Algorithmes de recherche avec adversaire

2I013 : Jeu à 2 joueurs

Cours 3

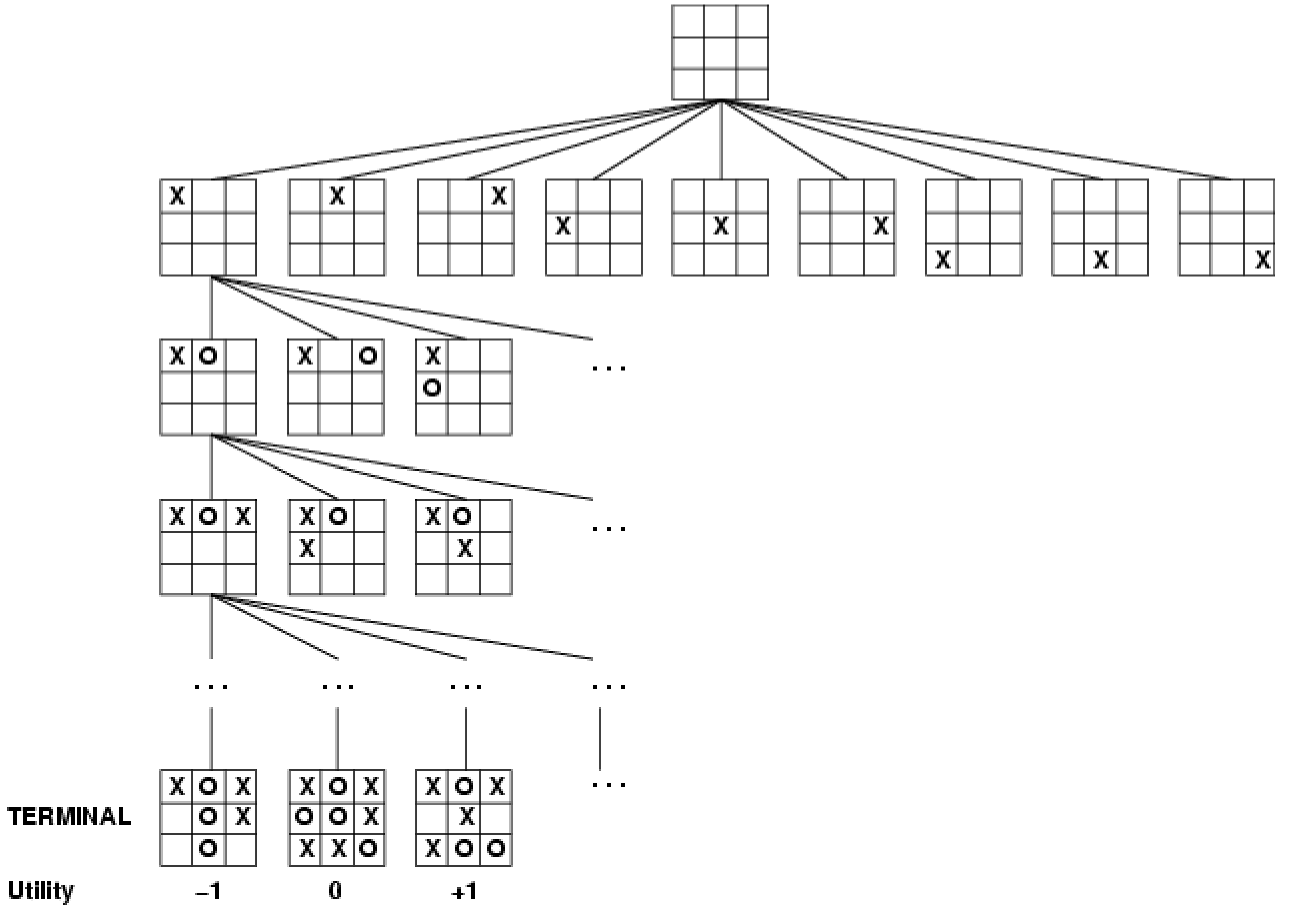
Jeux vs. problèmes de recherche

Deux problèmes majeurs :

- Ressources limitées

- Les “meilleurs” coups dépendent de la manière de jouer de l’adversaire

Arbre de jeu (2 joueurs, déterministe, tours)



Ressources limitées

Nombre de noeuds terminaux ~ b^m

- Vitesse d'exploration = 10^4 noeuds/sec
- On ne dispose que de 100 secs pour chaque coup
- Dans quelles conditions peut-on envisager une exploration exhaustive de l'abre de décision ?

Ressources limitées

Nombre de noeuds terminaux ~ b^m

- Vitesse d'exploration = 10^4 noeuds/sec
- On ne dispose que de 100 secs pour chaque coup
- Dans quelles conditions peut-on envisager une exploration exhaustive de l'abre de décision ?

➔ 100 x 10^4 = 10^6 noeuds considérés au maximum

➔ b^m doit être inférieur à 10^6

➔ Avec b=10, on ne peut considérer que des jeux où le nombre maximal de coups est de 6

Ressources limitéess

- Peu de chances d’être à même d’atteindre une situation cible dans des conditions réelles de jeu

- Approche standard:
  - Test d’arrêt :

    e.g., profondeur limite
  - Fonction d’évaluation

    = estimation de l’utilité d’une position

Fonctions d'évaluation

- Typiquement une combinaison linéaire de caractéristiques
  - *Eval(s)* = $w_1\ f_1(s) + w_2\ f_2(s) + \ldots + w_n\ f_n(s)$

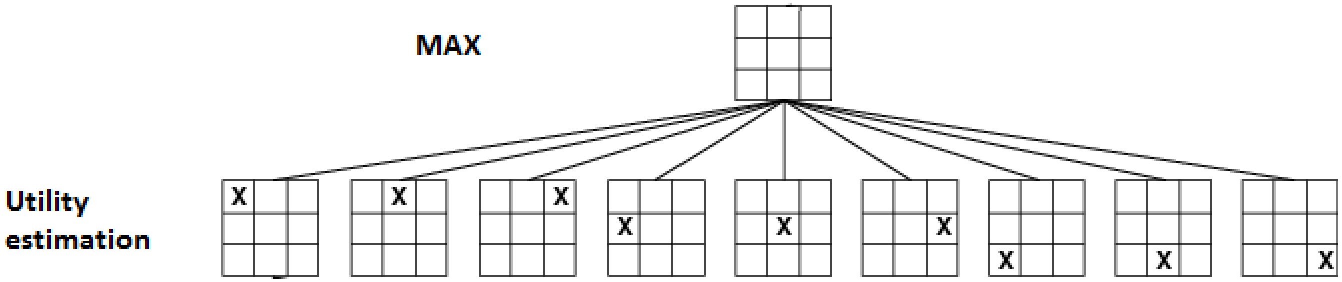
  e.g., $w_1 = 9$ avec $f_1(s)$ = (Nombre de pions noirs) – (nombre de pions blancs), etc.

- Quelles fonctions définir pour une bonne estimation de l'utilité dans le cadre des jeux Awélé et Othello ?
- Quels poids attribuer aux différentes composantes de l'évaluation ?

Horizon 1

- Profondeur limité à 1

  → Estimations effectuées sur les conséquences immédiates des actions



Horizon 1

- Ecrire les fonctions :
  - Evaluation : etatdejeu --> réel, qui retourne un score d’évaluation d’un état de jeu
  - Estimation : etatdejeu * Coup * profondeur --> réel, fonction récursive qui retourne un score d’utilité estimée pour un etat de jeu à partir d’appels de la fonction d’evaluation sur les feuilles de l’arbre (lorsque profondeur=pmax)
  - Decision : etatdejeu * List[Coup] --> Coup, qui retourne le coup dont le score correspond au score d’évaluation maximal de la liste de coups passée en paramètre

Jeu à 2 joueurs → Adversaire

- Au-delà d'une recherche à profondeur 1, composante inconnue : l'adversaire

  → La recherche doit prendre en compte un modèle de l'adversaire pour estimer l'utilité des nœuds en profondeur paire

- Cas Optimiste :
  - on considère que l'adversaire joue toujours le coup qui nous arrange le plus

Jeu à 2 joueurs → Adversaire

- Cas Optimiste :
  - on considère que l'adversaire joue toujours le coup qui nous arrange le plus

- Modifier la fonction d'estimation pour définir un joueur optimiste à profondeur m

Jeu à 2 joueurs → Adversaire

- Cas Optimiste peu réaliste :
  - Il est très peu probable que l'adversaire joue le coup qui nous arrange le plus

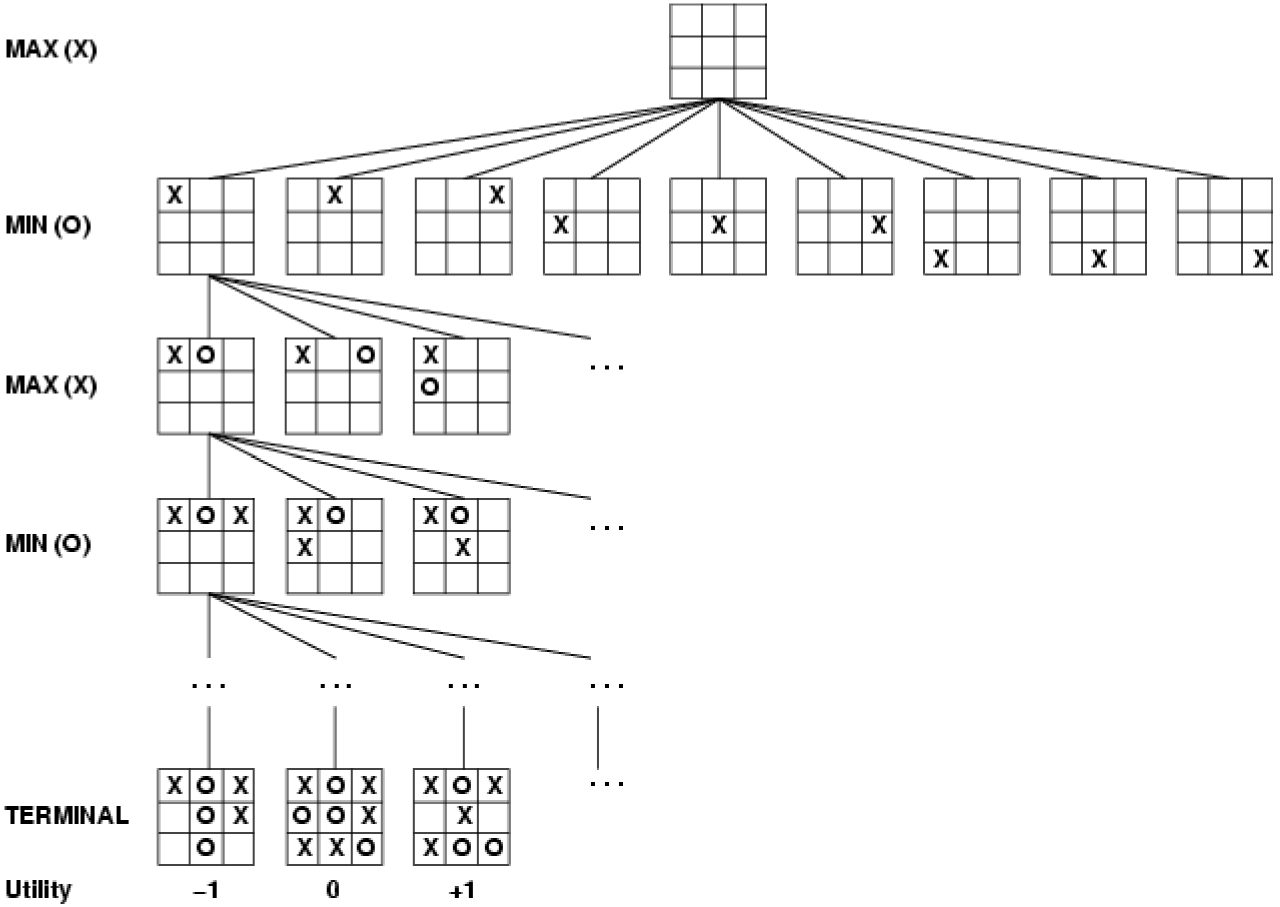
- Adversaire aléatoire :
  - Comment doit-on modifier la fonction d'estimation pour maximiser ses chances de victoire contre un joueur aléatoire ?

Jeu à 2 joueurs → Adversaire

- En pratique, on considère que le joueur adversaire :
  - suit la même stratégie que nous (même façon d'évaluer les nœuds terminaux)
  - joue toujours le coup qui nous arrange le moins

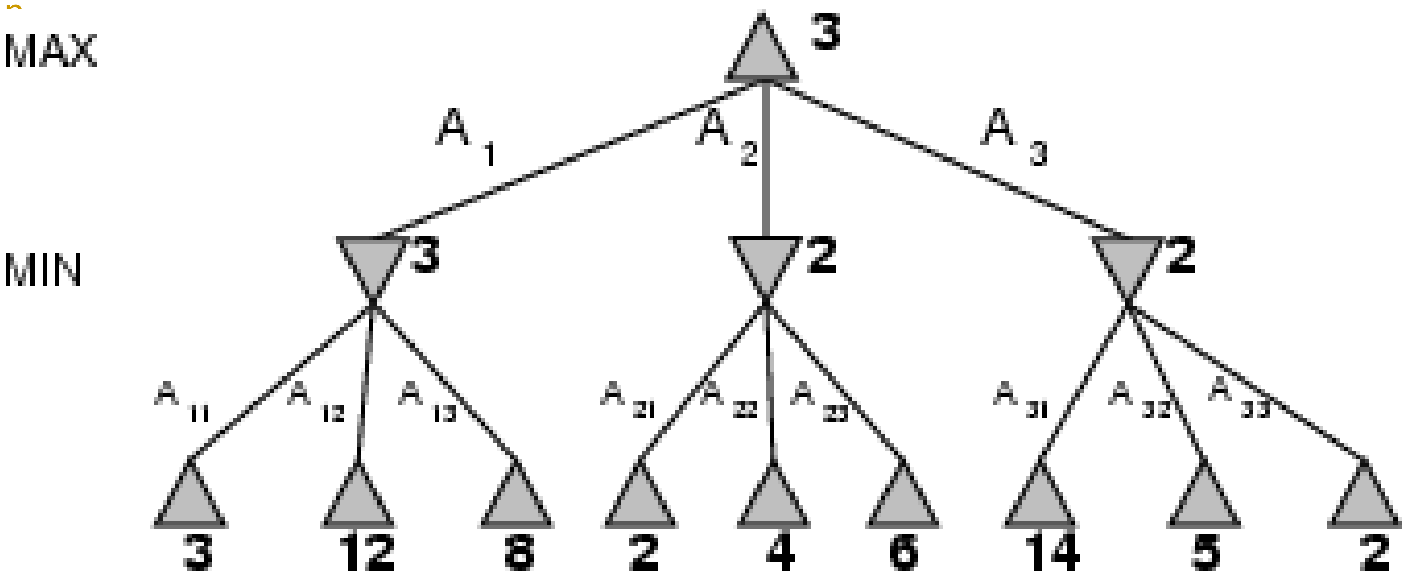
→ Algorithe MiniMax

Algorithme MiniMax



Minimax

- Jeu parfait pour des jeux déterministes
- Idée: Choisir le coup avec la meilleure valeur de minimax
= meilleure situation atteignable contre le meilleur coup
- E.g., jeu à 2 joueurs:



Algorithm 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
```

Propriétés de minimax

- Complet? Oui (si l'arbre est fini)
- Optimal? Oui (contre un adversaire optimal)
- Complexité en temps ? $O(b^m)$
- Complexité en mémoire ? $O(bm)$ (profondeur d'abord)

- Echecs : $b \approx 35$, $m \approx 100$ pour des parties "raisonnables"
  - ➔ solution exacte non calculable

MiniMax avec test d'arrêt

MinimaxCutoff est identique à *MinimaxValue* sauf que

1. *Final?* est remplacé par *ProfondeurMax?*
2. *Utilité* est remplacé par *Eval*

En pratique:

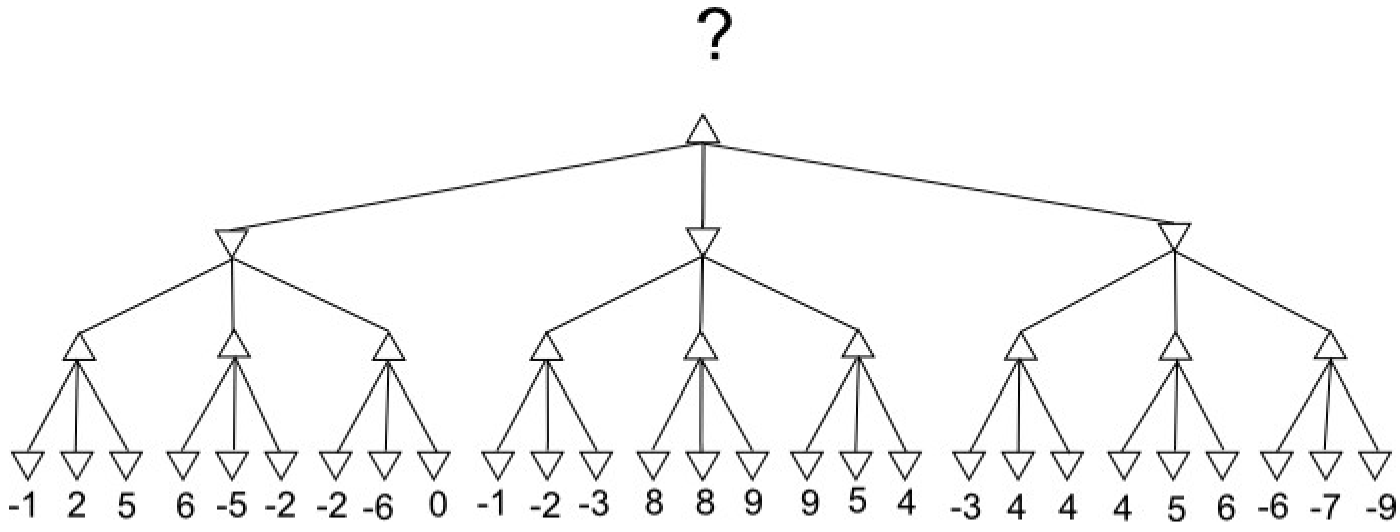
$b^m = 10^6$, $b=35 \rightarrow m=4$

Exploration à un horizon de 4 coups => très mauvais joueur d'échecs

- Horizon à 4 coups ≈ novice
- Horizon à 8 coups ≈ typique PC
- Horizon à 12 coups ≈ Deep Blue, Kasparov

MiniMax avec test d'arrêt

- Quel coup jouer ?



MiniMax avec test d'arrêt

- MiniMax
  - Modifier la fonction d'estimation pour définir un joueur MiniMax à profondeur m
    - Fonction maxValue : etatdejeu → réel
    - Fonction minValue : etatdejeu→ réel

MiniMax à profondeur limitée

- MiniMax à profondeur limitée
  - Complet? Non
  - Optimal? Non
  - Complexité en temps ? $O(b^l)$
  - Complexité en mémoire ? O(bl) (profondeur d'abord)

- Complexité en temps relativement élevée… Mais on peut éviter d'explorer certaines branches !!

Elagage α-β

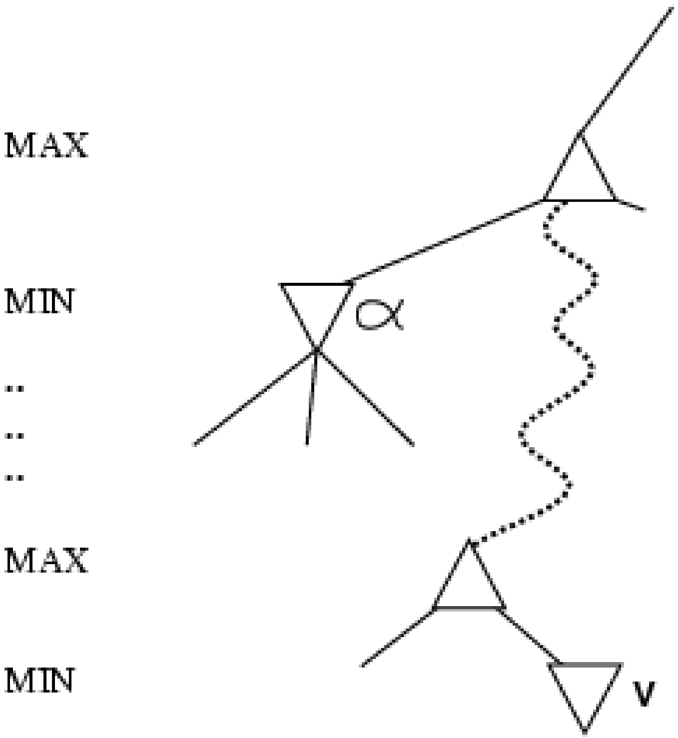
- Idée : certaines branches de l’arbre de recherche sont inutiles
  - 2 cas :
    - Le score qu’elles retourneront sera forcément < à un max déjà trouvé
    - Le score qu’elles retourneront sera forcément > à un min déjà trouvé

→ Forme de raisonnement sur la pertinence de certains calculs (sorte de meta-raisonnement)

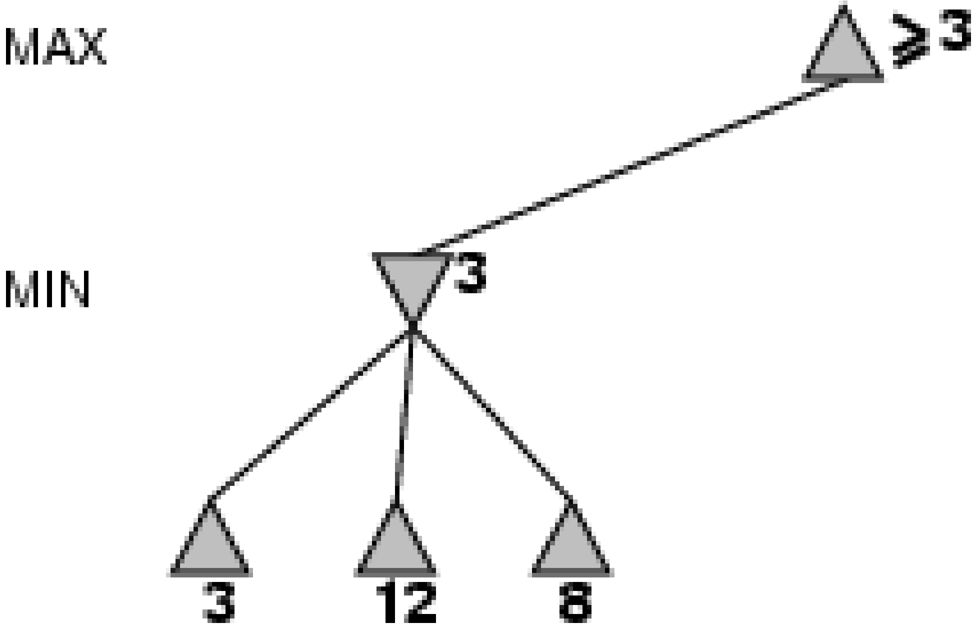
Pourquoi α-β?

- α est la valeur du meilleur (i.e., + grande valeur) coup trouvé jusqu'ici le long du chemin jusqu'à max
- Si v est moins bon que α, max l'évitera

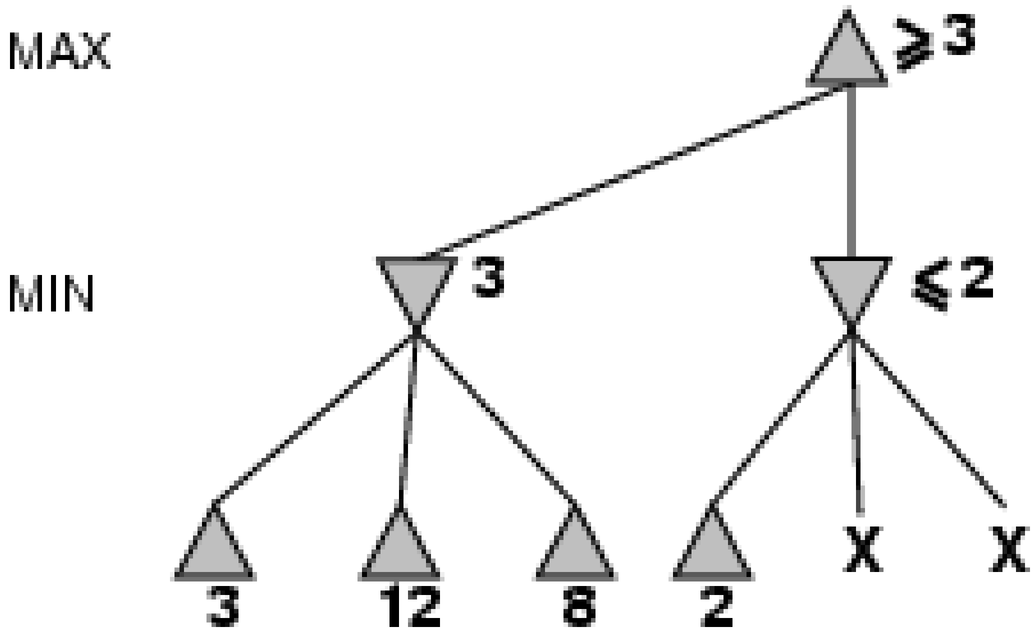
  → élimine la branche
- β est défini de façon similaire pour le min



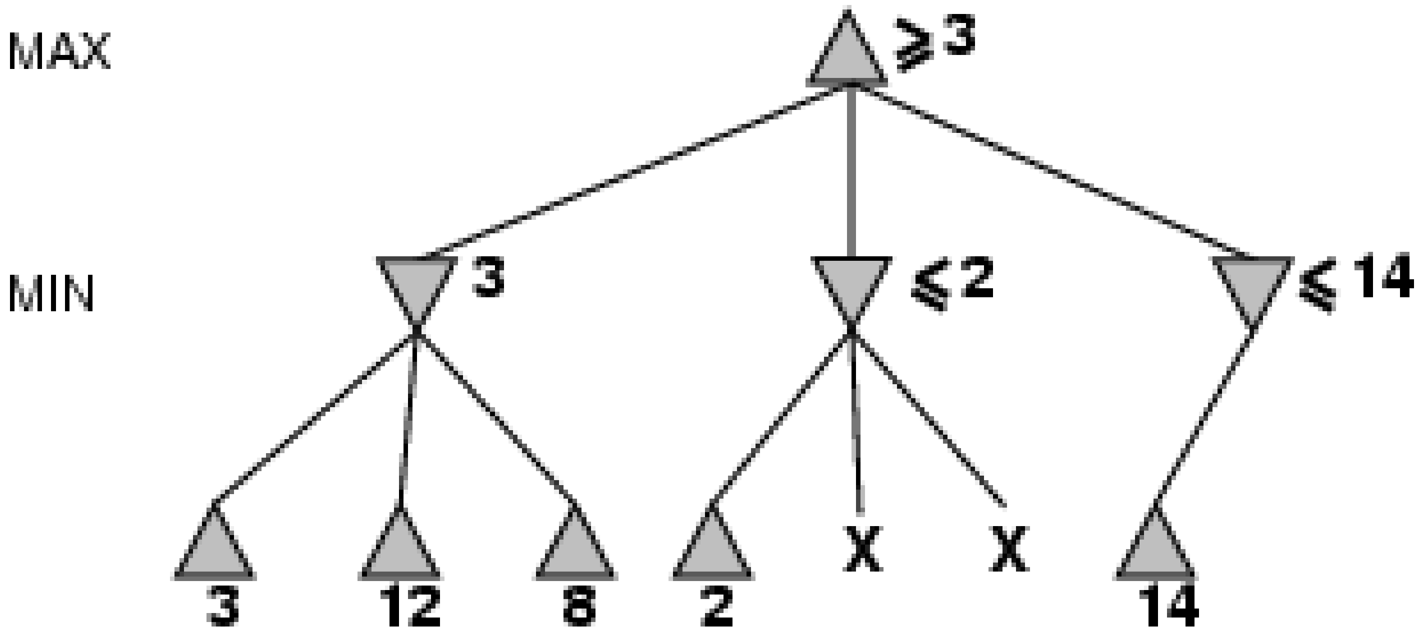
Elagage α-β : exemple



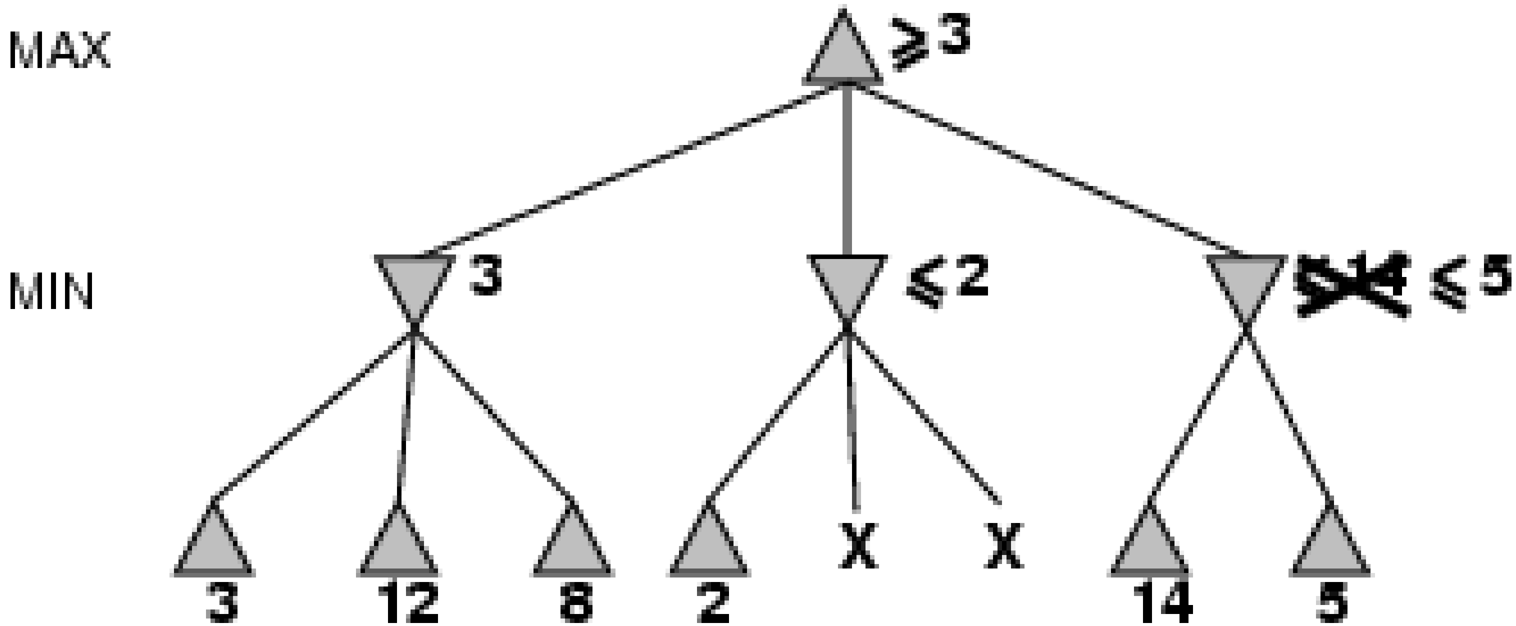
Elagage α-β : exemple



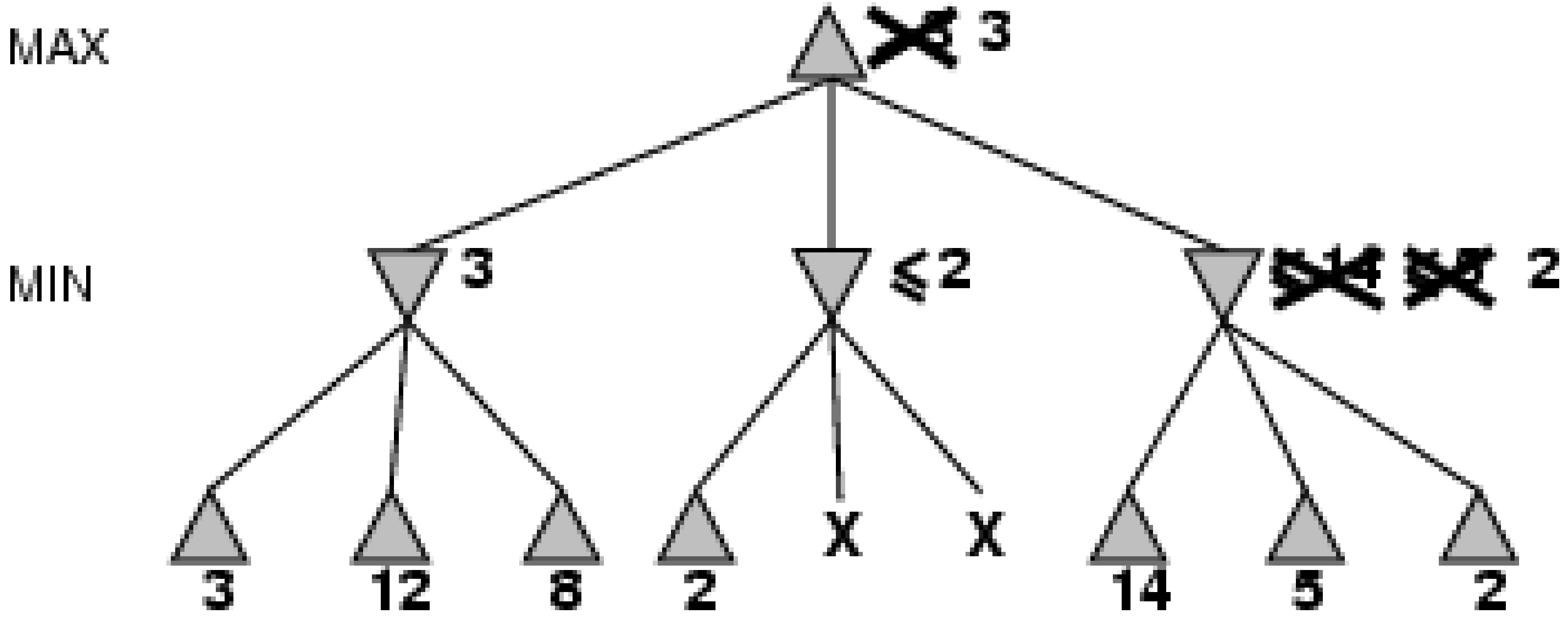
Elagage α-β : exemple



Elagage α-β : exemple



Elagage α-β : exemple



Propriétés de α-β

- L’élagage n’affecte pas le résultat final

- Un bon ordonnancement des coups améliore l’efficacité de l’élagage

- Avec “un tri parfait," compléxité en temps = $O(b^{m/2})$

  → permet de doubler la profondeur de recherche

L'algorithme α-β

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

   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-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 ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

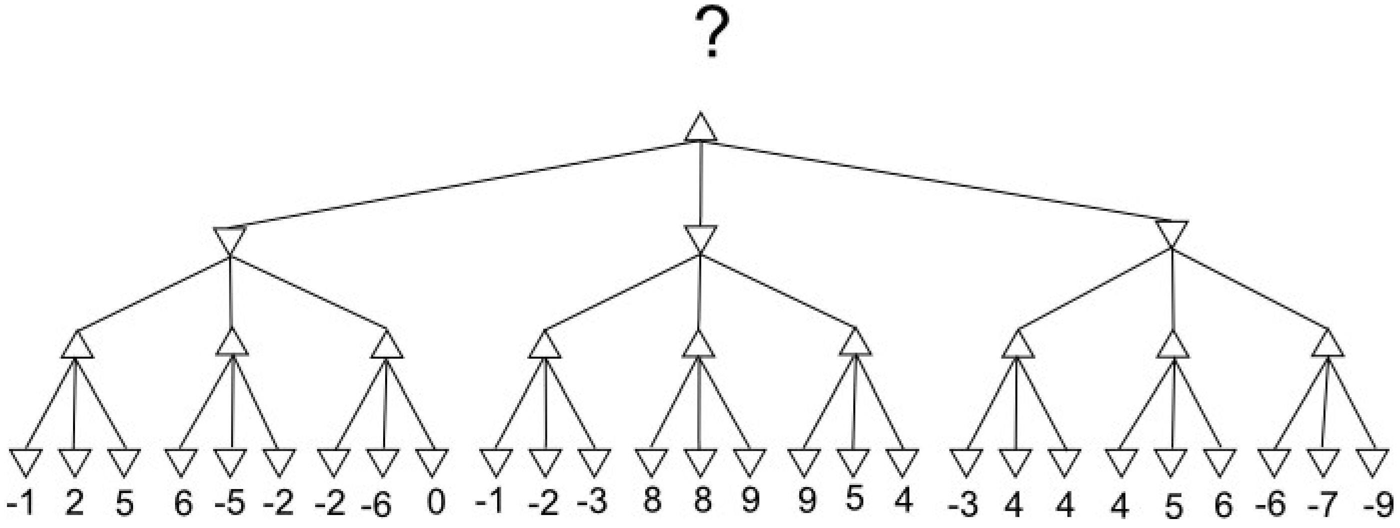
L' algorithme α-β

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*)
  $v \leftarrow +\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s, α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

L' algorithme α-β

- Combien de noeuds évalués?



L'algorithme negamax alpha-beta

Idée : min(a,b) = - max (-a,-b)

```
function negamax(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    childNodes := generateMoves(node)
    childNodes := orderMoves(childNodes)
    value := −∞
    foreach child in childNodes do
        value := max(value, −negamax(child, depth − 1, −β, −α, −color))
        α := max(α, value)
        if α ≥ β then
            break (* cut-off *)
    return value


)
```

L’algorithme negascout

Idée : hypothèse que première branche est la meilleure puis vérification avec fenêtre alpha-beta restreinte

```
function pvs(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    for each child of node do
        if child is first child then
            score := −pvs(child, depth − 1, −β, −α, −color)
        else
            score := −pvs(child, depth − 1, −α − 1, −α, −color) (* search with a null window *)
            if α < score < β then
                score := −pvs(child, depth − 1, −β, −score, −color) (* if it failed high,
                                                                 do a full re-search *)
        α := max(α, score)
        if α ≥ β then
            break (* beta cut-off *)
    return α
```

MTD-f

Idée : ajuster incrémentalement la fenêtre alpha-beta

```
function MTDF(root : node_type; f : integer; profondeur: integer) : integer;

      g := f;
      upperbound := +INFINITY;
      lowerbound := -INFINITY;
      repeat

            if g == lowerbound then beta := g + 1 else beta := g;
            g := AlphaBetaWithMemory(root, beta – 1, beta, profondeur);
            if g < beta then upperbound := g else lowerbound := g;

      until lowerbound >= upperbound;
      return g;
```

avec f une valeur arbitraire à l’initialisation

- on teste différentes fenêtres alpha beta de largeur 1
- si g > beta courant, alors la vraie valeur est hors de la fenêtre et g est lowerbound
- si g < beta courant, alors la vraie valeur est en dessous de g (g est upperbound)
- au début fenêtre centrée sur f : + f proche de la vraie valeur, + on va vite
- Utilisation d’un alpha beta avec memoire pour éviter de tout recalculer à chaque nouvelle fenêtre

Alpha-beta with memory

```
function AlphaBetaWithMemory(n : node_type; alpha , beta , d : integer) : integer;

      if retrieve(n) == OK then /* Transposition table lookup */
            if n.lowerbound >= beta then return n.lowerbound;
            if n.upperbound <= alpha then return n.upperbound;
            alpha := max(alpha, n.lowerbound);
            beta := min(beta, n.upperbound);

      if d == 0 then g := evaluate(n); /* leaf node */
      else if n == MAXNODE then
            g := -INFINITY; a := alpha; /* save original alpha value */
            c := firstchild(n);
            while  (g < beta) and (c != NOCHILD) do
                  g := max(g, AlphaBetaWithMemory(c, a, beta, d – 1));
                  a := max(a, g);
                  c := nextbrother(c);
      else /* n is a MINNODE */
            g := +INFINITY; b := beta; /* save original beta value */
            c := firstchild(n);
            while (g > alpha) and (c != NOCHILD) do
                  g := min(g, AlphaBetaWithMemory(c, alpha, b, d – 1));
                  b := min(b, g);
                  c := nextbrother(c);

      /* Traditional transposition table storing of bounds */
      /* Fail low result implies an upper bound */
      if g <= alpha then n.upperbound := g; store n.upperbound;
      /* Found an accurate minimax value – will not occur if called with zero window */
      if g > alpha and g < beta then n.lowerbound := g; n.upperbound := g; store n.lowerbound,
      n.upperbound;
      /* Fail high result implies a lower bound */
      if g >= beta then n.lowerbound := g; store n.lowerbound;
      return g;
```

Jeux déterministes en pratique

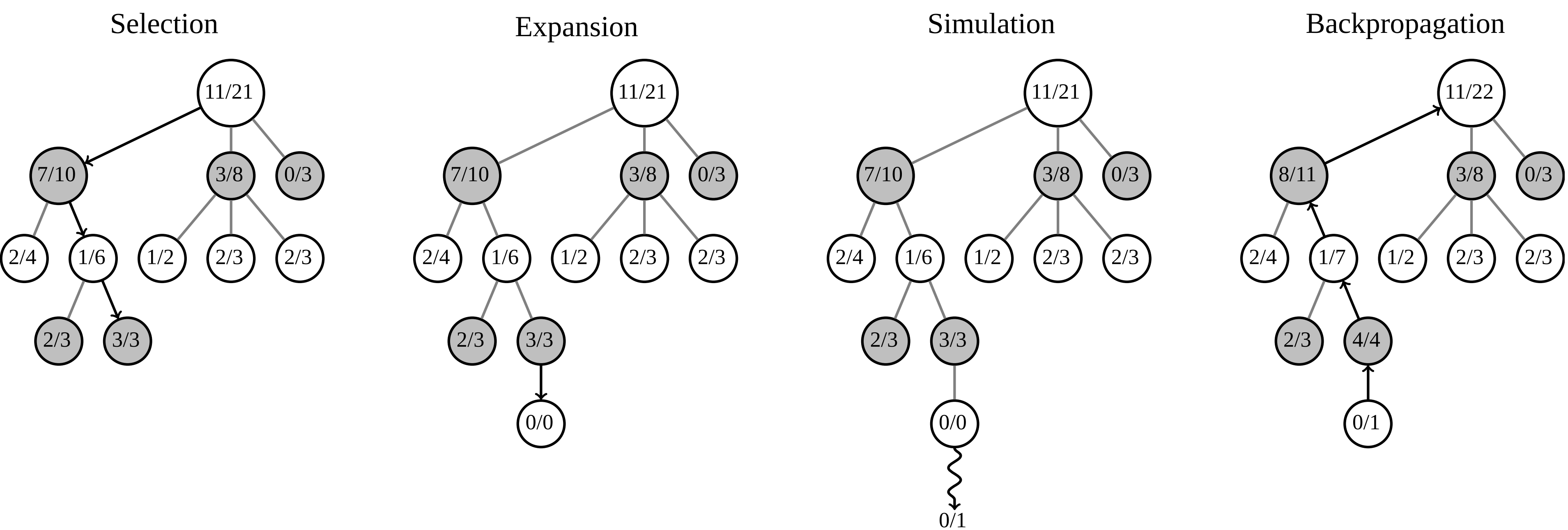
- Dames: Chinook a gagné le championnat du monde en 1994. utilise une base des coups idéaux pour toutes les positions avec moins de 8 pièces, 444 milliards de positions.

- Echecs: Deep Blue a battu Garry Kasparov en 1997. Deep Blue évalue 200 million positions par seconde, et utilise des stratégies explorant jusqu’à 40 coups.

- Othello: Les champions humains ne jouent pas contre les ordinateurs, trop bons.

- Go: Les champions humains ne jouent pas contre les ordinateurs sur ce genre d'algo, trop mauvais. Au go, b > 300.

MonteCarlo Tree Search



- Quand facteur de branchement trop élevé : estimation des situations par simulations (rollouts)

MonteCarlo Tree Search

- Sélection selon UCT (extension d'UCB aux arbres) :

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

Avec w_i le nombre de victoires à partir de la situation i, n_i le nombre de simulations à partir de i, N_i le nombre de simulations total à partir de la situation courante et c un paramètre de contrôle de l'exploration.

→ Compromis exploitation / exploration

- Possibilité d'apprendre
  - des stratégies de sélection (priors),
  - des stratégies d'expansion,
  - des stratégies d'évaluation (heavy playouts)

AlphaGo : Réseaux de neurones pour l'exploration