

# CHAPITRE 4

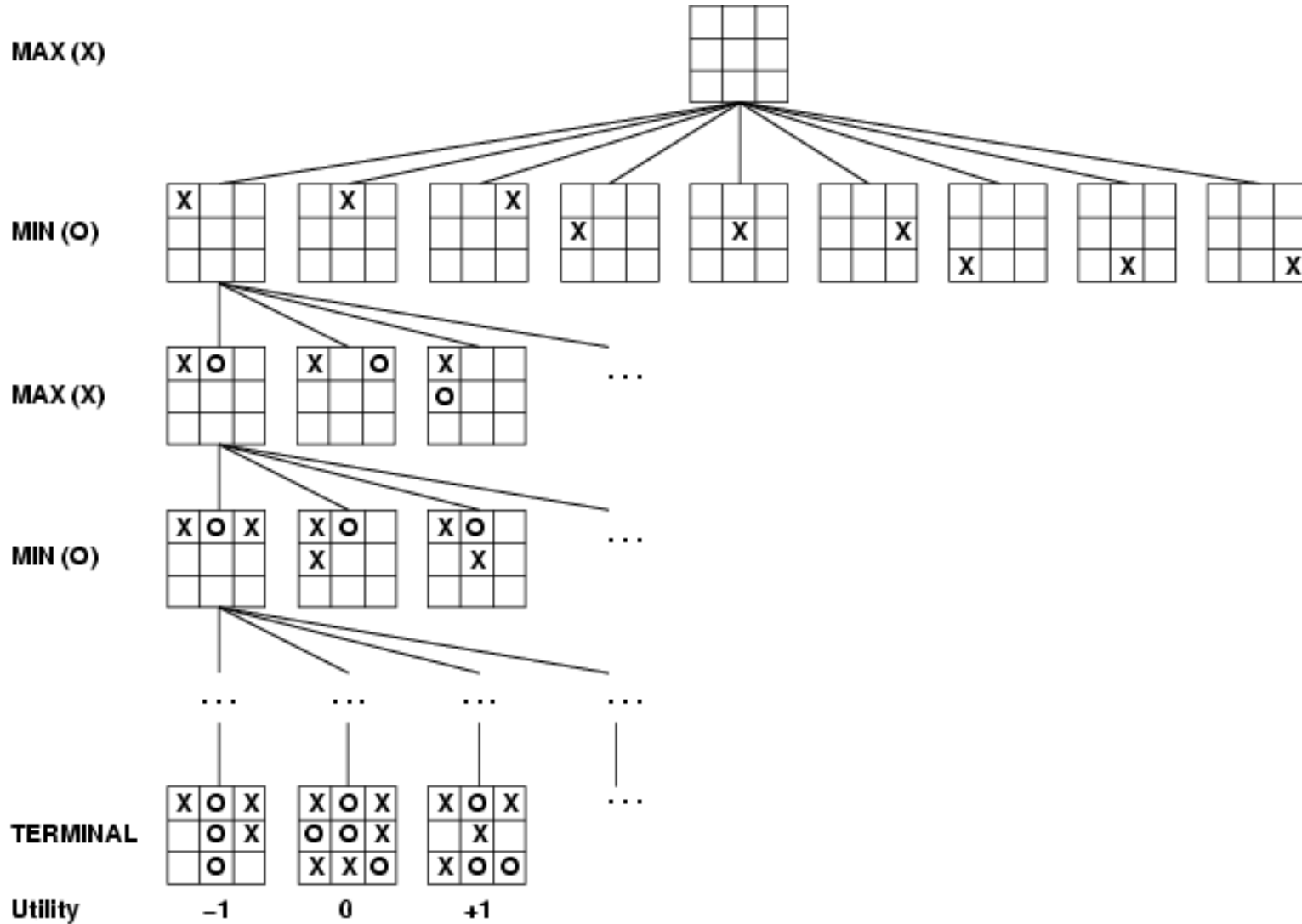
## LES ALGORITHMES DE JEUX

©Haythem Ghazouani

# OBJECTIFS

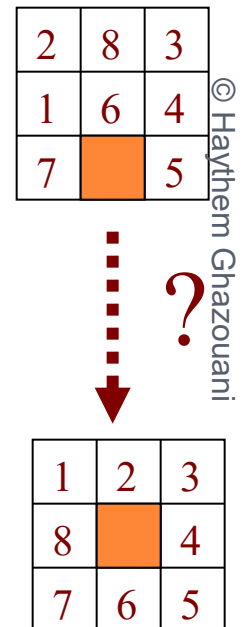
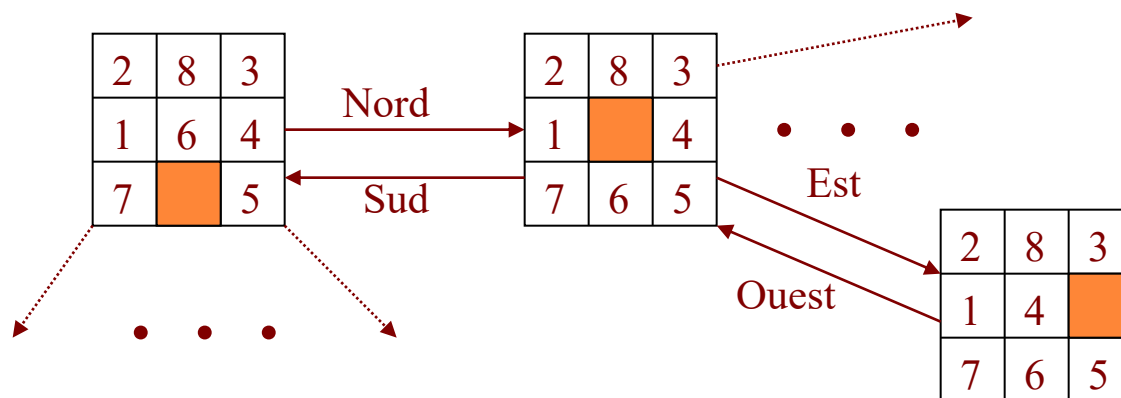
- Comprendre l'approche générale pour développer une IA pour un jeu avec adversaires
- Comprendre et pouvoir appliquer l'algorithme *minimax*
- Comprendre et pouvoir appliquer l'algorithme d'élagage *alpha-bêta*
- Savoir traiter le cas de décisions imparfaites en temps réel (temps de réflexion limité)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

# ARBRE DU JEU *TIC-TAC-TOE* (*JEU DU MORPION*)



# RAPPEL SUR A\*

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



## 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 les jeux ?
- R : Dans les jeux, l'environnement est multi-agent. 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.

# RELATION ENTRE LES JOUEURS

- 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, Connect 4, etc.
  - **Mixte**
    - il y a tout un spectre entre les jeux purement coopératifs et les jeux avec adversaires (ex. : alliances)

# PARTICULARITÉ DES JEUX À TOUR DE RÔLE

- Dépendamment des jeux, certains joueurs peuvent être:
  - Coopératifs
  - Rivals
- Les joueurs peuvent avoir une connaissance totale ou partielle de l'état du jeu.
- Ici nous considérons d'abord les jeux entre *deux adversaires, à somme nulle, à tour de rôle, avec une connaissance parfaite de l'état du jeu, avec des actions déterministes*.
- Nous aborderons brièvement les généralisations à plusieurs joueurs et avec des actions aléatoires.

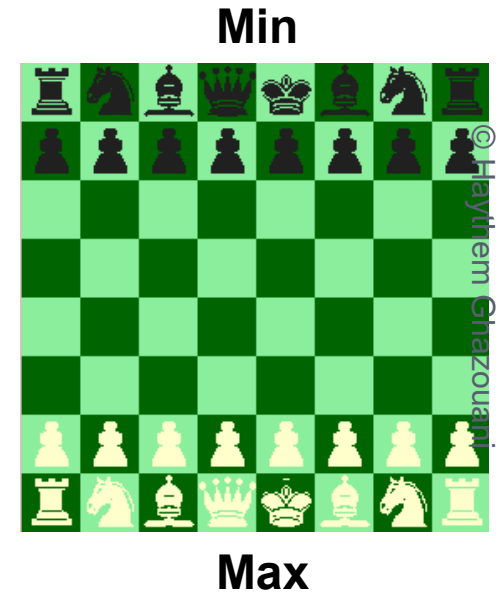


# HYPOTHÈSES POUR CE COURS

- Dans ce cours, nous aborderons les :
  - jeux à **deux adversaires**
  - jeux à **tour de rôle**
  - jeux à **somme nulle**
  - jeux avec **complètement observés**
  - jeux **déterministes** (sans hasard ou incertitude)
- Brièvement, 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).

# JEUX ENTRE DEUX ADVERSAIRES

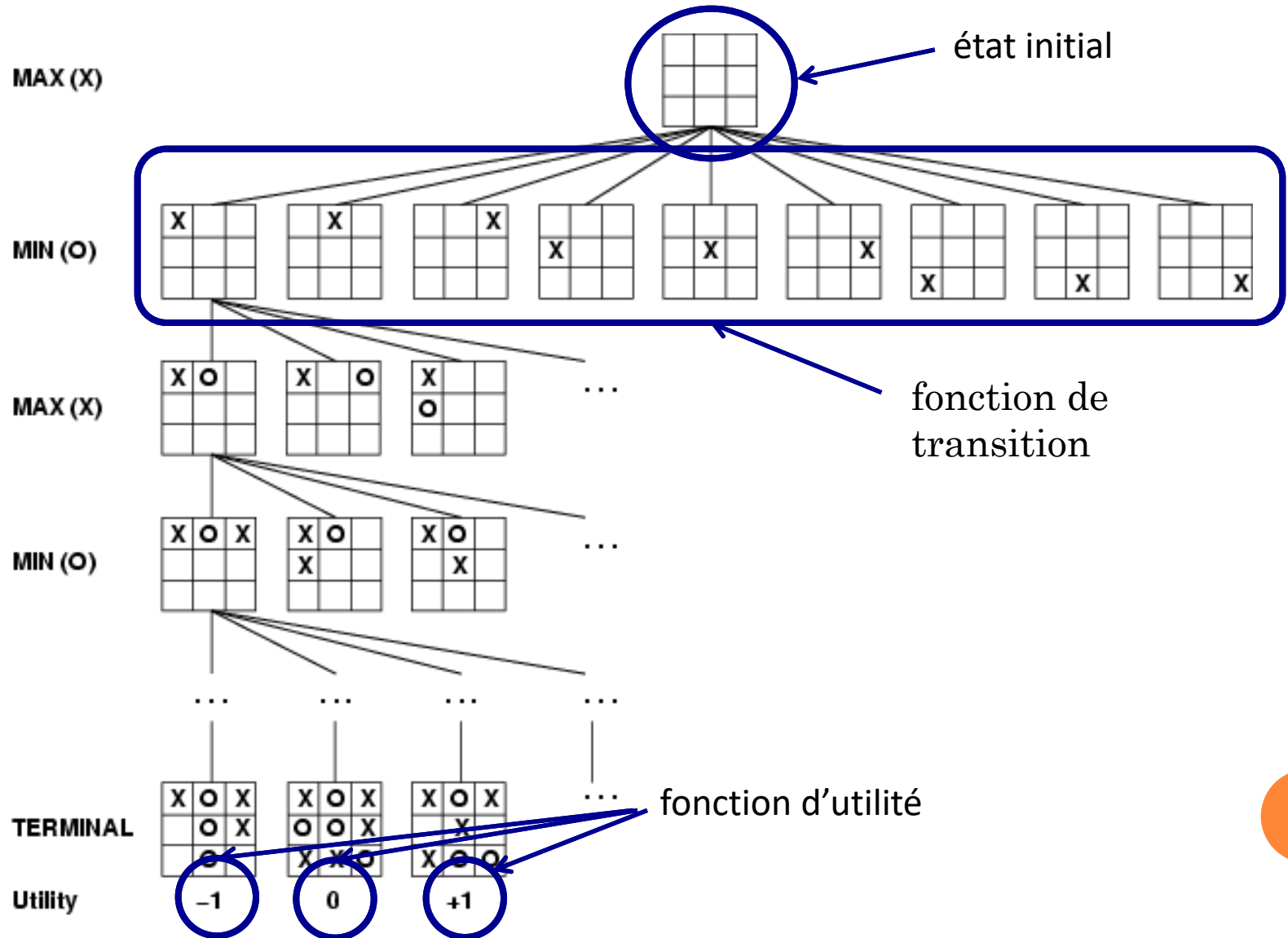
- Noms des joueurs : Max vs. Min
  - Max est le premier à jouer (notre joueur)
  - Min est son adversaire
- On va interpréter le résultat d'une partie comme la **distribution d'une récompense**
  - peut voir cette récompense comme le résultat d'un pari
  - Min reçoit l'opposé de ce que Max reçoit



# ARBRE DE RECHERCHE

- Comme pour les problèmes que  $A^*$  peut résoudre, 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 **noeud (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)

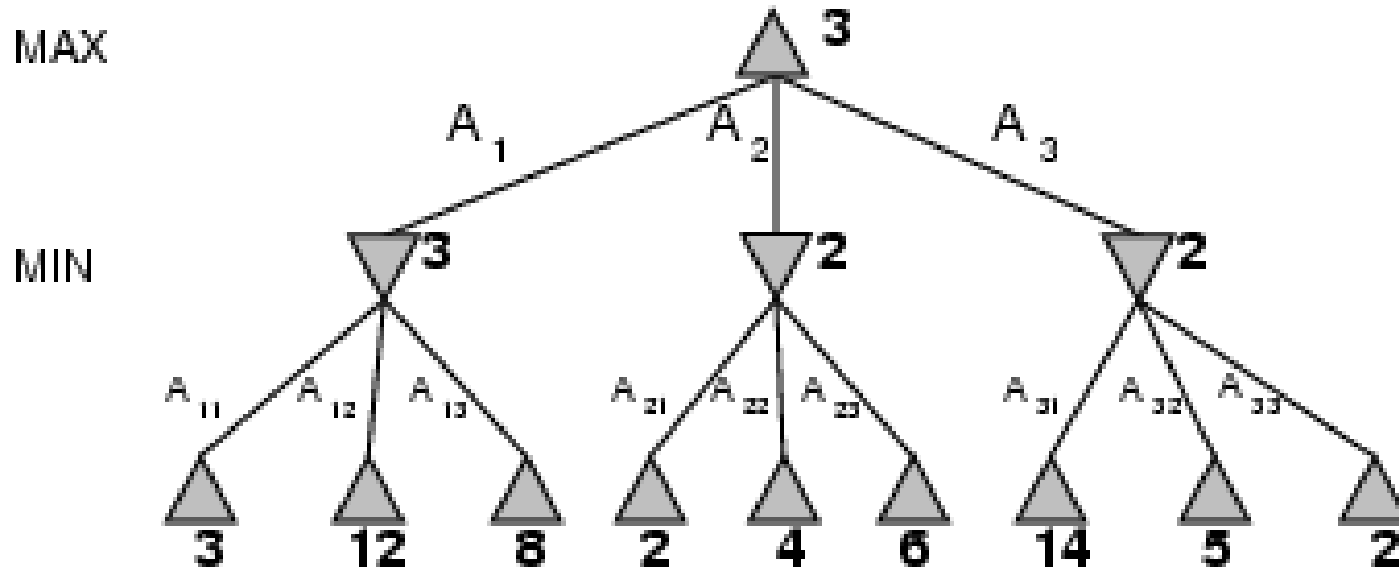
# ARBRE DE RECHERCHE TIC-TAC-TOE



# 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*

- **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).

EXPECTED-MINIMAX-VALUE( $n$ ) =

UTILITY( $n$ )

Si  $n$  est un nœud terminal

$\max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$

Si  $n$  est un nœud Max

$\min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$

Si  $n$  est un nœud Min

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

# ALGORITHME *MINIMAX*

**function** MINIMAX-DECISION(*state*) *returns an action*

$v \leftarrow \text{MAX-VALUE}(\text{state})$

**return** the *action* in **SUCCESSORS**(*state*) with value *v*

**function** MAX-VALUE(*state*) *returns a utility value*

**if** **TERMINAL-TEST**(*state*) **then return** **UTILITY**(*state*)

$v \leftarrow -\infty$

**for** *a, s* in **SUCCESSORS**(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$

**return** *v*

**function** MIN-VALUE(*state*) *returns a utility value*

**if** **TERMINAL-TEST**(*state*) **then return** **UTILITY**(*state*)

$v \leftarrow \infty$

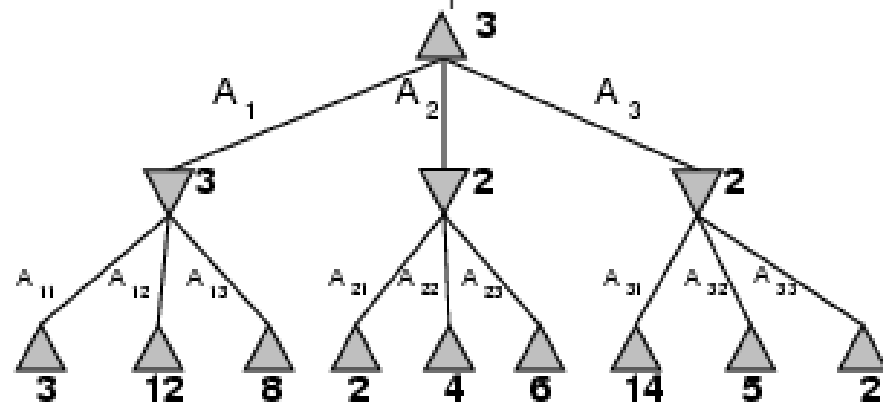
**for** *a, s* in **SUCCESSORS**(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$

**return** *v*

MAX

MIN



# 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

## ○ Deux approches

- la première maintient l'exactitude de la solution
- la deuxième introduit une approximation

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

- **idée** : identifier des chemins dans l'arbre qui sont explorés inutilement

### 2. Couper la recherche et remplacer l'utilité par une 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

# *ALPHA-BETA PRUNING*

- L'algorithme alpha-beta tire son nom des paramètres suivant décrivant les bornes des valeurs d'utilité enregistrée durant le parcours.
  - $\alpha$  est la valeur du meilleur choix pour Max (c.-à-d., plus grande valeur) trouvé jusqu'ici:
  - $\beta$  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  $\alpha$  est la valeur du meilleur choix pour Max (c.-à-d., plus grande valeur) trouvé jusqu'ici:
  - Si on est **dans un nœud Min** est que sa valeur  $v$  devient **inférieure  $\alpha$**  (donc « pire que  $\alpha$  » du point de vue de Max), il faut arrêter la recherche (couper la branche).

MAX

MIN

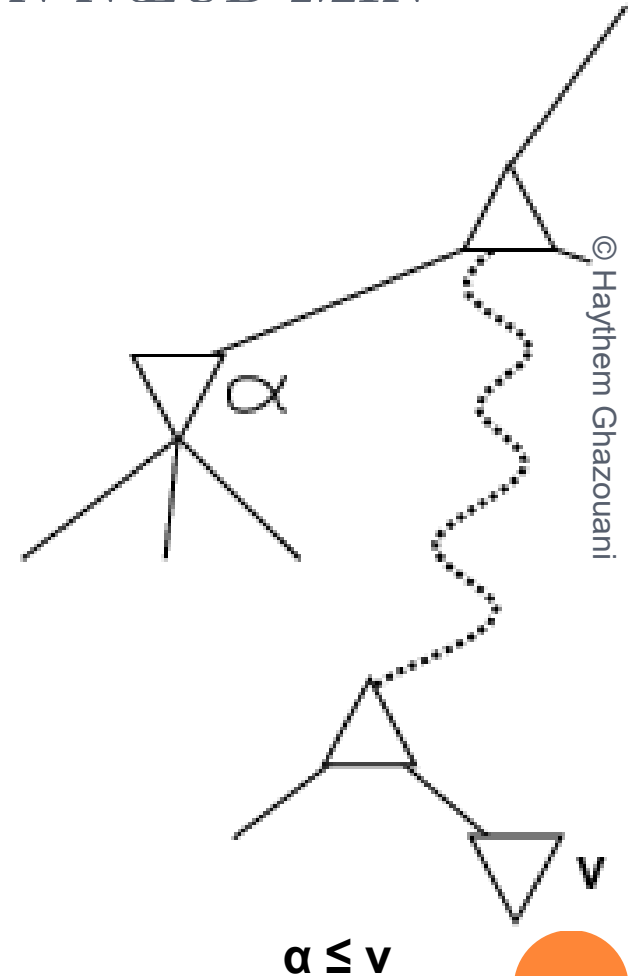
..

..

..

MAX

MIN



© Haythem Ghazouani

# ALPHA-BETA PRUNING

## *CONDITION POUR COUPER DANS UN NŒUD MAX*

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

# EXEMPLE D'ALPHA-BETA PRUNING

Faire une recherche en profondeur jusqu'à la première feuille

MAX

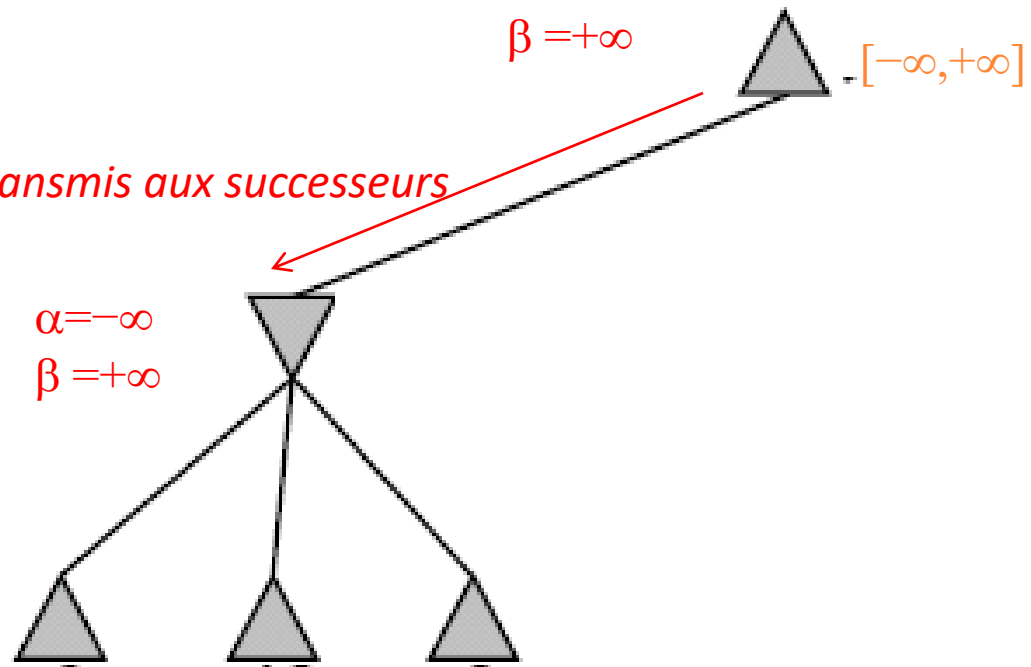
MIN

Valeur initial de  $\alpha, \beta$

$$\alpha = -\infty$$

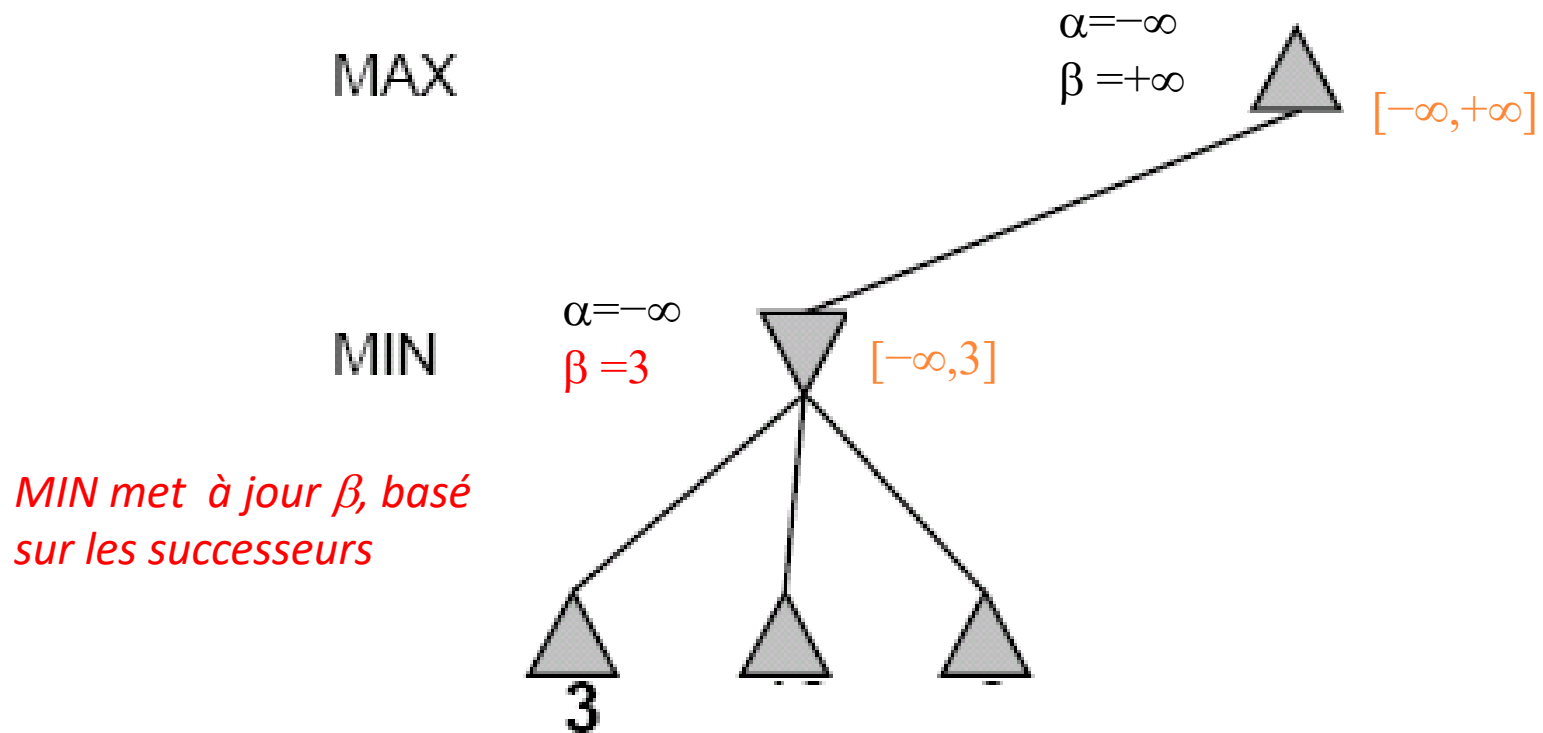
$$\beta = +\infty$$

$\alpha, \beta$ , transmis aux successeurs



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

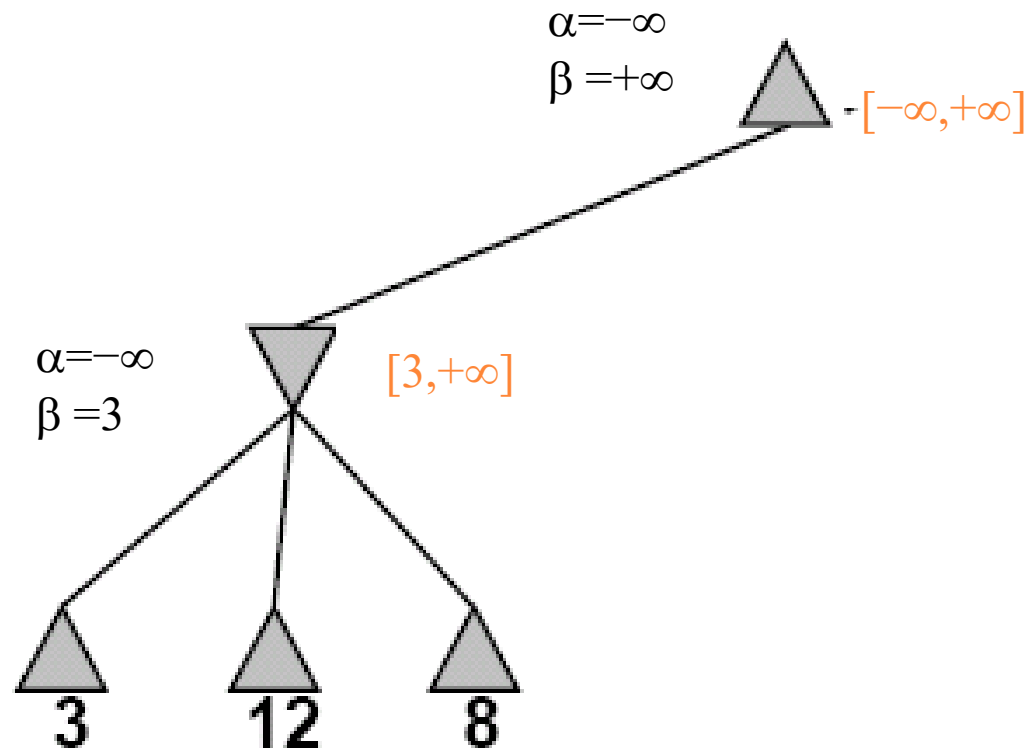
# EXEMPLE D'ALPHA-BETA PRUNING



# EXEMPLE D'ALPHA-BETA PRUNING

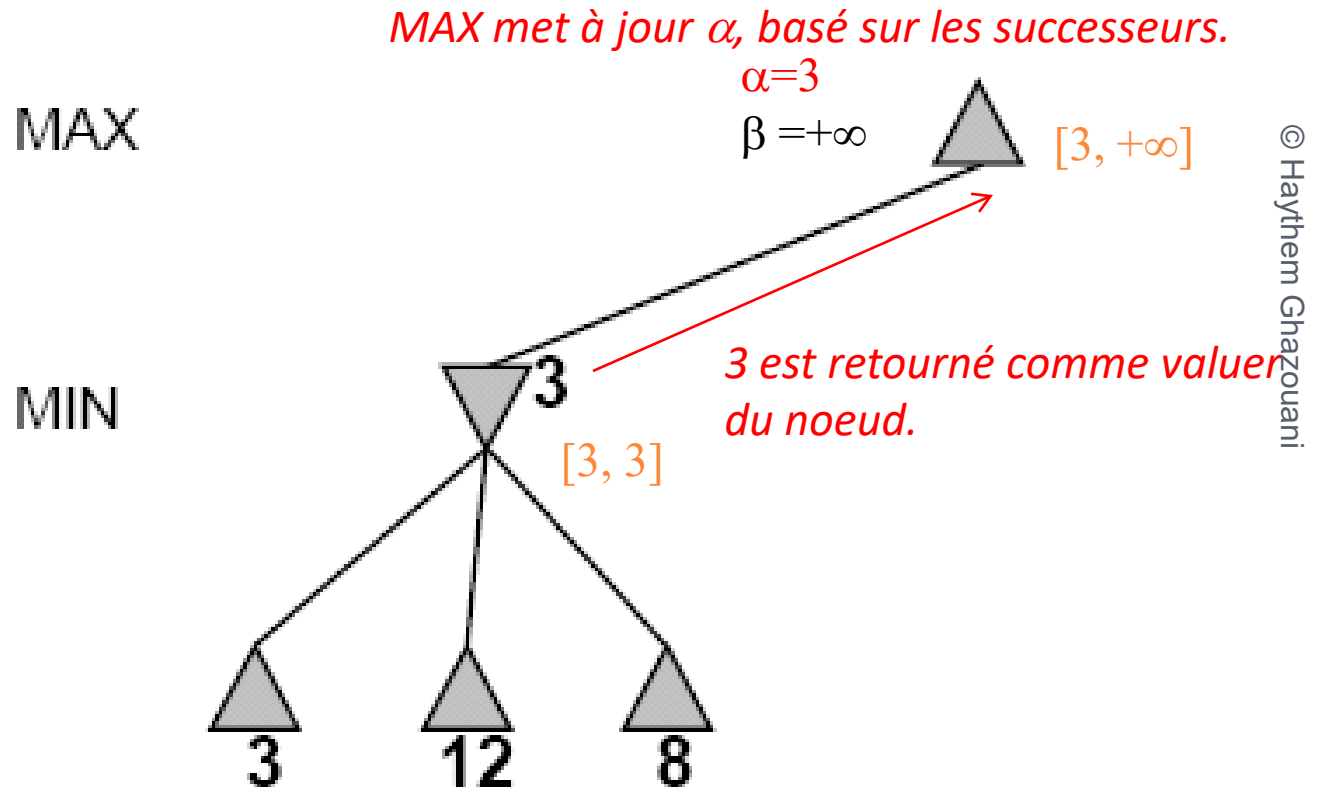
MAX

MIN



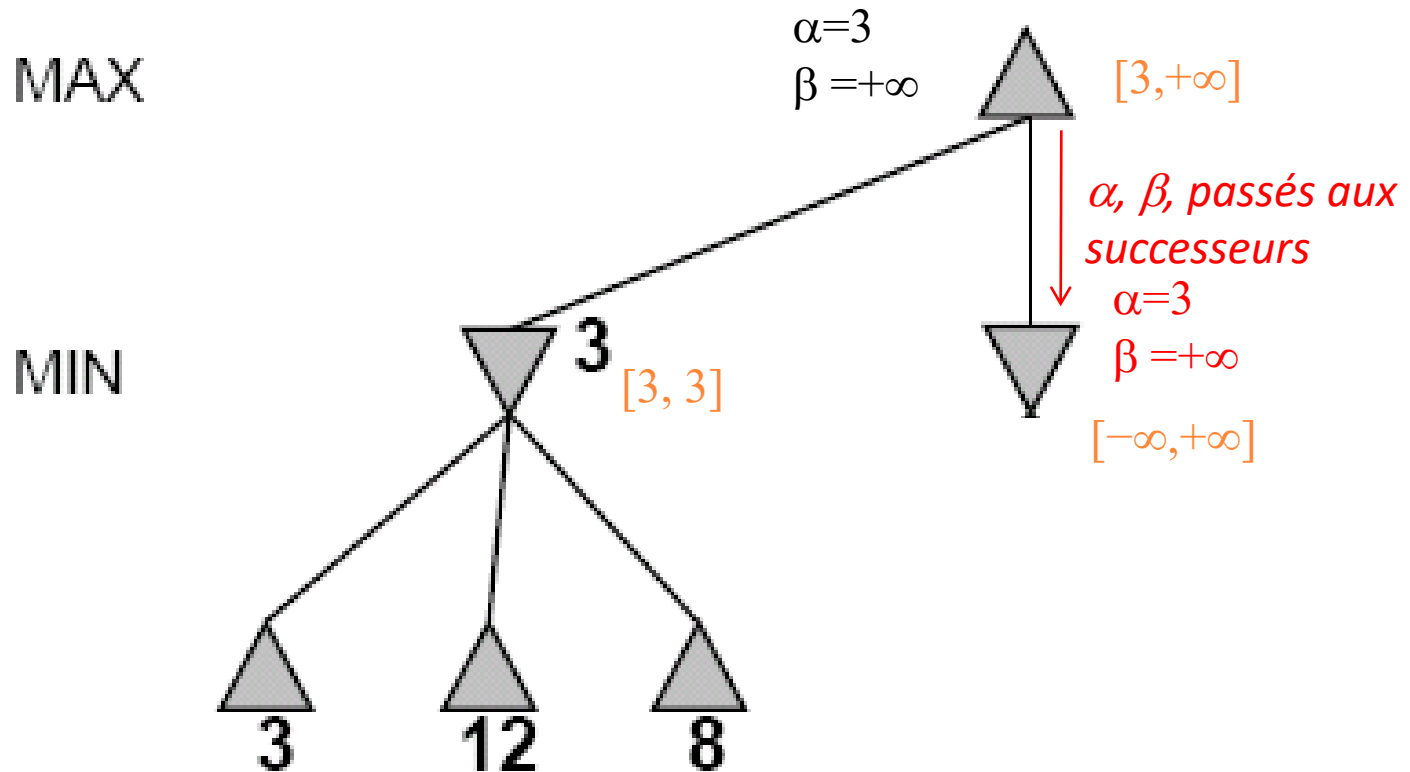
*MIN met à jour  $\beta$ , basé  
sur les successeurs.  
Aucun changement.*

# EXEMPLE D'ALPHA-BETA PRUNING

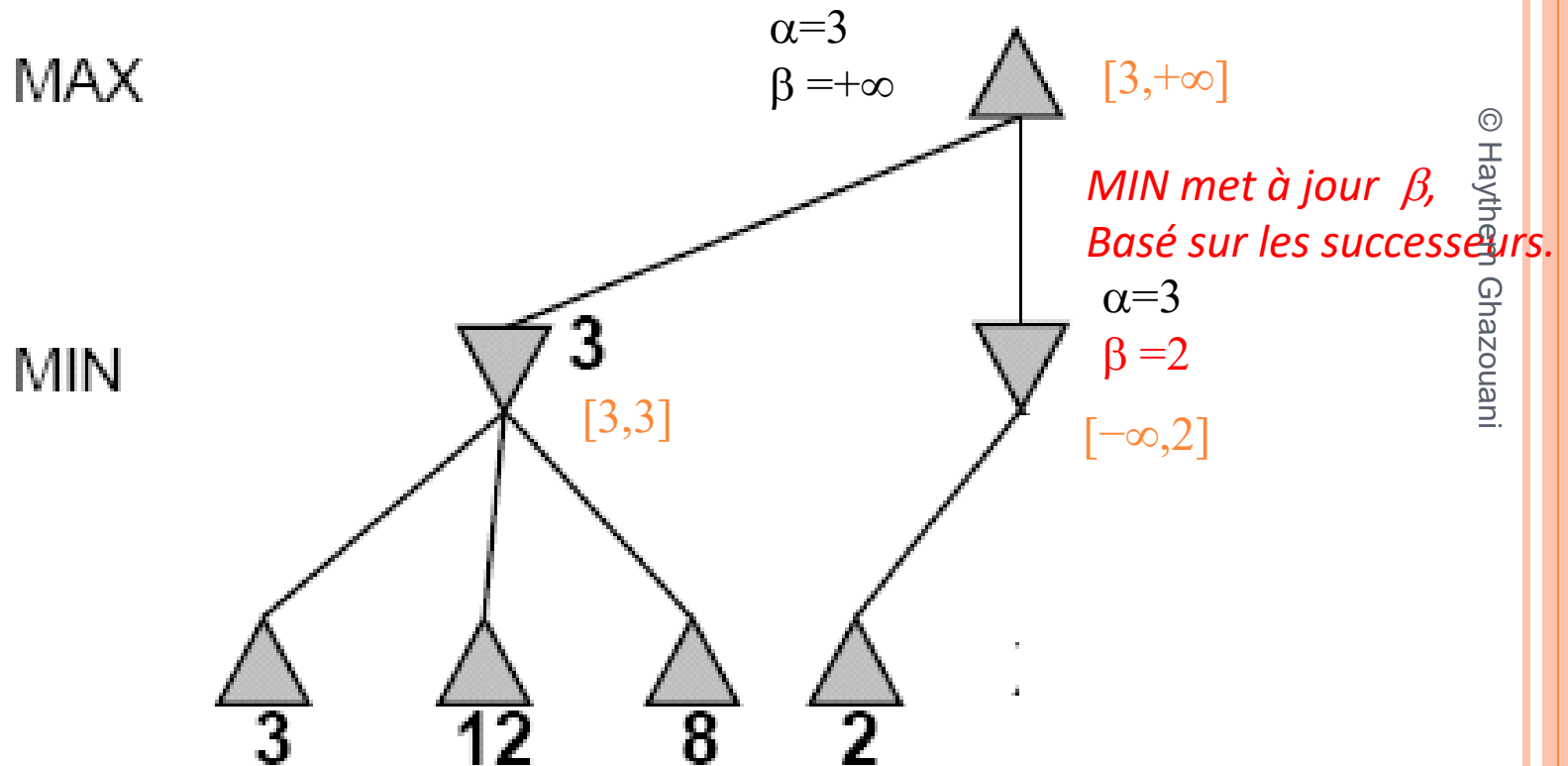




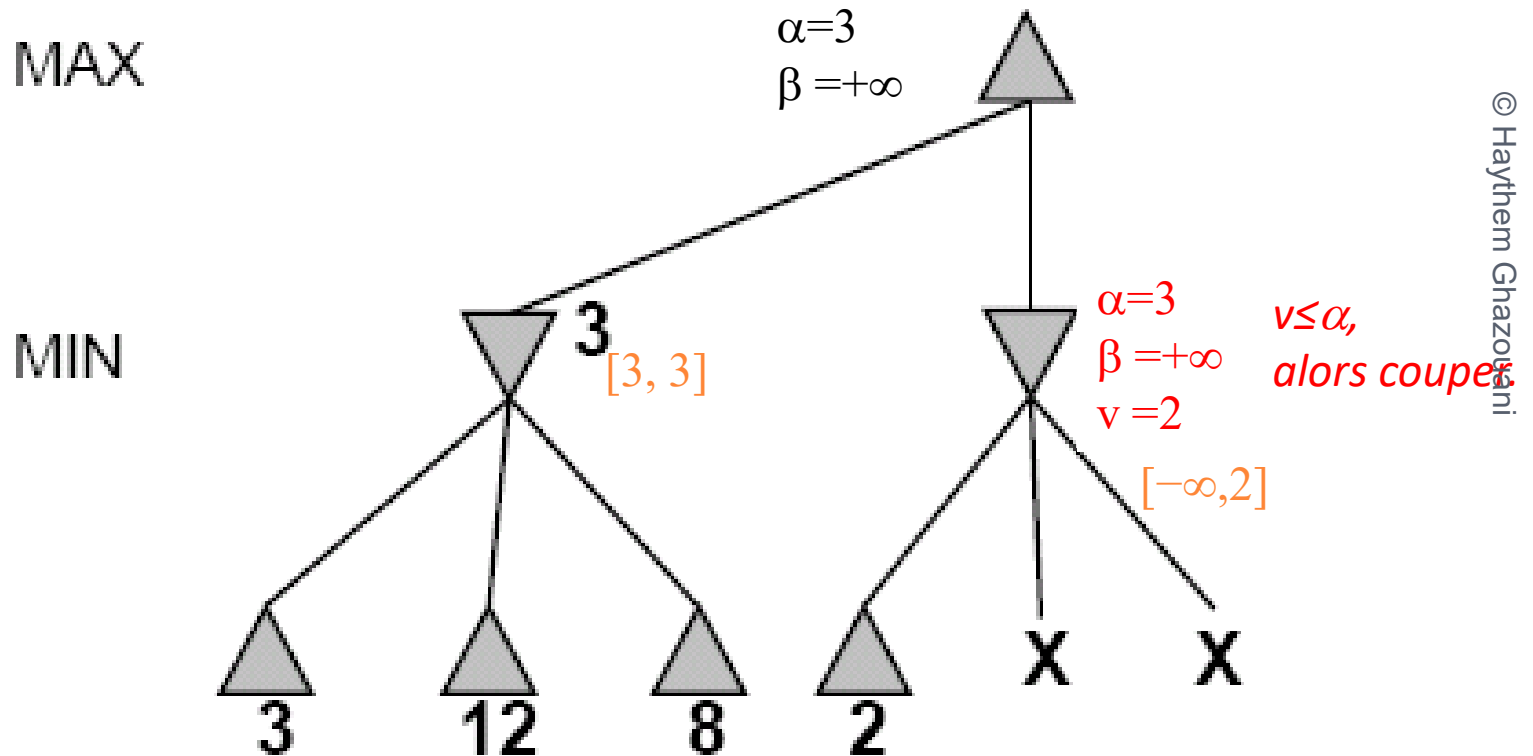
# EXEMPLE D'ALPHA-BETA PRUNING



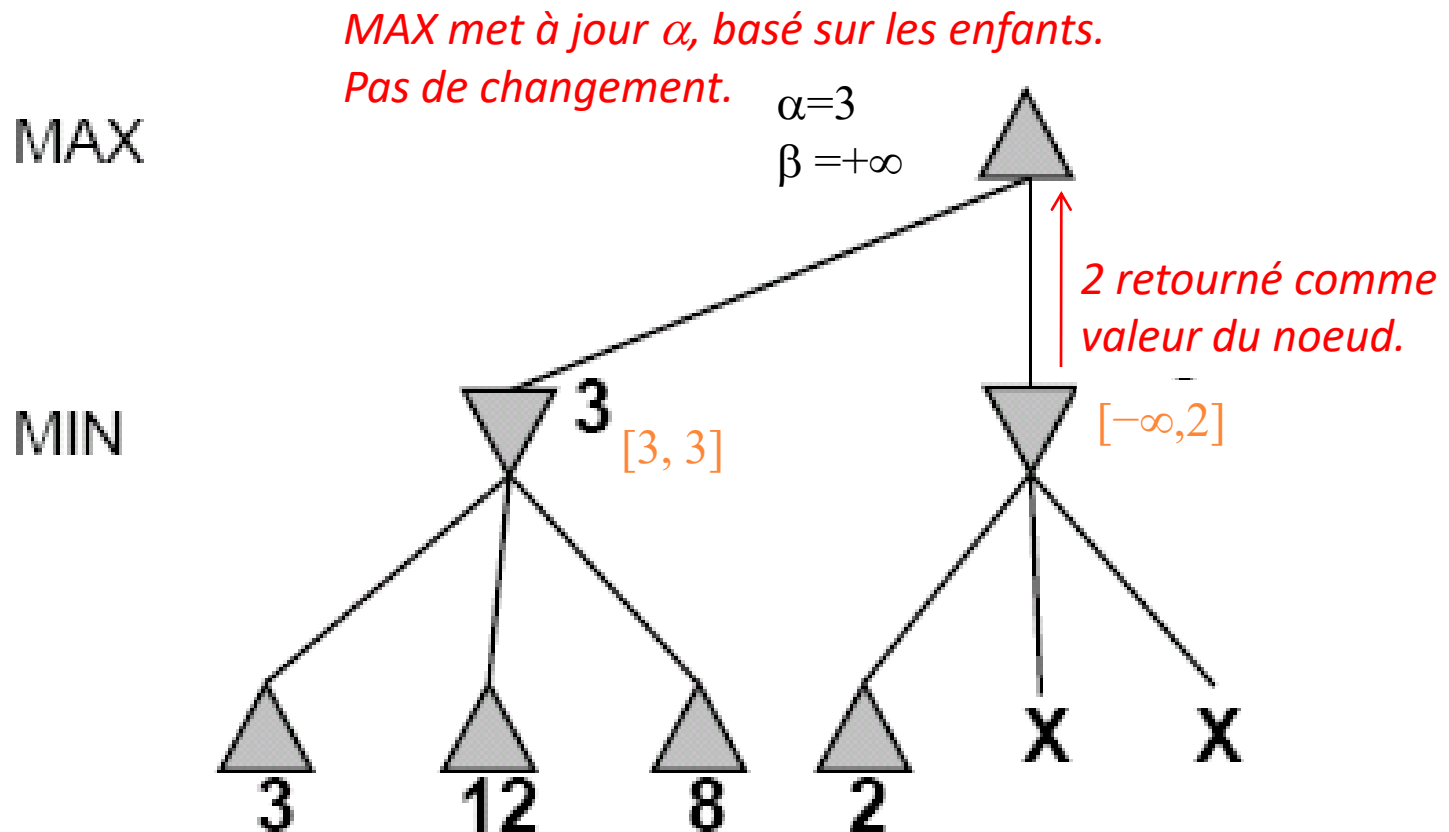
# EXEMPLE D'ALPHA-BETA PRUNING



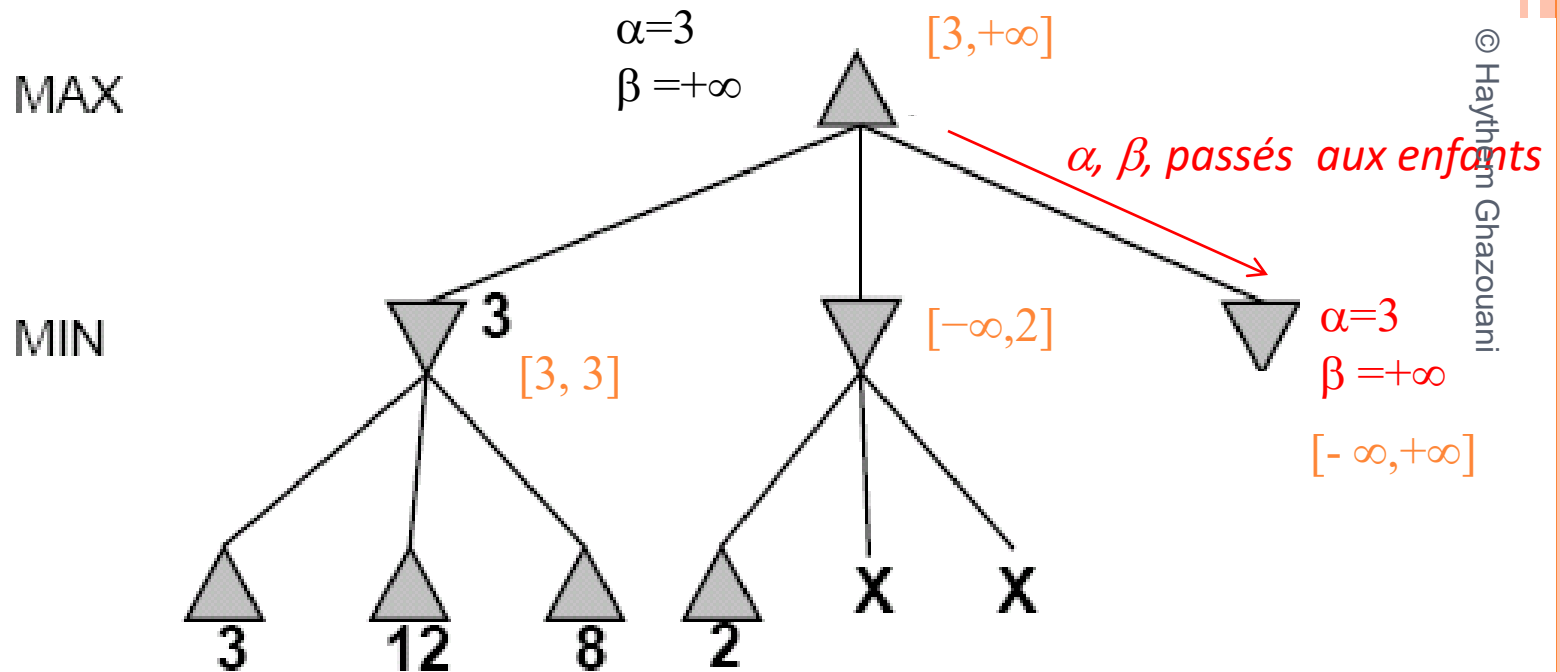
# EXEMPLE D'ALPHA-BETA PRUNING



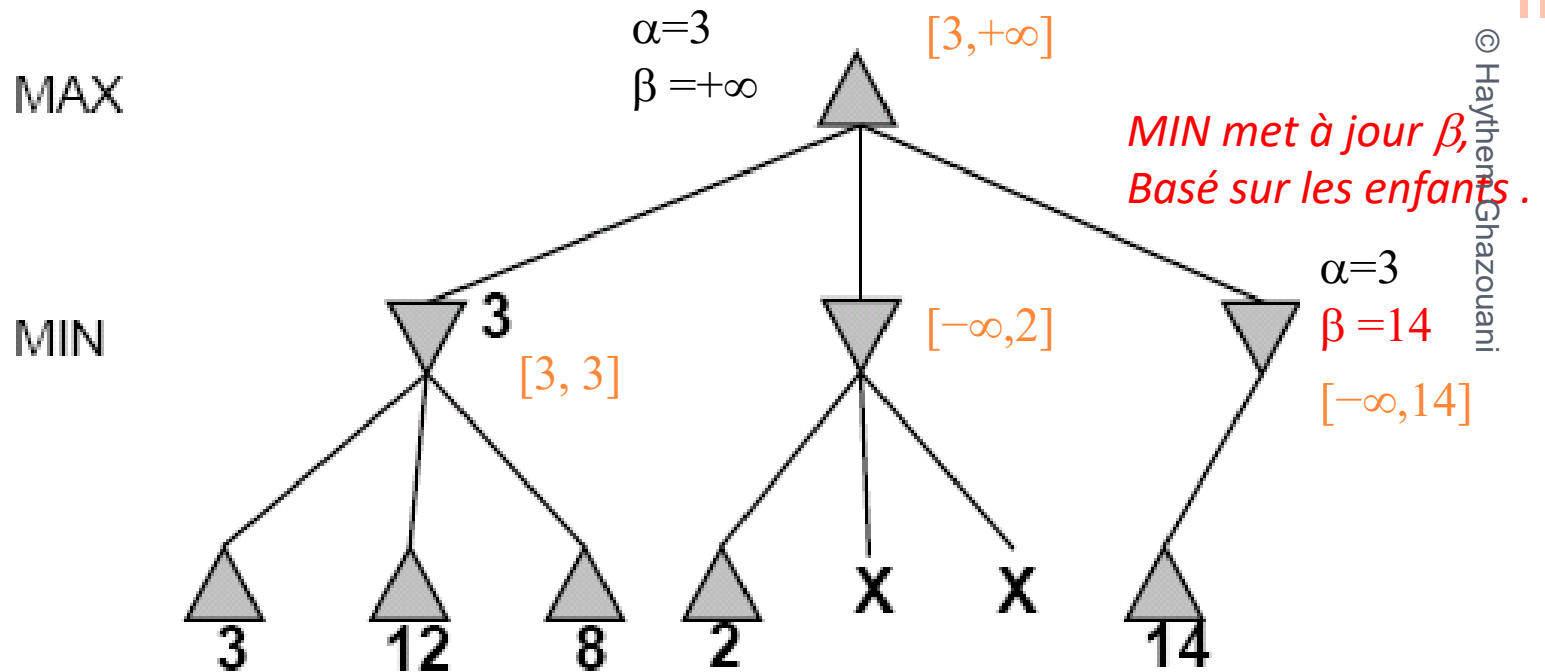
# EXEMPLE D'ALPHA-BETA PRUNING



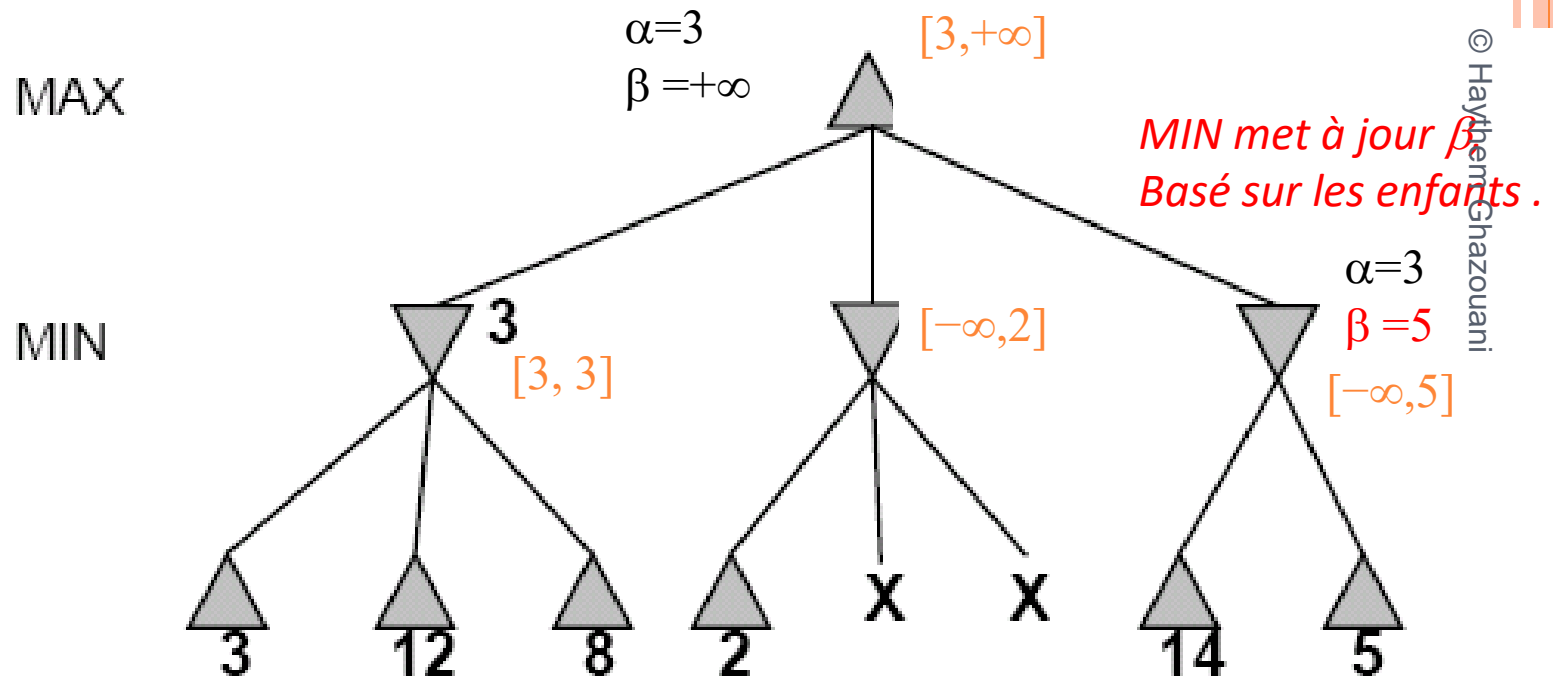
# EXEMPLE D'ALPHA-BETA PRUNING



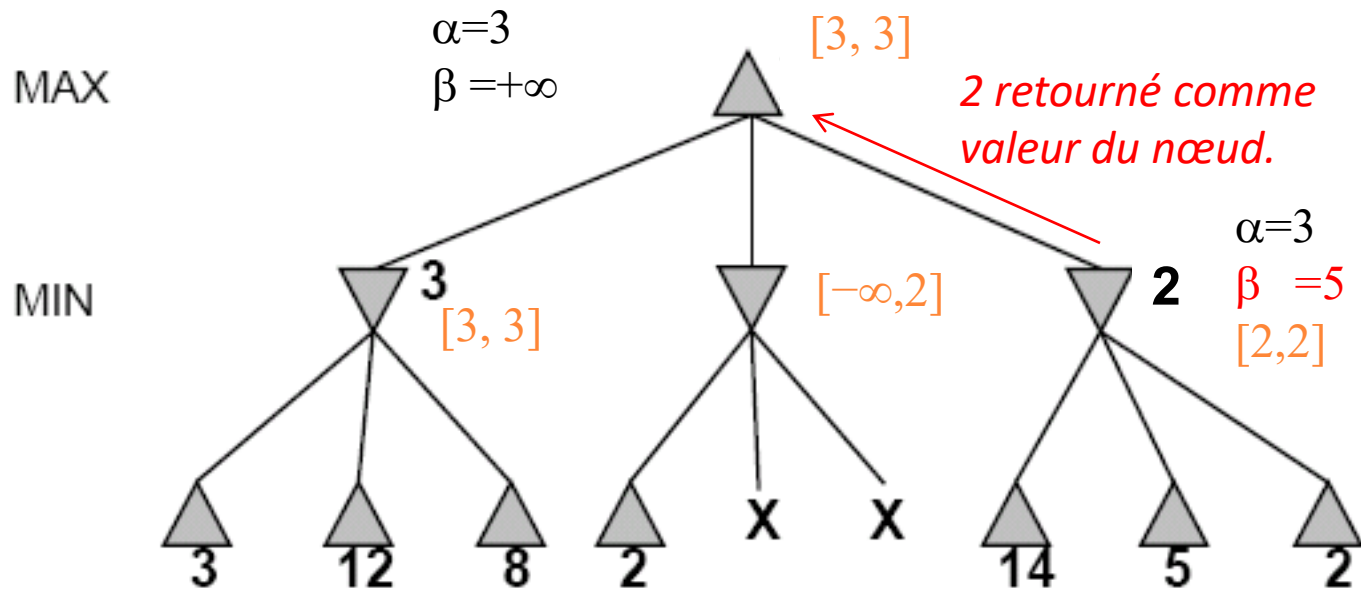
# EXEMPLE D'ALPHA-BETA PRUNING



# EXEMPLE D'ALPHA-BETA PRUNING

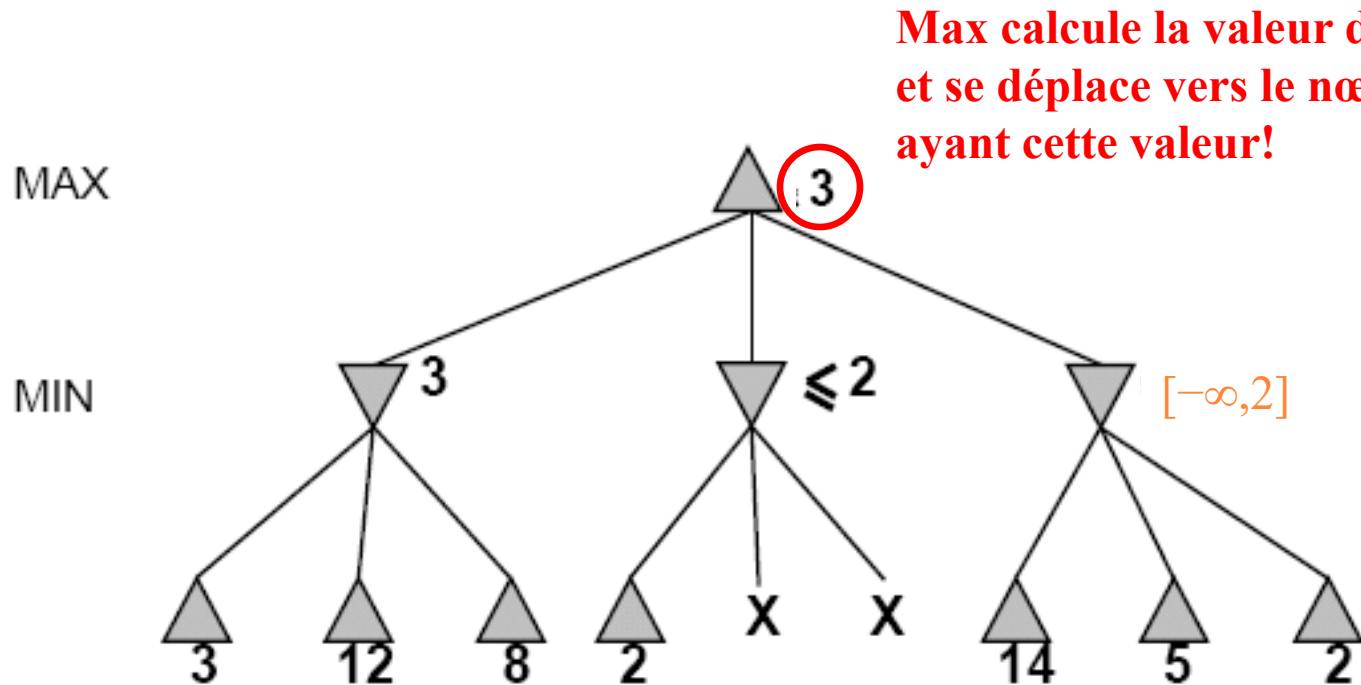


# EXEMPLE D'ALPHA-BETA PRUNING





# 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(*state*) *returns an action*

**inputs:** *state*, current state in game

$v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$

**return** the *action* in *SUCCESSORS*(*state*) with value  $v$

---

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) *returns a utility value*

**inputs:** *state*, current state in game

$\alpha$ , the value of the best alternative for MAX along the path to *state*

$\beta$ , the value of the best alternative for MIN along the path to *state*

**if** *TERMINAL-TEST*(*state*) **then return** *UTILITY*(*state*)

$v \leftarrow -\infty$

**for**  $a, s$  in *SUCCESSORS*(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$

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

$\alpha \leftarrow \text{MAX}(\alpha, v)$

**return**  $v$

# ALGORITHME *ALPHA-BETA PRUNING* (*SUITE*)

```
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: state, current state in game
            $\alpha$ , the value of the best alternative for MAX along the path to state
            $\beta$ , the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 
```

# NEGAMAX – VERSION ÉLÉGANTE DE A-B PRUNING

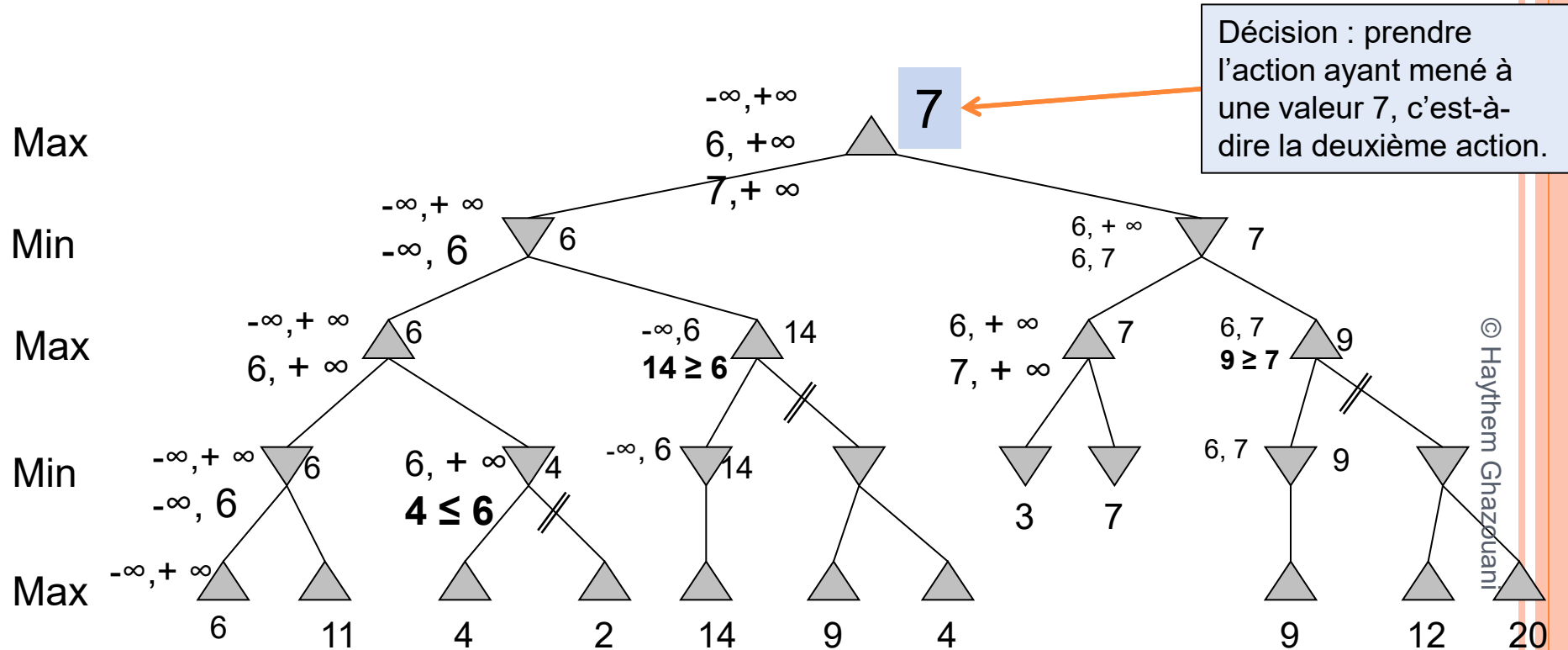
<http://en.wikipedia.org/wiki/Negamax>

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





*Appel initial* : negamax(initialState, depth, -inf, +inf, 1)


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

## AUTRE EXEMPLE



## 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)

$\alpha, \beta$   Valeur retournée

Valeur si  
feuille

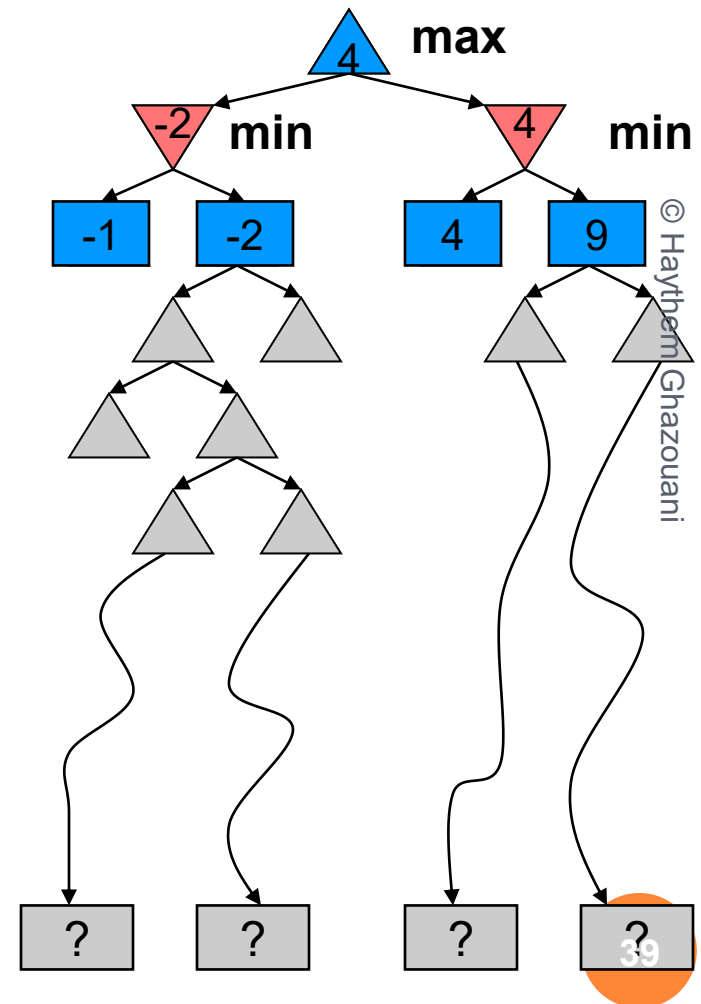
37

# PROPRIÉTÉS DE *ALPHA-BETA PRUNING*

- L'élagage n'affecte pas le résultat final de *minimax*.
- Dans le pire des cas, alpha-beta *prunning* ne fait aucun élagage; il examine  $b^m$  nœuds terminaux comme l'algorithme *minimax*:
  - $b$ : le nombre maximum d'actions/coups légales 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 plus profondément comparé à *minimax*!

# DÉCISIONS EN TEMPS RÉEL

- 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)

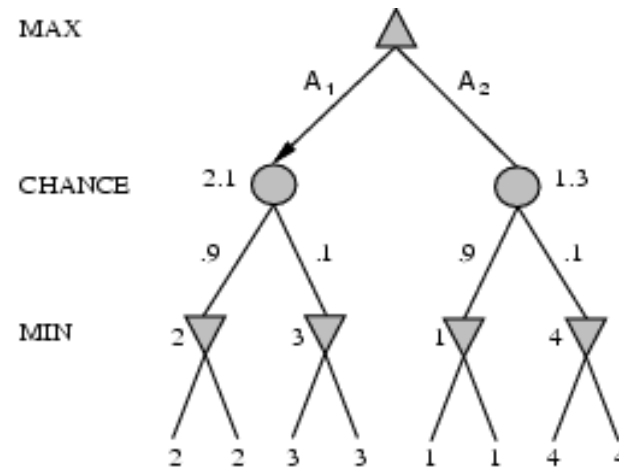
	X	O

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

O	X	X
	O	

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

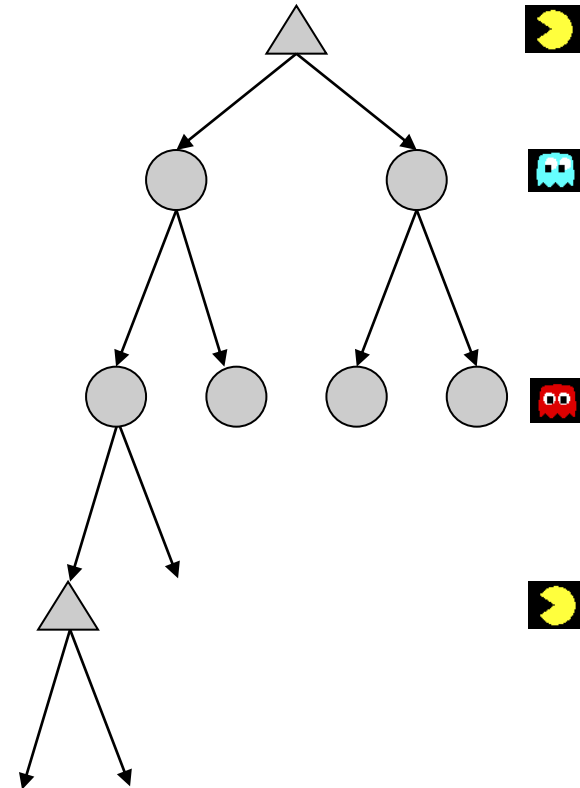
# GÉNÉRALISATION AUX ACTIONS ALÉATOIRES



- 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

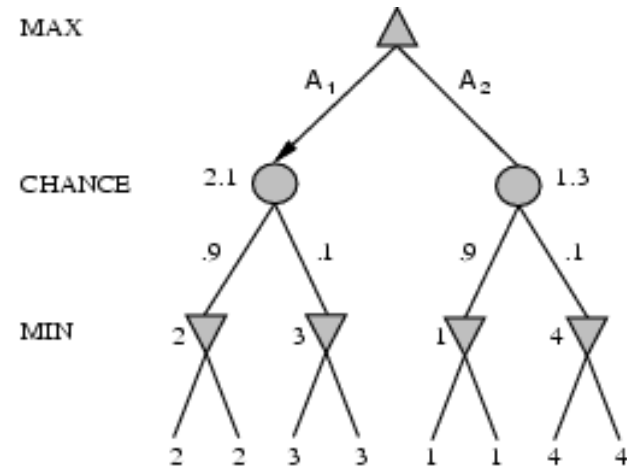
- Un modèle probabiliste des comportement des l'opposant:
  - Le modèle peut être une simple distribution de probabilités
  - Le modèle peut être plus sophistiqué, demandant des inférences/calculs élaborés
  - Le modèle peut représenter des actions stochastiques/incontrôlables (à cause de de l'opposant, 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



© Haythem Ghazouani

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

# ALGORITHME EXPECTIMAX



© Haythem Ghazouani

EXPECTIMAX ( $n$ ) =

UTILITY( $n$ )

$\max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$

$\min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$

$\sum_{s \in \text{successors}(n)} P(s) * \text{EXPECTEDMINIMAX}(s)$

Si  $n$  est un nœud terminal

Si  $n$  est un nœud Max

Si  $n$  est un nœud Min

Si  $n$  est un nœud chance

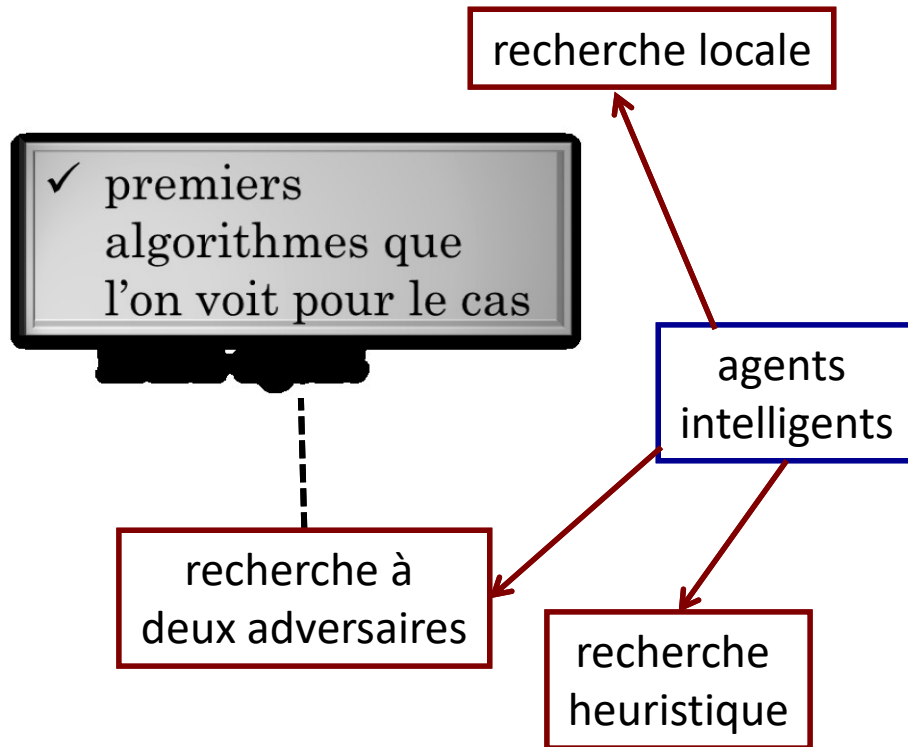
Ces équations donne la programmation récursive des valeurs jusqu'à la racine de l'arbre.

# 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.
- Othello: les champions humains refusent la compétition contre des ordinateurs, parce que ces derniers sont trop bons!
- Go: les champions humains refusent la compétition contre des ordinateurs, parce que ces derniers sont trop mauvais!

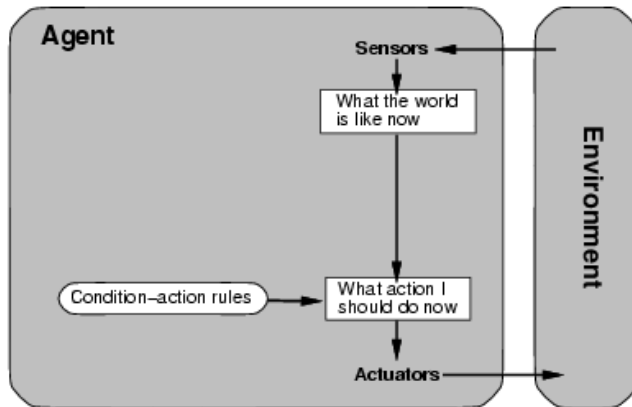
# OBJECTIFS DU COURS

## Algorithmes et concepts

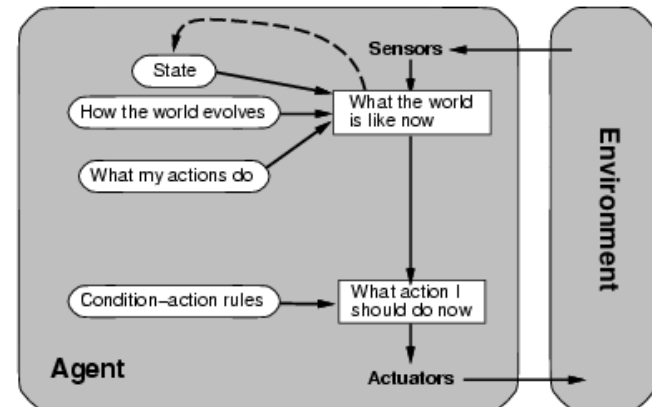


# ALGORITHMES POUR JEUX À TOUR DE RÔLE : POUR QUEL TYPE D'AGENT?

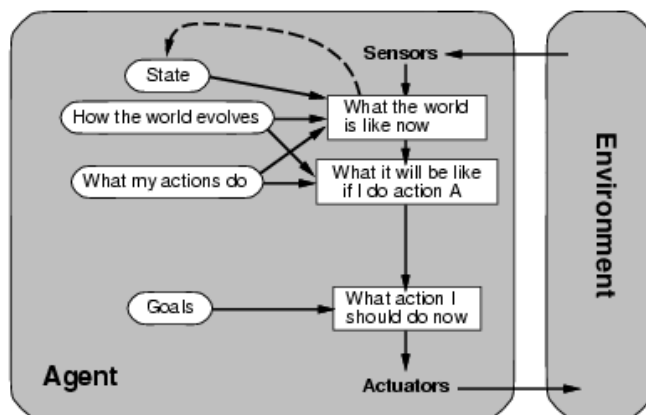
## Simple reflex



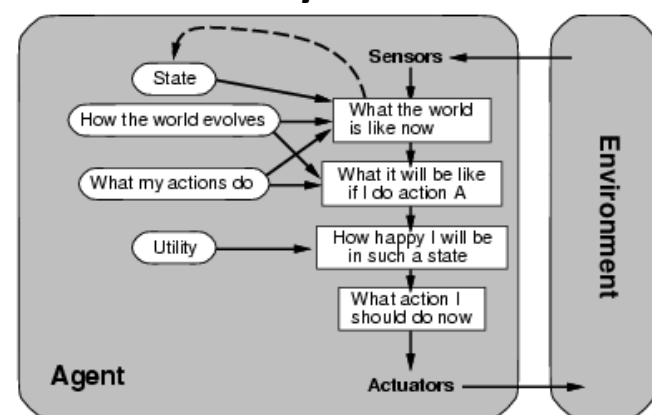
## Model-based reflex



## Goal-based

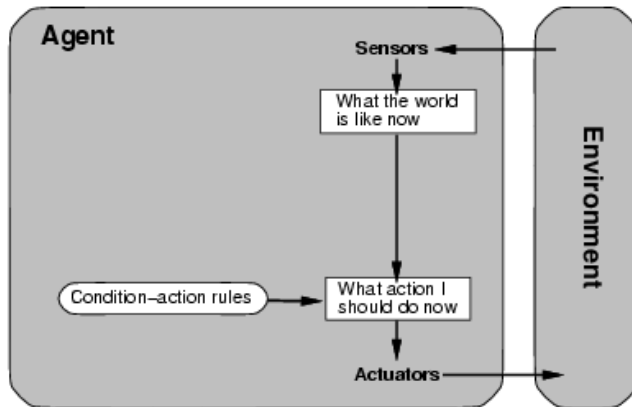


## Utility-based

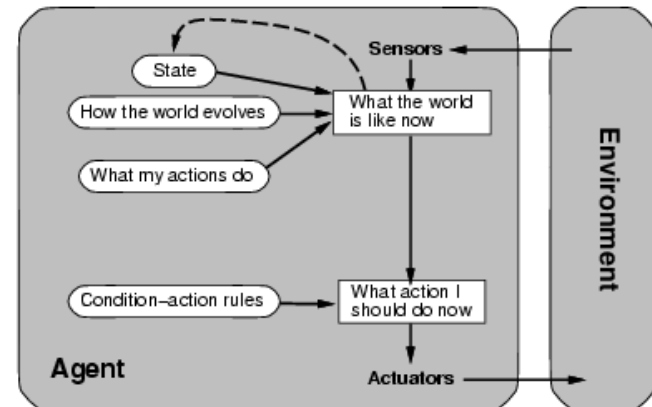


# ALGORITHMES POUR JEUX À TOUR DE RÔLE : POUR QUEL TYPE D'AGENT?

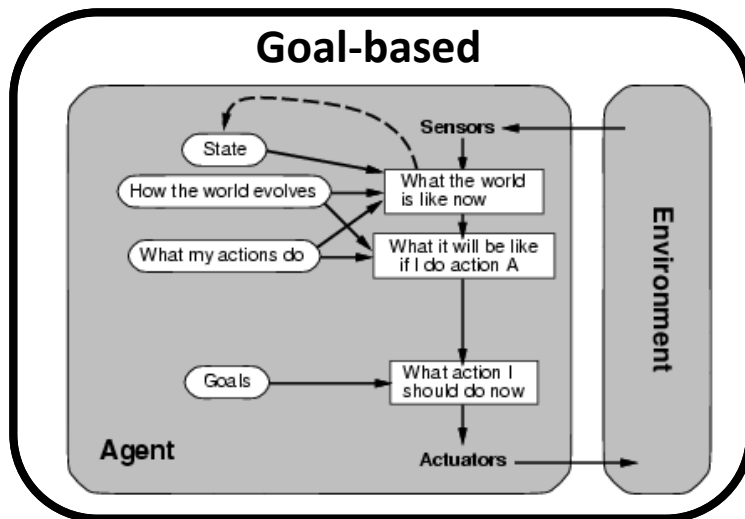
## Simple reflex



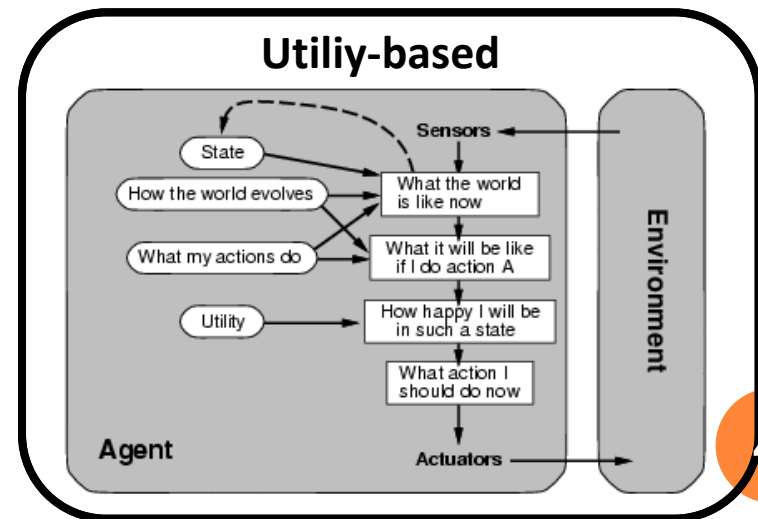
## Model-based reflex



## Goal-based



## Utility-based





# CONCLUSION

- La recherche sur les jeux révèlent 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

# 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
  - expectimax
- Connaître leurs propriétés théoriques
- Simuler l'exécution de ces algorithmes
- Décrire comment traiter le cas en temps réel