Jeux de la soustraction

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
- ☞ réutiliser les concepts de graphe et de parcours en largeur

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 ?

> **Solution :** Oui, car c'est un jeu à deux joueurs séquentiel (les joueurs jouent l'un après l'autre) à information parfaite (tous les coups déjà joués sont connus de tous) et complète (tous les bâtonnets sont visibles) pour lequel il n'y a pas de hasard (on ne peut en choisir que un, deux ou trois).

A2. Jouer avec votre voisin en prenant sept bâtonnets.

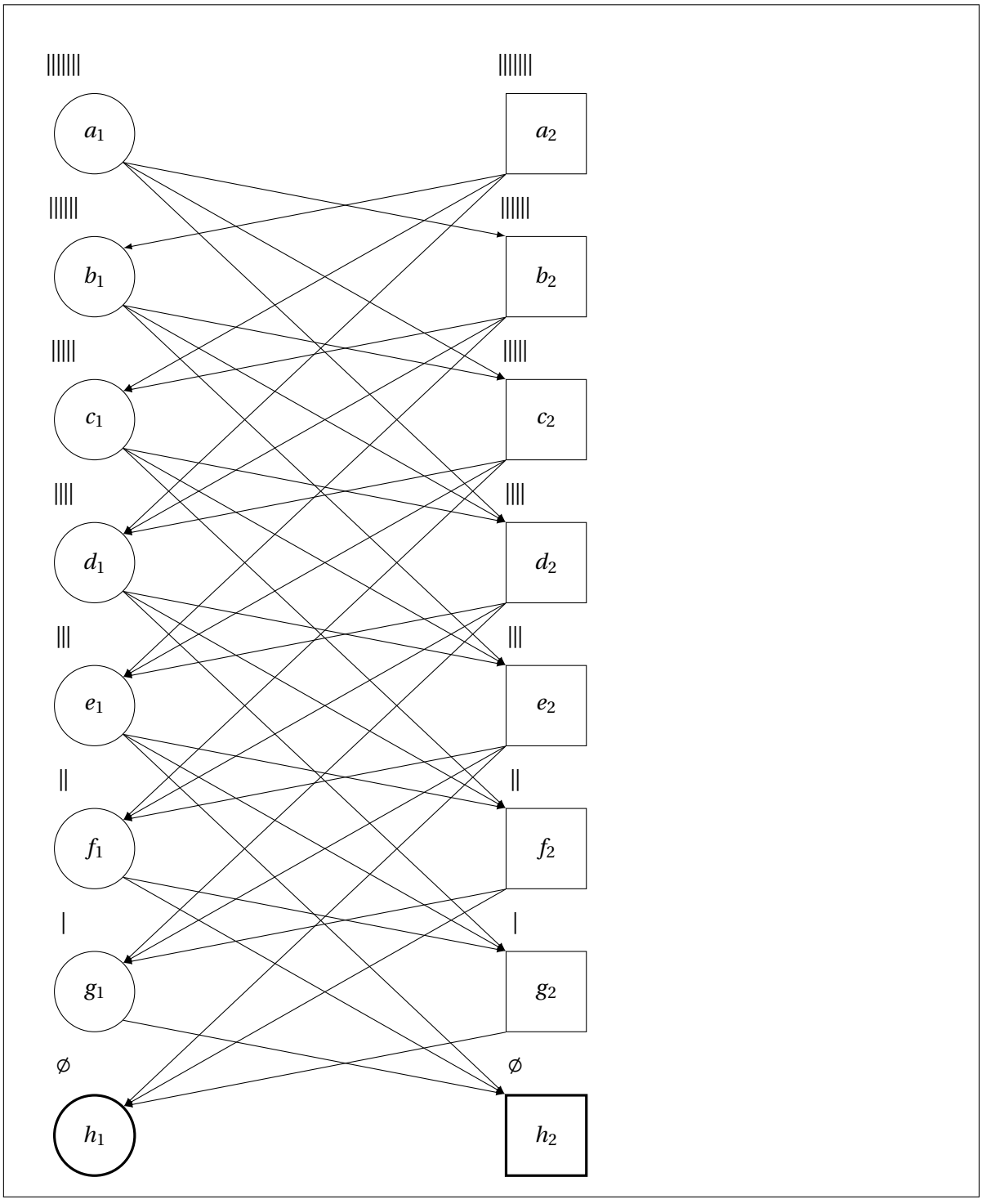
A3. Combien de positions possibles existe-t-il pour cette partie à sept bâtonnets ?

> **Solution :** Sur la table, il peut rester 0,1,2,3,4,5,6 ou 7 bâtonnets. Il existe donc huit positions potentielles par joueur sur l'arène de jeu.

A4. Construire sur le papier l'arène de ce jeu pour $n = 7$. 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 $2n+1$.

> **Solution :** Modélisation par graphe orienté biparti d'un jeu de soustraction à sept bâtonnets. Les sommets des joueurs 1 et 2 sont distingués par des cercles (1) et des carrés (2).

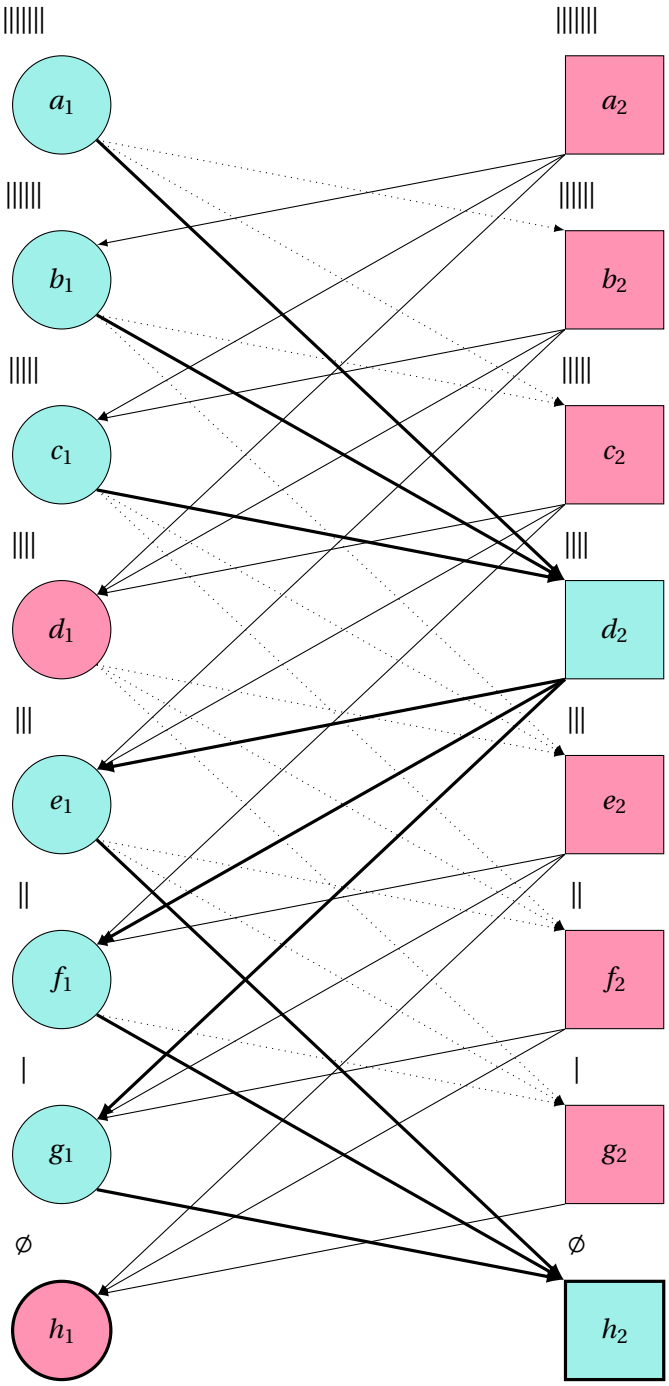
1. On peut y jouer avec des pièces, des stylos ou des allumettes...
2. La variante misère en fait le perdant.



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

Solution : Modélisation par graphe orienté biparti d'un jeu de soustraction à sept bâtonnets. Les sommets des joueurs 1 et 2 sont distingués par des cercles (1) et des carrés (2). L'attracteur

du joueur 1 est en cyan.



A6. Que fait le code suivant ?

```
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]]
```

Solution : Ce code permet de construire l'arène d'un jeu de soustraction à n bâtonnets sous la forme d'un graphe d'adjacence.

B Attracteur

Pour trouver l'attracteur $\mathcal{A}$ d'un joueur, il suffit de parcourir l'arène de jeu en partant de sa condition de gain et en inversant les arcs, c'est-à-dire en parcourant le graphe transposé. C'est l'algorithme que l'on développe dans cette section.

a Fonction utiles

B7. Écrire une fonction de signature `players_vertices(n : int)-> (list[int], list[int])` qui renvoie les listes des sommets du joueur 1 et du joueur 2. Par exemple, `players_vertices(7)` renvoie le tuple `([0, 1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14, 15])`.

Solution :

```
def player_vertices(n):
    return [i for i in range(n + 1)], [i for i in range(n + 1, 2 * n + 2)]
```

B8. Écrire une fonction de signature `gain_condition(n: int)-> (int, int)` qui renvoie les conditions de gain du joueur 1 et du joueur 2. Par exemple `gain_condition(7)` renvoie `(8,0)`.

Solution :

```
def gain_condition(n):
    return n + 1, 0
```

B9. Écrire une fonction de signature `start_positions(n: int)` qui renvoie les positions de départ des joueurs 1 et 2 sur l'arène de jeu. Par exemple, `start_positions(7)` renvoie `(7,15)`.

Solution :

```
def start_positions(n):
    return n, 2 * n + 1
```

B10. Écrire une fonction de signature `previous_positions(p: int, n:i nt)-> list[int]` dont les paramètres sont `p` un sommet du graphe et `n` le nombre de bâtonnets. Cette fonction renvoie la liste des sommets qui conduisent à `p` dans l'arène du jeu.

Solution :

```
def previous_positions(p, n):
    pos_list = []
    if p <= n:  # p appartient au joueur 1
        p1 = n + 1 + p + 1
        p2 = n + 1 + p + 2
        p3 = n + 1 + p + 3
        for pos in [p1, p2, p3]:
            if pos <= 2 * n + 1:
                pos_list.append(pos)
    else:          # p appartient au joueur 2
        p1 = p % (n + 1) + 1
        p2 = p % (n + 1) + 2
        p3 = p % (n + 1) + 3
        for pos in [p1, p2, p3]:
            if pos <= n:
                pos_list.append(pos)
    return pos_list
```

b Calcul de l'attracteur d'un joueur

L'algorithme 1 détaille comment parcourir l'arène de jeu transposée afin de trouver l'attracteur d'un joueur. La remarque fondamentale est la suivante : lorsqu'on découvre un nouveau sommet depuis l'attracteur :

- soit ce sommet est un sommet du joueur et on peut l'ajouter à l'attracteur sans risques,
- soit ce sommet est un sommet de l'adversaire. Dans ce cas, on ne peut l'ajouter à l'attracteur que si les arcs qui en sortent ne conduisent qu'à des sommets de l'attracteur (cas où l'adversaire est captif).

Le rang d'un sommet est mémorisé en même tant que le sommet : il permet de choisir une stratégie gagnante. Lorsque le rang d'un sommet diminue, on se rapproche de la victoire.

B11. Écrire une fonction de signature `build_attractor(n, player)` qui renvoie l'attracteur du joueur 1 si `player` vaut 0 et 2 sinon dans le cas du jeu de la soustraction. Cette fonction parcourt en largeur l'arène de jeu transposée. Elle renvoie une liste de tuples (sommet, rang) qui représente l'attracteur. Par exemple, `build_attractor(7,0)` renvoie la liste `[(9, 0), (1, 1), (2, 1), (3, 1), (13, 2), (5, 3), (6, 3), (7, 3), (17, 4)]`

Algorithme 1 Calcul de l'attracteur d'un joueur, jeu de la soustraction

```
1 : Fonction ATTRACTEUR(n)
2 :     V ← l'ensemble de sommets du joueur dont on calcule l'attracteur
3 :     C_g ← la condition de gain du joueur
4 :     s_1 et s_2 les positions de départ des joueurs
5 :     A ← ∅                                                    ▷ l'attracteur
6 :     d_in ← les degrés entrants des sommets l'arène de jeu transposé
7 :     r ← 0 le rang initiale (de la condition de gain)
8 :     F ← une file d'attente
9 :     ENFILER(F, (C_g, r))                ▷ On va parcourir en largeur le graphe transposé
10 :    tant que F n'est pas vide répéter
11 :        u, r ← DÉFILER(F)
12 :        AJOUTER(A, (u, r))                                   ▷ augmentation de l'attracteur
13 :        si u est un sommet de départ alors                   ▷ Exploration achevée, early exit
14 :            renvoyer A          ▷ L'attracteur du joueur comporte nécessairement soit s_1 soit s_2
15 :        pour chaque sommet à v conduisant à u répéter
16 :            Décrémenter le degré de v dans d_in          ▷ On l'a découvert une fois depuis l'attracteur
17 :            si v appartient au joueur ou si on ne peut arriver à v que depuis l'attracteur alors
18 :                ENFILER(F, (v, r + 1))
```

Solution :

```python
def in_degrees(n):
    degrees = [3 for _ in range(2 * n + 2)]  # degrés entrant dans le graphe
        transposé
    degrees[0], degrees[n + 1] = 0, 0  # initialisation des noeuds 0,1,2
    degrees[1], degrees[n + 2] = 1, 1
    degrees[2], degrees[n + 3] = 2, 2
    return degrees


def build_attractor(n, player=0):
    Cg = gain_condition(n)[player]
    # position finale de gain de la partie
    s1, s2 = start_positions(n)
    # position de départ des joueurs
    V = player_vertices(n)[player]
    # les sommets du joueur dont on calcule l'attracteur
    din = in_degrees(n)
    A = []  # attracteur
    rank = 0
    F = [(Cg, 0)]
    while len(F) > 0:
        u, rank = F.pop(0)  #
        # parcours en largeur
        A.append((u, rank))
        # le rang auquel on a trouvé le sommet (sert pour définir la
            stratégie)
        # print(u, rank, din)
        if u == s1 or u == s2:
```

```
                # On a fini d'explorer le graphe # EARLY EXIT
                return A
            for v in previous_positions(u, n):
                din[v] -= 1
                # On vient de l'attracteur, donc on décrémente
                if v in V or din[v] == 0:
                    # soit dans V soit on ne vient QUE de l'attracteur
                    F.append((v, rank + 1))  # enfiler
```

B12. Écrire une fonction de signature `in_attractor(A, pos)` qui renvoie, si `pos` est une position de l'attracteur du joueur, l'**index** de cette position dans la liste `A` et le **rang** de cette position. Si ce n'est pas le cas, la fonction renvoie `(None, None)`.

Solution :

```
def in_attractor(A, pos):
    for k in range(len(A)):
        v, r = A[k]
        if pos == v:
            return k, r
    return None, None
```

B13. Écrire une fonction de signature `next_in_attractor(A, pos)` qui permet d'implémenter la stratégie gagnante. Si `pos` est une position de l'attracteur du joueur, la fonction renvoie la position jouable suivante dans l'attracteur, c'est-à-dire telle que le rang est plus petit. Sinon la fonction renvoie `None`.

Solution :

```
def next_in_attractor(A, pos):
    k, rank = in_attractor(A, pos)
    if rank:
        for i in range(k, -1, -1):
            v, r = A[i]
            if r < rank:
                return v
    else:
        return None
```

C Programmation du jeu humain vs. ordinateur

On dispose d'un programme implémentant le jeu de la soustraction. L'humain joue en premier. L'ordinateur utilise les fonction précédentes qui se trouve dans une fichier (module) nommé `attractor.py`.

C14. Jouer et vérifier que vous ne pouvez que perdre lorsque n est pair.

C15. Jouer et vérifier que vous pouvez gagner lorsque n est impair.