Leçon 16 : Exemples d'algorithmes pour l'étude des jeux

Prérequis : Graphe ; Arbre ; Principe d'induction

Niveau : MPI

Motivation 1 : La théorie des jeux s'intéresse aux interactions entre des individus (joueurs) qui effectuent des choix selon les règles d'un jeu. ⇒ Applications en sociologie, économie

I - Jeux d'accessibilité à deux joueurs

I.1 - Définitions

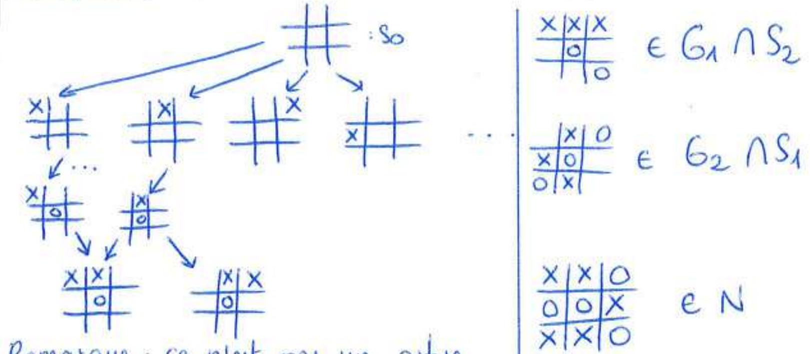
Notation 2 : Pour $G = (S, A)$, on note $Fin(G) = \{v \in S \mid deg^+(v) = 0\}$.

Def 3 : Un jeu à deux joueurs est :

- une arène : un graphe biparti $G = (S_1 \sqcup S_2, A)$
- un sommet initial s_0
- une partition de $Fin(G)$ en G_1, G_2, N

Intuition 4 : Les sommets de S_1 sont ceux où J_1 joue. Un arc $x \to y$ correspond à un coup possible. G_1 sont les états gagnants pour J_1, G_2 ceux pour J_2, N ceux d'un match nul.

Exemple 5 : Modélisation du tic-tac-toe :



Remarque : ce n'est pas un arbre.

Def 4 (Partie) : Une partie d'un jeu (G, s_0, G_1, G_2, N) est un chemin de s_0 à un sommet $s_f \in Fin(G)$.

Def 5 (Stratégie) : On appelle stratégie pour le joueur $i \in \{1, 2\}$ toute fonction $\varphi : V_i \to V$ tq $\forall u \in V_i \setminus Fin(G)$, $(u, \varphi(u)) \in A$.

φ est une stratégie gagnante si pour toute partie $P = s_0, \dots, s_f$, $(\forall j \in [\![0, f-1]\!],\ s_j \in V_i \Rightarrow s_{j+1} = \varphi(s_j)) \Rightarrow s_f \in G_i$

s est une position gagnante s'il existe une stratégie gagnante depuis s.

Intuition 6 : φ est une stratégie gagnante si elle assure la victoire à J_i quels que soient les coups joués par son adversaire.

I.2 - Attracteurs

On se place du point de vue du joueur 1, mais la situation est symétrique avec le joueur 2.

Définition 7 (Attracteur) : Soit (G, s_0) une arène, et $F \subseteq V$. On note $Attr_i(F)$ l'ensemble des sommets depuis lesquels J_1 a une stratégie gagnante pour arriver en F en au plus i étapes.

Propriété 8 :

- $Attr_0(F) = F$
- $Attr_{i+1}(F) = Attr_i(F) \cup \{u \in V_1 \mid N^+(u) \cap Attr_i(F) \neq \emptyset\} \cup \{u \in V_2 \mid N^+(u) \subset Attr_i(F)\}$

Propriété 9 : $(Attr_i(F))$ est stationnaire, on note $Attr(F)$ sa limite.

Stratégie gagnante depuis $Attr(F)$:

$$\varphi : V_1 \longrightarrow V$$
$$v \in Attr_{i+1}(F) \setminus Attr_i(F) \longmapsto w \in N^+(v) \cap Attr_i(F)$$

Propriété 10 : Pour un jeu (G, s_0, G_1, G_2, N), $Attr(G_1)$ est l'ensemble des positions gagnantes de J_1.

Développement 1 : Stratégies gagnantes pour le jeu de Nim à 1 puis plusieurs tas.

II - Jeux Min-Max

II.1 - Algorithme Min-Max

On considère ici des jeux tq la déf 3, en remplaçant la partition de Fin(G) par une fonction de coût $c : Fin(G) \to \mathbb{Z}$. Le joueur 1 (appelé Max) essaye de maximiser la valeur finale, et le joueur 2 (appelé Min) de la minimiser.

Remarque 11 : La partie précédente est un cas particulier avec $c(G_1) = 1$, $c(N) = 0$, $c(G_2) = -1$.

définition 12 : Une stratégie optimale pour le joueur Max (resp. Min) est une stratégie maximisant (resp. minimisant) le coût.

Algorithme 13 : On fait une recherche exhaustive de tous les coups en prenant à chaque fois celui ayant le résultat max (resp. min) quand Max (resp. Min) joue. Cela détermine une stratégie optimale.

Remarque 14 : En pratique souvent infaisable tant le graphe est gros.

Exemple : Les échecs avec +1000 victoire blanche, -1000 victoire noire et 0 nulle, le graphe a plus de 10^{44} sommets.

Déf 15 : Une heuristique $h : V \to \mathbb{Z}$ est une estimation du coût de l'état.

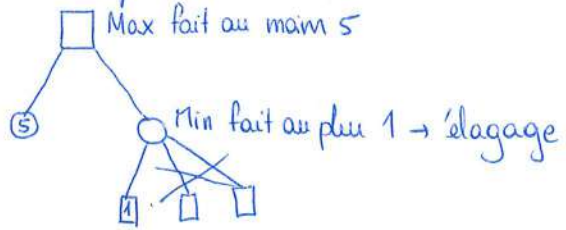
Exemple 16 : La fonction qui aux échecs vaut la différence entre les sommes des valeurs des pièces des Max et celle des pièces de Min.

Algo 17 : Min-Max (j, depth, u) :

```
Si depth == 0 :
    return h(u)
f = max si j == 1 et min sinon
return f({Min-Max(3-j, depth+1, v) | v ∈ N⁺(u)})
```

II.2 - Élagage α-β

Idée 18 : Il n'est pas nécessaire d'explorer tout l'arbre : Si je suis Min et qu'au coup précédent Max peut faire 5, dès que je vois que je peux faire moins, je peux arrêter d'explorer car Max ne fera pas ce coup.



Algorithme 19 :

```
Alphabeta(j, alpha, beta, u) : //On appelle Alphabeta(1, -∞, +∞, s0)
  res = -∞
  if joueur == 1 : //Cas joueur Max
    Pour v voisin de u :
      e = alphabeta(3-j, max(res, alpha), beta, v)
      Si e > beta : retourner e   //élagage
      Sinon : res = max(e, res)
  Sinon : //Symétrique, à faire en exercice
```

Remarque 20 : Cet algorithme est exact. On peut, comme pour Min-Max, ajouter une profondeur et une heuristique.

Remarque 21 : α (resp. β) représente la valeur de la meilleure option trouvée pour l'instant pour le joueur Max (resp Min)

TP 22 : Comparaison des temps d'exécution de l'algo Min-Max et Alpha-Beta pour le jeu du tic-tac-toe (exploration complète sans heuristique).

III - Les jeux à un joueur

III.1 - Graphe d'état

Def 23: Un graphe d'état est la donnée d'un graphe orienté $G = (S, A)$ pondéré par $c : A \to \mathbb{N}$, d'un état initial s_0 et d'un ensemble d'états finaux $F \subseteq S$.

Remarque 24: S représente les configurations d'un jeu à 1 joueur, A les coups possibles, c le coût d'un coup et F les configurations gagnantes.

Exemple 25: Trouver un chemin entre l'état initial et un des états finaux minimisant le coût total.

Exemple 26: Dans le jeu du taquin, les états sont les dispositions possible du plateau. L'état final est le plateau remis dans l'ordre. Chaque case a un degré sortant ≤ 4 qui correspond aux déplacements possibles de la case vide (vers le haut, le bas, à gauche, à droite). Tous les déplacements ont un coût unitaire.

Remarque 27: Le graphe d'état est en général très gros. Il est donc nécessaire de mettre en place des stratégies pour orienter la recherche du chemin.

III.2 - L'algorithme A*

Principe 28: A* est une variante de Dijkstra pour calculer un plus court chemin entre un sommet s_0 et un sommet s_f. On visite les sommets par estimation de leur proximité à s_f grâce à une fonction f définie par :

$f(s) = d(s) + h(s)$ où - $d(s)$: coût d'un
- $h(s)$: estimation du coût entre s et s_f

Exemple 29: Dans le jeu du taquin, h peut être :
- nbr de chiffres mal placés
- somme des distances de Manhattan des cases à leur position finale.

Algorithme 30: A*(G, s_0, s_f, h) :

```
D = tableau initialisé à +∞
D[s0] = 0
P = file de priorité vide
ajouter (s0, h[s0]) à P
Tant que (P non vide) :
    s, _ = extract(P)
    Si s == sf : retourner d[s]
    Pour s' successeur de s :
        Si D[s] + w[s, s'] < D[s'] :
            D[s'] = D[s] + w[s, s']
            Ajouter (s', D[s'] + h[s']) à P
Retourner 'Not Found'
```

Remarque 31: Si $h = 0$, on retrouve Dijkstra

Def 32: Une heuristique est admissible si $\forall u \in S$, $h(u) \leq \mathrm{dist}(u, s_f)$

Théorème 33: Si h est admissible, A* renvoie la distance d'un pcc de s_0 à s_f s'il existe, et Not Found sinon.

Def 34: h est monotone si $\forall (u,v) \in A$, $h(u) \leq h(v) + w(u,v)$

Propriété 35: Si h est monotone et $h(s_f) = 0$, alors A* est correct et extrait chaque nœud au plus une fois.

Développement 2: Démonstration du théorème 33 et de la propriété 35.