PC*

Lycée Marcelin Berthelot

Informatique

Chapitre I
Théorie des graphes

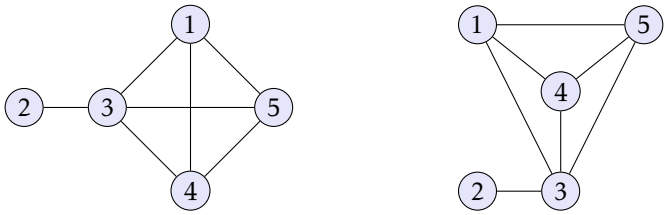
1. Vocabulaire et représentation

De manière générale, un graphe permet de représenter les connexions d'un ensemble en exprimant les relations entre ses éléments : réseau de communication, réseau routier, circuit électronique, ... , mais aussi relations sociales ou interactions entre espèces animales.
Le vocabulaire de la théorie des graphes est utilisé dans de nombreux domaines : chimie, biologie, sciences sociales, etc., mais c'est avant tout une branche à part entière et déjà ancienne des mathématiques (le fameux problème des ponts de Königsberg d'Euler date de 1736). Néanmoins, l'importance accrue que revêt l'aspect algorithmique dans ses applications pratiques en fait aussi un domaine incontournable de l'informatique. Pour schématiser, les mathématiciens s'intéressent avant tout aux propriétés globales des graphes (graphes eulériens, graphes hamiltoniens, dénombrement, ...) là où les informaticiens vont plutôt chercher à concevoir des algorithmes efficaces pour résoudre un problème faisant intervenir un graphe (recherche du plus court chemin, problème du voyageur de commerce, ...). Tout ceci forme un ensemble très vaste, dont nous n'aborderons que quelques aspects, essentiellement de nature algorithmique.

1.1 Graphes non orientés

Un graphe $G = (V, E)$ est défini par l'ensemble fini $V = \{v_1, v_2, \dots, v_n\}$ dont les éléments sont appelés *sommets* (*vertices* en anglais), et par l'ensemble fini $E = \{e_1, e_2, \dots, e_m\}$ dont les éléments sont appelés *arêtes* (*edges* en anglais).
Une arête e de l'ensemble E est définie par une paire non ordonnée de sommets, appelés les *extrémités* de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont *adjacents*.
Enfin, on appelle *ordre* d'un graphe le nombre de sommets n de ce graphe. Le *degré* $d(v)$ d'un sommet v est le nombre de sommets qui lui sont adjacents, et le *degré* d'un graphe le degré maximum de tous ses sommets.

Les graphes tirent leur nom du fait qu'on peut les représenter graphiquement : à chaque sommet de G on fait correspondre un point du plan et on relie les points correspondant aux extrémités de chaque arête. Il existe donc une infinité de représentations possibles. Par exemple, les deux dessins qui suivent représentent le même graphe G, avec $V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)\}$:



G est un graphe d'ordre 5 ; il possède un sommet de degré 1 (le sommet 2), trois sommets de degré 3 (les sommets 1, 4 et 5) et un sommet de degré 4 (le sommet 3).

Remarque. On peut observer que ce graphe a la particularité de posséder une représentation (celle de droite) pour laquelle les arêtes ne se coupent pas. Un tel graphe est dit *planaire*. Cette notion ne sera pas abordée dans la suite de ce cours.

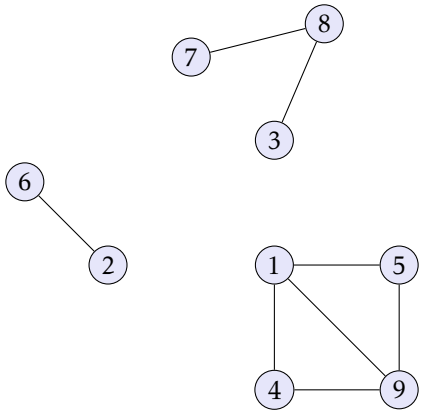
Un graphe est dit *simple* lorsqu'aucun sommet n'est adjacent à lui-même. Par la suite, nous nous restreindrons à l'étude des graphes simples.

Théorème 1.1 — *Si G est un graphe non orienté simple, La somme des degrés de ses sommets est égal à deux fois le nombre de ses arêtes :*

$$\sum_{v \in V} \deg(v) = 2|E|.$$

■ Chemins

Un *chemin* de longueur k reliant les sommets a et b est une suite finie $x_0 = a, x_1, \dots, x_k = b$ de sommets tel que pour tout $i \in [\![0, k-1]\!]$, la paire (x_i, x_{i+1}) soit une arête de G. Ce chemin est dit *cyclique* lorsque $a = b$.
La *distance* entre deux sommets a et b est la plus petite des longueurs des chemins reliant a et b, si tant est qu'il en existe. Lorsque tous les sommets sont à distance finie les uns des autres, on dit que le graphe est *connexe*. Un graphe non connexe peut être décomposé en plusieurs *composantes connexes*, qui sont des sous-graphes connexes maximaux.
À titre d'exemple, le graphe suivant possède trois composantes connexes :



Démontrons maintenant deux résultats généraux relatifs à la connexité et la cyclicité :

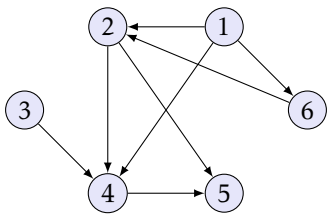
Théorème 1.2 — *Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.*

Lemme — *Si dans un graphe* G *tout sommet est de degré supérieur ou égal à 2, alors* G *possède au moins un cycle.*

Théorème 1.3 — *Un graphe acyclique d'ordre n comporte au plus $n-1$ arêtes.*

1.2 Graphes orientés

On obtient un graphe *orienté* en distinguant la paire de sommets (a, b) de la paire (b, a). Dans ce cas, on définit pour chaque sommet v son degré *sortant* $\mathrm{d}_+(v)$, égal au nombre d'arcs (dans le cas d'un graphe orienté, on parle souvent d'arc plutôt que d'arêtes) dont il est la première composante, et son degré *entrant* $\mathrm{d}_-(v)$, égal au nombre d'arcs dont il est la seconde composante. Par exemple, le graphe qui suit est défini par $\mathrm{V} = \{1, 2, 3, 4, 5, 6\}$ et $\mathrm{E} = \{(1,2), (1,4), (1,6), (2,4), (2,5), (3,4), (4,5), (6,2)\}$:



Le sommet 1 a un degré sortant égal à 3 et un degré entrant égal à 0, tandis que le sommet 2 a un degré entrant et un degré sortant tous deux égaux à 2.

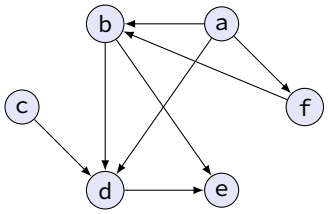
Les notions de chemin et de distance s'étendent sans difficulté aucune au cas des graphes orientés.

1.3 Mise en œuvre pratique

Deux méthodes principales s'offrent à nous pour représenter un graphe en machine :
- à l'aide de *listes d'adjacence* (à chaque sommet on associe la liste de ses voisins) ;
- à l'aide de *matrice d'adjacence* (le coefficient d'indice (i, j) traduit l'existence ou non d'une liaison entre deux sommets).

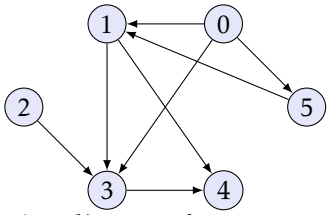
■ Listes d'adjacence

Cette représentation est économique en mémoire, en particulier pour des graphes ayant peu d'arcs ou arêtes. Elle consiste à définir un dictionnaire dont les clefs sont les sommets et les valeurs les listes des successeurs de ces sommets.




```
G = {'a': ['b', 'd', 'f'],
     'b': ['d', 'e'],
     'c': ['d'],
     'd': ['e'],
     'e': [],
     'f': ['b']}
```

Si on numérote les sommets de 0 à $n-1$ on peut même se contenter d'une liste de listes.




```
G = [[1, 3, 5], [3, 4], [3], [4], [], [1]]
```

Désorientation d'un graphe

Un inconvénient potentiel de la représentation par listes d'adjacence est qu'il est difficile de déterminer rapidement si un graphe est non orienté : il faut pour chacune des arêtes vérifier que l'arête réciproque est aussi présente.

Exercice 1

Rédiger une fonction `estOriente(G)` qui prend pour argument un graphe représenté par listes d'adjacence et renvoie `True` s'il s'agit d'un graphe orienté, et `False` sinon.

Désorienter un graphe consiste à calculer le plus petit graphe non orienté G′ contenant G. Pour toute arête reliant les sommets a et b il faut donc ajouter l'arête reliant b à a, si celle-ci ne figure pas déjà dans le graphe.

Exercice 2

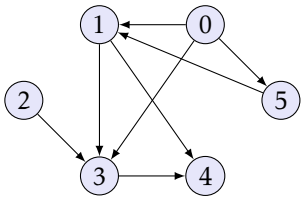
Écrire une fonction `desoriente(G)` qui prend pour argument un graphe orienté G représenté par liste d'adjacence et qui renvoie en nouveau graphe représentant la désorientation de G.

■ Matrice d'adjacence

Lorsque le nombre d'arcs ou d'arêtes est important, on ordonne les sommets $V = \{v_1, v_2, \dots, v_n\}$ afin de représenterle graphe G par une matrice $M \in \mathcal{M}_n(\{0,1\})$:

$$m_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in E \\ 0 & \text{sinon} \end{cases}$$

autrement dit, d'un point de vue informatique, par un tableau bi-dimensionnel. En outre, compte tenu de l'indexation des tableaux en Python il peut être judicieux de numéroter les sommets de 0 à $n-1$:




```
G = [[0, 1, 0, 1, 0, 1],
     [0, 0, 0, 1, 1, 0],
     [0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 0, 0],
     [0, 1, 0, 0, 0, 0]]
```

Déterminer si un graphe est orienté est chose aisée (il suffit de vérifier que la matrice est symétrique), de même de la désorientation d'un graphe, mais elle présente l'inconvénient d'occuper beaucoup plus d'espace mémoire, en particulier lorsque le nombre d'arêtes est réduit vis-à-vis du nombre de sommets : si $n = |V|$ et $p = |E|$, le coût spatial de la représentation par matrice d'adjacence est un $O(n^2)$, contre un $O(n+p)$ pour une liste d'adjacence.

Exercice 3

On considère un graphe non orienté dont les sommets sont notés $0, 1, \ldots, n-1$.

a. Rédiger une fonction `mat2lst(G)` qui prend en argument un graphe représenté par matrice d'adjacence et renvoie sa représentation par listes d'adjacence.

b. Rédiger une fonction `lst2mat(G)` qui prend en argument un graphe représenté par listes d'adjacence et renvoie sa représentation par matrice d'adjacence.

2. Parcours d'un graphe

Parcourir un graphe, c'est énumérer l'ensemble des sommets accessibles par un chemin à partir d'un sommet donné, dans l'objectif de leur faire subir un certain traitement. Différentes solutions sont possibles mais en règle générale celles-ci tiennent à jour deux listes : la liste des sommets rencontrés (« déjàVu ») et la liste des sommets en cours de traitement (« àTraiter ») et ces méthodes vont différer par la façon dont sont insérés puis retirés les sommets dans cette structure de données.

Le parcours générique à partir d'un sommet source s_0 se déroule de la façon suivante :

```
procedure PARCOURS(sommet : s_0)
    àTraiter ← s_0
    déjàVu ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        for t ∈ voisins(s) do
            if t ∉ déjàVu then
                àTraiter ← t
                déjàVu ← t
```

S'il y a lieu, le traitement associé à un sommet s peut être placé en différents endroits, en général au moment où le sommet s entre dans « àTraiter » ou au moment où il en sort.

Coût du parcours

Dans ce qui suit, nous considérerons un graphe défini par listes d'adjacence, avec $V = [\![0, n-1]\!]$.

Lors d'un parcours, chaque sommet entre au plus une fois dans la liste des sommets à traiter, et n'en sort donc aussi qu'au plus une fois. Si ces opérations d'entrée et de sortie dans la liste sont de coût constant, le coût total des manipulations de la liste « àTraiter » est un $O(n)$.
Chaque liste d'adjacence est parcourue au plus une fois donc le temps total consacré à scruter les listes de voisinage est un $O(p)$ où $p = |E|$ est le nombre d'arêtes/arcs, *à condition de déterminer si un sommet a déjà été vu en coût constant*. Dans ce cas, le coût total d'un parcours est un $O(n+p)$. Pour réaliser cette condition, la solution que nous adopterons consistera à numéroter les sommets de 0 à $n-1$ et utiliser un tableau booléen pour représenter « déjàVu », destiné à marquer chaque sommet au moment où il entre dans la liste « àTraiter ».

2.1 Parcours en largeur et en profondeur

Nous allons maintenant nous intéresser à deux types de parcours, qui diffèrent par la façon d'extraire les sommets de la liste « àTraiter » : les parcours en largeur et en profondeur.
Comme on peut le constater figure 1, seule la ligne 7 diffère : pour sortir de la liste « àTraiter », le parcours en largeur suit le principe d'une *file* : « premier entré, premier sorti » et le parcours en profondeur le principe d'une *pile* : « premier entré, dernier sorti ». L'ordre du traitement des sommets ne va pas être le même, et chaque parcours aura donc un usage propre.

■ Parcours en largeur

Il consiste à traiter tous les sommets à une distance égale à 1 du sommet initial, puis à une distance égale à 2, à 3, etc. Ce type de parcours est donc idéal pour trouver la plus courte distance entre deux sommets du graphe : il suffit d'ajouter au parcours en largeur un tableau `dist` destiné à enregistrer la distance à la source s_0. Lorsqu'on découvre un nouveau sommet t en examinant les voisins du sommet s, `dist[t]` prend la valeur `dist[s] + 1`.

```
1  def parcoursLargeur(G, s0):
2      n = len(G)
3      dejaVu = [False for _ in range(n)]
4      aTraiter = [s0]
5      dejaVu[s0] = True
6      while len(aTraiter) > 0:
7          s = aTraiter.pop(0)
8          for t in G[s]:
9              if not dejaVu[t]:
10                 aTraiter.append(t)
11                 dejaVu[t] = True
```

```
1  def pseudoParcoursProfondeur(G, s0):
2      n = len(G)
3      dejaVu = [False for _ in range(n)]
4      aTraiter = [s0]
5      dejaVu[s0] = True
6      while len(aTraiter) > 0:
7          s = aTraiter.pop()
8          for t in G[s]:
9              if not dejaVu[t]:
10                 aTraiter.append(t)
11                 dejaVu[t] = True
```

Figure 1 – Parcours en largeur et en pseudo-parcours en profondeur.

```
def distance(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    dist = [None for _ in range(n)]
    aTraiter = [s0]
    dejaVu[s0] = True
    dist[s0] = 0
    while len(aTraiter) > 0:
        s = aTraiter.pop(0)
        for t in G[s]:
            if not dejaVu[t]:
                dist[t] = dist[s] + 1
                aTraiter.append(t)
                dejaVu[t] = True
    return dist
```

```
from collections import deque

def distance(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    dist = [None for _ in range(n)]
    aTraiter = deque([s0])
    dejaVu[s0] = True
    dist[s0] = 0
    while len(aTraiter) > 0:
        s = aTraiter.popleft()
        for t in G[s]:
            if not dejaVu[t]:
                dist[t] = dist[s] + 1
                aTraiter.append(t)
                dejaVu[t] = True
    return dist
```

Figure 2 – Calcul des distances à un sommet à l'aide d'un parcours en largeur.

Remarque. Nous l'avons dit plus haut, pour réaliser une complexité en $O(n+p)$ il est nécessaire que la méthode permettant de sortir un élément de la liste `aTraiter` soit de complexité constante, ce qui n'est pas le cas de la méthode `.pop(0)`. Pour pallier à ce problème, on peut utiliser la structure `deque` (*double ended queue*) du module `collections`. Cette structure fonctionne peu ou prou comme une liste, si ce n'est qu'elle permet en outre l'insertion et la suppression en temps constant à gauche à l'aide des méthodes `.appendleft()` et `.popleft()`. Comme on peut le constater figure 2, les modifications sont minimes.

■ Parcours en profondeur

Au contraire d'un parcours en largeur, le parcours en profondeur consiste à s'enfoncer le plus possible dans un graphe avant de remontrer vers les sommets déjà rencontrés. La version présentée figure 1 n'est pas un vrai parcours en profondeur car tous les voisins d'un même sommet entrent dans `aTraiter` au même moment. Cette version ne sera donc utilisée que lorsque l'ordre de traitement des sommets n'a pas d'importance, par exemple pour détecter la connexité d'un graphe ou calculer ses composantes connexes (voir figure 3).

En revanche, si un traitement nécessite que l'ordre du parcours en profondeur soit impérativement suivi, la version présentée figure 1 n'est pas suffisante. Dans ce cas de figure, le plus simple est d'adopter une version récursive du parcours en profondeur, sur le modèle présenté figure 4 : la fonction récursive est définie lignes 5-9 et appelée sur le sommet s_0 ligne 11.

```python
def estConnexe(G):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    aTraiter = [0]
    dejaVu[0] = True
    vus = 1
    while len(aTraiter) > 0:
        s = aTraiter.pop()
        for t in G[s]:
            if not dejaVu[t]:
                aTraiter.append(t)
                dejaVu[t] = True
                vus += 1
    return vus == n
```

```python
def composantesConnexes(G):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    composantes = []
    for i in range(n):
        if not dejaVu[i]:
            aTraiter = [i]
            dejaVu[i] = True
            comp = [i]
            while len(aTraiter) > 0:
                s = aTraiter.pop()
                for t in G[s]:
                    if not dejaVu[t]:
                        aTraiter.append(t)
                        dejaVu[t] = True
                        comp.append(t)
            composantes.append(comp)
    return composantes
```

FIGURE 3 – Test de connexité d'un graphe et calcul des composantes connexes.

```python
def parcoursProfondeur(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]

    def parcours(s):
        if not dejaVu[s]:
            for t in G[s]:
                dejaVu[t] = True
                parcours(t)

    parcours(s0)
```

FIGURE 4 – Un parcours en profondeur récursif.

Détection de cycle dans un graphe non orienté
L'exemple typique pour lequel l'ordre dans lequel les sommets sont découverts est important est la détection d'un cycle dans un graphe. En effet, dans un « vrai » parcours en profondeur, si un sommet tout juste découvert a pour voisin un sommet en cours de traitement, c'est qu'il y a un cycle. Dans la version récursive, les sommets en cours de traitement sont situés dans la pile de récursivité, qui n'est pas accessible. Aussi on remplace la liste booléenne `dejaVu` par une liste pouvant prendre trois valeurs : `'blanc'` pour les sommets non encore découverts, `'gris'` pour les sommets en cours de traitement, et `'noir'` pour les sommets dont le traitement est fini. On obtient la version présentée figure 5 : la fonction récursive définie lignes 5-14 renvoie `True` en présence d'un cycle et `False` dans le cas contraire. Les lignes 16 à 20 se contentent de tester la présence d'un cycle partant de chaque sommet non encore rencontré du graphe.

Remarque. Cette version de l'algorithme de détection d'un cycle ne s'applique pas au cas d'un graphe non orienté car dès lors que deux sommets u et v sont voisins, le chemin $u \to v \to u$ est détecté comme un cycle par cette version.

3. Graphe pondéré et plus court chemin

Déterminer le plus court chemin entre deux sommets d'un graphe non pondéré n'est pas difficile : il suffit d'effectuer un parcours en largeur à partir d'un des deux sommets jusqu'à trouver l'autre.

Cependant, connaître le nombre minimal d'arêtes à parcourir entre deux sommets n'est pas toujours suffisant :

```python
def cycle(G):
    n = len(G)
    dejaVu = ['blanc' for _ in range(n)]

    def parcours(s):
        dejaVu[s] = 'gris'
        for t in G[s]:
            if dejaVu[t] == 'blanc':
                if parcours(t):
                    return True
            elif dejaVu[t] == 'gris':
                    return True
        dejaVu[s] = 'noir'
        return False

    for i in range(n):
        if dejaVu[i] == 'blanc':
            if parcours(i):
                return True
    return False
```

FIGURE 5 – Détection de cycle à l'aide d'un parcours en profondeur.

de nombreux problèmes ajoutent une *pondération* à chaque arête et définissent le *poids* d'un chemin comme la somme des poids des arêtes qui le composent. Commençons par donner quelques définitions.

3.1 Graphe pondéré

Étant donné un graphe (orienté ou non) $G = (V, E)$, on appelle *pondération* une application $w : E \to \mathbb{R}$, et pour tout $(a, b) \in E$ on dit que $w(a, b)$ est le *poids* de l'arête (a, b).
Il sera par la suite commode de prolonger la définition de w sur $V \times V$ en posant :

$$\forall (a,b) \in (V \times V) \setminus E, \quad w(a,b) = \begin{cases} 0 & \text{si } a = b \\ +\infty & \text{sinon} \end{cases}$$

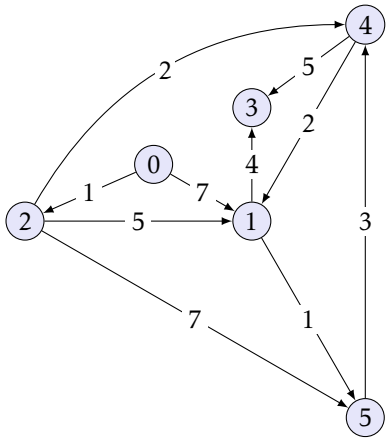
Le *poids* d'un chemin est la somme des poids des arêtes qui le composent. On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b, s'il en existe. Dans le cas contraire, on posera $\delta(a, b) = +\infty$.

Remarque. En présence de poids négatifs il convient de prendre quelques précautions pour assurer l'existence de ce minimum. En particulier, s'il existe un chemin menant de a et b et comprenant un circuit fermé de poids strictement négatif, il convient de poser $\delta(a, b) = -\infty$ car dans ce cas on peut faire indéfiniment décroitre le poids de ce chemin en multipliant les passages par cette boucle. Par la suite, nous éviterons cette situation en supposant *que tous les poids sont positifs ou nuls*.

■ Représentation en mémoire

Dans cette section on représentera un graphe pondéré en adaptant la représentation par matrice d'adjacence : celle-ci sera maintenant une matrice à valeurs dans $\mathbb{R}_+ \cup \{+\infty\}$ contenant les poids des différentes arêtes (illustration figure 6).

Remarque. Rappelons que Python offre la possibilité de définir un flottant représentant $+\infty$ en écrivant `float('inf')`.




```
inf = float('inf')

G = [[0, 7, 1, inf, inf, inf],
     [inf, 0, inf, 4, inf, 1],
     [inf, 5, 0, inf, 2, 7],
     [inf, inf, inf, 0, inf, inf],
     [inf, 2, inf, 5, 0, inf],
     [inf, inf, inf, inf, 3, 0]]
```

FIGURE 6 – Un exemple de représentation d'un graphe pondéré.

3.2 L'algorithme de DIJKSTRA

Cet algorithme résout le problème des plus courts chemins à partir d'une source $s_0 \in V$: il prend pour arguments un graphe pondéré G et un sommet s_0 et renvoie un tableau contenant l'ensemble des distances minimales reliant s_0 à un sommet quelconque de G. Cet algorithme nécessite de supposer que tous les poids sont positifs ou nuls.

■ Principe de l'algorithme

L'algorithme de Dijkstra remplit progressivement un tableau d de longueur n de sorte qu'à la fin de cet algorithme $d[v]$ soit égal à $\delta(s_0, v)$, le poids du plus court chemin entre s_0 et v. Pour ce faire, on fait évoluer une partition de $[\![1, n]\!]$: un sous-ensemble S (initialement vide) et son complémentaire $\overline{S}$.
L'ensemble S est destiné à représenter les sommets dont on a déterminé dans le tableau d le poids du chemin minimal à partir de s_0 (donc dont le traitement est terminé). Pour les éléments de $\overline{S}$, le tableau d contient le poids du plus court chemin *ne passant que par des sommets appartenant à* S. Ainsi, le tableau d est initialement défini par :

$$\forall v \in V, \quad d[v] = \begin{cases} 0 & \text{si } v = s_0 \\ +\infty & \text{sinon} \end{cases}$$

À chaque itération on choisit le sommet s de $\overline{S}$ dont la valeur associée dans le tableau d est minimale pour le transférer dans S, et on met à jour le tableau d en remplaçant $d[v]$ par $\min(d[v], d[s] + w(s, v))$. Ainsi, chaque élément de $\overline{S}$ va progressivement être transféré dans S.

On représente classiquement la partition $S \cup \overline{S}$ par un tableau de type `dejaVu`, de sorte que l'algorithme de Dijkstra s'écrit :

```
1  def dijkstra(G, s0):
2      n = len(G)
3      dejaVu = [False for _ in range(n)]
4      d = [float('inf') for _ in range(n)]
5      d[s0] = 0
6      for _ in range(n):
7          mini = float('inf')
8          for v in range(n):
9              if not dejaVu[v] and d[v] <= mini:
10                 s, mini = v, d[v]
11         dejaVu[s] = True
12         for v in range(n):
13             if not dejaVu[v]:
14                 d[v] = min(d[v], d[s] + G[s][v])
15     return d
```

Les lignes 7-10 ont pour objet de déterminer parmi les sommets non traités celui dont la valeur associée dans le tableau d est minimale. Les lignes 11-14 mettent à jour le tableau d en conséquence.
Cette démarche est répétée n fois de sorte que tous les sommets soient traités.

Observons comment se déroule cet algorithme sur le graphe représenté figure 6, à partir du sommet 0, en observant l'évolution du tableau d au cours de l'algorithme :

Initialisation

	0	1	2	3	4	5
déjàVu	·	·	·	·	·	·
d	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

étape 1

	0	1	2	3	4	5
déjàVu	×	·	·	·	·	·
d	0	7	1	$+\infty$	$+\infty$	$+\infty$

Le sommet 0 est traité, l'algorithme a découvert deux chemins vers les sommets 1 et 2.

étape 2

	0	1	2	3	4	5
déjàVu	×	·	×	·	·	·
d	0	6	1	$+\infty$	3	8

Le sommet 2 est traité, l'algorithme a découvert deux chemins vers les sommets 4 et 5 et un chemin plus court vers le sommet 1.

étape 3

	0	1	2	3	4	5
déjàVu	×	·	×	·	×	·
d	0	5	1	8	3	8

Le sommet 4 est traité, l'algorithme a découvert un chemin vers le sommet 3 et un chemin plus court vers le sommet 1.

étape 4

	0	1	2	3	4	5
déjàVu	×	×	×	·	×	·
d	0	5	1	8	3	6

Le sommet 1 est traité, l'algorithme a découvert un chemin plus court vers le sommet 5.

étape 5

	0	1	2	3	4	5
déjàVu	×	×	×	·	×	×
d	0	5	1	8	3	6

Le sommet 5 est traité, aucun nouveau chemin n'est découvert.

étape 5

	0	1	2	3	4	5
déjàVu	×	×	×	×	×	×
d	0	5	1	8	3	6

Le sommet 3 est traité, l'algorithme est terminé.

■ Complexité de l'algorithme de Dijkstra

Tel que nous l'avons écrit, l'algorithme de Dijkstra a une complexité quadratique en $O(n^2)$. Il est possible de mieux faire en utilisant des structures de données mieux adaptées pour représenter l'ensemble S, mais ces considérations sont hors-programme.

■ Détermination des chemins de poids minimaux

Pour garder trace du chemin parcouru, il suffit d'utiliser un tableau c que l'on modifie en même temps que d : lorsque $d[v]$ est remplacé par $d[v]+w(s,v)$, on mémorise s dans $c[v]$. Ainsi, à la fin de l'algorithme $c[v]$ contient le sommet précédant v dans un chemin minimal allant de s_0 à v, ce qui permet de reconstituer le chemin optimal une fois l'algorithme terminé.

Pour obtenir le chemin minimal on modifie donc l'algorithme de Dijkstra pour qu'il crée puis renvoie le tableau c, et on utilise cette version modifiée pour obtenir le plus court chemin entre deux sommets a et b :

```
def predecesseurs(G, s0):
    n = len(G)
    dejaVu = [False for _ in range(n)]
    d = [float('inf') for _ in range(n)]
    c = [None for _ in range(n)]
    d[s0] = 0
    for _ in range(n):
        mini = float('inf')
        for v in range(n):
            if not dejaVu[v] and d[v] <= mini:
                s, mini = v, d[v]
        dejaVu[s] = True
        for v in range(n):
            if not dejaVu[v] and d[s] + G[s][v] < d[v]:
                d[v] = d[s] + G[s][v]
                c[v] = s
    return c
```

```
def plusCourtChemin(G, a, b):
    pred = predecesseurs(G, a)
    if pred[b] is not None:
        chemin = [b]
        while True:
            chemin.append(pred[chemin[-1]])
            if chemin[-1] == a:
                return list(reversed(chemin))
```

3.3 L'algorithme A*

Dans la pratique, il est rare que tous les chemins de poids minimaux issus de s_0 nous intéressent ; en général seuls la distance et le chemin minimal entre une source a et un but b nous intéressent. On peut bien sûr utiliser l'algorithme de Dijkstra à partir de a et l'arrêter dès que le chemin minimal reliant a et b est trouvé, mais cette optimisation ne permet guère d'accélérer le processus lorsque l'objectif b fait partie des derniers points traités.

De manière absolue, on ne peut guère faire mieux que l'algorithme de Dijkstra, mais dans certains cas il est possible d'éliminer bon nombre de calculs inutiles en utilisant une *heuristique* ; c'est ce que fait l'algorithme A* (prononcer « A star »).

Cet algorithme suppose connue une fonction h (l'heuristique) capable d'évaluer grossièrement le poids du plus court chemin entre tout sommet s et le but b. L'algorithme fonctionne alors comme celui de Dijkstra, si ce n'est qu'au lieu de sélectionner à chaque étape le sommet s minimisant la quantité $d[s]$, on sélectionne celui minimisant la quantité $h(s)+d[s]$.

Nous admettrons que si la fonction h vérifie : $\forall s \in V,\ 0 \leqslant h(s) \leqslant \delta(s,b)$ alors l'algorithme A renvoie la distance minimale entre a et b.*

Remarque. La difficulté consiste à déterminer une fonction h vérifiant cette condition. Dans le cas d'un graphe dont les poids sont générés aléatoirement, cette condition parait bien difficile à satisfaire. En revanche, *sur un graphe constitué de points du plan et dont les arêtes sont pondérées par leurs longueurs respectives*, la distance euclidienne constitue une heuristique admissible, qui plus est facile à calculer. C'est exactement la situation d'un GPS d'une voiture vous menant d'un point a à un point b.

Chapitre II
Programmation dynamique

1. Dictionnaires

Cette première partie a pour objet l'étude d'une structure de données que vous avez déjà rencontré en première année : les dictionnaires. Cette structure de données répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps).

Les dictionnaires sont couramment utilisées en informatique : c'est par exemple la structure de données utilisée pour gérer les systèmes de noms de domaine (DNS, pour *Domain Name System*). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique (en IPv4 par exemple, celles-ci sont représentées sous la forme xxx.xxx.xxx.xxx, où xxx est un nombre hexadécimal variant entre 0 et 255). Pour faciliter l'accès aux systèmes qui disposent de ces adresses, un mécanisme a été mis en place pour associer un nom (plus facile à retenir) à une adresse IP. Ce mécanisme utilise un dictionnaire dans laquelle les clefs sont les noms de domaine et les valeurs les adresses IP.

1.1 Description

Un *dictionnaire* (ou mieux une *table d'association*) est un type de données associant un ensemble de clefs à un ensemble de valeurs. Plus formellement, si C désigne l'ensemble des clefs et V l'ensemble des valeurs, un dictionnaire est un sous-ensemble T de $C \times V$ tel que pour toute clef $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$. Les éléments de T sont appelés des *associations*.

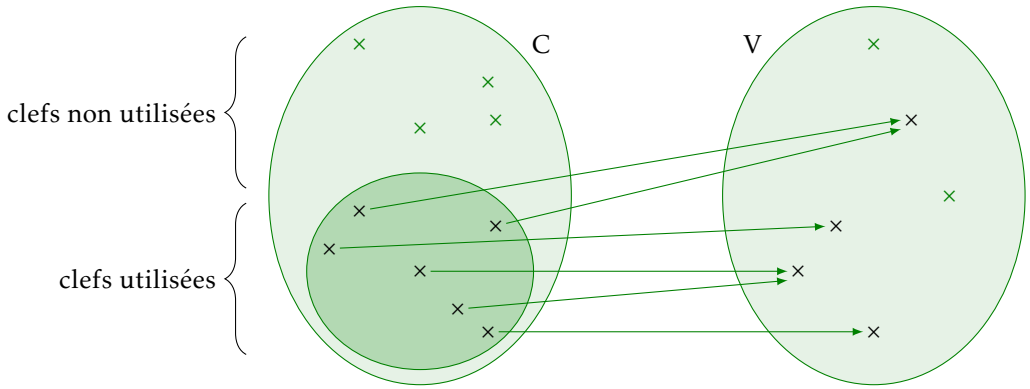


Figure 1 – Représentation informelle d'un dictionnaire.

Un dictionnaire supporte en général les opérations suivantes :

- *ajout* d'une nouvelle association $(c, v) \in C \times V$ dans T ;
- *suppression* d'une association (c, v) de T ;
- *existence* d'une association (c, v) dans T pour une clef $c \in C$ donnée ;
- *lecture* de la valeur v associée à une clef c présente dans T.

En Python, la création d'un dictionnaire se réalise en suivant la syntaxe `{c1: v1, ... , cn: vn}` où $c_1, \ldots, c_n$ sont des clefs (nécessairement deux-à-deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associées. Ainsi, `{ }` crée un dictionnaire vide.

Si `D` est un dictionnaire et `c` une clef,

- `D[c] = v` ajoute une nouvelle association si la clef n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon ;

- `del D[c]` supprime une association si la clef est présente dans le dictionnaire, et déclenche l'exception `KeyError` sinon ;
- l'expression `c in D` renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;
- `D[c]` renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire, et déclenche l'exception `KeyError` sinon.

Notons que, pour des raisons qui apparaîtrons plus loin, les clefs d'un même dictionnaire doivent impérativement appartenir à un type immuable.

■ Parcours d'un dictionnaire

Dans certaines situations, il peut être utile de parcourir l'ensemble des clefs d'un dictionnaire, l'ensemble des valeurs d'un dictionnaire, voire les deux. C'est pourquoi trois méthodes sont associées aux objets de ce type :

- `D.keys` renvoie la liste [1] des clefs d'un dictionnaire `D` ;
- `D.values` renvoie la liste des valeurs présentes dans le dictionnaire `D` ;
- `D.items` renvoie la liste des couples (clefs, valeurs) du dictionnaire `D`.

Ainsi, pour parcourir un dictionnaire on utilisera suivant les besoins l'une de ces trois syntaxes :

```
for k in D.keys:    for v in D.values:    for (k, v) in D.items:
```

Notons que l'itération sur les clefs `for k in D.keys:` peut même s'écrire plus simplement `for k in D:`

1.2 Mise en œuvre pratique d'un dictionnaire

Il est communément admis (même si ce n'est pas tout-à-fait correct) que les quatre opérations de base d'un dictionnaire se réalisent avec une complexité temporelle constante ; pour comprendre comment une telle performance est possible, nous allons décrire quelques méthodes d'implémentation de cette structure.

■ Deux solutions inenvisageables en pratique

Si les clefs étaient des entiers compris entre 0 et $m-1$, le problème serait simple : il suffirait d'utiliser un tableau de taille m. Comme ce n'est en général pas le cas, on se ramène à cette situation en utilisant une fonction $f : \mathrm{C} \to [\![0, m-1]\!]$ associant aux différentes clefs $c \in \mathrm{C}$ possibles un entier dans $[\![0, m-1]\!]$.
Mais un autre problème, bien plus grave, se pose : l'ensemble C est en général de cardinal extrêmement grand : ce peut-être par exemple l'ensemble des objets de type `int`, ou `float`, donc un cardinal de l'ordre de 2^{64}, et l'espace mémoire dévolu à un dictionnaire serait bien plus grand que nos moyens de stockages actuels (il faudrait plus de 100 milliards de tera-octets pour stocker un tel tableau…)

Une autre solution consisterait à stocker un dictionnaire dans une liste dynamique, initialement vide, que l'on remplirait progressivement avec des couples clef/valeur. Mais ici se pose un problème de performance, car les opérations usuelles d'un dictionnaire : lecture, ajout, suppression seraient de complexité linéaire, alors qu'on s'attend à une complexité constante (ou presque).

```python
def find_lst(L, clef):
    for (c, v) in L:
        if c == clef:
            return v
    raise KeyError(clef)
```

```python
def add_lst(L, clef, valeur):
    for k in range(len(L)):
        (c, v) = L[k]
        if c == clef:
            L[k] = (clef, valeur)
            return None
    L.append((clef, valeur))
```

Figure 2 – Les opérations de lecture et d'ajout dans une liste sont de complexité linéaire.

1. En toute rigueur ce n'est pas une liste mais vous pouvez faire comme si.

■ Une solution médiane

La solution consiste à adopter une solution qui mélange les deux approches précédentes : nous allons utiliser un tableau de taille m et une fonction $f : \mathrm{C} \to [\![0, m-1]\!]$ qui à toute clef associe un entier, mais avec une valeur m bien plus petite que le cardinal de C. Une telle fonction ne sera donc pas injective : il existera des couples (c_1, c_2) dans C tels que $c_1 \neq c_2$ et $f(c_1) = f(c_2)$. Un tel couple (c_1, c_2) est appelé une *collision*.
Ainsi, deux clefs qui entrent en collision vont être associées à la même case du tableau, et c'est pourquoi chaque case du tableau contiendra une liste dans laquelle les associations clefs/valeurs seront stockées suivant le principe décrit plus haut.
La figure 3 illustre cette solution avec $f(c_1) = f(c_4)$ et $f(c_3) = f(c_5) = f(c_6)$.

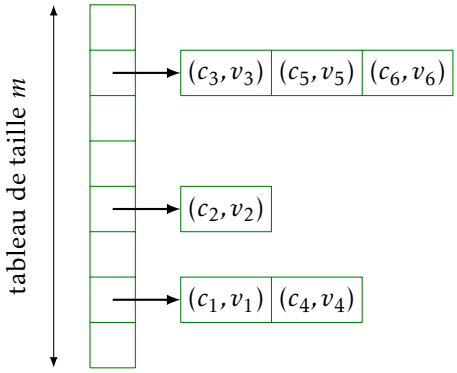


FIGURE 3 – Les clefs qui entrent en collision sont stockées dans le même paquet.

```
def find(D, clef):
    L = D[f(clef)]
    return find_lst(L, clef)
```

```
def add(D, clef, valeur):
    L = D[f(clef)]
    add_lst(L, clef, valeur)
```

FIGURE 4 – Les opérations de lecture et d'ajout dans un dictionnaire.

1.3 Un peu de probabilités

L'idéal consiste, on s'en doute, à remplir le dictionnaire en faisant en sorte que les m listes soient de tailles à peu près égales. Dans ce cas, le temps d'exécution par rapport à la solution naïve utilisant une seule liste serait divisé par m, ce qui constituerait une amélioration loin d'être négligeable.

Formalisons un peu ceci : on considère l'espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$ où $\Omega = \mathrm{C}$ est l'ensemble des clefs, $\mathcal{A} = \mathscr{P}(\Omega)$ et $\mathbb{P}$ la probabilité définie sur $\mathcal{A}$ par $\mathbb{P}(\{c\}) = \dfrac{1}{\operatorname{card} \mathrm{C}}$.
Faisons ensuite l'hypothèse (importante) que $\mathrm{F} : c \mapsto f(c)$ est une *variable aléatoire suivant une loi uniforme*, et intéressons-nous au problème de la recherche d'une clef dans un dictionnaire contenant n enregistrements.
Ces n enregistrements sont modélisés par une suite $(\mathrm{F}_1, \dots, \mathrm{F}_n)$ de variables aléatoires indépendantes et identiquement distribuées, de même loi que F, et la clef recherchée dans le dictionnaire est représentée par F_{n+1}.
On note $\mathrm{N} = \operatorname{card}\left\{k \in [\![1, n]\!] \;\middle|\; \mathrm{F}_k = \mathrm{F}_{n+1}\right\}$ le nombre de collisions entre F_{n+1} et les clefs déjà présentes dans le dictionnaire et X le nombre de comparaisons nécessaires pour trouver cette clef.

Cas où la clef recherchée ne se trouve pas dans le dictionnaire

On a $\mathrm{N} = \displaystyle\sum_{k=1}^{n} \mathbb{1}_{\mathrm{F}_{n+1}=\mathrm{F}_k}$ où $\mathbb{1}_{\mathrm{F}_{n+1}=\mathrm{F}_k} \sim \mathcal{B}(1/m)$ donc $\mathrm{N} \sim \mathcal{B}(n, 1/m)$.
Dans ce cas, $\mathrm{X} = \mathrm{N}$ (il faut parcourir toute la liste figurant dans la case de rang F_{n+1} pour constater que la clef ne figure pas dans le tableau) et donc $\mathbb{E}(\mathrm{X}) = \dfrac{n}{m}$.

Cas où la clef recherchée est présente dans le dictionnaire
Cette fois-ci $\mathrm{N}-1 \sim \mathcal{B}(n-1,1/m)$, et il est raisonnable de considérer que la clef peut se trouver de manière équiprobable à tous les emplacement de la liste correspondante, et donc que $(\mathrm{X} \mid \mathrm{N}=j) \sim \mathcal{U}(j)$.

Ainsi, $\mathbb{P}(\mathrm{X}=k) = \sum_{j=k}^{n} \mathbb{P}(\mathrm{X}=k \mid \mathrm{N}=j)\mathbb{P}(\mathrm{N}=j) = \sum_{j=k}^{n} \frac{1}{j}\binom{n-1}{j-1}\left(\frac{1}{m}\right)^{j-1}\left(1-\frac{1}{m}\right)^{n-j}$.

On calcule alors $\mathbb{E}(\mathrm{X}) = \sum_{k=1}^{n} k\mathbb{P}(\mathrm{X}=k)$ en inversant l'ordre de sommation :

$$\begin{aligned}\mathbb{E}(\mathrm{X}) &= \sum_{j=1}^{n}\sum_{k=1}^{j}\frac{k}{j}\binom{n-1}{j-1}\left(\frac{1}{m}\right)^{j-1}\left(1-\frac{1}{m}\right)^{n-j} = \frac{1}{2}\sum_{j=1}^{n}(j+1)\binom{n-1}{j-1}\left(\frac{1}{m}\right)^{j-1}\left(1-\frac{1}{m}\right)^{n-j}\\ &= \frac{1}{2}\sum_{j=1}^{n} j\binom{n-1}{j-1}\left(\frac{1}{m}\right)^{j-1}\left(1-\frac{1}{m}\right)^{n-j} + \frac{1}{2}\end{aligned}$$

On calcule cette somme en posant $f(x) = \sum_{j=1}^{n}\binom{n-1}{j-1}x^j\left(1-\frac{1}{m}\right)^{n-j}$, de sorte que $\mathbb{E}(\mathrm{X}) = \frac{1}{2}f'\left(\frac{1}{m}\right)+\frac{1}{2}$.

Or $f(x) = x\left(x+1-\frac{1}{m}\right)^{n-1}$ ce qui donne, tous calculs faits, $\mathbb{E}(\mathrm{X}) = \frac{n-1}{2m}+1$.

■ Complexité dans le pire des cas et complexité en moyenne

Nous venons de calculer ce qu'il convient d'appeler la *complexité en moyenne* de la recherche d'une clef dans un dictionnaire : le gain en temps est en moyenne amélioré d'un facteur $1/m$, et lorsque m est du même ordre de grandeur que n, les espérances calculées sont bornées et il est alors légitime de parler de *complexité constante en moyenne*.

Attention cependant à ne pas confondre cette complexité avec la complexité dans le pire des cas, qui correspond ici à la situation (très improbable) où toutes les clefs utilisées seraient stockées dans la même liste et où la recherche nécessiterait dans le pire des cas à n comparaisons. Mais à cet égard, l'inégalité de Bienaymé-Tchebichev peut nous rassurer : les listes seront courtes, sauf dans des cas extrêmement rares.

Remarque. La notion de complexité en moyenne n'étant pas au programme, on considérera par la suite que les complexités relatives aux fonctions de base d'un dictionnaire sont constantes lorsqu'il s'agira de faire l'analyse de la complexité temporelle d'un algorithme.

1.4 Fonction de hachage

Parlons maintenant un peu de la fonction f : pour obtenir de bonnes performances nous avons dit qu'il fallait que cette fonction réalise deux objectifs :

- être facile à calculer ;
- avoir une distribution la plus uniforme possible.

Pour définir cette fonction, on utilise une fonction h, appelée *fonction de hachage*, associant un entier à une clef de C, et on définit f en posant :

$$f(c) = h(c) \bmod m$$

Choix de la fonction de hachage
C'est un problème complexe que nous n'aborderons pas. Sachez seulement que Python propose une fonction de hachage : si `x` est un objet immuable (`int`, `str`, `float`, ...) alors `hash(x)` renvoie un entier répondant autant que faire ce peut aux exigences d'une fonction de hachage :

```
In [1]: hash(12345)
Out[1]: 12345

In [2]: hash("12345")
Out[2]: -3486454717630806608

In [3]: hash((1, 2, 3, 4, 5))  # un tuple est immuable
Out[3]: -5659871693760987716

In [4]: hash([1, 2, 3, 4, 5])  # une liste est mutable
TypeError: unhashable type: 'list'
```

Une fonction de hachage ne peut prendre en entrée une variable mutable, car elle doit être déterministe : le résultat d'un hachage doit être toujours le même sur la même entrée (or le contenu d'une variable mutable peut être modifié sans modifier la dite variable…)

Remarque. Une fonction de hachage a d'autres usages. Par exemple, lorsque vous choisissez un mot de passe sur un site sécurisé, ce dernier ne va pas stocker votre mot de passe en clair, mais va stocker le résultat de l'application d'une fonction de hachage h à votre mot de passe. Sachant qu'il est très difficile de trouver les antécédents d'un entier par la fonction h, pirater le site ne mettra pas en cause la sécurité de votre mot de passe. Une autre application des fonctions de hachage est le contrôle d'intégrité d'un fichier informatique : à chaque fichier est associé une valeur par une fonction de hachage (vous connaissez peut-être le MD5) qui permet de vérifier si un fichier téléchargé a été transmis correctement.

1.5 Résolution des collisions par adressage ouvert

Il existe une autre solution pour implémenter un dictionnaire que nous allons nous contenter d'évoquer brièvement : elle consiste, en cas de collision, à chercher un emplacement libre dans lequel déposer la nouvelle association. La recherche d'un emplacement libre porte le nom de *sondage*.

- le sondage *linéaire* consiste à partir de $i = f(c)$ et à chercher une place libre en testant successivement les cases d'indices $(i+1) \bmod m$, $(i+2) \bmod m$, $(i+3) \bmod m$,… jusqu'à trouver un emplacement libre.
- le sondage *quadratique* procède de même mais en sondant les cases d'indices $(i+1) \bmod m$, $(i+1+2) \bmod m$, $(i+1+2+3) \bmod m$,…

L'inconvénient d'un sondage linéaire est qu'il y a un risque de former des « agrégats », autrement dit de longues successions de cases contigües occupées, qui nuisent à la répartition uniforme recherchée.

0	1	2	3	4	5	6	7	8
(c_5, v_5)		(c_1, v_1)	(c_4, v_4)	(c_3, v_3)			(c_2, v_2)	

FIGURE 5 – Un exemple de sondage linéaire : $f(c_6) = 2$ mais l'association (c_6, v_6) sera stockée dans la case 5.

Remarque. D'autres solutions plus complexes existent, comme par exemple sonder les cases $i + h'(c) \bmod m$, $i + 2h'(c) \bmod m$, $i + 3h'(c) \bmod m$,… où h' est une autre fonction de hachage, mais aucune ne résout complètement le problème de la création d'agrégats.

Évidemment, un adressage ouvert exige que le nombre n de clefs soit inférieur à la taille de la table m. En outre, on imagine aisément que lorsque le rapport $\alpha = \dfrac{n}{m}$ se rapproche de 1, il devient de plus en plus difficile de trouver un emplacement vide : si $\alpha = 0{,}5$ un ajout nécessite en moyenne deux sondages, contre dix lorsque $\alpha = 0{,}9$. Aussi, lorsque α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.

1.6 Ensembles

Les dictionnaires permettent aussi de représenter des ensembles : il suffit de supprimer les valeurs associées aux clefs pour faire d'un dictionnaire un ensemble de clefs.

En Python, le type correspondant s'appelle `set`. Un ensemble peut être défini par l'énumération de ses éléments initiaux : `s = {2, 3, 5, 7, 11, 13}` définit un ensemble. En revanche, pour qu'il n'y ait pas de confusion avec le dictionnaire vide il faut, pour définir l'ensemble vide, écrire `set()`.

Exemple. La conversion d'une liste en ensemble est un moyen simple de supprimer les doublons. Dans le script suivant, je tire au hasard 1 000 nombres compris entre 1 et 1 000 en supprimant les doublons.

```
In [1]: from random import randint
In [2]: s = set(randint(1, 1000) for _ in range(1000))

In [3]: len(s)
Out[3]: 630
```

Avec suppression des doublons il n'en reste que 630.

Mise en garde : attention, le type `set` est hors programme, mais il est très simple de les représenter par un dictionnaire, en associant à chaque clef une valeur arbitraire (`None` par exemple). Le code ci-dessus peut ainsi être écrit de la façon suivante en restant dans le cadre du programme :

```
In [4]: s = {randint(1, 1000): None for _ in range(1000)}

In [5]: len(s)
Out[5]: 633
```

2. Programmation dynamique

2.1 Un problème posé par la programmation récursive

Nous allons nous intéresser au calcul du coefficient binomial $\binom{n}{p}$. Une solution consiste à utiliser la programmation récursive et la formule de Pascal, ce qui nous amène à écrire :

```
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n - 1, p - 1) + binom(n - 1, p)
```

Malheureusement, cette fonction s'avère très peu efficace, même pour de relativement faibles valeurs de n et p : à titre d'illustration, il faut 54 secondes à mon ordinateur pour calculer $\binom{30}{15}$.

La raison en est facile à comprendre : lorsqu'on observe par exemple l'arbre de calcul de $\binom{5}{2}$ on constate que des appels récursifs sont identiques et donc superflus (figure 6).
Nous pouvons par exemple constater que le calcul de $\binom{5}{2}$ nécessite de calculer trois fois $\binom{2}{1}$. Et l'expérience montre que le calcul de $\binom{30}{15}$ fait appel 40 116 600 fois (!) au calcul de $\binom{2}{1}$.

Complexité temporelle
Pour évaluer la complexité de cette fonction, on note $C(n,p)$ le nombre d'additions réalisées par cette fonction, on dispose des relations :

$$C(n,0) = C(n,p) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

On démontre alors par récurrence sur $n \in \mathbb{N}$ que pour tout $p \in [\![0,n]\!]$, $C(n,p) = \binom{n}{p} - 1$. Or la formule de Stirling permet d'établir l'équivalent : $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ par cette fonction est donc de complexité exponentielle.

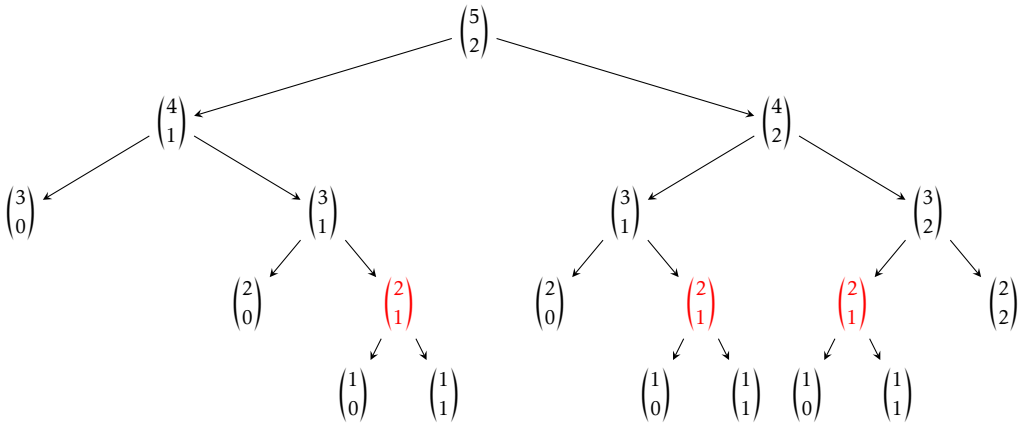


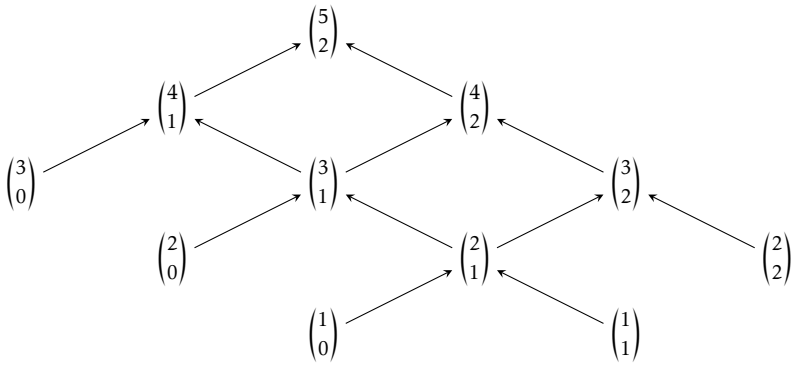
FIGURE 6 – Le calcul de $\binom{5}{2}$ fait appel trois fois au calcul de $\binom{2}{1}$.

Où est le problème ?

Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes : le calcul de $\binom{n-1}{p-1}$ et de $\binom{n-1}{p}$, *sous-problèmes qui sont en interaction*.
Par exemple, on constate sur la figure 6 que le calcul de $\binom{4}{1}$ et le calcul de $\binom{4}{2}$ font tous deux appel au même sous-problème : le calcul de $\binom{3}{1}$.
Ainsi, la présence de sous-problèmes en interaction peut faire croitre très rapidement la complexité d'une fonction, au point d'en rendre son usage rédhibitoire.

■ La solution proposée par la programmation dynamique

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands.
Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma :



Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n+1)\times(p+1)$ (dont seule la partie pour laquelle $i \geqslant j$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits (figure 7).
Il faut faire attention à bien respecter la relation de dépendance (modélisée par les flèches sur le schéma ci-dessus) pour remplir les cases de ce tableau : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases destinées à recevoir $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

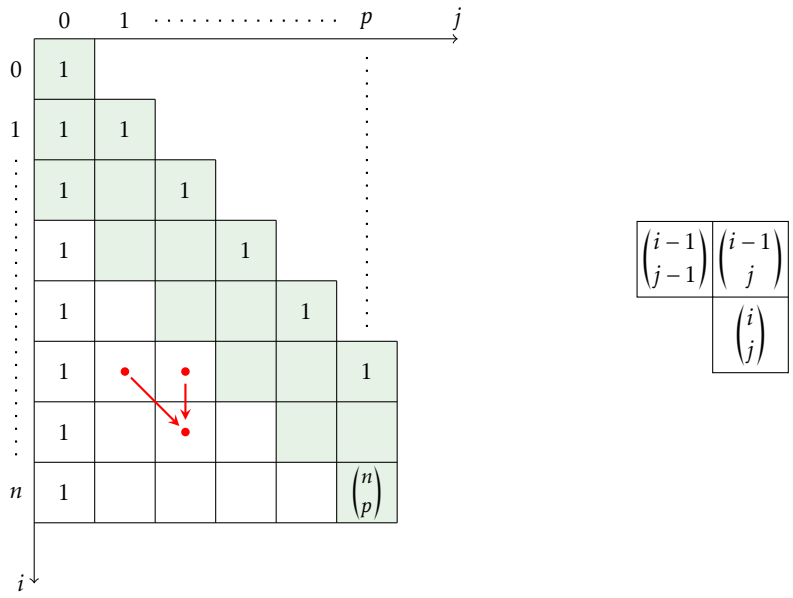


FIGURE 7 – Le schéma de dépendance du calcul de $\binom{n}{p}$.

```python
def binom(n, p):
    t = [[0 for j in range(p + 1)] for i in range(n + 1)]
    for i in range(0, n + 1):
        t[i][0] = 1
    for i in range(1, p + 1):
        t[i][i] = 1
    for i in range(2, n + 1):
        for j in range(1, min(p, i) + 1):
            t[i][j] = t[i - 1][j - 1] + t[i - 1][j]
    return t[n][p]
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est bien plus efficient que l'algorithme récursif initial puisque sa complexité temporelle et spatiale est maintenant en $O(np)$.

Remarque. Notons que cette solution n'est pas encore optimale : il est facile de constater sur le schéma de dépendance que l'algorithme ci-dessus remplit des cases inutiles pour le calcul de $\binom{n}{p}$: seules celles qui sont colorées sont nécessaires. On peut d'ailleurs observer qu'on peut se contenter d'utiliser un tableau unidimensionnel contenant les valeurs $\binom{i+j}{j}$ pour $j \in [\![0, p]\!]$ et de faire varier i entre 0 et $n-p$:

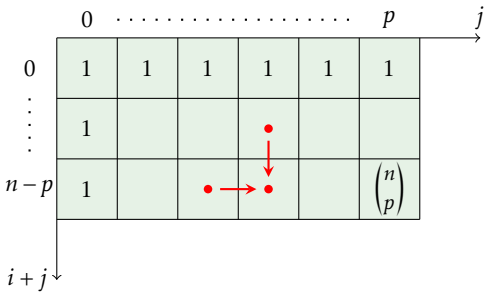


FIGURE 8 – Le schéma de dépendance du calcul de $\binom{i+j}{j}$.

```
def binom(n, p):
    t = [1 for j in range(p + 1)]
    for i in range(n - p):
       for j in range(1, p + 1):
            t[j] = t[j] + t[j - 1]
    return t[p]
```

La complexité est maintenant un $O(p(n-p))$.

■ Mémoïsation

Un inconvénient de la programmation dynamique réside dans la perte de lisibilité de l'algorithme, comparativement à l'algorithme récursif. L'idéal serait donc de combiner l'élégance de la programmation récursive avec l'efficacité de la programmation dynamique.

La solution existe, elle porte le nom de *mémoïsation*. Elle consiste à associer à la fonction un dictionnaire qui va mémoriser le résultat du calcul réalisé. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire.

Le calcul du coefficient binomial va alors prendre la forme qui suit :

```
1  binom_dict = {}
2
3  def binom(n, p):
4      if (n, p) not in binom_dict:
5          if p == 0 or n == p:
6              b = 1
7          else:
8              b = binom(n - 1, p - 1) + binom(n - 1, p)
9          binom_dict[(n, p)] = b
10     return binom_dict[(n, p)]
```

On peut observer que le programme récursif se retrouve presque mot pour mot lignes 5 à 8.

Calculons $\binom{5}{2}$ avec cette fonction, puis observons le contenu du dictionnaire :

```
In [1]: binom(5, 2)
Out[1]: 10

In [2]: binom_dict
Out[2]: {(3, 0): 1, (2, 0): 1, (1, 0): 1, (1, 1): 1, (2, 1): 2, (3, 1): 3, (4, 1): 4,
         (2, 2): 1, (3, 2): 3, (4, 2): 6, (5, 2): 10}
```

On peut constater qu'on y retrouve les 10 valeurs nécessaires pour réaliser ce calcul. J'ai représenté figure 9 l'ordre dans lequel ces valeurs ont été introduites dans le dictionnaire.

Remarque. La mémoïsation est tellement utile pour résoudre les problèmes de programmation dynamique que cette fonctionnalité est présente dans la librairie standard de Python : `lru_cache` est un décorateur de la bibliothèque `functools` qui associe automatiquement un dictionnaire à la définition d'une fonction récursive. Ainsi, pour ajouter à la fonction `binom` initiale un dictionnaire, il suffit d'écrire :

```
from functools import lru_cache

@lru_cache
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n - 1, p - 1) + binom(n - 1, p)
```

et vous avez une fonction récursive efficiente !

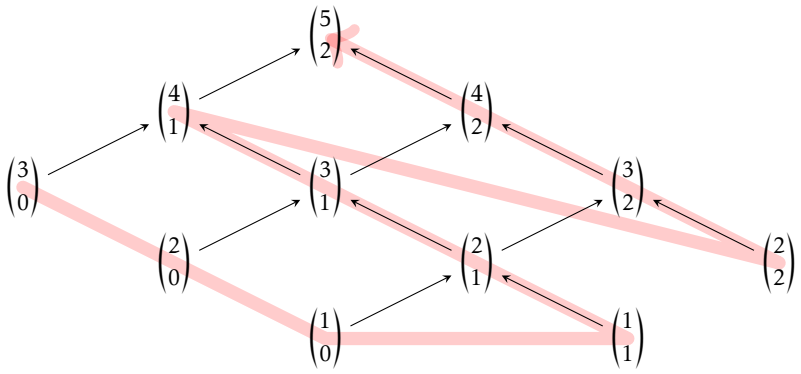


FIGURE 9 – Ordre d'entrée dans le dictionnaire.

2.2 Programmation dynamique et gloutonne

Tout comme les problèmes que l'on résout par une méthode « diviser pour régner », les problèmes que l'on résout par la programmation dynamique se ramènent à la résolution de sous-problèmes de tailles inférieures. Mais à la différence de la méthode « diviser pour régner », ces sous-problèmes *ne sont pas indépendants*, ce qui impose d'accompagner la programmation récursive par une analyse fine des relations de dépendance, ou beaucoup plus simplement par l'utilisation de la mémoïsation qui gère les relations de dépendance à notre place.

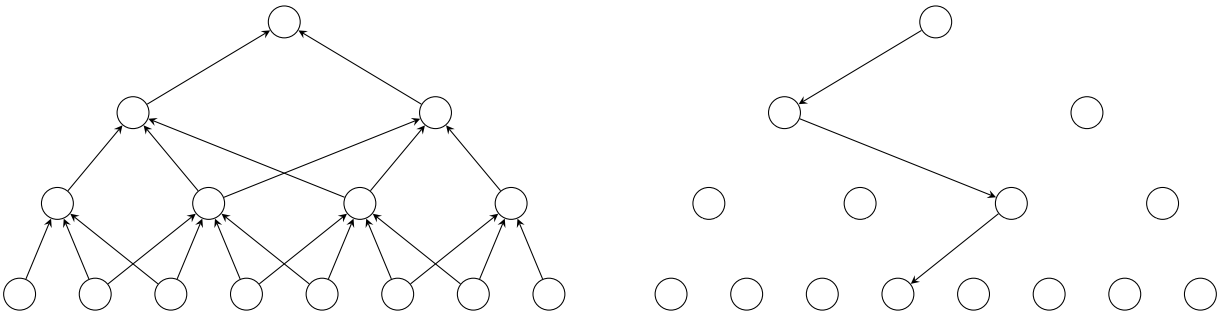


FIGURE 10 – Une illustration des programmations dynamique et gloutonne.

■ Problèmes d'optimisation

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique dès lors que la solution optimale peut être déduite des solutions optimales des sous-problèmes (c'est le *principe d'optimalité* de Bellman, du nom de son concepteur). Cette méthode garantit d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas sa complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.
Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la *programmation gloutonne*. Alors que la programmation dynamique se caractérise par la résolution par taille croissante de *tous* les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local *et un seul* en suivant une heuristique (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal). On ne peut en général garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci.

Pour apprécier la différence entre programmation dynamique et programmation gloutonne, nous allons maintenant étudier le problème suivant :

■ Le problème du sac à dos

Il se pose dans les termes suivants :

> *étant donnés n objets de valeurs $c_1,\dots,c_n$ et de poids respectifs $w_1,\dots,w_n$, comment remplir un sac à dos maximisant la valeur emportée $\sum_{i\in I} c_i$ tout en respectant la contrainte $\sum_{i\in I} w_i \leqslant W_{max}$?*

Pour élaborer un algorithme glouton résolvant le problème, il faut définir une heuristique, ici un critère de priorité pour le choix des objets à prendre. Nous pouvons par exemple choisir en priorité les objets dont le rapport $\dfrac{\text{valeur}}{\text{poids}} = \dfrac{c_i}{w_i}$ est maximal, et remplir le sac tant que c'est possible :

```
def glouton(c, w, Wmax):
    r = sorted(
        [(c[k], w[k]) for k in range(len(c))], key=lambda t: t[0] / t[1], reverse=True
    )
    s, p = 0, Wmax
    for k in range(len(c)):
        if r[k][1] <= p:
            s += r[k][0]
            p -= r[k][1]
    return s
```

Pour évaluer la qualité de cette heuristique, j'ai réalisé 1 000 expériences pour chacune desquelles j'ai :

- pris au hasard 100 objets de valeurs comprises entre 1 et 20 et de poids compris entre 1 et 30 ;
- calculé la valeur optimale obtenue pour un poids maximal égal à 300 à la fois par l'algorithme glouton ci-dessus et par l'algorithme dynamique que nous étudierons plus loin.

Sur les 1 000 expériences, 421 ont donné le même résultat pour chacun des deux algorithmes, et dans le cas des 579 autres, l'algorithme glouton a toujours rendu un résultat au moins égal à 98% du résultat de l'algorithme dynamique.
On peut donc considérer ici qu'à défaut de donner un résultat toujours exact, l'algorithme glouton donne un résultat acceptable tout en ayant une complexité moindre que l'algorithme dynamique.

La solution dynamique

Pour résoudre ce problème, nous allons noter $f(k,W)$ la valeur maximale qu'il est possible d'atteindre avec les k premiers objets pour un poids total égal à W.
Si l'objet d'indice k est dans la solution optimale, alors $w_k \leqslant W$ et $f(k,W) = c_k + f(k-1, W-w_k)$; s'il n'y est pas alors $f(k,W) = f(k-1,W)$. On en déduit :

$$f(k,W) = \begin{cases} \max\big(c_k + f(k-1,W-w_k), f(k-1,W)\big) & \text{si } w_k \leqslant W \\ f(k-1,W) & \text{sinon} \end{cases}$$

Pour calculer cette valeur, nous allons utiliser un tableau bi-dimensionnel de taille $(n+1)\times(W_{max}+1)$ destiné à contenir les valeurs de $f(k,w)$ pour $k \in [\![0,n]\!]$ et $W \in [\![0, W_{max}]\!]$.
Nous prendrons comme valeurs initiales $f(0,W) = f(k,0) = 0$, et notre but est de calculer $f(n, W_{max})$.

Pour remplir ce tableau, il est primordial de respecter l'ordre de dépendance des cases de ce tableau : la case $f(k,W)$ ne peut être calculée que lorsque les cases $f(k-1,W)$ et $f(k-1,W-w_k)$ auront été remplies.
En considérant que les valeurs c_k et w_k sont données sous forme de tableaux, on en déduit l'algorithme :

```
def sacAdos(c, w, Wmax):
    n = len(c)
    f = [[0 for j in range(Wmax + 1)] for i in range(n + 1)]
    for k in range(n):
        for W in range(Wmax + 1):
            if w[k] <= W:
                f[k + 1][W] = max(c[k] + f[k][W - w[k]], f[k][W])
            else:
                f[k + 1][W] = f[k][W]
    return f[n, Wmax]
```

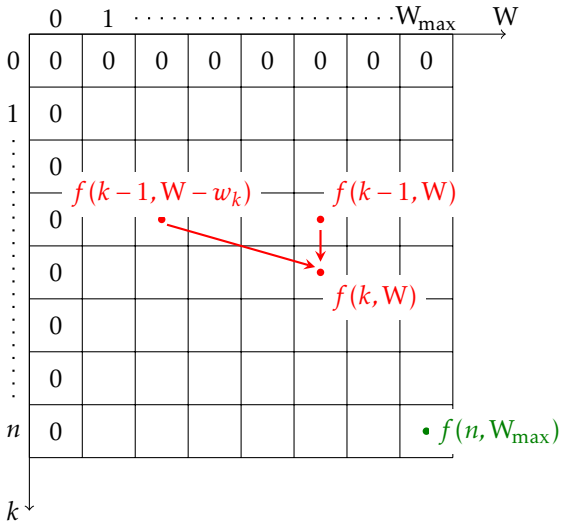


FIGURE 11 – Ordre de dépendance du sac à dos.

Il apparaît clairement que la complexité temporelle de cet algorithme est proportionnel au produit nW_{max}, soit en $O(nW_{max})$. L'algorithme glouton quant à lui est en $O(n \log n)$ (le coût du tri).

Remarque. Nous n'avons pas utilisé ici la technique de mémoïsation pour résoudre le problème. Cette dernière, lorsqu'elle est utilisée, nous permet de moins nous préoccuper de l'ordre de dépendance qui est géré par la récursivité :

```
def sacAdos(c, w, Wmax):
    dico = {}
    def f(k, W):
        if (k, W) not in dico:
            if k == 0 or W == 0:
                x = 0
            elif w[k - 1] <= W:
                x = max(c[k - 1] + f(k - 1, W - w[k - 1]), f(k - 1, W))
            else:
                x = f(k - 1, W)
            dico[(k, W)] = x
        return dico[(k, W)]
    return f(len(c), Wmax)
```

Remarque. Cet algorithme calcule la valeur maximale qui peut être emportée dans le sac, mais pas la façon d'y parvenir. Pour la connaître il faut utiliser le tableau (ou le dictionnaire) calculé par la fonction précédente, et retrouver le chemin qui mène de la case initiale à la case finale.
Par exemple, si on modifie la fonction non récursive (la première) pour qu'elle renvoie le tableau `f` qui a été calculé au lieu de la valeur `f[len(c)][Wmax]`, le fonction qui détermine les objets à choisir s'écrira :

```
def objetsAchoisir(c, w, Wmax):
    f = sacAdos(c, w, Wmax)
    sac = []
    k, W = len(c), Wmax
    while k > 0:
        if f[k][W] > f[k - 1][W]:
            sac.append((c[k - 1], w[k - 1]))
            W -= w[k - 1]
        k -= 1
    return sac
```

2.3 Distance d'édition

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.
Par exemple, on peut passer du mot `polynomial` au mot `polygonal` en suivant les étapes suivantes :

- suppression de la lettre `'i'` : `polynomial` $\to$ `polynomal` ;
- remplacement du `'n'` par un `'g'` : `polynomal` $\to$ `polygomal` ;
- remplacement du `'m'` par un `'n'` : `polygomal` $\to$ `polygonal` ;

donc la distance d'édition entre ces deux mots est égale au plus à 3, et on se convaincra aisément qu'il n'est pas possible de faire mieux.
Nous allons calculer la distance d'édition entre deux mots $a = a_1 a_2 \cdots a_m$ et $b = b_1 b_2 \cdots b_n$ en généralisant le problème, c'est à dire en définissant la distance d'édition $\mathrm{d}(i,j)$ entre les mots $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$.
Dans le chemin reliant de manière optimale $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$, plusieurs cas de figure peuvent se rencontrer :

- a_i a été supprimé, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j) + 1$;
- b_j a été ajouté, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i,j-1) + 1$;
- a_i a été remplacé par b_j, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1) + 1$;
- $a_i = b_j$, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1)$.

On en déduit que $\mathrm{d}(i,j) = \begin{cases} \min\big(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(i-1,j-1)\big) + 1 & \text{si } a_i \neq b_j \\ \min\big(\mathrm{d}(i-1,j) + 1, \mathrm{d}(i,j-1) + 1, \mathrm{d}(i-1,j-1)\big) & \text{si } a_i = b_j \end{cases}$.
Les conditions initiales sont clairement : $\mathrm{d}(i,0) = i$ et $\mathrm{d}(0,j) = j$, ce qui conduit au schema de dépendance représenté figure 12. puis à l'algorithme :

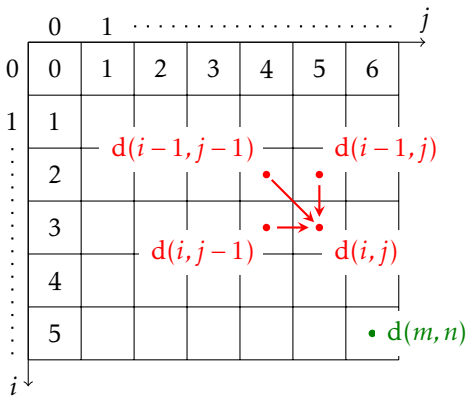


Figure 12 – Schéma de dépendance pour la distance de Levenshtein.

```python
def dist(a, b):
    m, n = len(a), len(b)
    d = [[0 for j in range (n + 1)] for i in range(m + 1)]
    for i in range(1, m + 1):
        d[i][0] = i
    for j in range(1][n + 1):
        d[0][j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                d[i][j] = min(d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1])
            else:
                d[i][j] = min(d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + 1)
    return d[m][n]
```

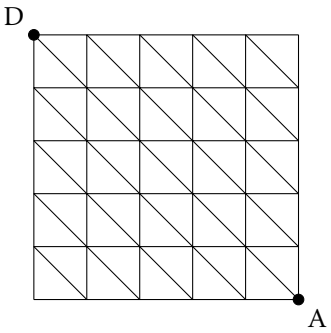
Cette fonction a une complexité temporelle et spatiale en $O(mn)$.

3. Exercices

Exercice 1

Partant du coin supérieur gauche d'une grille $n \times n$, on souhaite calculer le nombre de chemins menant au coin inférieur droit en ne suivant que les trois directions suivantes :

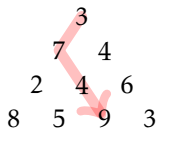
les trois directions possibles



Rédiger une fonction `chemins(n)` qui répond à la question.

Exercice 2

En partant du sommet du triangle ci-dessous et en se déplaçant vers les nombres adjacents de la ligne inférieure, le total maximum que l'on peut obtenir pour relier le sommet à la base est égal à 23 :



$3 + 7 + 4 + 9 = 23$

```
t = [[3],
     [7, 4],
     [2, 4, 6],
     [8, 5, 9, 3]]
```

Rédiger une fonction calculant le total maximum d'un chemin reliant le sommet à la base d'un tel triangle de hauteur n. On pourra considérer que les valeurs de ce triangle sont stockées dans un tableau bi-dimensionnel $n \times n$ (autrement dit, `t[i][j]` contient la $(j+1)^{\text{e}}$ valeur de la $(i+1)^{\text{e}}$ ligne).

Exercice 3

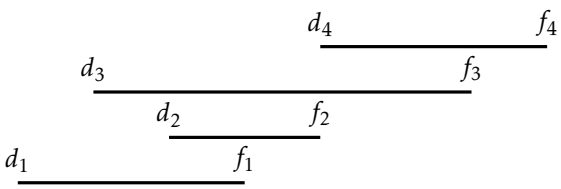
Étant donné une chaîne de caractères $a = a_1 a_2 \cdots a_m$, on appelle *sous-chaîne de longueur k* toute chaîne de caractère $a_{i_1} a_{i_2} \cdots a_{i_k}$ avec $1 < i_1 < i_2 < \cdots < i_k < m$.

a. Rédiger une fonction `pgscc(a, b)` qui, étant donné deux chaînes de caractères $a = a_1 \cdots a_m$ et $b = b_1 \cdots b_n$, renvoie la longueur de la plus grande sous-chaîne commune à a et b.

b. Modifier votre algorithme pour qu'il renvoie cette fois une sous-chaîne commune de longueur maximale.

Exercice 4 [Ordonnancement d'intervalles pondérés]

On considère une succession d'intervalles de temps $[d_i, f_i]$, $1 \leqslant i \leqslant n$, chacun d'eux étant associé à une valeur v_i. On dit que deux intervalles $[d_i, f_i]$ et $[d_j, f_j]$ *ne se chevauchent pas* lorsque $f_i \leqslant d_j$ ou $f_j \leqslant d_i$. Rédiger une fonction `ordonnancement(d, f, v)` qui détermine la somme maximale des valeurs d'une sous-famille d'intervalles ne se chevauchant pas. On supposera $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$.



Exercice 5

Si A et B sont deux matrices de tailles respectives $a \times b$ et $c \times d$, le produit AB n'est possible que si $b = c$, et dans ce cas la matrice produit AB est de taille $a \times d$, et peut être calculée à l'aide de acd multiplications.

Sachant que le produit matriciel est associatif mais pas commutatif, le produit ABC peut être calculé de deux manières : (AB)C ou A(BC), qui ne nécessitent pas *a priori* le même nombre de multiplications. Par exemple, si A est de taille 10×100, B de taille 100×5, et C de taille 5×50, le produit (AB)C nécessite $5000 + 2500 = 7500$ multiplications et le produit A(BC), $25000 + 50000 = 75000$ multiplications.

On considère une chaîne de matrices $\mathrm{M}_0\mathrm{M}_1\mathrm{M}_2\cdots\mathrm{M}_{n-1}$ de tailles respectives $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$. Rédiger une fonction calculant le nombre minimal de multiplications nécessaires pour effectuer ce produit matriciel. On pourra considérer que les valeurs de $m_0, m_1, \ldots, m_n$ sont rangées dans un tableau de taille $n+1$.

Chapitre III
Bases de données

1. Description d'une base de données

1.1 Introduction

Chacun d'entre nous utilise quotidiennement, mais sans souvent en avoir conscience, des bases de données. Ces dernières regroupent des informations relatives à un domaine précis, souvent ordonnées sous une forme très structurée. Les exemples sont innombrables : gestion des stocks (magasins en ligne, bibliothèques, ...), gestion des réservations (trains, avions, spectacles ...), gestion du personnel d'une entreprise, création de listes diverses (vos playlists musicales préférées par exemple), etc.

Prenons l'exemple du site de réservation de la SNCF. Les clients peuvent réaliser un certain nombre de tâches, parmi lesquelles :

- la consultation des trains répondants à certains critères (date, trajet, places disponibles, etc) ;
- la réservation d'une place dans le train choisi ;
- l'annulation d'une réservation.

Le personnel de la SNCF, outre les opérations précédentes, peut en outre :

- modifier la composition ou les horaires des trains ;
- ajouter des trains supplémentaires ;
- supprimer des trains.

À l'exception de la consultation, les autres opérations sont complexes car elles doivent gérer le principe de simultanéité qui régit les bases de données : plusieurs clients peuvent chercher à réserver en même temps des places dans le même train, et il ne s'agit pas d'attribuer la même place à plusieurs personnes différentes. Il en va de même de la suppression d'un train, qui doit tenir compte des personnes ayant déjà réservé dans celui-ci. C'est pourquoi nous nous limiterons, dans ce cours d'introduction aux bases de données, à la simple consultation des données.

1.2 Structure d'une base de données

L'accès à une base de données se fait usuellement par l'intermédiaire d'une application qui traduit les demandes de l'utilisateur (en général via une interface graphique) dans un langage dédié à la communication avec une base de données. Ce langage, le SQL (*Structured Query Language*) est devenu un standard disponible sur presque tous les systèmes de gestion des bases de données (SGBD).

Figure 1 – Communication avec une base de données.

Le SQL comporte des instructions relatives :

- à l'interrogation de bases de données ;
- à la création et la modification des données ;
- au contrôle d'accès des données.

Comme indiqué dans le préambule, nous nous intéresserons uniquement aux instructions de la première catégorie.

Dans la suite de ce cours j'utiliserai MySQL comme système de gestion pour interroger une base de données intitulée `world.sql`. Cette base de donnée contient un certain nombre d'informations géographiques et politiques mondiales (en 2001).
Commençons par observer son contenu :

```
mysql> show tables;
+-----------------+
| Tables_in_world |
+-----------------+
| city            |
| country         |
| countrylanguage |
+-----------------+
```

Une base de données est un ensemble de *tables* (ou de *relations*, les deux termes étant synonymes dans ce contexte) que l'on peut représenter sous forme de tableaux bi-dimensionnels. Dans l'exemple qui nous intéresse, notre base de données est composée de trois tables :

- `city`, qui contient des informations relatives à certaines villes du monde ;
- `country`, qui contient des informations relatives aux pays ;
- `countrylanguage`, qui contient des informations relatives aux langues parlées dans le monde.

Examinons le contenu de la première, ou plutôt un extrait de son contenu qui est très important :

```
mysql> SELECT * FROM city LIMIT 10;
+----+----------------+-------------+---------------+------------+
| ID | Name           | CountryCode | District      | Population |
+----+----------------+-------------+---------------+------------+
|  1 | Kabul          | AFG         | Kabol         |    1780000 |
|  2 | Qandahar       | AFG         | Qandahar      |     237500 |
|  3 | Herat          | AFG         | Herat         |     186800 |
|  4 | Mazar-e-Sharif | AFG         | Balkh         |     127800 |
|  5 | Amsterdam      | NLD         | Noord-Holland |     731200 |
|  6 | Rotterdam      | NLD         | Zuid-Holland  |     593321 |
|  7 | Haag           | NLD         | Zuid-Holland  |     440900 |
|  8 | Utrecht        | NLD         | Utrecht       |     234323 |
|  9 | Eindhoven      | NLD         | Noord-Brabant |     201843 |
| 10 | Tilburg        | NLD         | Noord-Brabant |     193238 |
+----+----------------+-------------+---------------+------------+
```

Nous pouvons constater qu'une table est un tableau bi-dimensionnel : les en-têtes des différentes colonnes sont appelés des *attributs*, les lignes des *enregistrements*. Ainsi, la table `city` possède 5 attributs : `ID`, `Name` (le nom d'une ville), `CountryCode` (le code du pays dans lequel se trouve cette ville), `District` (sa région d'appartenance) et `Population` (son nombre d'habitants).

Précisons maintenant les caractéristiques de chacun de ces attributs :

```
mysql> describe city;
+-------------+----------+------+-----+
| Field       | Type     | Null | Key |
+-------------+----------+------+-----+
| ID          | int      | NO   | PRI |
| Name        | char(35) | NO   |     |
| CountryCode | char(3)  | NO   |     |
| District    | char(20) | NO   |     |
| Population  | int      | NO   |     |
+-------------+----------+------+-----+
```

Nous constatons qu'un attribut est un objet *typé* (dans ce contexte, le type d'un attribut est son *domaine*). `ID` et `Population` sont des entiers, `Name` une chaîne d'au plus 35 caractères, `CountryCode` une chaîne d'au plus 3 caractères, et `District` une chaîne d'au plus 20 caractères. Ces caractéristiques sont fixées lors de la création d'une table et influent sur les opérations que nous serons susceptibles de réaliser sur les valeurs de ces attributs :

`+`, `-`, `*`, `*` pour les entiers et les nombres flottants, `=`, `<>`, `<`, `<=`, `>`, `>=` pour les comparaisons (pour les chaînes de caractères, l'ordre lexicographique est utilisé).
La colonne suivante indique que tous les attributs doivent posséder une valeur ; enfin, la troisième colonne apporte une information importante : chaque enregistrement d'une table doit pouvoir être identifié de manière unique ; c'est le rôle de la *clef primaire*, constitué d'un ou plusieurs attributs. Ici, la clef primaire est constituée de l'unique attribut `ID`. Il ne peut donc y avoir deux enregistrements ayant le même `ID`.

Regardons maintenant le contenu de la table `country` :

```
mysql> describe country;
+----------------+---------------------------------------------------+------+-----+
| Field          | Type                                              | Null | Key |
+----------------+---------------------------------------------------+------+-----+
| Code           | char(3)                                           | NO   | PRI |
| Name           | char(52)                                          | NO   |     |
| Continent      | enum('Asia','Europe','North America',             | NO   |     |
|                | 'Africa','Oceania','Antarctica','South America') |      |     |
| Region         | char(26)                                          | NO   |     |
| SurfaceArea    | decimal(10,2)                                     | NO   |     |
| IndepYear      | smallint                                          | YES  |     |
| Population     | int                                               | NO   |     |
| LifeExpectancy | decimal(3,1)                                      | YES  |     |
| GNP            | decimal(10,2)                                     | YES  |     |
| GovernmentForm | char(45)                                          | NO   |     |
| HeadOfState    | char(60)                                          | YES  |     |
| Capital        | int                                               | YES  |     |
+----------------+---------------------------------------------------+------+-----+
```

Ici, la clef primaire est l'attribut `Code`, qui est une chaîne de trois caractères. On constate en outre que certaines données ne sont pas obligatoires : les attributs `IndepYear`, `LifeExpectancy`, `GNP`, `HeadOfState` et `Capital` peuvent être absents, auquel cas la valeur qui leur est attribuée est `NULL`.
Voyons un extrait de cette table, par exemple l'enregistrement qui concerne la France :

```
mysql> select * from country where Name = 'France';
+------+--------+-----------+----------------+-------------+-----------+------------+
| Code | Name   | Continent | Region         | SurfaceArea | IndepYear | Population |
+------+--------+-----------+----------------+-------------+-----------+------------+
| FRA  | France | Europe    | Western Europe |   551500.00 |       843 |   59225700 |
+------+--------+-----------+----------------+-------------+-----------+------------+

+----------------+------------+----------------+----------------+---------+
| LifeExpectancy | GNP        | GovernmentForm | HeadOfState    | Capital |
+----------------+------------+----------------+----------------+---------+
|           78.8 | 1424285.00 | Republic       | Jacques Chirac |    2974 |
+----------------+------------+----------------+----------------+---------+
```

Deux attributs sont remarquables : l'attribut `Code` (clef primaire de la table) correspond à l'attribut `CountryCode` de la table `city`. Cette remarque est importante car c'est elle qui nous permettra de chercher des informations croisées dans ces deux tables. L'attribut `Capital` est plus intriguant : pourquoi la capitale de la France est-elle désignée par un entier ? Il se trouve que cet attribut correspond à l'attribut `ID` de la table `city`, comme on peut s'en convaincre en consultant l'enregistrement dont l'attribut `ID` vaut `2974` :

```
mysql> select * from city where ID = 2974;
+------+-------+-------------+----------------+------------+
| ID   | Name  | CountryCode | District       | Population |
+------+-------+-------------+----------------+------------+
| 2974 | Paris | FRA         | Île-de-France  |    2125246 |
+------+-------+-------------+----------------+------------+
```

Regardons enfin le contenu de la troisième et dernière table :

```
mysql> describe countryLanguage;
+-------------+---------------+------+-----+
| Field       | Type          | Null | Key |
+-------------+---------------+------+-----+
| CountryCode | char(3)       | NO   | PRI |
| Language    | char(30)      | NO   | PRI |
| IsOfficial  | enum('T','F') | NO   |     |
| Percentage  | decimal(4,1)  | NO   |     |
+-------------+---------------+------+-----+
```

Cette table décrit les langues utilisées dans les différents pays en précisant le pourcentage de locuteurs et si cette langue est officielle ou non. Une même langue peut être parlée dans différents pays, et dans un même pays plusieurs langues peuvent être pratiquées. Ainsi, aucun des attributs ne peut prétendre être une clef primaire, c'est pourquoi nous avons ici un couple d'attributs en guise de clef primaire : le couple `(CountryCode, Language)`.

Ainsi, pour pouvoir efficacement interroger une base de données, il importe de connaître avec précision le contenu de chacune des tables, et les attributs qui vont nous permettre de les relier entre elles. Cet ensemble d'informations est appelé le *schéma de la base de donnée* ; celui de notre exemple est représenté figure 2.

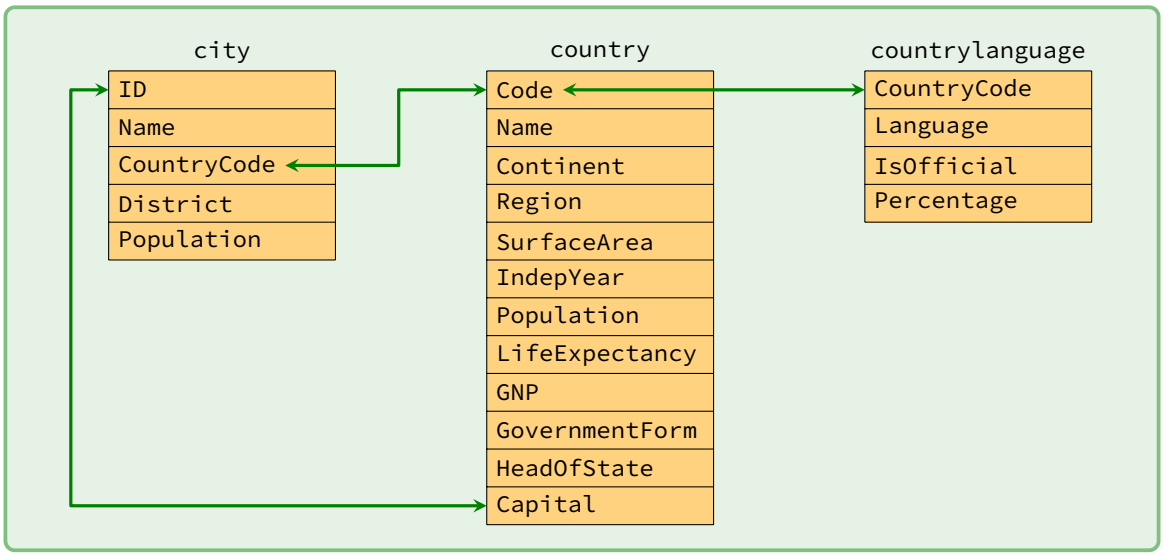


FIGURE 2 – le schéma de la base de données `world`.

2. Requêtes SQL

Nous allons maintenant nous intéresser à la syntaxe des requêtes qui vont nous permettre d'interroger la base de données. Cette syntaxe est proche de la langue anglaise ; en général, la lecture d'une requête permet d'en comprendre le sens. Elle n'en reste pas moins assez rigide dans sa structure, et les mots clefs que nous allons utiliser doivent être rangés dans un ordre bien précis :

SELECT ... **FROM** ... **WHERE** ... **GROUP BY** ... **HAVING** ... **ORDER BY** ... **LIMIT** ... **OFFSET** ...

Seuls les deux premiers (select et from) sont indispensables, les autres sont optionnels, mais s'ils sont présents ils doivent être placés dans cet ordre.
Notons enfin que ces noms de commandes sont insensibles à la casse. Autrement dit, vous pouvez les écrire indifféremment aen majuscules ou en minuscule.

2.1 Sélection des attributs et des enregistrements

- On sélectionne un ou plusieurs attributs d'une table par la syntaxe :

```
SELECT a1, a2, ... , an FROM table
SELECT DISTINCT a1, a2, ... , an FROM table
```

Le mot-clef distinct permet d'éviter l'apparition de doublons parmi les résultats obtenus.
En algèbre relationnelle, cette opération s'appelle une *projection* et se note $\boxed{\pi_{(A_1,\dots,A_n)}(R)}$ où $A_1,\dots,A_n$ sont les attributs sélectionnés dans la relation R.

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_1	b_1	c_3

$\pi_{(A,B)}(R)$

A	B
a_1	b_1
a_2	b_2

Exemple. La liste des différentes langues présentes dans la base de donnée `world` s'obtient en écrivant :

```
mysql> SELECT DISTINCT Language FROM countrylanguage;
+---------------------------+
| Language                  |
+---------------------------+
| Dutch                     |
| English                   |
| Papiamento                |
| Spanish                   |
.............................
| Bemba                     |
| Chewa                     |
| Lozi                      |
| Nsenga                    |
+---------------------------+
457 rows in set (0,00 sec)
```

457 langues sont présentes dans la base de données.

- On sélectionne les enregistrements d'une table qui satisfont une expression logique par la syntaxe :

```
SELECT * FROM table WHERE expression_logique
```

En algèbre relationnelle cette opération s'appelle une *sélection* et se note $\boxed{\sigma_E(R)}$ où E est l'expression logique et R une relation.

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_4	c_4

$\sigma_E(R)$

A	B	C
a_2	b_2	c_2
a_4	b_4	c_4

Ces deux opérations peuvent bien entendu être combinées pour extraire de la table une sélection d'attributs et d'enregistrements :

```
SELECT a1, ... , an FROM table WHERE expression_logique
```

ce qui se note en algèbre relationnelle : $\pi_{(A_1,\dots,A_n)}(\sigma_E(R))$.
Par exemple, nous pouvons visualiser les villes françaises et leurs populations respectives en écrivant :

```
mysql> SELECT Name, Population FROM city WHERE CountryCode = 'FRA';
+----------------------+------------+
| Name                 | Population |
+----------------------+------------+
| Paris                |    2125246 |
| Marseille            |     798430 |
| Lyon                 |     445452 |
| Toulouse             |     390350 |
....................................
| Argenteuil           |      93961 |
| Tourcoing            |      93540 |
| Montreuil            |      90674 |
+----------------------+------------+
40 rows in set (0,00 sec)
```

- Le *renommage* permet la modification du nom d'un attribut d'une relation. Renommer l'attribut a en l'attribut b dans la relation R s'écrit $\boxed{\rho_{a\leftarrow b}(\mathrm{R})}$ en algèbre relationnelle et à l'aide du mot-clef as en SQL :

```
SELECT a AS b FROM table
```

  La nécessité du renommage apparaîtra lorsque nous aborderons les sous-requêtes.

Filtrage des résultats

À la toute fin d'une requête il est possible de trier les résultats et de n'en renvoyer qu'une partie. Ces opérations se notent :

- **ORDER BY** *a* **ASC** / **DESC** pour trier suivant l'attribut a par ordre croissant/décroissant ;
- **LIMIT** n pour limiter la sortie à n enregistrements ;
- **OFFSET** n pour débuter à partir du n^{e} enregistrement.

Par exemple, les cinq villes mondiales les plus peuplées sont :

```
mysql> SELECT Name FROM city ORDER BY Population DESC LIMIT 5;
+-----------------+
| Name            |
+-----------------+
| Mumbai (Bombay) |
| Seoul           |
| São Paulo       |
| Shanghai        |
| Jakarta         |
+-----------------+
```

et les cinq suivantes sont :

```
mysql> SELECT Name FROM city ORDER BY Population DESC LIMIT 5 OFFSET 5;
+-------------------+
| Name              |
+-------------------+
| Karachi           |
| Istanbul          |
| Ciudad de México  |
| Moscow            |
| New York          |
+-------------------+
```

Exercice 1

a. Rédiger une requête SQL donnant le nom des dix pays asiatiques les plus grands.

b. Rédiger une requête SQL donnant le nom et la date d'indépendance des dix pays les plus anciens.

c. Rédiger une requête SQL donnant la liste des langues non officielles pratiquées en France, par ordre décroissant d'importance.

d. Rédiger une requête SQL donnant le nom des cinq pays européens les moins densément peuplés.

2.2 Jointures et sous-requêtes

La *jointure* est une opération qui porte sur deux relations R_1 et R_2 et retourne une relation qui comporte les enregistrements combinés de R_1 et de R_2 qui satisfont une contrainte logique E. Cette nouvelle relation se note $\boxed{R_1 \bowtie_E R_2}$.

En SQL on réalise une jointure par la requête :

```
SELECT * FROM table1 JOIN table2 ON expression_logique
```

R_1

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3

R_2

D	E
a_1	e_1
a_2	e_2
a_2	e_3

$R_1 \bowtie_{A=D} R_2$

A	B	C	D	E
a_1	b_1	c_1	a_1	e_1
a_2	b_2	c_2	a_2	e_2
a_2	b_2	c_2	a_2	e_3

Seules les conditions d'égalité entre deux attributs figurent au programme.

Remarque. Deux tables reliées par une jointure peuvent posséder des attributs de même nom mais n'ayant rien à voir. C'est le cas par exemple de l'attribut `Name` présent dans les deux tables `city` et `country`. Pour les distinguer lors d'une jointure il est nécessaire de faire précéder le nom de l'attribut par le nom de la table, séparé par un point. Par exemple, lors d'une jointure entre les deux tables de notre base de données exemple, le nom des villes est désigné par `city.name` et celui des pays par `country.Name`.

Notons que le mot-clef as permet de renommer les tables ainsi que les attributs afin d'éviter d'écrire des noms à rallonge.

Exemple. Les deux tables `city` et `country` possèdent deux jointures naturelles :

- on peut identifier les attributs `city.countryCode` et `country.Code`. Dans ce cas, la table résultante de la jointure possédera autant d'entrées que la table `city` et permettra de relier à chaque ville de la base les informations du pays auquel elle appartient ;
- on peut identifier les attributs `city.ID` et `country.Capital`. Dans ce cas, la table résultante de la jointure possédera autant d'entrées que la table `country` et à chaque pays sera attaché les informations relatives à sa capitale. En revanche, les villes qui ne sont pas des capitales ne seront pas présentes dans cette jointure.

Par exemple, si on souhaite connaître la capitale de l'Ouzbekistan et son nombre d'habitants on écrira :

```
mysql> SELECT city.Name, city.Population
    -> FROM city JOIN country ON city.ID = country.Capital
    -> WHERE country.Name = 'Uzbekistan';
+---------+------------+
| Name    | Population |
+---------+------------+
| Toskent |    2117500 |
+---------+------------+
```

Exercice 2

a. Rédiger une requête SQL donnant la liste des pays ayant adopté le Français parmi leurs langues officielles.

b. Rédiger une requête SQL donnant les cinq villes les plus peuplées d'Europe.

c. Rédiger une requête SQL donnant les cinq villes les plus peuplées d'Europe parmi celles qui ne sont pas des capitales.

d. Rédiger une requête SQL donnant les capitales des pays dans lesquels l'allemand est langue officielle.

■ Sous-requêtes

Revenons au problème consistant à déterminer la capitale de L'Ouzbekistan. Nous l'avons résolu à l'aide d'une jointure, mais il aurait aussi été possible de procéder à deux requêtes successives : une première requête pour déterminer l'identifiant de la capitale de l'Ouzbekistan, une seconde pour connaître le nom de cette ville.
Il est possible d'imbriquer la première au sein de la seconde pour n'en faire qu'une seule ; on parle alors de sous-requête. Celle-ci doit impérativement être délimitée par des parenthèses, et peut être située :

- en lieu et place d'une table après le from ;
- ou au sein d'une expression logique après le where.

Par exemple, la détermination de la capitale de l'Ouzbekistan peut être obtenue par la requête :

```
mysql> SELECT city.Name FROM city
    -> WHERE ID = (SELECT Capital FROM country WHERE name = 'Uzbekistan');
+---------+
| Name    |
+---------+
| Toskent |
+---------+
```

Exercice 3

a. Donner la liste des pays dont l'espérance de vie est supérieure ou égale à celle de la France.

b. Donner le pourcentage de pays dont le PNB est supérieur ou égal à la moyenne.

Notons enfin que lorsqu'une sous-requête prend la place d'une table, il est impératif de renommer les attributs de la sous-requête pour qu'ils soient utilisés dans la requête principale.

2.3 Fonctions d'agrégation

SQL possède un certain nombre de fonctions statistiques (voir la liste figure 3) qui par défaut s'appliquent à l'ensemble des enregistrements sélectionnés par la clause du where.

COUNT()	nombre d'enregistrements
MAX()	valeur maximale d'un attribut
MIN()	valeur minimale d'un attribut
SUM()	somme d'un attribut
AVG()	moyenne d'un attribut

Figure 3 – Fonctions statistiques.

Par exemple, pour obtenir la population de l'Europe on écrit :

```
mysql> SELECT SUM(Population) FROM country WHERE Continent = 'Europe';
+-----------------+
| SUM(Population) |
+-----------------+
|       730074600 |
+-----------------+
```

Exercice 4

Rédiger une requête SQL déterminant la ville la moins peuplée de la base de données.

Mais il est aussi possible de regrouper les enregistrements d'une table par *agrégation* à l'aide du mot-clef group by. Ce regroupement permet d'appliquer la fonction statistique à chacun des groupes : le résultat de la requête est l'ensemble des valeurs prises par la fonction statistique sur chacun des regroupements.

R

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_3	c_3
a_3	b_4	c_4
a_3	b_5	c_5

group by A →

R

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_3	c_3
a_3	b_4	c_4
a_3	b_5	c_5

On utilise la syntaxe suivante (dans laquelle a_1 et a_2 sont des attributs et f une fonction d'agrégation) :

```
SELECT f(a1) FROM table WHERE expression_logique GROUP BY a2
```

Cette requête sélectionne les enregistrements de la table qui vérifient l'expression logique, les regroupe selon la valeur de l'attribut a_2, puis applique la fonction f sur l'attribut a_1 à chacun des groupes obtenus.

Notons enfin que le mot-clef having permet d'imposer des conditions sur les groupes à qui on applique la fonction d'agrégation. La syntaxe générale de la requête prend alors la forme suivante :

```
SELECT f(a1) FROM table WHERE e1 GROUP BY a2 HAVING e2
```

Cette requête sélectionne les enregistrements de la table qui vérifient l'expression logique e_1, les regroupe selon la valeur de l'attribut a_2, puis applique la fonction f sur l'attribut a_1 à chacun des groupes vérifiant l'expression logique e_2.
Par exemple, pour obtenir les populations de chacun des continents on écrira :

```
mysql> SELECT Continent, SUM(Population) FROM country
    -> GROUP BY Continent;
+---------------+------------------+
| Continent     | SUM(Population) |
+---------------+------------------+
| North America |       482993000 |
| Asia          |      3705025700 |
| Africa        |       784475000 |
| Europe        |       730074600 |
| South America |       345780000 |
| Oceania       |        30401150 |
| Antarctica    |               0 |
+---------------+------------------+
```

Pour ordonner cette liste, il faut renommer l'attribut calculant la population totale de chaque continent :

```
mysql> SELECT Continent, SUM(Population) AS Pop FROM country
    -> GROUP BY Continent ORDER BY Pop DESC;
+---------------+------------+
| Continent     | Pop        |
+---------------+------------+
| Asia          | 3705025700 |
| Africa        |  784475000 |
| Europe        |  730074600 |
| North America |  482993000 |
| South America |  345780000 |
| Oceania       |   30401150 |
| Antarctica    |          0 |
+---------------+------------+
```

Enfin, si on veut se restreindre aux continents qui comportent plus de 40 pays, on écrira :

```
mysql> SELECT Continent, SUM(Population), COUNT(*) FROM country
    -> GROUP BY Continent HAVING COUNT(*) > 40;
+-----------+-----------------+----------+
| Continent | SUM(Population) | COUNT(*) |
+-----------+-----------------+----------+
| Asia      |      3705025700 |       51 |
| Africa    |       784475000 |       58 |
| Europe    |       730074600 |       46 |
+-----------+-----------------+----------+
```

Exercice 5

a. Rédiger une requête SQL donnant la liste des districts des États-Unis dont la population totale est supérieure à 3 000 000.

b. Rédiger une requête SQL déterminant les cinq langues les plus parlées au monde.

c. Rédiger une requête SQL déterminant, pour chaque continent, le pays le plus peuplé.

2.4 Opérateurs ensemblistes

On peut procéder à des opérations ensemblistes entre deux tables (en général le résultat de requêtes) à condition que celles-ci aient même structure. Ces opérations sont :

- union pour calculer la réunion $A \cup B$ de deux requêtes ;
- intersect pour calculer l'intersection $A \cap B$ de deux requêtes ;
- except pour calculer la différence $A \setminus (A \cap B)$ de deux requêtes.

Ces opérations présentent surtout un intérêt lorsque les deux requêtes sont issues de deux tables différentes. MySQL (ainsi que d'autres SGBD) n'implémente d'ailleurs que l'union, les deux autres opérations ensemblistes pouvant être aisément remplacées par des requêtes équivalentes :

- select a from table1 intersect select b from table2 est équivalent à

```
SELECT a FROM table1 WHERE a IN (SELECT b FROM table2)
```

- select a from table1 except select b from table2 est équivalent à

```
SELECT a FROM table1 WHERE a NOT IN (SELECT b FROM table2)
```

Autrement dit, seule l'union présente un réel intérêt.

Il est enfin possible de réaliser le produit cartésien de deux tables (ou plus) en suivant la syntaxe

```
SELECT A1, A2 FROM table1, table2
```

mais la création d'un produit cartésien doit être réalisé avec prudence, car le cardinal du résultat est égal au produit des cardinaux des tables qui le composent :

```
mysql> select count(*) from city
    -> union select count(*) from country
    -> union select count(*) from countrylanguage
    -> union select count(*) from city, country, countrylanguage;
+-----------+
| count(*)  |
+-----------+
|      4079 |
|       239 |
|       984 |
| 959282904 |
+-----------+
```

3. Exercices

On souhaite utiliser une base de données pour stocker les résultats obtenus par une communauté de joueurs en ligne. On suppose qu'on dispose d'une base de données comportant deux tables : `Joueurs(id_j, nom, pays)` et `Parties(id_p, date, duree, score, id_joueur)` où :

- `id_j` de type entier, est la clef primaire de la table `Joueurs` ;
- `nom` est une chaîne de caractères donnant le nom du joueur ;
- `pays` est une chaîne de caractères donnant le pays du joueur ;
- `id_p` de type entier, est la clef primaire de la table `Parties` ;
- `date` est la date (AAAAMMJJ) de la partie ;
- `duree` de type entier, est la durée en secondes de la partie ;
- `score` de type entier, est le nombre de points marqués au cours de la partie ;
- `id_joueur` est un entier qui identifie le joueur de la partie.

Exercice 6

Rédiger une requête SQL qui renvoie la date, la durée et le score de toutes les parties jouées par Alice, listées par ordre chronologique.

Exercice 7

Alice vient de réaliser un score de n points. Rédiger une requête SQL qui renvoie la position qu'aura le score n dans le classement des parties par ordre de score (on suppose que la partie que vient de jouer Alice n'est pas encore insérée dans la base de données). En cas d'ex aequo pour le score n le rang sera le même pour tous les joueurs ayant ce score.

Exercice 8

Rédiger une requête SQL qui renvoie le record de France pour ce jeu.

Exercice 9

Rédiger une requête SQL qui renvoie le rang d'Alice, c'est-à-dire sa position dans le classement des joueurs par ordre de leur meilleur score.

Chapitre IV
Théorie des jeux

Dans ce chapitre nous allons nous intéresser à l'étude théorique et algorithmique de jeux à deux joueurs antagonistes jouant alternativement, joueurs que l'on dénomme traditionnellement Adam et Ève. Les jeux qui nous intéressent sont à information totale : à tout instant d'une partie chacun des joueurs a une vision complète de l'état du jeu. Ceci exclus la plus-part des jeux de cartes (on ne connait pas le jeu de l'adversaire) mais inclus des jeux tels les échecs, les dames, le go, etc.
Dans un premier temps, nous allons nous intéresser à des jeux « simples » pour lesquels il est possible de déterminer (au moins pour de petites configurations) une stratégie gagnante, puis nous verrons, pour les jeux les plus complexes, comment bâtir une stratégie à l'aide d'une heuristique.

1. Jeux sur un graphe

1.1 Le jeu de Chomp

Le premier jeu auquel nous allons nous intéresser se joue à l'aide d'une tablette de chocolat rectangulaire dont le coin supérieur gauche est empoisonné : Chaque joueur choisit à tour de rôle un carré et le mange, ainsi

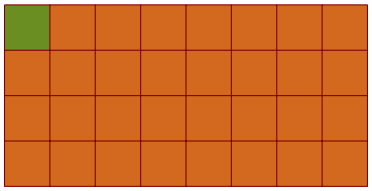


Figure 1 – La configuration initiale du jeu de Chomp.

que tous les morceaux situés à la droite et en dessous du carré choisi. Bien évidemment, le joueur qui n'a plus d'autre choix que de manger le carré empoisonné a perdu.
On trouvera figure 2 un exemple de partie perdue par Adam, qui a commencé à jouer en premier.

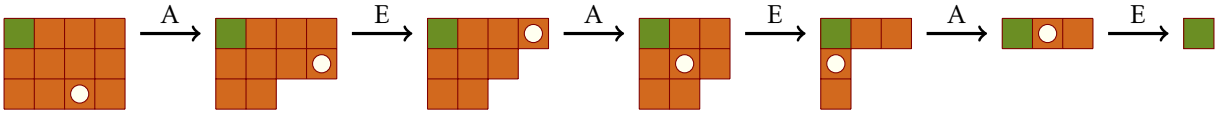


Figure 2 – Un exemple de partie du jeu de Chomp.

■ Modélisation

À ce type de jeu est associé un graphe orienté (S, A) appelé *arène*. Chaque sommet de S représente une configuration du jeu (une *position*), l'une d'entre elles étant la *position de départ*. Une arête $a = (s_1, s_2) \in A$ reliant deux sommets indique la possibilité pour un joueur de passer de la position s_1 à la position s_2 en un coup. Par exemple, l'arène associée au jeu de Chomp pour une tablette $(2, 3)$ est représentée figure 3.
Pour visualiser une partie de Chomp, il suffit d'imaginer un jeton initialement posé sur la position initiale s_0. À tour de rôle, chaque joueur le déplace le long d'une arête issue de la position courante s et le pose sur un successeur de s. Une *partie* est donc un chemin d'origine s_0 dans l'arène.
Ce jeu fait partie des jeux d'*accessibilité* : le graphe associé ne comporte pas de cycle (ce qui assure que toute partie est finie), et il est déterminé par un ensemble de positions (les *cibles*) qui sont sans successeurs. Ainsi, dans le jeu de Chomp le nœud du graphe associé au seul carré empoisonné est la seule cible du jeu, et l'atteindre signifie la fin de la partie (et la victoire pour celui qui l'atteint).

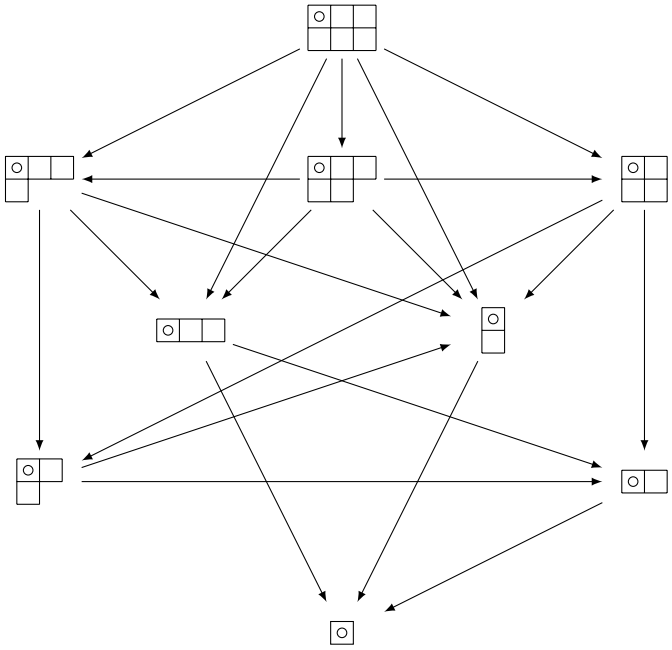


FIGURE 3 – L'arène associée au jeu de Chomp (2, 3).

Une fois fixé le joueur qui commence (Adam par exemple), on peut, en doublant chacun des sommets qu'on indexera par le nom du joueur, créer un nouveau graphe dont les sommets seront partitionnés en deux sous-ensembles $S = S_a \cup S_e$ avec $S_a \cap S_e = \varnothing$ où S_i est l'ensemble des positions à partir desquelles le joueur i jouera. Un tel graphe est dit *biparti* (illustration figure 4).

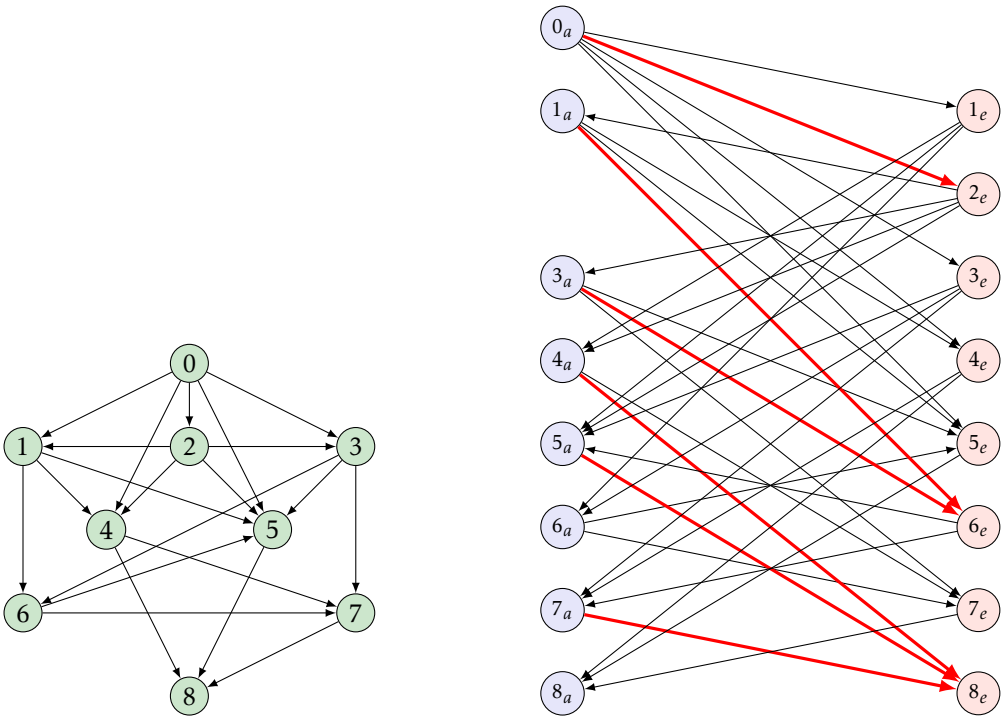


FIGURE 4 – Graphe et graphe biparti associés au jeu de Chomp (2, 3).

Dans un graphe biparti, les arêtes ne peuvent relier qu'un nœud de S_a à un nœud de S_e, et réciproquement.

Stratégies gagnantes
Une *stratégie* pour Adam est une application $f : S'_a \subset S_a \to S_e$ tel que pour $s \in S'_a$, $(s, f(s))$ est une arête du graphe biparti. Ainsi, de manière informelle, une stratégie consiste à déterminer, pour chaque position de S'_a, le mouvement suivant à jouer. Une partie $s_0 \to s_1 \to \cdots \to s_n$ est dite *jouée suivant* f lorsque pour tout $k \in [\![0, n-1]\!]$, si $s_k \in S'_a$ alors $s_{k+1} = f(s_k)$. Une stratégie est dite *gagnante* pour Adam si toute partie jouée en suivant cette stratégie est gagnante pour Adam. On définit bien évidemment de manière analogue les stratégies et les stratégies gagnantes pour Ève.
Sur le graphe biparti de la figure 4 j'ai fait figurer en gras une stratégie gagnante pour Adam. Il doit commencer par poser le jeton sur le sommet 2, puis :

- si Ève joue son coup suivant sur 1 ou 3, poursuivre sur 6. Ève ne peut alors que suivre sur 5 ou 7, et Adam peut alors conclure en jouant sur 8 ;
- si Ève joue son coup suivant sur 4 ou 5, poursuivre (et terminer) sur 8.

Ève possède elle aussi une stratégie gagnante définie par $f(1_e) = f(3_e) = 6_a$ et $f(4_e) = f(5_e) = f(7_e) = 8_a$, mais comme elle ne joue pas en premier il est nécessaire qu'Adam ne joue pas sur le sommet 2 au début de la partie pour qu'Ève puisse suivre cette stratégie.

Positions gagnantes
Une position s est dite *gagnante* lorsqu'il existe une stratégie gagnante pour une partie débutant au sommet s.
Par exemple, pour le jeu de Chomp $(2,3)$ les positions 0, 1, 3, 4, 5, 7 sont gagnantes, les positions 2, 6 et 8 sont perdantes.

Théorème 1.1 — *Dans le jeu de Chomp (p,q) il existe une stratégie gagnante pour le premier joueur dès lors que $(p,q) \neq (1,1)$.*

Démonstration. Si $p = 1$ ou $q = 1$, un coup suffit à Adam pour atteindre la position finale.
Supposons maintenant $p > 1$ et $q > 1$, et faisons choisir à Adam le carré en bas à droite. Supposons que ce coup soit perdant ; dans ce cas Ève possède un coup gagnant. Mais tout mouvement choisi par Ève à cette étape du jeu aurait pu être choisi par Adam pour entamer la partie. Cela signifie qu'ou bien ce premier mouvement est gagnant, ou bien il en existe un autre qui soit gagnant. Dans tous les cas, Adam possède donc un coup gagnant. □

Ce type de preuve est appelé un *vol de stratégie*. C'est malheureusement une preuve non constructive : savoir qu'une stratégie gagnante existe ne suffit pas à nous apprendre comment la trouver. C'est ce à quoi nous allons employer maintenant.

1.2 Détermination des positions gagnantes

Considérons un graphe orienté acyclique (S, A) associé à un jeu d'accessibilité.

Lemme — *Dans un graphe acyclique il existe des sommets sans successeur.*

Démonstration. Supposons qu'il n'existe pas de sommet sans successeur. Partant d'un sommet s_0 quelconque on pourrait alors construire une suite (s_n) de sommets tels que pour tout $n \in \mathbb{N}$, $(s_n, s_{n+1}) \in A$. Mais S est de cardinal fini donc il existe $p < q$ tel que $s_p = s_q$, ce qui signifie que $(s_p, s_{p+1}, \ldots, s_q)$ est un cycle, ce qui est absurde. □

Définition. — *Un sous-ensemble S' de sommets est dit* stable *si tout sommet de S' n'a aucun successeur dans S'. Un sous-ensemble S' de sommets est dit* absorbant *si tout sommet n'appartenant pas à S' possède au moins un successeur dans S'. Un sous-ensemble S' de sommets est un* noyau *s'il est à la fois stable et absorbant.*

Par exemple, dans le jeu de Chomp $(2,3)$, les sommets $(2,6,8)$ forment un noyau du graphe.

Théorème 1.2 — *Tout graphe orienté et acyclique possède un unique noyau.*

Démonstration. Raisonnons par récurrence sur le nombre n de sommets.
– si $n = 1$, l'unique sommet est aussi l'unique noyau du graphe.

– Si $n > 1$, supposons le résultat acquis jusqu'au rang $n-1$. Par hypothèse, il existe au moins un sommet s sans successeur. Nécessairement, ce sommet doit appartenir à un éventuel noyau (car un noyau est absorbant). Considérons le graphe (S', A') obtenu en supprimant de (S, A) le sommet s ainsi que ces prédécesseurs.

– Si (S', A') est le graphe vide, c'est que $\{s\}$ est un noyau de (S, A), et c'est le seul (tous les autres sommets admettent s comme successeur).

– Dans le cas contraire, observons que (S', A') reste acyclique. Par hypothèse de récurrence il possède un unique noyau N'. Dans ce cas, $N = N' \cup \{s\}$ est un noyau de (S, A).
Pour justifier qu'il est unique, considérons un noyau quelconque N de (S, A). Il doit contenir s, et $N \setminus \{s\}$ est un noyau de (S', A'). Il est donc égal à N'.

□

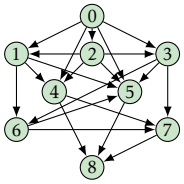
Cette preuve a le mérite de d'être constructive ; pour calculer le noyau de (S, A) il suffit de :

– chercher un sommet s de (S, A) sans successeur ;

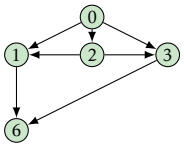
– supprimer de (S, A) s et ses prédécesseurs ;

– recommencer tant qu'il reste de sommets.

Voici par exemple comment on calcule le noyau du jeu de Chomp $(2, 3)$:



8 n'a pas de successeur ; il appartient au noyau et on le supprime ainsi que tous ses prédécesseurs.



6 n'a pas de successeur ; il appartient au noyau et on le supprime ainsi que tous ses prédécesseurs.

2 n'a pas de successeur ; il appartient au noyau et on le supprime ainsi que tous ses prédécesseurs.

Le noyau est donc $(2, 6, 8)$.

Remarque. Dans le cas du jeu de Chomp les éléments du noyau sont les positions perdantes et les autres sommets les positions gagnantes. La stratégie gagnante consiste, pour chaque sommet qui n'est pas dans le noyau, à jouer un coup qui s'y ramène.

■ Algorithme de calcul du noyau

On considère un graphe acyclique (S, A). Il est représenté en machine par la donnée d'un dictionnaire d dont les clefs sont les sommets et les valeurs les successeurs des clefs (sous forme de liste).
Par exemple, le graphe du jeu de Chomp (2,3) est représenté en mémoire par le dictionnaire :

```
d = {
    0: [1, 2, 3, 4, 5],
    1: [4, 5, 6],
    2: [1, 3, 4, 5],
    3: [5, 6, 7],
    4: [7, 8],
    5: [8],
    6: [5, 7],
    7: [8],
    8: [],
}
```

Exercice 1

a. Rédiger une fonction `sansSucesseur(d)` qui renvoie un sommet sans successeur.

b. Rédiger une fonction `predecesseurs(d, j)` qui renvoie la liste de tous les prédécesseurs du sommet j.

c. Rédiger une fonction `supprimeSommet(d, j)` qui supprime un sommet du graphe.

d. En déduire une fonction `noyau(d)` qui renvoie la liste des sommets constituant le noyau de (S, A).

Remarque. L'algorithme que vous venez d'écrire a une complexité en $O(n^3)$. Sachant que le jeu de Chomp (p,q) possède $n = \binom{p+q}{p} - 1$ sommets, on conçoit que le calcul du noyau se révèle rapidement d'un coût redhibitoire dès lors que p et q deviennent trop grands car pour $p = q$ on a $n \sim \binom{2p}{p} \sim \frac{4^p}{\sqrt{\pi p}}$ donc la complexité croît exponentiellement avec p.

1.3 Fonction de Sprague-Grundy

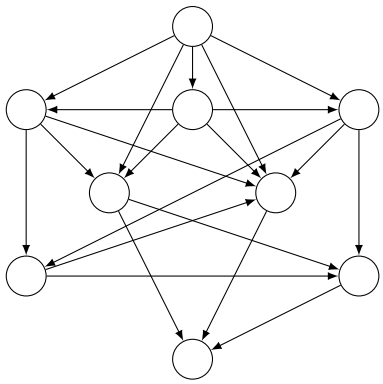
On considère un graphe orienté acyclique (S, A) associé à un jeu impartial dans lequel le perdant est celui qui ne peut plus jouer. On définit le *nimber* $n(s)$ d'un sommet $s \in S$ de la façon suivante :

- si s est une position gagnante (une position sans successeur), $n(s) = 0$;
- sinon, $n(s)$ est le plus petit entier positif ou nul n'apparaissant pas dans la liste des nimbers de ses successeurs.

La fonction n ainsi définie s'appelle aussi la fonction de Sprague-Grundy.

Exercice 2

Reporter sur le graphe ci-dessous (associé au jeu de Chomp (2, 3)) le nimber de chacun de ses sommets :



Théorème 1.3 — *Le noyau du graphe correspond aux sommets s vérifiant $n(s) = 0$.*

Démonstration. Posons $S' = \left\{ s \in S \;\middle|\; n(s) = 0 \right\}$. Par définition S est stable (un sommet de S′ ne peut avoir parmi ses successeurs un autre sommet de nimber nul).
De plus, pour tout sommet $s \in S \setminus S'$, $n(s) > 0$ donc s possède un successeur de nimber nul, autrement dit appartenant à S′. S′ est donc absorbant. □

Exercice 3

On représente un graphe par un dictionnaire d dont les clefs sont les sommets et les valeurs la liste des successeurs de la clef correspondante. Le but de cet exercice est de construire un dictionnaire n dont les clefs sont les sommets et les valeurs les nimbers de ces sommets.

a. Rédiger une fonction `sansNimber(d, n)` qui renvoie un sommet s ne possédant pas encore de nimber mais dont tous les successeurs en possèdent. Cette fonction renverra `None` dans le cas où un tel sommet n'existe pas.

b. En déduire une fonction `nimber(d)` qui renvoie le dictionnaire des nimbers des différents sommets composant ce graphe.

1.4 Le jeu de Wythoff

Ce jeu se joue sur un échiquier sur lequel se déplace une reine. Chacun des deux joueurs la déplace alternativement, mais seuls sont autorisés les mouvements vers la gauche, vers le bas, ou en diagonale en bas à gauche. Le gagnant est celui qui mène la reine dans le coin inférieur gauche.

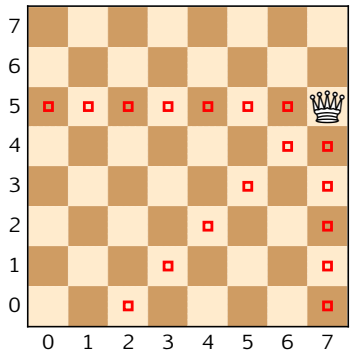


Figure 5 – Les mouvements autorisés dans le jeu de Wythoff.

Il s'agit là encore d'un jeu qui peut être modélisé par un graphe orienté acyclique, les sommets étant les cases de l'échiquier et les arêtes les mouvements autorisés entre deux cases.

Exercice 4

a. Rédiger une fonction `grundy(n, p)` qui prend pour arguments deux entiers n et p et qui renvoie un tableau $n \times p$ contenant les valeurs de la fonction de Sprague-Grundy de chacune de ses cases.

b. En déduire une fonction `wythoff(n, p)` qui prend pour argument une position (n, p) sur l'échiquier et qui, si cette position n'est pas perdante, renvoie la position où jouer pour gagner.

2. Algorithme min-max

Dans le cas d'un jeu à deux joueurs plus complexe, le calcul du noyau n'est pas possible. L'algorithme que nous avons écrit à une complexité en $O(n^3)$, où n est le nombre de sommets du graphe, autrement dit le nombre de positions que l'on peut rencontrer lors d'une partie. Cependant, cet entier n est souvent extrêmement grand : il est estimé de l'ordre de 10^{32} pour les dames, entre 10^{43} et 10^{50} pour le jeu d'échecs, de l'ordre de 10^{100} pour le jeu de go. Il devient donc nécessaire de s'appuyer non plus sur une évaluation exacte de la position, mais sur une estimation de la valeur de la position atteinte.

2.1 Heuristique

Dans la suite de cette section, nous supposerons posséder une fonction h qui à toute position légale p du jeu associe une valeur dans $\mathbb{R}$, de sorte que :

- plus $h(p)$ est grand, meilleure est la position pour Adam ;
- plus $h(p)$ est petit, meilleure est la position pour Ève.

Une telle fonction est appelée une *heuristique*.

■ Puissance 4

Pour illustrer cette section, nous allons prendre l'exemple du Puissance 4 : le but du jeu est d'aligner une suite de quatre pions de même couleur sur une grille comptant six rangées et sept colonnes. Tour à tour, les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans la dite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

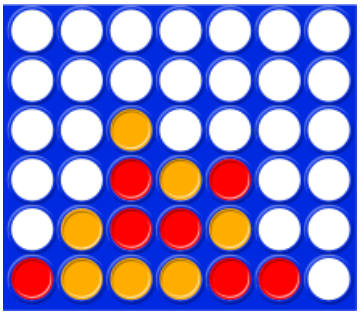


Figure 6 – Une position de puissance 4.

Une heuristique simple consiste à attribuer à chaque case une valeur, par exemple le nombre d'alignements potentiels de quatre pions lorsqu'on place un pion à cet emplacement, puis à sommer les cases occupées (positivement pour les pions d'Adam, négativement pour ceux d'Ève).

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Figure 7 – Les valeurs de chaque case pour l'heuristique utilisée.

Par exemple, si on convient que les pions jaunes sont ceux d'Adam, la valeur de l'heuristique de la position présentée figure 6 est égale à $4+5+7+6+8+13+11-3-5-4-8-10-11-11=2$.

Évidemment, l'heuristique sera égale à $+\infty$ pour une position gagnante pour Adam, et à $-\infty$ pour une position gagnante pour Ève.

2.2 Min-Max

Au moment où l'un des deux protagonistes doit jouer, plusieurs possibilités s'offrent à lui (entre une et sept pour le puissance 4). Une solution simple pour choisir le coup à jouer consiste à calculer l'heuristique correspondant à chacune des configurations atteignables et à jouer celle d'heuristique maximale (pour Adam) ou minimale (pour Ève).

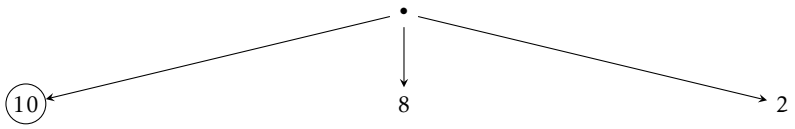


Figure 8 – C'est à Adam de jouer, trois coups sont possibles.

Ainsi, dans l'exemple de la figure 8, Adam choisira le coup le plus à gauche, correspondant à une évaluation égale à 10.
Mais Adam peut aussi tenir compte du coup que va jouer Ève ensuite, et donc calculer l'heuristique de chacune des positions qu'Ève pourra atteindre. Si on observe la figure 9, on constate qu'il vaut mieux pour Adam jouer le coup central, en partant du principe qu'Ève joue au mieux son coup.

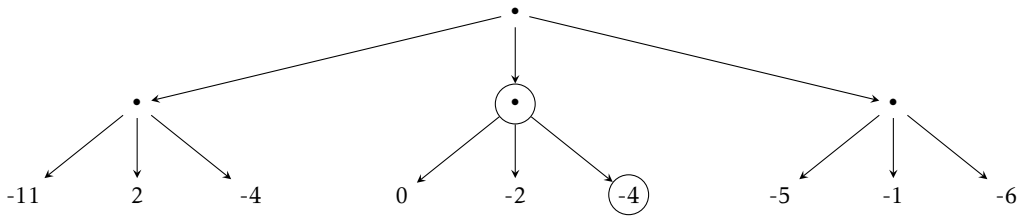


Figure 9 – C'est à Adam de jouer, en tenant compte du coup suivant d'Ève.

Bien évidemment, on peut réitérer ce raisonnement et tenir compte du coup suivant, joué cette fois par Adam. La figure 10 montre qu'en tenant compte des deux coups suivant, Adam a en fait intérêt à jouer le coup le plus à droite.

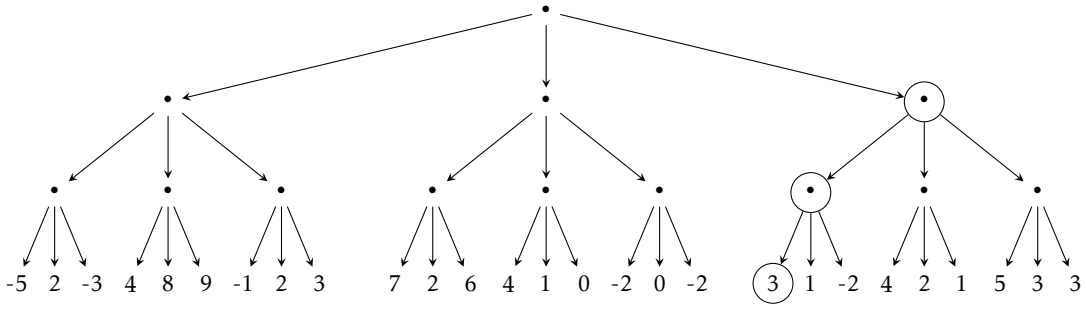


Figure 10 – Le meilleur coup d'Adam, en tenant compte des deux coups suivants.

On peut répéter ce raisonnement, mais le nombre de configurations à examiner ayant tendance à croitre exponentiellement, il est nécessaire de limiter la profondeur de la recherche.

■ L'algorithme min-max

Pour calculer le meilleur coup d'Adam, il faut donc commencer par calculer la valeur de l'heuristique de toutes les positions atteignables en n coups (les feuilles de l'arbre). Si n est pair, Ève aura joué le dernier coup : le père de chacune de ces feuilles se verra donc attribuer le minimum des valeurs de ses fils. À l'inverse, si n est impair le père de chacune de ces feuilles se verra attribuer la valeur maximale de ses fils (car Adam aura joué en dernier). Ainsi, de proche en proche chaque position de l'arbre se verra attribuer une valeur (illustration figure 11).

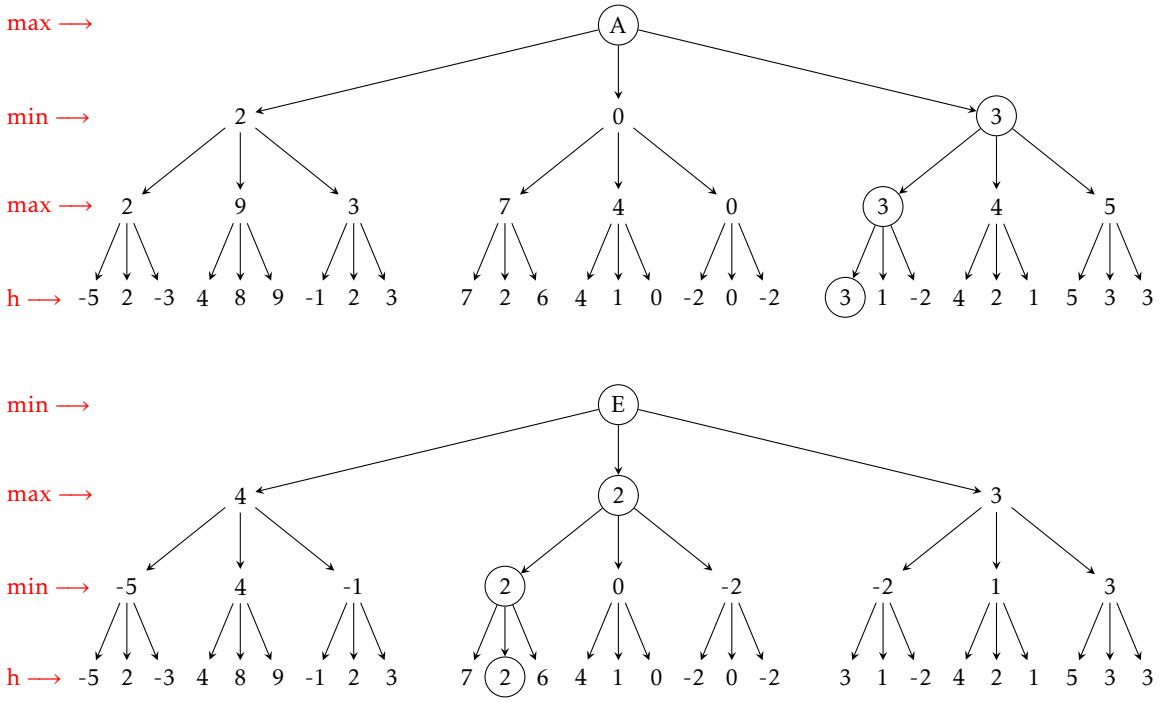


FIGURE 11 – Le résultat de l'algorithme min-max à une profondeur 2 (le premier arbre si c'est à Adam de jouer, le second si c'est à Ève).

Nous allons écrire deux fonctions :

- `maximin(p, n)` (destinée à Adam) va chercher à maximiser l'heuristique après n coups en partant de la position p, en supposant que son adversaire joue au mieux ;
- `minimax(p, n)` (destinée à Ève) va chercher à minimiser l'heuristique après n coups en partant de la position p, en supposant que son adversaire joue au mieux.

Ces deux fonctions sont mutuellement récursives : pour calculer `maximin(p, n)` on calcule pour chaque position $p_1,\dots,p_k$ atteignable à partir de p la valeur de l'heuristique des positions `minimax(pi, n-1)` avant de choisir la position conduisant à la valeur maximale.
De manière symétrique, pour calculer `minimax(p, n)` on calcule pour chaque position $p_1,\dots,p_k$ atteignable à partir de p la valeur de l'heuristique des positions `maximin(pi, n-1)` avant de choisir la position conduisant à la valeur minimale.

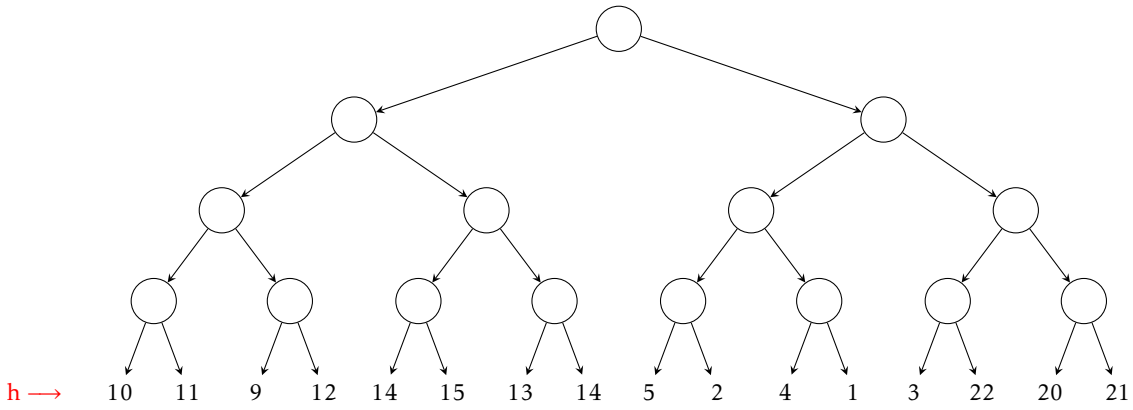
Pour la rédaction de l'algorithme, on suppose définies la fonction `h(p)` qui prend pour argument une position du jeu et renvoie la valeur de son heuristique, ainsi que la fonction `successeurs(p)` qui renvoie la liste des positions atteignables à partir de la position p.

```
def minimax(p, n):
    if n == 0 or successeurs(p) == []:
        return h(p)
    mini = np.inf
    for pk in successeurs(p):
        s = maximin(pk, n - 1)
        if s < mini:
            mini = s
    return mini
```

```
def maximin(p, n):
    if n == 0 or successeurs(p) == []:
        return h(p)
    maxi = -np.inf
    for pk in successeurs(p):
        s = minimax(pk, n - 1)
        if s > maxi:
            maxi = s
    return maxi
```

Exercice 5

Calculer la valeur de la position associée à l'arbre ci-dessous, dans le cas où c'est le joueur qui cherche à maximiser l'heuristique qui doit jouer.

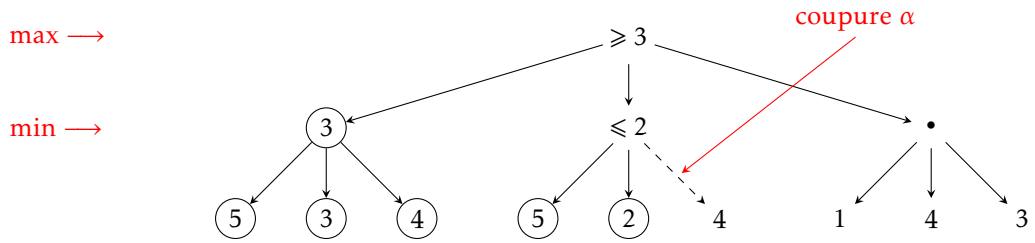


2.3 Élagage alpha-beta (hors programme)

L'élagage alpha-beta est une amélioration de l'algorithme min-max qui vise à ne pas explorer certaines des branches de l'arbre dont on sait qu'elle n'interviendront pas dans l'évaluation de la position courante.

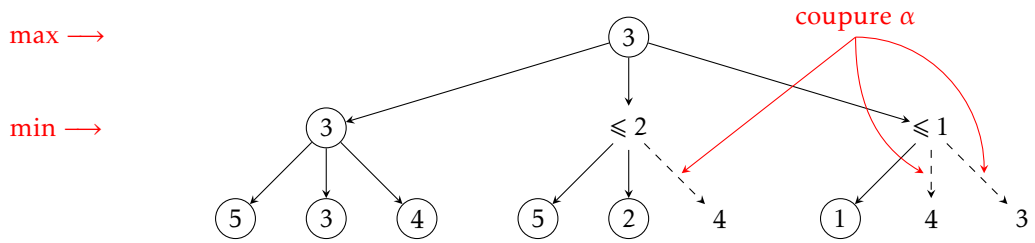
■ Coupure alpha

Considérons l'exemple ci-dessous, qui concerne une étape de calcul de minimum. L'exploration est en cours, les feuilles et nœuds qui ont été examinés sont marqués d'un cercle.



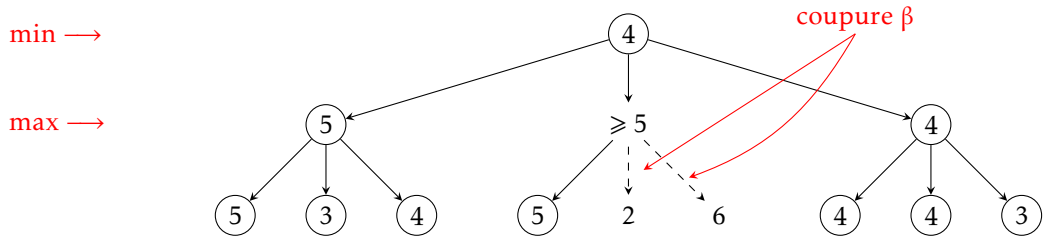
On constate qu'il n'est pas nécessaire de poursuivre l'exploration de la branche centrale : on sait déjà que sa valeur sera inférieure ou égale à 2 donc qu'elle ne sera pas choisie à l'étape suivante qui sera un calcul de maximum.

Une deuxième coupure α se produit après l'exporation du premier fils de la branche de droite : le minimum sera inférieur ou égal à 1 donc ne sera pas pris en compte pour le calcul du maximum de l'étape suivante :



■ Coupure beta

La situation est bien évidemment symétrique dans le cas d'une étape de calcul de maximum, comme l'illustre l'exemple ci-dessous :



Exercice 6

Reprendre l'exemple de l'exercice 5 en indiquant les positions des coupures α et β (on suppose les nœuds explorés de la gauche vers la droite).

■ Mise en œuvre pratique

La mise en œuvre de cette amélioration consiste à ajouter deux arguments (traditionnellement alpha et beta) qui désignent respectivement une borne inférieure et une borne supérieure de la valeur du nœud.

```
1  def minimax(alpha, beta, p, n):
2      if n == 0 or successeurs(p) == []:
3          return h(p)
4      mini = np.inf
5      for pk in successeurs(p):
6          s = maximin(alpha, beta, pk, n-1)
7          if s < mini:
8              mini = s
9          if mini <= alpha:
10             return mini
11         beta = min(beta, mini)
12     return mini
```

```
def maximin(alpha, beta, p, n):
    if n == 0 or successeurs(p) == []:
        return h(p)
    maxi = -np.inf
    for pk in successeurs(p):
        s = minimax(alpha, beta, pk, n-1)
        if s > maxi:
            maxi = s
        if maxi >= beta:
            return maxi
        alpha = max(alpha, maxi)
    return maxi
```

Les lignes 9-10 correspondent à une coupure α, la ligne 11 à une mise à jour du majorant de la valeur du nœud.

Chapitre V
Algorithmes d'apprentissage

Sous le terme d'intelligence artificielle se cachent un ensemble de techniques permettant à des machines d'accomplir des tâches et de résoudre des problèmes normalement réservés aux humains, comme par exemple reconnaître et localiser des objets dans une image. Depuis quelques années, on associe presque toujours l'intelligence aux capacités d'apprentissage : c'est grâce à l'apprentissage qu'un système intelligent capable d'exécuter une tâche peut améliorer ses performances avec l'expérience.
On distingue deux type d'apprentissage : le plus fréquent, *l'apprentissage supervisé*, consiste pour un opérateur à montrer à la machines des milliers voire des millions d'exemples étiquetés avec leur catégorie, qui permettront à la machine de déterminer elle-même les paramètres pertinents pour classer chaque objet dans la catégorie qui lui correspond. Une fois cette phase d'apprentissage terminée, la machine doit être capable de généraliser à des objets pas encore vus.
L'apprentissage non supervisé est plus ambitieux, mais il est aussi plus proche de notre propre modèle d'apprentissage, basé sur l'observation. Il consiste à faire en sorte qu'à partir d'un ensemble de données la machine soit capable de créer ses propres catégories, et si possible que ces catégories soient pertinentes pour nous !

Dans ce chapitre nous allons nous intéresser à deux algorithmes que l'on utilise dans ces deux modes d'apprentissage : l'algorithme des k plus proches voisins, et l'algorithme des k moyennes.

1. Apprentissage supervisé

1.1 Un premier exemple

Nous allons considérer un ensemble de points de coordonnées $(x,y) \in [0,1]^2$, qui peuvent appartenir à deux catégories représentées par des losanges bleus et des disques rouges et nous allons fournir à la machine un jeu de 250 données d'entrainement (voir figure 1).

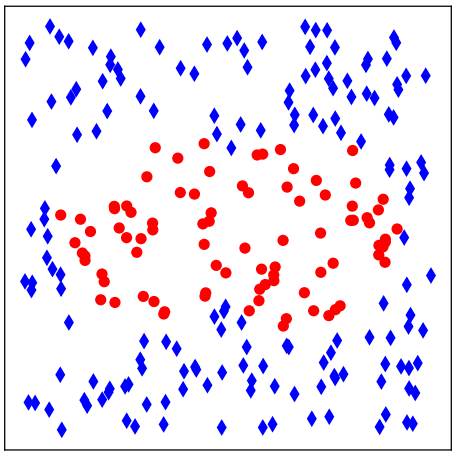


Figure 1 – 250 données d'entrainement.

Une fois ces données acquises nous souhaitons qu'à partir d'un point de coordonnées $(x,y) \in [0,1]^2$, la machine lui attribue une catégorie (bleu ou rouge). La méthode des k plus proches voisins consiste à déterminer les k données d'entrainement les plus proches du point (x,y), et à attribuer à ce point la catégorie majoritaire parmi ces voisins.

Dans l'algorithme qui suit, nous allons supposer que `donnees` est un tableau contenant les données d'apprentissage sous la forme (x,y,c) avec $(x,y) \in [0,1]^2$ et $c =$ *"blue"* ou $c =$ *"red"*.

```
def plusProchesVoisins(m, k):
    t = sorted(donnees, key=lambda d: (d[0] - m[0]) ** 2 + (d[1] - m[1]) ** 2)[:k]
    r = 0
    for d in t:
        if d[2] == "red":
            r += 1
    if 2 * r > k:
        return "red"
    else:
        return "blue"
```

Remarque. La fonction `sorted` possède un argument facultatif `key` qui indique, le cas échéant, une fonction qui est appelée sur chaque élément de la liste avant d'effectuer les comparaisons.
Dans le cas présent, chaque donnée d est triée suivant les valeurs croissantes de $(d_0-m_0)^2+(d_1-m_1)^2$, autrement dit suivant le carré de la distance euclidienne entre les points d et m.

Puisqu'il n'y a ici que deux catégories, si nous prenons un entier k impair nous n'aurons pas à gérer le cas d'égalité.
Avec la fonction `plusProchesVoisins`, nous disposons d'une fonction qui attribue une catégorie à tout point de $[0,1]^2$, en se basant sur la catégorie de ses k plus proches voisins. J'ai représenté figure 2 les catégories attribuées à chacun des points du plan par cet algorithme pour $k=3$ et pour $k=7$.

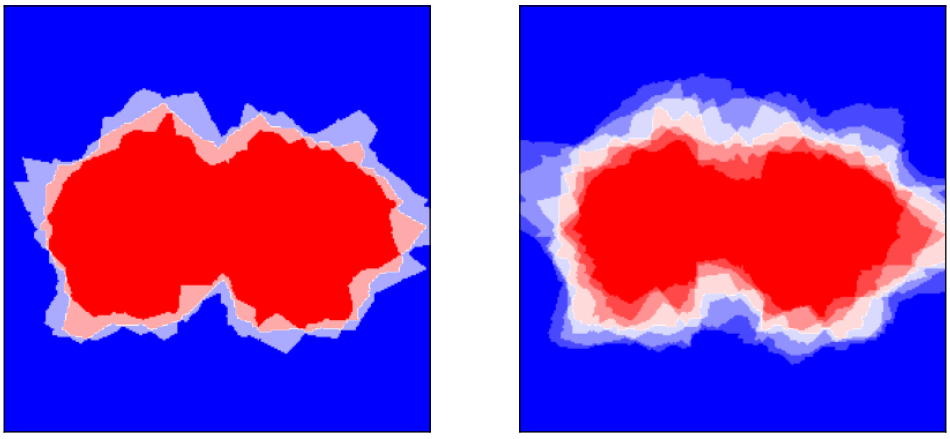


FIGURE 2 – Le résultat de `plusProchesVoisins` pour $k=3$ et $k=7$.

■ Matrice de confusion

Comment interpréter les résultats que nous avons obtenus ?
Dans un problème d'apprentissage automatique, il est d'usage de séparer les données en deux sous-ensembles : les données réservées à l'apprentissage et les données destinées à tester la qualité de l'apprentissage. Nous allons donc supposer que l'on dispose d'un tableau `test` contenant 100 autres données, toujours sous la forme (x,y,c) avec $(x,y)\in[0,1]^2$ et $c=$ *"blue"* ou $c=$ *"red"*, et nous allons appliquer la méthode du plus proche voisin à chacun de ces points. Si notre objectif est de reconnaitre si un point est rouge ou pas, les points tests vont se répartir en quatre catégories :

- les vrais positifs (les points rouges reconnus comme tels) ;
- les vrais négatifs (les points bleus reconnus comme tels) ;
- les faux positifs (les points bleus reconnus à tort comme rouges) ;
- les faux négatifs (les points rouges reconnus à tort comme bleus).

Ces quatre valeurs sont rangées dans une matrice de la forme $C=\left(\begin{array}{c|c}\text{VP} & \text{FN}\\ \hline \text{FP} & \text{VN}\end{array}\right)$ appelée *matrice de confusion*.

Voici par exemple les matrices que l'on obtient avec différentes valeurs de k, pour le même jeu de données et de test :

k	1	3	7	17	31
C	$\begin{pmatrix} 27 & 4 \\ 3 & 66 \end{pmatrix}$	$\begin{pmatrix} 30 & 1 \\ 3 & 66 \end{pmatrix}$	$\begin{pmatrix} 28 & 3 \\ 4 & 65 \end{pmatrix}$	$\begin{pmatrix} 27 & 4 \\ 4 & 65 \end{pmatrix}$	$\begin{pmatrix} 28 & 3 \\ 10 & 59 \end{pmatrix}$

La matrice de confusion peut aider à choisir la meilleure valeur de k ; ici par exemple on choisira plutôt $k = 3$.

1.2 Reconnaissance de caractères

Dans la réalité on utilise pas la totalité des échantillons pour l'apprentissage : une partie est préservée pour servir à tester la qualité de la reconnaissance. Dans l'exemple qui suit, nous disposons de 1 797 images de 8×8 pixels en niveau de gris représentant des chiffres de 0 à 9 (un extrait de la base de données est présenté figure 3).

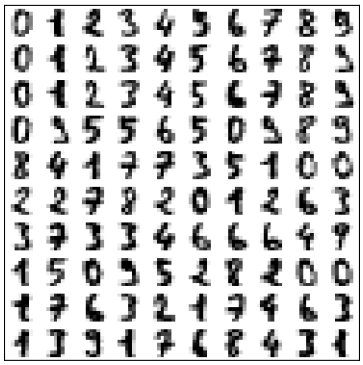


Figure 3 – Un extrait de la banque de données.

Plus précisément, ces données nous sont fournies par l'intermédiaire de deux tableaux :

- `X` est une matrice de taille $1\,797 \times 64$; pour tout $k \in [\![0, 1796]\!]$, `X[k]` est un vecteur de $\mathbb{R}^{64}$ qui représente l'image digitalisée d'un chiffre ;
- `Y` est un vecteur de taille 1 797 ; pour tout $k \in [\![0, 1796]\!]$, `Y[k]` est le label de `X[k]`, autrement dit l'entier compris entre 0 et 9 qui est représenté par `X[k]`.

Nous allons partager ces données en deux parties sensiblement égales ; la première consacrée à l'apprentissage, la seconde au test. Dans la partie consacrée à l'apprentissage, on va choisir 90 représentants de chacune de nos dix classes, il restera donc 897 données pour nous servir de test.

```
donnees = set()
test = set()
nb = [0] * 10
for i in range(1797):
    if nb[Y[i]] < 90:
        donnees.add(i)
        nb[Y[i]] += 1
    else:
        test.add(i)
```

Définissons la distance entre les images i et j :

```
def dist(X, Y):
    s = 0
    for i in range(len(X)):
        s += (X[i] - Y[i]) ** 2
    return s
```

Écrivons maintenant le fonction `plusProchesVoisins` correspondante. Cette fonction renvoie la ou les labels les plus présents parmi les k voisins (sous la forme d'une liste).

```
def plusProchesVoisins(i, k):
    t = sorted(donnees, key=lambda j: dist(X[i], X[j]))[:k]
    d = [Y[j] for j in t]
    m, sol = 0, []
    for c in range(10):
        if d.count(c) > m:
            m, sol = d.count(c), [c]
        elif d.count(c) == m:
            sol.append(c)
    return sol
```

Appliqué à une donnée de test, trois cas de figure sont possibles :

- la fonction `plusProchesVoisins` renvoie une seule valeur, exacte ;
- la fonction `plusProchesVoisins` renvoie une seule valeur, inexacte (la machine se trompe) ;
- la fonction `plusProchesVoisins` renvoie plusieurs valeurs (la machine est indécise).

L'expérience montre que pour $k = 3$, sur les 897 valeurs de test :

- la machine reconnait le bon caractère 858 fois (soit un taux de réussite de plus de 95%) ;
- la machine se trompe 31 fois ;
- la machine est indécise 8 fois.

Les erreurs que la machine à commise sont présentées figure 4.

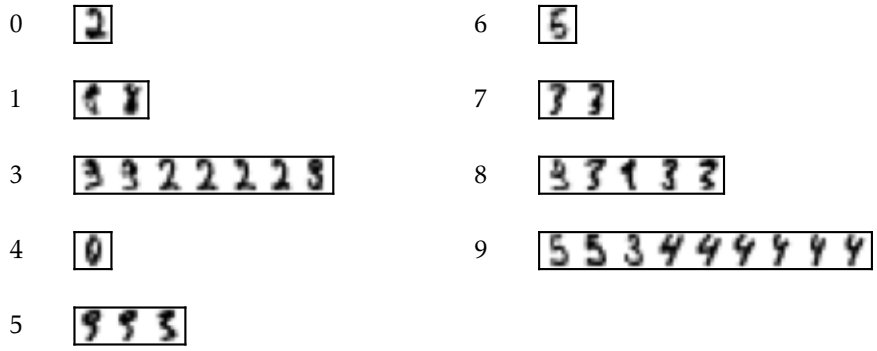


Figure 4 – Les erreurs commises par la machine.

■ Quelles améliorations envisager ?

La méthode que je viens de vous présenter est bien entendu simpliste, mais peut être améliorée de plusieurs façons, par exemple :

- au lieu de considérer les k plus proches voisins, considérer tous les voisins contenus dans une boule de rayon ϵ centrée autour de l'objet à classer ;
- pondérer la contribution de chaque voisin par un coefficient inversement proportionnel à la distance à l'objet à classer ;
- ou encore utiliser d'autres distances que la distance euclidienne.

2. Apprentissage non supervisé

2.1 Partitionnement des données

Nous allons maintenant nous intéresser à un problème plus complexe : comment comprendre la structure des données sans aider la machine en lui donnant des données d'apprentissage étiquetées ?
En pratique il s'agit pour la machine de regrouper les données proches au sein d'une même classe, à charge pour l'humain d'interpréter ensuite ce regroupement.

Par exemple, en regardant la figure 5 nous voyons tous trois classes de points. Nous devons faire en sorte qu'il en soit de même de la machine. Nous allons quand même l'aider un peu en lui imposant le nombre k de classes.

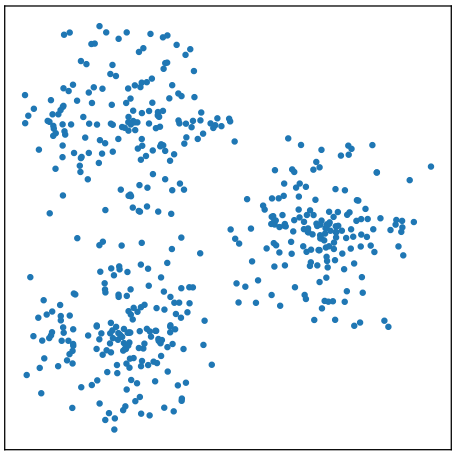


Figure 5 – Les données semblent se regrouper en trois classes.

En appelant $C_1,\dots,C_k$ ces classes on note

- μ_j le *barycentre* de C_j, autrement dit $\mu_j = \dfrac{1}{\operatorname{card} C_j}\displaystyle\sum_{x\in C_j} x$;
- m_j le *moment d'inertie* de C_j, soit $m_j = \displaystyle\sum_{x\in C_j} \|x-\mu_j\|^2$.

L'objectif de la machine va être de minimiser la somme des moments d'inertie $\displaystyle\sum_{j=1}^{k} m_j$.

■ L'algorithme des k moyennes

Le calcul de la classification optimale, c'est-à-dire minimisant la somme des moments d'inertie, est un problème très coûteux en temps, aussi allons nous employer un algorithme glouton. Ce dernier nous assurera d'obtenir *in fine* une « bonne » solution, mais pas forcément la meilleure. *L'algorithme des k moyennes* consiste à réaliser la succession d'opérations suivantes :

(1) on choisit aléatoirement k centres $\mu_1,\dots,\mu_k$;
(2) chacun des points du nuage est associé au centre μ_j le plus proche ; on crée ainsi k classes $C_1,\dots,C_k$;
(3) on calcule les barycentres $\mu_1,\dots,\mu_k$ de ces classes, qui remplacent les valeurs précédentes ;
(4) on reprend le calcul à partir de l'étape (2).

Nouas admettrons que cet algorithme converge, autrement dit qu'à partir d'une certaine étape les centres $\mu_1,\dots,\mu_k$ ne se déplacent plus et donc que les classes $C_1,\dots,C_k$ sont stabilisées.
Mais répétons-le, cet algorithme donne une configuration pour laquelle la somme des moments d'inertie est un minimum local, mais pas forcément le minimum global.

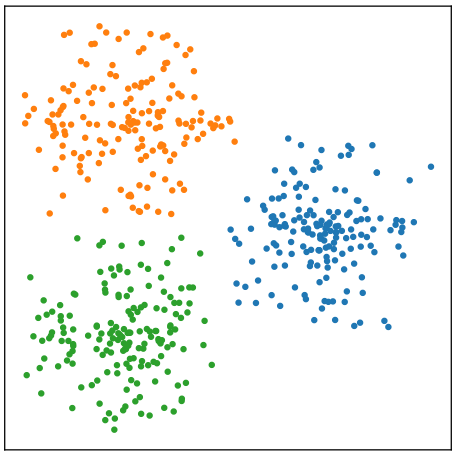


FIGURE 6 – Le résultat de l'algorithme des k moyennes pour $k = 3$.

2.2 Reconnaissance de caractères

Considérons de nouveau nos 1 797 images de chiffres de 0 à 9, mais cette fois-ci sans prendre en compte leur label. Nous allons appliquer l'algorithme des k moyennes avec $k = 10$, de manière à obtenir un regroupement de ces images en 10 classes, qu'on espère correspondre aux différents chiffres représentés. Rappelons que ces images sont regroupées dans une liste `X` , chaque image `X[i]` étant représentée par un vecteur de $\mathbb{R}^{64}$.

Nous commençons par écrire une fonction qui calcule le barycentre d'un ensemble d'images. Ce dernier est un vecteur de $\mathbb{R}^{64}$; il représente lui aussi une image 8×8.

```
def barycentre(s):
    return sum([X[j] for j in s]) / len(s)
```

Passons maintenant à l'algorithme des k moyennes :

```
1  def kmoyennes(X, k):
2      mu = np.array([X[rd.randint(len(X))] for _ in range(k)])
3      while True:
4          s = [set() for _ in range(k)]
5          for i in range(len(X)):
6              dmin = np.inf
7              for j in range(k):
8                  if dist(X[i], mu[j]) < dmin:
9                      jmin, dmin = j, dist(X[i], mu[j])
10             s[jmin].add(i)
11         newmu = np.array([barycentre(s[j]) for j in range(k)])
12         if (newmu == mu).all():
13             break
14         mu = newmu
15     return s
```

La ligne 2 choisit k images initiales. Ensuite (ligne 5) pour chaque image X_i on calcule la valeur de μ_j la plus proche de X_i (lignes 6-9) pour ranger ce dernier dans la classe associée (ligne 10). On recalcule les différents barycentres (ligne 11) et on s'arrête lorsque ces derniers n'ont pas été modifiés (lignes 12-13).
Voici les images correspondants aux barycentres des dix classes que l'on obtient par cet algorithme pour $k = 10$:

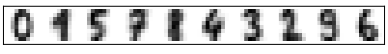


FIGURE 7 – Les barycentres des 10 classes obtenues.

Manifestement, les classes obtenues correspondent bien à notre souhait. Si on passe maintenant en revue toutes les images de chacune de ces dix classes en tenant compte de leur label, on obtient 1 426 succès et 371 échecs. Autrement dit, 79 % des images ont étés rangées dans les bonnes classes. Plus précisément, on peut pour chacune des 10 classes calculer le pourcentage de réussite :

classe	0	1	5	7	8	4	3	2	9	6
réussite	99 %	60 %	91 %	85 %	45 %	98 %	87 %	85 %	56 %	97%

Le plus mauvais score est obtenu pour la cinquième classe qui ne comporte que 45 % d'images représentant le chiffre 8 ; à l'inverse, la première classe comporte 99 % d'images représentant le chiffre 0.

2.3 Application à la compression de données

Considérons l'image représentée figure 8. Ses dimensions sont de 415×626 pixels. Chaque pixel est un triplet

Figure 8 – L'image initiale et sa réduction à 16 couleurs.

(r, g, b) où r, g et b sont des flottants de l'intervalle $[0, 1[$ codés sur 32 bits représentant la quantité de rouge, vert et bleu du pixel. L'image est représentée en machine par un tableau numpy `img` :

```
In [1]: img.shape
Out[1]: (415, 626, 3)
```

Cette image initiale comporte 121 915 couleurs différentes. Notre objectif est de réduire ce nombre à seulement 16 couleurs. Pour ce faire, on applique l'algorithme des k moyennes pour regrouper les différents pixels en 16 classes de couleurs voisines. Pour ce faire, il faut définir une distance entre deux couleurs c et c', mais puisque c et c' sont représentées par un triplet de $[0, 1]^3$, on peut utiliser une distance euclidienne :

$$\text{si } c = (r, v, b) \text{ et } c' = (r', v', b') \text{ alors } \mathrm{d}(c, c') = \sqrt{(r - r')^2 + (v - v')^2 + (b - b')^2}.$$

Une fois le regroupement en 16 classes effectué, on calcule la couleur moyenne de chacune de ces 16 classes, puis on attribue cette valeur moyenne à chacun des pixels de la classe correspondante. Le résultat est une nouvelle image, présentée à droite figure 8.
Enfin, la figure 9 présente deux réductions à 16 couleurs, la première avec la norme euclidienne, la seconde avec la norme infinie.

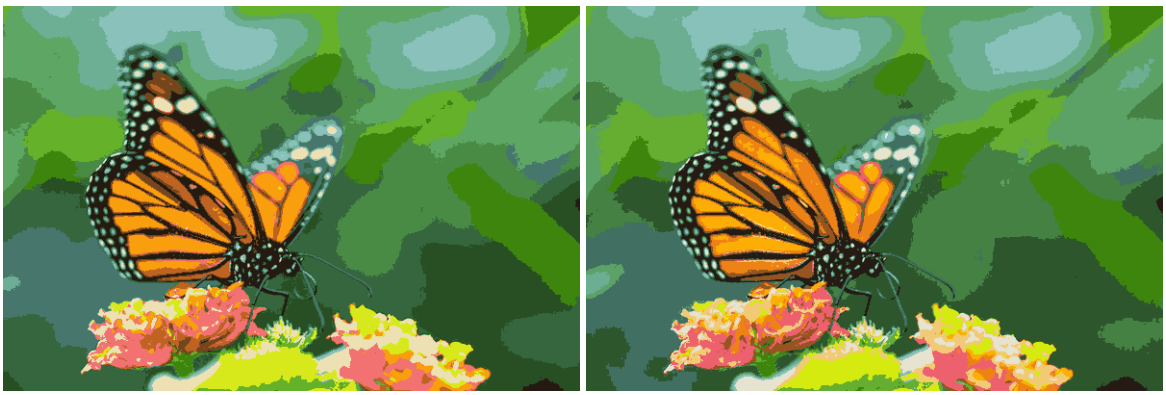


Figure 9 – les réductions obtenues avec la norme euclidienne et la norme infinie.