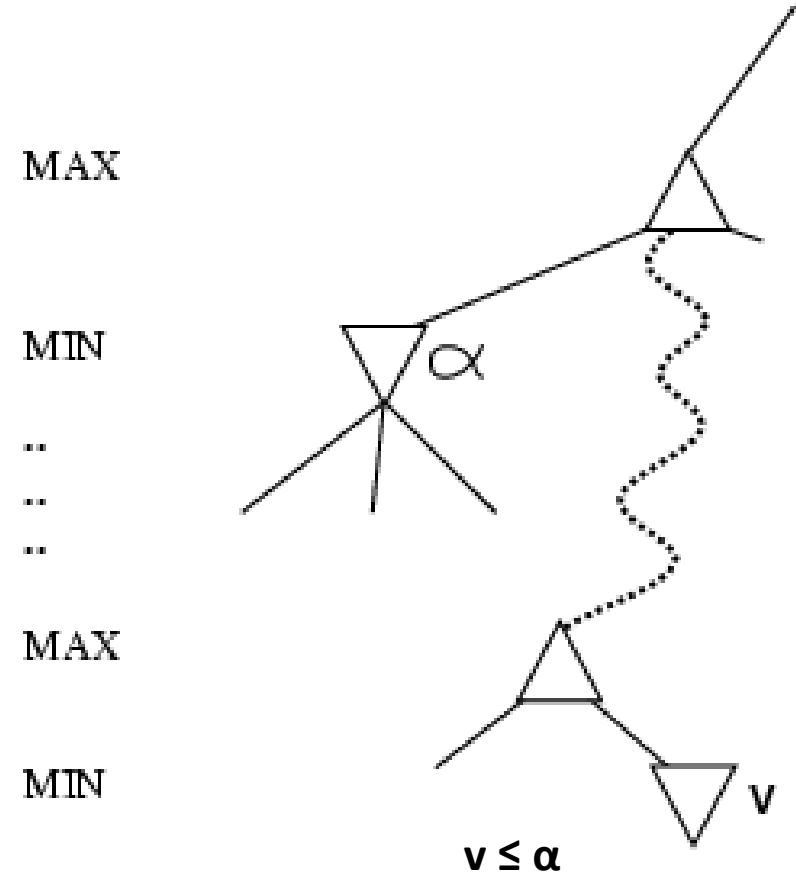
Alpha-beta pruning

- L'élagage alpha-beta tire son nom des paramètres suivant décrivant les bornes des valeurs d'utilité enregistrée durant le parcours.
 - ◆ α est la valeur du meilleur choix pour Max (c.-à-d., plus grande valeur) trouvée jusqu'ici:
 - ◆ β est la valeur du meilleur choix pour Min (c.-à-d., plus petite valeur) trouvée jusqu'ici.

Alpha-beta Pruning

Condition pour couper dans un nœud Min

- Sachant que α est la valeur du meilleur choix pour Max (c.-à-d., plus grande valeur) trouvée jusqu'ici:
 - ◆ Si on est **dans un nœud Min** est que sa valeur v **devient inférieure α** (donc « pire que α » du point de vue de Max), il faut arrêter la recherche (couper la branche).



Alpha-beta Pruning

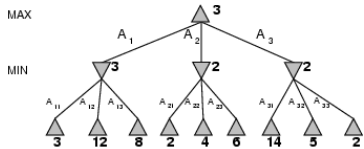
Condition pour couper dans un nœud Max

- Sachant que β est la valeur du meilleur choix pour Min (c.-à-d., plus petite valeur) trouvée jusqu'ici:
 - ◆ Si on est **dans un nœud Max** et que sa valeur **devient supérieur à β** (donc « pire que β » du point de vue de Min), il faut arrêter la recherche (couper la branche).

Exemple d'Alpha-beta pruning

Faire une recherche en profondeur jusqu'à la première feuille

MAX



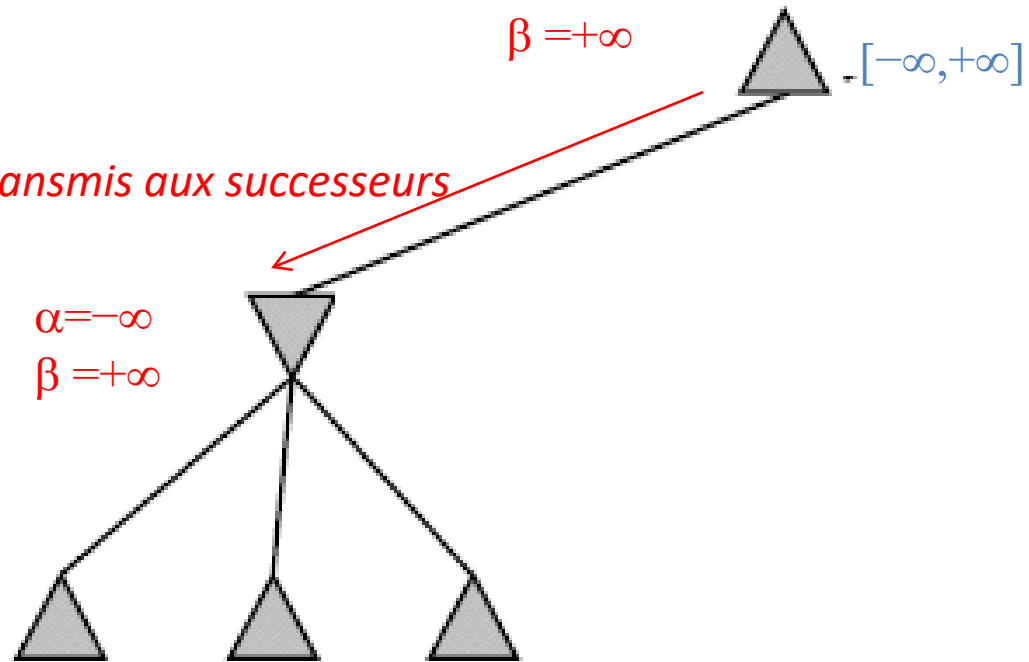
MIN

Valeur initial de α, β

$$\alpha = -\infty$$

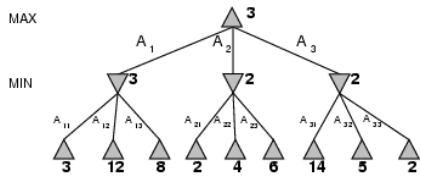
$$\beta = +\infty$$

α, β , transmis aux successeurs



Entre croches $[,]$: Intervalle des valeurs possibles pour le nœud visité.

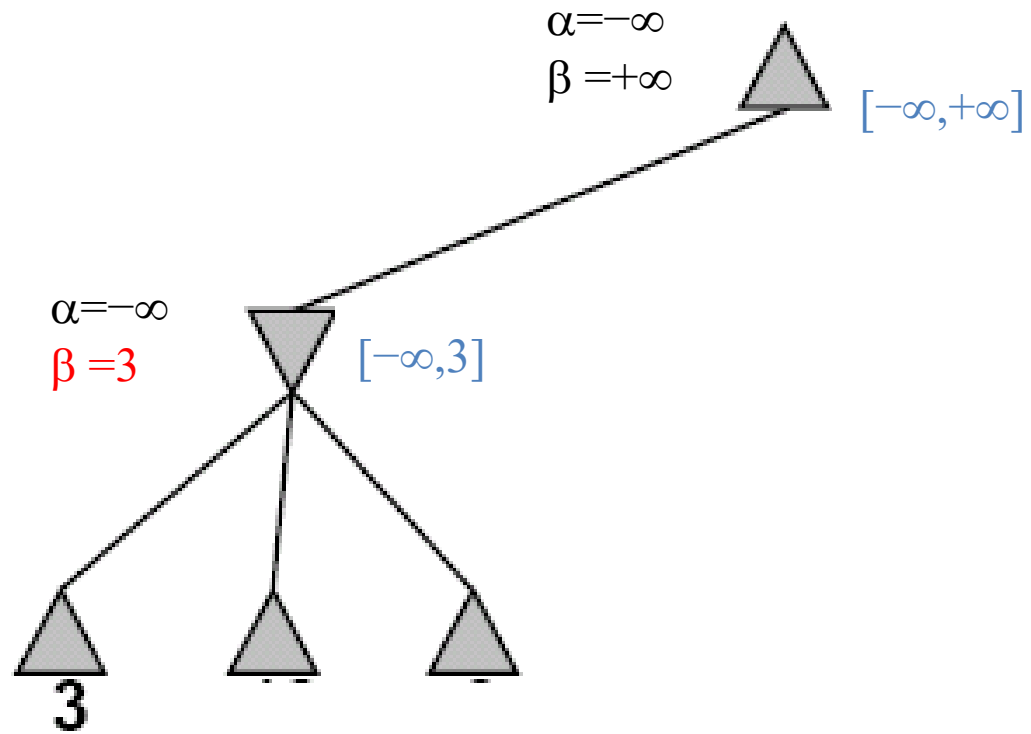
Exemple d'Alpha-beta pruning



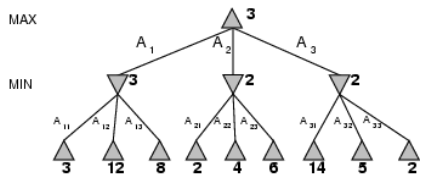
MAX

MIN

MIN met à jour β , basé sur les successeurs



Exemple d'Alpha-beta pruning

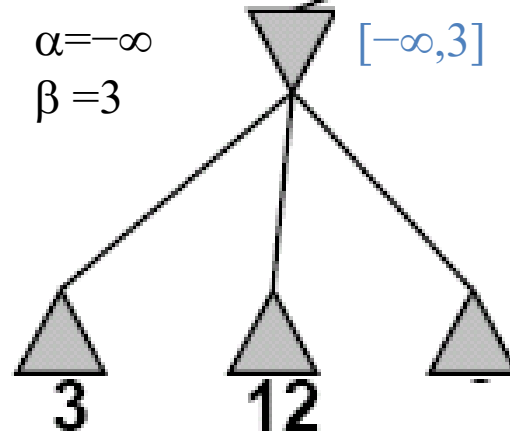
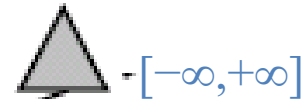


MAX

MIN

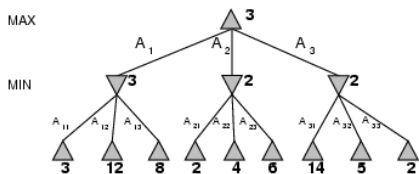
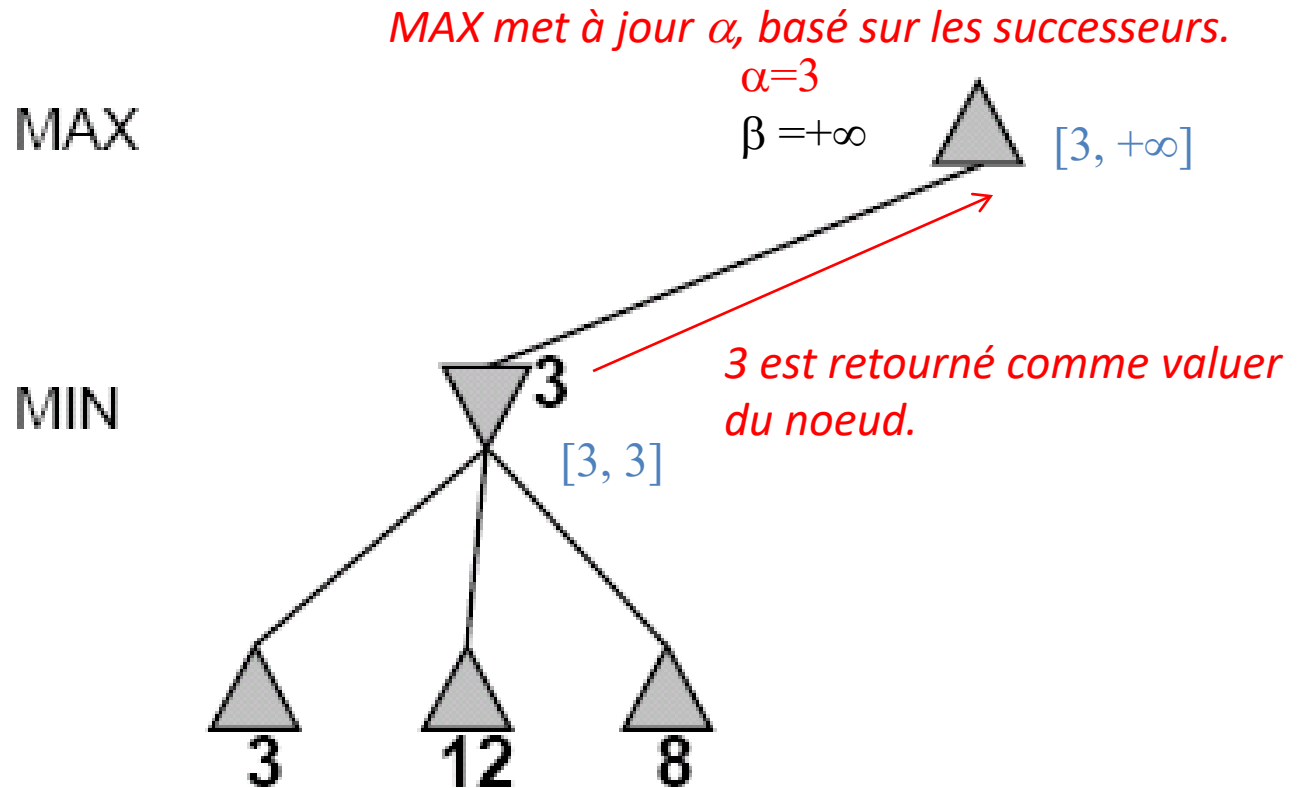
$$\alpha = -\infty$$

$$\beta = +\infty$$



*MIN met à jour β , basé sur les successeurs.
Aucun changement.*

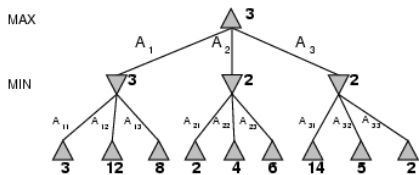
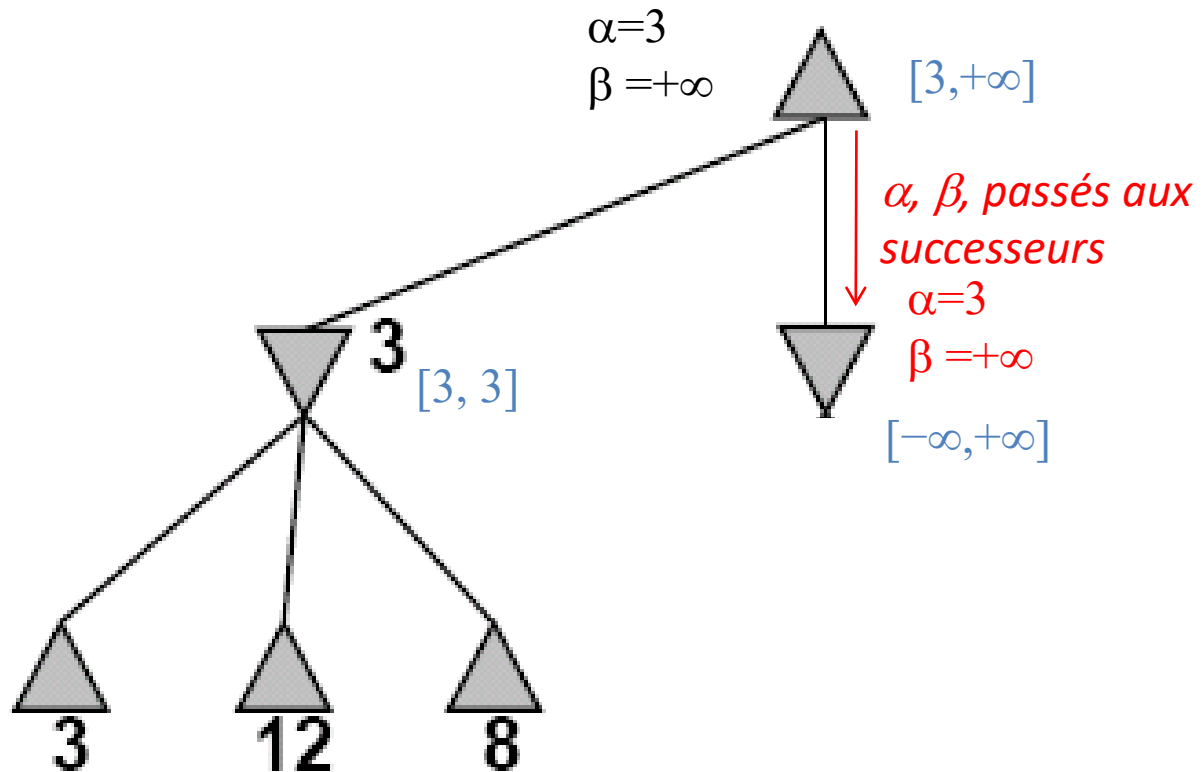
Exemple d'Alpha-beta pruning



Exemple d'Alpha-beta pruning

MAX

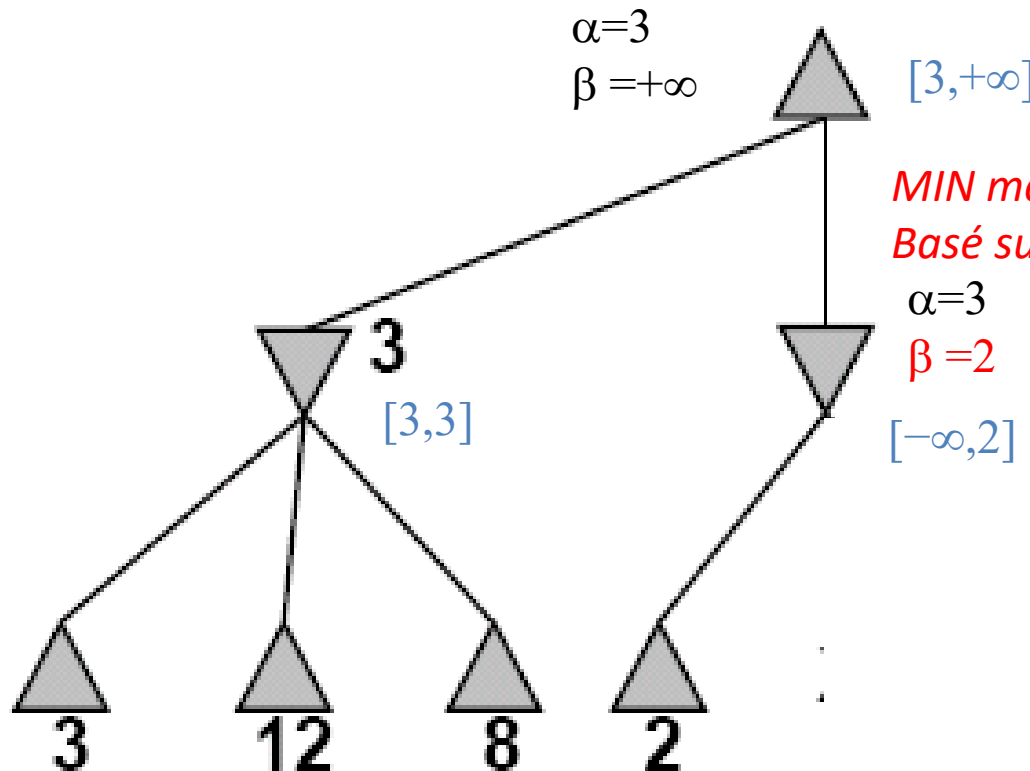
MIN



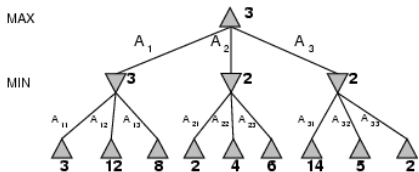
Exemple d'Alpha-beta pruning

MAX

MIN



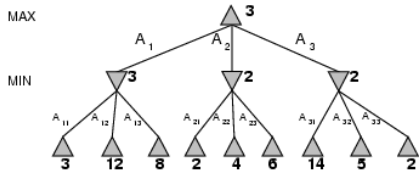
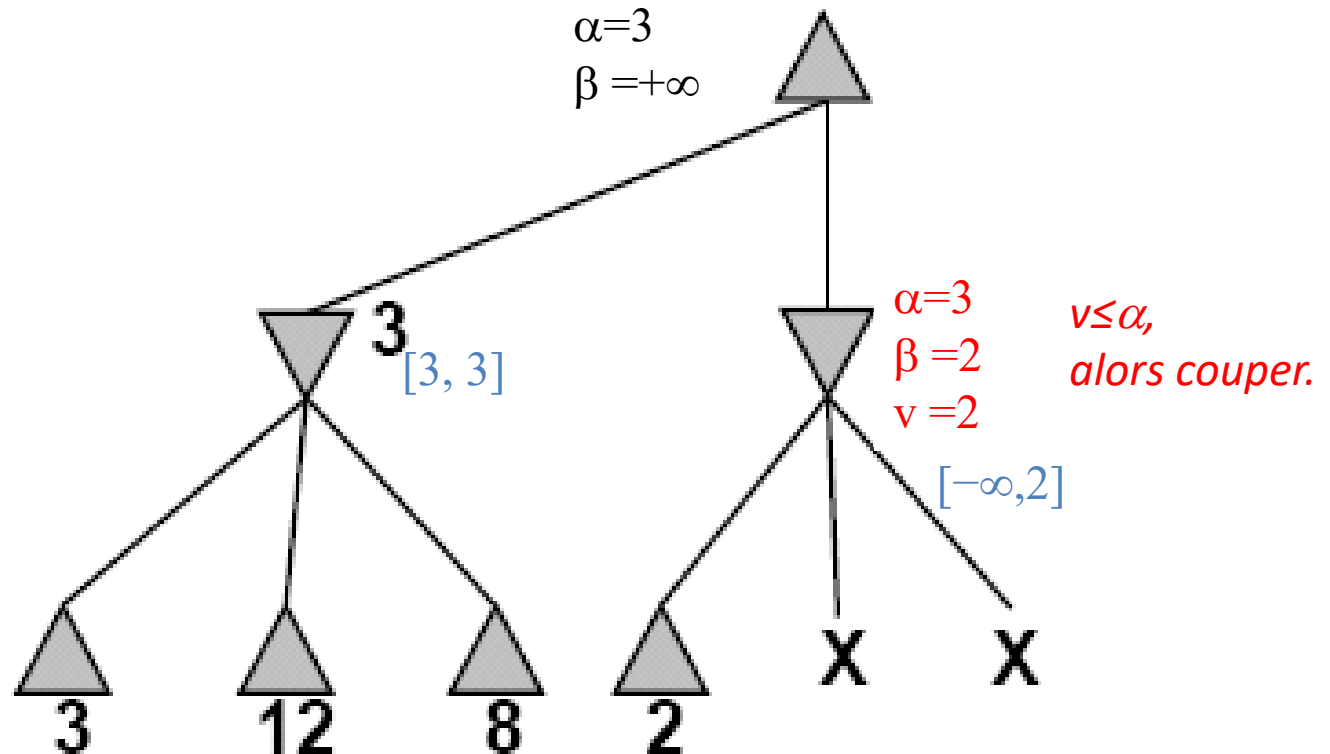
*MIN met à jour β ,
Basé sur les successeurs.*



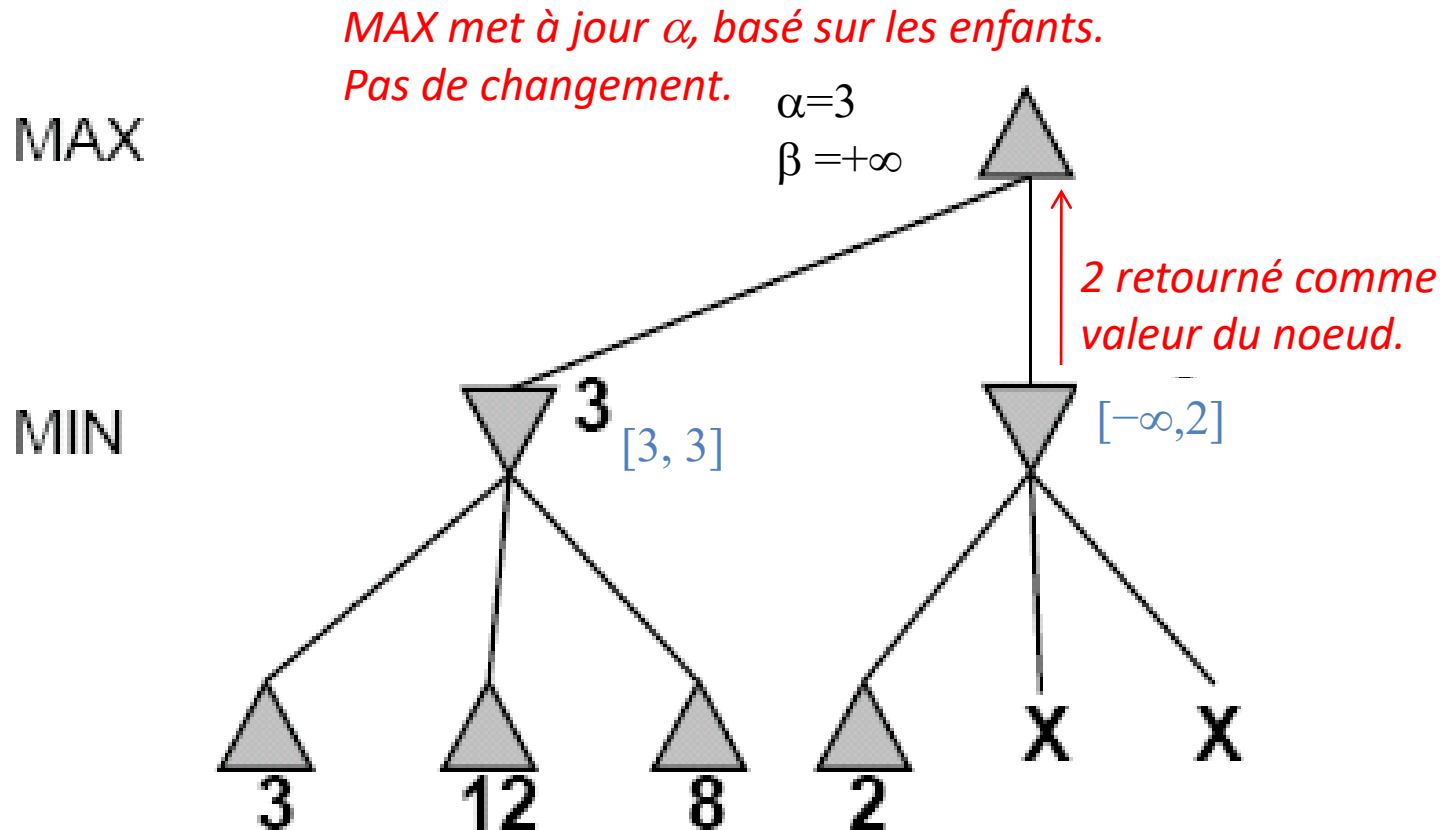
Exemple d'Alpha-beta pruning

MAX

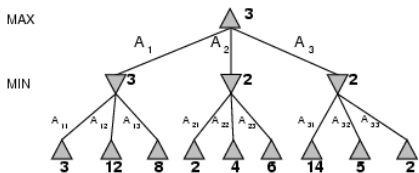
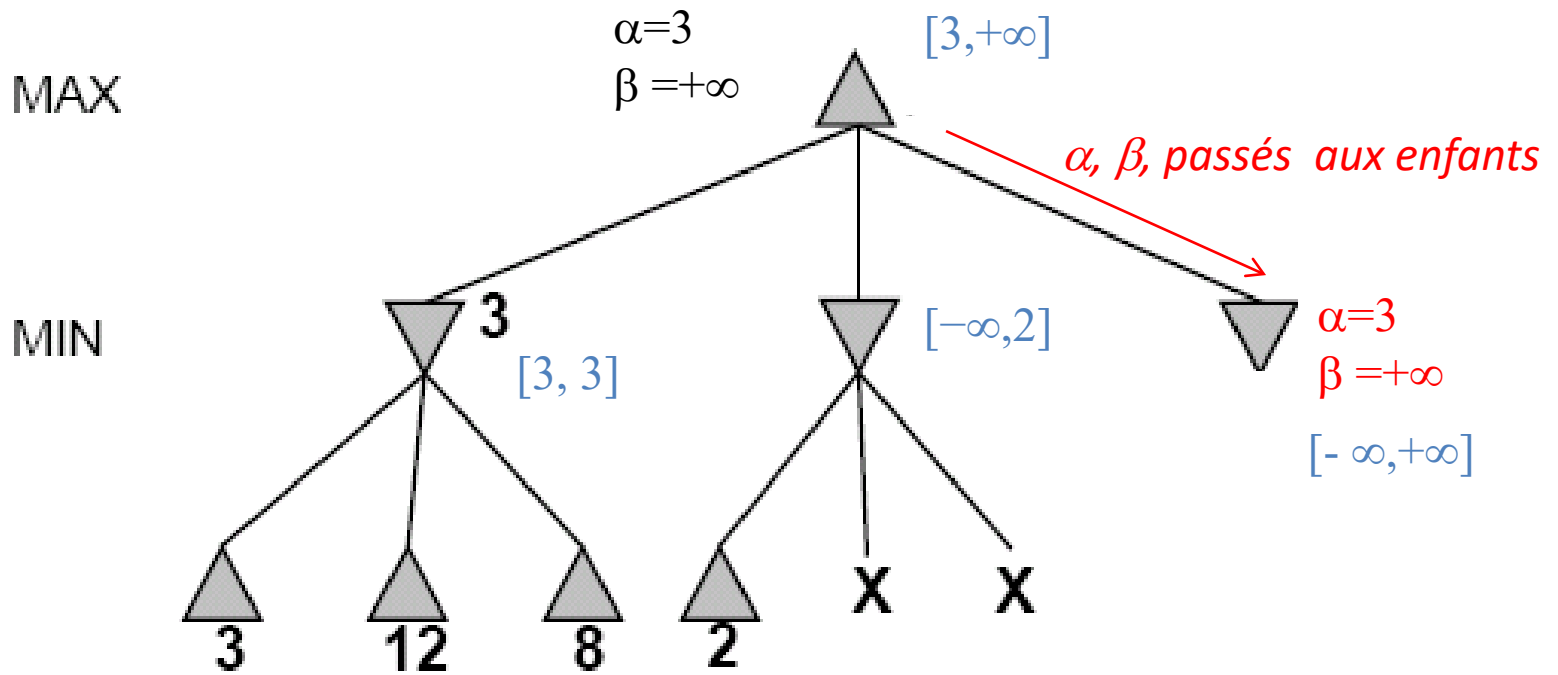
MIN



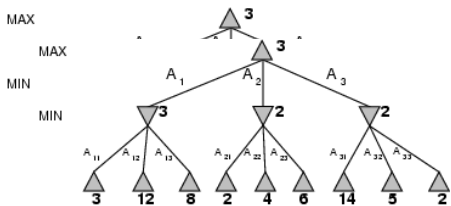
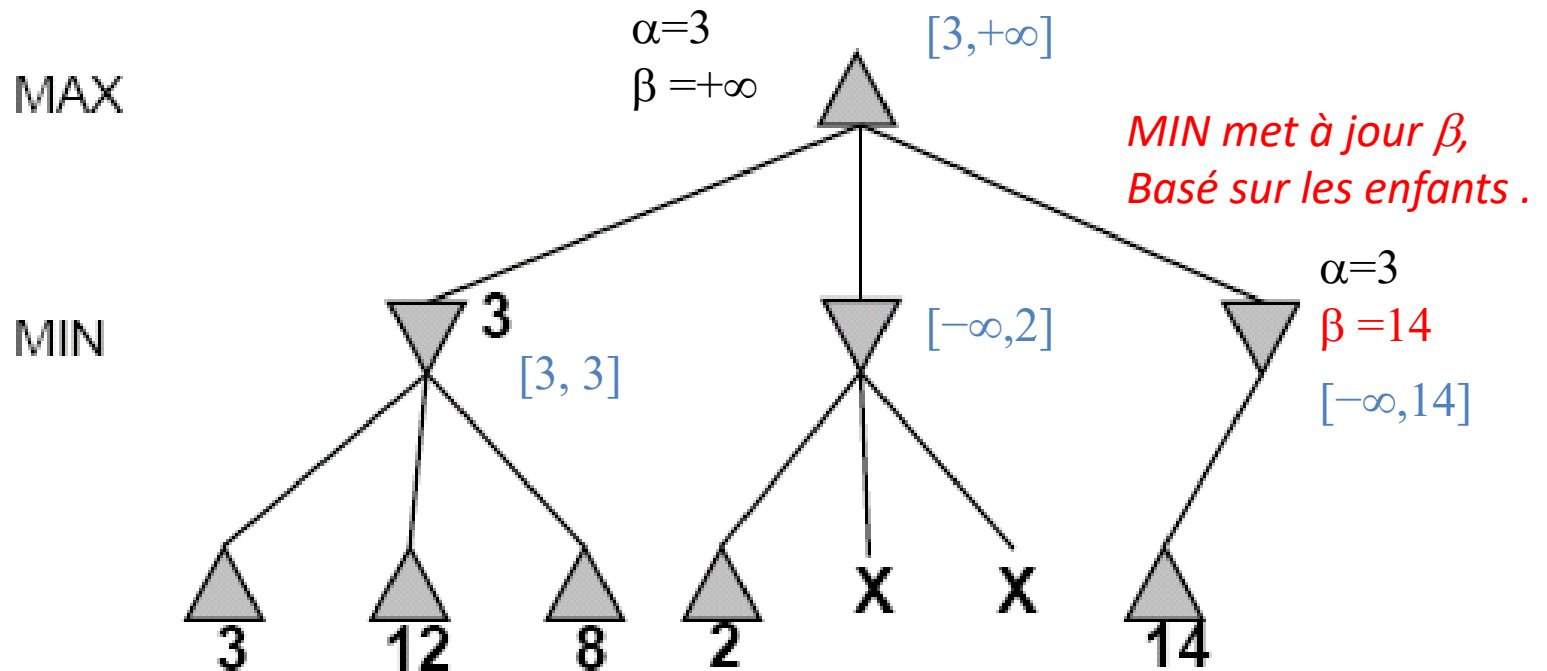
Exemple d'Alpha-beta pruning



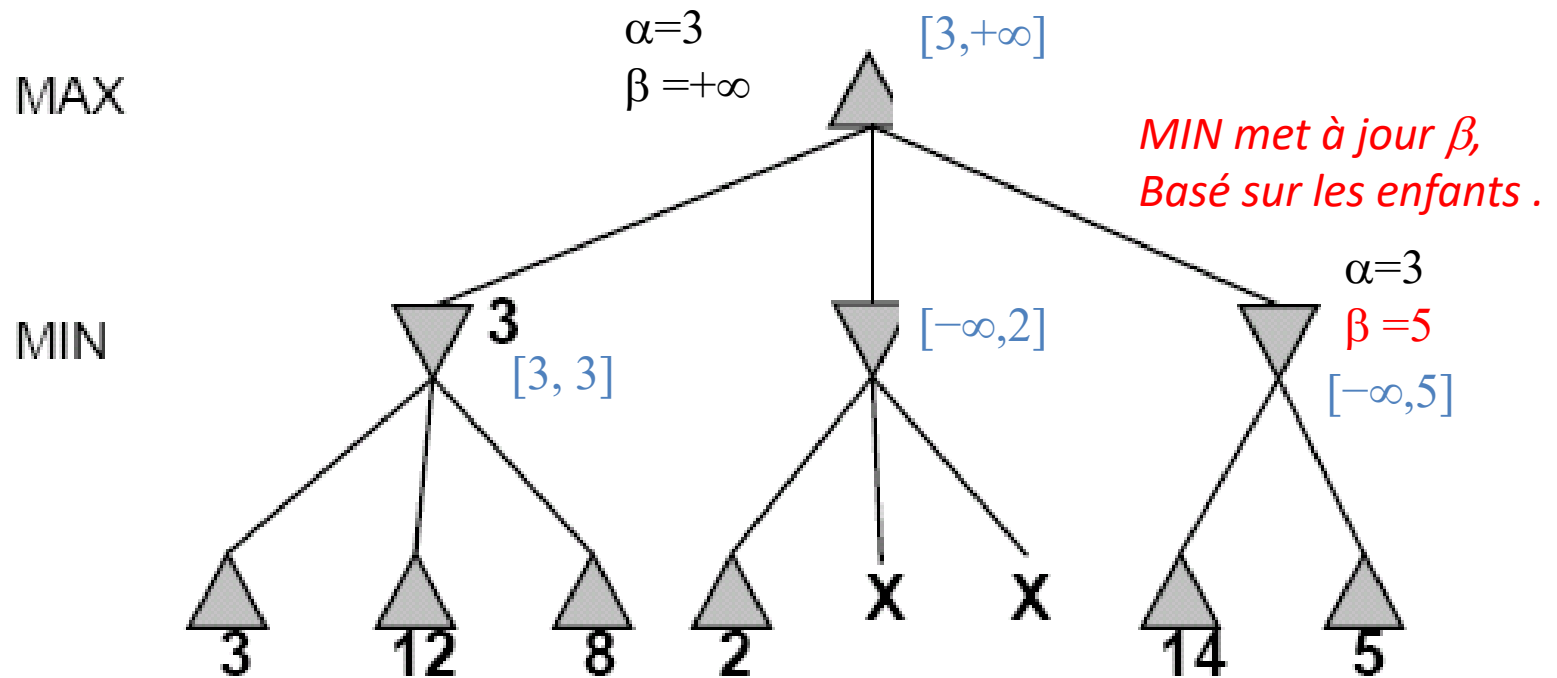
Exemple d'Alpha-beta pruning



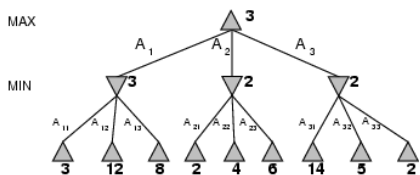
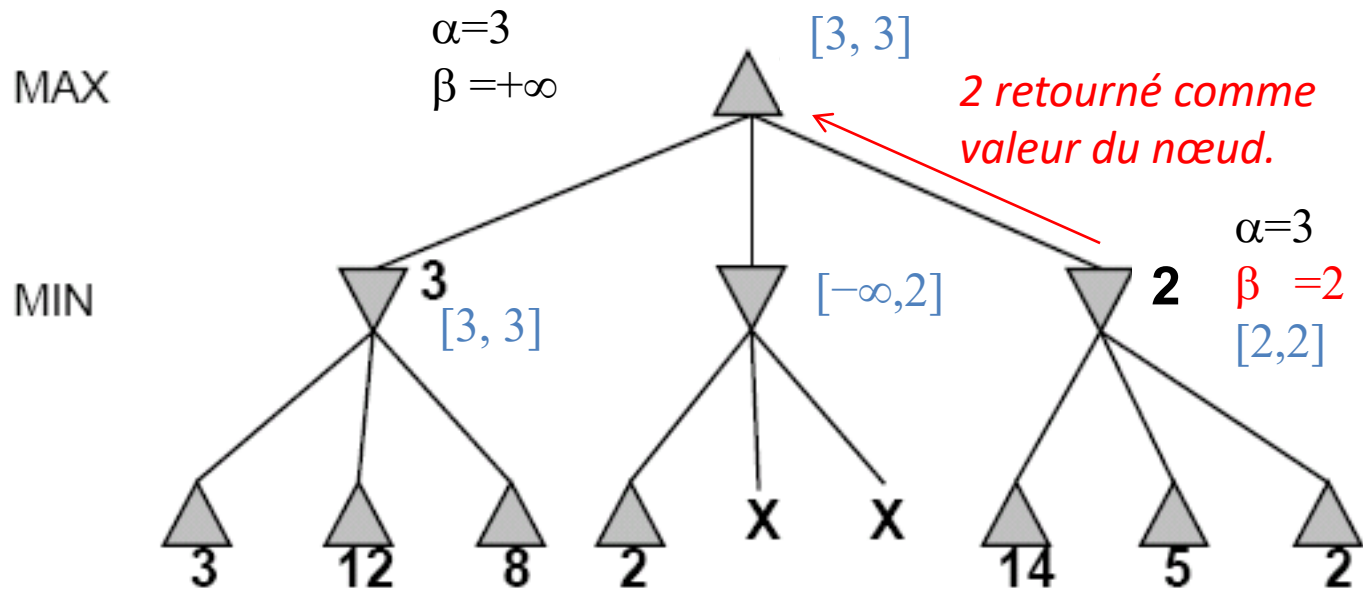
Exemple d'Alpha-beta pruning



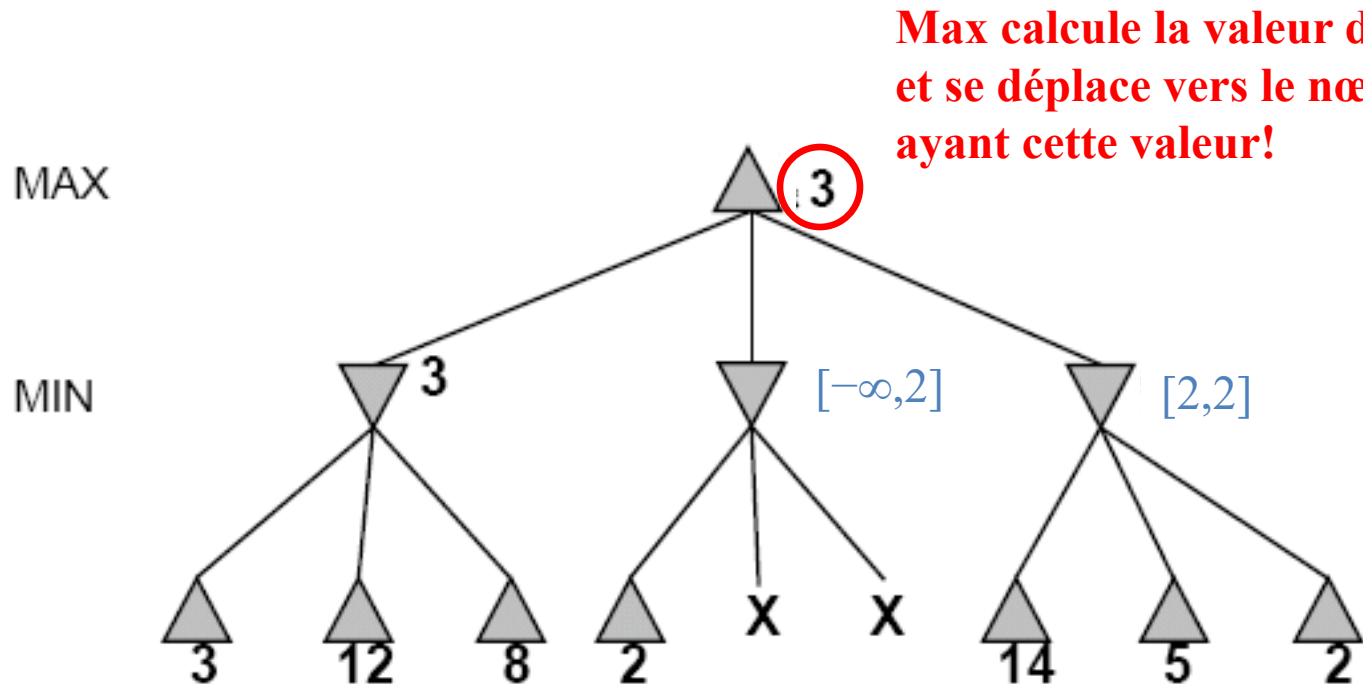
Exemple d'Alpha-beta pruning



Exemple d'Alpha-beta pruning



Exemple d'Alpha-beta pruning



Max calcule la valeur du nœud,
et se déplace vers le nœud
ayant cette valeur!

Algorithme *alpha-beta pruning*

function ALPHA-BETA-SEARCH(*game, state*) **returns** an action

player \leftarrow *game*.TO-MOVE(*state*)

value, move \leftarrow MAX-VALUE(*game, state, $-\infty, +\infty$*)

return *move*

function MAX-VALUE(*game, state, α, β*) **returns** a (*utility, move*) pair

if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state, player*), *null*

v $\leftarrow -\infty$

for each *a* **in** *game*.ACTIONS(*state*) **do**

v2, a2 \leftarrow MIN-VALUE(*game, game*.RESULT(*state, a*), *α, β*)

if *v2* > *v* **then**

v, move \leftarrow *v2, a*

$\alpha \leftarrow$ MAX(α, v)

if *v* $\geq \beta$ **then return** *v, move*

return *v, move*

function MIN-VALUE(*game, state, α, β*) **returns** a (*utility, move*) pair

if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state, player*), *null*

v $\leftarrow +\infty$

for each *a* **in** *game*.ACTIONS(*state*) **do**

v2, a2 \leftarrow MAX-VALUE(*game, game*.RESULT(*state, a*), *α, β*)

if *v2* < *v* **then**

v, move \leftarrow *v2, a*

$\beta \leftarrow$ MIN(β, v)

if *v* $\leq \alpha$ **then return** *v, move*

return *v, move*

Negamax – Version élégante de α - β pruning

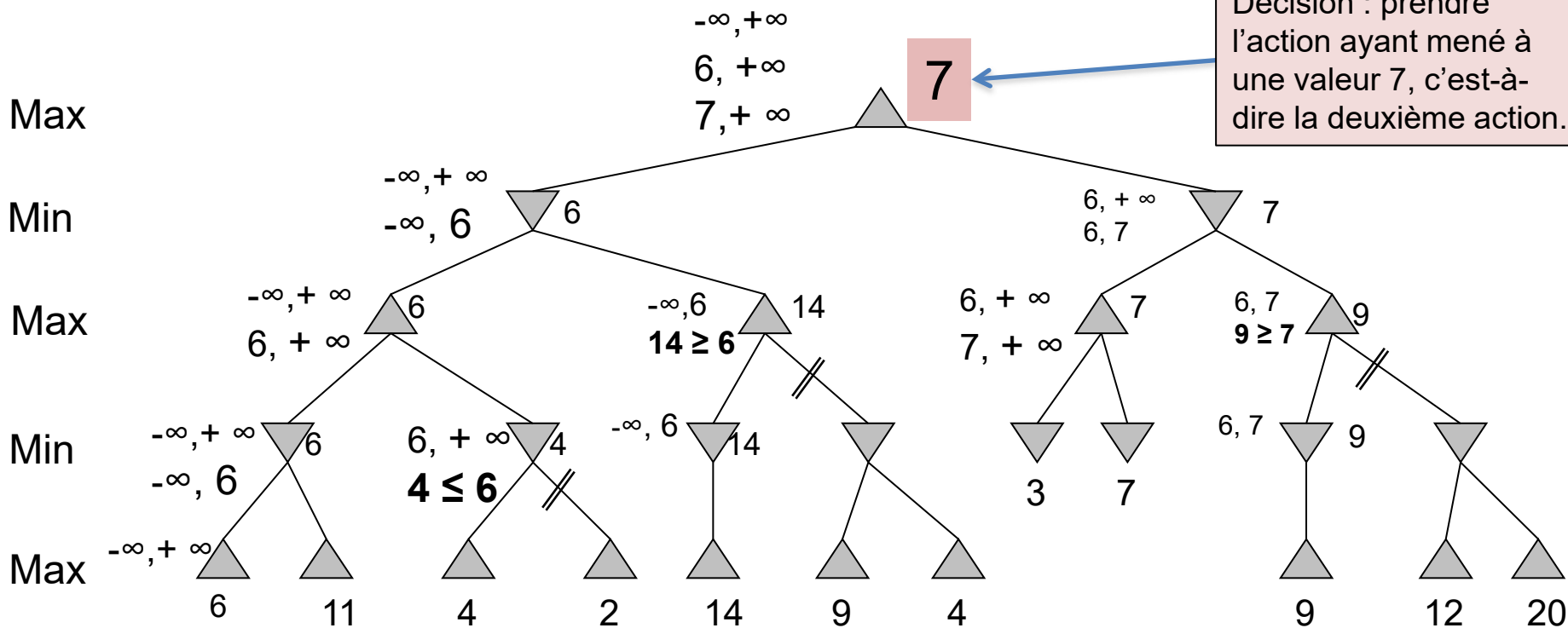
<http://en.wikipedia.org/wiki/Negamax>

```
1. fonction negamax(state, depth,  $\alpha$ ,  $\beta$ , player)
2.   if TerminalState(state) or depth = 0 then
3.     return player * Utility(state)
4.   else
5.     foreach child in Successors(state)
6.       bestVal  $\leftarrow$  - negamax(state, depth-1, - $\beta$ , -  $\alpha$ , -player)
7.       // les instructions 7 à 10 implémentent  $\alpha$ - $\beta$  pruning
8.       if bestVal  $\geq$   $\beta$ 
9.         return bestVal
10.      if bestVal  $\geq$   $\alpha$ 
11.         $\alpha \leftarrow$  bestVal
12.    return bestVal
```

Appel initial : negamax(initialState, depth, $-\infty$, $+\infty$, 1)





Signification de la variable player : 1 (max), -1 (min).

Autre exemple



Décision : prendre l'action ayant mené à une valeur 7, c'est-à-dire la deuxième action.

Légende de l'animation

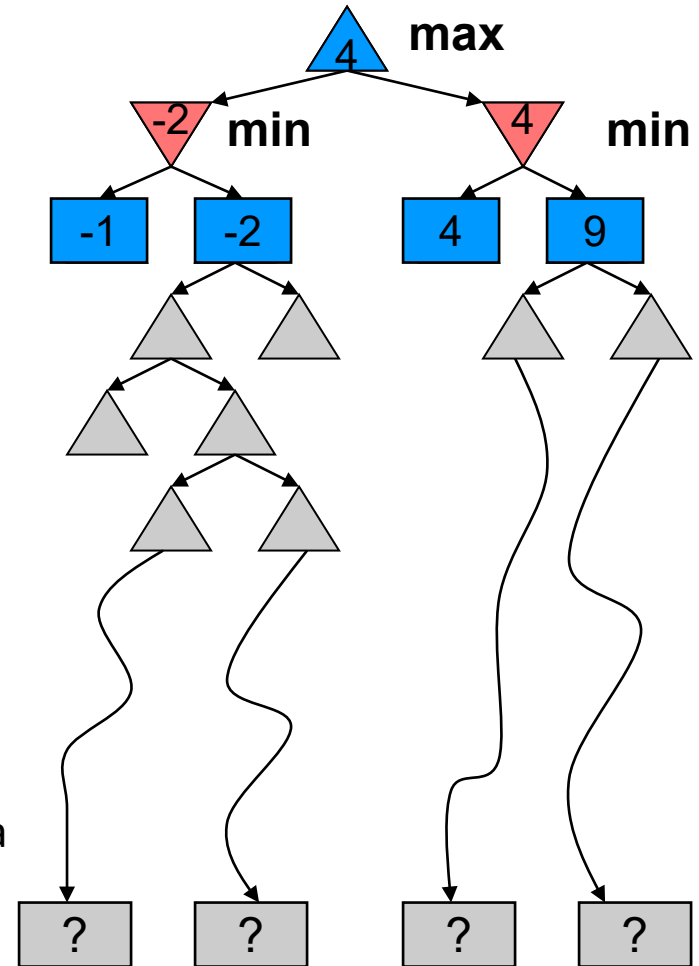
-  Nœud de l'arbre pas encore visité
-  Nœud en cours de visite (sur pile de récursivité)
-  Nœud visité
-  Arc élagué (pruning)

Propriétés de *alpha-beta pruning*

- L'élagage n'affecte pas le résultat final de *minimax*.
- Dans le pire des cas, *alpha-beta pruning* ne fait aucun élagage; il examine b^m nœuds terminaux comme l'algorithme *minimax*:
 - » b : le nombre maximum d'actions/coups légaux à chaque étape
 - » m : nombre maximum de coup dans un jeu (profondeur maximale de l'arbre).
- Un bon ordonnancement des actions à chaque nœud améliore l'efficacité.
 - ◆ Dans le meilleur des cas (ordonnancement parfait), la complexité en temps est de $O(b^{m/2})$
 - » On peut faire une recherche deux fois moins profondément comparé à *minimax*!

Fonction d'évaluation

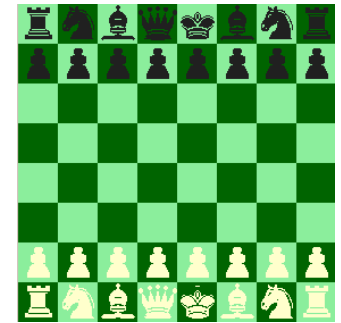
- En général, des décisions imparfaites doivent être prises en temps réel :
 - ◆ Pas le temps d'explorer tout l'arbre de jeu
- Approche standard :
 - ◆ couper la recherche :
 - » par exemple, limiter la profondeur de l'arbre
 - » voir le livre pour d'autres idées
 - ◆ fonction d'évaluation heuristique
 - » estimation de l'utilité qui aurait été obtenue en faisant une recherche complète
 - » on peut voir ça comme une estimation de la « chance » qu'une configuration mènera à une victoire
 - ◆ La solution optimale n'est plus garantie



Exemple de fonction d'évaluation

- Pour le jeu d'échec, une fonction d'évaluation typique est une somme (linéaire) pondérée de “métriques” estimant la qualité de la configuration:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$



- Par exemple:
 - ◆ w_i = poids du pion,
 - $f_i(s)$ = (nombre d'occurrence d'un type de pion d'un joueur) – (nombre d'occurrence du même type de pion de l'opposant),
 - ◆ etc

Exemple de fonction d'évaluation

- Pour le *tic-tac-toe*, supposons que Max joue avec les X.

$Eval(s) =$

(nombre de ligne, colonnes et diagonales disponibles pour Max) - (nombre de ligne, colonnes et diagonales disponibles pour Min)

	X	O

$$Eval(s) = 6 - 4 = 2$$

O	X	X
	O	

$$Eval(s) = 4 - 3 = 1$$

Résumé

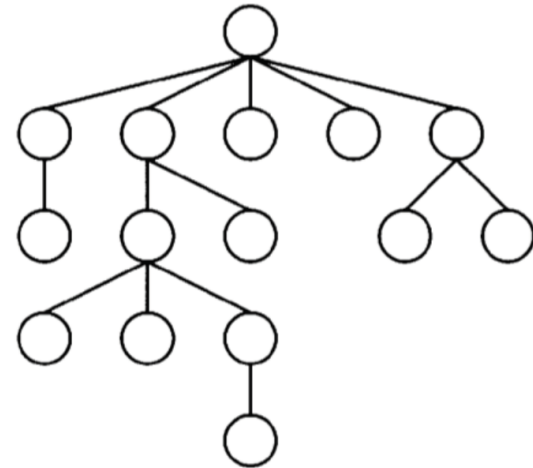
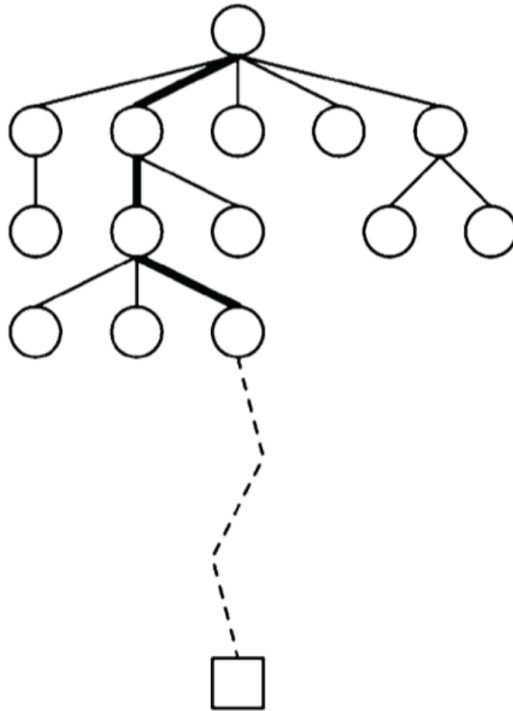
- La recherche sur les jeux révèle des aspects fondamentaux applicables à d'autres domaines
- La perfection est inatteignable dans les jeux : il faut approximer
- Alpha-bêta a la même valeur pour la racine de l'arbre de jeu que minimax
- Dans le pire des cas, il se comporte comme minimax (explore tous les nœuds)
- Dans le meilleur cas, il peut résoudre un problème de profondeur 2 fois plus grande dans le même temps que minimax

Plusieurs applications aux jeux mais aussi des limites

- Plusieurs algorithmes pour les jeux à tour de rôle sont basés sur alpha-beta pruning.
 - ◆ Stockfish est un algorithme pour les jeux à tour de rôle, basé sur alpha-beta pruning et des heuristiques.
 - ◆ Stockfish était un des meilleurs algorithmes pour les jeux d'échec avant la publication d'AlphaZero en 2017.
- Alpha-beta pruning a des limites, notamment pour le jeu de go:
 - ◆ Le jeu de Go a un facteur de branchement de 361 (vs 35 pour le jeu d'échec)
 - ◆ La fonction d'évaluation est beaucoup plus difficile à concevoir heuristiquement pour le jeu de Go

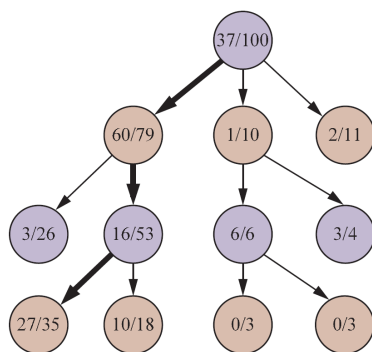
Principe de base de MCTS

- Le principe de Monte-Carlo Tree-Search (MCTS) est:
 - ◆ Parcourir (construire) l'arbre de jeu par **échantillonnage aléatoire**.
 - ◆ **Simuler** une partie complète pour évaluer la fonction d'utilité.

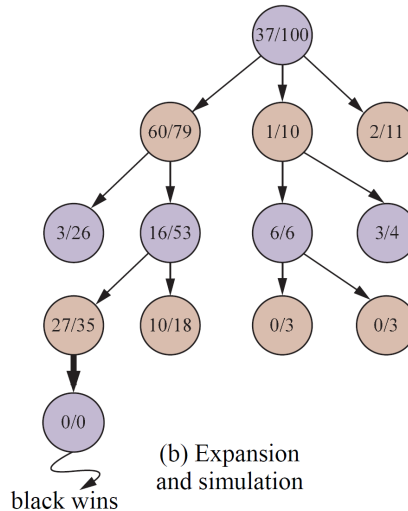


Exemple de MCTS

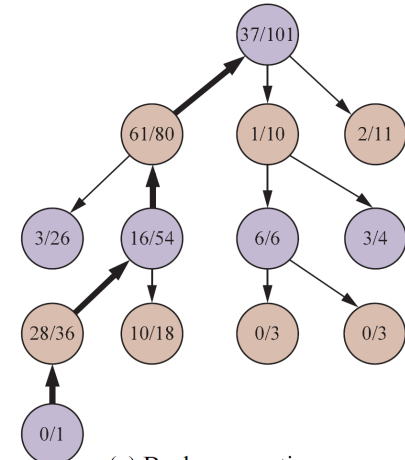
- Chaque état est étiqueté par son **utilité**: *nombre de simulations gagnées par le joueur qui fait l'action (move) / nombre total de simulations*
- **Sélection**: Commençant par la racine de l'arbre, en suivant une politique qui balance **exploration vs exploitation**, descendre à un nœud non encore complètement expansé (il existe une action non encore choisie)
- **Expansion**: Agrandir l'arbre au nœud sélectionné, en choisissant une action non encore choisie, et en construisant un successeur correspondant.
- **Simulation**: Simuler une partie complète à partir de l'enfant.
- **Rétropropagation**: Mettre à jour l'utilité des nœuds
- **À la fin, choisir l'action avec le plus grand nombre de simulations**



(a) Selection



(b) Expansion and simulation



(c) Backpropagation

Algorithme de MCTS

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*

tree \leftarrow NODE(*state*)

while IS-TIME-REMAINING() **do**

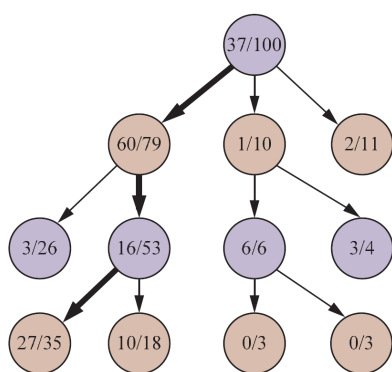
leaf \leftarrow SELECT(*tree*)

child \leftarrow EXPAND(*leaf*)

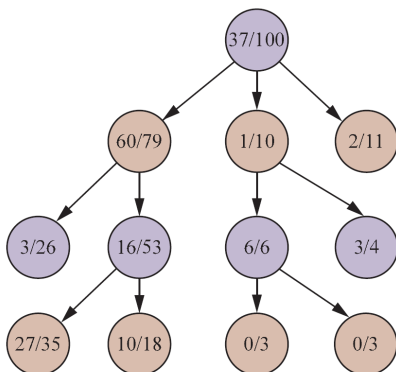
result \leftarrow SIMULATE(*child*)

BACK-PROPAGATE(*result*, *child*)

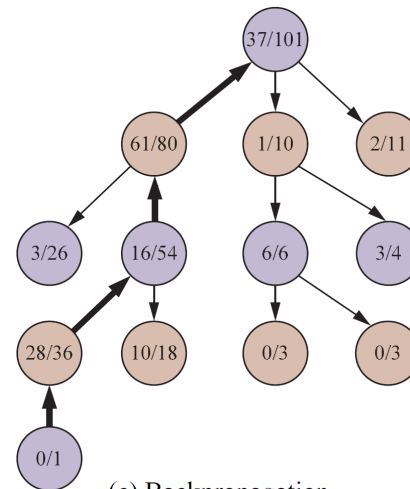
return the move in ACTIONS(*state*) whose node has highest number of playouts



(a) Selection



(b) Expansion and simulation
black wins



(c) Backpropagation

Algorithme UCT

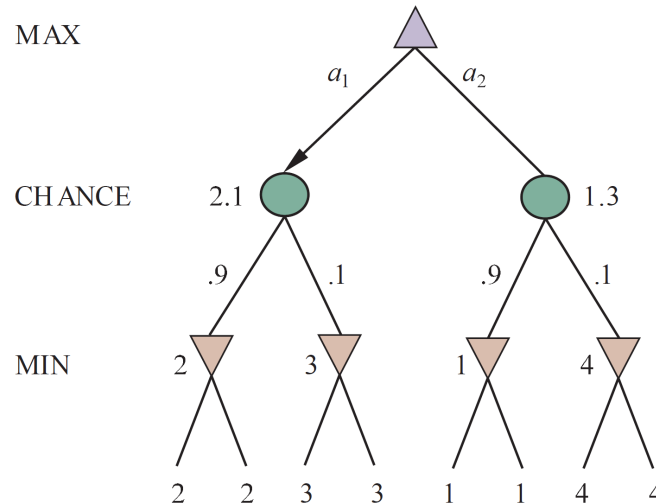
- UCT (*Upper Confidence Bound on Trees*) est une version de MCTS qui utilise l'algorithme UCB1 (*Upper Confidence Bound 1*) pour implémenter la politique de sélection des nœuds permettant de résoudre le dilemme exploration vs exploitation

- $$UCB1(s) = \frac{U(s)}{N(s)} + c * \sqrt{\frac{\log N(\text{Parent}(s))}{N(s)}}$$

- ◆ $U(s)$: Utilité de l'état s
 - ◆ $N(s)$: nombre de simulations ayant passé par l'état s
 - ◆ $\text{Parent}(s)$: le parent de s
 - ◆ C : un hyper paramètre qui balance entre l'exploitation vs l'exploration. Théoriquement $\sqrt{2}$, mais en pratique choisi empiriquement.
- On pourrait appliquer UCB1 sur le Q-value en utilisant $Q(s,a)$ et $N(s,a)$ au lieu de $U(s)$ et $N(s)$



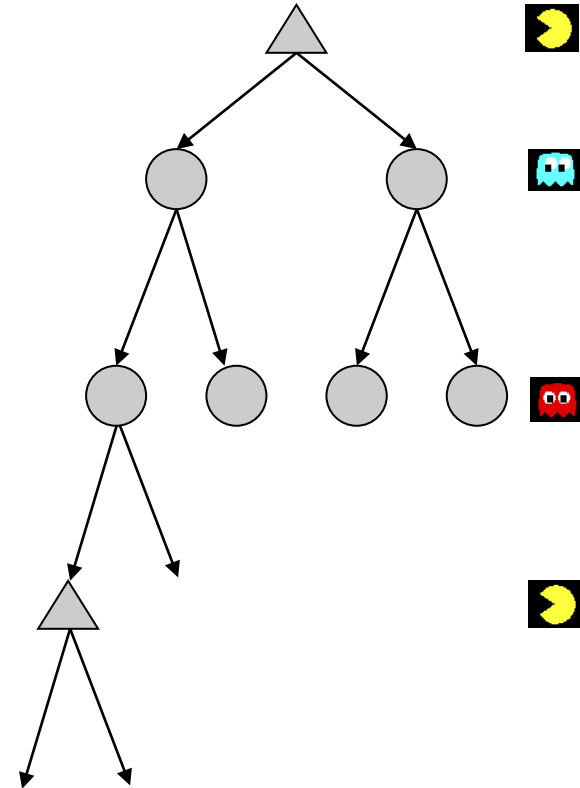
Généralisation aux actions aléatoires



- Exemples :
 - ◆ Jeux où on lance un dé pour déterminer la prochaine action
 - ◆ Actions des fantômes dans Pacman
- **Solution** : On ajoute des nœuds chance, en plus des nœuds Max et Min
 - ◆ **L'utilité d'un nœud chance est l'utilité espérée**, c.-à-d., la moyenne pondérée de l'utilité de ses enfants

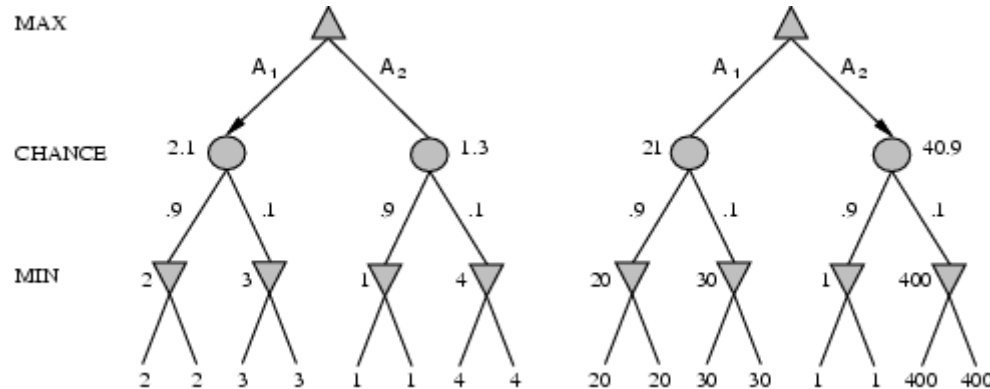
Algorithme Expectimax

- Un model probabiliste des comportement de l'opposant:
 - ◆ Le modèle peut être une simple distribution de probabilités
 - ◆ Le modèle peut être plus sophistiqué, demandant des inférences/calculs élaborés
 - ◆ Le modèle peut représenter des actions stochastiques/incontrôlables (à cause de de l'opposant, l'environnement)
 - ◆ Le modèle pourrait signifier que des actions de l'adversaire sont probables
- Pour cette leçon, supposer que (de façon magique) nous avons une distribution de probabilités à associer aux actions de l'adversaire/environnement



Avoir une croyance probabiliste sur les actions d'un agent ne signifie pas que l'agent lance effectivement un dé!

Algorithme Expectiminimax



EXPECTIMINIMAX(s) =

UTILITY(s)

$\max_{a \in \text{Actions}(s)} \text{EXPECTIMINIMAX}(\text{RESULT}(s,a))$

$\min_{a \in \text{Actions}(s)} \text{EXPECTIMINIMAX}(\text{RESULT}(s,a))$

$\sum_r P(r) * \text{EXPECTIMINIMAX}(\text{RESULT}(s,r))$

if IS-TERMINAL(s)

if TO-MOVE(s)=MAX

if TO-MOVE(s) = MIN

if TO-MOVE(s) = CHANCE

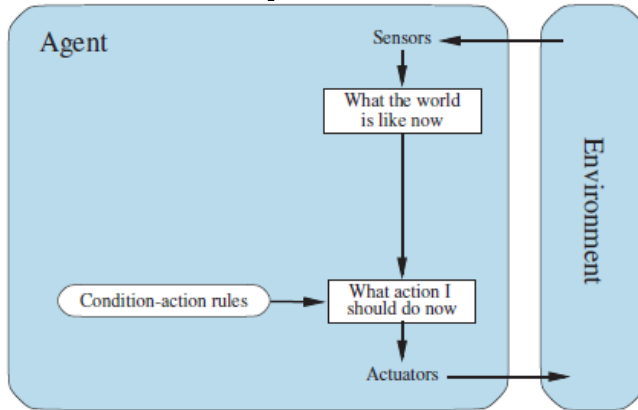
Ces équations donne la programmation récursive des valeurs jusqu'à la racine de l'arbre.

Quelques succès et défis

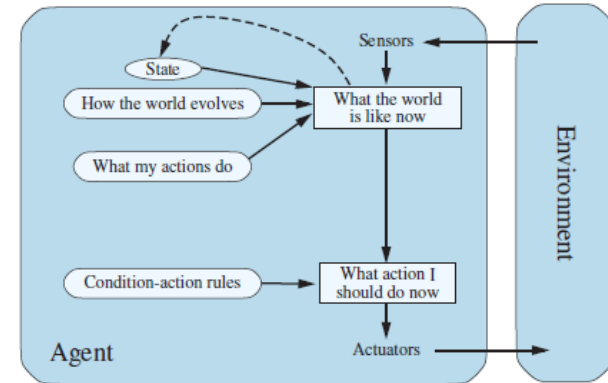
- Jeu de dames: En 1994, Chinook a mis fin aux 40 ans de règne du champion du monde Marion Tinsley.
- Jeu d'échecs: En 1997, Deep Blue a battu le champion du monde Garry Kasparov dans un match de six jeux.
- Go: AlphaGo bat le champion mondial pour la première fois en 2015!

Planification pour quel type d'agents?

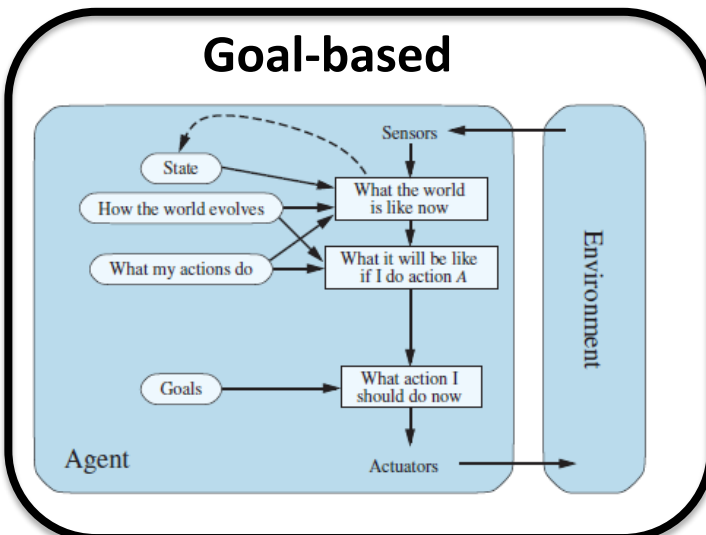
Simple reflex



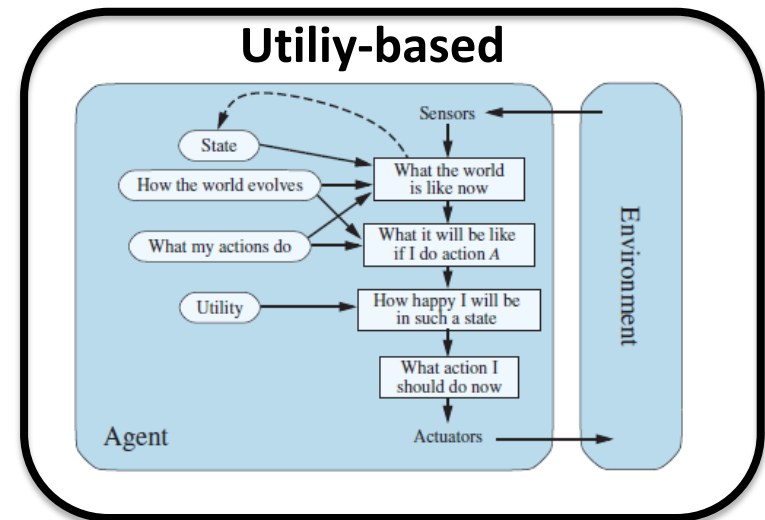
Model-based reflex



Goal-based



Utility-based



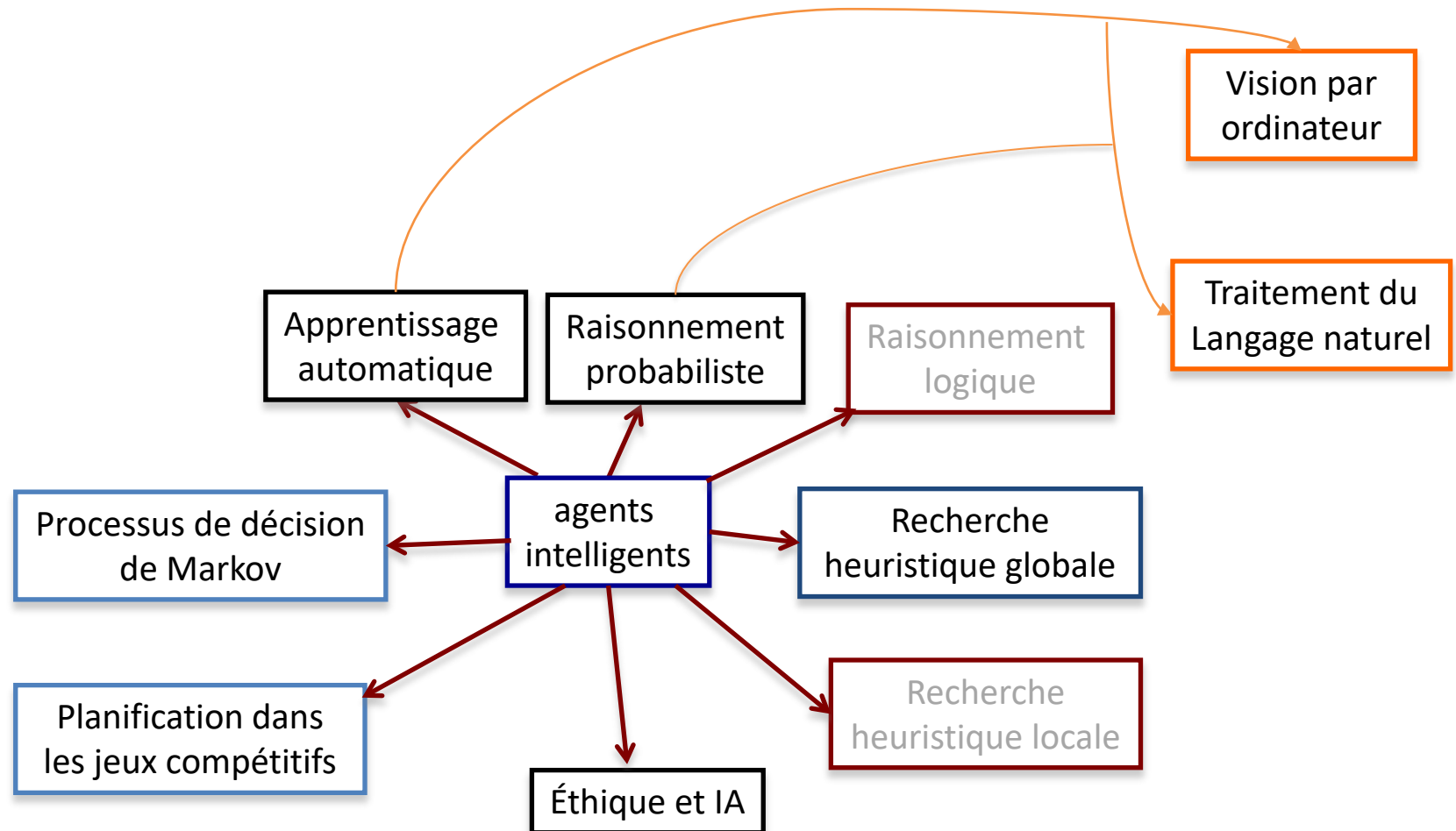
Vous devriez être capable de...

- Décrire formellement le problème de recherche associée au développement d'une IA pour un jeu à deux adversaires
- Décrire les algorithmes:
 - ◆ Minimax
 - ◆ Élagage alpha-bêta
 - ◆ Expectiminimax (pas couvert à l'examen)
 - ◆ Monte-Carlo Tree-Search (pas couvert à l'examen)
- Connaître leurs propriétés théoriques
- Simuler l'exécution de ces algorithmes
- Décrire comment traiter le cas en temps réel

Sujets couverts par le cours

Concepts et algorithmes

Applications



EXTRA: PAS COUVERT

AlphaGo Zero & Alpha Zero

AphaGo: Silver d. et al. Mastering the game of Go with DNNwand tree search. [Nature, 2017](#)

AlphaGo Zero: Silver d. et al. Mastering the game of Go without human knowledge. [Nature, 2017](#)

AlphaZero: Silver D. et al. Mastering Chess and Shogi by Self-Play with a General RL Algorithm. [arXiv 1712.01815](#)

- AlphaGo Zero combine l'apprentissage automatique avec tree-search (pas MCT!)
- Un réseau de neurone f_θ prend en entrée l'état s du jeu. Il a deux sorties (p, v) :
 - ◆ **politique:** $p(a|s)$ est la probabilité de choisir l'action a (inclus ne rien faire) dans l'état s .
 - ◆ **valeur:** $v(s)$ est la probabilité (pour les noirs) de gagner à partir de l'état s .
- Le réseau est entraîné en jouant contre lui-même pour générer des exemples (s_t, π_t, z_t) : π_t est la politique suivie à partir de s_t et $z_t \in \{-1, 1\}$ est le résultat de la partie jouée à partir de s_t (+1 si on gagne, -1 sinon).

$$(p, v) = f_\theta(s) \text{ avec la } \text{loss}(p, v, \pi, z) = (z - v)^2 - \pi \log p + c \|\theta\|^2$$

- La politique π_t est générée par UCT avec une politique de sélection

$$UCB(s, a) = Q(s, a) + c * \frac{p(s, a)}{1 + N(s, a)}$$

Noté $U(s, a)$ dans le papier

- La politique de simulation est $p(a|s)$ définie par le reseau de neurones f_θ

