Cinquième partie V

Programmation des jeux de réflexion

Plan

En bref ...

L'algorithme MINIMAX

Limiter les ressources

L'élagage α-β

Jeux non-déterminsites

Programmation des jeux vs. résolution de problèmes

- L'adversaire est imprévisible $\Rightarrow$ la solution doit être contingente
- Un temps limite est imposé $\Rightarrow$ quand il n'est pas possible d'atteindre le but il faut être capable de l'approximer
- Pistes étudiées :
  1. algorithme pour joueur parfait (Von Neumann, 1944)
  2. horizon fini, évaluation approaximative (Zuse, 1945 ; Shannon 1950)
  3. élagage pour réduire le coût de recherche (McCarthy, 1956)

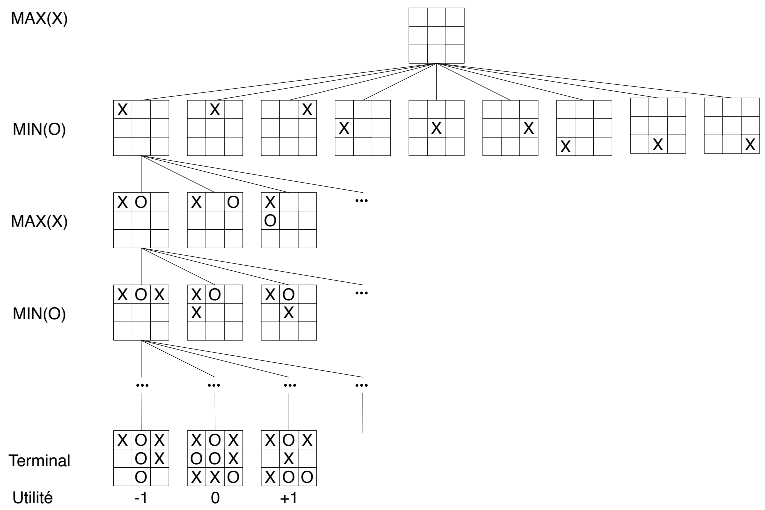
Programmation des jeux de réflexion

- Les différents types de jeux

imformation	déterministe	hasard
parfaite	échec, reversi, go, morpion	backgammon, monopoly
imparfaite		bridge, pocker, scrabble

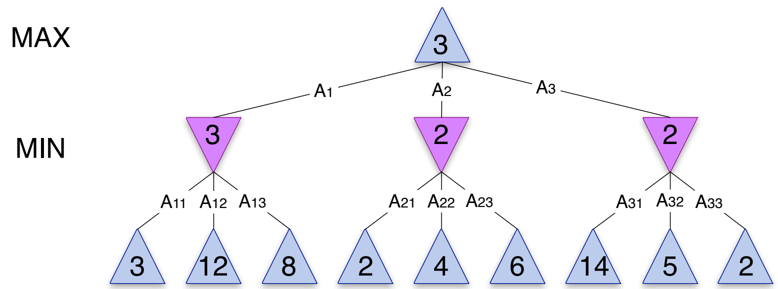
L'algorithme Minimax

L'arbre de recherche MiniMax



L'algorithme recherche MiniMax

- Donne le coup parfait pour un jeu déterministe à information parfaite
- Idée : choisir le meilleur coup vers la position avec la meilleure valeur **minimax** = meilleur valeur possible contre le meilleur jeu de l'adversaire
- Exemple d'un jeu à deux coups :



L'algorithme MiniMax

Algorithme

funtion MINIMAX-DECISION(*game*) **returns** *an operator*
  foreach *op in* OPERATORS[*game*] **do**
    VALUE[*op*] ← MINIMAX-VALUE(APPLY(*op*, *game*), *game*)
  return *the op with the highest* VALUE[*op*]

funtion MINIMAX-VALUE(*state*, *game*) **returns** *an utility value*
  if TERMINAL-TEST[*game*](*state*) **then**
    return UTILITY[*game*](*state*)
  else if MAX *is to move in state* **then**
    return *the highest* MINIMAX-VALUE *of* SUCCESSORS(*state*)
  else
    return *the lowest* MINIMAX-VALUE *of* SUCCESSORS(*state*)

L'algorithme MiniMax

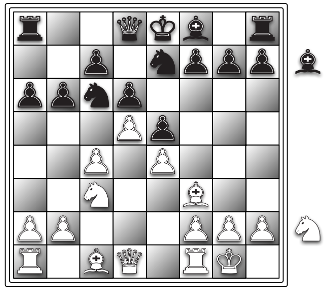
- Minimax s'arrête toujours si l'arbre est fini
- Optimal, si l'adversaire est optimal
- Si b est le nombre de coups possibles par situation et m la profondeur maximale de l'arbre, minimax a une complexité en
  - temps de $O(b^m)$
  - en espace $O(bm)$
- Pour les échecs par exemple : $b \approx 35$, $m \approx 100 \Rightarrow$ solution exacte impossible

Limiter les ressources

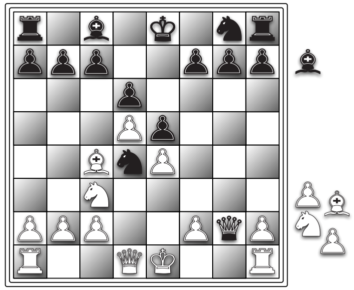
L'algorithme MiniMax

- Par exemple, on a 100 secondes et on peut explorer 10^4 nœuds par secondes. Donc, on peut regarder 10^6 nœuds par coup.
- Approches standard :
  - Test d'arrêt (*cutoff*) : par exemple limiter la profondeur
  - Fonction d'évaluation = valeur estimée d'une position

L'algorithme MiniMax



Noir joue
(blanc en meilleur position)



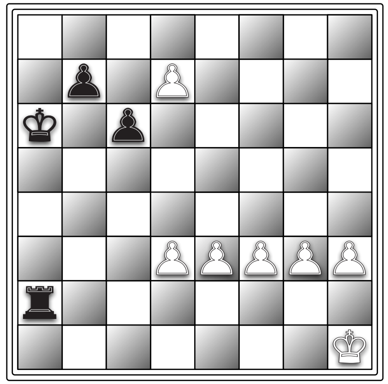
Blanc joue
(noir gagne)

Aux échecs on choisit par exemple une somme linéaire pondérée de caractéristiques.

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

e.g., $w_1 = 9$ et $f_1 =$ (le nombre de reines blanches) - (le nombre de reines noires), etc.

L'algorithme MiniMax



Dans cette position Noir peut mettre le roi blanc en échec un certain nombre de fois mais le pion blanc va se transformer inévitablement en reine. On en s'aperçoit très tard.

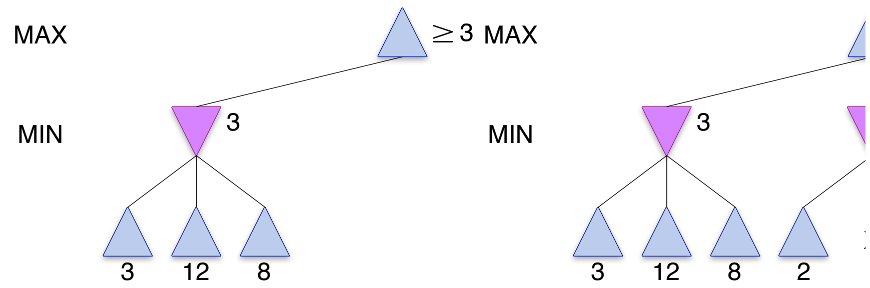
L'algorithme MiniMax

- On peut facilement modifier l'algorithme minimax en ajoutant un test d'arrêt (remplacer Terminal-Test par Cutoff-Test et en remplaçant Utility par Eval.
- En pratique : problèmes de performances, e.g., $b^m = 10^6$ et $b = 35$. Donc $m = 4$.
  - Aux échecs on ne pourrait que regarder 4 demi-coups en avance
    - Un humain normal $\approx$ 4 coups d'avance
    - Un ordinateur classique et un expert humain $\approx$ 8 coups d'avance
    - Deep Blue, Kasparov $\approx$ 12 coups d'avance
- Idée : Élaguer des branches inutiles

  $\Rightarrow$ algorithme α-β

L'élagage α-β

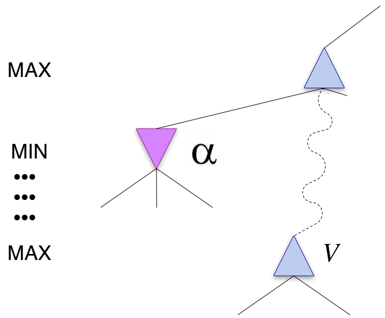
L'algorithme α-β



L'algorithme α-β

- L'élagage n'affecte pas le résultat final
- Un bon choix améliore l'efficacité de l'élagage
- Avec un « choix parfait » la complexité en temps est $O(b^{m/2})$
  - $\Rightarrow$ double la profondeur de recherche par rapport à Minimax
  - $\Rightarrow$ peut facilement atteindre des profondeurs de l'ordre de 8 coups d'avance aux échecs

L'algorithme α-β



- α est la meilleure valeur (la plus grande) pour MAX trouvée jusqu'à présent en dehors du chemin actuel
- Si V est pire que α MAX va l'éviter $\Rightarrow$ élaguer la branche
- Définir β d'une manière similaire pour MIN : β est la meilleur valeur (la plus petite) pour MIN jusqu'à présent

L'algorithme α-β

Algorithme

funtion $\textsc{Max-Value}(\textit{state}, \textit{game}, \alpha, \beta)$ **returns** *the minimax value of state*

input: *state, current state in game*
game, game description
α, the best score for $\textsc{Max}$ *along the path to state*
β, the best score for $\textsc{Min}$ *along the path to state*

if $\textsc{Cutoff-Test}(\textit{state})$ **then return** $\textsc{Eval}(\textit{state})$
foreach *s in* $\textsc{Successors}(\textit{state})$ **do**
$\alpha \leftarrow \textsc{Max}(\alpha, \textsc{Min-Value}(s, \textit{game}, \alpha, \beta))$
if $\alpha \geq \beta$ **then return** β
return α

L'algorithme α-β

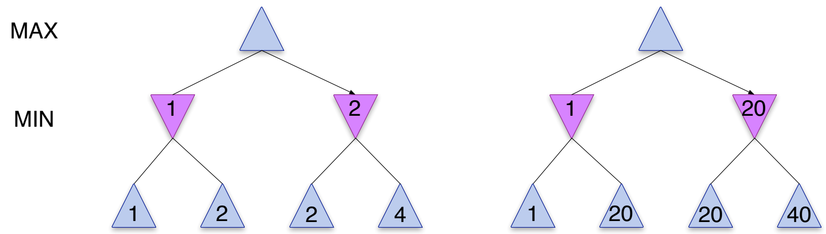
Algorithme

funtion MIN-VALUE(*state*, *game*, α, β) **returns** *the minimax value of state*

  input: *state, current state in game*
   game, game description
   α, the best score for MAX *along the path to state*
   β, the best score for MIN *along the path to state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  foreach *s in* SUCCESSORS(*state*) **do**
   $\beta \leftarrow$ MIN(β,MIN-VALUE(s, *game*, α, β))
   if $\beta \geq \alpha$ **then return** α
  return β

L'algorithme α-β



- Seulement l'ordre est important
- Le comportement est préservé pour chaque transformation monotone de la fonction EVAL

L'algorithme α-β

- L'ordre dans lequel on visite les fils est important
- Si on trouve rapidement une bonne valeur, on élague plus de nœuds
- On peut trier les fils par leur utilité
- Il y a d'autres améliorations
- Au mieux, on visite $\sqrt{n}$ nœuds au lieu de $n = b^m$ pour minimax
- Aux échecs on utilise au début du jeu des bases de données d'ouvertures, au milieu α-β et à la fin des algorithmes spéciaux

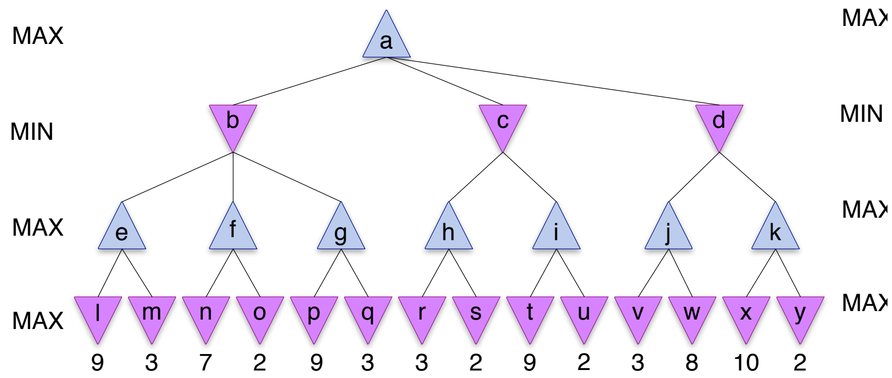
L'algorithme SSS*

- Définition : Une stratégie (complète) pour le joueur max, étant donné un arbre de jeux A, est un sous-arbre, qui
  - contient la racine de A
  - dont chaque nœud Max a exactement un fils
  - dont chaque nœud Min a tous ses fils
- Une stratégie partielle pour le joueur max, étant donné un arbre de jeux A, est un sous-arbre de A, qui
  - contient sa racine
  - dont chaque nœud Max a au plus un fils

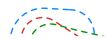
L'algorithme SSS*

- Une stratégie partielle S représente implicitement toutes les stratégies complètes C_S auxquelles on aboutie en développant S
- La valeur d'une stratégie est le minimum des valeurs des feuilles
- La valeur d'une stratégie partielle S donne une borne supérieur pour toutes ses stratégies complètes C_S
- Idée : On recherche a compléter les stratégies partielles suivant leur valeur jusqu'à présent
- Une fois quand a trouvé la stratégie optimale complète à partir d'un nœud x on ne considère plus les autres de x

L'algorithme SSS*



Trois stratégies partielles



L'algorithme SSS*

- $\textsc{Successors}(x)$ donne pour tout nœud x la liste des successeurs
- $\textsc{Eval}(x)$ donne l'évaluation d'un noeud x
- $\textsc{Max}(x)$ est vrai si et seulement si x est un nœud max
- On utilise une pile P pour mettre les nœuds encore à traités
- $\textsc{Ordonnate}(P, \langle x, f, h\rangle)$ met $\langle x, f, h\rangle$ dans la pile, ordonnée en ordre descendant par les valeurs h. Si dans P il y a déjà un élément avec la même valeur h on range $\langle x, f, h\rangle$ avant.
- Il y a deux types de nœuds :
  - f : fermé (la stratégie optimale pour lui est connu)
  - v : vivant

L'algorithme SSS*

Algorithme

funtion $\textsc{Valeur}(x_0$ *: node)* **returns** *un réel*

- $P \leftarrow (\langle x_0, v, \infty\rangle)$
- **repeat**
  - $\langle x, s, h\rangle \leftarrow \textsc{First}(P)$; $P \leftarrow \textsc{Rest}(P)$; $x_1, x_2, \ldots, x_m \leftarrow S(x)$
  - **if** $s = v$ **then**
    - **if** $m = 0$ **then** $\textsc{Ordonnate}(P, \langle x, f, \textsc{Min}(h, \textsc{Eval}(x))\rangle)$
    - **else**
      - **if** $\textsc{Max}(x)$ **then** $\textsc{Stack}(P, \langle x_1, v, h\rangle, \ldots, \langle x_m, v, h\rangle)$
      - **else** $\textsc{Stack}(P, \langle x_1, v, h\rangle)$
  - **else**
    - **if** $\textsc{Max}(x)$ **then**
      - **if** x *has a child* y **then** $\textsc{Stack}(P, \langle y, v, h\rangle)$
      - **else** $z \leftarrow \textsc{Parent}(x)$; $\textsc{Stack}(P, \langle z, f, h\rangle)$
    - **else** $z \leftarrow \textsc{Parent}(x)$; $\textsc{Stack}(P, \langle z, f, h\rangle)$; $\textsc{Remove-All-Child}(z)$
- **until** $P = \langle x, f, h\rangle$
- **return** h

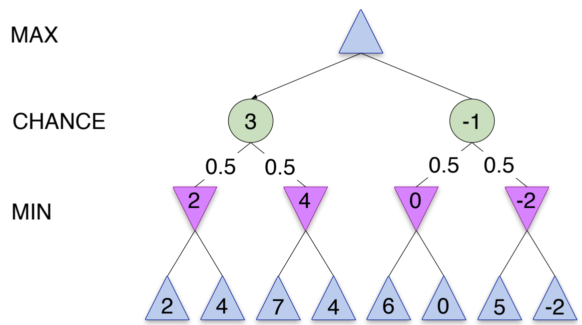
L'algorithme SSS*

$\langle a, v, \infty \rangle$
$\langle b, v, \infty \rangle \langle c, v, \infty \rangle \langle d, v, \infty \rangle$
$\langle e, v, \infty \rangle \langle c, v, \infty \rangle \langle d, v, \infty \rangle$
$\langle l, v, \infty \rangle \langle m, v, \infty \rangle \langle c, v, \infty \rangle \langle d, v, \infty \rangle$
$\langle m, v, \infty \rangle \langle c, v, \infty \rangle \langle c, v, \infty \rangle \langle l, f, 9 \rangle$
$\langle c, v, \infty \rangle \langle d, v, \infty \rangle \langle l, f, \infty \rangle \langle m, f, 3 \rangle$
$\langle h, v, \infty \rangle \langle d, v, \infty \rangle \langle l, f, \infty \rangle \langle m, f, 3 \rangle$
$\langle r, v, \infty \rangle \langle s, v, \infty \rangle \langle d, v, \infty \rangle \langle l, f, 9 \rangle \langle m, f, 3 \rangle$
$\langle s, v, \infty \rangle \langle d, v, \infty \rangle \langle l, f, 9 \rangle \langle r, f, 3 \rangle \langle m, f, 3 \rangle$
$\langle d, v, \infty \rangle \langle l, f, 9 \rangle \langle r, f, 3 \rangle \langle m, f, 3 \rangle \langle s, f, 2 \rangle$
$\langle j, v, \infty \rangle \langle l, f, 9 \rangle \langle r, f, 3 \rangle \langle m, f, 3 \rangle \langle s, f, 2 \rangle$
$\langle v, v, \infty \rangle \langle w, v, \infty \rangle \langle l, f, 9 \rangle \langle r, f, 3 \rangle \langle m, f, 3 \rangle \langle s, f, 2 \rangle$
$\langle w, v, \infty \rangle \langle l, f, 9 \rangle \langle v, f, 3 \rangle \langle r, f, 3 \rangle \langle m, f, 3 \rangle \langle s, f, 2 \rangle$
$\langle l, f, 9 \rangle \langle w, f, 8 \rangle \langle v, f, 3 \rangle \langle r, f, 3 \rangle \langle m, f, 3 \rangle \langle s, f, 2 \rangle$
etc.

Jeux non-déterminsites

Jeux non-déterministes

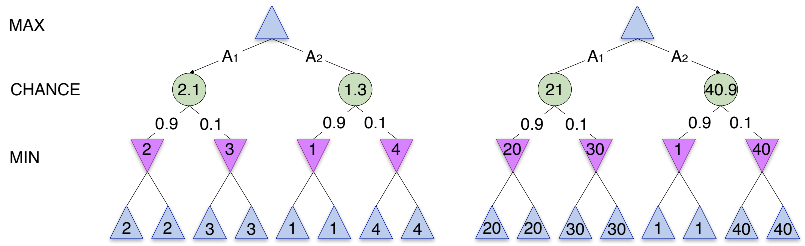
- Dans le Backgammon par exemple, le lancer des dés détermine les coups corrects. Exemple simplifié avec lancer d'une pièce de monnaie :



Algorithmes pour les jeux non-déterministes

- Expectimax donne coup parfait comme Minimax
- On doit considérer les nœuds CHANCE
- On doit ajouter dans l'algorithme :
  if *state* is a chance node **then return** average of Expectimax-Value of Successors(*state*)
- L'algorithme α-β peut être adapté, si Eval est bornée.

Valeur exacte des nœuds est importante



- Le comportement est préservé seulement pour chaque transformation positive et linéaire de EVAL
- EVAL doit être proportionnelle au gain attendu