# 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 ?

> **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.

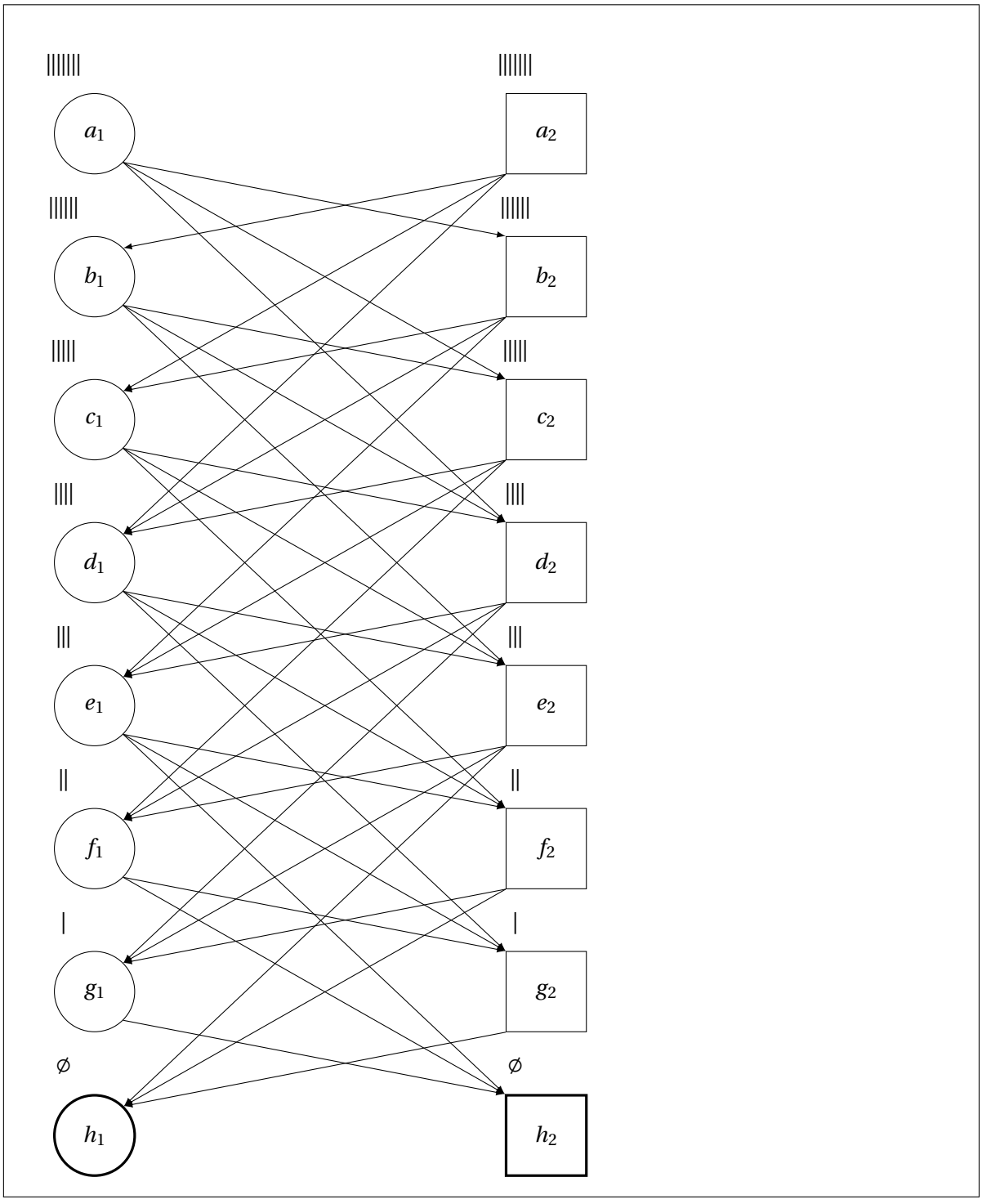
A3. Combien de positions possibles existe-t-il pour cette partie à sept bâtonnets ?

> **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. 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).

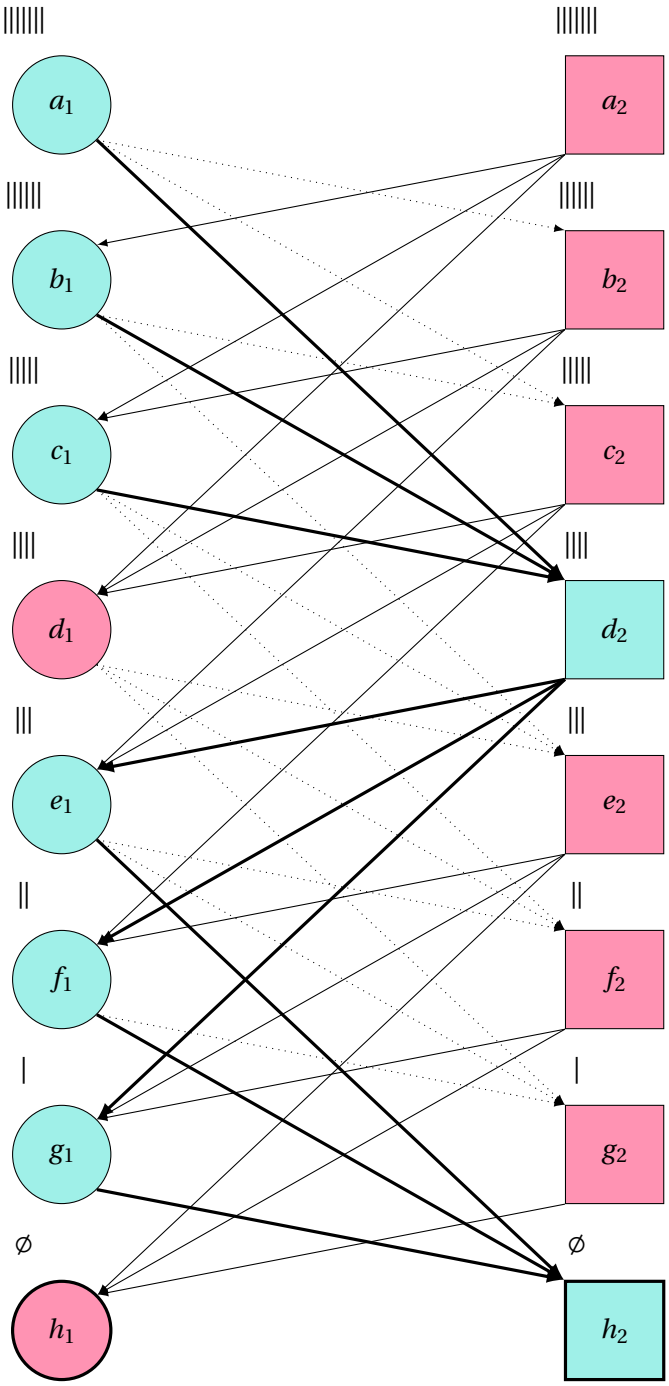
1. On peut y jouer avec des pièces, des stylos ou des allumettes...
2. La variante misère en fait le perdant.




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.

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 :

```
[[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], []]
```

**Solution :**

```
def g_transpose(g):
    tg = [[] for _ in range(len(g))]
    for i in range(len(g)):
        for j in g[i]:
            tg[j].append(i)
    return tg
```

A8. Écrire une fonction Python qui calcule le degré entrant d'un graphe. Par exemple, pour l'arène du jeu à cinq bâtonnets, cette fonction renvoie :

```
    {6: 3, 7: 3, 8: 3, 9: 2, 10: 1, 0: 3, 1: 3, 2: 3, 3: 2, 4: 1}
```

Cette fonction est de prototype `in_degrees(g)` et renvoie un dictionnaire dont les clefs sont les entiers de la liste `g` et les valeurs le nombre d'occurrences de ces entiers. En effet, le graphe étant modélisé par une liste d'adjacence, le nombre d'occurrences des sommets dans la liste traduit directement le degré entrant de chaque sommet. Il est conseillé d'utiliser un dictionnaire pour compter les éléments et donc les degrés entrants.

**Solution :**

```
def in_degrees(g):
    counts = {}
    for adj in g:
```

```python
        for i in adj:
            # counts[i] = counts.get(i, 0) + 1
            if i in counts:
                counts[i] += 1
            else:
                counts[i] = 1
    return counts
```

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 dont on calcule l'attracteur
**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
  $d_{in} \leftarrow$ dictionnaire des degrés entrants du graphe transposé
  $r \leftarrow 0$ le rang de la condition de gain
  **pour** chaque sommet $v \in C_g$ **répéter** ▷ On part de la condition de gain
    AUGMENTER_ATTRACTEUR($v, \mathcal{A}, g^t, d_{in}, V, r$) ▷
  **renvoyer** $\mathcal{A}$
**Fonction** AUGMENTER_ATTRACTEUR($v, \mathcal{A}, g^t, d_{in}, V, r$)
  **si** $v \notin \mathcal{A}$ **alors**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{v\}$
    **pour** chaque voisin $u$ de $v$ dans $g^t$ **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($u, \mathcal{A}, g^t, d_{in}, V, r+1$)

A9. Coder une fonction qui calcule l'attracteur d'un joueur. Elle aura pour prototype `attractor(g, V, Cg)` où `g` l'est l'arène du jeu, `V` l'ensemble des sommets du joueur et `cg` sa condition de gain. On implémentera l'attracteur et la condition de gain par de dictionnaire dont la clef est le sommet et la valeur le rang du sommet. L'ensemble des sommets appartenant à un joueur peut être un type `list` ou un type `set`.

**Solution :**

```python
def attract_grow_with(v, A, tg, dsg, V, rank):
    if v not in A:  # if not already in attractor
        A[v] = rank
        for u in tg[v]:  # up in the graph if neighbours exist
            dsg[u] -= 1  # decr. since we come from A_j
            if u in V or dsg[u] == 0:
```

```
                # v in V1 OR (v in V2 and the only way out is towards A_j)
                # print("IN --> ", u)
                attract_grow_with(u, A, tg, dsg, V, rank + 1)
            # else:
                # print("OUT --> ", u)


def attractor(g, V, Cg):
    n = len(g)
    A = {}  # attractor
    tg = g_transpose(g)  # graphe transposé
    dsg = in_degrees(tg)  # degrés entrants de TG
    rank = 0
    for v in Cg:
        attract_grow_with(v, A, tg, dsg, V, rank)
    return A

if __name__ == "__main__":
    arene = [[], [6], [6,7], [6,7,8], [7, 8, 9], [8, 9, 10],
    [], [0], [0,1], [0,1,2], [1, 2, 3], [2, 3, 4]]
    V1 = {0, 1, 2, 3, 4, 5}  # PLAYER 1 VERTEX SET
    V2 = {6, 7, 8, 9, 10, 11}  # PLAYER 2 VERTEX SET
    Cg1 = {0: 0}
    Cg2 = {6: 0}

    print(in_degrees(arene))

    a = attractor(arene, V1, Cg1)
    print("Attractor of Player 1 is -> ", a)

    b = attractor(arene, V2, Cg2)
    print("Attractor of Player 2 is -> ", b)
```

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.

**Solution :**

```
def next_in_attractor(a, att, p):
    v_next = []
    for v in a[p]:
        if v in att:
            v_next.append((att[v], v))
    if v_next: # une solution existe
        return min(v_next)[1] # choisir celle de rang minimum
    else:
        return None # pas de prochaine position dans l'attracteur
```

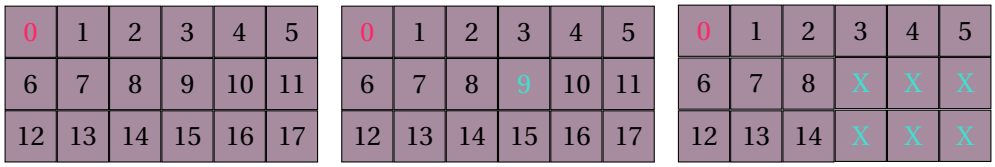



FIGURE 1 – Tablettes de chocolat et jeu de Chomp : après avoir pris le carré numéro 9, le joueur a mangé tous les X.

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

**Solution :**

```
cf. site
```

## 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 ?

**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 carrés de chocolat sont visibles) pour lequel il n'y a pas de hasard (on peut choisir n'importe quel carré restant dans la tablette).

On implémente[4] 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)}
```

3. à votre convenance, le plaisir avant tout !
4. Cette implémentation est discutable, on pourrait utiliser un tableau Numpy.

**Solution :**

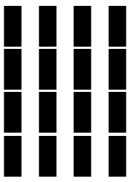
```
1  def make_tab(n, m):
2    return {(i, j) for i in range(n) for j in range(m)}
```

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.

**Solution :** Il suffit de choisir le carré situé juste à côté du carré empoisonné.

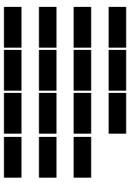
B4. Monter que pour $m = n \geqslant 2$, c'est à dire une tablette carrée, il existe une stratégie gagnante pour le premier joueur.

**Solution :** il suffit que le premier joueur prenne le carré (1,1), juste en dessous à droite du carré empoisonné. Ensuite, il reste une ligne et une colonne qui comporte le même nombre de carrés $k$. Le premier joueur n'a qu'à placer le second joueur dans cette position, où il y a toujours autant de carrés sur la ligne et la colonne : à chaque tour, il suffit qu'il prenne autant de carré que le second joueur. Voici un exemple de partie :



```
      ----> 16 squares

You took : (3, 3)
```

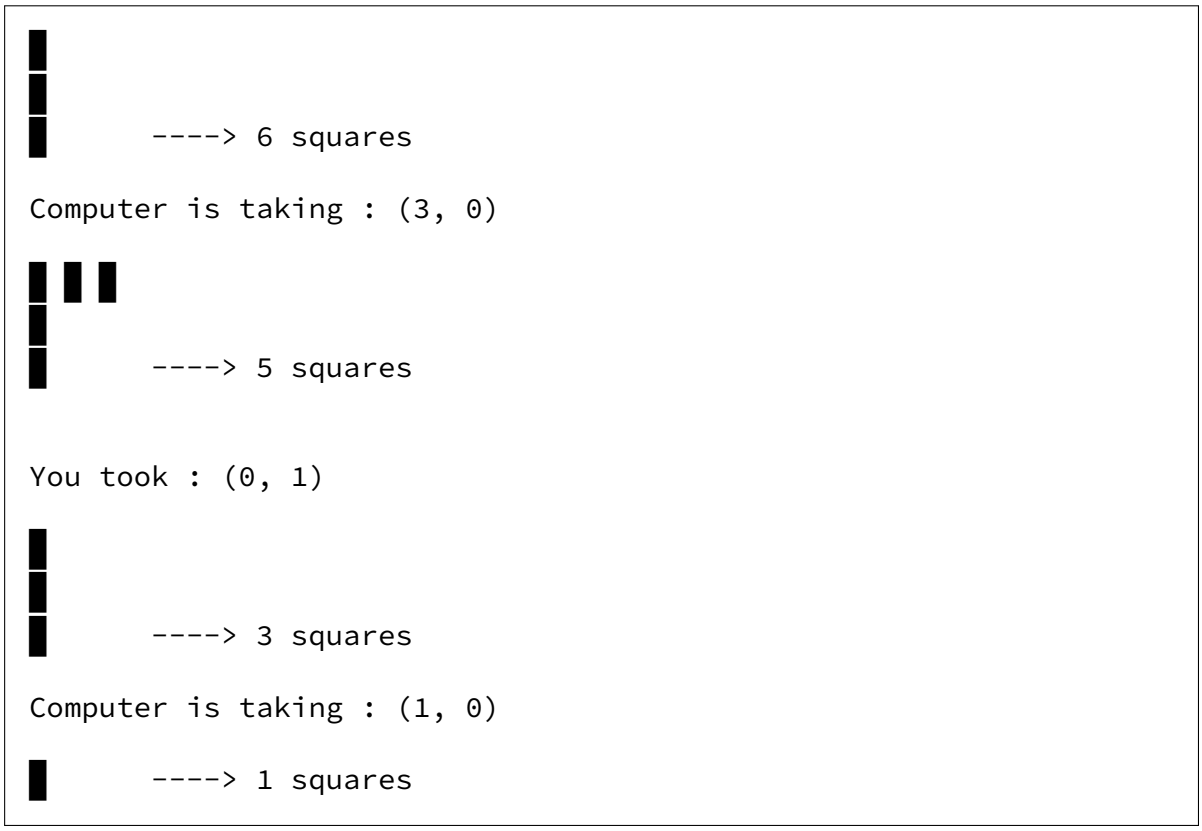


```
     ----> 15 squares

Computer is taking : (1, 1)
```

```
     ----> 7 squares

You took : (0, 3)
```

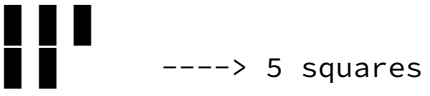



B5. Écrire une fonction de prototype `eatset(tab, i,j)` qui renvoie la nouvelle tablette après que le joueur a mangé le carré d'indice `(i,j)` comme expliqué sur la figure 1.

**Solution :**

```
1  def eat(tab, i, j):
2    eaten_tab = set()
3    for (pi, pj) in tab:
4      if pi < i or pj < j:
5        eaten_tab.add((pi, pj))
6    return eaten_tab
```

B6. Écrire une fonction de prototype `showtab(tab)` qui affiche une tablette de chocolat sur la console. Le caractère unicode `"\u2588"` 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 :




L'arène du jeu de Chomp n'est pas simple à concevoir car, lorsque `N` et `M` augmentent, 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.

Pour des dimensions petites, l'arbre minimax peut être parcouru en entier. On fera donc abstraction de l'heuristique dans un premier temps.

**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

```
1 : Fonction MINIMAX(p, s, H , L)
2 :     si p est une feuille alors
3 :         renvoyer s(p)
4 :     si L = 0 alors
5 :         renvoyer H(p)                          ▷ On arrête d'explorer, on estime
6 :     si p est contrôlé par J_max alors
7 :         M ← −∞
8 :         p_M un nœud vide
9 :         pour chaque fils f de p répéter
10 :            v ←MINIMAX(f,s,H, L − 1)            ▷ v est un score de J_min
11 :            si v > M alors
12 :                M ← v
13 :                p_M = f
14 :        renvoyer M, p_M                        ▷ Valeur maximale trouvée et la racine de cette solution
15 :    sinon
16 :        m ← +∞
17 :        p_m un nœud vide
18 :        pour chaque fils f de p répéter
19 :            v ←MINIMAX(f,s,H, L − 1)            ▷ v est un score de J_max
20 :            si v < m alors
21 :                m ← v
22 :                p_m = f
23 :        renvoyer m, p_m                        ▷ Valeur minimale trouvée et la racine de cette solution
```

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 ?

**Solution :** Il faut une profondeur de 16 au minimum.

Il est possible de limiter la profondeur d'exploration grâce à la technique de l'élagage $\alpha\beta$. Dans le cas du jeu de Chomp, le score maximal est +1 et le score minimal de -1 : soit le premier joueur gagne, soit c'est le second. Il suffit donc 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.

**Solution :** Supposons que l'algorithme soit arrivé à la profondeur maximale. On peut essayer de qualifier la solution ou pas pour le joueur :

1. si la tablette de chocolat est une ligne ou une colonne, alors on sait qu'on possède une stratégie gagnante pour le joueur. On peut donc renvoyer 1.
2. si la tablette est un composée d'une ligne et d'une colonne, alors on sait qu'on possède une stratégie gagnante pour le joueur. On peut donc renvoyer 1.
3. si la tablette est un carré, alors on sait qu'on possède une stratégie gagnante pour le joueur. On peut donc renvoyer 1.
4. dans les autres cas, on peut considérer que l'autre gagne...

Cette heuristique est insuffisante mais c'est un exemple pour commencer.

On pourrait également construire des indicateurs propres à une position et conjecturer sur les valeurs à prendre de ces indicateurs ou faire appel à l'algorithme de Monte-Carlo pour calculer le résultat d'un certain nombre de parties aléatoires à la suite d'un mouvement. Ainsi, on pourrait disposer d'une heuristique statistique qui nous indiquerait la bonne direction s'il y en a une !

**Solution :** Voir le fichier joint !