Chapitre 5

Théorie des jeux (partie 2)

Dans cette seconde et dernière partie du cours sur la théorie des jeux, nous allons essayer de trouver un algorithme qui permet de maximiser les gains dans un jeu d'accessibilité sans avoir à parcourir tous le graphe comme le calcul des attracteurs.

Table des matières

1 Heuristique **2**

2 Maximiser l'heuristique : l'algorithme MiniMax **3**

1 Heuristique

Dans ce chapitre, nous allons supposer que nous avons un jeu d'accessibilité et nous cherchons à établir une stratégie intelligente. Qu'entendons-nous par intelligente ? Il est difficile de donner une définition d'intelligence, en revanche, il est plus facile de comprendre ce qu'est une stratégie bête. Un exemple de stratégie bête pouvant être la stratégie qui étant donnée une position dans le jeu choisit un déplacement possible au hasard parmi tous les choix possibles. Pas possible de faire plus bête a priori.

À partir de là, on peut comprendre ce que doit faire une stratégie intelligente : elle doit, en particulier, être meilleure que la stratégie au hasard, autrement dit sur un grand nombre de parties, la stratégie intelligente doit battre très souvent la stratégie aléatoire. Car si ce que vous faites n'est pas mieux que le hasard, *what's the point ?*

Une méthode consiste à attribuer à une position de jeu un nombre de sorte que plus le nombre est grand plus on est susceptible de gagner. On cherche donc une fonction notée h qui à chaque position du graphe/de l'arène attribue un nombre. Par convention, si on est à une position gagnante, on note $h(p) = +\infty$ et si on est à une position perdante, on note $h(p) = -\infty$

Exemple 1.
- Aux échecs, on peut attribuer à chacune des pièces un nombre de points (classiquement 1 par pion, 3 par fou et cavalier, 5 par tour, 9 par dame et 0 pour le roi), et faire la différence entre ses points et ceux de l'adversaire. Plus vous avez un score élevé et plus vous avez de matériels que votre adversaire et donc vous êtes plus susceptibles de gagner.
- Au puissance 4, on peut attribuer une valeur à chaque case, plus une case peut être impliqué dans un grand nombre d'alignement de quatre pièces plus si vous y placez un jeton, vous êtes susceptible que ce pion vous serve à gagner (en clair, mieux vaut jouer au centre que dans les coins). On peut alors faire la somme des valeurs des emplacements occupés par vos jetons à laquelle vous retirez la somme des valeurs des emplacements de votre adversaire.

Évidemment, ces exemples de fonctions ne reflètent que très partiellement la réalité. On peut gagner une partie d'échecs avec moins de pièces que son adversaire si celles-ci sont bien placées. De même, à puissance 4, on peut très bien gagner avec un jeton dans le coin tandis que son adversaire qui a joué au milieu perd. Nous ne cherchons à décrire que partiellement la réalité.

Pour cette raison, on dit que la fonction h s'appelle une heuristique, elle va coder notre compréhension très partielle du jeu. Autrement dit, c'est une façon de coder nos préjugés sur le jeu en question. Notons que l'on a déjà vu un exemple d'utilisation de l'heuristique avec l'algorithme A* vu en PCSI : par rapport à l'algorithme de Dijkstra, en utilisant une heuristique (la distance à vol d'oiseau), on force l'algorithme à aller explorer certains sommets a priori plus intéressants que d'autres. Ici, c'est la même chose, l'heuristique va nous pousser à privilégier certaines positions car la valeur de ces sommets est plus élevé.

Codons l'exemple d'heuristique décrite pour le jeu de puissance 4. Pour cela, codons d'abord une fonction qui renvoie un tableau (une liste de liste) notée `Nb` telle que `Nb[i][j]` ait pour valeur le nombre d'alignements de 4 pièces passant à la i-ième ligne et la j-ième colonne. Pour cela, pour chaque alignement de quatre pièces, (ici `puissance` est une variable dont la valeur est 4). On fait `Nb[i][j]=Nb[i][j]+1` pour tous les positions `(i,j)` de l'alignement :

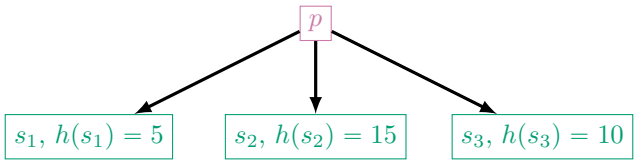
```
def Poids():
    Nb=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            for (x,y) in [(1,0),(0,1),(1,1),(1,-1)]:
                if EstDansCadre(i+(puissance-1)*x,j+(puissance-1)*y):
                    for k in range(puissance):
                        Nb[i+k*x][j+k*y]=Nb[i+k*x][j+k*y]+1
    return Nb
Nb=Poids()
```

Ensuite, codons l'heuristique, il s'agit d'un calcul de somme :

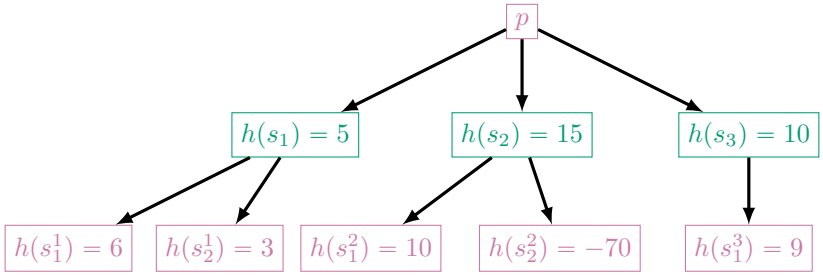
```
def H(T,joueur):
    S=0
    for i in range(n):
        for j in range(p):
            if T[i][j]==joueur:
                S=S+Nb[i][j]
            if T[i][j]==3-joueur:
                S=S-Nb[i][j]
    return S
```

2 Maximiser l'heuristique : l'algorithme MiniMax

Supposons que l'on soit dans un jeu d'accessibilité à une position p et qu'on dispose d'une heuristique h :



Alors, il semblerait naturel de joue la position B car c'est elle qui a la plus grande heuristique, mais ce serait jouer de façon court-termiste. En effet, maintenant, c'est à l'adversaire de jouer et peut-être qu'il va pouvoir jouer à partir de B un coup qui fera diminuer drastiquement l'heuristique. Pour voir quoi faire, reprenons le graphe avec plus de profondeur :



Sur cet exemple, on voit que si on joue le coup s_2, alors l'adversaire, s'il fait le bon choix, peut choisir un coup qui nous laissera avec une heuristique de -70 et donc où est susceptible de perdre. En effet si l'adversaire a la même vision du jeu, il aura intérêt à jouer le coup telle que notre heuristique soit la plus faible possible. Ici, on cherche donc le coup parmi s_1, s_2 et s_3 tels que le minimum des successeurs soit le plus grand possible. Ainsi, on cherche le maximum des minima des heuristiques. Ainsi, c'est le coup s_3 que l'on a intérêt de jouer.

On pourrait continuer comme ça pendant longtemps jusqu'à explorer une partie jusqu'à la victoire ou la défaite ou le match nul. Mais la puissance de calcul est limité. Ainsi, on peut définir une profondeur maximale. L'algorithme va consister à maximiser un minimum de maximum de minimum de maximum etc. avec autant de maxima/minima que la profondeur l'autorise. Pour cette raison, on appelle cette algorithme `MiniMax` il est fondé sur les principes suivants :

- `MiniMax(p) = h(p)` si `p` est un sommet qui a atteint la profondeur `pro` où h est l'heuristique.
- `MiniMax(p) = max(minimax(s1),..., minimax(sn))` si `p` est un sommet controlé par le joueur avec comme successeurs `s1,..., sn`.
- `MiniMax(p) = min(minimax(s1),..., minimax(sn))` si `p` est un sommet contrôlé par l'adversaire avec comme successeurs `s1,..., sn`.

On remarque alors qu'un maximum avec une profondeur de `pro` est un maximum de minimum calculé avec une profondeur de `pro`-1, de même un minimum avec une profondeur de `pro` est un minimum de maximum calculé avec une profondeur de `pro`-1.

Lorsque la profondeur vaut 0, alors on évalue la position grâce à l'heuristique. Sans oublier les cas de fin de parties (victoires/défaites/nulles). On a alors toutes les cartes en main [1] pour programmer l'algorithme MiniMax en Python :

1. Les cartes ne sont pourtant pas très utiles pour jouer à Puissance4...

```
def maximin(T,L,joueur,pro):
    if pro==0 or CoupsPossibles(L)==[]:
        return (H(T,joueur),None)
    maxi=-np.inf
    for col in CoupsPossibles(L):
        lig=L[col]
        Jouer(T,L,joueur,col)
        if CoupGagnant(T,lig,col,joueur):
            m=np.inf
        else:
            m=minimax(T,L,joueur,pro-1)[0]
        if m>=maxi:
            maxi=m
            c=col
        Annulercoup(T,L,col)
    return (maxi,c)

def minimax(T,L,joueur,pro):
    if pro==0 or CoupsPossibles(L)==[]:
        return (H(T,joueur),None)
    mini=np.inf
    for col in CoupsPossibles(L):
        lig=L[col]
        Jouer(T,L,3-joueur,col)
        if CoupGagnant(T,lig,col,3-joueur):
            m=-np.inf
        else:
            m=maximin(T,L,joueur,pro-1)[0]
        if m<=mini:
            mini=m
            c=col
        Annulercoup(T,L,col)
    return (mini,c)
```

L'algorithme peut être long à exécuter pour deux raisons : à chaque étape suivant le type de jeu, il peut avoir beaucoup de coups possibles à regarder, de plus, plus la profondeur est grande plus la complexité sera importante. La mémoïsation pourrait être utile car on peut aboutir à la même position avec des ordre de coups différents, mais cela complexifierait encore l'algorithme. Il existe aussi d'autres méthodes qui n'explorent que certaines parties de l'arbre, mais celles-ci sont hors programme.

Notons que pour un jeu comme Puissance 4, pour explorer l'ensemble du jeu, comme il y a 35 coups possibles, il faudrait une profondeur de 35 ce qui est beaucoup trop. Dans les faits, on prend donc une profondeur qui donne un résultat acceptable à défaut d'être parfaitement précis.

Une difficulté est de trouver une heuristique qui soit le plus fidèle à la probabilité de gagner. Dans les faits, on peut faire affronter plusieurs heuristiques entre elles et prendre celle qui gagne le plus souvent contre les autres.