Jeux d'accessibilité

INFORMATIQUE COMMUNE - TP n° 3.7 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ coder le calcul de attracteur pour un jeu d'accessibilité
- ☞ appliquer l'algorithme Minimax sur un jeu simple

A Jeu de la soustraction

Le jeu de la soustraction est un jeu à deux joueurs. Devant eux se trouvent N bâtonnets [1]. Les joueurs jouent l'un après l'autre et ont le droit de retirer un, deux ou trois bâtonnets. Le gagnant est celui qui tire le dernier [2] bâtonnet.

A1. Le jeu de la soustraction est-il un jeu d'accessibilité? Pourquoi?

A2. Jouer avec votre voisin en prenant sept bâtonnets.

A3. Combien de positions possibles existe-t-il pour cette partie à sept bâtonnets?

A4. Construire sur le papier l'arène de ce jeu. On choisit la convention suivante : les sommets contrôlés par le premier joueur sont numérotés de 0 à n, ceux du second joueur de $n+1$ à $2n+1$.

A5. Calculer à la main l'attracteur du premier joueur et le reporter sur la figure précédente.

A6. Que fait le code suivant?

```python
def mystery_code(n):
  size = n + 1
  a = [[] for _ in range(2 * size)]
  for i in range(size):
    for j in range(1, 4):
      if i - j >= 0:
        a[i].append(size + i - j)
        a[i + size].append(i - j)
  return a
```

Un indice : pour $n = 7$, cette fonction renvoie :

```
[[], [8], [9, 8], [10, 9, 8], [11, 10, 9], [12, 11, 10], [13, 12, 11], [14, 13,
   12], [], [0], [1, 0], [2, 1, 0], [3, 2, 1], [4, 3, 2], [5, 4, 3], [6, 5, 4]]
```

A7. Écrire une fonction Python qui renvoie le transposé d'un graphe orienté, c'est à dire le graphe dont tous les arcs sont inversés. Cette fonction aura pour prototype `g_transpose(g)` où `g` est un graphe sous la forme d'une liste d'adjacence. Elle renvoie un graphe sous la forme d'une liste d'adjacence. Par exemple, pour l'arène du jeu à sept bâtonnets, elle renvoie :

1. On peut y jouer avec des pièces, des stylos ou des allumettes. . .
2. La variante misère en fait le perdant.

```
[[9, 10, 11], [10, 11, 12], [11, 12, 13], [12, 13, 14], [13, 14, 15], [14, 15],
    [15], [], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6], [5, 6, 7], [6, 7],
    [7], []]
```

A8. Écrire une fonction Python qui calcule le degré entrant d'un graphe. Cette fonction est de prototype `in_degrees(g)` et renvoie une liste d'entiers. Le degré du sommet 0 est à l'indice 0 de cette liste... Par exemple, pour l'arène du jeu à sept bâtonnets transposée, elle renvoie :

```
 [0, 1, 2, 3, 3, 3, 3, 3, 0, 1, 2, 3, 3, 3, 3, 3]
```

Pour trouver l'attracteur d'un joueur, il faut parcourir l'arène de jeu en inversant les arcs, c'est à dire en transposant le graphe. Ainsi, en partant de la conditions de gain C_g et en remontant les arcs, on parvient à trouver $\mathcal{A}$. On rappelle les détails de la procédure sur l'algorithme 1.

Algorithme 1 Calcul de l'attracteur d'un joueur

Entrée : g le graphe biparti de l'arène de jeu
Entrée : V l'ensemble de sommets du joueur
Entrée : C_g la condition de gain du joueur

Fonction ATTRACTEUR(g, V, C_g)
    $\mathcal{A} \leftarrow \emptyset$ ▷ l'attracteur
    $g^t \leftarrow$ le transposé du graphe g ▷ Pour remonter le graphe
    $d_{in} \leftarrow$ tableau des degrés entrants du graphe transposé ▷ Pour compter ce qui entre
    pour chaque sommet $v \in C_g$ **répéter** ▷ On part de la condition de gain
        AUGMENTER_ATTRACTEUR($v, \mathcal{A}, g^t, d_{in}, V$)
    renvoyer $\mathcal{A}$
Fonction AUGMENTER_ATTRACTEUR($v, \mathcal{A}, g^t, d_{in}, V$)
    si $v \notin \mathcal{A}$ **alors**
        $\mathcal{A} \leftarrow \mathcal{A} \cup \{v\}$
        pour chaque voisin u de v **répéter** ▷ Pour remonter le graphe
            $d_{in}[u] \leftarrow d_{in}[u] - 1$ ▷ On passe par ce sommet une fois depuis $\mathcal{A}$
            si $u \in V$ ou $d_{in}[u] = 0$ **alors** ▷ Soit $u \in V$ soit tous ses arcs entrant viennent de $\mathcal{A}$
                AUGMENTER_ATTRACTEUR($u, \mathcal{A}, g^t, d_{in}, V$)

A9. Coder une fonction qui calcule l'attracteur d'un joueur. Elle aura pour prototype `attractor(a, cg)` où `a` l'est l'arène du jeu et `cg` la condition de gain d'un joueur. On implémentera les ensembles de sommets (attracteur, condition de gain ou l'ensemble des sommets appartenant à un joueur) par des type `set`.

A10. Tester sur les jeux de la soustraction de dimension 5, 7, 13, 15 et 20.

A11. Coder une fonction qui donne la stratégie gagnante, c'est à dire le prochain sommet dans l'attracteur ou bien `None` s'il n'y a pas de solution. Son prototype est
`next_in_attractor(a, att, p)`
où `a` est l'arène de jeu, `att` l'attracteur du joueur et `p` est la position sur l'arène. On remarquera que la position appartient forcément à un joueur et il faut donc fournir l'attracteur qui correspond à ce joueur.

A12. Coder un script pour jouer contre l'ordinateur.

Est-ce un jeu d'accessibilité ? Le modéliser Générer automatiquement le graphe Calculer les attracteurs

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

0	1	2	3	4	5
6	7	8	X	X	X
12	13	14	X	X	X

FIGURE 1 – Tablettes de chocolat et jeu de Chomp : après avoir pris le carré numéro 9, le joueur a mangé tous les X.

B Minimax et chocolat

Le jeu de Chomp est le jeu des amateurs de chocolat et de roulette russe. Deux gourmets se partagent une tablette de chocolat rectangulaire prédécoupée en $N \times M$ carrés. L'un après l'autre, chaque joueur désigne un carré et la mange ainsi que tous les carrés se trouvant à droite et au dessous de ce carré. Le carré de chocolat situé en haut et à gauche est en or (ou empoisonné[3]). Le joueur qui prend le dernier carré de chocolat a gagné (ou perdu). Pour la suite de ce TP, on choisit la version misère avec le carré empoisonné, c'est plus fun.

B1. Le jeu de Chomp est-il un jeu d'accessibilité ? Pourquoi ?

On implémente une tablette de cholocat par un type `set`. Pour créer un type `set` on utiliser le constructeur `set()` ou bien `{}`. Ces structures sont mutables et non ordonnées.

Chaque carré de chocolat est un tuple à deux éléments `(i,j)` représentant la ligne `i` et la colonne `j` d'un carré de chocolat.

B2. Écrire une fonction de prototype `make_tab(n, m)` qui renvoie l'ensemble de tous les carrés de chocolat d'une tablette de taille `n` par `m`. Par exemple, pour `n=3` et `m=6`, la fonction renvoie :

```
{(0, 1), (1, 2), (0, 0), (1, 1), (0, 2), (1, 0)}
```

B3. Monter que pour $m = 1$ et $n \geqslant 2$ ou $n = 1$ et $m \geqslant 2$, il existe une stratégie gagnante pour le premier joueur.

B4. Monter que pour $m = n \geqslant 2$, c'est à dire une tablette carrée, il existe une stratégie gagnante pour le premier joueur.

B5. Écrire une fonction de prototype `eatset(tab, i,j)` qui renvoie la nouvelle tablette après que le joueur a mangé le carré d'indice `(i,j)` comme expliqué sur la figure 1.

B6. Écrire une fonction de prototype `showtab(tab)` qui affiche une tablette de chocolat sur la console. Le caractère unicode "Ž588" permet de faire afficher une bloc rectangulaire noir. Par exemple, pour la tablette `{(0, 1), (0, 0), (1, 1), (0, 2), (1, 0)}`, la fonction affiche sur la console :

```
█ █ █
█ █      ----> 5 squares
```

L'arène du jeu de Chomp n'est pas simple à concevoir car Lorsque `N` et `M` augmente le nombre de positions du jeu explose rapidement. On cherche donc à résoudre le problème différemment en utilisant l'approche Minimax dont on rappelle l'algorithme ci-dessous.

---

3. à votre convenance, le plaisir avant tout !

Algorithme 2 Minimax

Entrée : p un position dans l'arbre de jeu (un nœu de l'arbre)
Entrée : s la fonction de score sur les feuilles
Entrée : $\mathcal{H}$ l'heuristique de calcul du score pour un nœud interne
Entrée : L la profondeur maximale de l'arbre Minimax

Fonction MINIMAX(p, s, $\mathcal{H}$, L)
  si p est une feuille **alors**
    renvoyer $s(p)$
  si $L = 0$ **alors**
    renvoyer $\mathcal{H}(p)$ ▷ On arrête d'explorer, on estime
  si p est contrôlé par J_{max} **alors**
    $M \leftarrow -\infty$
    p_M un nœud vide
    pour chaque fils f de p **répéter**
      $v \leftarrow$MINIMAX(f,s,$\mathcal{H}$, $L-1$) ▷ v est un score de J_{min}
      si v > M **alors**
        $M \leftarrow v$
        $p_M = f$
    renvoyer M, p_M ▷ Valeur maximale trouvée et la racine de cette solution
  sinon
    $m \leftarrow +\infty$
    p_m un nœud vide
    pour chaque fils f de p **répéter**
      $v \leftarrow$MINIMAX(f,s,$\mathcal{H}$, $L-1$) ▷ v est un score de J_{max}
      si v < m **alors**
        $m \leftarrow v$
        $p_m = f$
    renvoyer m, p_m ▷ Valeur minimale trouvée et la racine de cette solution

Pour des dimensions petites, l'abre minimax peut être parcouru en entier. On fera donc abstraction de l'heuristique dans un premier temps : lorsque la profondeur de l'arbre maximale sera atteinte, on peut lever une exception, par exemple avec l'instruction `assert L`=0, "Max depth reached!" !.

B7. Code une fonction `minimax(tab, L=5, player_is_max=True)`. Les paramètres sont `tab`, l'ensemble des carrés d'une tablette de chocolats, `L` la profondeur d'exploration de l'arbre maximale et `player_is_max` un booléen qui indique si c'est le premier joueur `max` qui joue. Cette fonction renvoie un tuple composé du score maximal atteignable et de la position correspondante qui conduit à ce score.

B8. Pour une tablette de chocolat de 3x6, quelle est la profondeur minimale d'exploration qu'il faut choisir afin que l'ordinateur puisse gagner tout le temps ?

Il est possible de limite la profondeur d'exploration grâce à la technique de l'élagage $\alpha\beta$. Dans le cas du jeu de Chomp, comme le score maximal est +1, il suffit d'observer que, dès qu'on a trouvé un score de 1 en parcourant les fils, on ne trouvera pas mieux. On fait la même observation pour le score minimal de -1.

B9. Implémenter cette amélioration dans code de la fonction `minimax`.

B10. En utilisant la technique de memoïsation issue de la programmation dynamique, accélérer les calculs de l'algorithme précédent.

B11. Afin de limiter la profondeur d'exploration nécessaire, proposer et implémenter une heuristique pour le jeu de Chomp.