Min-Max et Alpha-Beta :

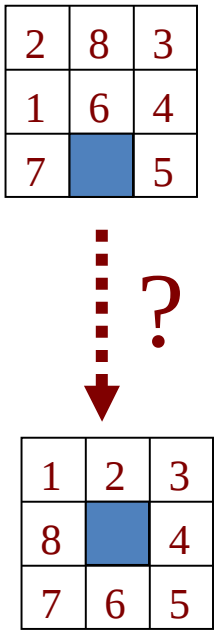
Algorithmes de recherche pour des jeux avec des adversaires

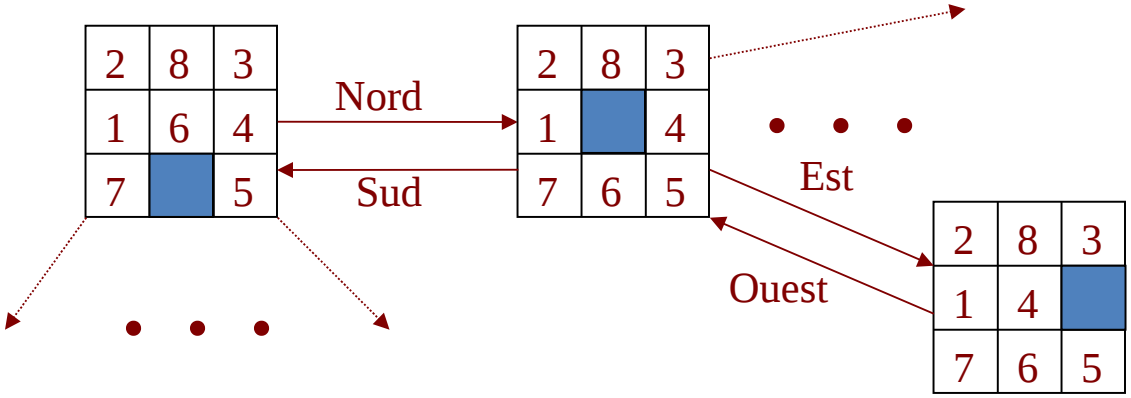
Objectifs

- Se familiariser avec :
  - Jeux entre deux adversaires
  - Algorithme minimax
  - Élagage alpha-beta
  - Décisions imparfaites en temps réelle
- Travail pratique
  - Implémentation du jeu TicTacToc

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, le monde n'est pas statique, mais est dynamique. Le joueur adverse peut modifier l'environnement.

- Q : Comment peut-on résoudre ce problème ?

R : Algorithme Min-Max et Alpha-Beta

Relation entre les joueurs

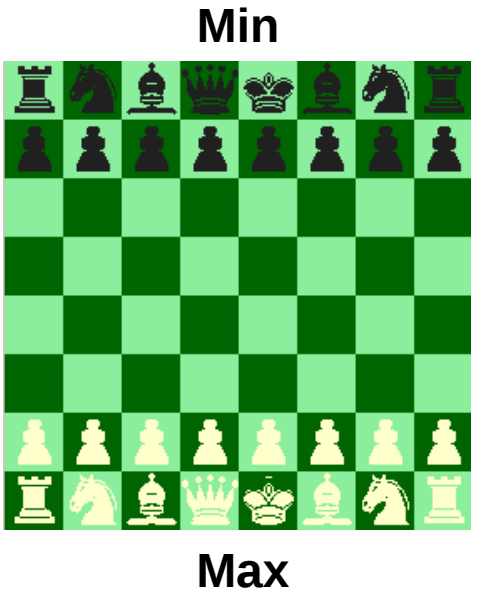
- Dans un jeu, des joueurs peuvent être :
  - Coopératifs
    - Ils veulent atteindre le même but.
  - **Des adversaires en compétition**
    - Un gain pour les uns, est une perte pour les autres.
    - Cas particulier : les jeux à somme nulle (zero-sum games).
      - Jeux d’échecs, de dame, tic-tac-toe, etc.

Hypothèses

- Nous aborderons les :
  - Jeux à deux adversaires
  - Jeux à tour de rôle
  - Jeux à somme nulle
  - Jeux avec observation totale
  - Jeux déterministes (sans hasard ou incertitude)
- Il sera possible de généraliser

Jeux entre deux adversaires

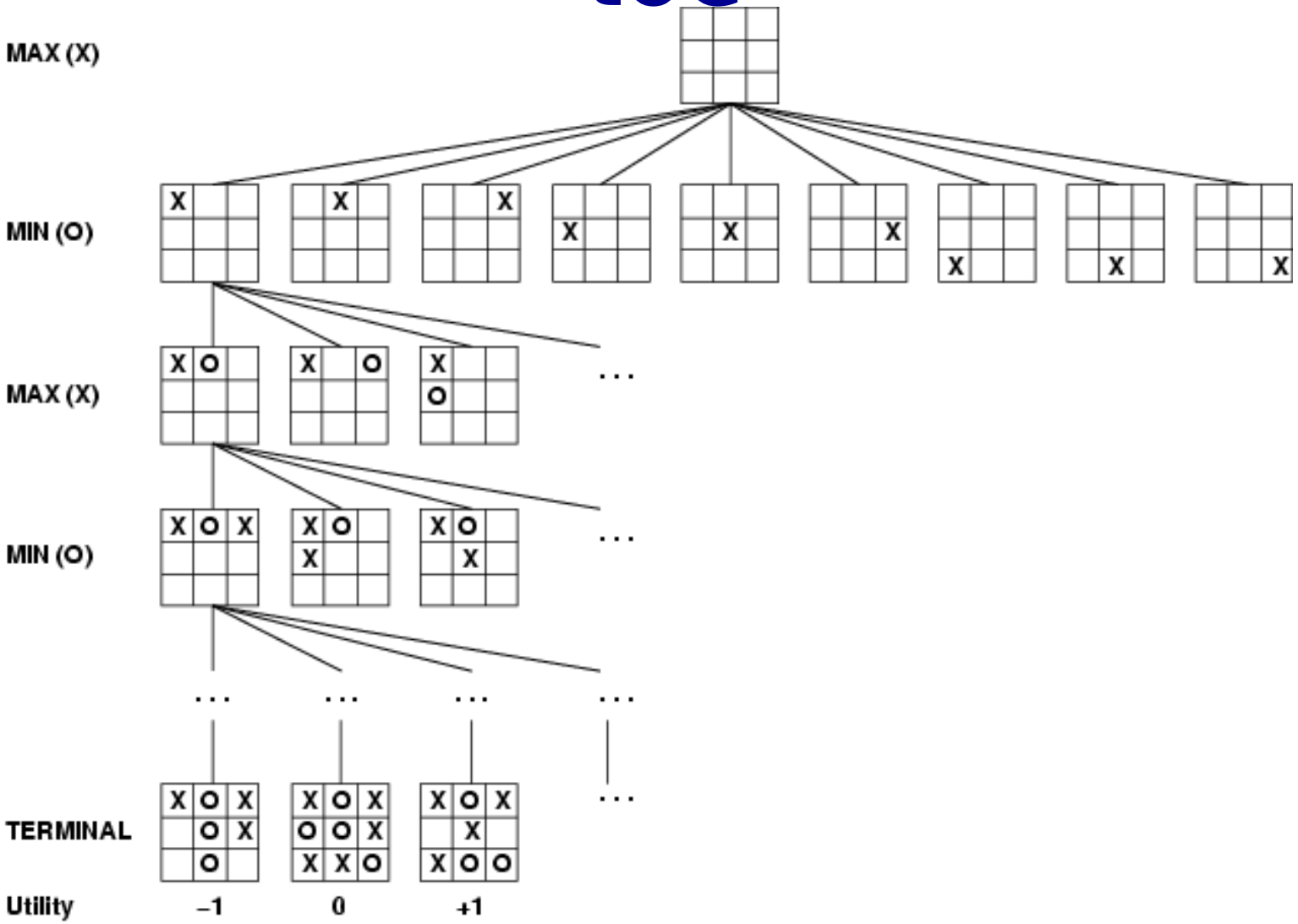
- Noms des joueurs : **Max** *vs* **Min**
  - **Max** est le premier à jouer (notre joueur)
  - **Min** est son adversaire



Arbre de recherche

- Un problème de jeu peut être vu comme un problème de recherche dans un arbre :
  - Un état (configuration) initial
  - Un ensemble d'opérateurs (coups légaux ou actions légales)
    - Ces opérateurs génèrent des transitions dans l'arbre
  - Un test de terminaison
    - Indique si le jeu est terminé
  - Une fonction d'utilité pour les états finaux

Arbre de recherche tic-tac-toe



Algorithme minimax

- Idée : à chaque tour, choisir l'action qui correspond à la plus grande valeur minimax

EXPECTED-MINIMAX-VALUE(n) =

UTILITY(n) Si n est un nœud terminal

$\max_{s \in successors(n)}$ EXPECTED-MINIMAX-VALUE(s) Si n est un nœud Max

$\min_{s \in successors(n)}$ EXPECTED-MINIMAX-VALUE(s) Si n est un nœud Min

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

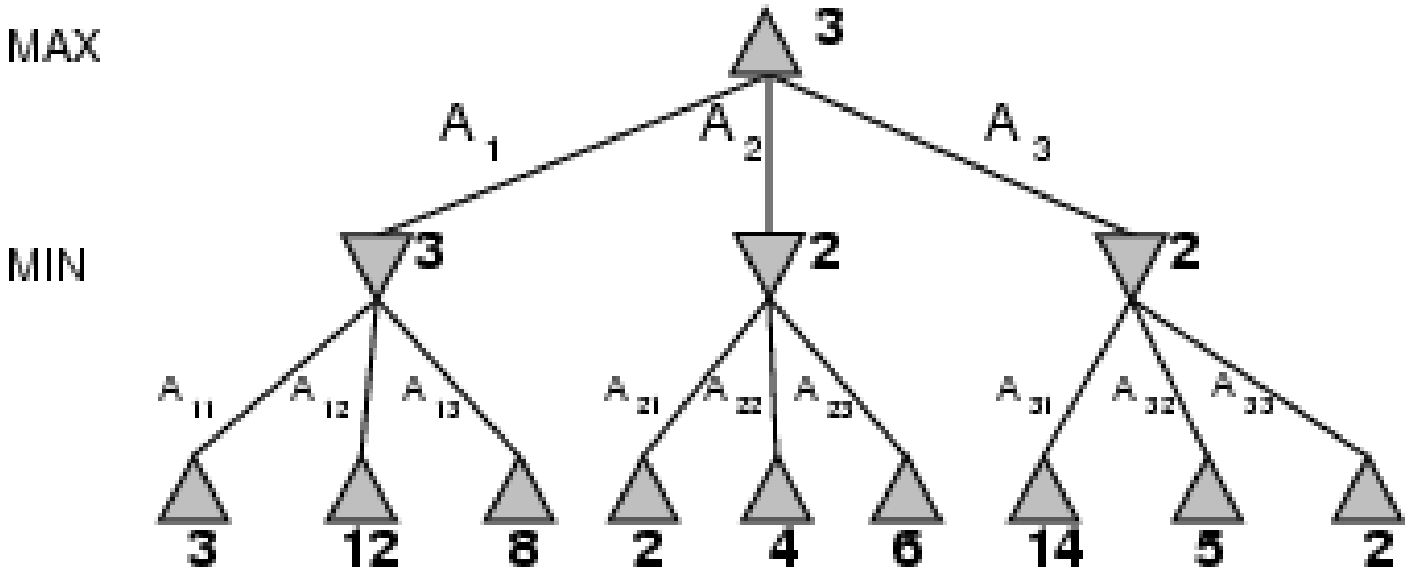
Algorithme minimax

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(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 ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

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.

- Ex



Propriétés de minimax

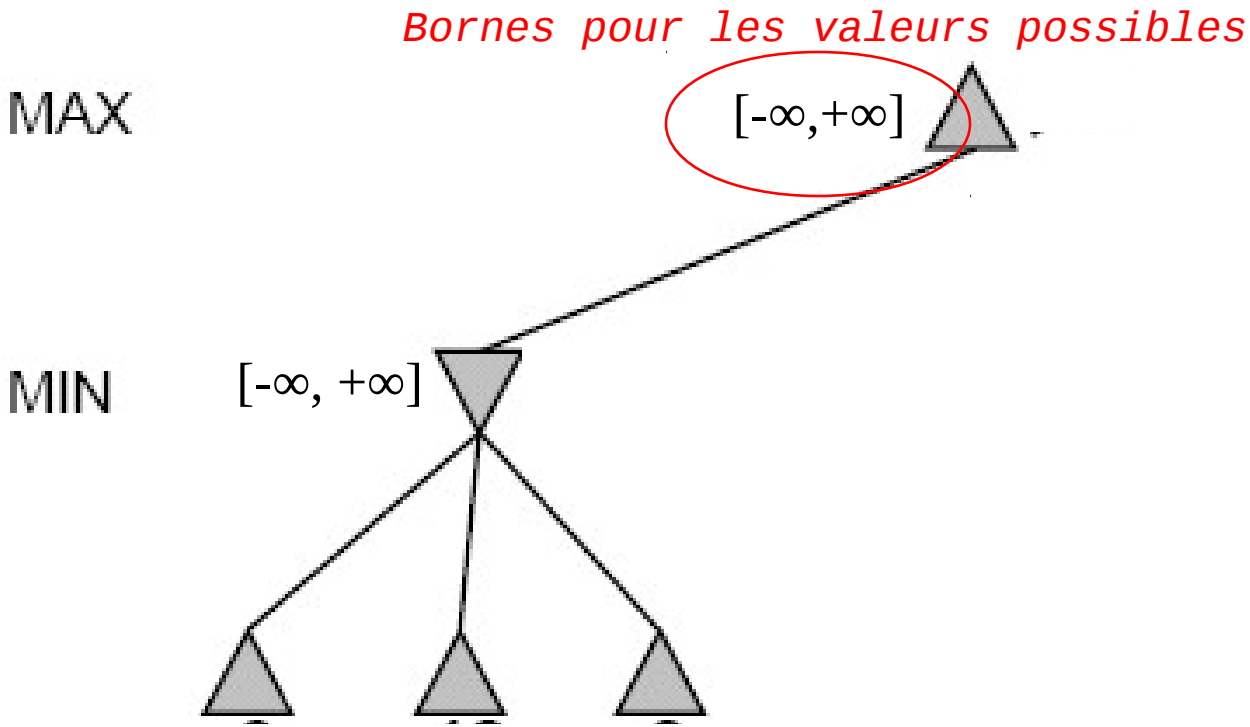
- Complet ?
  - Oui (si l'arbre est fini)
- Optimal ?
  - Oui (contre un adversaire qui joue de façon optimale)
- Complexité en temps ?
  - $O(b^m)$:
    - b: le nombre maximum coups (actions) légaux à chaque étape
    - m: nombre maximum de coups dans un jeu (profondeur maximale de l'arbre).
- Complexité en espace mémoire ?
  - $O(bm)$, vue que 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 un temps raisonnable.

(*alpha-beta pruning*)

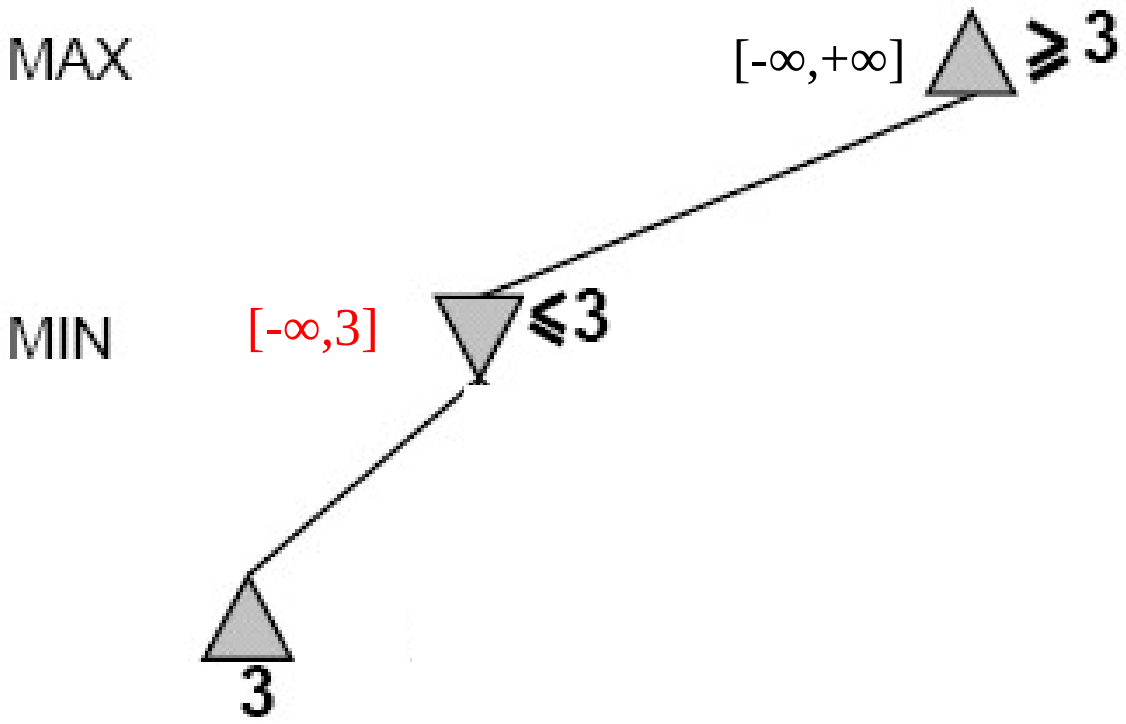
MINIMAX AVEC ÉLAGAGE ALPHA-BETA

Élagage alpha-beta

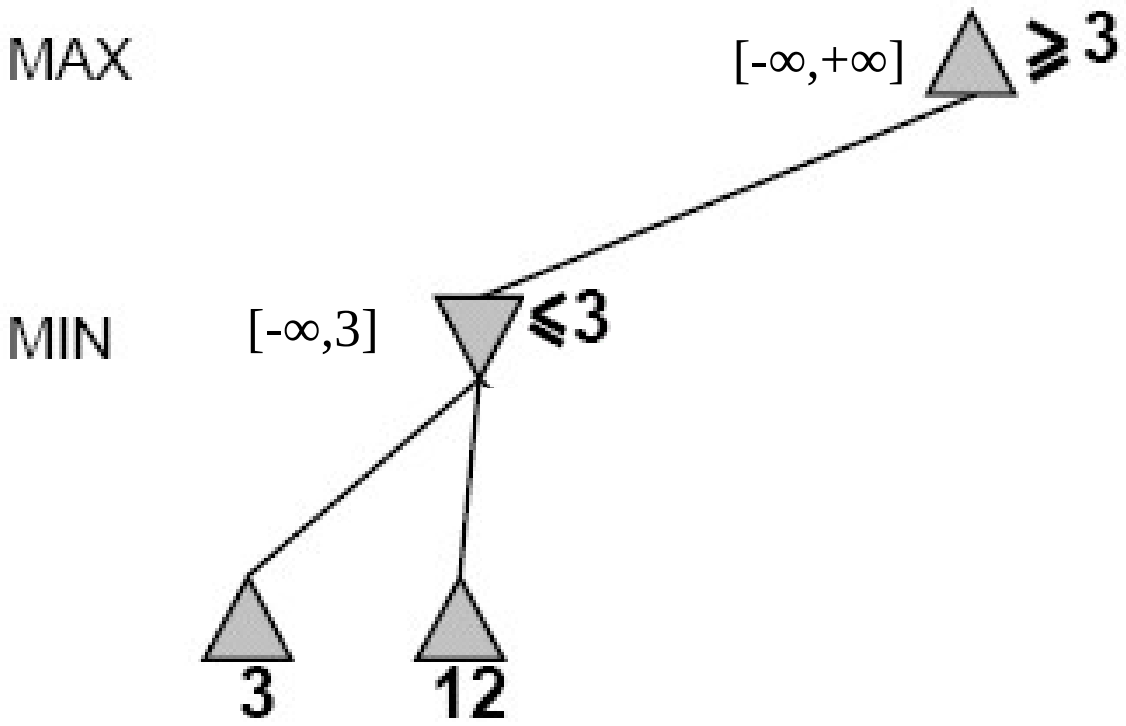
α : meilleure (plus grande) valeur de MAX jusqu'ici; elle ne décroît jamais;
MAX ne considère jamais les nœuds MIN successeurs (en bas de lui) ayant des valeurs plus petites que α.
β : meilleure (plus petite) valeur de MIN jusqu'ici; elle ne croît jamais;
MIN ne considère jamais les nœuds MAX successeurs (en bas de lui) ayant des valeurs plus grandes que β.



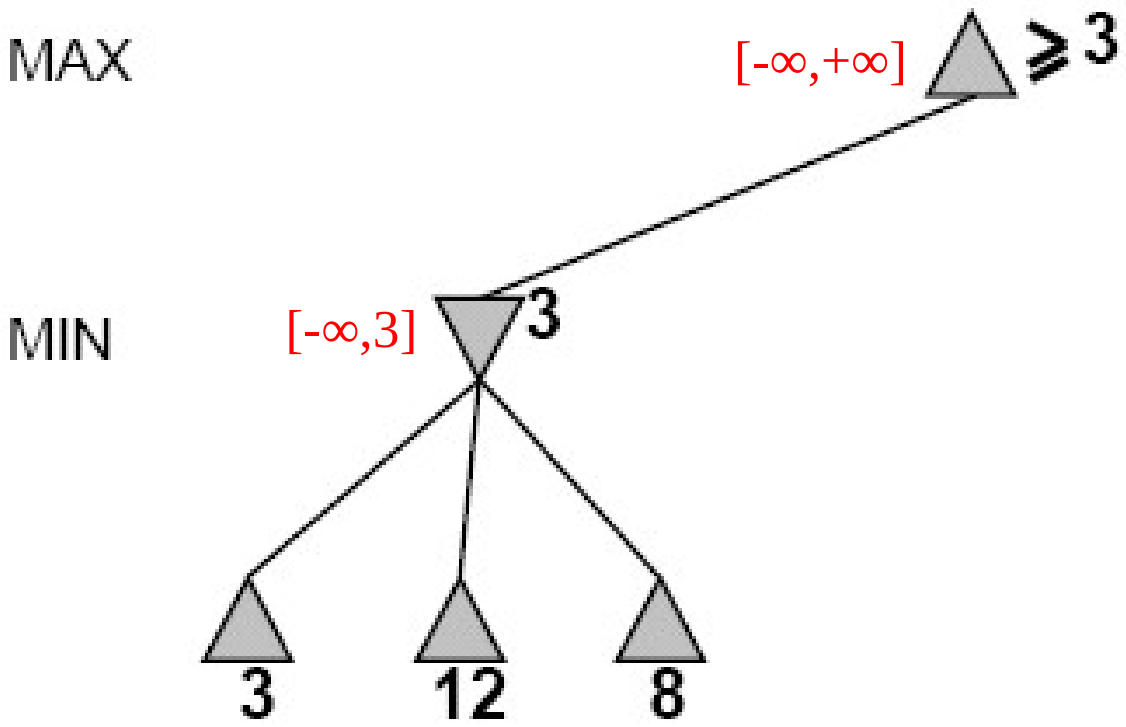
Élagage alpha-beta



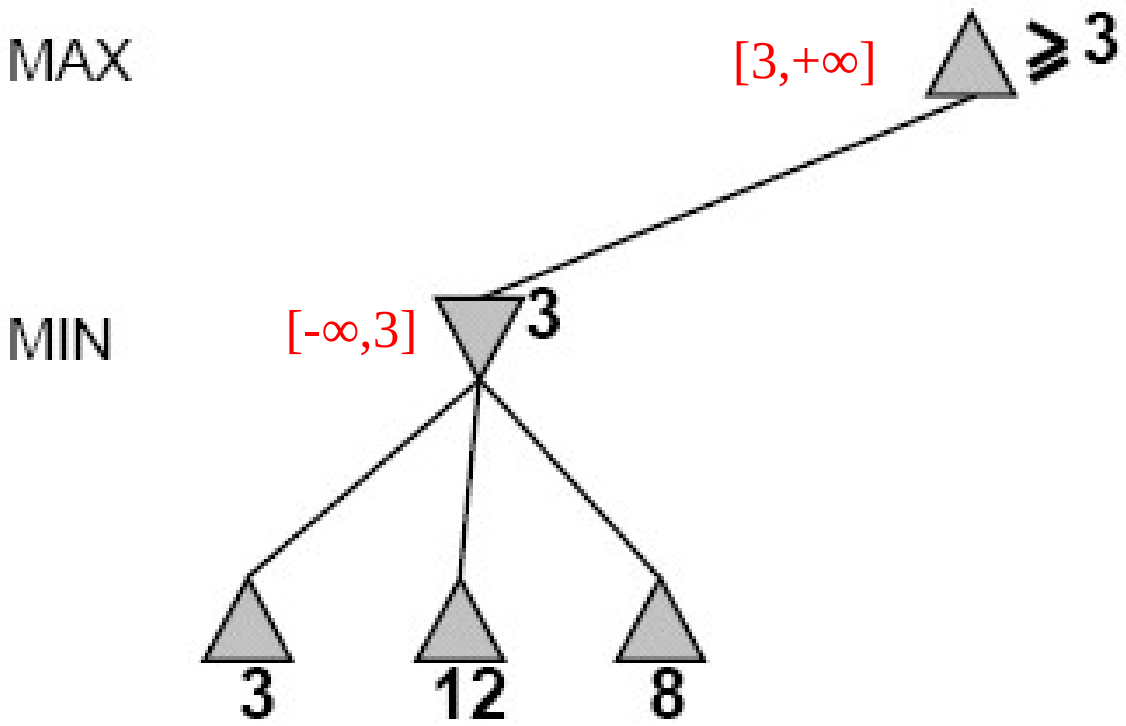
Élagage alpha-beta



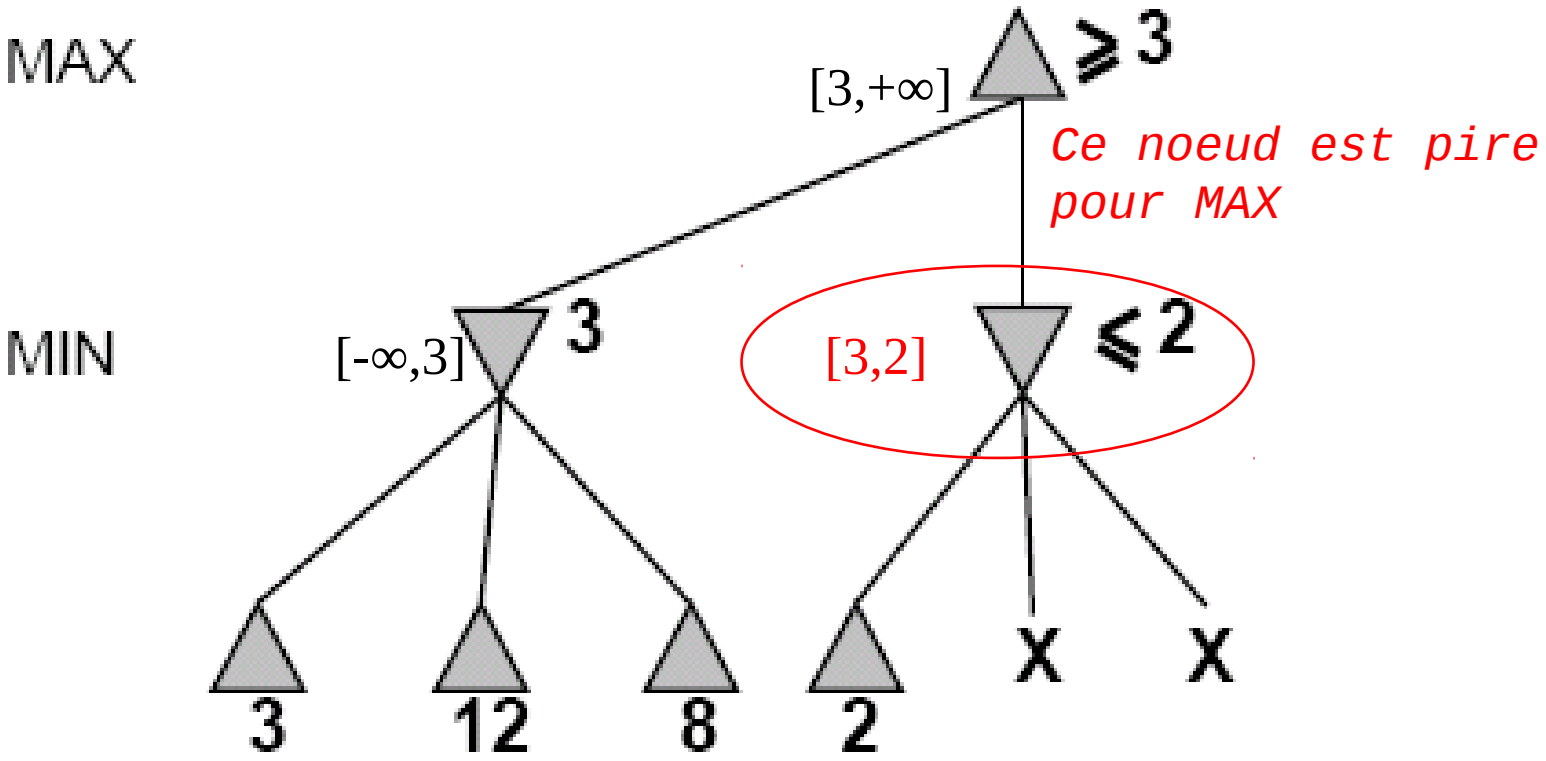
Élagage alpha-beta



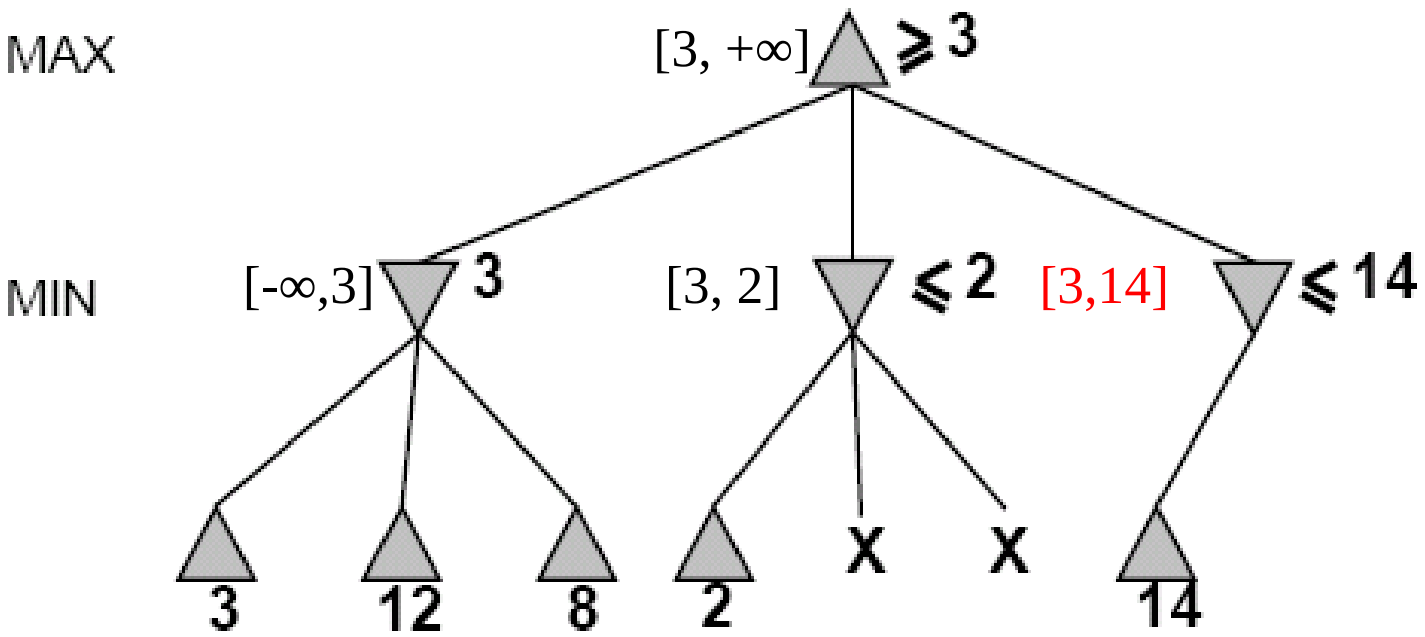
Élagage alpha-beta



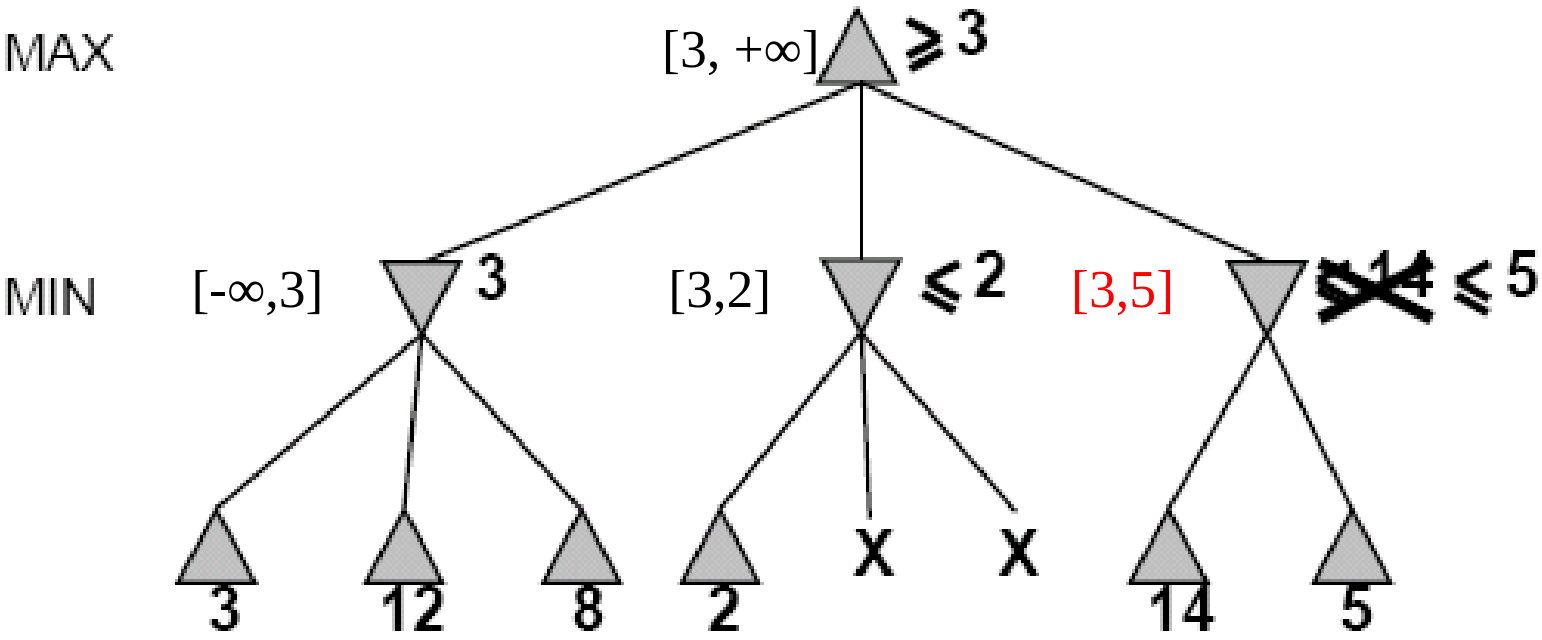
Élagage alpha-beta



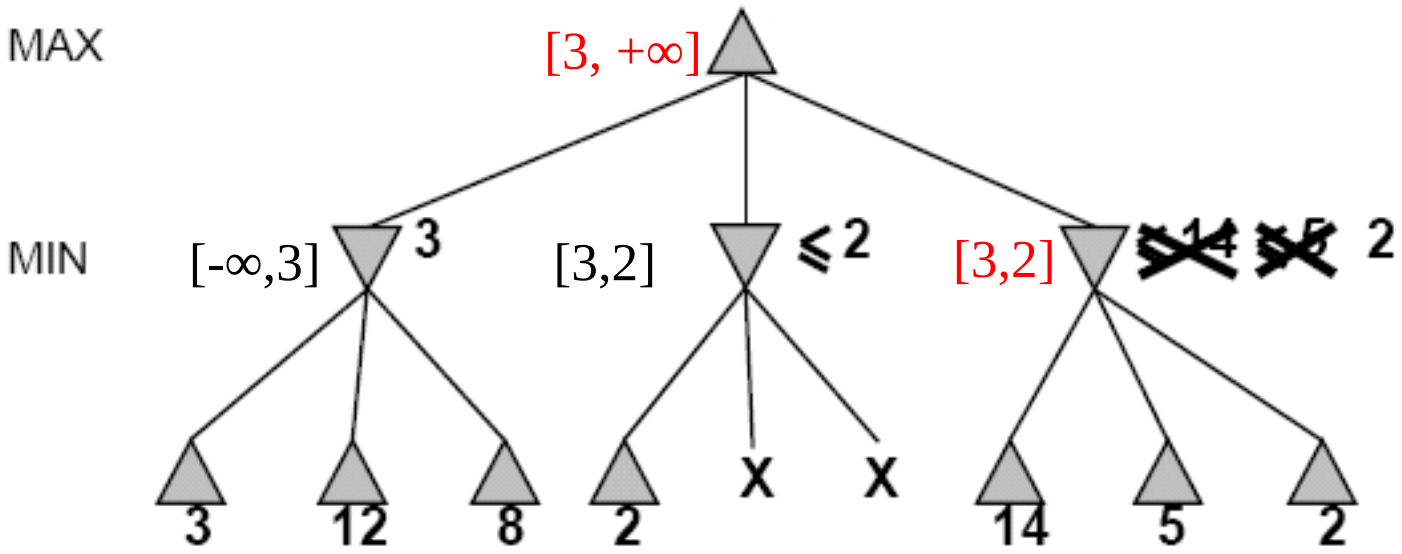
Élagage alpha-beta



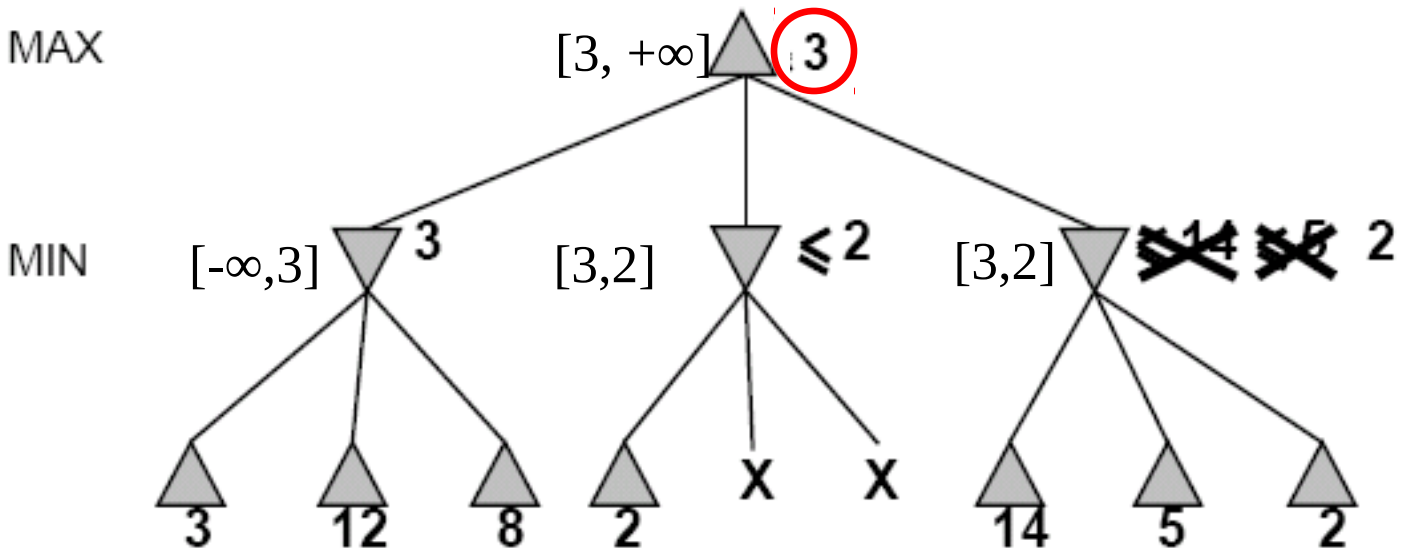
Élagage alpha-beta



Élagage alpha-beta

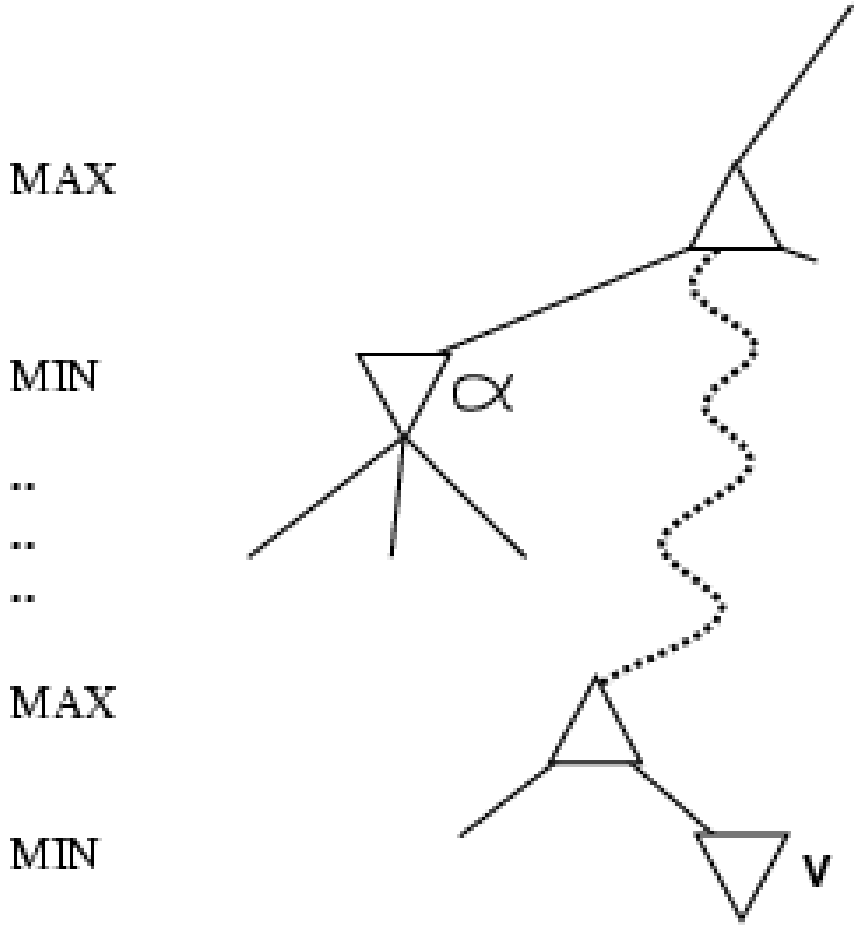


Élagage alpha-beta



D’ou vient le nom alpha-beta ?

- L’algorithme alpha-beta tire son nom des paramètre suivant décrivant les bornes des valeurs d’utilité enregistrée durant le parcourt.
  - α est la valeur du meilleur choix pour Max (c.-à-d., plus grande valeur) trouvé jusqu’ici:
    - Si le nœud *v* a une valeur pire que α, *Max* n’ira jamais à *v: c*ouper la branche
  - β est défini de manière analogue pour *Min*.



Algorithme avec élagage *alpha-beta*

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game

  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  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
    α, the value of the best alternative for MAX along the path to *state*
    β, 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**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

Algorithme avec élagage *alpha-beta (2)*

```
function MIN-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Propriétés de l'algorihme *alpha-beta*

- L'élagage n'affecte pas le résultat final de *minimax*.
- Dans le pire des cas, l'élagage alpha-beta (*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* !

Version unifiée de l'algorithme minimax avec élagage alpha-beta

```
fonction SearchAlphaBeta(state, α, β)  /* α < β */
   if TerminalTest(state) then
      return Utility(state)
   else
       best ←  -∞
       for (a,s) in Sucessors(state)
           v ← -SearchAlphaBeta(s,-β,-α)
           if v > best then
               best ← v
               if best > α then
                   α ← best
                   if α >= β then
                       return best
       return best
```

Avec l’algorithme alpha-beta

DÉCISIONS EN TEMPS RÉEL

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.
  - Fonction d'évaluation
    - Estimer des configurations par rapport à leur "chance" d'être gagnantes.

Exemple de fonction d'évaluation

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

$$\mathit{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Par exemple:
  - $w_1 = 9$,

    $f_1(s)$ = (number of white queens) – (number of black queens),
  - etc

Exemple de fonction d'évaluation

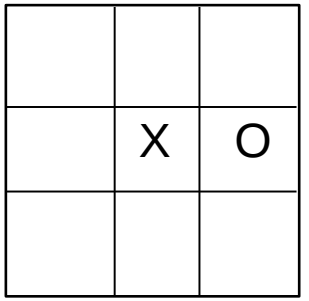
- Pour le *tic-tac-toe*, supposons que Max joue avec les X.

Eval(*s*) =

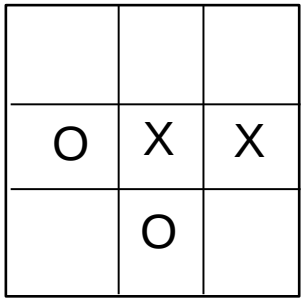
if s is win for Max, + ∞
if s is win for Min, - ∞

else

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



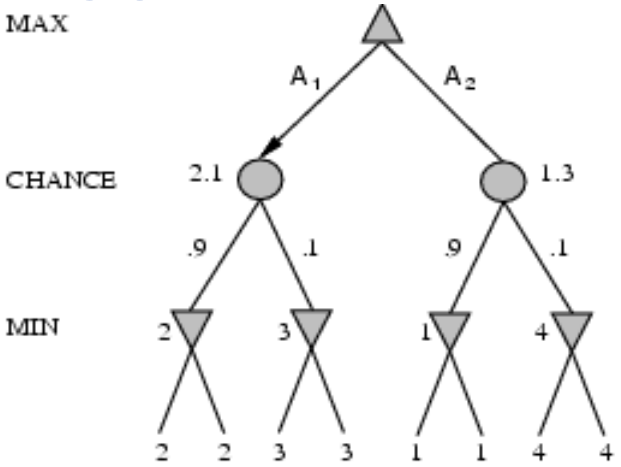
Eval(*s*) = 6 - 4 = 2



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

Généralisation aux actions aléatoires

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



EXPECTED-MINIMAX-VALUE(n) =

UTILITY(n) Si n est un nœud terminal

$\max_{s \in successors(n)}$ MINIMAX-VALUE(s) Si n est un nœud Max

$\min_{s \in successors(n)}$ MINIMAX-VALUE(s) Si n est un nœud Min

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

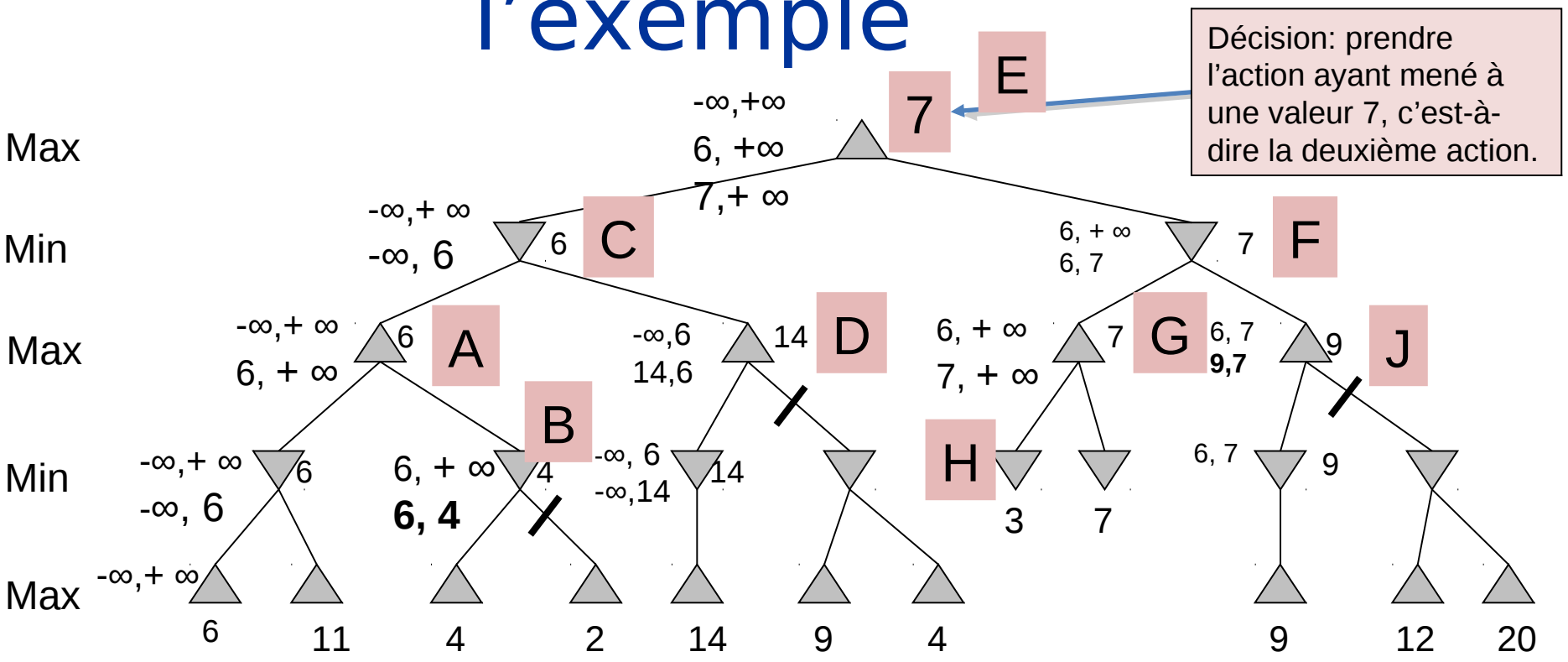
- Jeu de dames: En 1994, Chinook a mis fin aux 40 ans de règne du champion du monde Marion Tinsley. Chinook utilisait une base de données de coups parfaits pré-calculés pour toutes les configurations impliquant 8 pions ou moins: 444 milliards de configurations!

- 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 million 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 programs utilisent des bases de règles empiriques pour calculer le prochain coup.

Application Alpha-Beta sur l'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)
- **α, β** Valeur retournée
- Valeur si feuille

Exemple d'application

La valeur (v) du nœud, ainsi que la paire [α, β] à chaque nœud, à la terminaison de l'algorithme.

***Rappel** :*

α : meilleure (plus grande) valeur de MAX jusqu'ici; elle ne décroît jamais; MAX ne considère jamais les nœuds MIN successeurs (en bas de lui) ayant des valeurs plus petites que α.

β : meilleure (plus petite) valeur de MIN jusqu'ici; elle ne croît jamais; MIN ne considère jamais les nœuds MAX successeurs (en bas de lui) ayant des valeurs plus grandes que β.

A a α = 6 (A ne sera jamais plus petit que 6)

Ainsi, B est α -coupé puisque 4 < α = 6

C a β = 6 (C ne sera jamais plus grand que 6)

Ainsi, D est β-coupé puisque 14 > β = 6

(La première fois qu'on atteint G, en descendant, on a [α, β] = [-∞,+∞]. En remontant de H, [α, β] = [3,+∞]. En remontant de I, on a [α, β] = [7,+∞]).

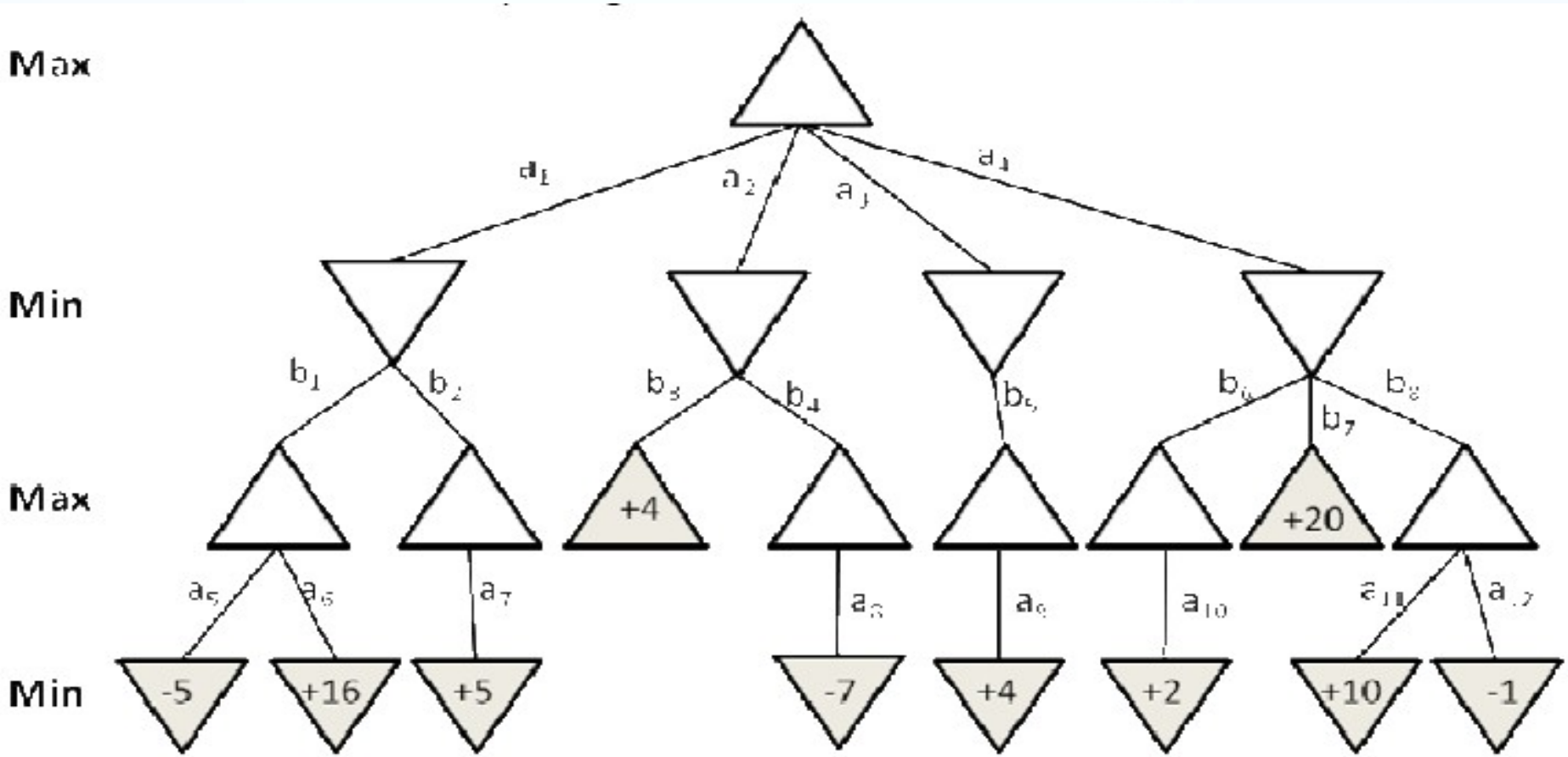
F a β = 7 (F ne sera jamais plus grand que 7)

Ainsi, J est β-coupé puisque 9 > β = 7

A la fin, E = 7.

37

Autre-Exemple : Appliquer Alpha-Beta sur cet exemple



38

Résumé

- Les recherches sur les jeux révèlent des aspects fondamentaux intéressants applicables à d'autres domaines.
- La perfection est inatteignable dans les jeux: il faut approximer.
- *Alpha-beta* 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).
- En général il peut être deux fois plus rapide que *minimax*.