Chapitre 1

Rappels : listes, dictionnaires, fichiers

I. Les listes en python

A. Introduction

Rappel : en python, il existe différentes classes d'objets : `int, float, complex, boolean, string, list, dict, ...`

Définitions :

- Une liste est une structure de donnée séquentielle.
- L'ensemble des listes en python est représenté par la classe `list`.
- Une liste se définit par la donnée explicite de ses éléments entre crochets :

```
> > L=[1 , 1. , 3.4 , 1j , 1+2j , True, ''Pothier'' , [1,4,''MP1''] ]
```

Ses éléments peuvent être de n'importe quel type.

Propriétés :

1. Une liste peut être définie « par compréhension », comme on le ferait pour un ensemble en mathématiques :

```
> > n=10; Lx=[i/10 for i in range(n)]
> > Ly=[f(x) for x in Lx]
```

   où f est une fonction définie préalablement.
2. La longueur (*length* en anglais) d'une liste `L` est donnée par l'instruction `len(L)`.
3. Pour accéder au contenu de la case numéro i, on écrit `L[i]`.
   ATTENTION : en python, la première case a toujours le numéro 0 !
4. Le contenu de la dernière case de la liste `L` est donc accessible via l'instruction `L[len(L)-1]`.
   On peut également accéder au contenu de la dernière case via l'instruction `L[-1]`, à celui de l'avant-dernière case via l'instruction `L[-2]`,...Ainsi Python numérote les cases de droite à gauche à partir de -1 et en retranchant 1 à chaque case.

5. Une liste est un objet modifiable : si on reprend la liste `L` définie ci-dessus et que l'on rajoute l'instruction `L[0]=[True,False]`, la liste `L` est modifiée en la nouvelle liste

   ```
   > > L=[ [True,False] , 1. , 3.4 , 1j , 1+2j , True, ''Pothier'' , [1,4,''MP1''] ]
   ```

   Remarquons qu'au contraire, une chaîne de caractères n'est pas modifiable...
6. Opérations + et $*$.
   (a) Pour deux listes ℓ et ℓ_2, l'opération $\ell + \ell_2$ effectue la concaténation de ces deux listes.
   (b) Si $a \in \mathbb{N}$, $a * \ell$ (ou bien $\ell * a$) effectue la duplication n fois de la liste ℓ (et renvoie la liste vide si $n = 0$).

B. Opérations plus avancées sur les listes

Extraction de tranche (slicing)

Soit ℓ une liste. Alors l'instruction ℓ`[i:j:k]`, où i, j, k sont a priori trois entiers *relatifs*, donne la sous-liste de ℓ constituée des éléments tels que :
• i est l'indice (inclus) de début dans ℓ ;
  j est l'indice (exclu) de fin dans ℓ ;
  k est le pas.
• si i n'est pas précisé, il vaut par défaut 0 ;
  si j n'est pas précisé, il vaut par défaut `len(`ℓ`)` ;
  si k n'est pas précisé, il vaut par défaut 1.

Exemple : si `L=[6,9,2,5,8,0,1,7,3,10,19]`, que donnent :
`L[2:3:4]`, `L[2:5]`, `L[3:]`, `L[::5]`, `L[-1:-5:-2]`, `L[-2:4:-1]`, `L[-2:5:1]` ?

Remarque : l'extraction de tranche fonctionne de la même façon pour les chaînes de caractères.

Copie d'une liste

ATTENTION : une liste en Python n'est en fait qu'un pointeur vers la structure de donnée présente en mémoire.

Exemple : soit ℓ une liste. Si on écrit :

```
> > ℓ2 = ℓ,
```

ℓ_2 n'est pas une liste indépendante, mais pointe vers le même objet en mémoire qui est la liste ℓ, comme on peut le vérifier avec la ligne de commande :

```
> > id(ℓ) = = id(ℓ2)
True
```

Pour obtenir une copie indépendante de la liste ℓ, plusieurs possibilités (au choix) :

```
> > ℓ2 = [e for e in ℓ ]
> > ℓ2 = ℓ+[]
> > ℓ2 = ℓ[:] ou ℓ2 = ℓ[::]   # ne marche pas pour les tableaux numpy !
> > ℓ2=list(ℓ)
> > ℓ2 = copy.deepcopy(ℓ)   # avec la bibliothèque copy, pour les sous-listes
```

Les méthodes de la classe liste

Il existe en Python une façon efficace de faire des opérations sur les listes, en utilisant une **méthode**. La syntaxe est la suivante pour la liste `L` :

```
> > L.methode(paramètres éventuels)
```

Exemples : parmi les quatre méthodes qui suivent sur les listes, seules les deux premières sont au programme :

- `L.append(x)` pour ajouter l'élément x à la fin de la liste L.
- `L.pop()` pour retirer le dernier élément de la liste L.
- `L.sort()` pour trier une liste d'éléments ordonnés par ordre croissant.
- `L.reverse()` pour inverser l'ordre des éléments dans la liste L.

Exercice : 1) Donner le résultat renvoyé par la suite d'instructions

```
> > L=['a','E','?','b',';','1','A']
> > L.sort()
```

2) Même question pour

```
> > L=[[]]*10
> > L[2].append(57)
```

Conversion liste ⟷ chaîne de caractères (TIPE)

L'expérience montre que ces conversions sont parfois bien utiles (notamment en TIPE, où il arrive qu'on ait besoin de récupérer des données sous forme d'un fichier texte pour les transformer en données numériques).

- Conversion d'un nombre au format texte en un nombre « numérique ».

La syntaxe est la suivante pour les entiers :

```
> > texte='15'
> > int(texte)
15
```

et pour les flottants :

```
> > texte='15.1488'
> > float(texte)
15.1488
```

- Conversion d'une chaîne de caractères en liste avec la méthode `split`.

La syntaxe est la suivante :

```
> > chaine=''MP1:Lycée:Pothier''
> > chaine.split(":")
['MP1','Lycée','Pothier']
```

- Conversion d'une liste de chaînes de caractères en chaîne de caractères avec la méthode `join`.

La syntaxe est la suivante :

```
> > liste=['MP1','Lycée','Pothier']
> > ":".join(liste)
'MP1:Lycée:Pothier'
```

Enfin, n'oubliez pas que l'instruction `help(···)` peut vous aider à vous rafraîchir la mémoire si vous connaissez le nom d'une fonction (ou d'une méthode) mais si vous ne savez plus quelle est son rôle ou sa syntaxe...

C. Complexité des opérations usuelles sur les listes

Soit L une liste de longueur n. Alors :
• l'accès à un élément de la liste (`L[i]`) est de complexité $O(1)$.
• l'accès à la longueur de la liste (`len(L)`) est aussi de complexité $O(1)$.
• la suppression (avec la méthode `.pop`) ou l'ajout (avec la méthode `.append`) d'un élément a une complexité dans le pire des cas en $O(n)$.
• l'ajout de n éléments (en utilisant n fois la méthode `.append`) a également une complexité dans le pire des cas en $O(n)$.

II. Les dictionnaires en python

Définition II..1

un dictionnaire (de type `dict` en python) présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments par le biais d'un indice, on y accède par le biais d'une *clé* . On crée un dictionnaire en suivant la syntaxe `{ c1:v1,..., cn:vn}` où $c_1, ..., c_n$ sont des clés, nécessairement deux à deux distinctes, et $v_1, ..., v_n$ les valeurs qui leur sont associées.

Exemples :

1. L'instruction `dico={ }` crée un dictionnaire vide, de nom `dico` .

2. L'instruction

   ```
   enseignants_MP1 ={ "Anglais" : "Gauthier", "SII" : "Bézelgues", "Maths" : "Blache",
   "Français" : "Colrat", "Informatique" : "Fromenteau", "Physique" : "Muller", }
   ```

   définit un dictionnaire, et l'instruction `enseignants_MP1["Maths"]` renvoie alors `'Blache'` .

3. Il est possible de simuler un ensemble à l'aide d'un dictionnaire, en faisant correspondre à chaque élément la valeur `None` . Par exemple, l'ensemble d'entiers $\{1; 3; 5; 7; 9\}$ sera représenté par le dictionnaire `E = { 1:None , 3:None, 5:None, 7:None, 9:None }`.

Propriétés II..2

Si d est un dictionnaire et c une clé, alors :
• L'expression `c in d` renvoie[a] un booléen indiquant si la clé c est présente dans le dictionnaire `d` .
• L'expression `d[c]` renvoie la valeur associée à la clé c si cette dernière est présente dans le dictionnaire.
• L'instruction `d[c]=v` crée une nouvelle association si la clé c n'est pas présente dans le dictionnaire d, et modifie l'association antérieure sinon.
• L'instruction `del d[c]` supprime l'entrée associée à la clé c dans le dictionnaire d.
• Le nombre de clés présentes dans le dictionnaire d est accessible via l'instruction `len(d)` .

a. en temps constant, ce qui est meilleur que pour une liste.

Il est possible d'itérer sur les clés d'un dictionnaire avec la construction `for k in d.keys()`, mais on peut aussi directement itérer sur les clés et les valeurs avec la construction `for k,v in d.items()`. Par exemple, les deux lignes

```
for k,v in enseignants_MP1.items():
      print("Le cours de",k,"est enseigné par M. ou Mme",v)
```

renvoie

```
Le cours de Anglais est enseigné par M. ou Mme Gauthier
Le cours de SII est enseigné par M. ou Mme Bézelgues
Le cours de Maths est enseigné par M. ou Mme Blache
Le cours de Français est enseigné par M. ou Mme Colrat
Le cours de Informatique est enseigné par M. ou Mme Fromenteau
Le cours de Physique est enseigné par M. ou Mme Muller
```

Remarque : ces deux méthodes sont celles préconisées par le programme de classes préparatoires, mais il est plus naturel d'utiliser directement la construction `for k in d` qui a le même effet que la construction `for k in d.keys()`. Le programme précédent s'écrit alors de façon équivalente :

```
for k in enseignants_MP1:
      print("Le cours de",k,"est enseigné par M. ou Mme", enseignants_MP1[k])
```

III. Quelques opérations usuelles sur les fichiers (TIPE)

En TIPE, il peut être intéressant de sauvegarder des données dans un fichier annexe, par exemple les coordonnées des points d'un graphique obtenu après un temps de calcul assez long. Dans ce but, on va revoir comment manipuler des fichiers.

A. Notions de base

Instructions relatives à la structure arborescente

Il est bon de connaître les instructions suivantes (tapées dans la console) :

- Les instructions suivantes donnent l'adresse du répertoire courant :
```
> > import os
> > os.getcwd()
```
- `ls` permet d'obtenir le contenu du répertoire courant.
- `cd nom_repertoire` permet de se placer dans le sous-répertoire `nom_repertoire` du répertoire courant.
- `cd ..` (attention à l'espace entre cd et ..) permet de remonter d'un niveau dans l'arborescence des répertoires, à partir du répertoire courant.

Chemin relatif et chemin absolu

La position d'un fichier dans l'arborescence se détermine à l'aide d'un chemin. L'origine de ce chemin peut être ou bien la racine, ou bien le répertoire courant. La description d'un chemin se fait à l'aide de la suite des sous-répertoires qui permettent d'accéder au fichier considéré, en commençant par :

- la racine pour un chemin absolu ;
- le répertoire courant pour un chemin relatif.

Exemple : supposons que le chemin absolu du fichier `donnees.csv` est donnée par :
`/Users/fabrice_blache/Boulot/CPGE/MP1/Informatique/data/donnees.csv`
Si on se trouve dans le répertoire MP1, le chemin relatif de ce fichier est donné par :
`/Informatique/data/donnees.csv`

B. Lecture et écriture dans un fichier

- La fonction `open` permet d'ouvrir un fichier avec Python. Elle prend en paramètres le chemin (absolu ou relatif) menant au fichier à ouvrir ainsi que le mode d'ouverture (”r” pour une ouverture en lecture ou bien ”w” pour une ouverture en écriture ou bien ”a” pour une ouverture en écriture en mode ajout).
- La méthode `read` permet de lire l'intégralité du fichier.
- La méthode `write` permet d'écrire une chaîne de caractères dans un fichier.
- La méthode `close` permet de fermer le fichier (vivement conseillé quand on a récupéré les données voulues).

Remarques :

1. Avec la fonction `open`, le résultat obtenu est une chaîne de caractères. En particulier, la méthode `split` (vue en I.B.4.) peut être utilisée pour convertir cette chaîne en liste.
2. En mode ”w” ou ”a”, si le fichier n'existe pas, il est automatiquement créé.

Exemple 1 (ouverture en mode lecture et lecture) :
Supposons que l'on dispose d'un fichier texte `fichier.txt` dans le répertoire courant.
```
> > mon_fichier=open("fichier.txt","r")
> > contenu=mon_fichier.read()
> > print(contenu)
> > mon_fichier.close()
```

Exemple 2 (ouverture en mode écriture et écriture de données) :

```
> > mon_fichier=open("fichier.txt","w")
> > mon_fichier.write("Une autre ligne de données")
> > mon_fichier.close()
```

↯ L'ouverture en écriture ("w") d'un fichier écrase son contenu !

Exemple 3 (ouverture en mode écriture (ajout) et écriture de données) :

```
> > mon_fichier=open("fichier.txt","a")
> > mon_fichier.write("Une autre ligne de données")
> > mon_fichier.close()
```

Exemple : création d'un fichier de données au format csv (exploitable par un tableur).

Chapitre 2

Rappels : récursivité

I. Pile

A. Définitions

Définition : Une pile en informatique (stack en anglais) est une structure de donnée séquentielle basée sur le principe « dernier arrivé, premier sorti » (last in, first out, abrégé en LIFO en anglais).

Remarques :
• une pile est une structure de donnée séquentielle, au même titre que les listes. Elle est semblable à une pile d'assiettes, où les seules opérations possibles sont :
⋆ ajouter une assiette (une donnée) au sommet de la pile ;
⋆ retirer l'assiette (la donnée) du sommet de la pile.

• il existe aussi la notion de file (queue en anglais, FIFO).
• la pile est une structure de donnée plus simple et plus restrictive qu'un tableau.
• les microprocesseurs disposent nativement d'une pile (de capacité limitée) parmi leurs registres. Peut-être avez-vous déjà rencontré une erreur de type « stack overflow »...

Exemples dans la vie courante : pile ou file ?
• distribution d'eau à l'aide de gobelets ;
• retour à la page précédente d'un navigateur ;
• onglet « undo » des éditeurs de texte ;
• serveur d'impression.

Opérations caractéristiques pour une pile (complexité en $O(1)$)

On les appelle aussi primitives. Ce sont :
• empiler un élément (push en anglais) ;
• dépiler un élément (pop en anglais) ;
• tester si la pile est vide (auquel cas on ne peut pas dépiler) ;
• lire l'élément au sommet de la pile (qui s'appelle peek en anglais).

Implémentation en mémoire

- Pour implémenter une pile, les données en mémoire n'ont pas à être contiguës. L'objet pile pointe vers l'adresse du sommet, ce dernier contient la donnée stockée ainsi que l'adresse mémoire de l'élément suivant, et ainsi de suite.

- Pour pouvoir obtenir la taille de la pile en $O(1)$, celle-ci doit aussi être stockée (on dit encapsulée) dans l'objet pile. Autrement on ne l'obtient qu'en temps linéaire O(taille).

- Une pile est moins coûteuse (en espace mémoire) à implémenter qu'un tableau, car il n'est plus nécessaire de réserver de l'espace mémoire contigu. A chaque empilement, il suffit de réserver de l'espace mémoire pour le seul couple (donnée, adresse) et cela n'importe où dans la mémoire disponible.

- La contrepartie est dans l'impossibilité d'effectuer certaines opérations que l'on pouvait faire sur un tableau (ou une liste), par exemple accéder directement à des données (comme dans un répertoire téléphonique).

II. Récursivité

Rappel : une fonction en Python peut appeler toute autre fonction qui lui est visible, c'est-à-dire dont la définition dans le code la précède (au sens large) ; en particulier elle peut s'appeler elle-même.

A. Notion de récursivité

Définition : Une fonction (en Python ou dans un autre langage de programmation) récursive est une fonction qui s'appelle elle-même.

Exemple : le cas le plus simple d'utilisation de la récursivité est d'appeler une fonction f pour un entier $n \in \mathbb{N}^*$ (donc un appel de la forme $f(n)$) qui appelle le calcul de $f(n-1)$,..., jusqu'à donner la valeur initiale $f(0)$.
Le calcul de la factorielle correspond exactement à cette situation :

```
def factorielle(n):
       if n==0:
            return 1
       else:
            return n*factorielle(n-1)
```

B. Pile d'exécution

Les appels successifs d'une fonction récursive sont stockés dans une pile d'exécution.
Par exemple pour le calcul de 4! :

On empile (« phase de descente ») :

$5) factorielle(0) = 1$
$4) factorielle(1) = 1 * factorielle(0)$
$3) factorielle(2) = 2 * factorielle(1)$
$2)\ factorielle(3) = 3 * factorielle(2)$
$1)\ factorielle(4) = 4 * factorielle(3)$

PILE d'exécution

Puis on dépile (« phase de montée ») :

$5) factorielle(0) = 1$
$6) factorielle(1) = 1 * 1 = 1$
$7) factorielle(2) = 2 * 1 = 2$
$8)\ factorielle(3) = 3 * 2 = 6$
$9)\ factorielle(4) = 4 * 6 = 24$

PILE d'exécution

et le résultat retourné est 24. Il est aisé de voir que la complexité pour la fonction récursive factorielle est linéaire en temps et en espace mémoire.

Exemple : calculer à l'aide d'une fonction récursive le n-ème terme de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$, définie par $F_0 = 0$, $F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_n + F_{n+1}$.

C. Avantages et inconvénients des fonctions récursives

⇝ Avantages. L'écriture reflète le mode de pensée par récurrence, elle est concise et élégante (on peut noter qu'une fonction récursive peut toujours s'écrire avec des boucles, mais cela peut être extrêmement compliqué à programmer). On va voir qu'on peut résoudre des problèmes compliqués par des algorithmes récursifs très simples.

⇝ Inconvénients.
• Complexité en espace.
Du fait de la pile d'exécution, le nombre d'appels récursifs en Python est a priori limité environ à 1000 ; on peut l'augmenter avec l'instruction `setrecursionlimit`, mais cela peut faire planter le programme....
• Complexité en temps.
Elle peut très vite devenir gigantesque (exemple de la suite de Fibonacci). Pour être efficace, on voit qu'il faudrait stocker en mémoire les valeurs de F_n au fur et à mesure des calculs ; le terme informatique pour cela est la *mémoïsation* (cela peut se faire en Python à l'aide d'un dictionnaire, voir la suite du cours).
• Il faut être prudent dans la programmation d'une fonction récursive : on peut avoir des appels récursifs infinis, par exemple si on essaie de faire l'appel `factorielle(-1)`...il est donc fondamental

de se demander si l'algorithme se termine, et donc de prévoir quel test d'arrêt (cas de base) ou quel test d'erreur on va mettre dans la fonction récursive.

D. Correction (ou validité) et terminaison d'une fonction récursive

⇝ Terminaison et correction d'un algorithme.

Si l'algorithme récursif ne contient pas de cas de base, il y aura un « problème de terminaison. » Si l'algorithme se termine mais ne donne pas la bonne solution, on a un « problème de correction (ou de validité) » .
⇝ Pour savoir si un algorithme est correct, on peut faire des tests ou bien donner une démonstration mathématique. L'inconvénient des tests est qu'ils ne permettent pas toujours de trouver les bugs obscurs ; celui des preuves mathématiques est qu'elles peuvent également être fausses...Pour plus de sureté, on utilisera en informatique les deux en parallèle !

Définitions :

- Un algorithme se termine si son exécution sur machine s'arrête quelle que soit la nature des données en entrée.
- La correction d'un algorithme est partielle quand le résultat est correct lorsque l'algorithme s'arrête, et la correction est totale lorsqu'elle est partielle et que l'algorithme se termine.

⇝ Les méthodes de preuves pour une fonction récursive.

Dans une preuve de terminaison, il s'agit de montrer que la fonction parvient toujours au traitement du (ou des) cas de base en un nombre fini d'appels. La méthode est la suivante :
• la preuve s'effectue par récurrence sur la taille du problème à résoudre.
• la base de la récursivité est l'initialisation dans la récurrence.
• pour l'hérédité dans la récurrence, on suppose que l'algorithme se termine pour les problèmes de taille inférieure ou égale à n et on montre qu'il se termine pour un problème de taille $n + 1$.

Dans une preuve de correction, il s'agit de montrer que l'algorithme calcule bien le résultat attendu. La méthode est la même que pour la terminaison, c'est-à-dire par récurrence sur la taille du problème à résoudre.

Exercice :

1. Ecrire une fonction `exposant` qui prend en entrée un entier naturel n, et renvoie le plus grand entier naturel i vérifiant $2^i \leqslant n$.
2. Ecrire une fonction récursive `som_bin` qui prend en entrée un entier naturel n et renvoie la somme des « 1 » dans son développement en écriture binaire, calculée de façon récursive en utilisant la fonction `exposant` précédente. Par exemple, l'appel `som_bin(22)` renverra 3 puisque $22 = 2^4 + 2^2 + 2^1$.

Exemples d'utilisation graphique de la récursivité : 1) flocon de Von Koch.

```
import numpy as np # pour pouvoir utiliser les fonctions usuelles
import matplotlib.pyplot as plt # pour la partie graphique

plt.close('all')
plt.figure(1)
plt.axis('equal')

def trait(x0,y0,long,angle): # pour tracer un trait
    plt.axis('equal')
    m=100
    X=[x0+long*np.cos(angle)*k/m for k in range(m)]
    Y=[y0+long*np.sin(angle)*k/m for k in range(m)]
    plt.plot(X,Y,'k')


# un morceau du flocon
def vonk(x0,y0,n,cote,angle): # n est la profondeur dans la récursivité
    if n==0:
        trait(x0,y0,cote,angle)
    else:
        vonk(x0,y0,n-1,cote/3,angle)
        x1=x0+cote/3*np.cos(angle) ; y1=y0+cote/3*np.sin(angle)
        vonk(x1,y1,n-1,cote/3,angle+np.pi/3)
        x2=x1+cote/3*np.cos(angle+np.pi/3) ; y2=y1+cote/3*np.sin(angle+np.pi/3)
        vonk(x2,y2,n-1,cote/3,angle-np.pi/3)
        x3=x0+2*cote/3*np.cos(angle) ; y3=y0+2*cote/3*np.sin(angle)
        vonk(x3,y3,n-1,cote/3,angle)

# tout le flocon
def flocon(x0,y0,n,cote,angle):
    x1=x0 ; y1=y0
    for _ in range(3):
        vonk(x1,y1,n,cote,angle)
        x1=x1+cote*np.cos(angle)
        y1=y1+cote*np.sin(angle)
        angle-=np.pi/3*2
    plt.show()
```

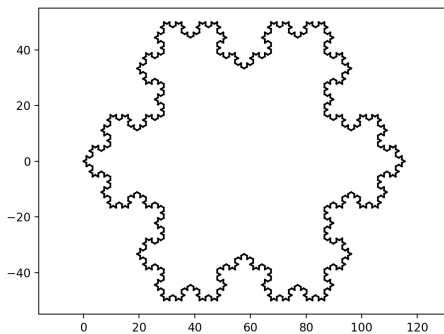
Par exemple, l'appel `flocon(0,0,4,100,np.pi/6)` produit le graphique ci-dessous.

2) Dessin d'un arbre

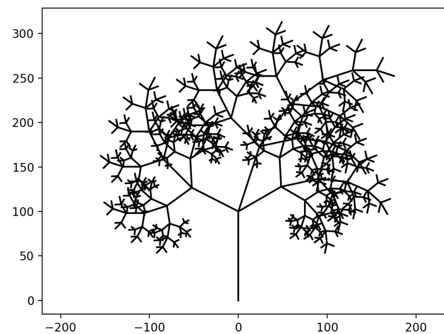
```
# Dessin d'un arbre
def arbre(x0,y0,n,cote,angle):
    if n==0:
        return None
        #trait(x0,y0,cote,angle)
    else:
        trait(x0,y0,cote,angle)
        x1=x0+cote*np.cos(angle) ; y1=y0+cote*np.sin(angle)
        arbre(x1,y1,n-1,cote/1.7,angle+np.pi/4+np.pi/10)
        arbre(x1,y1,n-1,cote/1.3,angle-np.pi/10)
        arbre(x1,y1,n-1,cote/1.8,angle-np.pi/3)

def trace_arbre(x0,y0,n,cote,angle):
    arbre(x0,y0,n,cote,angle)
    plt.show()
```

Cette fois, l'appel `trace_arbre(0,0,7,100,np.pi/2)` produit le graphique ci-dessous.



(a) Flocon de Von Koch ($n = 4$).



(b) Arbre construit récursivement ($n = 7$).

E. Exemple important : l'exponentiation rapide

Le but de ce paragraphe est de montrer qu'une programmation astucieuse peut considérablement changer la vitesse d'exécution d'une fonction. On suppose qu'on ne dispose pas ici de la fonction puissance.

F. Les solutions proposées

1] Première méthode : solution « naïve » itérative, à l'aide d'une boucle for. La fonction est la suivante (pour des paramètres x flottant et n entier naturel) :

```
def puissance(x,n):
    puis=1
    for i in range(n):
        puis=puis*x
    return puis
```

La complexité en temps est en O(n) (linéaire) et la complexité en espace en O(1).

2] Deuxième méthode : solution « naïve » à l'aide d'une fonction récursive. La fonction est la suivante (pour des paramètres x flottant et n entier naturel) :

```
def puissance_rec(x,n):
    if n==0:
        return 1
    else:
    return x*puissance_rec(x,n-1)
```

La complexité en temps est en O(n) (linéaire) mais la complexité en espace est en O(n).

3] Troisième méthode : récursive rapide. Le principe de la méthode repose sur les propriétés suivantes :

$$x^0 = 1,\ x^{2p} = (x^2)^p,\ \text{et } x^{2p+1} = x \cdot x^{2p}.$$

Ainsi, la définition « récursive » sera la suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0\,; \\ (x^2)^{n/2} & \text{si } n \text{ est pair}\,; \\ x \cdot (x^2)^{(n-1)/2} & \text{si } n \text{ est impair.} \end{cases}$$

La fonction est la suivante (toujours pour des paramètres x flottant et n entier naturel) :

```
def puissance_rec_rapide(x,n):
    if n==0:
        return 1
    elif n%2==0:
        return puissance_rec_rapide(x**2,n/2)
```

```
    else:
        return x*puissance_rec_rapide(x**2,(n-1)/2)
```

Observez la simplicité de l'écriture et la lisibilité de la fonction !

4] Quatrième méthode : itérative rapide. La fonction est la suivante (toujours pour des paramètres x flottant et n entier naturel) :

```
def puissance_rapide(x,n):
    result=1
    j=x
    expos=n
    while expos>0:
        if expos%2==0:
            j=j**2
            expos=expos/2
        else:
            result=result*j
            expos=expos-1
    return result
```

On constate que c'est beaucoup moins lisible...

⇝ Complexité et extension aux matrices.

On peut voir par exemple sur le calcul de x^{1000} que l'exponentiation rapide est vraiment beaucoup plus rapide : en fait, on peut démontrer que, dans les cas les plus défavorables, l'algorithme d'exponentiation rapide a une complexité temporelle logarithmique, d'ordre $O(\ln n)$) (pour des ordres de grandeur, voir le paragraphe sur la complexité).
Pour calculer les différents termes de la suite de Fibonacci, on peut utiliser l'exponentiation rapide sur les matrices, en traduisant le problème sous une forme matricielle judicieuse. On obtient ainsi une fonction récursive très efficace.

Chapitre 3

Complexité d'un algorithme

I. Principes

A. Définitions

Pour évaluer l'efficacité d'un algorithme, on introduit des mesures numériques. Elles sont de deux natures :

- *temporelle* : évaluer l'ordre de grandeur du "temps" d'éxécution d'un algorithme.

  ⇝ On l'évalue au moyen de l'ordre de grandeur du nombre d'opérations élémentaires, en fonction de la taille des données, notée n, nécessaires pour mener à bien l'algorithme.

  Cela dépend bien sûr de la machine utilisée et aussi des données. On exprime cette vitesse par comparaison avec des suites usuelles : un temps d'exécution en $O(n)$ est un temps d'éxécution asymptotiquement linéaire par rapport à la taille des données.

- *spatiale :* évaluer l'ordre de grandeur du nombre de cases mémoires accédées.

  ⇝ Au temps, pas si lointain, où la mémoire des ordinateurs coûtait très cher, l'objectif d'une bonne programation était de réduire la complexité spatiale. C'est maintenant la complexité temporelle qui est plutôt le point sensible.

Remarques :

1. Ces deux notions sont liées : sur des machines séquentielles (un seul processeur qui exécute les instructions de maniére séquentielle) une inscription en mémoire nécessitant au moins une unité de temps, la complexité temporelle est au moins égale à la complexité spatiale.
   Ce n'est pas le cas pour des machines parallèles.
2. L'étude de la complexité d'un algorithme indique un comportement asymptotique. Il arrive qu'un algorithme grossier soit plus rapide qu'un algorithme fin sur un petit nombre de données.

B. Le langage mathématique pour décrire la complexité d'un algorithme

Définitions :

❍ Soit $t(n)$ le temps d'exécution d'un algorithme sur un système de données de taille n. Soit f une fonction à valeurs $\mathbb{R}_+^*$.
On dit que l'algorithme a une complexité :
- en $O(f(n))$ si $\dfrac{t(n)}{f(n)}$ est majoré ;
- en $\Theta(f(n))$ si $\dfrac{t(n)}{f(n)}$ est compris entre deux constantes strictement positives.

❍ On dit que l'algorithme est de complexité :
- bornée si $t(n) = O(1)$;
- logarithmique si $t(n) = O(\log_2(n)) = O(\ln(n))$;
- linéaire si $t(n) = O(n)$;
- polynomiale si $t(n) = O(n^k) \qquad (k \geq 2)$;
- exponentielle si $t(n) = O(r^n)$, pour un $r > 1$.

⇝ La taille des données numériques est bornée par leur représentation. Les temps d'exécution de l'addition de deux nombres, de leur multiplication et plus généralement de n'importe quelle opération mathématique usuelle sont bornés par une constante dépendant de la machine.

❍ On appelle opérations élémentaires toutes les opérations consistant en :
- une comparaison, un test, une opération logique,
- une opération arithmétique, une fonction mathématique,
- l'accès à un élément d'une structure de donnée (liste, chaine,...), sa création, sa modification,
- l'affectation de variable,
- saisie/impression/retour d'une donnée à l'écran, ou dans un fichier
- toute instruction ou opération prédéfinie s'effectuant sur une donnée de taille fixée par le système, et dont le temps d'exécution est majoré par une constante ne dépendant que du système.

II. Un peu de pratique

A. Ordres de grandeurs

On considère une machine qui serait capable d'effectuer 10^8 opérations élémentaires par seconde. Le tableau suivant indique un ordre de grandeur relatif du temps d'éxécution d'un algorithme suivant sa complexité :

n	$\log_2(n)$	n	$n\log_2(n)$	n^2	2^n	10^n
10	2×10^{-8}sec	10^{-7}sec	3×10^{-7}sec	10^{-6}sec	10^{-5}sec	100sec
20	3×10^{-8}sec	2×10^{-7}sec	8×10^{-7}sec	4×10^{-6}sec	10^{-2}sec	31709ans
50	4×10^{-8}sec	5×10^{-7}sec	3×10^{-6}sec	3×10^{-5}sec	100jours	
10^3	7×10^{-8}sec	10^{-5}sec	10^{-4}sec	10^{-2}sec		
10^5	2×10^{-7}sec	10^{-3}sec	$0,016$sec	100 sec		
10^6	$2,6\times10^{-7}$sec	10^{-2}sec	$0,2$sec	$2,8$ heures		

B. Complexité dans le pire des cas et en moyenne

Pour évaluer la complexité d'un algorithme on peut étudier la complexité dans le meilleur des cas, dans le pire des cas, ou en moyenne.

Définitions : si $t(d)$ représente, pour l'algorithme A, la complexité en temps sur la donnée d, et D_n l'ensemble des données de taille n, alors :

- $\max_{d\in D_n}(t(d))$ est appelé complexité dans le pire des cas.
- $\min_{d\in D_n}(t(d))$ est appelé complexité dans le meilleur des cas.
- $\sum_{d\in D_n} p(d)t(d)$, où $p(d)$ est la probabilité d'obtenir la donnée d en entrée de l'algorithme, est la complexité en moyenne (HP).
  En particulier si toutes les données sont équiprobables, cela vaut $\dfrac{1}{|D_n|}\sum_{d\in D_n} t(d)$.

Remarques :

1. Il est immédiat que la complexité en moyenne est comprise entre la complexité dans le meilleur et le pire des cas.
2. Ces différents types de complexité donnent des renseignements particuliers sur l'efficacité de l'algorithme : une mauvaise complexité dans le pire des cas donnera un algorithme qui risque d'être trop long pour des données défavorables, même si la complexité en moyenne est favorable.
3. Toutefois c'est la complexité en moyenne, plus lourde à calculer (dans de nombreux problèmes on ne sait pas la calculer) qui renseigne le mieux. D'autant que son calcul donne souvent des indications sur la fréquence des cas défavorables.
4. La complexité dans le meilleur des cas n'est pas très intéressante d'un point de vue théorique.

III. Exemples

A. Recherche d'un élément dans un tableau

Problématique : dans un tableau de taille n (ie une liste en python), où chaque élément peut prendre p valeurs distinctes, on veut déterminer si l'élément x figure dans le tableau.

1] Recherche « naïve » d'un élément dans un tableau. On parcourt le tableau en entier. Dans ce cas, toutes les complexités sont en $O(n)$, ie linéaires.

2] Recherche d'un élément dans un tableau avec arrêt. On parcourt le tableau et on s'arrête dès qu'on a trouvé l'élément x (s'il est dans le tableau).
La complexité dans le meilleur cas est $O(1)$, dans le pire $O(n)$. Et celle en moyenne ? Pour répondre à cette question, on fait les calculs suivants :
il y a a p^n tableaux possibles. Le nombre de tableaux de coût k ($1 \le k \le n-1$) est égal à $(p-1)^{k-1}p^{n-k}$ (les $k-1$ premiers éléments sont distincts de x et les $n-k$ derniers sont quelconques).
Le nombre de tableaux de coût n est $p(p-1)^{n-1}$. On obtient donc le coût moyen :

$$\begin{aligned} m(n) &= \frac{1}{p^n}(np(p-1)^{n-1} + \sum_{k=1}^{n-1} k(p-1)^{k-1}p^{n-k}) \\ &= n\frac{(p-1)^{n-1}}{p^{n-1}} + \frac{1}{p}\sum_{k=1}^{n-1} k(\frac{p-1}{p})^{k-1}. \end{aligned}$$

On calcule $\sum_{k=1}^{n-1} k(\frac{p-1}{p})^{k-1}$ (en dérivant $x \longmapsto \sum_{k=0}^{n-1} x^k$). On en déduit :

$$m(n) = p - \frac{(p-1)^n}{p^{n-1}} \leq p.$$

Si p est fixé, on a donc une complexité moyenne en $O(1)$ et si p peut prendre des grandes valeurs en fonction de n, la complexité est en $O(p)$: c'est nettement mieux que la complexité moyenne précédente !

B. Recherche dichotomique dans un tableau préalablement trié

L'algorithme itératif de recherche dichotomique d'un élément e dans le tableau (la liste) L déjà trié(e) est le suivant :

```
def dicho_rech(L,e):
    while len(L)>0:
        i=int(len(L)/2)
        if L[i]==e:
            return True
        elif L[i]<e:
            L=L[i+1:]
        elif L[i]>e:
            L=L[:i]
    return False
```

Propriété : la complexité de l'algorithme précédent est logarithmique dans le pire des cas.
Démonstration : pour une liste de taille N le nombre d'opérations élémentaires pour rechercher un élément par dichotomie est du même ordre que la profondeur de la dichotomie, puisque chaque étape consiste en au plus 2 affectations et 3 tests. Or, si cette profondeur vaut k, c'est que la liste de départ a un cardinal compris entre 2^k et 2^{k+1}. On en déduit que la complexité est dans le pire des cas en $\mathrm{O}(\log_2(N))$.

⇝ C'est un des intérêts immédiats des algorithmes de tri que l'on étudiera par la suite !

C. Produit matriciel

Pour des matrices de taille n, la complexité naïve est en $O(n^3)$. On peut faire un petit peu mieux (algorithme de Strassen par exemple).

D. Résolution d'un système linéaire par la méthode de Gauss

Pour les détails de la méthode, vous êtes renvoyés au cours de Mathématiques pages 36 et 37.
La complexité de cet algorithme est en $O(n^3)$.

E. Récurrences usuelles

1] Récurrence à un pas ⇝ Ce paragraphe est utile pour l'étude de la complexité de fonctions récursives, ce que l'on peut mettre sous la forme :

$$T(n) = aT(n-1) + f(n),\ a \in \mathbb{N}^*$$

C'est le cas d'un algorithme pour lequel la résolution pour un système de données de taille n se ramène à a sous-problèmes de taille $n-1$, la quantité $f(n)$ représentant le temps nécessaire pour découper puis recomposer le problème initial en sous problèmes.
⇝ C'est le cas pour les tours de Hanoï(a=2) et pour l'algorithme de tri rapide (voir cours ultérieur).

• Cas où $a = 1$ et $f(n) = O(n)$.
Par récurrence on a $T(n) = T(0) + \sum_{k=1}^{n} f(k)$. Il faut donc estimer asymptotiquement $\sum_{k=1}^{n} f(k)$. Mais si $f(n) = O(n)$, on peut montrer que cette somme est en $O(n^2)$: la complexité est quadratique.

• Cas où $a > 1$ et $f(n) = O(n)$.
On peut poser $U(n) = \dfrac{T(n)}{a^n}$; on a alors $U(n) = U(n-1) + \dfrac{f(n)}{a^n}$, ce qui permet d'utiliser le résultat précédent, d'où : $T(n) = a^n(T(0) + \sum_{k=1}^{n} \dfrac{f(k)}{a^k})$, ce qui donne une complexité en $O(a^n)$, ie exponentielle.

Remarque : en fait, si $a \geqslant 2$ et si f est une fonction polynomiale, on peut montrer que l'on obtient une complexité exponentielle $O(a^n)$.

2] Récurrence à plusieurs pas C'est le cas par exemple de la suite de Fibonacci. On a alors une complexité exponentielle.

Exercice : On donne le code python suivant :

```
from math import floor
def nombre(n):
      assert n>=1 and n==floor(n)
      if n==1:
            return 0
      elif n==2 or n==3:
                 return 1
      else:
            return nombre(n-2)+nombre(n-3)
```

1. Décrire à quoi sert la ligne
   `assert n>=1 and n==floor(n)` ?
2. Quelle sera la réponse à l'instruction `nombre(7)` ?
   Expliquer votre raisonnement.
3. Expliquer la signification de la valeur retournée par `nombre(n)`.
4. (a) On appelle $C(n)$ le nombre d'opérations (affectations, comparaisons,...) effectuées par la fonction `nombre`. Montrer que l'on a la relation de récurrence pour $n \geqslant 4$:
   $$C(n) = 8 + C(n-2) + C(n-3).$$
   (b) Déterminer une suite auxiliaire $D(n)$ vérifiant :
   $$D(n) = D(n-2) + D(n-3).$$
   (c) Que peut-on dire de la complexité de la fonction `nombre` ?
   Justifier votre réponse.

Chapitre 4

Rappels : tris

I. Introduction

Les algorithmes de tri sont très répandus et largement utilisés en informatique. Il en existe beaucoup (tri par sélection, tri par insertion, tri à bulles, tri rapide, tri fusion pour les tris qui comparent les éléments 2 à 2, tri radix pour un autre type de tri).
Certaines opérations sont beaucoup plus efficaces dans une liste qui est déjà triée ; on a vu par exemple au chapitre 2 que la recherche d'un nombre dans une liste d'éléments triés est de complexité logarithmique, ce qui est nettement mieux que dans une liste non triée (complexité linéaire dans le pire des cas et en moyenne).

II. Tri par insertion

C'est le tri « naturel » que l'on utilise usuellement pour trier un paquet de copies par ordre de notes, un paquet de fiches par ordre alphabétique, un paquet de cartes,...
Etant donné un tableau de N éléments, en partant de la gauche et en allant vers la droite, on insère au fur et à mesure chaque élément dans la partie gauche du tableau qui est déjà triée.

A. Code en Python

Tri par insertion

```
def tri_insertion(L) :
    n=len(L)
    for i in range(1,n) :
        x=L[i]
        k=i
        while k>0 and L[k-1]>x :
            L[k]=L[k-1]
            k=k-1
        L[k]=x
    return L
```

B. Correction de l'algorithme et invariant de boucle

On utilise l'invariant de boucle suivant :

« après le p-ème balayage, les p premiers éléments du tableau sont ordonnés dans le sens croissant. »

On montre que cela fonctionne par récurrence sur p :
• Initialisation : au premier passage, lorsque $i = 0$, le tableau est inchangé et le premier élément est bien ordonné ;
• Hérédité : supposons que les $p - 1$ premiers éléments du tableau sont ordonnés dans le sens croissant (pour $i \in [\![0; p-2]\!]$). Au p-ème balayage, lorsque $i = p - 1$, l'élément à insérer est $T[p-1]$. On le décale vers la gauche jusqu'à ce qu'il soit inséré da façon à avoir les p premiers éléments du tableau ordonnés dans le sens croissant. Remarquer ce qu'il se passe si $T[p-1] < T[k]$ pour $k < p - 1$.

C. Complexité du tri par insertion

On suppose qu'on trie un tableau de N éléments avec le tri par insertion.
• C'est un tri *en place*, qu'on peut effectuer directement sur le tableau passé en paramètre : c'est-à-dire que sa complexité spatiale (en espace mémoire) est $O(1)$.
• Dans le pire des cas, on effectue $\Theta(N^2)$ opérations élémentaires (N passages dans la boucle for et pour chaque i, i passage dans la boucle while) : la complexité temporelle est quadratique. C'est le cas pour un tableau ordonné au départ par ordre décroissant.

• Dans le meilleur des cas, on effectue $\Theta(N)$ opérations élémentaires (N passages dans la boucle for et pour chaque i, 1 passage dans la boucle while) : la complexité est linéaire. C'est le cas pour un tableau ordonné au départ par ordre croissant.

• Dans le cas d'un petit nombre d'éléments à trier, ou lorsque le tableau est déjà presque trié, c'est un algorithme très efficace ; c'est pour cela qu'on l'utilise pour trier des cartes ou des copies !

Toutefois, on va voir au paragraphe suivant que pour un grand nombre d'éléments à trier, avec des tableaux très désordonnés, il existe des algorithmes nettement plus efficaces.

III. Tri rapide (quick sort)

A. Introduction

• Un algorithme de tri a une complexité (temporelle) au moins linéaire, puisqu'il faut parcourir tous les éléments du tableau pour pouvoir les comparer.
On peut montrer que dans le pire des cas (et en moyenne également), la complexité pour un tableau de taille N ne peut être meilleure que $\Theta(N \ln(N))$. C'est le cas des deux algorithmes de tris que l'on va étudier dans ce chapitre : on parle alors d'algorithmes optimaux.

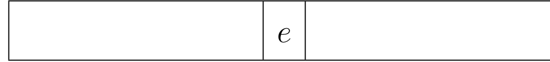
• Ces algorithmes optimaux sont basés sur le principe informatique de diviser pour régner :

pour trier un tableau de taille N, on trie (récursivement) deux sous-tableaux de taille $\approx N/2$.

Remarque : il faudra regrouper à chaque étape les deux sous-tableaux triés (ce qu'on appelle fusionner) ; cette opération ne doit pas être trop coûteuse en temps.

B. Principe

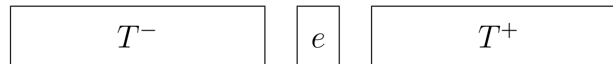
Etant donné un tableau T de N éléments :
• on choisit arbitrairement un élément e, appelé pivot, dans ce tableau (au hasard, mais on le choisira typiquement « au milieu » du tableau) ;



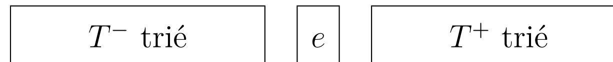
• décomposer les autres éléments du tableau T en deux sous-tableaux :

— un sous-tableau T^- constitué par les éléments qui sont inférieurs (ou égaux) à e ;

— un sous-tableau T^+ constitué par les éléments qui sont strictement supérieurs à e.



• Après appels récursifs sur les deux sous-tableaux T^- et T^+, le tableau T trié s'obtient en concaténant le sous-tableau T^- trié, suivi de l'élément e et du sous-tableau T^+ trié.



Exemple : trier le tableau $[15, 3, 2, 10, 8, 8, 9, 5, 6, 4, 1]$ à la main avec le tri rapide. On mettra en évidence à chaque étape le pivot et les sous-tableaux sur lequels on travaille.

Code en Python

Le code Python pour le tri rapide est le suivant :

```
def Tri_rapide(T):
      N=len(T)
      if N<=1:
          return T
      e=T[N//2] # choix d'un pivot central
      Tg,Td=[],[]
      for i in range(N):
          if x<=e and i != N//2:
              Tg.append(x)
          if x>e:
              Td.append(x)
      return Tri_rapide(Tg)+[e]+Tri_rapide(Td)
```

Remarques :

1. Le choix d'un pivot central permet d'optimiser l'algorithme dans le cas où la liste de départ est presque triée.
2. L'écriture de ce tri est simple, mais la complexité spatiale peut être importante. Il est possible d'effectuer un tri rapide *en place*, dont la complexité spatiale est en $O(len(T))$ (cf infra).

Correction de l'algorithme du tri rapide

Comme on utilise la récursivité, il s'agit ici de montrer que l'algorithme s'arrête, et qu'à la fin le tableau retourné est bien trié.

⇝ L'algorithme s'arrête bien puisqu'à chaque appel récursif chacun des deux tableaux est de longueur strictement inférieure à celle du tableau initial, et que pour un tableau de longueur inférieure ou égale à 1, l'appel récursif s'achève.

⇝ Montrons que le tableau retourné à la fin est trié, à l'aide d'une récurrence forte sur sa taille N :

• Initialisation : pour un tableau de longueur 0 ou 1, l'algorithme renvoie le même tableau, qui est bien trié.
• Hérédité : supposons que T est un tableau de longueur $N+1$.
L'algorithme renvoie le tableau obtenu par concaténation de $TRIRAPIDE(T^+)$, e et $TRIRAPIDE(T^-)$.
Les tableaux T^+ et T^- étant de longueur strictement inférieure à $N+1$, d'après l'hypothèse de récurrence les tableaux $TRIRAPIDE(T^+)$ et $TRIRAPIDE(T^-)$ sont triés.
Comme par construction tous les éléments du tableau $TRIRAPIDE(T^-)$ sont inférieurs à e et tous les éléments du tableau $TRIRAPIDE(T^+)$ sont strictement supérieurs à e, le tableau obtenu par concaténation est également trié.

Complexité du tri rapide

On suppose qu'on trie un tableau de N éléments avec le tri rapide.
Dans ce cas, les seules opérations non bornées en temps (ie qui dépendent de N) sont :

— `T.pop(N//2)` de complexité $\Theta(N)$;

— La boucle `for` (constitution des deux tableaux) qui prend un temps linéaire $\Theta(N)$;

— l'appel récursif sur deux tableaux de taille K et $N-K$;

— La concaténation `T1 + [e] + T2` de complexité $\Theta(N)$.

Ainsi, si $C(N)$ compte le nombre d'opérations pour un tableau de longueur N, on a la relation de récurrence :

$$C(N) = C(K-1) + C(N-K) + \Theta(N).$$

Dans le pire des cas, lorsque l'élément choisi est toujours extrémal (minimum ou maximum), on a N appels récursifs, avec $\Theta(N-k)$ opérations à l'étape k. On obtient donc $C(N) = \Theta(N^2)$, ie une complexité quadratique.

Dans le meilleur des cas, l'élément choisi est médian, et il y aura $\log_2(N)$ appels récursifs, avec de l'ordre de N opérations à chaque étape. On obtient

$$C(N) = \Theta\left(\sum_{k=1}^{\log_2(N)} 2^k \cdot \frac{N}{2^k}\right) = \Theta(N\log_2(N)).$$

Cette complexité est nettement meilleure que quadratique (voir graphiques page suivante) !

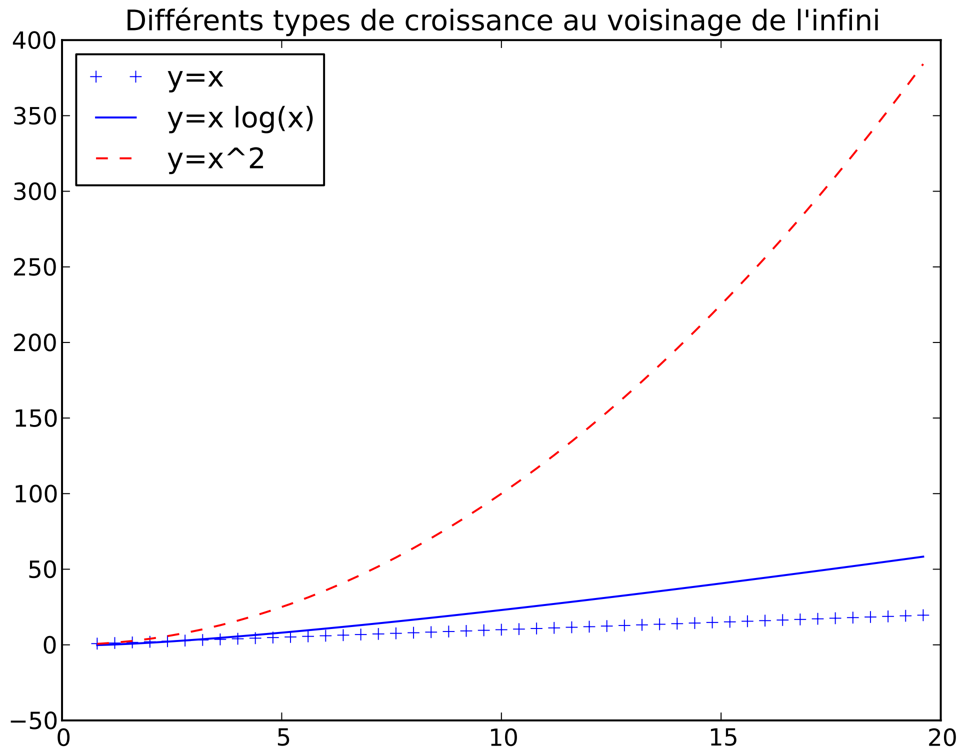


Figure 4.1: Croissances linéaire, quadratique et log-linéaire.

En résumé, le tri rapide a une complexité temporelle (pour un tableau de taille N) :

- en $O(N^2)$ dans le pire des cas ;
- en $N\log_2(N)$ dans le meilleur des cas ;
- en $N\log_2(N)$ en moyenne (admis et de toute façon hors programme).

Comme son nom l'indique, il est très rapide, en particulier sur des tableaux avec un grand nombre d'éléments.

Remarque : en revanche, la complexité spatiale du programme précédent est importante : elle est de l'ordre de N^2. Pour pallier cet inconvénient, on peut écrire un algorithme de tri rapide en place, dont la complexité spatiale est cette fois en $O(N)$:

```
def echange(tab,i,j):
    tab[i],tab[j]=tab[j],tab[i]

def partition(tab,g,d):
    assert g<d
    echange(tab,g,(g+d)//2)
    val=tab[g] ; m=g
    for k in range(g+1,d):
        if tab[k]<val:
            m=m+1
            echange(tab,k,m)
    if m!=g:
        echange(tab,g,m)
    return m

def tri_rapide_rec(tab,g,d):
    if g>=d-1:
        return
    m=partition(tab,g,d)
    tri_rapide_rec(tab,g,m)
    tri_rapide_rec(tab,m+1,d)


def tri_rapide(tab):
    tri_rapide_rec(tab,0,len(tab))
    return tab
```

IV. Tri fusion

A. Principe

C'est un autre exemple de tri basé sur le principe « diviser pour régner », utilisant la récursivité. Partant d'un tableau avec N éléments, on le coupe en deux sous-tableaux de taille égale (ou bien à un élément près), on trie chacun de ces deux sous-tableaux (de façon récursive) et on fusionne les deux sous-tableaux triés.
L'efficacité du tri fusion repose sur le fait que fusionner deux tableaux triés en un seul tableau trié se fait avec une complexité linéaire sur le nombre total d'éléments, dans le meilleur et dans le pire des cas.

Exemple : trier le tableau $[15, 3, 2, 10, 8, 8, 9, 5, 6, 4, 1]$ à la main avec le tri fusion. On mettra en évidence à chaque étape les sous-tableaux sur lequels on travaille.

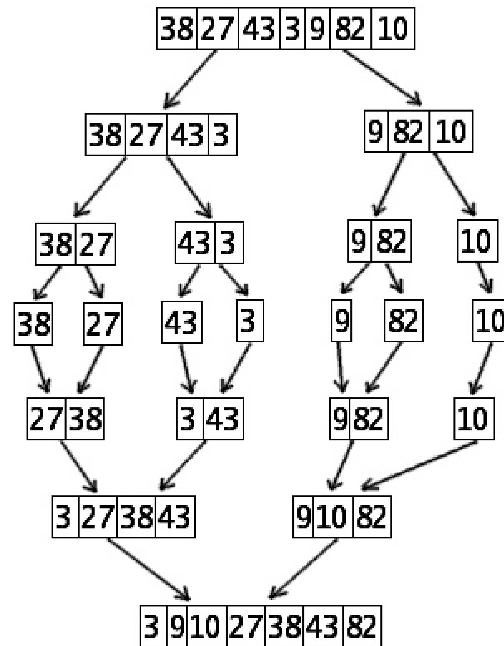


FIGURE 4.2: Un exemple de tri fusion.

B. Code en Python

On écrit d'abord une fonction auxiliaire `fusion`, qui renvoie le tableau obtenu par fusion des tableaux triés L_1 et L_2 donnés en entrée ; puis on donne la fonction effectuant le tri fusion :

Tri fusion

```
def fusion(L1,L2) :
    n1,n2=len(L1),len(L2)
    out=[]
    i1,i2=0,0
    for i in range(n1+n2) :
        if i2==n2 or ( i1<n1 and L1[i1]<L2[i2] ) :
            out.append(L1[i1])
            i1 += 1
        else :
            out.append(L2[i2])
            i2 += 1
    return out

def tri_fusion(L) :
    n=len(L)
    if n<2 :
        return L
    else :
        m=n//2
        return fusion(tri_fusion(L[:m]),tri_fusion(L[m:]))
```

C. Correction de l'algorithme du tri fusion

Comme pour le tri rapide, on utilise la récursivité, et il s'agit à nouveau de montrer que l'algorithme s'arrête, et qu'à la fin le tableau retourné est bien trié.

⇝ L'algorithme s'arrête bien puisqu'à chaque appel récursif chacun des deux tableaux est de longueur strictement inférieure à celle du tableau initial, et que pour un tableau de longueur égale à 1 (ce que l'on finit toujours par obtenir), l'appel récursif s'achève.

⇝ Montrons que le tableau retourné à la fin est trié, à l'aide d'une récurrence forte sur sa taille N :
• Initialisation : pour un tableau de longueur 1, l'algorithme retourne le même tableau, qui est bien trié.
• Hérédité : supposons que T est un tableau de longueur $N+1$. L'algorithme retourne le tableau obtenu par fusion de $T1$, e et $T2$. Les tableaux $T1$ et $T2$ étant de longueur strictement inférieure à $N+1$, d'après l'hypothèse de récurrence les tableaux $T1$ et $T2$ sont triés. Par définition de la fonction $FUSION$, le tableau obtenu par fusion de $T1$ et $T2$ comporte tous les éléments de T et il est également trié.

D. Complexité du tri fusion

On suppose qu'on trie un tableau de N éléments avec le tri fusion.
On peut associer naturellement un arbre binaire aux calculs récursifs. Dans cet arbre, il y a $\log_2(N)$ étages et à chaque étage il y a $\Theta(N)$ opérations. La complexité dans tous les cas (meilleur, pire et en moyenne) est donc $\Theta(N\log_2(N))$, et elle est donc optimale (au sens où on l'a définie au début de ce cours).

Remarque : à nouveau, la complexité en mémoire est de l'ordre de N^2.

Chapitre 5

Rappels : graphes

I. Graphes non orientés

Définitions I..1

- Mathématiquement, on appelle *graphe non orienté* tout couple $G = (S, A)$ où :
  - S est un ensemble fini non vide dont les éléments sont appelés *sommets* (ou *nœuds*) ;
  - A est un ensemble de paires $\{x, y\}$ avec x, y deux sommets distincts. Ces paires sont appelées *arêtes* ou encore *arcs*.
- Deux sommets reliés par une arête sont dits *adjacents*.
- Les *voisins* d'un sommet x sont les sommets qui sont adjacents au sommet x.
- Le degré d'un sommet x est le nombre d'arêtes de la forme $x - y$, c'est-à-dire le nombre de voisins du sommet x ; on le note $d(x)$.
- On notera $|S|$ le nombre de sommets du graphe G, et $|A|$ le nombre d'arêtes de G.

Remarques :

- dans la pratique, l'arête $\{x, y\}$ sera notée $x - y$ ou $y - x$. Intuitivement, une arête $x - y$ permet de passer du sommet x au sommet y et du sommet y au sommet x.
- dans la définition précédente, nous nous sommes interdits les *boucles*, c'est-à-dire les arêtes reliant un sommet à lui-même.
- dans la définition précédente, nous nous sommes également interdits d'avoir plusieurs arêtes entre deux sommets. Les graphes s'autorisant de telles arêtes sont appelés *multigraphes* .
- Pour un graphe non orienté, $|A| \leqslant |S|(|S| - 1)/2$.
- Si $|S| = n$ et que S n'est pas précisé, on prendra par convention $S = \{0, 1, ..., n - 1\}$.

Exemples :

1. le graphe entièrement déconnecté possède n sommets et aucune arête.
2. le graphe chemin à n sommets possède une arête entre deux sommets i et j de $\{0, 1, ..., n-1\}$ si et seulement si $|i - j| = 1$.
3. le graphe *cycle* à n sommets possède une arête entre deux sommets i et j de $\{0, 1, ..., n-1\}$ si et seulement si $j - i \equiv 1\ [n]$.
4. Le graphe des utilisateurs de Facebook a un sommet pour chaque utilisateur et une arête entre deux sommets lorsque deux utilisateurs sont amis. Notons que $|S|$ est de l'ordre de 10^9 et que $|A|$ est de l'ordre de 10^{11}, ce qui pose quelques difficultés algorithmiques.
5. Le réseau métropolitain de Paris peut être vu comme un graphe non orienté, où les sommets sont les stations et les arêtes les liaisons entre stations :

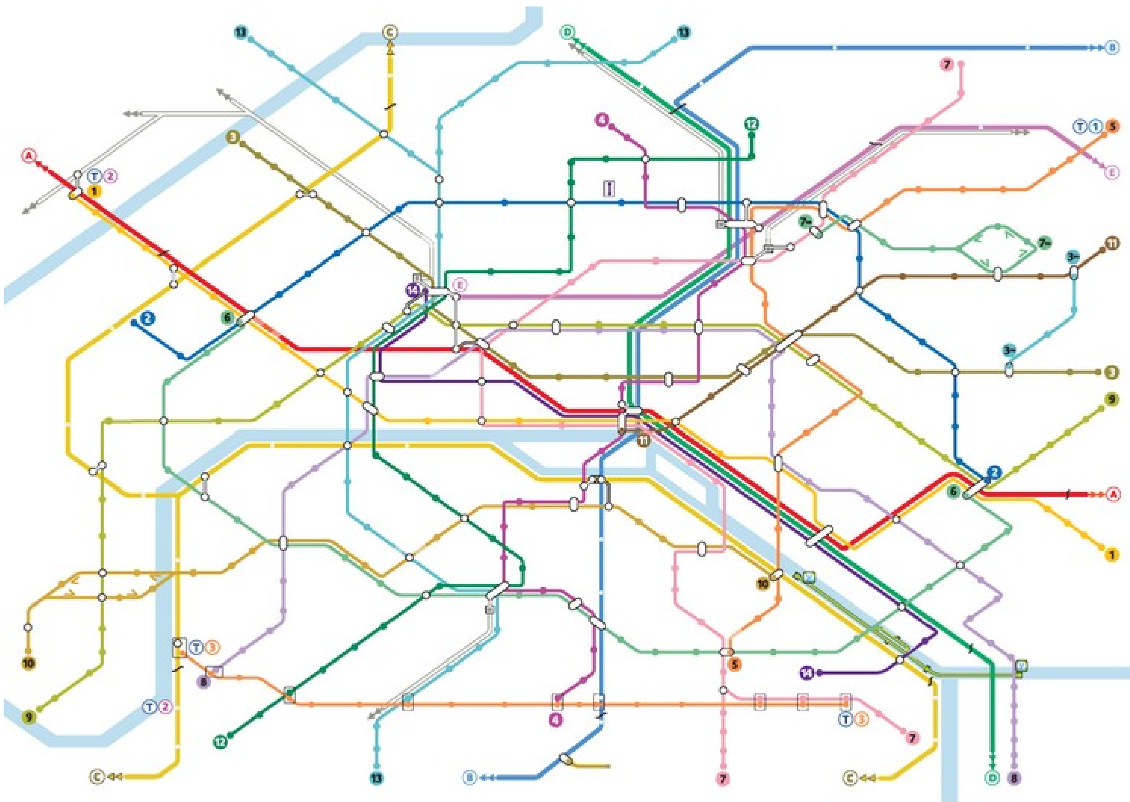


Figure 5.1: Carte du métro de Paris.

Définitions I..2

- On appelle *chemin* (noté c), toute suite de p sommets $z_0, z_1, ..., z_{p-1}$ telle que

$$\forall k \in \{0, ..., p-1\}, z_k - z_{k+1} \in A.$$

  Les sommets z_0 et z_{p-1} sont appelés *extrémités* du chemin et on dit que c relie z_0 à z_{p-1}. De plus, on définit la longueur ℓ du chemin c par $\ell(c) = p - 1$ (nombre d'arêtes utilisées).
- Un chemin peut être de longueur nulle ; dans ce cas, il relie un sommet à lui-même, sans arête.
- Un chemin $z_0, z_1, ..., z_{p-1}$ est dit :
  - *élémentaire* lorsqu'il ne passe pas deux fois par le même sommet, c'est-à-dire lorsque les sommets sont deux à deux distincts ;
  - *simple* s'il ne passe pas deux fois par la même arête, c'est-à-dire lorsque les arêtes $z_k - z_{k+1}$ sont deux à deux distinctes.
- Un sommet y est dit *accessible* depuis un sommet x lorsqu'il existe au moins un chemin reliant x à y. Il en résulte que tout sommet est accessible depuis lui-même.
- Si le sommet y est accessible depuis le sommet x, on dit que x et y sont *connectés* .

Propriétés I..3

1. Tout chemin élémentaire est simple, mais la réciproque est fausse.
2. Dans un graphe non orienté, la relation d'accessibilité est une relation d'équivalence sur l'ensemble des sommets S. Une classe d'équivalence pour cette relation s'appelle une *composante connexe* .

Définitions I..4

- On dit qu'un graphe non orienté est *connexe* lorsqu'il ne possède qu'une seule composante connexe, c'est-à-dire lorsque tous ses sommets sont connectés.
- Si y est un sommet accessible depuis un sommet x, on appelle distance entre x et y l'entier

$$d(x, y) = \min\{\ell(c) \ / \ c \text{ est un chemin de } x \text{ à } y\} .$$

  Un *chemin de longueur minimale* de x à y est un chemin c de x à y tel que $\ell(c) = d(x, y)$.

Remarques :

- Lorsque y n'est pas accessible depuis x, la convention est de poser $d(x, y) = +\infty$.
- Conformément à ce que l'on attend d'une distance, on a les égalités suivantes :

$$\begin{aligned} \forall x \in S, \quad & d(x, x) = 0, \\ \forall x, y \in S, \quad & d(x, y) = d(y, x), \\ \forall x, y, z \in S, \quad & d(x, z) \leqslant d(x, y) + d(y, z). \end{aligned}$$

Définitions I..5

- ❍ On appelle *cycle* tout chemin simple de longueur non nulle dont les extrémités sont identiques.
- ❍ On appelle *arbre* tout graphe connexe et acyclique (c'est-à-dire qu'il ne contient aucun cycle).

Remarques :

- ➪ Pourquoi impose-t-on un chemin *simple* dans la définition d'un cycle ?
- ➪ Dans un graphe non orienté, la longueur d'un cycle est supérieure ou égale à 3.
- ➪ Un cycle est dit élémentaire lorsque la seule répétition de sommets est celle de ses extrémités. Ainsi, un cycle est élémentaire si et seulement s'il ne contient pas d'autre cycle.

II. Graphes orientés

Définitions II..1

- ❍ On appelle *graphe orienté* tout couple $G = (S, A)$ où :
  - S est un ensemble fini non vide dont les éléments sont appelés *sommets* ;
  - A est un ensemble de couples (x, y) avec x, y deux sommets distincts. Ces paires sont appelées *arcs*.
- ❍ S'il y a un arc $x \to y$, on dit que x est un prédécesseur de y et que y est un successeur de x.
- ❍ Dans un graphe orienté, on appelle
  - *degré entrant* d'un sommet x, le nombre d'arcs de la forme $y \to x$.
  - *degré sortant* d'un sommet x, le nombre d'arcs de la forme $x \to y$.
  - *degré total* d'un sommet x, la somme de son degré entrant et de son degré sortant.

Remarques :

- ➪ dans la pratique, l'arc (x, y) sera noté $x \to y$. Intuitivement, un arc $x \to y$ permet de passer du sommet x au sommet y mais pas du sommet y au sommet x.
- ➪ Le degré entrant d'un sommet est son nombre de prédécesseurs et le degré sortant, son nombre de successeurs.
- ➪ Pour un graphe orienté, $|A| \leqslant |S|(|S| - 1)$.
- ➪ En pratique, on confondra souvent un graphe non orienté $G = (S, A)$ avec son graphe orienté associé G_o. Ce dernier possède les mêmes sommets que G. De plus, $x \to y$ est un arc de G_o si et seulement si $x - y \in A$. En particulier, dans G_o, dès que $x \to y$ est un arc, $y \to x$ en est un autre.
  Réciproquement, à un graphe orienté $G = (S, A)$, on associe le graphe non orienté G_o obtenu en « oubliant » l'orientation des arcs.

Exemples :

1. Le graphe du web possède un sommet pour chaque page web et un arc de x vers y lorsque la page x contient un lien vers la page y. C'est ce graphe que les moteurs de recherche parcourent pour construire leur index. La taille du graphe du web est inconnue mais Google indexe plus de 50 milliards de pages.
2. Le graphe des utilisateurs d'Instagram a un sommet pour chaque utilisateur et un arc de x vers y lorsque x est un « follower » de y. Contrairement au graphe des utilisateurs de Facebook qui est non orienté, celui d'Instagram l'est.
3. le graphe *circuit* à n sommets possède une arête entre deux sommets i et j de $\{0, 1, ..., n-1\}$ si et seulement si $j - i \equiv 1\ [n]$.
4. Le graphe des utilisateurs de Facebook a un sommet pour chaque utilisateur et une arête entre deux sommets lorsque deux utilisateurs sont amis. Notons que $|S|$ est de l'ordre de 10^9 et que $|A|$ est de l'ordre de 10^{11}, ce qui pose quelques difficultés algorithmiques.

Définitions II..2

❍ On appelle *chemin* (noté c) de longueur $\ell(c) = p$, toute suite de p sommets $z_0, z_1, ..., z_{p-1}$ telle que

$$\forall k \in \{0, ..., p-1\}, z_k \to z_{k+1} \in A.$$

Les sommets z_0 et z_{p-1} sont appelés *extrémités* du chemin et on dit que c relie z_0 à z_{p-1}.

❍ Comme pour un graphe non orienté, un chemin $z_0, z_1, ..., z_{p-1}$ est dit :
- *élémentaire* lorsqu'il ne passe pas deux fois par le même sommet, c'est-à-dire lorsque les sommets sont deux à deux distincts ;
- *simple* s'il ne passe pas deux fois par la même arête, c'est-à-dire lorsque les arcs $z_k -> z_{k+1}$ sont deux à deux distincts.

❍ Un sommet y est dit *accessible* depuis un sommet x lorsqu'il existe au moins un chemin reliant x à y.

Remarques :

➪ comme pour un graphe non orienté, les chemins élémentaires sont simples, mais la réciproque est fausse.

➪ Dans un graphe orienté, la relation d'accessibilité n'est plus une relation d'équivalence sur l'ensemble des sommets S, et les notions de connexité et de composante connexe n'ont plus de sens pour ces graphes.

➪ La notion de distance entre un sommet x et un sommet y est toujours définie. L'inégalité triangulaire reste vraie, mais la distance n'est plus symétrique.

➪ On appelle *circuit* tout chemin de longueur non nulle dont les extrémités sont identiques. La notion de circuit dans un graphe orienté correspond donc à celle de cycle dans un graphe non orienté. Cependant, contrairement à ce qui se passe dans le cas des graphes non orientés, il existe des circuits de longueur 2.

➪ Pour un graphe orienté, $|A| \leqslant |S|(|S| - 1)$.

III. Graphes pondérés

Définitions III..1

- On appelle *graphe pondéré* la donnée d'un graphe $G = (S, A)$ et d'une application $\rho : A \to \mathbb{R}_+$ appelée *poids* .
- On appelle *poids du chemin* $c = z_0, z_1, ..., z_{p-1}$ le réel positif

$$\rho(c) = \sum_{k=0}^{p-1} \rho(z_k, z_{k+1}).$$

- Si y est un sommet accessible depuis un sommet x, on définit le réel positif

$$\delta(x, y) = \min\{\rho(c) \ / \ c \text{ est un chemin de } x \text{ à } y\} \ .$$

Un *chemin de poids minimal* de x à y est un chemin c de x à y tel que $\rho(c) = \delta(x, y)$.

Remarques :

- un graphe pondéré peut être orienté ou non orienté.
- Il est possible de considérer des graphes pondérés avec des fonctions de poids prenant des valeurs négatives. Mais dans ce cours, puisque c'est une condition pour pouvoir appliquer l'algorithme de Dijkstra, nous nous limiterons à des fonctions de poids positives.
- On appelle *poids d'un graphe* la somme des poids de ses arêtes.
- Dans le cas où les poids sont des distances, par exemple si G est le graphe d'un réseau routier, on pourra parler de distance et de plus court chemin. Il ne faut cependant pas confondre le réel $\delta(x, y)$ avec l'entier $d(x, y)$ représentant le nombre minimal d'arcs reliant x et y.

Exemple : Voici le graphe pondéré (et non orienté) des connexions ferroviaires françaises, le poids représentant les temps de trajet en dizaines de minutes :

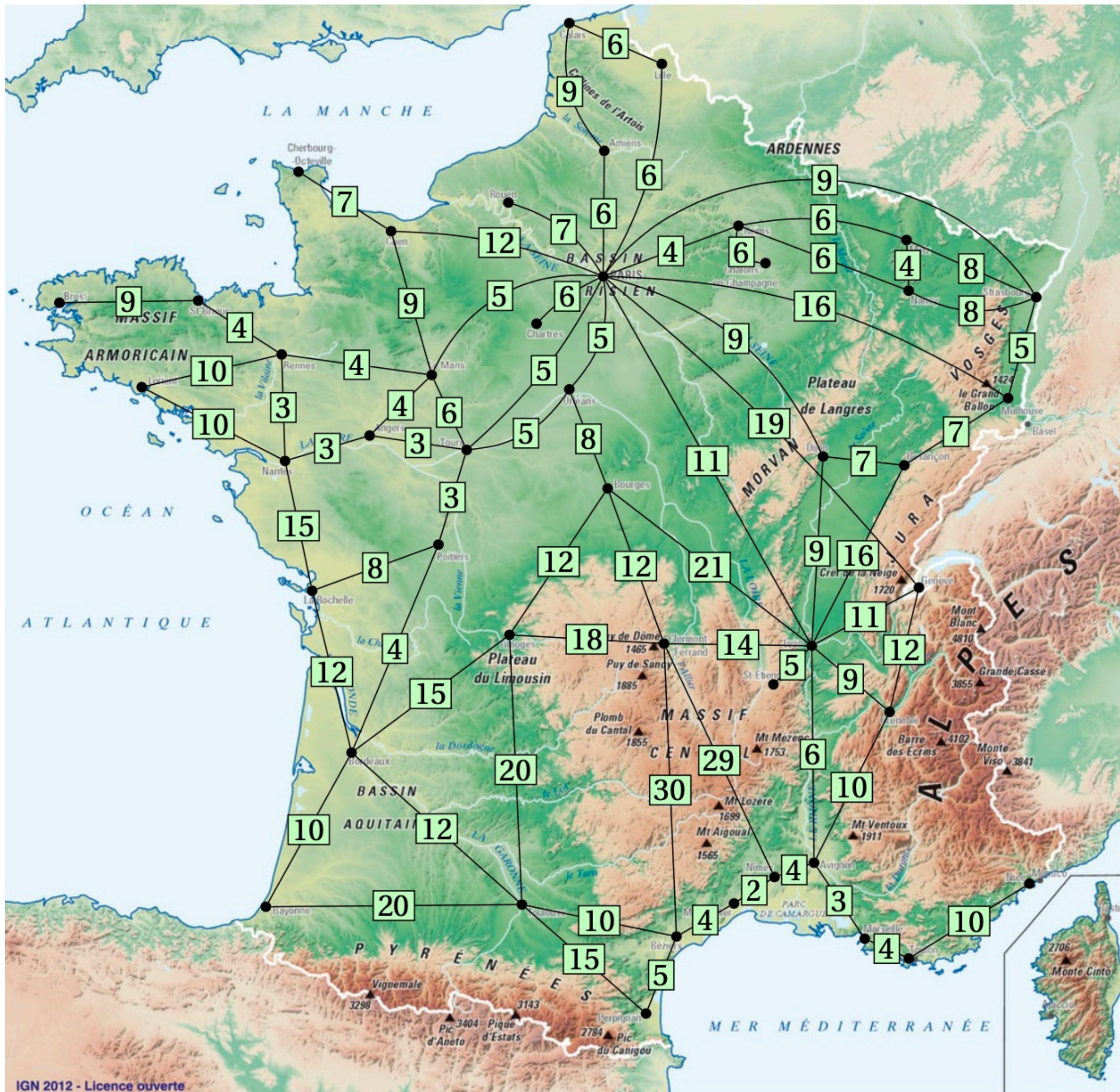


Figure 5.2: Connexions ferroviaires françaises.

IV. Représentation d'un graphe

Définitions IV..1

❍ Soit $G = (S, A)$ un graphe où $S = \{0, ..., n-1\}$. On appelle *matrice d'adjacence* la matrice $M \in \mathcal{M}_n(\mathbb{Z})$ définie par

$$\forall i, j \in \{0, ..., n-1\},\ m_{i,j} = \begin{cases} 1 & \text{si } j \text{ est un successeur de } i, \\ 0 & \text{sinon .} \end{cases}$$

Remarques :

- un graphe est non orienté si et seulement si sa matrice d'adjacence est symétrique.
- Puisqu'on interdit les boucles, une matrice d'adjacence n'a que des zéros sur sa diagonale.
- On peut représenter un graphe pondéré par une matrice d'adjacence $M \in \mathcal{M}_n(\mathbb{R})$. On prend pour coefficient $m_{i,j}$ la valeur $+\infty$ ou **None** lorsqu'il n'existe pas d'arc de i à j, et le poids $\rho_{i,j}$ de l'arc allant de i à j lorsqu'un tel arc existe.

Définition IV..2

❍ Soit $G = (S, A)$ un graphe où $S = \{0, ..., n-1\}$. On appelle *liste d'adjacence* le tableau T de longueur n tel que pour tout $i \in \{0, ..., n-1\}$, T_i est la liste des successeurs $j \in \{0, ..., n-1\}$ de i.

Remarques :

- On peut de même représenter un graphe à l'aide d'un dictionnaire.
- On peut représenter un graphe pondéré par une liste d'adjacence dans laquelle, pour tout sommet i, g_i est la liste des couples $(\rho_{i,j}, j)$ où j est un successeur de i.

V. Algorithmes sur les graphes

Supposons que vous êtes enfermé dans un labyrinthe de salles, connectées entre elles par des portes. Nous représentons ce labyrinthe par un graphe dont les sommets sont les salles et les arêtes sont les portes reliant ces salles entre elles. Si l'on souhaite sortir de ce labyrinthe, un réflexe naturel est d'emprunter au hasard les portes que l'on croise. Cependant, cette stratégie possède deux défauts importants : elle ne nous dit pas ce qu'on doit faire lorsqu'on tombe dans un cul-de-sac et elle ne nous empêche pas de tourner en rond.

Pour résoudre ces deux problèmes, la solution la plus simple est de marquer les salles. Au cours de notre exploration, nous choisirons donc de les placer successivement dans 3 états différents :
—Inconnu : c'est l'état dans lequel est une salle qui n'a pas encore été découverte.
—Découvert : C'est l'état dans lequel on place une salle lorsqu'on l'a aperçue par une porte.
—Visité : C'est l'état dans lequel est une salle dans laquelle nous sommes déjà entrés.

Pour ne pas tourner en rond, il suffit de ne pas entrer dans une salle qui a déjà été visitée. Pour ces salles, on peut choisir de marquer leur sol d'une croix blanche. Et pour savoir que faire lorsqu'on

est dans un cul-de-sac, il suffit de garder une trace des salles que l'on a découvertes, mais qui n'ont pas encore été visitées. Pour cela, on conserve avec nous un « sac » contenant une marque pour chacune d'elles.

Revenons au vocabulaire des graphes. À l'aide de ces deux outils, notre sac ainsi que le marquage des sommets, nous sommes armés pour parcourir l'ensemble des sommets accessibles depuis notre sommet de départ, appelé *source* . Pour cela, il nous suffit de suivre l'algorithme suivant, que nous appelons *parcours générique* :

Algorithme de parcours générique d'un graphe

```
Mettre le sommet source dans le sac
Tant que le sac n'est pas vide faire :
    Prendre un sommet x dans le sac
    si x n'a pas été visité alors
        Marquer le sommet x comme visité
        pour chaque arc x-y faire
            Mettre le sommet y dans le sac
```

La seule propriété dont notre sac a besoin est qu'on puisse y mettre des sommets pour les extraire plus tard. La structure de données que nous allons utiliser pour implémenter notre sac va déterminer l'ordre dans lequel les sommets en sont extraits :

— **Pile :** si nous utilisons une pile, pour laquelle c'est le dernier sommet qui a été placé dans le sac qui en est extrait, nous obtiendrons ce qu'on appelle un *parcours en profondeur* . Bien que tous les parcours nous permettent d'obtenir l'ensemble des sommets accessibles depuis un sommet, la simplicité de la structure de pile fait que c'est souvent ce parcours que nous utiliserons pour cela. Une des propriétés du parcours en profondeur est qu'il nous nous permet de détecter les cycles dans un graphe.

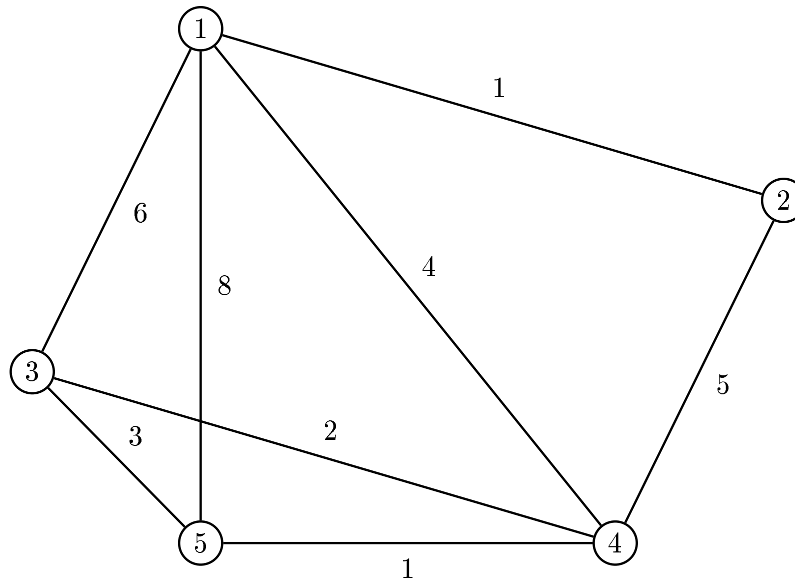
— **File :** si nous utilisons une file, pour laquelle c'est le premier sommet qui a été placé dans le sac qui en est extrait, nous obtiendrons ce qu'on appelle un *parcours en largeur* . Ce parcours nous sera utile pour trouver le chemin de longueur minimale entre la source et les sommets accessibles depuis cette dernière.

— **File de priorité :** l'utilisation d'une file de priorité permet de découvrir une famille d'algorithmes fonctionnant avec les graphes pondérés, par exemple l'algorithme de Dijkstra qui permet de trouver le chemin de poids minimal entre la source et les sommets accessibles depuis cette dernière ; pour cela, la priorité utilisée est le poids du chemin qui nous a permis de découvrir le sommet.

Nous verrons en TP un algorithme de parcours en profondeur ainsi que deux exemples de parcours en largeur avec file de priorité : l'algorithme de Dijkstra et l'algorithme A*.

VI. Exercice

On considère le graphe G non orienté et pondéré suivant, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



On peut par exemple imaginer que les sommets du graphe représentent des villes, que les arêtes sont les autoroutes existantes et que les nombres donnent la distance autoroutière en centaines de kilomètres entre ces villes.

1. Construire la matrice (liste de listes en Python) $(M_{i,j})_{0\leqslant i\leqslant 4, 0\leqslant j\leqslant 4}$ des distances du graphe G définie par : *pour tous indices i, j, $M_{i,j}$ vaut la distance entre les sommets $i+1$ et $j+1$.*
   On convient que, lorsque les sommets ne sont pas reliés, cette distance vaut -1. La distance du sommet i à lui même est égale à 0.
2. Ecrire une fonction `voisins(mat,i)` d'arguments une matrice des distances `mat` d'un graphe et un sommet i, renvoyant la liste des voisins du sommet i.
3. Ecrire une fonction `degre(mat,i)` d'arguments une matrice des distances `mat` d'un graphe et un sommet i, renvoyant le nombre de voisins du sommet i.
4. Ecrire une fonction `parcours_largeur(mat,s)`, d'arguments une matrice des distances `mat` d'un graphe et un sommet d'origine `s`, renvoyant la liste des sommets visités à partir du sommet d'origine au cours du parcours en largeur.

Chapitre 6

Bases de données

Introduction : une **base de données** est un ensemble d'informations qui est organisé de manière à être facilement accessible, géré et mis à jour. Elle est donc utilisée comme méthode de stockage, de gestion et de récupération de l'information. Les bases de données se retrouvent partout : les répertoires téléphoniques (nom, téléphone, adresse, mail, ...), les fiches de bibliothèque (auteur, titre, année, ...), les carnets de commandes (client, article, quantité, date, prix, ...), le registre de recensement national (nom, prénom, âge, ...), ...

I. Notion de bases de données

A. Modélisation

Les bases de données se modélisent facilement à l'aide de tableaux, appelé **tables** ou **relations**, dans lesquels :

- les colonnes sont appelées **attributs** et correspondent à des catégories d'information (prénom, nom, auteur, titre, année, ...)
- et les lignes sont appelées **enregistrements** et correspondent aux caractéristiques d'un élément (un individu du répertoire, une fiche de bibliothèque, une commande, ...).

La notion de table est apparue dans les années 1970 avec l'algèbre relationnelle, qui est une théorie mathématique en relation avec la théorie des ensembles, ayant pour but d'éclaircir et de faciliter l'utilisation d'une base de données. Cette théorie mathématiques est hors programme.
Voici un exemple de base de données qui nous servira tout au long de ce cours.

eleve					
prenom	nom	filiere	numero	lycee.origine	note.bac
Mathilde	Dufour	PCSI	2	Jean Zay	18
Léa	Dupond	MPSI	2	Pothier	14
Paul	Dugommier	PCSI	1	Pothier	12
Mathilde	Dugommier	MPSI	1	Jean Zay	14
Clément	Durand	PCSI	1	Claude de France	13

Attention, les attributs comme les enregistrements ne sont pas ordonnés a priori. Ainsi, nous ne parlerons pas de l'attribut 1 mais de l'attribut prenom.

L'ensemble des attributs de la table forment un uplet appelé **schéma de table** (ou **schéma relationnel**). On impose que chaque attribut apparaisse de manière unique dans la table afin de garantir une bonne lisibilité de la base de données. A chaque attribut on associe un **domaine** constitué de l'ensemble des valeurs que peut prendre cet attribut : entier, flottant, chaîne de caractères. Dans notre exemple, les attributs *prenom*, *nom*, *filiere* et *lycee.origine* ont pour domaine les chaînes de caractères. Il est tout à fait possible de spécifier ces domaines en précisant, par exemple, que le domaine de l'attribut *filiere* est réduit à l'ensemble des chaînes de caractères dénotant une filière de classe préparatoire $\{\text{MPSI}, \text{PCSI}, \text{PTSI}, ...\}$.

B. Clé primaire

Afin d'optimiser l'utilisation de la base de données et de limiter les potentielles confusions quant à sa lecture, il est préférable que les enregistrements d'une même table soient distincts deux à deux. Autrement dit, deux lignes distinctes de la table doivent différer d'au moins un attribut. Pour garantir la non répétition des enregistrements, on introduit le concept de clé lors de la création de la base de données.
On appelle **clé** un sous-ensemble d'attributs X tel que si deux enregistrements sont égaux sur les attributs de X alors il sont égaux partout. Dans notre exemple, les ensembles $\{prenom\}$ et $\{nom\}$ ne sont pas des clés car deux lignes différentes ont pour attribut *prenom* Mathilde et deux lignes différentes ont pour attribut *nom* Dugommier. En revanche, l'ensemble $\{prenom, nom\}$ est une clé.
On dira qu'une clé est **minimale** si l'ensemble X d'attribut la définissant est minimal pour l'inclusion. Autrement dit, si on considère un ensemble d'attribut X' obtenu à partir de X lui retirant un attribut quelconque, l'ensemble X' obtenu n'est plus une clé. Dans notre exemple, $\{\text{prenom}, \text{nom}\}$ est donc une clé minimale.
La **clé primaire** est alors choisie par le créateur de la base de données parmi les clés dites **minimales**. Le choix d'une telle clé permet une indexation des données à l'aide de cette clé seule et renforce l'efficacité des procédures d'interrogation de la table.
En pratique, après insertion de données supplémentaires, il se pourrait que la clé primaire choisie ne soit plus une clé primaire pour la nouvelle table. Dans notre exemple, on pourrait voir arriver une nouvelle élève Léa Dupond qui rendrait non primaire la clé $\{prenom, nom\}$. Afin d'éviter cet écueil, on introduit très souvent un attribut supplémentaire (commençant généralement par id) dans la table qui sera un entier incrémenté à chaque ajout d'un enregistrement. Cette attribut constituera alors une clé primaire pérenne.
Modifions notre exemple avec l'ajout d'une telle clé primaire, que l'on indiquera en soulignant les attributs la constituant.

eleve						
<u>id_eleve</u>	prenom	nom	filiere	numero	lycee.origine	note.bac
1	Mathilde	Dufour	PCSI	2	Jean Zay	18
2	Léa	Dupond	MPSI	2	Pothier	14
3	Paul	Dugommier	PCSI	1	Pothier	12
4	Mathilde	Dugommier	MPSI	1	Jean Zay	14
5	Clément	Durand	PCSI	1	Claude de France	13

II. Langage SQL

Quel que soit le logiciel utilisé pour créer la base de données, il est possible d'y effectuer des recherches grâce au langage **SQL** (Structured Query Langage). Ce langage reprend la structure de l'algèbre relationnelle en y ajoutant des moyens de calculs et autres améliorations (ordonnancement des résultats, par exemple). Il est très proche du langage humain (anglais).

A. Requêtes de base

Usuellement les mots clés de SQL sont écrits en majuscule (mais ce n'est pas obligatoire). Toutes les requêtes commencent par le mot clé SELECT.

- **Projection :** on garde les attributs A1, ..., Ap.

  SELECT A1,...,Ap FROM table

  Pour garder tous les attributs, on écrit *.
  Le mot clé FROM permet de spécifier le nom de la table à utiliser.

Projection de *prenom*

```
>>> SELECT prenom FROM eleve
```

prenom
Mathilde
Léa
Paul
Mathilde
Clément

Projection de *prenom* sans doublon

```
>>> SELECT DISTINCT prenom FROM eleve
```

prenom
Mathilde
Léa
Paul
Clément

- **Sélection :** on utilise les comparateurs =, ! =, <, ... et les connecteurs logiques *AND*, *OR*, *NOT*.

  SELECT A1,...,Ap FROM table WHERE Ai=$\cdots$

Sélection de *prenom* en MPSI

```
>>> SELECT prenom FROM eleve WHERE filiere=MPSI
```

prenom
Léa
Mathilde

- **Renommage :**

SELECT A1, . . . , A4, A5 AS a5, . . . , Ap AS ap FROM table

Renommage : notes sur 10

```
>>> SELECT prenom, nom, note.bac/2 AS note.sur.10 FROM eleve
```

prenom	nom	note.sur.10
Mathilde	Dufour	9
Léa	Dupond	7
Paul	Dugommier	6
Mathilde	Dugommier	7
Clément	Durand	6.5

- **Opérations entre attributs :** il est possible de réaliser des opérations entre attributs, par exemple

SELECT A1 + A2 + A3 FROM table

- **Opérations ensemblistes :** on peut combiner plusieurs requêtes à l'aide des mots-clés **UNION**, **INTERSECT** ou **EXCEPT** réalisant l'union, l'intersection et la différence de deux ensembles. Pour ce faire, il faut que **les tables aient des attributs coïncidant deux à deux**.

  Dans le cas où il existerait deux tables *eleve.21.22* et *eleve.22.23* donnant la liste des élèves de deuxième année du lycée sur deux années consécutives, nous pourrions récupérer la liste des 5/2 en faisant :

SELECT * FROM eleve.21.22 INTERSECT SELECT * FROM eleve.22.23

B. Affichage des résultats

Voici quelques requêtes pour agencer l'affichage des résultats.

- **ORDER BY :** couplé à une expression arithmétique des attributs, ordonne les résultats.

SELECT . . . FROM table ORDER BY . . .

Classement par note.bac ascendantes

```
>>> SELECT * FROM eleve ORDER BY note.bac
```

eleve						
id_eleve	prenom	nom	filiere	numero	lycee.origine	note.bac
3	Paul	Dugommier	PCSI	1	Pothier	12
5	Clément	Durand	PCSI	1	Claude de France	13
2	Léa	Dupond	MPSI	2	Pothier	14
4	Mathilde	Dugommier	MPSI	1	Jean Zay	14
1	Mathilde	Dufour	PCSI	2	Jean Zay	18

Classement par note.bac descendantes

```
>>> SELECT * FROM eleve ORDER BY note.bac DESC
```

eleve						
id_eleve	prenom	nom	filiere	numero	lycee.origine	note.bac
1	Mathilde	Dufour	PCSI	2	Jean Zay	18
2	Léa	Dupond	MPSI	2	Pothier	14
4	Mathilde	Dugommier	MPSI	1	Jean Zay	14
5	Clément	Durand	PCSI	1	Claude de France	13
3	Paul	Dugommier	PCSI	1	Pothier	12

S'il y a plusieurs expressions après ORDER BY, séparées par des virgules, les données sont triées pour l'ordre lexicographique (on compare la première expression, puis en cas d'égalité la deuxième, etc ...)

Classement par note.bac et ordre alphabétique

```
>>> SELECT * FROM eleve ORDER BY note.bac, nom
```

eleve						
id_eleve	prenom	nom	filiere	numero	lycee.origine	note.bac
3	Paul	Dugommier	PCSI	1	Pothier	12
5	Clément	Durand	PCSI	1	Claude de France	13
4	Mathilde	Dugommier	MPSI	1	Jean Zay	14
2	Léa	Dupond	MPSI	2	Pothier	14
1	Mathilde	Dufour	PCSI	2	Jean Zay	18

- **LIMIT :** permet de limiter le nombre total $n \in \mathbb{N}$ de résultats affichés.

$$\text{SELECT} \ldots \text{ FROM table LIMIT } n$$

Les trois premiers résultats

```
>>> SELECT * FROM eleve LIMIT 3
```

eleve						
id_eleve	prenom	nom	filiere	numero	lycee.origine	note.bac
1	Mathilde	Dufour	PCSI	2	Jean Zay	18
2	Léa	Dupond	MPSI	2	Pothier	14
3	Paul	Dugommier	PCSI	1	Pothier	12

- **OFFSET :** permet d'ignorer les $p \in \mathbb{N}$ premiers résultats.

$$\text{SELECT} \ldots \text{ FROM table OFFSET } p$$

Deux résultats seulement

```
>>> SELECT * FROM eleve OFFSET 3
```

eleve						
id_eleve	prenom	nom	filiere	numero	lycee.origine	note.bac
4	Mathilde	Dugommier	MPSI	1	Jean Zay	14
5	Clément	Durand	PCSI	1	Claude de France	13

Remarque : il peut être intéressant de coupler les instructions LIMIT et OFFSET (cf https ://sql.sh/cours/limit).

C. Fonctions d'agrégations

Les **fonctions d'agrégation** dans le langage SQL permettent d'effectuer des opérations statistiques sur un ensemble d'enregistrement.

- **COUNT :** compte le nombre de lignes.

SELECT COUNT(attribut) FROM table ;

Nombre de lignes dans la table

```
>>> SELECT Count(prenom) FROM eleve
```

COUNT(prenom)
5

- **MAX, MIN :** renvoie respectivement la valeur maximale et minimale de l'attribut demandé.

SELECT MAX(attribut) FROM table

Maximum de note.bac

```
> SELECT MAX(note.bac) FROM eleve
```

MAX(note.bac)
18

Minimum de note.bac

```
> SELECT MIN(note.bac) FROM eleve
```

MIN(note.bac)
12

- **SUM :** permet de sommer les éléments d'une colonne attribut.

SELECT SUM(attribut) FROM table

Somme de note.bac

```
>>> SELECT SUM(note.bac) FROM eleve
```

SUM(note.bac)
71

- **AVG :** permet de calculer la moyenne des éléments d'une colonne attribut.

SELECT AVG(attribut) FROM table

Moyenne de note.bac

```
>>> SELECT AVG(note.bac) FROM eleve
```

AVG(note.bac)
14.2

D. Agrégats

Toutes ces fonctions prennent tout leur sens lorsqu'elles sont utilisées avec la commande GROUP BY qui permet de filtrer les données sur une ou plusieurs colonnes.

- **GROUP BY :** indique sur quel attribut effectuer les agrégations.

SELECT agreg(attribut) FROM table GROUP BY ...

Moyenne de note.bac par filière

```
>>> SELECT AVG(note.bac)
     FROM eleve GROUP BY filiere
```

AVG(note.bac)
14.3
14

Filière et moyenne de note.bac par filière

```
>>> SELECT filiere,AVG(note.bac)
     FROM eleve GROUP BY filiere
```

filiere	AVG(note.bac)
PCSI	14.3
MPSI	14

- **HAVING :** permet d'effectuer une sélection sur des agrégats.

SELECT agreg(attribut) FROM table GROUP BY ... HAVING ...

Moyenne de note.bac >14

```
>>> SELECT filiere, AVG(note.bac)
     FROM eleve GROUP BY filiere
     HAVING AVG(note.bac)>14
```

filiere	AVG(note.bac)
PCSI	14.3

Moyenne de note.bac >14

```
>>> SELECT filiere,
     AVG(note.bac) AS moy
     FROM eleve GROUP BY filiere
     HAVING moy>14
```

filiere	AVG(note.bac)
PCSI	14.3

Remarque 1. *Les rôles des mots-clés WHERE et HAVING sont similaires. On utilisera WHERE en amont (avant) d'un agrégat et HAVING en aval (après).*

E. Composition de requêtes

Il est bien sûr possible de combiner tous les types de requêtes vus précédemment.

Qui a eu la plus haute note.bac ?

```
>>> SELECT * FROM eleve
WHERE note.bac=(SELECT MAX(note.bac) FROM eleve)
```

eleve						
id_eleve	prenom	nom	filiere	numero	lycee.origine	note.bac
1	Mathilde	Dufour	PCSI	2	Jean Zay	18

III. Tables multiples

Pour éviter la redondance d'informations, il est en général préférable d'organiser les données en différentes tables plutôt qu'en une unique base de données.
Modifions notre exemple en considérant cette fois trois tables comme suit.

eleve				
id.eleve	prenom	nom	id.classe	id.lycee
1	Mathilde	Dufour	834	1
2	Léa	Dupond	832	2
3	Paul	Dugommier	833	2
4	Mathilde	Dugommier	831	1
5	Clément	Durand	833	3

lycee	
id.lycee	nom.lycee
1	Jean Zay
2	Pothier
3	Claude de France

classe		
id.classe	filiere	numero
831	MPSI	1
832	MPSI	2
833	PCSI	1
834	PCSI	2

A. Produit cartésien

Il est possible d'effectuer un produit cartésien en langage SQL. Pour ce faire, on énumère les différentes tables dans la clause FROM déparés par **JOIN**. On obtient alors une table contenant toutes les colonnes de chaque table dont on a fait le produit.

Produit cartésien des trois tables

```
>>> SELECT * FROM eleve JOIN classe JOIN lycee
```

eleve × classe × lycee									
id_eleve	prenom	nom	id.classe	id.lycee	id.classe	filiere	numero	id.lycee	nom.lycee
1	Mathilde	Dufour	834	1	831	MPSI	1	1	Jean Zay
1	Mathilde	Dufour	834	1	831	MPSI	1	2	Pothier
1	Mathilde	Dufour	834	1	831	MPSI	1	3	Claude de France
1	Mathilde	Dufour	834	1	832	MPSI	2	1	Jean Zay
1	Mathilde	Dufour	834	1	832	MPSI	2	2	Pothier
1	Mathilde	Dufour	834	1	832	MPSI	2	3	Claude de France
...	...	...	...	...	...	...	...	...	...

nom et _prenom_ des élèves venant de Jean Zay

```
>>> SELECT e.prenom, e.nom FROM eleve AS e JOIN lycee AS L
    WHERE e.id.lycee=L.id.lycee AND L.id.lycee="1"
```

Jean Zay	
prenom	nom
Mathilde	Dufour
Mathilde	Dugommier

nom et _prenom_ des élèves de MPSI 1 venant de Jean Zay

```
>>> SELECT e.prenom, e.nom FROM eleve AS e JOIN lycee AS L JOIN classe AS c
    WHERE e.id.lycee=L.id.lycee AND c.id.classe=e.id.classe
    AND L.nom.lycee="Jean Zay" AND c.filiere="MPSI" AND c.numero="1"
```

Jean Zay	
prenom	nom
Mathilde	Dugommier

Le langage SQL considère les noms des colonnes de la table T sous la forme T.nom, il ne peut donc pas y avoir d'attributs au nom identique après produit cartésien. Si deux attributs ont le même nom dans deux tables différentes, il faudra penser à spécifier leur table d'origine dans chaque requête. Le plus simple est de renommer les attributs aux noms identiques pour lever toute ambiguïté et écourter l'écriture des requêtes.

B. Clés étrangères

On appelle **clé étrangère** d'une table un attribut lié à la clé primaire d'une autre table.

Ici, les tables *lycee* et *classe* possèdent toutes deux une clé primaire, respectivement *id.lycee* et *id.classe*. La table *eleve* quant à elle possède des **clés étrangères** : *id.classe* et *id.lycee*, renvoyant aux clés primaires des tables *classe* et *lycee*.

Ces clés étrangères sont nécessaires pour effectuer les *jointures*, que nous allons étudier maintenant.

C. Jointure

Dans le produit cartésien, un grand nombre de lignes n'ont pas de réel intérêt car certaines informations se retrouvent en contradiction les unes avec les autres. Il est donc naturel lors du produit d'identifier les colonnes des attributs de même nom et de ne sélectionner que les lignes cohérentes : on parle de **jointure naturelle** et on utilise le mot-clé **NATURAL JOIN**.

Jointure naturelle des trois tables

```
>>> SELECT * FROM eleve NATURAL JOIN classe NATURAL JOIN lycee
```

eleve × classe × lycee							
id_eleve	prenom	nom	id.classe	id.lycee	filiere	numero	nom.lycee
1	Mathilde	Dufour	834	1	PCSI	2	Jean Zay
2	Léa	Dupond	832	2	MPSI	2	Pothier
3	Paul	Dugommier	833	2	PCSI	1	Pothier
4	Mathilde	Dugommier	831	1	MPSI	1	Jean Zay
5	Clément	Durand	833	3	PCSI	1	Claude de France

L'utilisation du mot-clé **NATURAL JOIN** n'est pertinente que si la jointure entre les deux tables ne peut se faire que d'une façon, ce qui n'est pas toujours le cas. On préférera donc plutôt spécifier « à la main » les conditions de jointure avec le mot-clé **ON**.

Jointure des trois tables avec ON

```
>>> SELECT * FROM eleve AS e JOIN classe AS c JOIN lycee AS L
     ON e.id.lycee=L.id.lycee AND e.id.classe=c.id.classe
```

eleve × classe × lycee							
id_eleve	prenom	nom	id.classe	id.lycee	filiere	numero	nom.lycee
1	Mathilde	Dufour	834	1	PCSI	2	Jean Zay
2	Léa	Dupond	832	2	MPSI	2	Pothier
3	Paul	Dugommier	833	2	PCSI	1	Pothier
4	Mathilde	Dugommier	831	1	MPSI	1	Jean Zay
5	Clément	Durand	833	3	PCSI	1	Claude de France

On peut bien sûr ne vouloir sélectionner qu'une partie des résultats obtenus, il nous suffit alors d'ajouter des conditions de sélection avec WHERE.

Jointure pour les élèves en MPSI 1 venant de Jean Zay

```
>>> SELECT * FROM eleve AS e JOIN classe AS c JOIN lycee AS L
    ON e.id.lycee=L.id.lycee AND e.id.classe=c.id.classe
    WHERE e.nom.lycee="Jean Zay" AND c.filiere="MPSI" AND c.numero="1"
```

eleve × classe × lycee							
id_eleve	prenom	nom	id.classe	id.lycee	filiere	numero	nom.lycee
4	Mathilde	Dugommier	831	1	MPSI	1	Jean Zay

Chapitre 7

Programmation dynamique

Introduction La programmation dynamique est un paradigme de programmation introduit initialement au début des années 1950 par Richard Bellman pour résoudre efficacement des problèmes d'optimisation consistant à maximiser une somme de fonctions monotones et croissantes sous contraintes.

I. Explication de la stratégie

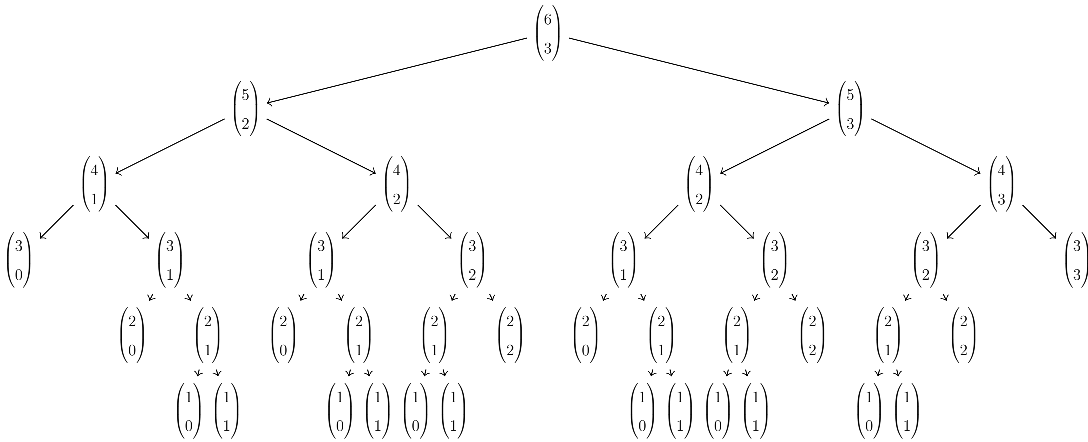
Avant de présenter le paradigme de la programmation dynamique, il est important de comprendre sur quel type de problèmes elle sert et, éventuellement, avec quelles autres stratégies on pourrait résoudre ces problèmes.

A. Arbre de dépendance

On regarde des problèmes dont la solution peut être obtenue à partir des solutions de sous-problèmes de même nature. On peut citer comme exemple les suites récurrentes linéaire d'ordre quelconque (dont le plus grand représentant à l'ordre 2 est la suite de Fibonacci) ou plus simplement les coefficients binomiaux.

Il est alors naturel d'associer à un tel problème un un graphe (potentiellement infini) orienté représentant la relation de dépendance entre les différents sous-problèmes. On peut alors, pour un sommet donné représentant des paramètres de notre problème, considérer l'ensemble des sommets atteignables depuis ce dernier. On obtient ainsi un arbre de dépendance dans lequel figure les sous-problèmes que l'on est susceptible de devoir traiter pour trouver notre solution originale.

Exemple. *On peut observer, sur la figure ci-dessous, l'arbre de dépendance de calcul du coefficient binomial* $\binom{6}{3}$ *:*



B. Stratégies ascendante et descendante

Devant un tel arbre, deux procédés peuvent venir à l'esprit :

— On peut commencer par regarder le haut de l'arbre (la racine) et on se demande de quoi on a besoin pour le calculer. La réponse étant de regarder ses voisins direct, on recommence avec les dits voisins et on descend au fur et à mesure dans l'arbre. Voilà pourquoi on parle de **stratégie descendante**.

  Dans l'exemple précédent, pour calculer $\begin{pmatrix} 6 \\ 3 \end{pmatrix}$ on a besoin de $\begin{pmatrix} 5 \\ 2 \end{pmatrix}$ qui a besoin de $\begin{pmatrix} 4 \\ 1 \end{pmatrix}$ etc....

  C'est un raisonnement très courant (même si pas systématique) en programmation récursive.

— Ou bien, en opposition, on peut commencer par le bas et remonter au fur et à mesure le long des ancêtres. On parlera ici de **stratégie ascendante**.

  On commence par calculer $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ et $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ puis, en remontant, on s'en sert pour calculer $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ puis, également avec $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$, on calcule $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$ etc...

C. La mémoïsation

Une autre chose frappante dans cet exemple des coefficients binomiaux est le nombre de fois où peuvent apparaître le même sommet dans l'arbre de dépendance. Cela n'apparaît pas dans tous les problèmes possédant une telle relation de dépendance, mais lorsque c'est le cas cela nous embête un peu.

Dans l'exemple précédent, cela veut dire que l'on peut potentiellement calculer trois fois $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$.

Evidemment, peut importe que notre stratégie soit ascendante ou descendante, on souhaiterait éviter d'avoir à refaire le calcul intégralement ces trois fois.

La résolution de ce problème lors d'une stratégie descendante se fait à l'aide d'une autre technique de programmation appelée la mémoïsation.
Cela consiste simplement à enregistrer momentanément tout calcul effectué lors de l'exécution du programme principal dans un cache afin, dans le cas ou on aurait de nouveau affaire au même calcul, de pouvoir accéder directement à son résultat sans refaire de calcul intermédiaire. Il y a derrière ce terme mémoïsation l'idée de résultats de calculs dont il faut se souvenir.

Remarque. *A l'origine, la programmation dynamique consiste à utiliser une approche ascendante. L'approche descendante, couplée à la mémoïsation, est autre chose.*

II. Exemples concrets et classiques

A. La suite de Fibonacci

Présentons à nouveau la célèbre suite de Fibonacci :

$$u_0 = 0, \quad u_1 = 1, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n$$

Vous avez certainement déjà vu plusieurs programmes capables de calculer le $n^{\text{ième}}$ terme de cette suite (dont peut être, sans savoir, la version programmation dynamique).

Une version récursive (peu efficace) :

Fibonacci récursif

```
def fibo_rec(n) :
    if n==0 or n==1 :
        return n
    else :
        return fibo_rec(n-1) + fibo_rec(n-2)
```

Cette version n'étant clairement pas de la programmation dynamique. En effet, elle ne sauvegarde aucun résultat intermédiaire et voit sa complexité exploser du fait du chevauchement des appels récursifs.

Pour régler cela avec la même stratégie algorithmique, on crée un cache global, sous la forme d'un dictionnaire, pour utiliser la mémoïsation :

Fibonacci récursif avec cache mémoire

```
def fibo_memo(n) :
    dico = {}
    def boucle(k) :
        if k in dico :
            return dico[k]
        elif k==0 or k==1 :
            dico[k] = k
            return dico[k]
        else :
            dico[k] = boucle(k-1)+boucle(k-2)
            return dico[k]
    return boucle(n)
```

Enfin pour utiliser une stratégie ascendante, il faut créer une structure pour stocker tous les termes de la suite en commençant par les deux premiers jusqu'au terme de rang n :

Fibonacci en programmation dynamique

```
def fibo(n) :
    L=[0,1]
    for i in range(1,n) :
        L.append(L[i-1]+L[i])
    return L[n]
```

B. Les coefficients binomiaux

Reprenons l'exemple ayant servi à illustrer le concept d'arbre de dépendance.
Une stratégie de programmation récursive pour le résoudre consisterait à calculer l'intégralité du triangle de Pascal en partant de la première ligne jusqu'à arriver au terme nous intéressant :

Calcul des coefficients binomiaux par programmation dynamique

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

On initialise un triangle (sous la forme d'une liste de listes dont les sous-listes sont de tailles croissantes) ne contenant que des « 1 », puis on modifie chaque case de l'intérieur grâce à la formule de Pascal.

C. Le problème du sac à dos

Voici, enfin, un problème d'optimisation tout aussi classique se résolvant par programmation dynamique mais que vous n'avez probablement pas vu ailleurs.
Le problème du sac à dos consiste à se donner un sac (à dos) ayant une capacité (un poids ou une contenance) maximale que l'on ne peut dépasser en le remplissant. On dispose également d'un ensemble d'éléments $\{x_0, x_1, \dots, x_{n-1}\}$ ayant chacun un entier correspondant (un poids ou un volume). Notre but est alors de maximiser la capacité du sac que l'on utilise, en mettant des éléments dedans sans en dépasser sa capacité maximale.

On donnera donc en entrée à notre algorithme un entier `M` représentant la capacité maximale du sac et une liste `L` donc l'élément en $i^{\text{ième}}$ position `L[i]` représentera la capacité utilisée par le $i^{\text{ième}}$ élément de notre ensemble x_i.
On va alors définir le sous problème de taille $p \times q$, pour $p \in [\![0, M]\!]$ et $q \in [\![0, n]\!]$, comme étant le même problème du sac à dos mais avec un sac de capacité maximale p et en se restreignant aux q premier objet de notre ensemble $\{x_0, x_1, \dots, x_{q-1}\}$.

L'idée de notre programme est d'exploiter une relation de récurrence entre le problème de taille $p \times q$ et les problèmes de taille inférieur. Le raisonnement est le suivant : la solution du problème de taille $p \times q$ va :

— soit ne pas contenir le $q^{\text{ième}}$ élément et donc est constituée d'une solution avec les $q-1$ premiers éléments pour un sac de même capacité maximale p donc est égale à la solution du problème de taille $p \times (q-1)$.

— soit contenir le $q^{\text{ième}}$ élément et donc est constituée d'une solution avec les $q-1$ premiers éléments pour un sac de capacité maximale à laquelle on a retranché l'entier correspondant au $q^{\text{ième}}$.

Sachant, qu'initialement, on a les solutions des problèmes de taille $0 \times q$ et $p \times 0$ très facilement puisque dans les deux cas on ne peux rien mettre dans le sac à dos.
On va donc stocker les solutions de nos sous-problèmes dans une matrice de taille $(M+1)\times(n+1)$ en initialisant la première ligne et la première colonne avec des 0. Puis on remplira cette matrice de haut en bas et de gauche à droite.

L'algorithme du sac à dos en programmation dynamique

```
def sac_a_dos(M,L) :
    n=len(L)
    out = [[0 for q in range(n+1)] for p in range(M+1)]
    for q in range(1,n+1) :
        for p in range(1,M+1) :
            if L[q-1] <= p and L[q-1]+out[p-L[q-1]][q-1] > out[p][q-1] :
                out[p][q] = L[q-1]+out[p-L[q-1]][q-1]
            else :
                out[p][q] = out[p][q-1]
    return out[M][n]
```

III. Un exemple classique mais plus technique : l'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme qui permet pour un graphe pondéré donné (orienté ou non) de donner la distance minimale entre tout couple de sommet.
Pour ce faire, si on se donne le graphe $G=(S,A)$ d'ordre n (de sommets numérotés $1,...,n$), l'idée est de calculer une suite de $n+1$ matrices $(M_p)_{0\leqslant p\leqslant n}$ de coefficients $m_{i,j}^{(p)}$. Le coefficient $m_{i,j}^{(p)}$, situé en $i^{\text{ième}}$ ligne et $j^{\text{ième}}$ colonne de la matrice M_p est égal à la distance minimale du sommet i au sommet j en ne s'autorisant à passer que par des sommets intermédiaires appartenant à l'ensemble $\{1,...,p\}$.
Cette suite de matrices est, en effet, facile à initialiser, puisque passer du sommet i au sommet j en ne passant par aucun sommet intermédiaire ne peut se faire que s'il existe un arc de i vers j et on connaît donc immédiatement le poids du chemin de distance minimal puisqu'il s'agit de la pondération du dit arc.
On dispose ensuite d'une relation de récurrence pour calculer M_{p+1} en fonction de M_p. En effet, il n'y a que deux possibilité pour un chemin de poids minimal allant du sommet i au sommet j qui ne passe que par les $p+1$ premiers sommets :

— soit il passe par le sommet $p+1$ auquel cas il est la concaténation d'un chemin de distance minimal du sommet i au sommet $p+1$ qui ne passe que par les p premiers sommets avec un chemin de distance minimal du sommet $p+1$ au sommet j qui ne passe que par les p premiers sommets. Et on aura donc juste à additionner les coefficients $m_{i,p+1}^{(p)}$ et $m_{p+1,j}^{(p)}$ pour avoir $m_{i,j}^{(p+1)}$.

— soit il ne passe pas par le sommet $p+1$ et donc son poids est égal au coefficient $m_{i,j}^{(p)}$.

On a donc juste à considérer le minimum entre ces deux quantités citées pour avoir le coefficient $m_{i,j}^{(p+1)}$.

Pour la rédaction de notre algorithme, on va supposer que notre graphe G est donné par sa matrice d'adjacence (nommée g dans le programme) avec des 0 sur la diagonale et en coefficient i,j le poids de l'éventuel arc du sommet i au sommet j et des -1 (pour représenter un $+\infty$) s'il n'existe pas d'arc entre les deux sommets i et j. Une telle matrice d'adjacence correspond exactement à la matrice M_0. Si le graphe n'était pas donné sous cette forme, il faudrait simplement calculer préalablement cette dernière.

Algorithme de Floyd-Warshall

```
def Floyd_Warshall(g) :
    n = len(g)
    out = [[[0 for j in range(n)] for i in range(n)] for p in range (n+1)]
    out[0] = g
    for p in range(1,n+1) :
        for i in range(n) :
            for j in range(n) :
                if out[p-1][i][j] < out[p-1][i][p-1] + out[p-1][p-1][j] :
                    out[p][i][j] = out[p-1][i][j]
                else :
                    out[p][i][j] = out[p-1][i][p-1] + out[p-1][p-1][j]
    return out[n]
```

Chapitre 8

Intelligence artificielle

Introduction. L'intelligence artificielle (IA) est « l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine ». C'est une définition très floue qui englobe tout un ensemble de concepts et de technologies. On peut, par exemple, y inclure la quasi intégralité de l'algorithmique ou encore la logique mathématique. Des instances, telle la CNIL, notant le peu de précision de la définition, ont présenté l'IA comme « le grand mythe de notre temps ».

I. L'apprentissage automatique

L'intelligence artificielle étant un vaste domaine, il n'est pas question de présenter ce concept dans sa globalité. Le but de ce cours est plutôt d'introduire deux approches algorithmiques référencées comme étant de l'apprentissage automatique, censées imiter un raisonnement humain.

La première, l'**apprentissage supervisé** (*supervised learning* en anglais), consiste à construire une fonction de prédiction à partir d'exemples annotés. Pour ce faire, on se donne un échantillon d'entrées possibles pour lesquelles on connaît le résultat attendu. Puis, grâce à cet échantillon, l'algorithme doit prédire le résultat d'une nouvelle entrée.

Exemple (Apprentissage supervisé)**.**

- *On dispose de différentes classes d'iris (setosa, versicolor, virginica) et de données déjà classées.*
- *On se donne un nouvel élément, ici une nouvelle fleur d'iris, et on veut la classer, c'est-à-dire déterminer la classe à laquelle elle appartient.*

La seconde, l'**apprentissage non supervisé**, consiste à se donner un jeu de données. Il s'agit alors de découvrir des structures sous-jacentes à ces données.

Exemple (Apprentissage non supervisé)**.**

- *On dispose d'éléments non classés, par exemple un ensemble d'animaux.*
- *On veut les regrouper en classes, par exemple par espèce.*

Remarque. *Noter que la différence entre ces deux apprentissages ne se fait pas uniquement sur l'existence de l'échantillon originel. Dans l'apprentissage non supervisé les différentes classes ne sont même pas prédéfinies.*

Les données dont on dispose peuvent être quantitatives ou qualitatives. Dans le cas quantitatif, on considère que les problèmes de prédiction sont des **problèmes de régression**. Tandis que les problèmes de prédiction de variables qualitatives sont des **problèmes de classification**.

II. Apprentissage supervisé : algorithme des k plus proches voisins (*kNN en anglais, pour k nearest neighbours*)

A. Principe de l'algorithme

Définition II..1

❍ Soit k un entier strictement positif, et $(x_i, c_i)_{1 \leqslant i \leqslant n}$ une base d'apprentissage où l'entier naturel $c_i \in \{1, 2, ..., C\}$ désigne la classe associée à chaque observation $x_i \in \mathbb{R}^d$. L'algorithme des k plus proches voisins (aussi appelé algorithme kNN) associe à une nouvelle observation $x \in \mathbb{R}^d$ la classe majoritaire parmi les classes des k éléments de la famille $(x_i)_{1 \leqslant i \leqslant n}$ les plus proches de x.

Remarques :

➪ Cet algorithme dépend de deux choix, qui peuvent influer fortement sur les performances : l'entier k (voir plus bas les matrices de confusion pour bien le choisir) et la définition de la distance entre observations ; pour cette dernière, le programme se limite à la distance euclidienne, mais d'autres choix peuvent, au cas par cas, s'avérer plus efficaces.

➪ On a une disjonction de cas selon la nature du problème traité :

- Si nous avons affaire à un problème de classification, le résultat est une classe d'appartenance. Notre sortie sera alors la classe majoritaire des k plus proches voisins.
- Si nous avons affaire à un problème de régression, la valeur de sortie est la moyenne des valeurs des k plus proches voisins.

Propriétés II..2

Concrètement, l'algorithme se découpe naturellement en trois étapes :

1. calculer la distance $d_i = \|x - x_i\|$ de x à chaque observation x_i $(1 \leqslant i \leqslant n)$;
2. trier par ordre croissant ces distances d_i pour obtenir une liste triée par ordre croissant $[d_{\sigma(1)}, d_{\sigma(2)}, ..., d_{\sigma(n)}]$ et retenir l'ensemble $I = \{\sigma(1), \sigma(2), ..., \sigma(k)\}$ des k premiers indices (ou bien retenir la classe des k premiers indices).
3. déterminer la classe majoritaire parmi les $(c_i)_{i \in I}$ (s'il n'y a pas de classe majoritaire, on peut départager les ex-æquo au moyen d'un autre critère, par exemple en pondérant les classes des différents voisins en fonction de leur rang).

B. Un exemple classique de classification

➪ Description du problème étudié

Considérons l'exemple de la classification d'une fleur de type iris en trois espèces (qui seront les classes considérées) : setosa, versicolor et virginica. Chaque observation est constituée de deux nombres (la longueur et la largeur des pétales), et peut donc être vue comme un point x dans $\mathbb{R}^2$. La figure suivante montre la répartition de ces points par classe pour la base d'apprentissage.

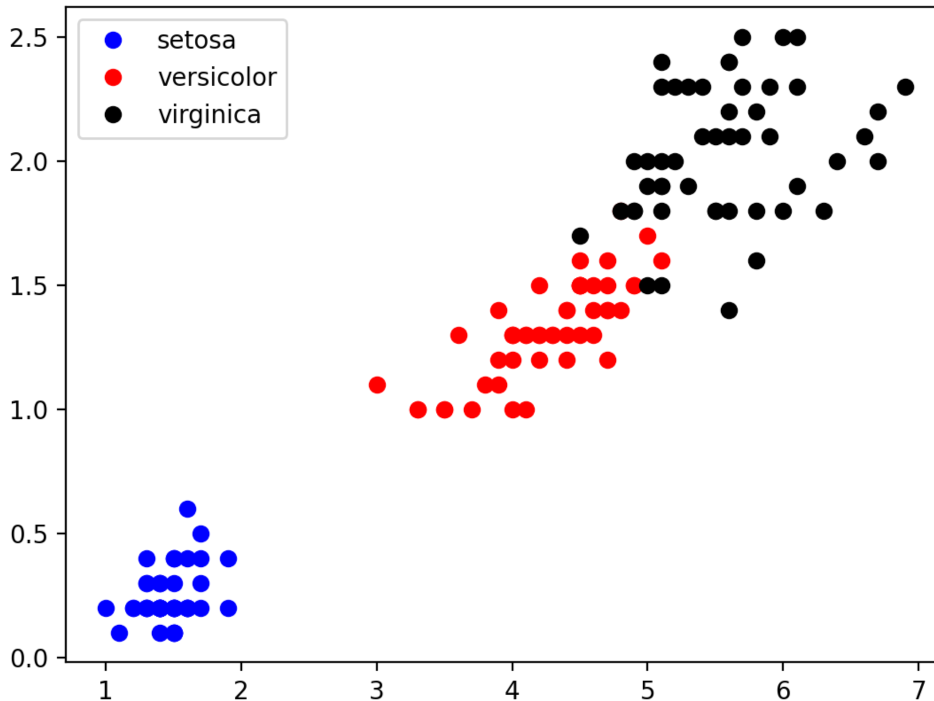


FIGURE 8.1: Données de la base *iris* : chaque fleur de la base (contenant 50 échantillons de chaque classe) est représentée par un point (longueur des pétales en abscisse, largeur en ordonnée) dont la couleur correspond à l'espèce (setosa, versicolor ou virginica).

On remarque que les trois classes ont naturellement tendance à créer des amas dans le plan des observations. L'algorithme kNN va justement exploiter cette propriété en utilisant le principe des classes majoritaires, pour prédire la classe d'une nouvelle observation.

➪ Description de l'algorithme kNN dans ce cas

Commençons par implémenter la distance euclidienne.

Distance euclidienne dans le plan

```
def distance(p1,p2) :
    x1,y1=p1
    x2,y2=p2
    return sqrt((x1-x2)**2+(y1-y2)**2)
```

Nous aurons également besoin d'un algorithme de tri, adapté pour trier une liste de couples par rapport à la première composante.

Tri fusion

```
def fusion(L1,L2) :
    n1,n2=len(L1),len(L2)
    out=[]
    i1,i2=0,0
    for i in range(n1+n2) :
        if i2==n2 or ( i1<n1 and L1[i1][0]<L2[i2][0] ) :
            out.append(L1[i1])
            i1 += 1
        else :
            out.append(L2[i2])
            i2 += 1
    return out

def tri_fusion(L) :
    n=len(L)
    if n<2 :
        return L
    else :
        m=n//2
        return fusion(tri_fusion(L[:m]),tri_fusion(L[m:]))
```

On obtient alors l'algorithme suivant (on supposera que la liste d'observations L est une liste constituée de triplets de la forme $[a, b, c]$, où a et b sont les coordonnées du point dans le plan et c est l'étiquette de ce point (c'est-à-dire la variété d'iris correspondante).

Algorithme des k plus proches voisins

```
def kNN_iris(L,x,k) :
    liste_dist=[(distance(x,e[0:2]),e[2]) for e in L]
    liste_dist=tri_fusion(liste_dist)
    compt=[0,0,0]
    for i in range(k) :
        if liste_dist[i][1]=='Iris-setosa' :
            compt[0] += 1
        elif liste_dist[i][1]=='Iris-versicolor' :
            compt[1] += 1
        else :
            compt[2] += 1
    if compt[0]==max(compt) :
        return'Iris-setosa'
    elif compt[1]==max(compt) :
        return'Iris-versicolor'
    else :
        return'Iris-virginica'
```

Question : que faire s'il y a des classes majoritaires ex-æquo ?

C. Évaluation de l'algorithme : matrice de confusion

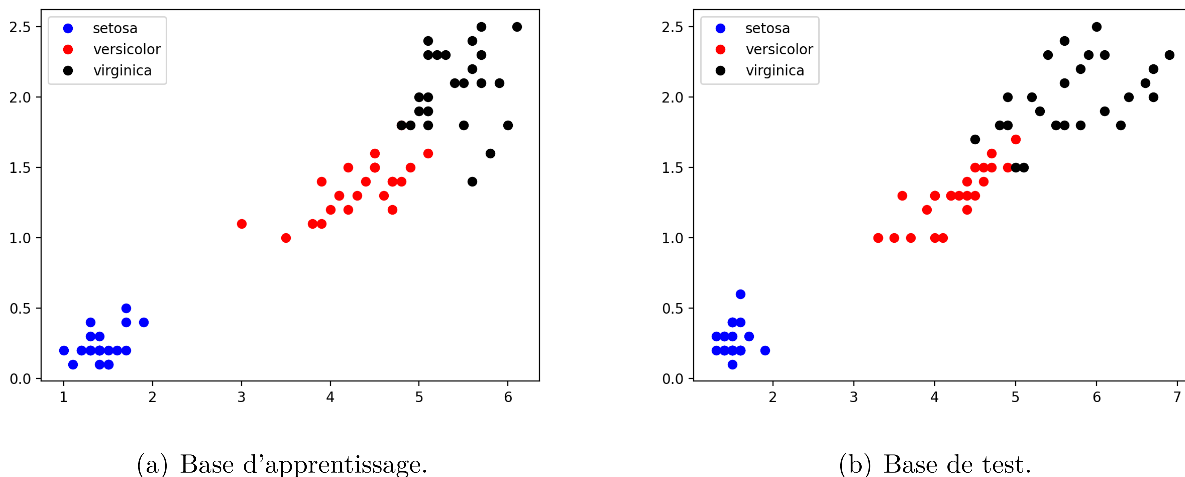
Un algorithme d'apprentissage ne produit pas en général des résultats parfaits, et il est donc important de pouvoir l'évaluer. En particulier, pour l'algorithme de prédiction utilisant la méthode des k plus proches voisins, le choix du paramètre k est loin d'être anodin : si ce dernier est trop petit alors la précision de la prédiction risque d'être faible et au contraire, s'il est trop grand, on aura toujours la même prédiction qui correspondra à la classe majoritaire dans toute la base d'apprentissage.
Pour cette évaluation, on utilise en général une base de test, distincte de la base d'apprentissage, sur laquelle les classes vraies sont également connues. On peut alors évaluer les performances de l'algorithme sur cette base de test en calculant la **matrice de confusion** associée.

Définition II..3 – Matrice de confusion

- La matrice de confusion d'un résultat de classification sur une base de test est une matrice carrée d'ordre C (le nombre de classes) dont le coefficient (i, j) compte le nombre d'observations pour lesquelles la classe vraie est i et la classe estimée est j.

Lorsque la matrice de confusion est **diagonale**, cela signifie que toutes les observations de la base de test ont été correctement classifiées. En pratique, cette situation se produit très rarement, et l'on évalue l'efficacité de l'algorithme de classification en examinant la dispersion des valeurs en dehors de la diagonale.
L'idée sous-jacente est d'**essayer diverses valeurs de** k afin d'obtenir une matrice de confusion qui ressemble le plus possible à une matrice diagonale, ce qui fournira une bonne estimation de la valeur k à choisir pour notre algorithme.

Exemple. *Reprenons l'exemple précédent de la classification d'une fleur de type iris parmi trois espèces. Si nous voulons évaluer l'algorithme kNN sur cet exemple, il nous faut constituer une base de test indépendante de la base d'apprentissage. Pour cela, une façon de procéder est de découper aléatoirement la base initiale de 150 échantillons (50 échantillons par classe) en deux bases (une base d'apprentissage et une base de test) de 25 échantillons par classe (voir figures ci-dessous).*



(a) Base d'apprentissage. (b) Base de test.

Figure 8.2: Découpage des données de la base iris.

Voici, représentée sous forme de tableau ci-dessous, la matrice de confusion correspondant à la classification des échantillons de la base de test au moyen de l'algorithme kNN (avec $k = 5$) associé aux échantillons de la base d'apprentissage. Cette matrice est clairement concentrée sur sa diagonale, ce qui montre que la classification kNN fonctionne très bien sur cet exemple :

	setosa (classés)	versicolor (classés)	virginica (classés)
25 setosa (vrais)	25	0	0
25 versicolor (vrais)	0	24	1
25 virginica (vrais)	0	3	22

III. Apprentissage non supervisé : algorithme des k-moyennes (k-means en anglais)

A. Introduction

Comme nous l'avons vu, la méthode kNN appartient à la famille des algorithmes d'apprentissage supervisé, car on connaît la classe de chaque échantillon servant à l'apprentissage, et l'algorithme cherche à partir de là à construire une fonction capable de prédire la classe de nouveaux échantillons.
L'algorithme des k-moyennes que nous allons présenter dans cette partie n'a pas le même objectif, car il s'agit d'un algorithme d'apprentissage non-supervisé, qui vise justement à partitionner automatiquement en k classes des échantillons donnés au départ.
À titre d'exemple introductif, considérons les 9 points de $\mathbb{R}^2$ représentés sur la figure ci-dessous à gauche. Le partitionnement de ces 9 points en 3 groupes de points proches les uns des autres a une solution naturelle, représentée sur la figure ci-dessous à droite.

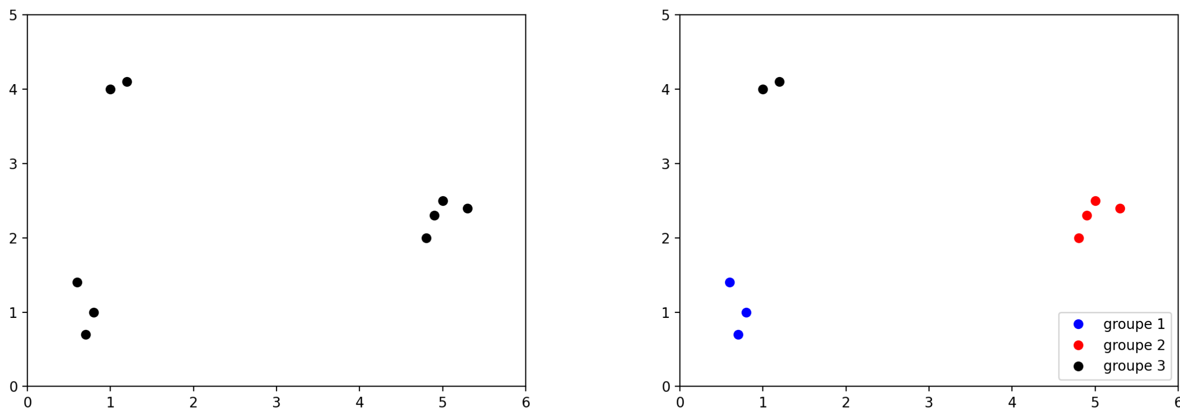


Figure 8.3: Un ensemble de points (à gauche) et leur partitionnement naturel en 3 groupes (à droite).

Pour autant, est-on capable de trouver ce partitionnement par un algorithme automatique ? L'algorithme des k-moyennes défini ci-dessous apporte une réponse positive à cette question.

B. Principe

En toutes généralités, le **partitionnement des k-moyennes** prend en entrée un jeu de données, un entier k et une fonction f. Le problème est alors de diviser les données en k groupes, souvent appelés **clusters**, de façon à minimiser la fonction f. Dans la pratique, on considère la distance d'un point à la moyenne des points de son cluster et la fonction à minimiser est alors la somme des carrés de ces distances.

Malheureusement, pour trouver la solution optimale à ce problème sans hypothèses supplémentaires sur le jeu de données il faudrait tester toutes les possibilités (combien en existe-t-il ?), ce qui est hors de portée des ordinateurs (complexité exponentielle). Le partitionnement des k-moyennes (ou *k-means* en anglais) est une heuristique classique pour ce problème, très utilisé en pratique et considéré comme efficace, mais qui ne garantit ni l'optimalité, ni un temps de calcul polynomial.

Définition III..1 – Algorithme des k-moyennes

- L'algorithme des k-moyennes est une heuristique [a] visant à partitionner un ensemble S de points d'un espace euclidien en k sous-ensembles S_1, S_2,...,S_k non vides tels que la quantité

$$V(S_1, S_2, ..., S_k) = \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \overline{x_i}\|^2, \quad \text{où } \overline{x_i} = \frac{1}{|S_i|} \sum_{x \in S_i} x$$

soit minimale. Il consiste à alterner pour $n = 1, 2, ...$ (jusqu'à convergence) à partir d'un choix initial de barycentres $\overline{x_1}^{(0)}$, $\overline{x_2}^{(0)}$,...,$\overline{x_k}^{(0)}$, les deux étapes suivantes :

1. Étape d'affectation : on calcule une nouvelle partition $S_1^{(n)}$, $S_2^{(n)}$,...,$S_k^{(n)}$ de S en rangeant chaque point $x \in S$ dans le groupe $S_i^{(n)}$ d'indice i pour lequel la distance $d(x, \overline{x_i}^{(n-1)}) = \|x - \overline{x_i}^{(n-1)}\|$ est minimale.
2. Étape de mise à jour des barycentres :

$$\forall i \in [\![1; k]\!], \quad \overline{x_i}^{(n)} = \frac{1}{|S_i^{(n)}|} \sum_{x \in S_i^{(n)}} x.$$

[a] c'est-à-dire une méthode de calcul qui fournit rapidement une solution réalisable et intéressante, mais pas nécessairement optimale ou exacte.

C. Description de l'algorithme

Entrée : S un ensemble fini de points d'un espace euclidien et $k \leqslant |S|$ un entier.

Sortie : Une partition de S en k sous-ensembles (non vides et disjoints) $S_1, \ldots, S_k$.

Après après avoir initialisé une partition aléatoirement, nous ferons évoluer la situation afin que chaque élément de S soit plus proche du barycentre de son sous-ensemble que des autres.

Algorithme des k-moyennes en pseudo-code

```
Partitionner les points de X en k sous-ensembles (non vides) S_1,...,S_k
Tant que la situation évolue faire :
    Calculer les barycentres x̄_1,...,x̄_k des sous-ensembles S_1,...,S_k
    pour chaque point x de S faire
        Trouver i tel que la distance ||x - x̄_i|| soit minimale puis placer x
         dans S_i
Renvoyer la partition S_1,...,S_k
```

Voici quelques remarques (hors programme) sur l'analyse de l'algorithme.

Terminaison : on peut montrer que la quantité $\sum_{i=1}^{k}\sum_{x\in S_i}||x-\overline{x_i}||^2$, qui peut ne prendre qu'un nombre fini de valeurs puisque le nombre de partitions possibles est fini, est strictement décroissante au fur et à mesure des itérations de la boucle `tant que` .

Correction : qui n'a pas vraiment de sens, car cet algorithme utilise une heuristique et il n'y a aucune garantie d'obtenir une solution optimale à la fin. Néanmoins, en général, on a une solution satisfaisante.

Complexité : dans le pire des cas, elle est exponentielle ; cela est dû, encore une fois, au nombre de partitions possibles.

D. Implémentation

On suppose que S est un sous-ensemble de $\mathbb{R}^n$, où n est la taille de chaque échantillon étudié.

On commence par implémenter la distance euclidienne à l'aide d'une fonction `produit_scalaire` prenant en entrées deux listes `x` et `y` codant des vecteurs de $\mathbb{R}^n$.

Produit scalaire et distance

```
def produit_scalaire(x,y) :
    n=len(x)
    out = 0
    for i in range(n) :
        out += x[i]*y[i]
    return out

def distance(x,y) :
    n = len(x)
    soustraction = [x[i]-y[i] for i in range(n)]
    return produit_scalaire(soustraction,soustraction)**0.5
```

Supposons que S soit un ensemble de cardinal $p \in \mathbb{N}$ dont les éléments sont numérotés de 0 à $p-1$ (grâce à une bijection). Dans la suite, nous modéliserons une partition de cet ensemble par une liste `partition` de sous-listes d'éléments de $[\![0, p-1]\!]$. La $i^{\text{ième}}$ sous-liste de `partition` représentant le sous-ensemble S_i sera composée des entiers numérotant les éléments de S_i dans S.

Cette représentation d'une partition nous permet de calculer facilement les barycentres de chaque sous-ensemble S_i.

Calcul du barycentre du sous-ensemble S_i

```
def barycentre(S,num_Si) :
    n=len(S[0])
    pi=len(num_Si)
    out = [0 for j in range(n)]
    for j in range(n) :
        for e in num_Si :
            out[j] += S[e][j]
        out[j] = out[j]/pi
    return out
```

Remarque. *On suppose ici que la variable* **`num_Si`** *contenant les indices du sous-ensemble S_i est non vide, sinon le programme plante.*

Il nous faut également une fonction qui, étant donné la liste des barycentres (appelée `M` dans la fonction ci-dessous, et supposée non vide), nous donne, pour un point donné, l'indice du plus proche d'entre eux.

Plus proche barycentre d'un point x

```
def plus_proche(x,M) :
    imin,min = 0,distance(x,M[0])
    for i in range(1,len(M)) :
        d=distance(x,M[i])
        if d<min :
            imin,min=i,d
    return imin
```

De plus, on a besoin d'une fonction initialisant aléatoirement une partition en k sous-ensembles d'un ensemble S à p éléments.
On utilise le module **random** grâce à l'instruction : **`import random as rd`** .

Fonction d'initialisation

```
def initialisation(p,k) :
    assert p >= k
    L=[i for i in range(p)]
    rd.shuffle(L) #mélange aléatoirement L
    partition=[[] for i in range(k)]
    for i in range(k) :
        partition[i].append(L[i]) #on assure l'existence de k sous-ensembles
    for i in range(k,p) :
        partition[rd.randrange(0,k)].append(L[i])
    return partition
```

Et enfin, il nous faudra tester si deux partitions (ayant le même nombre k de sous-ensembles) sont égales, c'est-à-dire si deux listes de listes sont les mêmes.

Test d'égalité de partition

```
def egal_part(part1,part2,k) :
    assert len(part1)==k and len(part2)==k
    for i in range(k) :
        h=len(part1[i])
        if h != len(part2[i]) :
            return False
        else :
            for j in range(h) :
                if part1[i][j] != part2[i][j] :
                    return False
    return True
```

Remarque. *En toute rigueur, il faudrait tester les égalités des sous-listes à permutation prêt des éléments. Cependant, excepté lors de l'initialisation, les sous-listes des partitions seront naturellement triées dans l'ordre croissant de leurs éléments. Cela ne changera donc pas grand chose.*

On obtient alors l'algorithme suivant dans lequel la condition **"Tant que la situation évolue"** se traduit par un test d'égalité de partitions.

Algorithme des k-moyennes

```
def k_moyennes(S,k) :
    p=len(S)
    assert p >= k
    part1 = initialisation(p,k)
    init = True
    while init or not egal_part(part1,part2,k) :
        if not init :
            part1 = part2
        init = False
        M = [barycentre(S,num_Si) for num_Si in part1]
        part2 = [[] for _ in range(k)]
        for i in range(p) :
            part2[plus_proche(S[i],M)].append(i)
    return part1
```

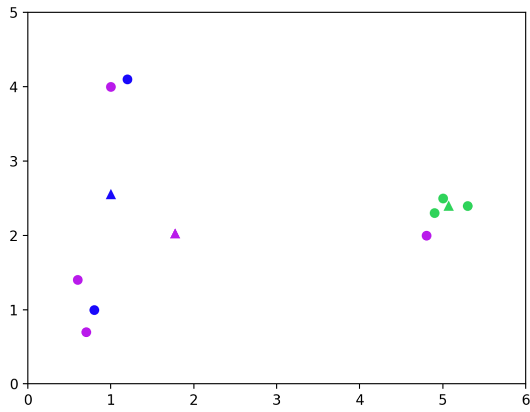
Remarques. • *La variable* `init` *n'a pas vraiment de rôle à jouer dans l'algorithme. Elle permet juste de faire un premier tour de boucle* `while` *sans avoir à construire une première fois la variable* `part2`*.*

• *La condition* ***"Tant que la situation évolue"*** *peut également se traduire par un test portant sur l'évolution de la quantité $V(S_1, ..., S_k)$ (voir TP).*

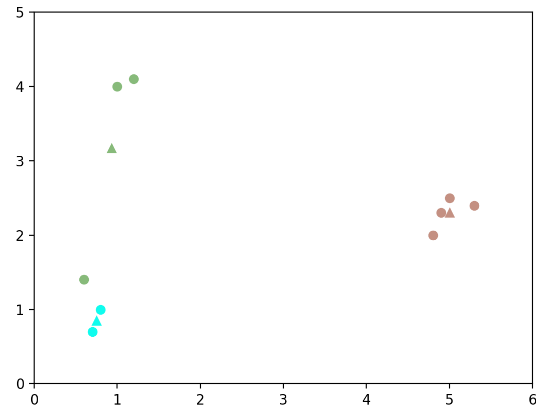
E. Exemples

1. Classement des points de la figure 8.3.

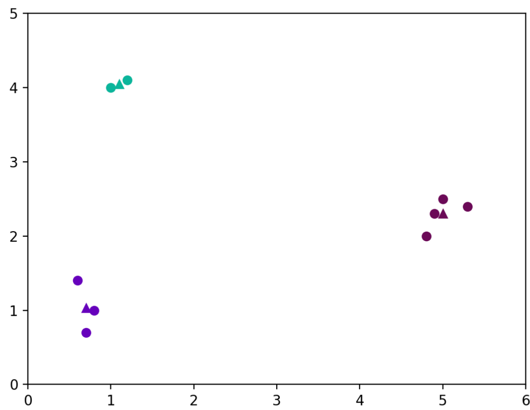
Dans ce cas, on obtient par exemple l'évolution ci-dessous.



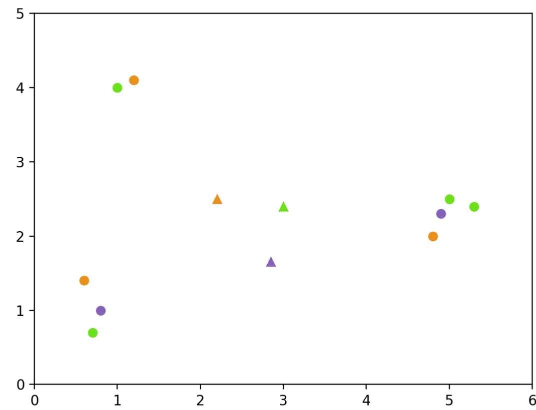
(a) Étape 1 : initialisation.



(b) Étape 2.



(c) Étape 3 : fin de l'évolution.



(d) Un exemple où l'algorithme échoue.

2. Classement des fleurs d'iris.

L'algorithme des k-moyennes appliqué à l'exemple des iris donne les graphiques suivants :

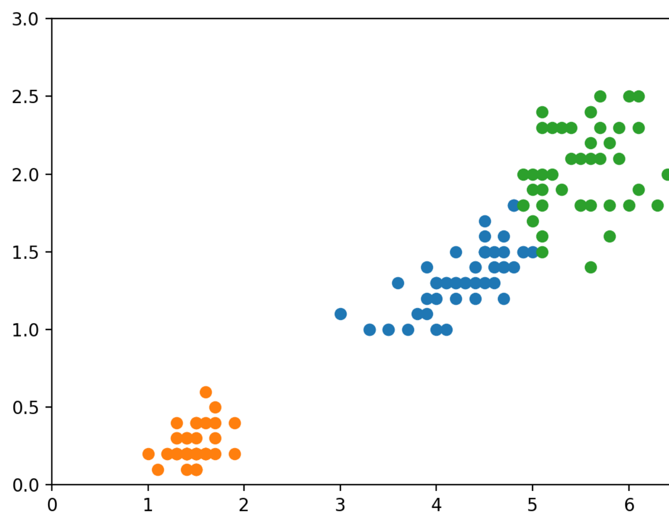


Figure 8.4: Résultat de l'algorithme des k-moyennes, pour $k = 3$, appliqué aux données d'iris.

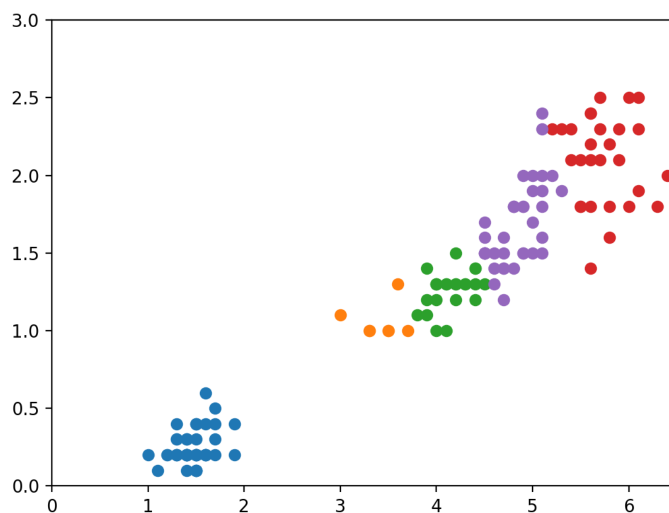


Figure 8.5: Résultat de l'algorithme des k-moyennes, pour $k = 5$, appliqué aux données d'iris.

F. Intérêt et limites de l'algorithme des k-moyennes

La détermination du minimum global de la variance intra-classe $V(S_1, ..., S_k)$ est un problème qui est considéré comme difficile, au sens où l'on conjecture qu'il n'existe pas d'algorithme de complexité polynomiale pour le résoudre. De ce point de vue, l'algorithme des k-moyennes est une heuristique plutôt efficace pour trouver une solution approchée du problème ; c'est un algorithme simple et rapide en général.

Comme nous l'avons vu sur le premier exemple, cet algorithme peut échouer dans certains cas. On peut, pour contourner ce problème, le relancer plusieurs fois et garder une solution telle que $V(S_1, ..., S_k)$ soit minimale ; c'est le même principe que l'on retrouve dans d'autres méthodes heuristiques comme le recuit simulé (qui est cependant une méthode probabiliste).

Enfin, l'algorithme des k-moyennes est également limité et n'est pas forcément adapté à des groupes de forme complexe, ou à des groupes de tailles très différentes, pour lesquels d'autres méthodes existent.

Chapitre 9

Jeux d'accessibilité à deux joueurs sur un graphe

Dans ce chapitre, nous allons nous intéresser à la modélisation de certains jeux à deux joueurs au moyen d'un graphe. Beaucoup de jeux classiques (morpion, échecs, Puissance 4, Othello par exemple) rentrent dans ce cadre, avec des différences notables selon la complexité du jeu :

• si l'espace des configurations est de petite taille, il sera possible de déterminer explicitement le caractère gagnant ou non d'une configuration.

• Sinon (pour le jeu d'échecs par exemple), une analyse exhaustive de toutes les configurations possibles n'est plus envisageable ; dans ce cas, nous verrons comment programmer un ordinateur grâce à l'utilisation conjointe d'une heuristique et d'un algorithme appelé min-max.

I. Jeu d'accessibilité

A. Graphe biparti

Définition I..1

Un graphe (orienté ou non orienté) G est dit biparti si l'on peut partitionner l'ensemble de ses sommets en deux sous-ensembles disjoints S_1 et S_2 tels que toute arête du graphe admet une extrémité dans S_1 et l'autre dans S_2. Dans la pratique, cette définition garantit que les joueurs vont jouer à tour de rôle.

Exemple : dans le graphe ci-dessus, les sommets ronds « appartiennent » au joueur J_1 (cf infra pour la signification de ce terme) et les sommets carrés « appartiennent » au joueur J_2.
Le déroulement du jeu est le suivant : initialement, un jeton est posé sur le sommet A du graphe et à chaque tour le jeton est déplacé vers l'un des successeurs du sommet sur lequel il est posé. Le joueur qui déplace le jeton est celui qui possède le sommet sur lequel il est posé. L'objectif du joueur J_1 est d'amener le jeton sur le sommet C (sommet gagnant pour lui-même) et celui de J_2 est d'amener le jeton sur le sommet H (sommet gagnant pour lui-même).

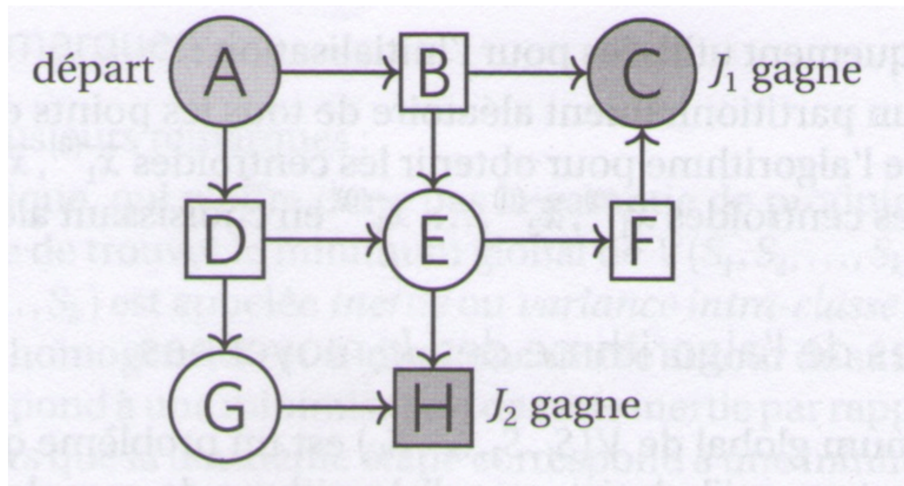


Figure 9.1: Un exemple très simple de graphe biparti.

Le graphe précédent est bien un graphe biparti. Décrire alors, avec un minimum d'information, une méthode qui permettra au joueur J_1 de gagner à coup sûr (c'est ce qu'on appelle une stratégie gagnante).

B. Définition générale

Le jeu précédent est un exemple très simple de jeu d'accessibilité sur un graphe G . Ce type de jeu est en réalité un modèle universel pour les jeux à deux joueurs qui possèdent les propriétés suivantes :

- le jeu est à information complète (il n'y a pas d'information cachée) ;
- le jeu est déterministe (pas d'intervention du hasard) ;
- les deux joueurs jouent à tour de rôle (pas de jeu simultané).

Les exemples les plus connus sont les jeux de morpion, d'échecs, de Puissance 4 et d'Othello.

▷ Chaque sommet du graphe G représente donc une configuration de jeu atteignable par l'un des deux joueurs. À chacune des configurations que le joueur J_1 (resp. J_2) peut rencontrer, on associe un sommet $s \in S_1$ (resp. $s \in S_2$) du graphe G ; ainsi, chaque sommet de G (qu'on appellera également « état ») représente donc une configuration atteignable par l'un des deux joueurs et le graphe G est biparti .

▷ Pour compléter la modélisation du jeu, il convient également de préciser :

- l'état initial s_0 (on convient en général que c'est le joueur J_1 qui débute la partie, donc que $s_0 \in S_1$) ;
- les configurations terminales (états finals) qui peuvent être gagnants pour le joueur J_1 ou gagnants pour le joueur J_2 ou des situations de match nul. On notera, pour un tel état final $s \in \mathcal{G}$: $s \in G_1$ s'il est gagnant pour J_1, $s \in G_2$ s'il est gagnant pour J_2 et $s \in N$ s'il y a match nul. Nous conviendrons qu'une partie infinie correspond à un match nul (noter que si le graphe G est fini et acyclique, comme c'est le cas pour tous les jeux de remplissage (morpion, Puissance 4, Othello), de telles parties infinies sont impossibles).

Définitions I..2

❍ Un jeu d'accessibilité à deux joueurs J_1 et J_2 est défini par :
- un graphe (fini) orienté $\mathcal{G} = (S, A)$ biparti selon la partition $S = S_1 \cup S_2$ (cf supra), appelé arène ;
- un état de départ $s_0 \in S$;
- une partition de l'ensemble F des états finals, sous la forme $F = G_1 \cup G_2 \cup N$ (états gagnants pour J_1, états gagnants pour J_2, états de match nul).

❍ Une partie est un chemin de G qui part de s_0 et qui soit termine en un état de F (état final), soit est infini (donc donne un résultat de match nul).

C. Exemples pratiques

⋆ **Le jeu de morpion**

Dans ce jeu, l'arène est un graphe à $3^9 = 19683$ états possibles (représentant toutes les grilles de $9 = 3 \times 3$ cases, où chaque case est vide ou occupée par une croix ou occupée par un rond), dont beaucoup représentent des positions non atteignables. L'état initial est la grille vide, et les états finals sont :

- les grilles contenant un et un seul alignement de trois symboles identiques (croix ou ronds) : ce sont les états gagnants ;
- les grilles complètement remplies sans alignement de trois symboles identiques (croix ou ronds) : ce sont les parties nulles.

Il ne peut y avoir, comme pour tous les jeux de remplissage, de partie infinie (autrement dit le graphe est acyclique).

⋆ **Un jeu d'empilement**

On considère le jeu suivant : on dispose de briques élémentaires de même forme, de couleurs identiques ou pas, que l'on peut empiler les unes sur les autres. On supposera qu'il y a C couleurs différentes (numérotées $0, 1, 2, ..., C - 1$ et qu'on dispose de N briques élémentaires de chaque couleur (ce qui fait un total de NC briques élémentaires).

Au début du jeu, chaque brique est indépendante et constitue à elle-seule une pile de hauteur 1. À tour de rôle, chaque joueur doit choisir deux piles compatibles et les empiler (en mettant celle de son choix au-dessus de l'autre). Deux piles sont dites compatibles si elles ont la même hauteur (c'est-à-dire qu'elles sont constituées d'un même nombre de briques) ou si leurs briques supérieures ont la même couleur. Le premier joueur qui ne peut plus jouer (parce qu'il n'y a plus 2 piles compatibles) a perdu.

▷ Expliciter l'arène de ce jeu d'empilement pour $C = 2$ et $N = 2$.

II. Stratégies gagnantes, attracteurs et positions gagnantes

A. Stratégies et stratégies gagnantes

Définitions II..1

- Soit $\mathcal{G} = (S_1 \cup S_2, A)$ une arène.
  Une **stratégie** (sans mémoire) pour J_1 est une fonction $f : S_1 \to S_2$ telle que $(s, f(s)) \in A$ pour tout état $s \in S_1$ qui n'est pas un état final de $\mathcal{G}$.
  Une stratégie (sans mémoire) pour J_2 est une fonction $g : S_1 \to S_2$ telle que $(s, g(s)) \in A$ pour tout état $s \in S_2$ qui n'est pas un état final de $\mathcal{G}$.
- Soit $\mathcal{G} = (S, A)$ une arène dont l'ensemble des états finals se décompose en $G_1 \cup G_2 \cup N$ (cf supra).
  Une stratégie f est dite **gagnante** pour J_1 si toute partie $\pi = (s_0, s_1, s_2, ...)$, vérifiant $s_{2n+1} = f(s_{2n})$ pour tout $n \in \mathbb{N}$, est finie et termine en un état de G_1. Une stratégie g est dite gagnante pour J_2 si toute partie $\pi = (s_0, s_1, s_2, ...)$, vérifiant $s_{2n+2} = g(s_{2n+1})$ pour tout $n \in \mathbb{N}$, est finie et termine en un état de G_2.

Remarques :

1. La stratégie f pour J_1 consiste, lorsqu'il se trouve à l'état non final $s \in S_1$, à aller systématiquement à l'état $f(s)$.
   De façon symétrique, la stratégie g pour J_2 consiste, lorsqu'il se trouve à l'état non final $s \in S_2$, à aller systématiquement à l'état $g(s)$.
2. La stratégie f (resp. g) ne nécessite pas d'être définie sur S_1 (resp. J_2) tout entier.
3. La notion de stratégie gagnante telle que définie précédemment se généralise sans difficulté à partir d'une position s donnée, qui n'est plus nécessairement s_0.

▷ Pour le jeu d'empilement vu plus haut (avec $C = 2$ et $N = 2$), décrire une stratégie gagnante pour le joueur J_1 (celui qui commence).

B. Attracteurs

Cette notion est centrale pour déterminer une stratégie gagnante dans le cas général. Pour un sous-ensemble F de S, on notera $\mathrm{Attr}_1^{\mathrm{n}}(\mathrm{F})$ l'ensemble des positions (de $S_1 \cup S_2$) à partir desquelles le joueur 1 possède une stratégie qui lui assure d'arriver à un état de F en au plus n coups. Cet ensemble peut être défini par récurrence de la façon suivante :

• $\mathrm{Attr}_1^0(\mathrm{F}) = \mathrm{F}$;

• un état s de $S_1 \setminus \mathrm{Attr}_1^{\mathrm{n}}(\mathrm{F})$ appartient à $\mathrm{Attr}_1^{\mathrm{n+1}}(\mathrm{F})$ s'il existe **un** successeur de s dans $\mathrm{Attr}_1^{\mathrm{n}}(\mathrm{F})$ (c'est J_1 qui joue et il n'a donc besoin que d'une possibilité de rester dans l'attracteur de F) ;

• un état s de $S_2 \setminus \mathrm{Attr}_1^{\mathrm{n}}(\mathrm{F})$ appartient à $\mathrm{Attr}_1^{\mathrm{n+1}}(\mathrm{F})$ si **tout** successeur de s appartient à $\mathrm{Attr}_1^{\mathrm{n}}(\mathrm{F})$ (cette fois, c'est J_2 qui joue et il doit donc être forcé de rester dans l'attracteur de F).

D'où la définition mathématique suivante :

Définition II..2

Pour chaque joueur J_i ($i \in \{1,2\}$, on définit l'attracteur d'ordre n d'un sous-ensemble $F \subset S$ d'états de l'arène $\mathcal{G} = (S, A)$ par $\mathrm{Attr_i^0(F) = F}$ et la relation de récurrence

$$\begin{aligned}\forall n \geqslant 1, \mathrm{Attr_i^{n+1}(F)} &= \mathrm{Attr_i^n(F)} \\ &\cup \{s \in S_i,\ \exists t \in \mathrm{Attr_i^n(F)}\ /\ (s,t) \in A\} \\ &\cup \{s \in S_{3-i},\ \forall t \in S_i,\ (s,t) \in A \Rightarrow t \in \mathrm{Attr_i^n(F)}\}.\end{aligned}$$

Remarque : par définition de l'attracteur, on a :

$$F = \mathrm{Attr_i^0(F)} \subset \mathrm{Attr_i^1(F)} \subset \mathrm{Attr_i^2(F)} \subset \cdots \subset \mathrm{Attr_i^{|S|}(F)},$$

et la suite est constante à partir du rang $|S|$. On note alors $\mathrm{Attr_i(F)} = \mathrm{Attr_i^{|S|}(F)}$ cette constante (c'est un sous-ensemble de S), et on l'appelle attracteur de F pour le joueur i.

C. Positions gagnantes

Lorsqu'on choisit pour sous-ensemble de S l'ensemble des états finals gagnants, on arrive à la notion de position gagnante.

Définition II..3

Pour $i \in \{1,2\}$, les positions gagnantes pour J_i sont les éléments de $\mathrm{Attr_i(G_i)}$.

On peut alors faire le lien avec la notion de stratégie gagnante :

Propriétés II..4

1. Soit $i \in \{1,2\}$. Le joueur J_i dispose d'une stratégie gagnante si et seulement si l'état initial s_0 est une position gagnante pour J_i.
2. Si le graphe $\mathcal{G}$ est acyclique et tous les états finals gagnants (donc pas de partie nulle), alors il existe une stratégie gagnante pour J_1 ou bien une stratégie gagnante pour J_2.

Exemples :

1. pour le jeu de morpion, il n'existe de stratégie gagnante ni pour J_1, ni pour J_2.
2. pour le Puissance 4, on sait qu'il existe une stratégie gagnante pour J_1, et qu'elle nécessite de jouer le premier coup dans la colonne centrale.
3. pour les échecs, on ignore s'il existe une stratégie gagnante pour l'un des deux joueurs.

D. Le programme en Python

Le graphe (orienté) G est supposé être représenté par le dictionnaire de ses sommets ; à chaque sommet est associé la liste (ou le tuple) de ses successeurs dans le graphe G.
On commence par créer avec la fonction `transpose` le graphe « transposé » de G, dont les sommets sont ceux de G, et dont les arêtes sont celles de G mais de sens contraire (ainsi, le successeur t d'un sommet s de G devient le prédécesseur de s dans le graphe transposé).

On détermine ensuite le dictionnaire des sommets de G, chacun ayant pour valeur son degré sortant dans G.
Enfin, la fonction `calcul_attracteur`, qui prend en entrée le graphe G, le dictionnaire des sommets S_1 contrôlés par J_1 et le dictionnaire des sommets G_1 victorieux pour J_1, renvoie le dictionnaire des sommets gagnants pour J_1 (i.e. l'attracteur de G_1).

```
def transpose(G):
    n=len(G)
    tG={}
    for sommet in G:
        tG[sommet]=[]
    for sommet in G:
        for succ in G[sommet]:
            tG[succ].append(sommet)
    return tG

def degres_sortants(G):
    dict={}
    for sommet in G:
        dict[sommet]=len(G[sommet])
    return dict

def calcul_attracteur(G,S1,G1):
    n=len(G)
    dict_dsG=degres_sortants(G)
    tG=transpose(G)
    Att={}
    def parcours(sommet):
        if sommet not in Att:
            Att[sommet]=True
            for v in tG[sommet]:
                dict_dsG[v]=dict_dsG[v]-1
                if v in S1 or dict_dsG[v]==0:
                    parcours(v)
    for s in G1:
        parcours(s)
    return Att
```

III. Algorithme min-max avec une heuristique

Nous avons vu que pour des jeux simples, il est possible de calculer explicitement une stratégie gagnante lorsque celle-ci existe. Mais pour la plupart des jeux « intéressants », le nombre élevé d'états et de parties possibles rend illusoire un tel calcul (penser au jeu d'échecs). On peut néanmoins trouver des stratégies intéressantes à l'aide d'heuristiques et d'un algorithme général appelé min-max. Notons que c'est grâce à cet algorithme (couplé à une heuristique sophistiquée et à des

raffinements hors programmes) que le programme informatique Deep Blue a battu le champion du monde d'échecs Garry Kasparov en 1997.

A. Fonction d'utilité

Soit $(\mathcal{G}, A)$ une arène, s_0 la position de départ et $G_1 \cup G_2 \cup N$ la décomposition des états finals associés (cf supra). Plaçons-nous du point de vue du joueur J_i ($i \in \{1,2\}$ et supposons que l'on dispose d'une fonction $U_i : S \to \overline{\mathbb{R}}$ (avec $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty; +\infty\}$, appelée fonction d'utilité du joueur J_i, qui pour chaque étt $s \in S$ évalue la position (du point de vue du joueur J_i). Cette fonction est un exemple d'heuristique, construite de façon empirique : aux échecs par exemple, on peut intégrer dans U un terme fondé sur la différence de matériel entre les deux camps, un terme évaluant la différence du nombre de cases contrôlées par chaque camp, un terme évaluant le nombre de déplacements possibles des pièces de chaque camp, etc.
Comme une partie gagnée est d'utilité maximale (et une partie perdue d'utilité minimale), on pose généralement

$$U_i(s) = \begin{cases} +\infty & \text{si } s \in G_i \\ -\infty & \text{si } s \in G_{3-i}. \end{cases}$$

Remarque : en général, les fonctions d'utilité de chaque joueur sont définies symétriquement à partir d'une fonction d'utilité générale U, qui évalue la position s du point de vue du joueur qui possède le sommet. On écrit alors simplement

$$U_1(s) = \begin{cases} U(s) & \text{si } s \in S_1 \\ -U(s) & \text{si } s \in S_2, \end{cases}$$

et symétriquement $U_2(s) = -U_1(s)$.

B. L'algorithme min-max

L'algorithme min-max permet de déterminer une stratégie optimale, en examinant tous les coups possibles sur une profondeur donnée n, et en choisissant une stratégie qui garantit une utilité optimale au bout des n coups.

▷ Un prérequis est d'avoir une fonction d'utilité (donc une heuristique) définie en chaque sommet.

Si l'on se place du point de vue du joueur J_1 qui se trouve en un état $s \in S_1$ non final, il a intérêt à jouer un coup qui maximise l'utilité, c'est-à-dire à choisir un successeur $\tilde{t}$ de s tel que

$$U_1(\tilde{t}) = \max\{U_1(t) \ / \ t \text{ successeur de } s\ \}\ .$$

S'il pousse l'analyse un cran plus loin et considère toutes les réponses possibles de son adversaire (qui lui va choisir un coup qui minimise l'utilité pour J_1), il lui est naturel de choisir un coup qui maximise l'utilité après son coup et la réponse de son adversaire. Cela revient à trouver un état T successeur de s, tel que cette fois (on note $\text{Succ}(s)$ l'ensemble des successeurs de s) :

$$U_1(T) = \max_{t \in \text{Succ}(s)} \begin{cases} \min\limits_{u \in \text{Succ}(t)} U_1(u) & \text{si Succ(t)} \neq \emptyset; \\ U_1(t) & \text{sinon.} \end{cases}$$

Cette construction se généralise à une profondeur quelconque : c'est l'algorithme min-max.

Définition III..1

Soit $i \in \{1, 2\}$. À partir d'une fonction d'utilité $U_i : S \to \overline{\mathbb{R}}$ pour le joueur J_i, on défnit par récurrence pour $n \geqslant 0$ les utilités maximales à n coups $U_i^{(r)}$ par

$$\forall s \in S,\ U_i^{(0)}(s) = U_i(s)$$

et la relation de récurrence

$$\forall n \geqslant 1, \forall s \in S, \quad U_i^{(n)}(s) = \begin{cases} U_i^{(n-1)}(s) & \text{si Succ(s)} = \emptyset, \\ \max\limits_{t \in \text{Succ}(s)} U_i^{(n-1)}(t) & \text{si } s \in S_i; \\ \min\limits_{t \in \text{Succ}(s)} U_i^{(n-1)}(t) & \text{si } s \in S_{3-i}. \end{cases}$$

L'algorithme min-max de profondeur $p \geqslant 1$ définit une stratégie f qui réalise l'utilité maximale à p coups $U_i^{(p)}$. Cette stratégie vérifie, pour tout état s non final :

$$f(s) \in \begin{cases} \text{argmax}_{t \in \text{Succ}(s)} U_i^{(p-1)}(t) & \text{si } s \in S_i; \\ \text{argmin}_{t \in \text{Succ}(s)} U_i^{(p-1)}(t) & \text{si } s \in S_{3-i}. \end{cases}$$

Exemple : on pourra commencer avec les valeurs finales : $3, 2, 6, 4, 7, 4, 5, 2, 3, 4, 2, 6$.

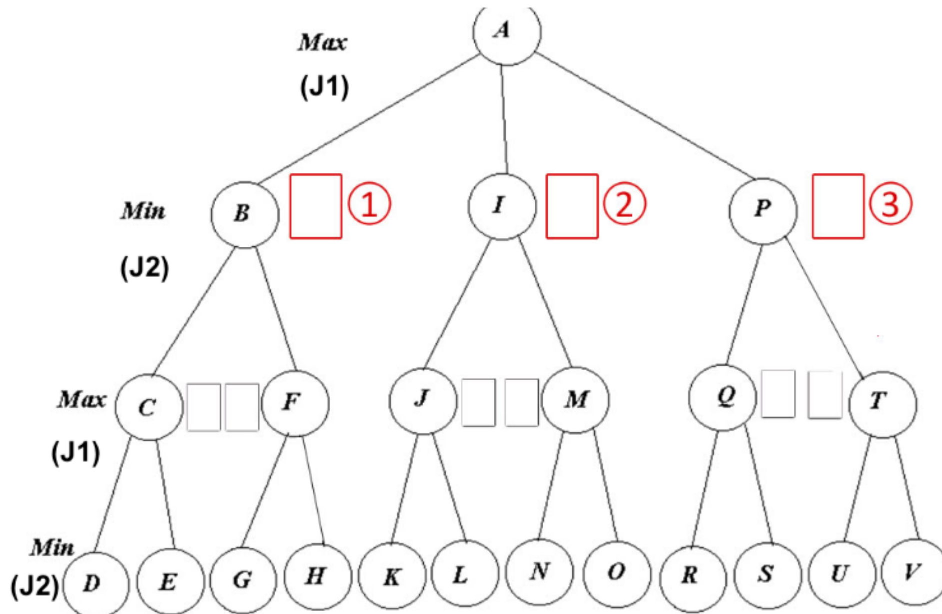


Figure 9.2: Un exemple de graphe binaire avec l'algorithme min-max.

C. Le programme en Python

Le graphe (orienté) G est supposé être représenté par le dictionnaire de ses sommets ; à chaque sommet est associé la liste (ou le tuple) de ses successeurs dans le graphe G.
On peut alors implémenter l'algorithme min-max à l'aide de deux fonctions récursives couplées, prenant en entrée le graphe G, un sommet s, une profondeur p et une heuristique h de la façon suivante :

```
import numpy as np

def minimax(G,s,p,h):
    if len(G[s])==0 or p==0:
        return h(s)
    mini=np.inf
    for succ in G[s]:
        m=maximin(G,succ,p-1,h)
        if m<mini:
            mini=m
    return mini

def maximin(G,s,p,h):
    if len(G[s])==0 or p==0:
        return h(s)
    maxi=-np.inf
    for succ in G[s]:
        m=minimax(G,succ,p-1,h)
        if m>maxi:
            maxi=m
    return maxi
```

Si l'état initial s est contrôlé par le joueur J_1, alors on fera appel à la fonction `maximin` puisque J_1 veut maximiser la valeur du sommet qu'il choisira.
Si l'état initial s est contrôlé par le joueur J_2, alors on fera appel à la fonction `minimax` puisque J_2 veut minimiser la valeur du sommet qu'il choisira.

D. Cas des jeux de petite taille

Dans le cas de jeux de petite taille (c'est-à-dire pour lesquels il est possible d'explorer tous les états à l'aide d'un programme informatique) et associés à des arènes acycliques (donc sans possibilité de partie infinie), l'algorithme min-max permet de retrouver les stratégies gagnantes (ou les stratégies de match nul le cas échéant). En effet, si l'on utilise la fonction d'utilité triviale définie par

$$U_1(s) = \begin{cases} +\infty & \text{si} \quad s \in G_1 \\ -\infty & \text{si} \quad s \in G_2 \\ 0 & \text{sinon,} \end{cases}$$

la fonction d'utilité à $|S|$ coups caractérise les positions gagnantes, perdantes et nulles pour le joueur J_1. L'algorithme min-max produit alors une stratégie gagnante si $U_1^{|S|} = +\infty$ et une stratégie qui garantit le match nul si $U_1^{|S|} = 0$.

Exemples :

1. Pour le jeu d'empilement vu plus haut (avec $C = 2$ et $N = 2$), (re)trouver une stratégie gagnante pour le joueur J_1 (celui qui commence) en utilisant le graphe et l'algorithme min-max.
2. Jeu de Nim simplifié (jeu d'allumettes).
   On considère que l'on commence avec une seule rangée de 6 allumettes. Chaque joueur retire à tour de rôle 1 ou 2 allumettes, et celui qui retire la dernière allumette perd la partie.
   (a) Modéliser ce jeu comme un jeu d'accessibilité sur un graphe (biparti).
   (b) Déterminer une stratégie gagnante pour J_1 en utilisant l'algorithme min-max.
3. Reprendre le jeu précédent avec 7 allumettes.

E. Cas général

Pour le jeu d'empilement vu plus haut avec C et N quelconques cette fois, pour le jeu de Puissance 4, et pour des jeux plus élaborés (échecs, othello), la taille du graphe associé rend illusoire des calculs exhaustifs. Si l'on veut construire une stratégie « intelligente » de jeu pour un ordinateur qui joue contre un être humain, une méthode consiste alors à déterminer une heuristique (la plus efficace possible) et à appliquer l'algorithme min-max précédent.

Ceci constitue un excellent départ pour les TIPE de cette année !