Table des matières

Première partie

Leçons

Leçon 1

Exemples d'algorithmes pour l'étude des jeux

Auteur·e·s: Emile Martinez
Références :

La théorie des jeux s'intéresse aux interactions entre des individus (joueurs) qui effectuent des choix selon les règles d'un jeu.

1 Jeux d'accessibilité à deux joueurs

I Définition

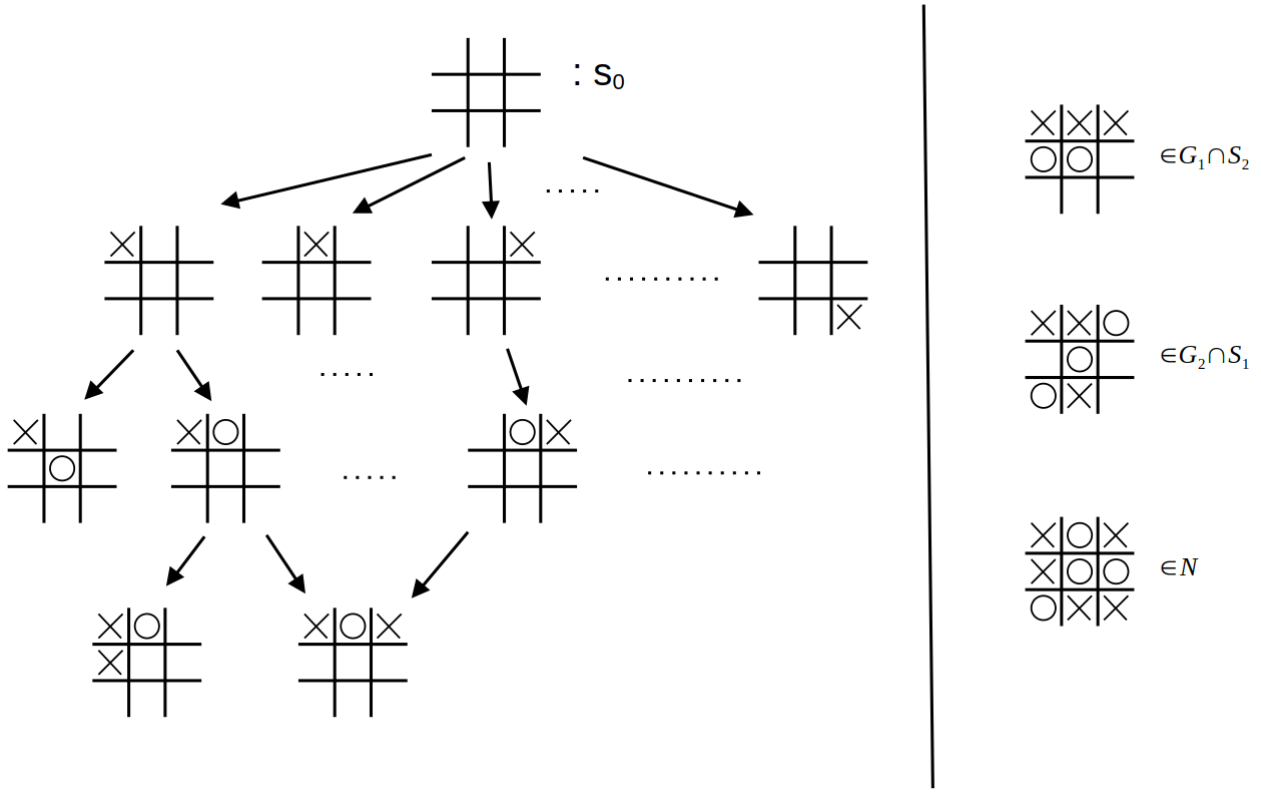
Notation : Pour $G = (S, A)$, on note $Fin(G) = \{v \in S / deg^+(v) = 0\}$

Définition 1.1 *Un jeu à deux joueurs est :*
- *une arène : un graphe orienté biparti* $G = (S_1 \sqcup S_2, A)$
- *un sommet de départ* s_0
- *une partition de* $Fin(G) = G_1 \sqcup G_2 \sqcup N$

Commentaire 1.1 *On peut mentionner ici qu'on peut se restreindre au graphes bipartie, car si quand on joue, c'est à nouveau à nous, on considère les deux coups à jouer d'affilée comme un seul. Mais y a des jeux où on peut rejouer, il faut alors travailler un peu pour modéliser.*

Idée 1.1 *Les sommets de* S_1 *sont ceux où J1 joue. Un arc* $x \to y$ *représente un coup possible pour le joueur qui contrôle l'état* x*, qui déplace alors le jeu dans l'état* y*.* G_i *sont les états gagnants pour Ji,* N *ceux d'une partie nulle.*

Exemple 1.1 *Représentation d'un échantillon de la modélisation du Tic-Tac-Toe*



Définition 1.2 (Partie) *Une partie d'un jeu (G, s_0, G_1, G_2, N) est un chemin de s_0 à un sommet $s_f \in Fin(G)$.*

Définition 1.3 *On appelle stratégie pour le joueur $i \in \{1, 2\}$ toute fonction $\varphi : V_i \to V$ tel que $\forall u \in V_i \backslash Fin(G)$, $(u, \varphi(u)) \in A$.*

Commentaire 1.2 *Ici on définit directement une stratégie sans mémoire, le programme se limitant à cela, et les jeux que nous considérerons ne nécessitant pas de stratégie avec mémoire.*

Définition 1.4 (Stratégie gagnante) *φ est une stratégie gagnante pour le joueur i si pour toute partie $P = s_0, \ldots, s_f$,*

$$\Big(\forall j \in [\![O, f-1]\!], s_j \in V_i \implies s_{j+1} = \varphi(s_j)\Big) \implies s_f \in G_i$$

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

Idée 1.2 *φ est une stratégie gagnante si quand les coups de Ji sont ceux de φ, alors Ji gagne indépendamment de ce que joue l'autre joueur.*

II Attracteurs

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

Définition 1.5 (Attracteur) *Pour une arène (G, s_0), et $F \subset V$ on note $Attr_i(F)$, l'ensemble des sommets depuis lesquels le joueur J1 a une stratégie pour arriver dans F en au plus i étapes. On note* $Attr(F) = \bigcup_{i=0}^{+\infty} Attr_i(F)$

Proposition 1.1 *Pour un jeu (G, s_0, G_1, G_2, N), $Attr(G_1)$ est l'ensemble des position gagnante de J1.*

Proposition 1.2

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

Proposition 1.3 *Si G est fini alors $(Attr_i(F))$ est croissante bornée, donc stationnaire. Sa limite est alors $Attr(F)$.*

Une stratégie gagnante depuis $Attr(F)$ est :

$$\begin{array}{rcl} \varphi : V_1 & \to & V \\ v & \mapsto & \begin{cases} \omega \in \mathcal{N}^+(v) \cap Attr_i(F) & \text{si } v \in Attr_{i+1}(F) \backslash Attr_i(F) \\ \omega \in \mathcal{N}^+(v) & \text{si } v \notin Attr(F) \end{cases} \end{array}$$

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

2 Jeux Min-Max

I Algorithme Min-Max

On considère ici des jeux à deux joueurs, en reprenant la définition 1.1 en remplaçant la partition de $Fin(G)$ par une fonction de coût $c : Fin(G) \to \mathbb{Z}$.

Le joueur 1 (appelé Max ici) essaye alors de maximiser par ses coups la valeur finale, et le joueur 2 (ici Min) essaye de la minimiser.

Remarque 1.1 *La partie précédente en est un cas particulier avec $c(G_1) = 1$, $c(N) = 0$ et $C(G_2) = 1$*

Définition 1.6 *Une stratégie optimale pour le joueur Max (resp. Min) est une stratégie maximisant (resp. minimisant) le coût.*

Algorithme 1.1 *On fait une recherche exhaustive de tous les coups, en prenant à chaque fois celui ayant le résultat maximum (resp.minimimum) quand Max (resp. Min) joue. Cela détermine une stratégie optimale (pour les deux joueurs).*

Remarque 1.2 *En pratique, c'est souvent infaisable tant le graphe est gros.*

Exemple 1.2 *Les echecs, avec +1000 victoire blanche, -1000 victoire noire et 0 nulle, le graphe a plus de* $10^4 4$ *sommets.*

Définition 1.7 *Une heuristique* $h : V \to \mathbb{Z}$ *est une estimation de à quoi mènerait dans le meilleur cas cette position*

Commentaire 1.3 *Ici on a une définition formelle avec l'intuition de son interprétation (et donc de son usage). (une heuristique, c'est simplement une fonction* $V \to \mathbb{Z}$*)*

Exemple 1.3 *La fonction qui aux échecs donne la somme de la valeur des pièces*

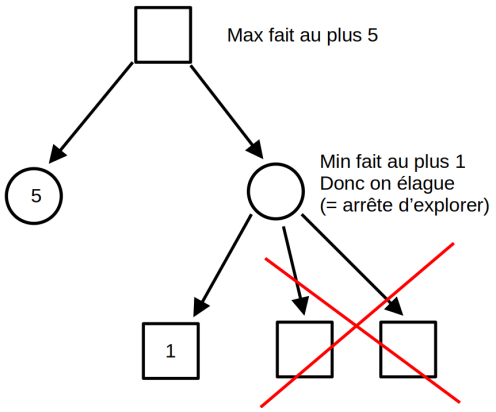
Idée 1.3 *On fait l'exploration exhaustive en renvoyant l'heuristique quand on a fait suffisament d'étapes.*

Algorithme 1.1 : $MinMax(j, prof, u)$

si $u \in Fin(G)$ ou $prof = 0$ **alors**
  retourner $h(u)$
si $i == 1$ **alors**
  $f \leftarrow \max$
sinon
  $f \leftarrow \min$
retourner $f\Big(\Big\{MinMax(3-j, prof-1, v) \ / \ v \in \mathcal{N}^+(u)\Big\}\Big)$

II Élagage α-β

Idée 1.4 *Il n'est pas nécessaire d'explorer tout l'arbre : si je suis Min et que Max au coup d'avant peut faire 5 avec un autre coup, dès que je vois que je peux faire 1, je peux arrêter d'explorer, car je sais que Max ne fera pas ce coup.*



Algorithme 1.2 : $Alphabeta(j, \alpha, \beta, u)$

```
si u ∈ Fin(G) alors
    retourner c(u)
si joueur = 1 alors
    res ← −∞
    pour v voisin de u faire
        e = Alphabeta(2, max(res, α), β, v)
        si e > β alors
            retourner e   // Élagage
        sinon
            res ← max(e, res)
    retourner res
sinon
    // Symétrique, à faire en exercice
```

Idée 1.5 *α (resp. β) est la valeur maximale (resp. minimale) que peut faire Max (resp. Min) parmi ce que l'on a explorer pour l'instant. (On appelle donc $alphabeta(1, -\infty, +\infty, s_0)$)*

Remarque 1.3 *Cet algorithme est exact. On peut, comme pour Min-Max ajouter une profondeur et une heuristique.*

Exercice 1.1 *Comparaison des temps d'exécution de Min-Max et de Alpha-Beta pour le Tic-Tac-Toe (exploration complète sans heuristique).*

3 Les Jeux à un joueur

I Graphe d'état

Définition 1.8 *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 $F \subset S$*

Remarque 1.4 *S représente les configurations d'un jeu à un joueur, A les transitions d'une configuration à une autre en un coup, c le coût de cette transition, et F les configurations gagnantes.*

Objectif : Trouver un chemin dans ce graphe entre l'état initial et l'un des états finaux de coût total minimal.

Exemple 1.4 *Dans le jeu du taquin, les états correspondent aux dispositions possibles du plateau. L'état final est le plateau remis dans l'ordre. Chaque case a un degré sortant inférieur ou égal à 4 qui correspond aux déplacements possibles de la case vide (vers le haut, le bas, à gauche ou à droite). Tous les déplacements ont un coût unitaire.*

Remarque 1.5 *le graphe d'état est en général trop gros pour être stocké entièrement en mémoire. Il est donc nécessaire de mettre en place des stratégies ou des heuristiques pour orienter la recherche du chemin*

II L'algorithme A*

Principe 1.1 *L'algorithme A* est une variante de l'algorithme de Dijkstra pour calculer un plus court chemin entre un sommet initial s_0 et un sommet final 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 $d(s)$ est le coût d'un plus court chemin entre s_0 et s et $h(s)$ i une estimation (heuristique) du coût entre s et s_f*

Exemple 1.5 *dans le cas du taquin, on peut penser aux heuristiques suivantes :*

- *nombre de chiffres mal placés*
- *somme des distances de Manhattan des cases à leur position finale*

Algorithme 1.3 : Algorithme A*

Entrées : W la matrice de poids du graphe ; h le tableau pour l'heuristique ; s_0 et s_f les sommets initiaux et finaux
Sorties : la distance d'un plus court chemin de s_0 à s_f

```
D ← tableau initialisé à ∞
D[s_0] ← 0
P ← file de priorité vide
Ajouter (s_0, h[s_0]) à P
tant que P non vide faire
    (s, _) ← extraire(P)
    si s = s_f alors
        retourner D[s]
    pour s' successeur de s faire
        c ← D[s] + W[s, s']
        si c < D[s'] alors
            D[s'] ← c
            Ajouter (s', c + h[s']) à P
retourner NonAccessible
```

Remarque 1.6 *Si $h = 0$, on retrouve Dijkstra*

Définition 1.9 *Une heuristique est dite admissible si $\forall u \in S, h(u) \leq d(u)$*

Théorème 1.1 (Correction) *Si h est admissible, alors A* renvoie la distance d'un court chemin de s_0 à s_f.*

Définition 1.10 *Une heuristique est dite monotone si $\forall (u, v) \in A, h(u) \leq h(v) + w(u, v)$*

Remarque 1.7 *Dans un graphe avec les distances euclidiennes entre nœuds, la distance à vol d'oiseau est une heuristique monotone.*

Proposition 1.4 *Si h est monotone et $h(s_f) = 0$, alors A* est correct (h est admissible) et extrait chaque noeud au plus une fois.*

Développement : Démonstration du théorème et de la propriété précédente.

Application 1.1 *Calcul des itinéraires par un GPS*