IFT 615 – Intelligence artificielle

Recherche pour jeux à deux adversaires

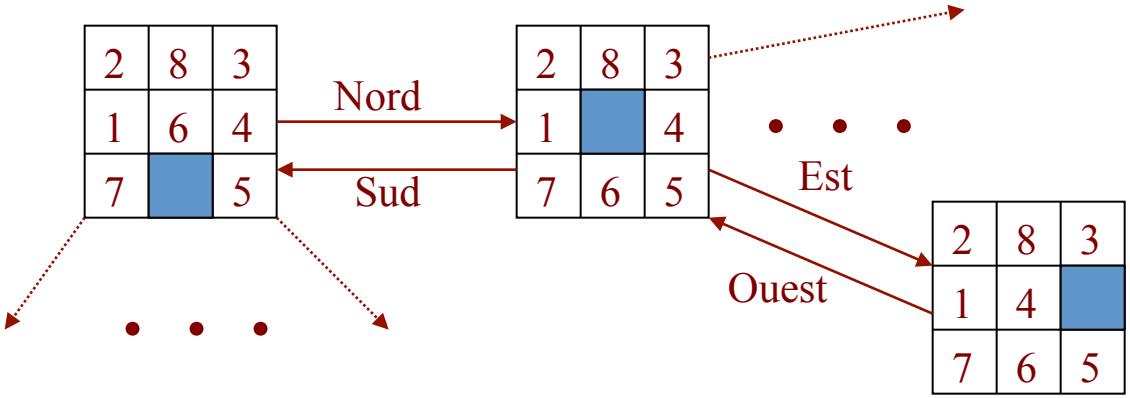
Hugo Larochelle
Département d'informatique
Université de Sherbrooke
http ://www.dmi.usherb.ca/~larocheh/cours/ift615.html

Objectifs

- Comprendre l'approche générale pour développer une IA pour un jeu à deux 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

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 !

Relation entre les joueurs

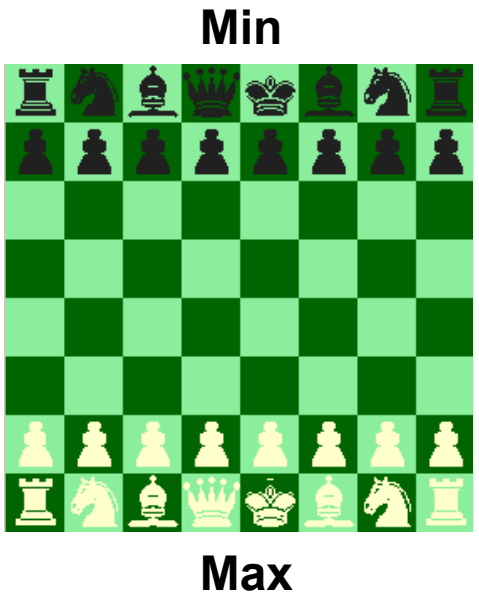
- Dans un jeu, des joueurs peuvent être :
  - **Coopératifs**
    - » ils veulent atteindre le même but
  - Des **adversaires** en compétition
    - » un gain pour les uns est une perte pour les autres
    - » cas particulier : les jeux à somme nulle (*zero-sum games*)
      - – jeux d'échecs, de dame, 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)

Hypothèses

- 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)

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

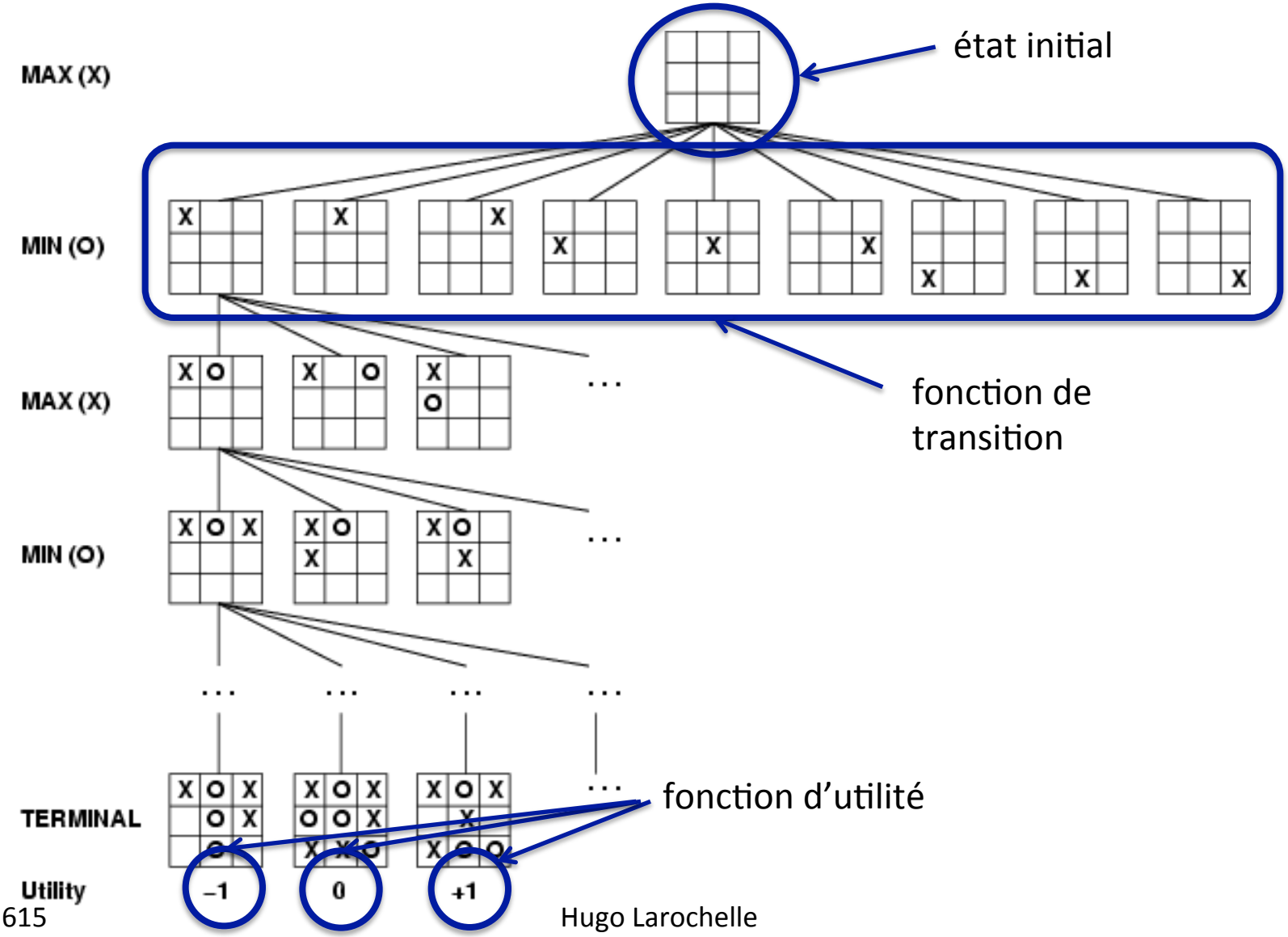


Arbre de recherche

- 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 la plus profitable pour le joueur Max ou le joueur Min, c.-à-d. qui correspond à la plus grande **valeur minimax**

$$\text{valeur-minimax}(n) = \begin{cases} \text{utilité}(n) & \text{Si } n \text{ est un nœud terminal} \\ \max_{n' \text{ successeur de } n} \text{valeur-minimax}(n') & \text{Si } n \text{ est un nœud Max} \\ \min_{n' \text{ successeur de } n} \text{valeur-minimax}(n') & \text{Si } n \text{ est un nœud Min} \end{cases}$$

- Ces équations donnent un **programme récursif** pour calculer les valeurs pour tous les noeuds dans l'arbre de recherche

Algorithme *minimax*

Algorithme minimax(*noeudInitial*)

1. retourne l'action choisie par tourMax(*noeudInitial*)

Algorithme tourMax(*n*)

1. si *n* correspond à une fin de partie, alors retourner utilité(*n*)
2. $u = -\infty$, a = void
3. pour chaque paire (a',n') donnée par transition(*n*)
    4. si l'utilité de tourMin(n') > u alors affecter $a = a'$, u = utilité de tourMin(n')
5. retourne l'utilité u et l'action a // *l'action ne sert qu'à la fin (la racine)*

Algorithme tourMin(*n*)

1. si *n* correspond à une fin de partie, alors retourner utilité(*n*)
2. $u = \infty$, a = void
3. pour chaque paire (a',n') donnée par transition(*n*)
    4. si l'utilité de tourMax(n') < u alors affecter $a = a'$, u = utilité de tourMax(n')
5. retourne l'utilité u et l'action a

Propriétés de minimax

- Complet (retourne une solution) ?
  - Oui (si l'arbre est fini)
- Optimal (retourne la meilleure solution) ?
  - Oui (contre un adversaire qui joue optimalement)

- Complexité en temps ?
  - $O(b^m)$:
    - b : le nombre maximum de coups (actions) légaux à chaque étape
    - m : nombre maximum de coups dans un jeu (profondeur maximale de l'arbre)
- Complexité en espace mémoire ?
  - O(bm), puisque **recherche en profondeur**

- Pour le jeu d'échec : b ≈ 35 et m ≈100 pour une partie « raisonnable »
  - il n'est pas réaliste d'espérer trouver une solution exacte en un temps raisonnable

- Existe-t-il des chemins dans l'arbre qui sont explorés inutilement ?

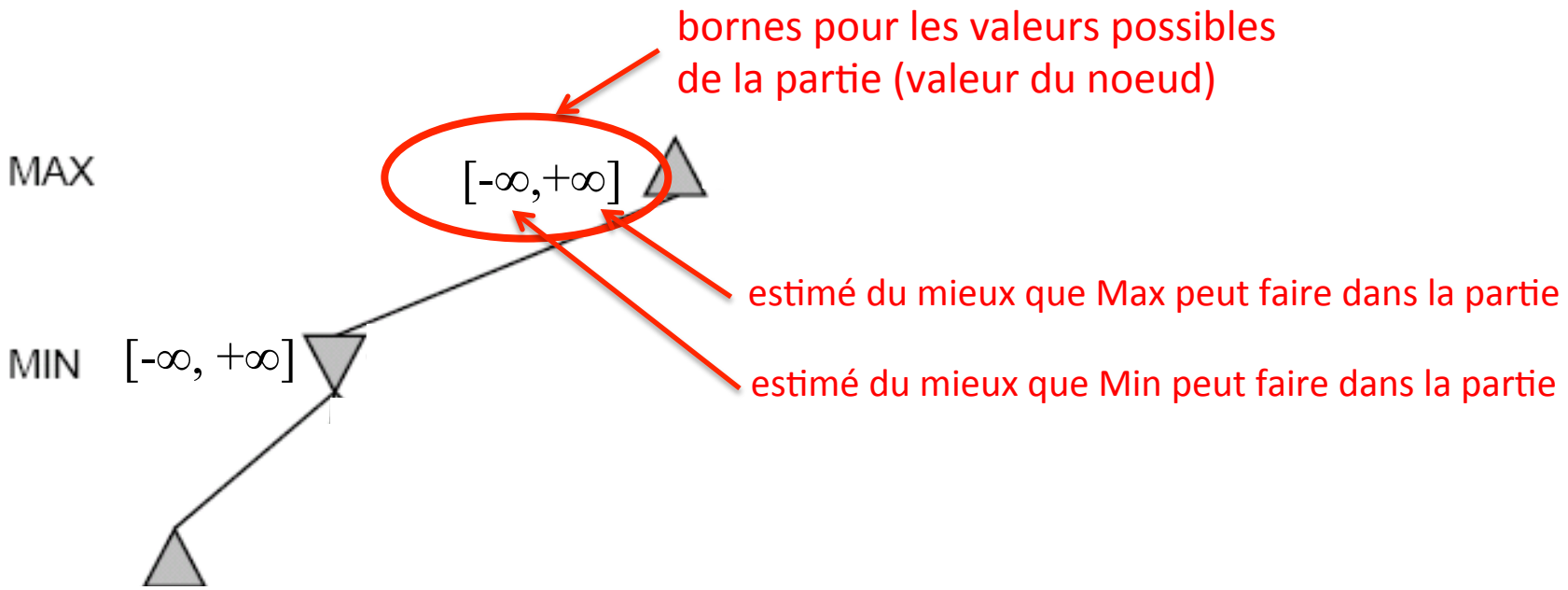
Comment accélérer la recherche

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

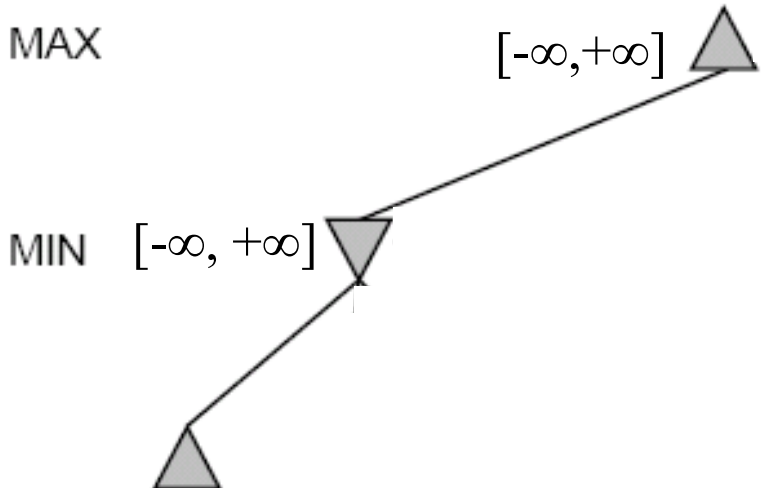
1. Élagage alpha-bêta (*alpha-beta pruning*)
  - **idée :** identifier des chemins dans l'arbre qui sont explorés inutilement

2. Couper la recherche et remplacer l'utilité par une fonction d'évaluation heuristique
  - i**dée :** 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

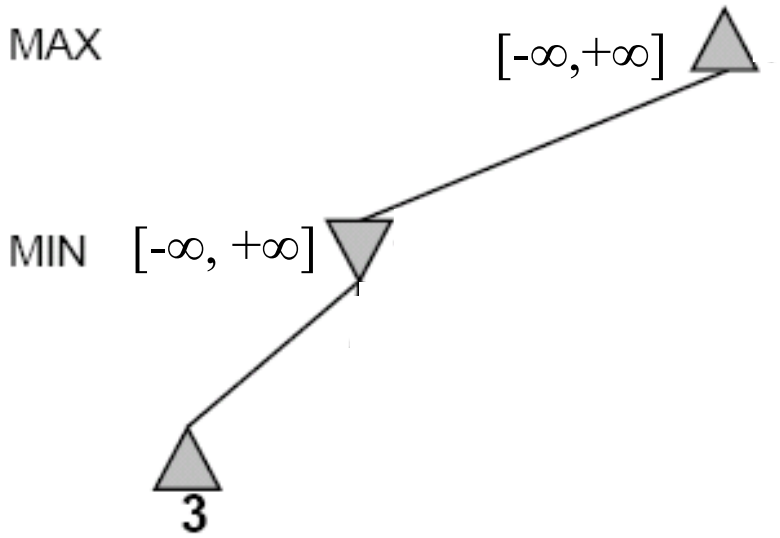
Alpha-beta pruning



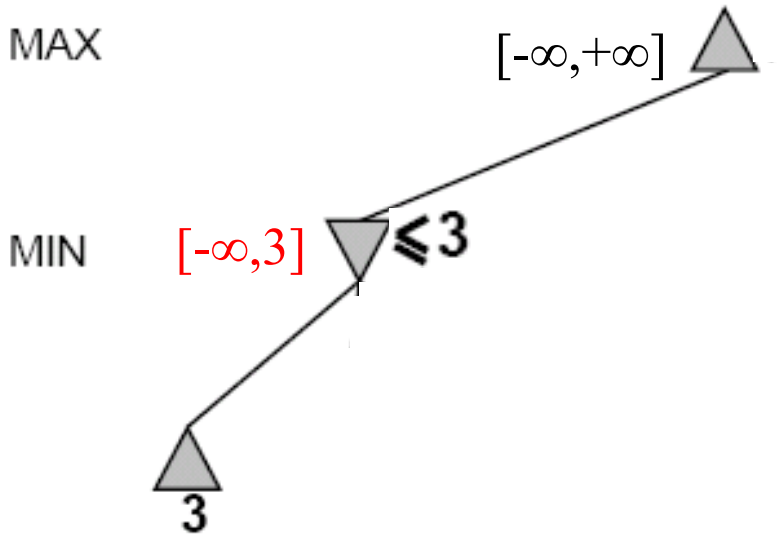
Alpha-beta pruning



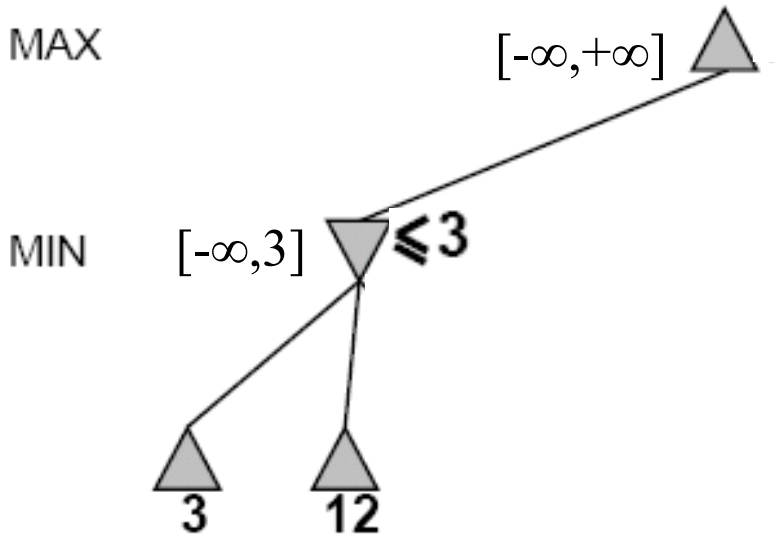
Alpha-beta pruning



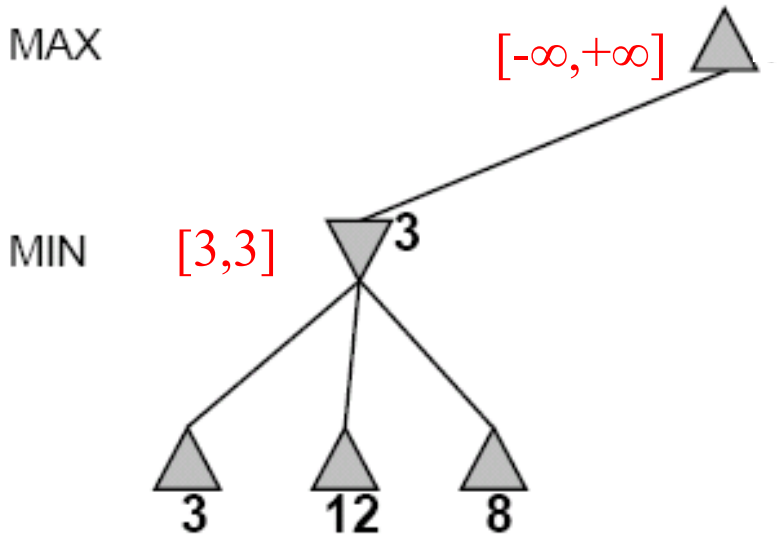
Alpha-beta pruning



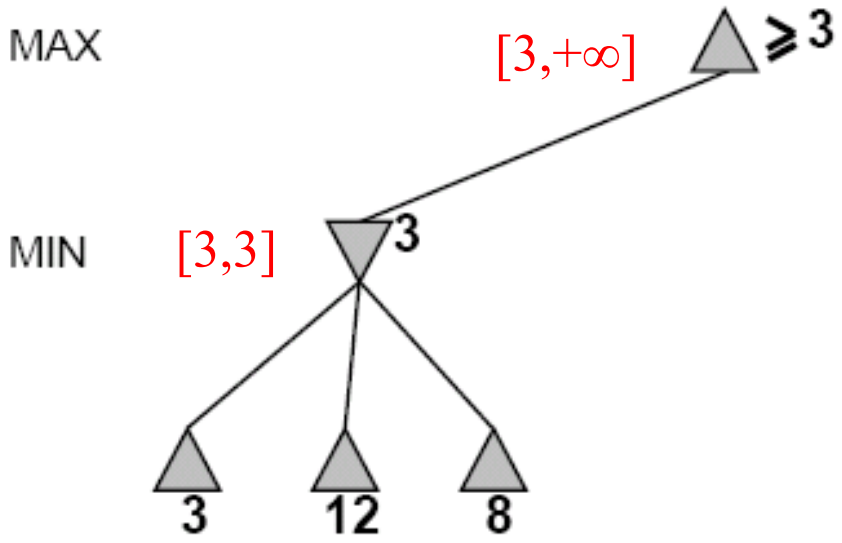
Alpha-beta pruning



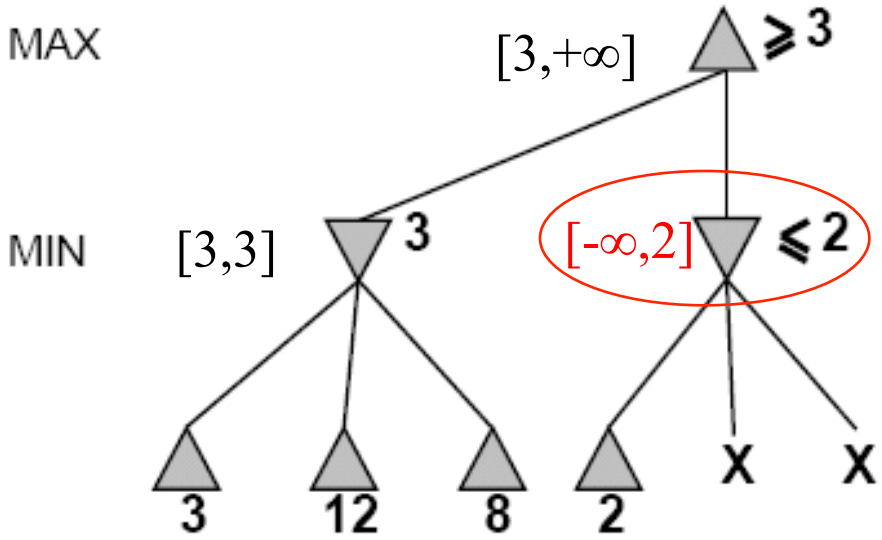
Alpha-beta pruning



Alpha-beta pruning

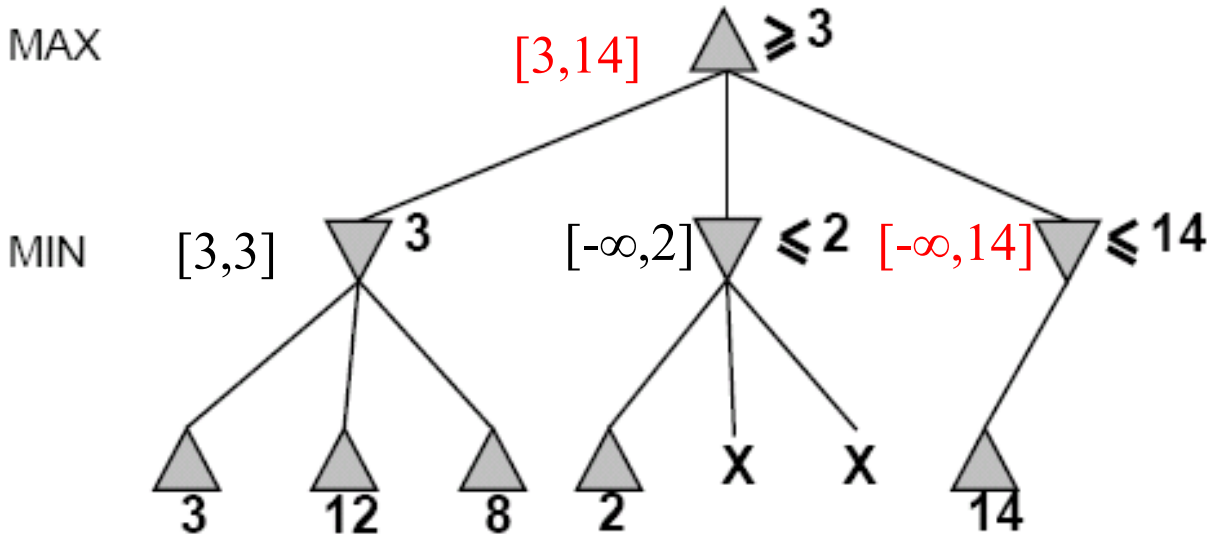


Alpha-beta pruning

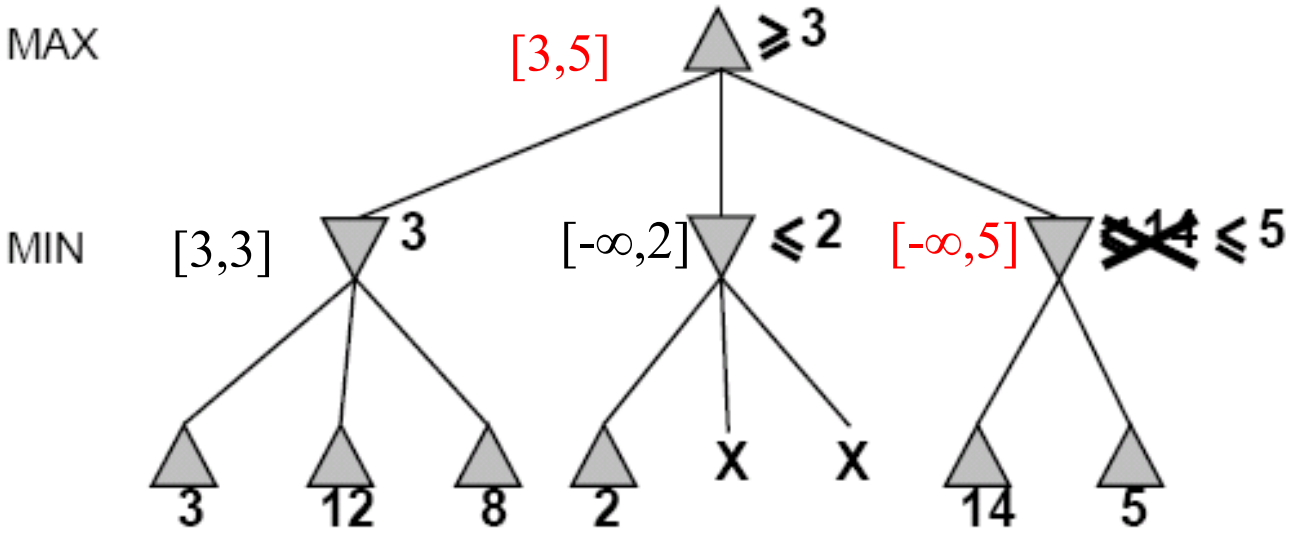


Ce noeud est pire pour Max

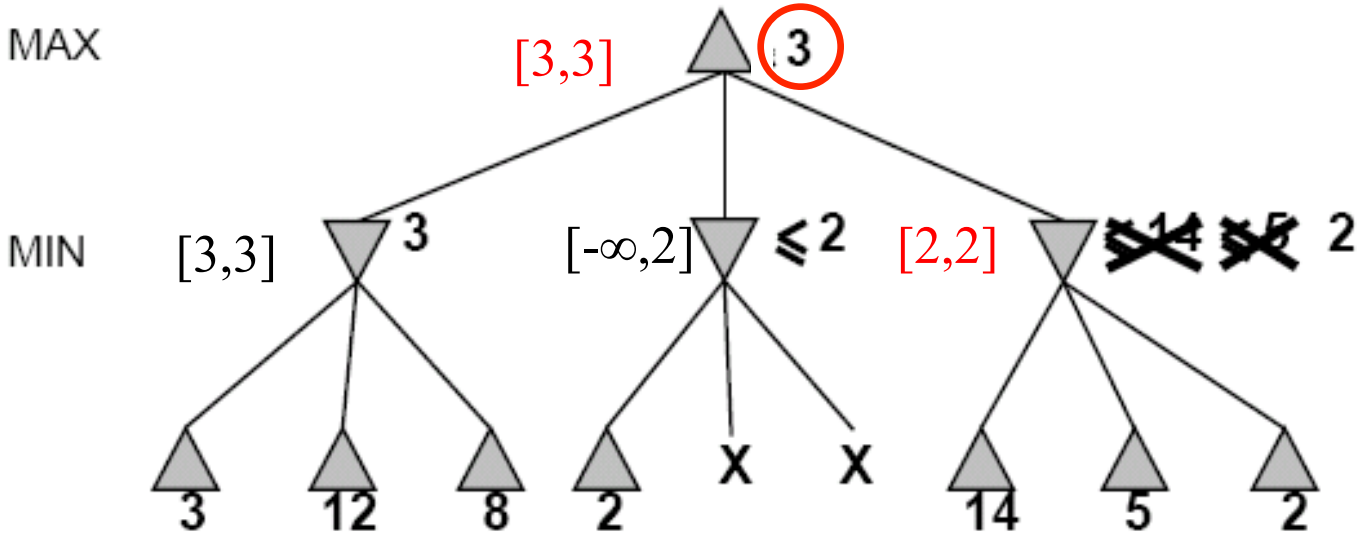
Alpha-beta pruning



Alpha-beta pruning



Alpha-beta pruning



Algorithme *élagage alpha-bêta*

Algorithme élagageAlphaBêta(*noeudInitial*)

1. retourne l'action choisie par tourMax(*noeudInitial*,-∞, ∞)

Algorithme tourMax(*n,* α, β)

1. si *n* correspond à une fin de partie, alors retourner utilité(*n*)
2. *u* = -∞, *a* = void
3. pour chaque paire (*a',n'*) donnée par transition(*n*)
    4. si l'utilité de tourMin(*n',* α, β) > *u* alors affecter *a* = *a'*, *u* = utilité de tourMin(*n',* α, β)
    5. si *u* ≥ β alors retourne l'utilité *u* et l'action *a*
    6. α = max(α, *u*)
7. retourne l'utilité *u* et l'action *a*

Algorithme tourMin(*n,* α, β)

1. si *n* correspond à une fin de partie, alors retourner utilité(*n*)
2. *u* = ∞, *a* = void
3. pour chaque paire (*a',n'*) donnée par transition(*n*)
    4. si l'utilité de tourMax(*n',* α, β) < *u* alors affecter *a* = *a'*, *u* = utilité de tourMax(*n',* α, β)
    5. si *u* ≤ α alors retourne l'utilité *u* et l'action *a*
    6. β = min (β, *u*)
7. retourne l'utilité *u* et l'action *a*

Algorithme *élagage alpha-bêta*

Algorithme élagageAlphaBêta(*noeudInitial*)

1. retourne l'action choisie par tourMax(*noeudInitial*, -∞, ∞)

Algorithme tourMax(*n*, α, β)

1. si *n* correspond à une fin de partie, alors retourner utilité(*n*)
2. *u* = -∞, *a* = void
3. pour chaque paire (*a'*,*n'*) donnée par transition(*n*)
4. si l'utilité de tourMin(*n'*, α, β) > *u* alors affecter *a* = *a'*, *u* = utilité de tourMin(*n'*, α, β)
5. si *u* ≥ β alors retourne l'utilité *u* et l'action *a*
6. α = max(α, *u*)
7. retourne l'utilité *u* et l'action *a*

Algorithme tourMin(*n*, α, β)

1. si *n* correspond à une fin de partie, alors retourner utilité(*n*)
2. *u* = ∞, *a* = void
3. pour chaque paire (*a'*,*n'*) donnée par transition(*n*)
4. si l'utilité de tourMax(*n'*, α, β) < *u* alors affecter *a* = *a'*, *u* = utilité de tourMax(*n'*, α, β)
5. si *u* ≤ α alors retourne l'utilité *u* et l'action *a*
6. β = min (β, *u*)
7. retourne l'utilité *u* et l'action *a*

ce qui a changé...

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 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 $O(b^{m/2})$
    - » si le temps de réflexion est limité, la recherche peut être jusqu'à deux fois plus profonde comparé à minimax!
  - dans le cas moyen d'un ordonnancement aléatoire : $O(b^{3m/4})$

Décisions en temps réel

- En général, des décisions imparfaites doivent être prises en temps réel :
  - supposons qu'on a 60 secs pour réagir et que l'algorithme explore 10^4 nœuds/sec
  - cela donne $6*10^5$ nœuds à explorer par coup

- 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 faisait une recherche complète
    - » on peut voir ça comme une estimation de la « chance » qu'une configuration mènera à une victoire

Exemple de fonction d'évaluation heuristique

- Pour le jeu d'échec, une fonction d'évaluation typique est une somme pondérée de *features* (caractéristiques) estimant la qualité de la configuration :

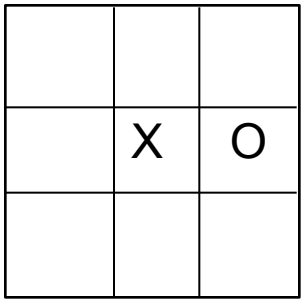
$$\text{Eval}(n) = w_1 f_1(n) + w_2 f_2(n) + \ldots + w_d f_d(n)$$

- Par exemple :
  - $w_1 = 9, f_1(n) = (\textit{number of white queens}) - (\textit{number of black queens})$
  - etc.

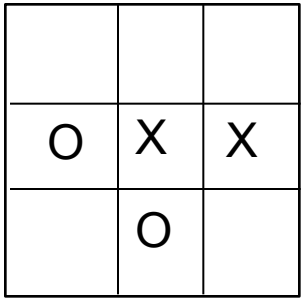
Exemple de fonction d'évaluation

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

Eval(n) = (nb. de rangées, colonnes et diagonales disponibles pour Max) –
(nb. de rangées, colonnes et diagonales disponibles pour Min)

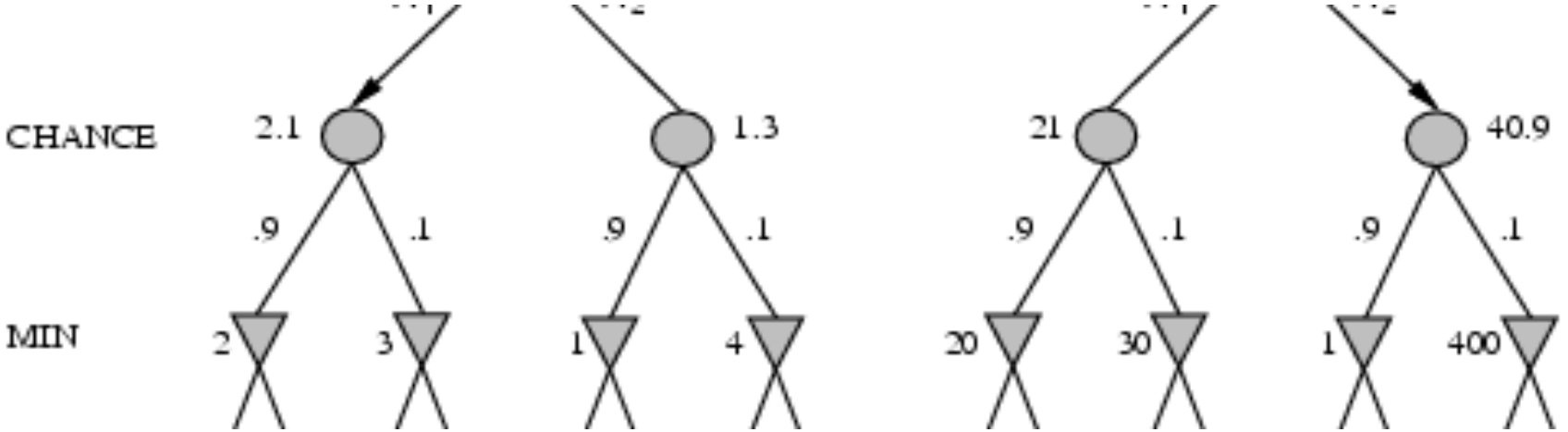


Eval(n) = 6 - 4 = 2



Eval(n) = 4 - 3 = 1

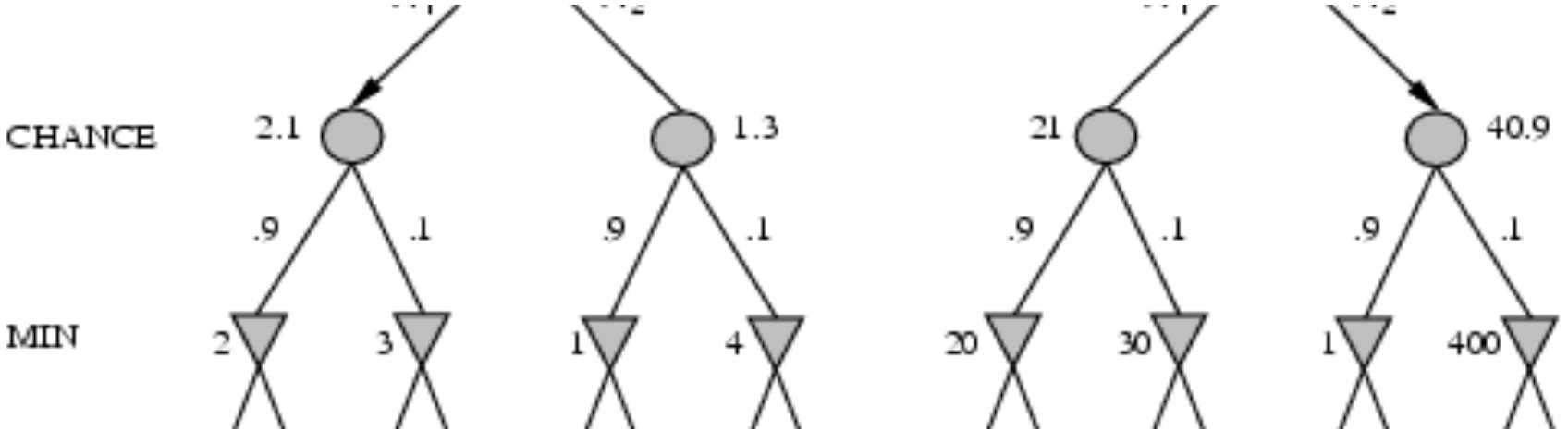
Généralisation aux actions aléatoires



- Par exemple, des jeux où on lance un dé pour déterminer la prochaine action

- **Solution :** on ajoute des noeuds chance, en plus des noeuds Max et Min
  - ces nouveaux noeuds calculs l'utilité moyenne pondérée de l'utilité de ses enfants (c.-à-d. l'utilité espérée)

Généralisation aux actions aléatoires



minimax-espérée (*n*) =

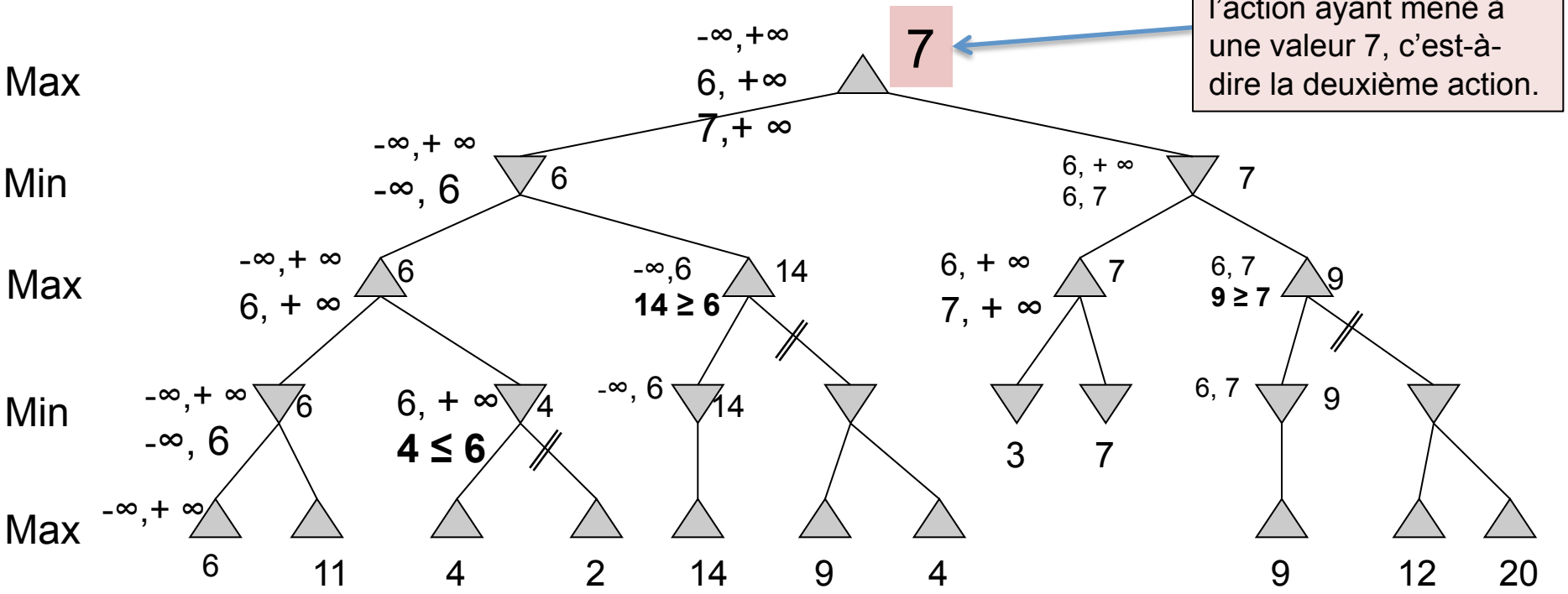
utilité(*n*)	Si *n* est un terminal
max $_{n'\text{ successeur de }n}$ minimax-espérée(*n'*)	Si *n* est un nœud Max
min $_{n'\text{ successeur de }n}$ minimax-espérée (*n'*)	Si *n* est un nœud Min
$\sum_{n'\text{ successeur de }n}$ *P(n')* * minimax-espérée(*n'*)	Si n est noeud chance

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. Chinook utilisait une base de données de coups parfaits pré-calculés pour toutes les configurations impliquant 8 pions ou moins : 444 milliards de configurations!
- **Jeu d'échecs**
  - En 1997, Deep Blue a battu le champion du monde Garry Kasparov dans un match de six parties. Deep Blue explorait 200 millions de configurations par seconde!
- **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! Dans le jeu GO, le facteur de branchement (b) dépasse 300! La plupart des programmes utilisent des bases de règles empiriques pour calculer le prochain coup.

Autre exemple : Question #2 – 2009H

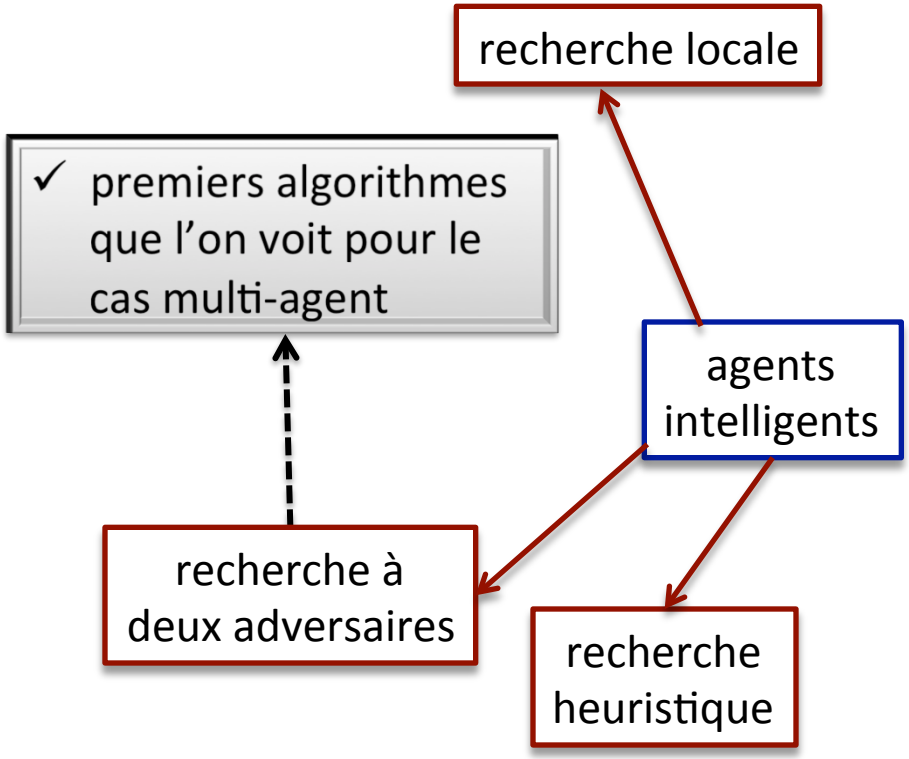


Légende de l'animation

- Nœud de l'arbre pas encore visité
- Nœud en cours de visite (sur pile de récursivité)
- Nœud visité
- Arc élagué (pruning)
- **α, β** Valeur retournée
- Valeur si feuille

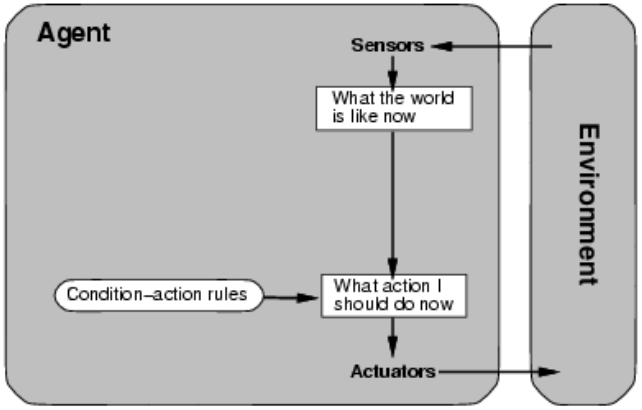
Objectifs du cours

Algorithmes et concepts

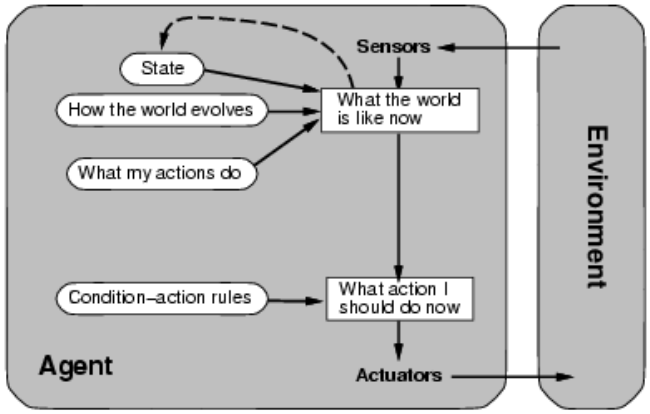


Recherche pour jeux à deux adversaires : pour quel type d'agent?

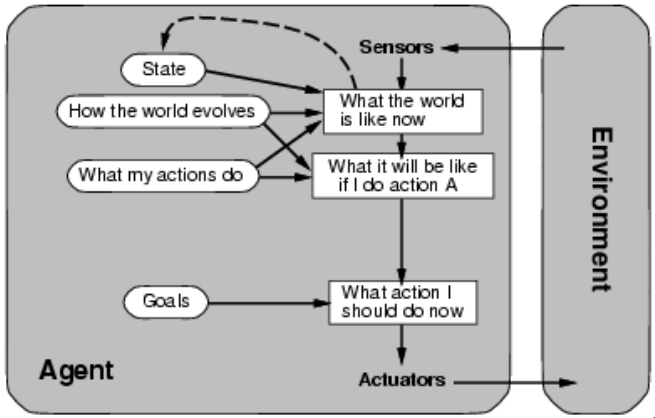
Simple reflex



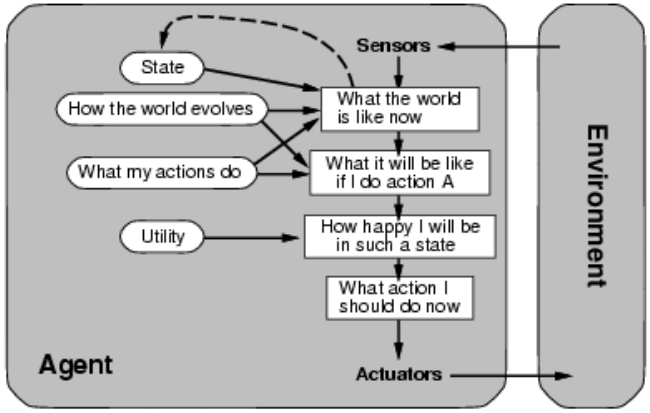
Model-based reflex



Goal-based



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