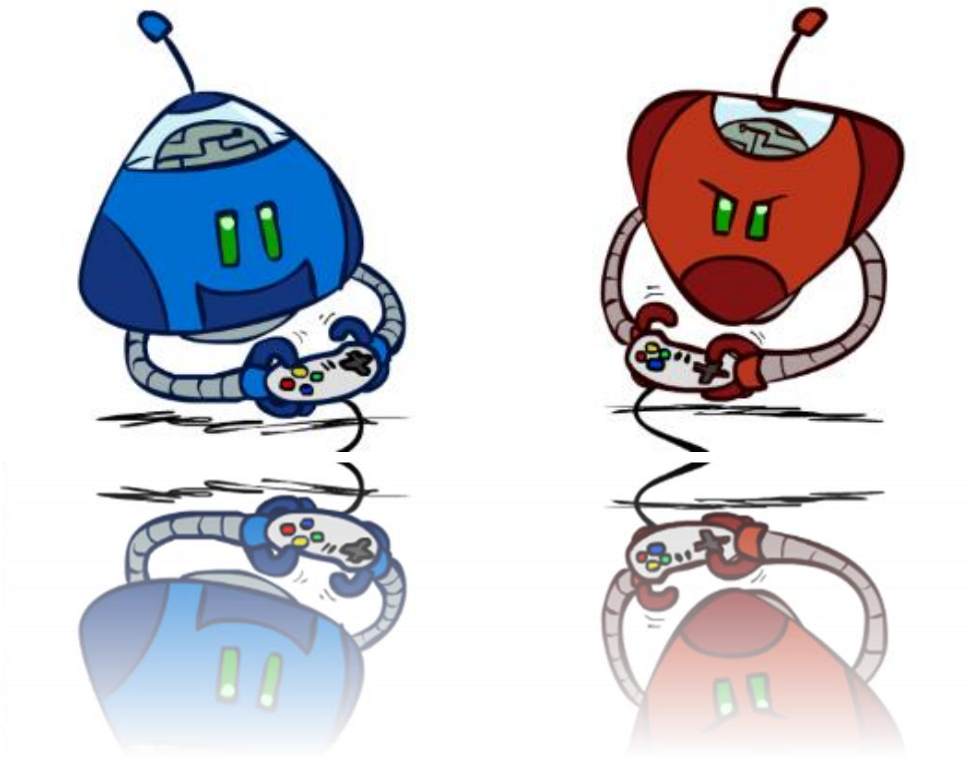


Chapitre IV.

RECHERCHE EN ADVERSITE

Introduction

Dans les environnements multiagents, chaque agent doit prendre en considération les actions de l'autre agent et comment, ces dernières, vont affecter son propre confort, ce qui crée une sorte de conflit entre les deux. On parle ici d'un monde dynamique, ou chaque adverse peut modifier l'environnement, les évènements seront, alors, imprévisibles.

Afin de résoudre ce genre de problèmes, on doit utiliser des algorithmes de recherche pour des jeux avec des adversaires (Adversial search) qu'on va introduire dans ce chapitre.

I. Les jeux

Selon la théorie des jeux, chaque environnement multiagents est considéré, comme étant un jeu, tant que l'impact d'un agent sur l'autre est considérable quelque soit la relation entre les deux, soit coopérative, soit compétitive.

Cependant, l'intelligence artificielle a un point de vu plus simple : un jeu peut être déterministe, alterne, a somme nulle ou bien a information parfaite.

1. Types de jeux

On peut classifier les jeux selon plusieurs critères :

- Déterministe ou stochastique ;
- Nombre de joueurs ;
- Type d'adversité : somme nulle ou à plusieurs valeurs
- Type d'information : Complete ou non

Cependant, dans ce cours, on va surtout étudie les jeux déterministes et à somme-nulle.

Qu'est ce que c'est, alors, un jeu déterministe ? A somme-nulle ?

- ***Les Jeux déterministes :***

Un jeu déterministe est un jeu que l'on peut appliquer un modèle mathématique afin de déterminer le résultat à un instant donne en se basant sur des paramètres comme :

- Les Etats (S) en commençant par un état initial S0 ;
- Les joueurs alternes (P = {1, ……, N}) ;
- Les actions (A) : chaque action dépend du joueur et de l'état ;
- La fonction de transition (S x A -> S)
- Le test d'arrêt (S -> {t , f}): indique si le jeu est terminé ;
- Les utilités (S x P -> R) : ce sont les fonctions objectives
- La solution d'un joueur est une stratégie qui associe un état à la meilleure action possible.

- *Les jeux à somme nulle :*

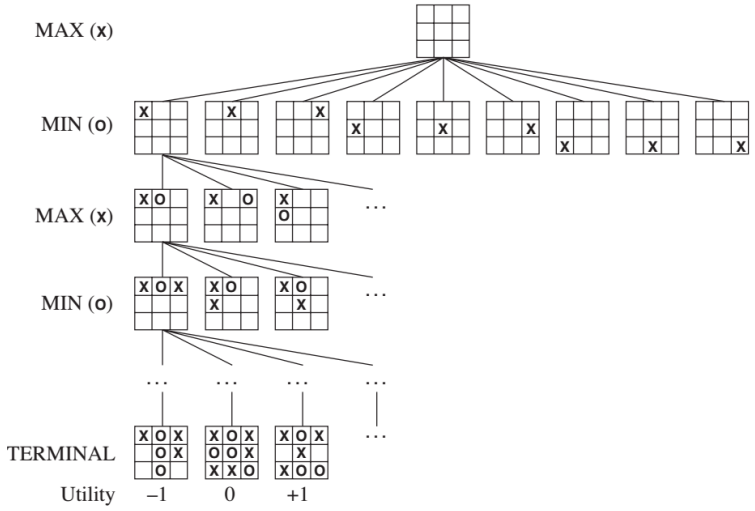
Par contre les jeux à différentes valeurs oú chaque agent possède sa propre fonction d'utilité, les jeux a somme-nulle se caractérise par une compétition pure entre les joueurs qui ont des utilités opposées. En effet, lorsqu'un agent cherche à maximiser la fonction d'utilité, l'autre essaie de minimiser cette dernière. (ex : jeu d'échecs).

2. Arbre de recherche :

Un problème de jeu peu être vu comme un problème de recherche dans un arbre ou les nœuds représentent les états, et les transitions représentent les mouvements.

Prenons l'exemple du jeu X-O :

- À l'état initial, MAX a 9 mouvements possibles
- Il place, donc, un X tandis que MIN place un O, alternativement, jusqu'on arrive a l'état terminal ou l'un des deux joueurs remplit le même ligne(colonne ou diagonale) ou bien toutes les cases sont remplisses.
- La valeur de feuille indique la valeur de l'utilité selon le point de vu de MAX



II. Algorithme MiniMax :

1. Principe :

A chaque tour, choisir l'action qui correspond a la plus grande valeur minimax, ce qui donne l'action la plus optimale contre un joueur

$$\text{EXPECTED-MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{Si } n \text{ est un nœud terminal} \\ \max_{s \in successors(n)} \text{EXPECTED-MINIMAX-VALUE}(s) & \text{Si } n \text{ est un nœud Max} \\ \min_{s \in successors(n)} \text{EXPECTED-MINIMAX-VALUE}(s) & \text{Si } n \text{ est un nœud Min} \end{cases}$$

2. Algorithme :

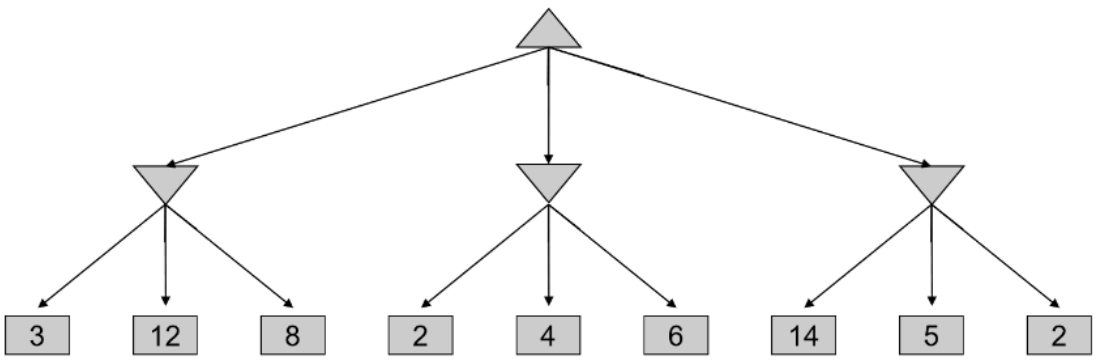
```
function DÉCISION-MINIMAX(état)
   v ← VALEUR-MAX(état)
   return action dans SUCCESSEURS(état) ayant la valeur v

function VALEUR-MAX(état)
   if TEST-TERMINAL(état) then
      return UTILITÉ(état)
   v ← −∞
   for a,s in SUCCESSEURS(état) do
      v ← MAX(v,VALEUR-MIN(s))
   return v

function VALEUR-MIN(état)
   if TEST-TERMINAL(état) then
      return UTILITÉ(état)
   v ← +∞
   for a,s in SUCCESSEURS(état) do
      v ← MIN(v,VALEUR-MAX(s))
   return v
```

3. Propriétés et Efficacité :

Il s'agit d'une exploration en profondeur d'abord complète de l'arbre de jeu. Il est :

- Complet si l'arbre est fini.
- Optimal contre un adversaire qui joue de façon optimale
- Sa complexité en temps = O(b^m), avec :
    - b : nombre maximal de coups (actions) légaux à chaque étape
    - m : nombre maximal de coups dans un jeu(profondeur maximale de l'arbre)
- sa complexité en espace = O(bm)

4. Exemple d'un arbre MinMax :



 Type MIN  Type MAX

- Les valeurs intermédiaires (deuxième niveau) sont de type MIN, donc les valeurs choisies seront le minimum des valeurs de chaque sous arbre ; ainsi on a :

La première valeur depuis la gauche serait MIN(3,12,8) = 3, la deuxième valeur MIN(2,4,6) = 2 et la troisième valeur MIN(14,5,2) = 2.

- La racine de l'arbre est de type MAX, et donc la valeur choisie serait le maximum ; ainsi on a :

Valeur de la racine est MAX(3,2,2) = 3

5. Limitation des ressources :

- *Jeu d'échecs : Problème de l'algorithme MinMax*

L'algorithme MinMax tend à être très lent pour des jeux comme les échecs. Dans ce jeu par exemple, pour chaque tour, le joueur a plusieurs choix à prendre, le facteur de branchement est très grand, et donc le plus profond qu'on va, plus de temps ça prend. En moyenne, un jeu d'échecs a un facteur de branchement proche de 30. Ce qui veut dire, qu'on crée 30 sous arbres à chaque mouvement pour trouver le nœud optimal. Ainsi cet algorithme est peu performant dans ces conditions.

Pour réduire le temps de recherche, on a proposé deux variantes de l'algorithme Minmax :

- *Recherche Minmax avec profondeur bornée:*

On explore l'arbre à une profondeur déterminée. Or ceci implique d'arrêter l'exploration avant d'avoir atteint les feuilles de l'arbre, ce qui nécessite de pouvoir évaluer l'état du jeu à un instant quelconque. On utilise donc une fonction heuristique h(s,A) qui évalue l'état s pour l'agent A. Cette fonction heuristique d'évaluation est théoriquement idéale avec une incertitude qui est égale à 0, mais en pratique ces valeurs ne sont qu'approximatives.

Dans le cours, on a vu des fonctions d'évaluation de sommes pondérées. Ainsi ces fonctions donnent des valeurs aux nœuds de l'arbre à n'importe quel moment dans le jeu et à n'importe quelle profondeur, ce qui permet de choisir le meilleur chemin possible en explorant à une profondeur limitée.

A noter que ces valeurs ne sont que des approximations, plus on explore plus profondément et plus la précision de ces valeurs est moins importante (le paradigme classique du **compromis entre la précision et la profondeur**).

- *Recherche Minimax avec coupes alpha-beta:* Introduite dans l'axe suivant

III. Algorithme MinMax avec élagage alpha-beta :

L'idée est de modifier l'algorithme de telle façon qu'on élague certaines branches de l'arbre sans risques. On distingue deux types de coupures : alpha et beta.

On considère un arbre dont la racine est de type MAX, en développant l'arbre sur le deuxième niveau, on pose une valeur ALPHA = première valeur développée, ainsi en trouvant une valeur inférieure, on sait qu'en aucun cas ce serait la valeur choisie, donc on élague le sous arbre de cette valeur ; ce type de coupure est dit alpha. En trouvant une valeur supérieure à la valeur dont on a initié ALPHA, ALPHA prend la nouvelle valeur et ainsi de suite.

Par analogie, on considère un arbre dont la racine est de type MIN, en développant l'arbre sur le deuxième niveau, on pose une valeur BETA = première valeur développée, ainsi en trouvant une valeur supérieure, on sait qu'en aucun cas ce serait la valeur choisie, donc on élague le sous arbre de cette valeur ; ce type de coupure est dit beta. En trouvant une valeur inférieure à la valeur dont on a initié BETA, BETA prend la nouvelle valeur et ainsi de suite.

Les algorithmes max-value et min-value vus en cours :

α: MAX'meilleure option
β: MIN meilleure option

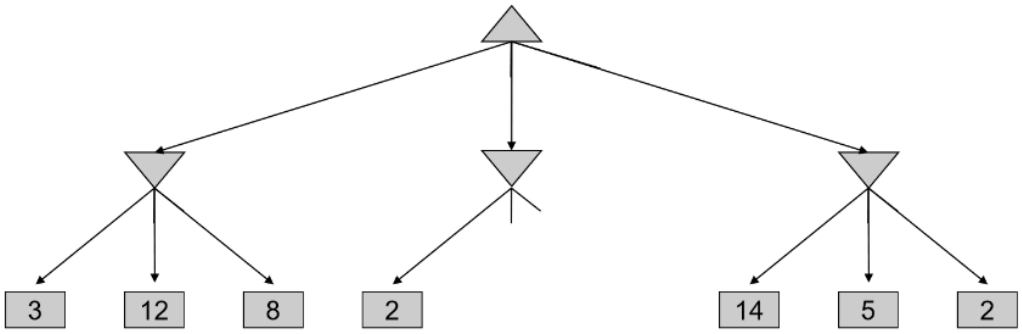
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

- A savoir que quelques soient les branches élaguées, la valeur de la racine reste la même. Cependant les valeurs des nœuds intermédiaires dans l'arbre peuvent être erronées. Ainsi, cette version n'est pas aussi performante pour **choisir des actions.**

- L'ordre des fils affecte directement l'efficacité de l'élagage. Pour un ordre parfait, on peut aller jusqu'à **doubler** la limite de profondeur initiale.

- Exemple de l'élagage alpha-beta :



Les valeurs intermédiaires sont de type MIN, donc les valeurs choisies seront les minimums de chaque sous arbre, ainsi la première valeur depuis la gauche serait un 3. En passant au second sous arbre, la première valeur rencontrée est un 2, puisqu'on choisira le minimum, la valeur choisie à ce niveau-là serait certainement inférieure ou égale à 2. Or la racine de l'arbre est de type MAX, donc la valeur choisie serait certainement supérieure ou égale à 3. Conséquemment on élague sans risques le reste du second sous arbre. Le temps de recherche est optimisé.

Remarque :

Ces optimisations de l'algorithme MinMax, bien qu'elles nous fassent gagner en matière de temps, nous font perdre en matière d'optimalité.