Jeux « parfaits » à 2 joueurs

Algorithmes MINIMAX

et

Monte-Carlo Tree Search

Cours Intelligence Artificielle HMIN107 – Marie-Laure Mugnier

Les problèmes de jeux

- Un des défis favoris de l'IA
  - 1997 : Deep Blue bat Gary Kasparov aux échecs

  - 20 ans plus tard : AlphaGo bat Lee Sedol au go

Qu’y a-t-il de particulier dans les jeux ?

- Deux joueurs : on ne peut pas prévoir ce que fera l'adversaire
  - Il faut donc élaborer une stratégie visant à gagner quels que soient les choix de l'adversaire
  - Une stratégie n'est pas un chemin de l'arbre de recherche, mais un sous-arbre de l'arbre de recherche dont toutes les feuilles sont gagnantes
- Problème : la combinatoire
  - Les jeux intéressants sont trop compliqués pour envisager une solution exhaustive
    - Échecs : branchement ≈35 et hauteur (max) ≈100

Différents types de jeux

	Déterministe	Avec hasard
Information complète	morpion, dames, échecs, othello, go	backgammon, monopoly
Information incomplète	mastermind	bridge, poker, scrabble

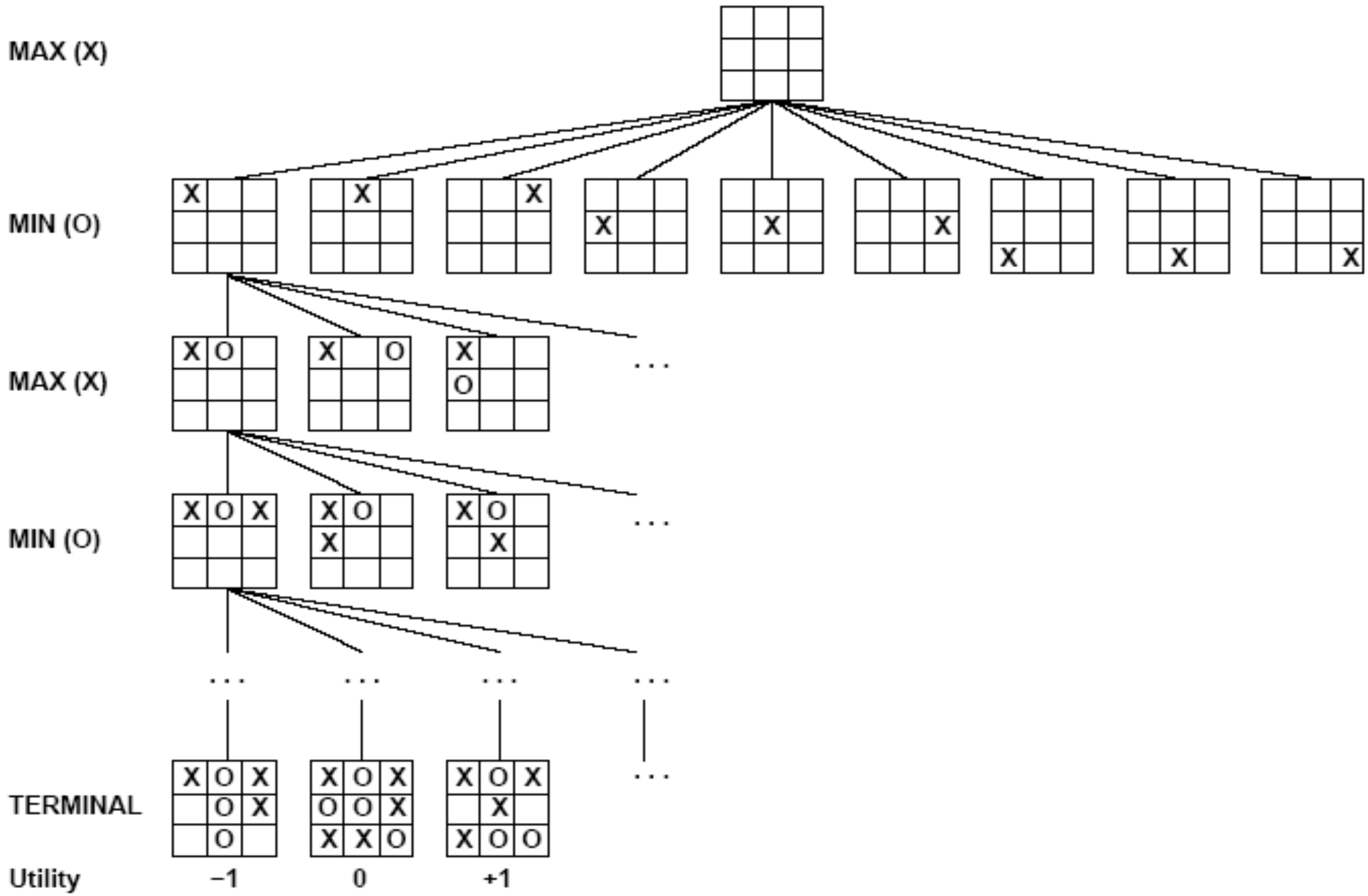
- On se limite ici aux jeux parfaits à 2 joueurs
  - Déterministes, complets, à tour, et « but de gain »
  - Les joueurs : Max et Min (Max commence)

Exploration d'un espace des états

- Etat
  - une situation de jeu : description du plateau de jeu
  - une indication du tour (« à qui le tour »)
- Etat initial
  - description de la situation de départ, Max commence
- Actions
  - déterminées par les règles du jeu
  - définissent une fonction « successeur » sur les états
    - la fonction successeur permute les tours des joueurs
- Test d'état but : *« échec et mat »* par exemple
- Fonction score qui indique pour un joueur et un état terminal si la partie est gagnée (1), perdue (-1), nulle (0)

La somme des fonctions score des 2 joueurs pour un même état terminal est 0

L'exemple du morpion



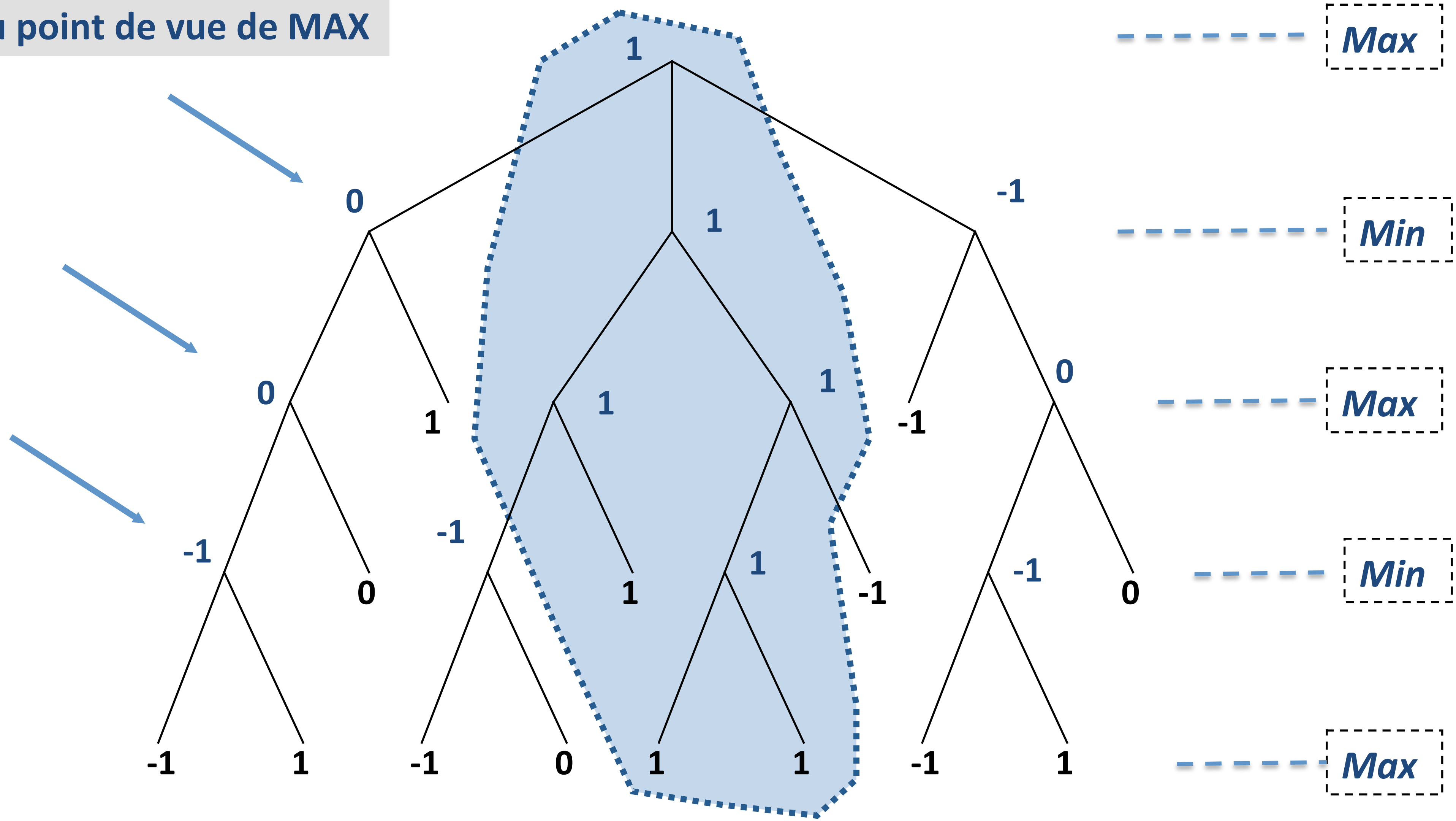
Algorithme Minimax

(Shannon 1950)

- Idée : « mon adversaire joue au mieux »
  - MAX choisit l'action qui maximise son gain en supposant que MIN choisit l'action qui minimise le gain de MAX
  - Stratégie optimale si les 2 joueurs jouent parfaitement
- Technique :
  - On explore l'arbre jusqu'aux feuilles
  - On calcule le score aux feuilles
  - On propage les scores vers la racine pour permettre le choix du « meilleur » sous arbre
    (et donc du prochain coup à jouer)

Exemple de Minimax sur un arbre complet

Les scores sont vus du point de vue de MAX



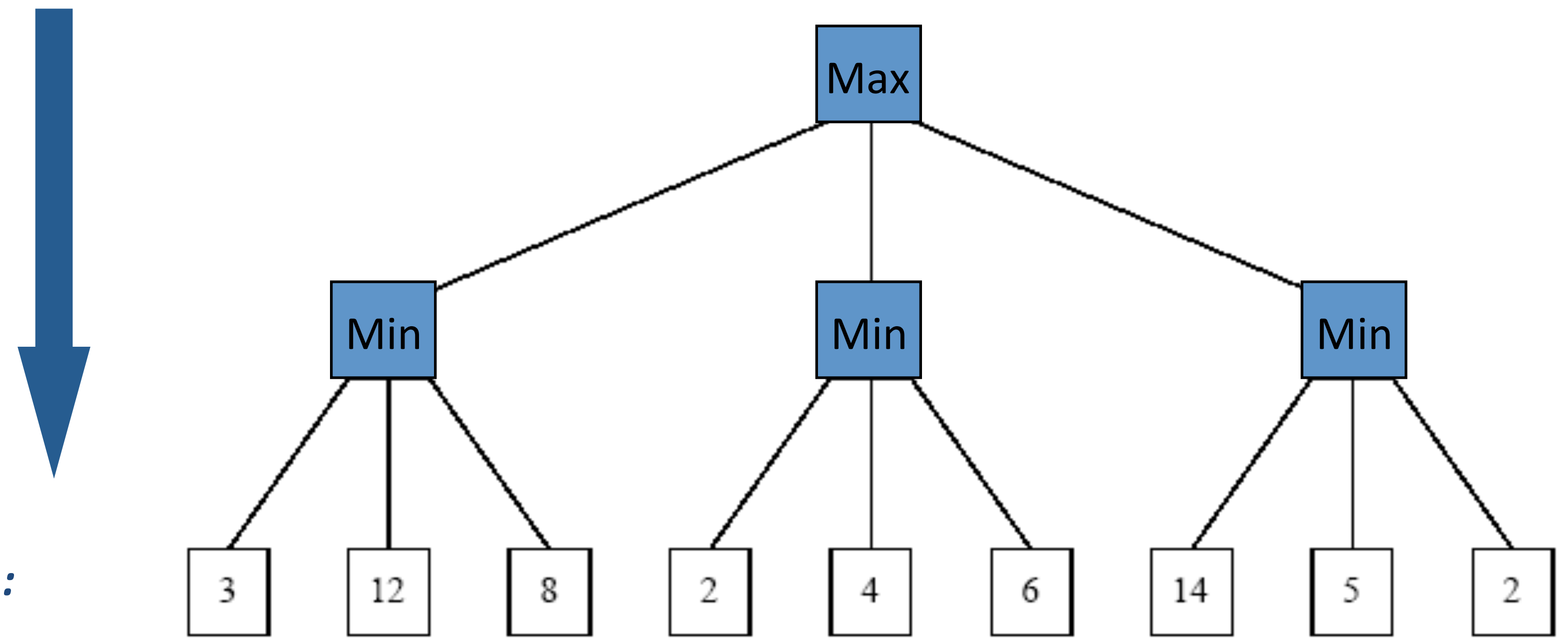
Prise en compte de la combinatoire

- Minimax repose sur un calcul de l'arbre complet
  - Impossible sur les jeux « intéressants »

    ≈ 10^{120} parties d'échec (et ≈ 10^{761} parties de go !)
- Solutions
  - Introduire une profondeur maximale d'exploration de l'arbre : un horizon
  - Disposer d'une fonction d'évaluation statique d'un état du jeu

Exemple de Minimax avec évaluation et horizon=2

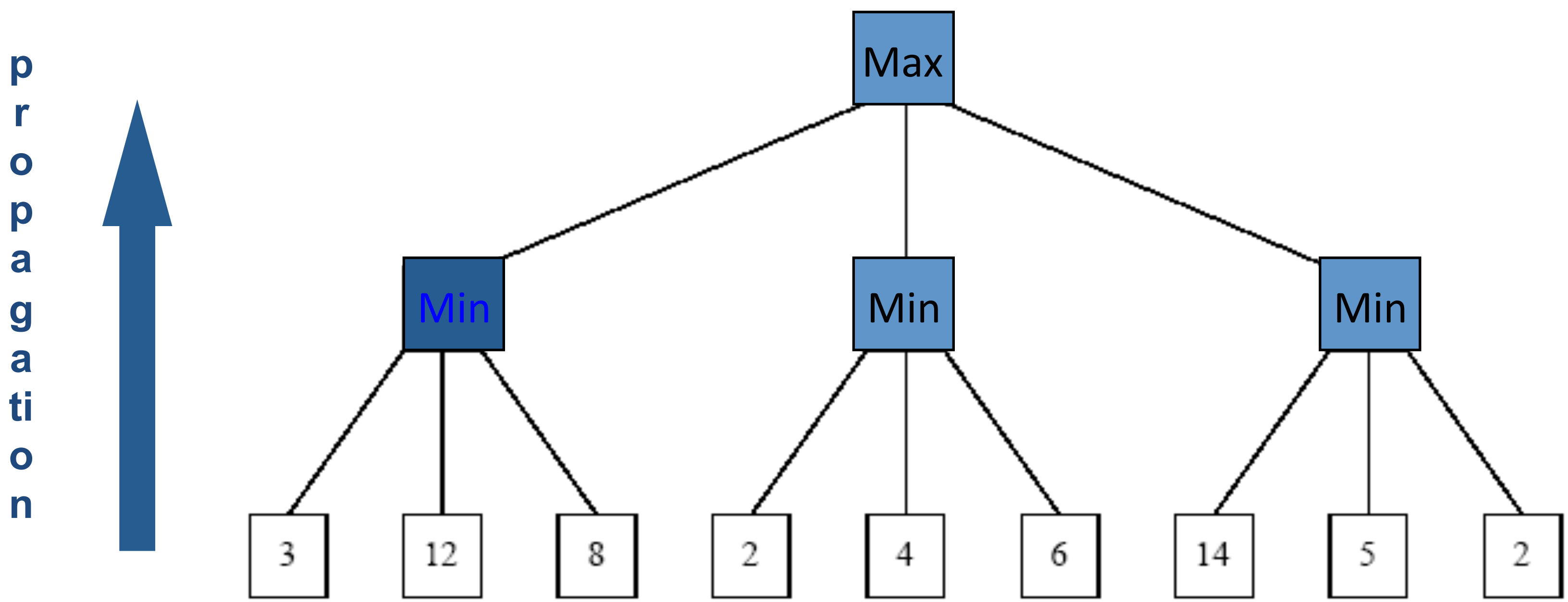
exploration



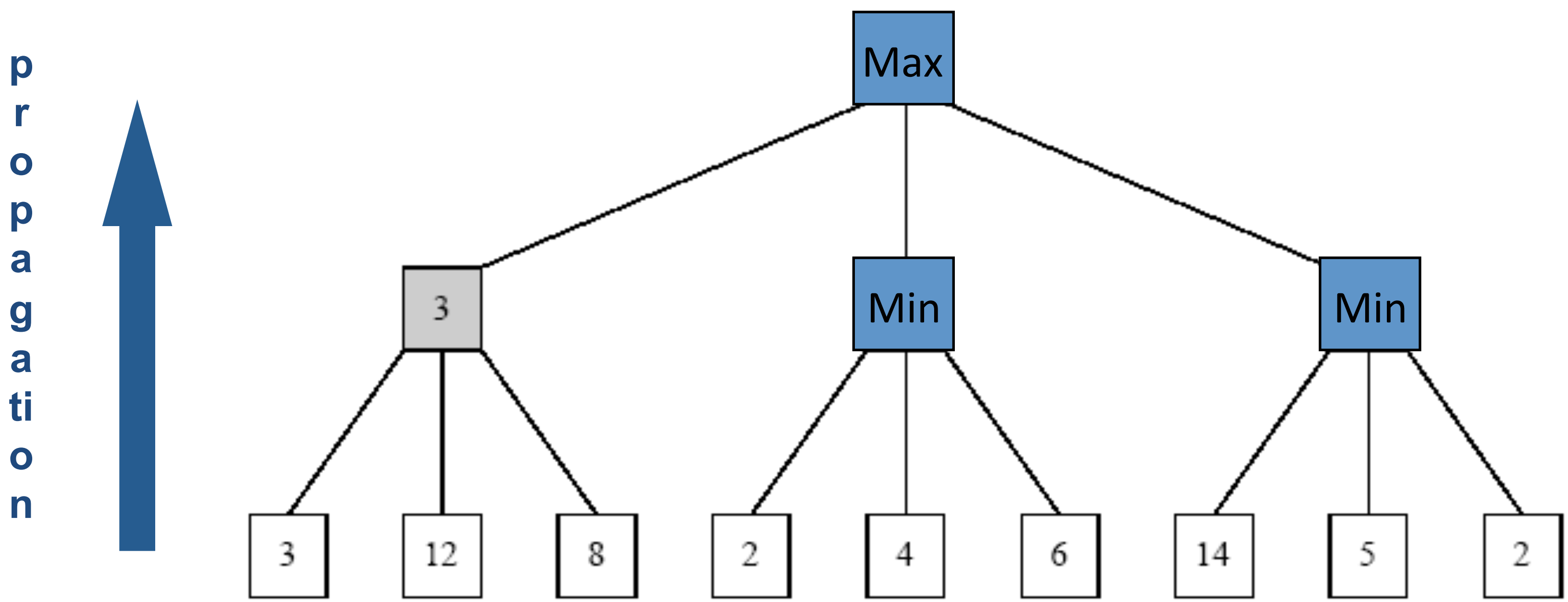
évaluation :

- précise si fin de partie
- sinon estimation de la configuration atteinte

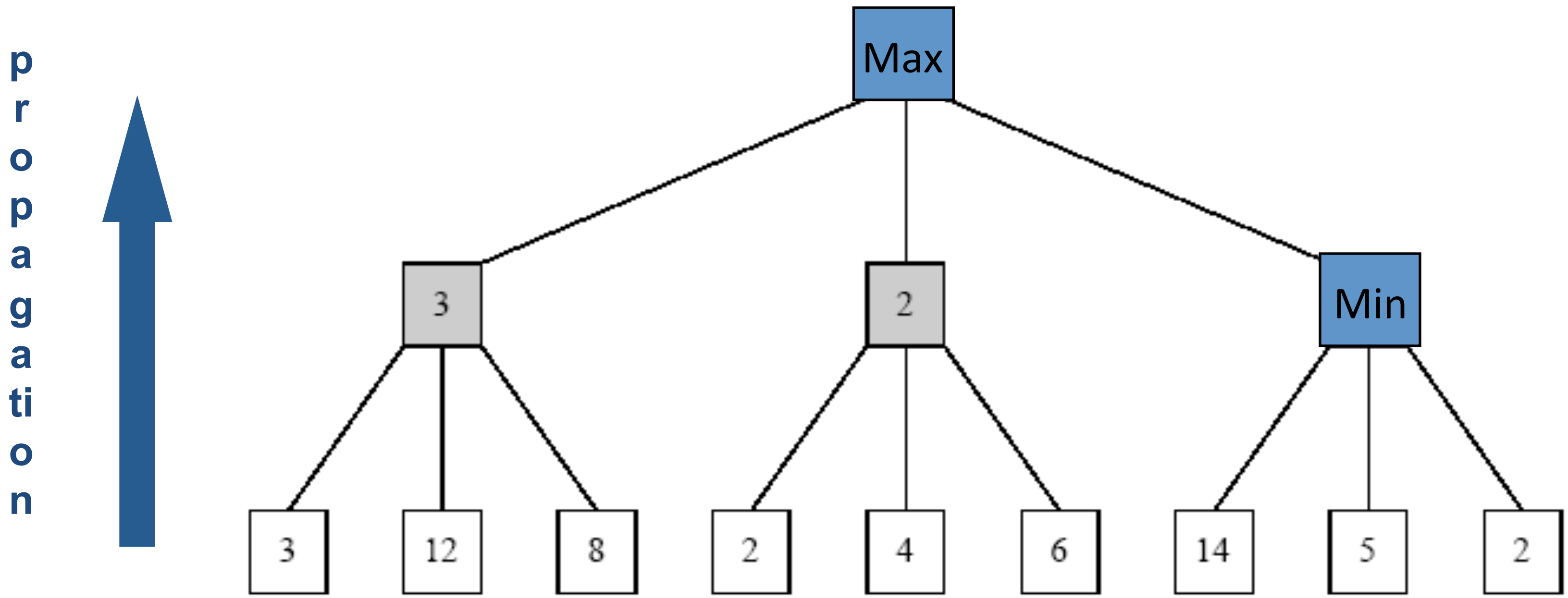
Exemple de Minimax avec évaluation et horizon=2



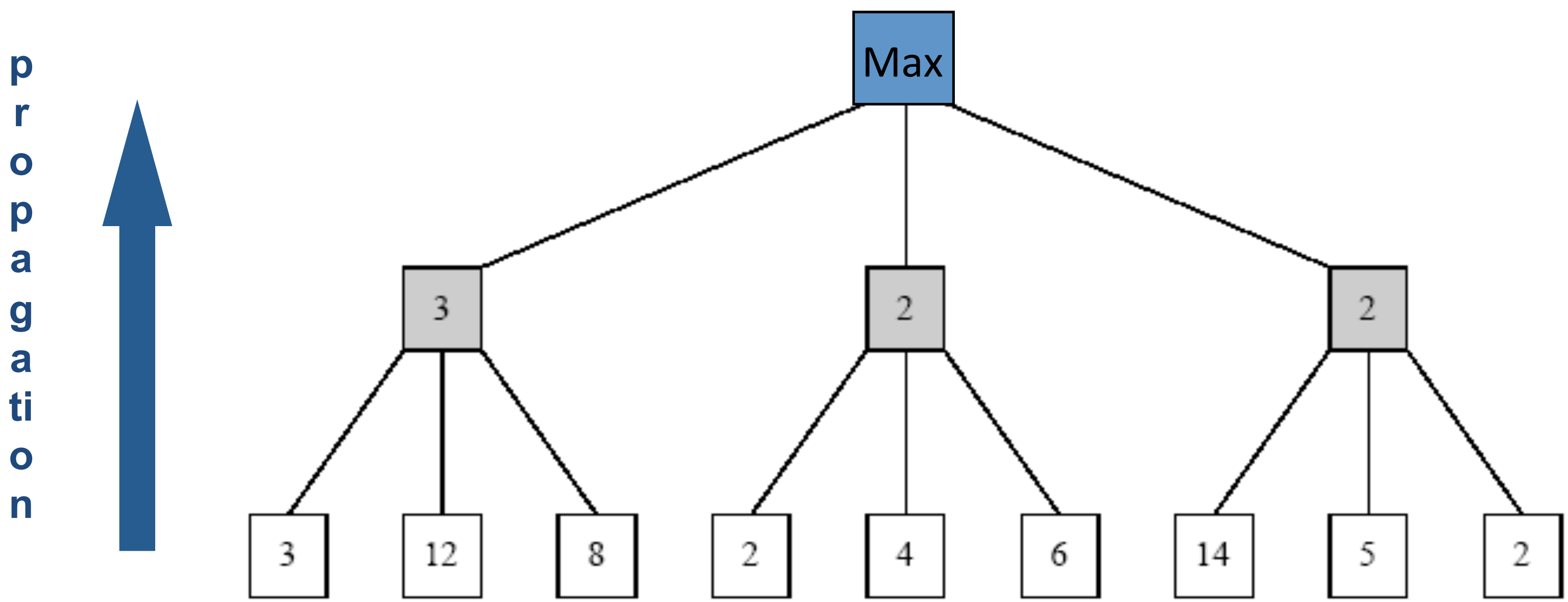
Exemple de Minimax avec évaluation et horizon=2



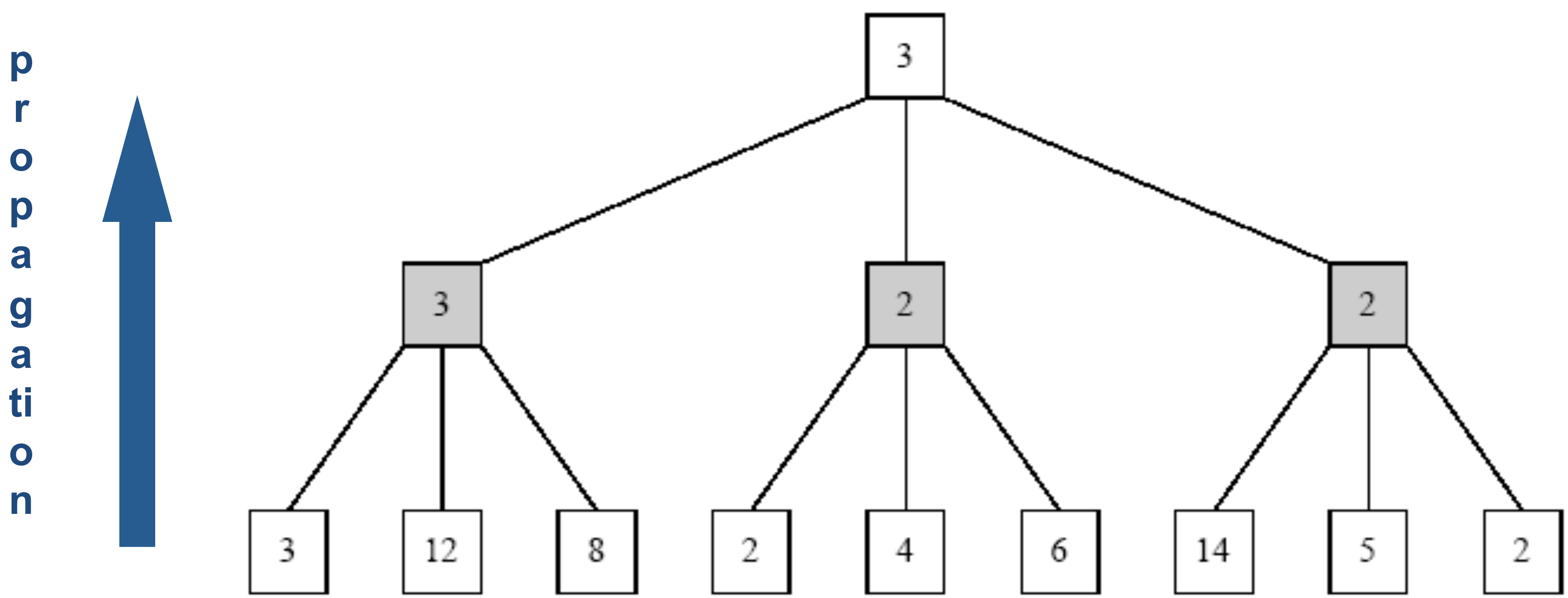
Exemple de Minimax avec évaluation et horizon=2



Exemple de Minimax avec évaluation et horizon=2



Exemple de Minimax avec évaluation et horizon=2



Principe de Minimax

Faire remonter à la racine une valeur calculée récursivement

- minimax(e) = eval (e) si e est une feuille
    - « vraie » feuille (fin de partie)
    - feuille car l'horizon est atteint

- minimax(e) = max (minimax(e1) ...minimax(ek)) si e est un noeud « MAX »
    - où e1 ... ek sont les fils de e

- minimax(e) = min (minimax(e1) ...minimax(ek)) si e est un noeud « MIN »
    - où e1 ... ek sont les fils de e

L'algorithme Minimax (action)

```
Fonction  JouerMinimax  (Etat e-courant, Horizon h) : Action
Début
    valOpt ← -∞          // meilleur score trouvé
    Pour chaque action a possible à partir de e-courant
    |    val ← Minimax(Appliquer(a,e-courant), h)
    |    Si val > valOpt
    |    |    valOpt ← val
    |    |    opt ← a
    |    Finsi
    FinPour
    Retourner opt        // un meilleur coup à jouer
Fin;
```

L'algorithme Minimax (score)

```
Fonction Minimax (Etat e, Horizon h) : Valeur
Début
    Si e est terminal, retourner (score(e) x ∞)  // calcul sûr          OK si score = -1, 0, +1
    Si h=0, retourner eval(e)                    // estimation
    Si e.tour=Max
    |       val ← -∞;
    |       Pour chaque action a possible à partir de e
    |           val←Max(val, Minimax(Appliquer(a,e), h-1);
    |       FinPour
     Sinon      // e.tour=Min
    |       val ← +∞;
    |       Pour chaque action a possible à partir de e
    |           val←Min(val, Minimax(Appliquer(a,e), h-1);
    |       FinPour
     Finsi
    Retourner val
Fin
```

Propriétés de Minimax

- Complétude (trouve un coup gagnant s'il existe)
  - Non dès lors que h < profondeur max de l'arbre
- Optimale (trouve un meilleur coup)
  - Non pour les mêmes raisons

- Complexité en temps

  $O(b^{h+1})$ [b : branchement ; h : horizon = profondeur explorée]
- Complexité en espace

  O(b x h) si les successeurs sont gardés en mémoire (file)

  O(h) si un seul successeur est généré à la fois

Fonction d'évaluation et horizon

- Nécessite d’identifier les critères de gain
  → des heuristiques
- Souvent exprimée comme une somme pondérée de critères

$$e = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$$

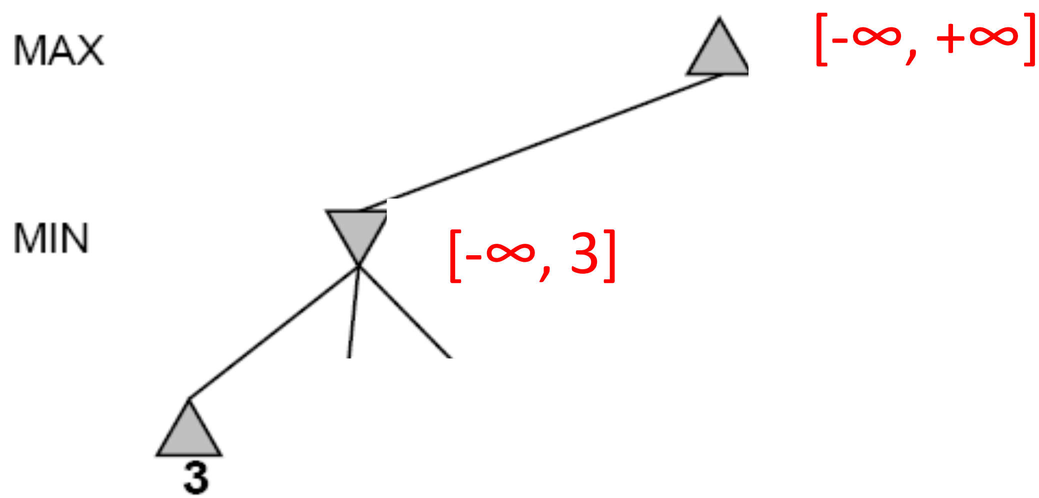
- Exemple des échecs
  – nombre de pièces restantes de chaque type pondéré par le poids des pièces (*reine>tour>…> pion*)
- L'horizon détermine la force du joueur virtuel
  – Pour les échecs
    - h=4 ≈ joueur novice
    - h=8 ≈ joueur « maître »
    - h=12 + base de « situation/coup à jouer » ≈ deep blue

Amélioration de Minimax : $\alpha\beta$-pruning

- Idée : tirer parti au maximum des informations obtenues lors de la recherche en profondeur
  - Lorsqu'une branche remonte une valeur *m* pour Max, on continue la recherche avec cette information :
    - Dès qu'un nœud Min calcule une valeur ≤ *m* il n'est pas utile de poursuivre l'exploration de ce noeud Min
      - La valeur du nœud Min sera ≤ m
      - La branche choisie par Max sera ≥ m
  - Même idée en inversant Min, Max et ≤

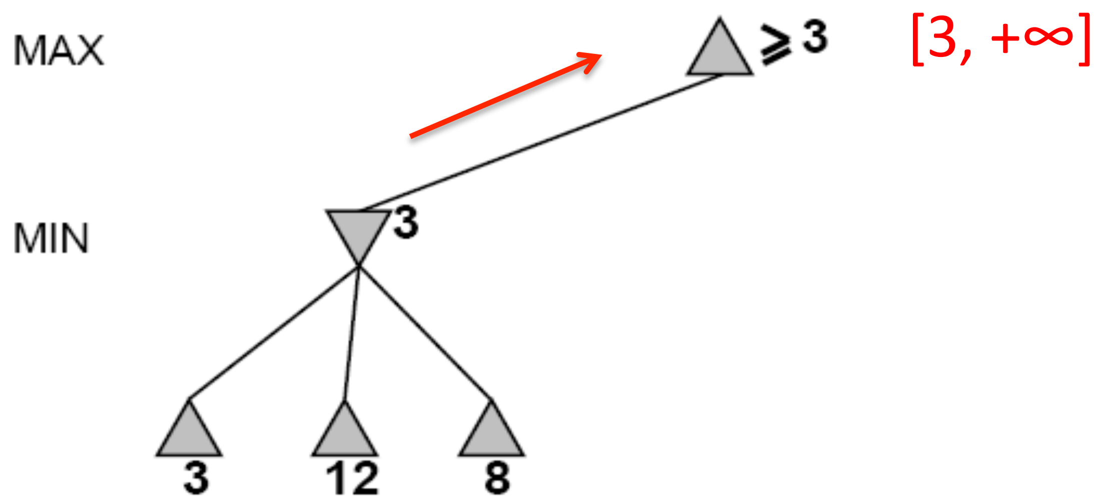
Sur un exemple…

A chaque noeud, intervalle [α,β] avec α = minimum assuré pour MAX
β = maximum que peut espérer MAX



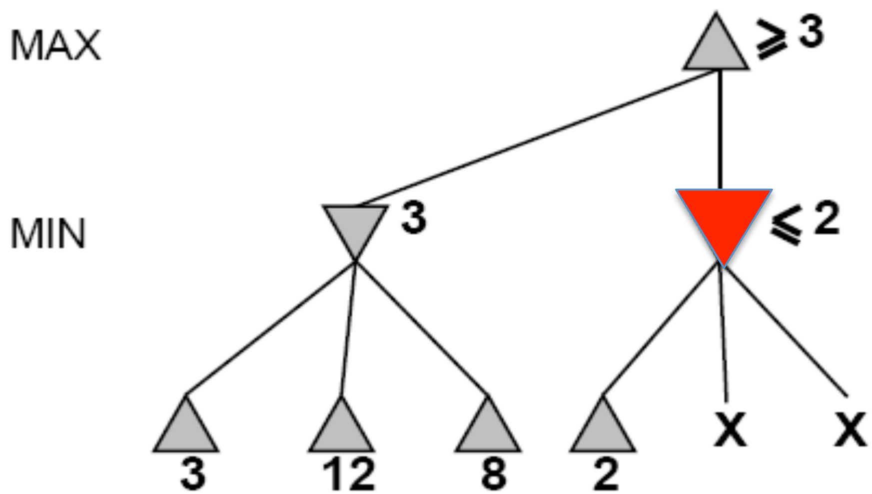
Sur un exemple…

A chaque noeud, intervalle [α,β] avec α = minimum assuré pour MAX
β = maximum que peut espérer MAX



Sur un exemple…

A chaque noeud, intervalle [α,β] avec α = minimum assuré pour MAX
β = maximum que peut espérer MAX

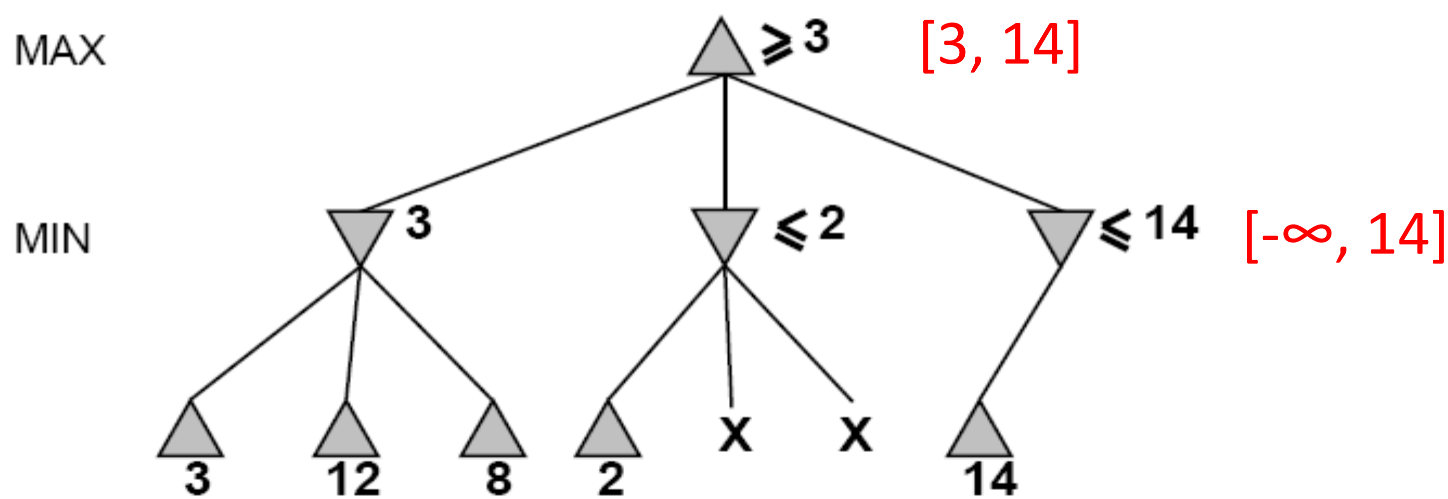


[3, +∞]

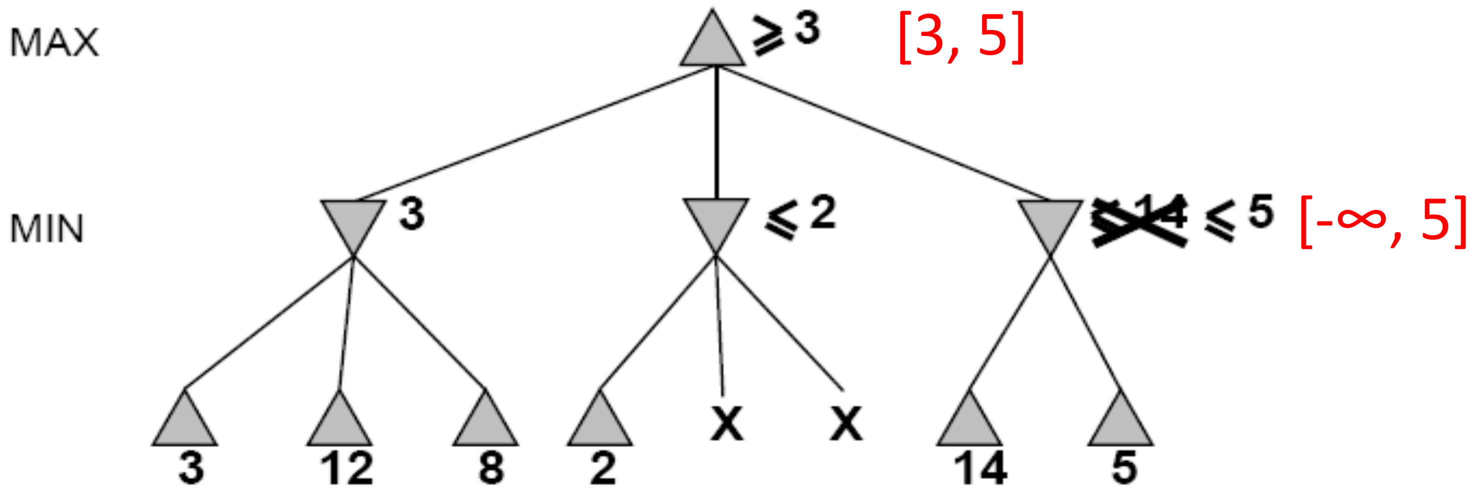
Le père transmet α = 3

Ce noeud Min fixerait
$\beta \leq \alpha$
Max ne choisira
jamais ce noeud
donc inutile
de continuer à l'explorer

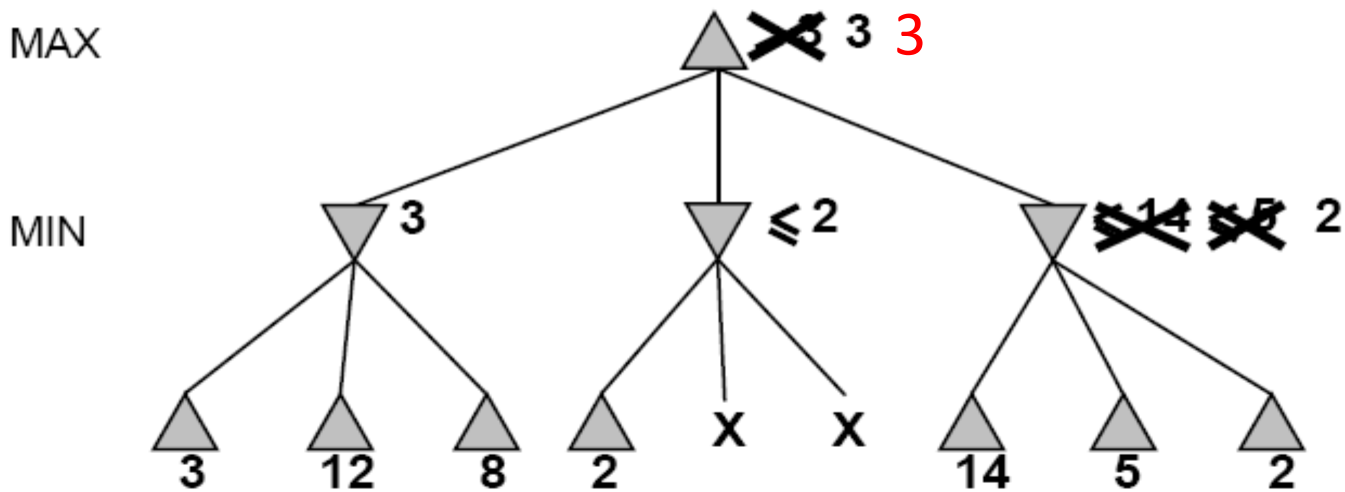
Sur un exemple…



Sur un exemple…

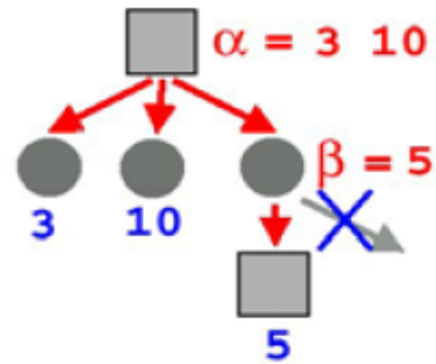


Sur un exemple…

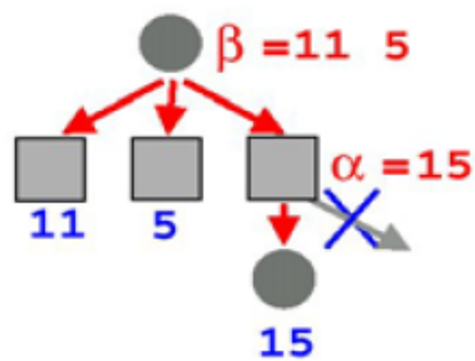


Les règles de l'élagage α-β

- Sur un nœud Min, interrompre la recherche si sa β-valeur calculée $\leq$ α-valeur transmise par son nœud père → son père Max préfèrera β



- Sur un nœud Max, interrompre la recherche si son α-valeur calculée $\geq$ β-valeur transmise par son nœud père → son père Min préfèrera β



L'algorithme αβ-pruning (McCarthy 1963)

<u>Fonction</u> Joueurαβ (Etat e-courant, Horizon h) : Action
<u>Début</u>
    valOpt ← -∞ *// meilleur score trouvé*
    <u>Pour chaque</u> action ***a*** possible à partir de e-courant
        val ← Calculerαβ(Appliquer(a,e-courant), h,-∞,+∞)
        <u>Si</u> val > valOpt
            valOpt ← val
            opt ← a
        <u>Finsi</u>
    <u>FinPour</u>
    <u>Retourner</u> opt *// un meilleur coup à jouer*
<u>Fin</u>

```
Fonction  Calculerαβ ( Etat e, Horizon h, α, β) : Valeur
Début
    Si e est terminal alors retourner (score(e) x ∞)
    Si h=0, retourner eval(e)
    Si e.tour=Max
        Pour chaque action a possible à partir de e
            α←Max(α, Calculerαβ(Appliquer(a,e), h-1, α, β)
            Si α ≥ β alors retourner β
        FinPour
        retourner α
    Sinon           // e.tour=Min
        Pour chaque action a ∈ j.Actions avec a possible à partir de e faire
            β←Min(β, Calculerαβ(Appliquer(a,e),h-1 , α, β)
            Si α ≥ β alors retourner α
        FinPour
        retourner β
    Finsi
Fin
```

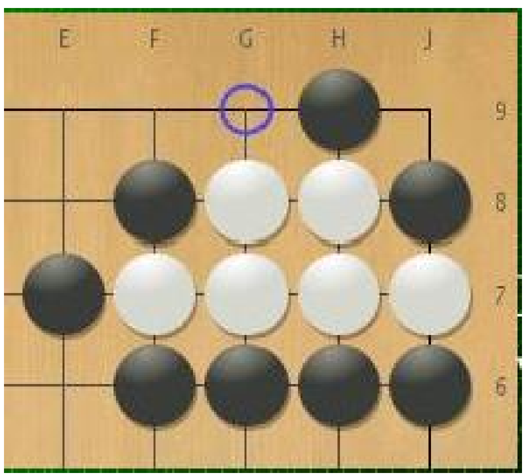
Propriétés

- L'élagage n'affecte pas le résultat final
  - Preuve basée sur l'invariant : $\forall x \ \alpha(x) \leq \beta(x)$
- L'ordre dans lequel les actions sont étudiées est important car il influe sur l'élagage
- Résultats
  - Othello : facile
  - Dames : on sait résoudre
  - Echecs : on arrive à une profondeur > 14 avec une bonne fonction d'évaluation
  - **Go : ??**

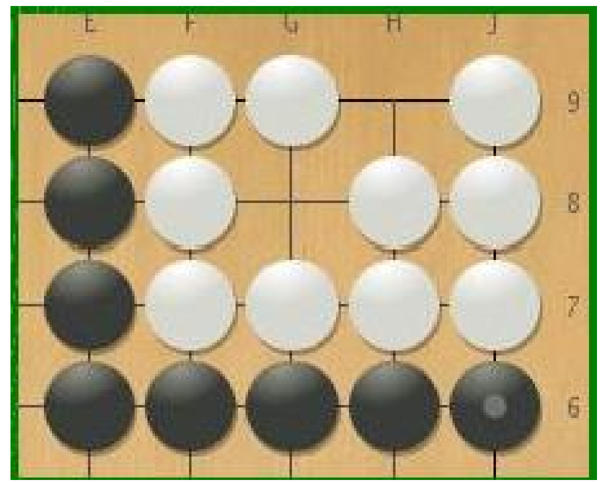
The game of Go in one slide

Rules

- Each player puts a stone on the goban, black first
- Each stone remains on the goban, except:



group w/o degree freedom is killed



a group with two eyes can't be killec

- The goal is to control the max. territory

[Source : M. Sebag]

Difficultés du jeu de go pour Minimax

1. Facteur de branchement gigantesque

   19 x 19 = 361 coups possibles au départ
   En moyenne, environ 250 coups possibles

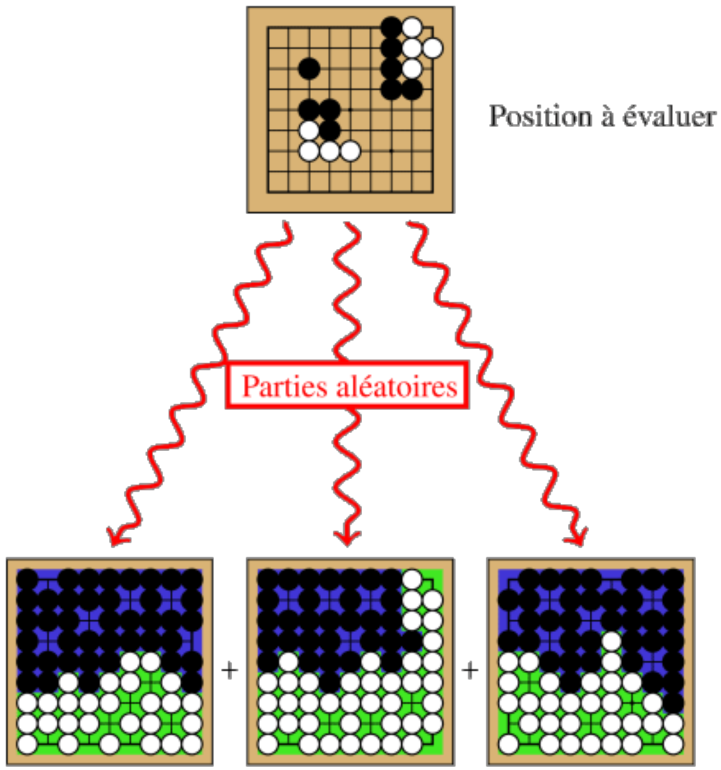
2. Pas de bonne fonction d'évaluation

Des situations qui semblent très similaires
peuvent avoir des résultats très différents
Difficile d'évaluer la sûreté d'une pierre et la taille des territoires

→ Pendant quasiment 20 ans après Deep Blue,
aucun programme d'IA n'a réussi à « bien jouer » au Go

jusqu'à 2016 : AlphaGo contre le champion du monde Lee Sedol
(4 matchs gagnés sur 5)

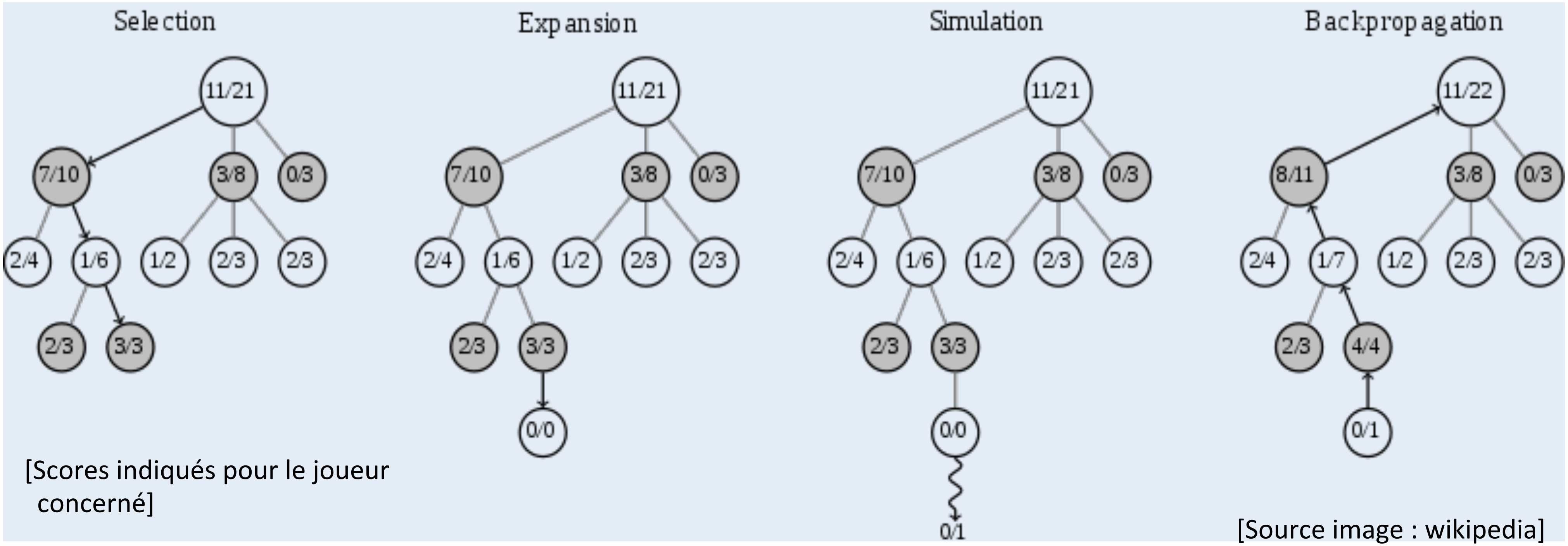
Idée de base : « aller jusqu'au bout d'une partie ... beaucoup de fois »
puis choisir le coup qui donne le plus de parties gagnées



Mais ne pas faire ces simulations indépendamment les unes des autres
→
utiliser les résultats des simulations pour guider la construction d'un arbre de jeu suivant un compromis entre :
- se focaliser sur les coups prometteurs
- se focaliser sur les coups où l'estimation est très incertaine

Monte-Carlo Tree Search (Rémi Coulom, 2006
concepteur de Crazy Stone, 1er programme de go avec MCTS)

Monte Carlo Tree Search: algorithme



[Scores indiqués pour le joueur concerné]

[Source image : wikipedia]

Itérer

Sélection : partant de la racine, descendre jusqu'à un noeud *x* à expandre prioritairement
Expansion : ajouter un fils *y* à ce noeud *x*
Simulation : depuis y, jouer une partie jusqu'au bout par mouvements aléatoires (ou + ou - guidés)
Backpropagation : propager l'information *gagné/perdu* en remontant jusqu'à la racine pour mettre à jour les scores des noeuds (*nombre parties gagnées / nombre de parties jouées*)

Jusqu'à fin des ressources disponibles

Retourner le fils de la racine qui a le meilleur score

MCTS : propriétés

Plus le nombre de simulations est grand, plus le résultat se rapproche de celui de minimax

- Intérêts
  - algorithme générique applicable à n'importe quel jeu
  - pas besoin de fonction d'évaluation
  - concentre l'exploration sur des sous-arbres « prometteurs »
    donc particulièrement bien adapté lorsque le facteur de branchement est très grand
  - algorithme anytime : peut être interrompu n'importe quand

- Inconvénients
  - Quand il y a très peu de chemins menant à une victoire, ceux-ci peuvent ne pas être vus

    (ex. le match perdu par AlphaGo contre Sedol ?)