Informatique 2:
Algorithmique sur les tableaux

4TPU148U

Université de Bordeaux

Équipe enseignante informatique 2

https://moodle1.u-bordeaux.fr/course/view.php?id=10750

2022–2023

Table des matières **3**

Chapitre 1. Tableau : structure de données séquentielle

1.1 Tableaux

Définition 1.1.1 Un tableau à une dimension est une structure de données qui permet de stocker un certain nombre d'éléments repérés par un indice encore appelé clé (index). Les tableaux vérifient généralement les propriétés suivantes :

- Tous les éléments ont le même type de base.
- Le nombre maximum d'éléments que l'on peut stocker est fixé.
- La **taille** d'un tableau est le nombre de cases mémoire qui le composent.
- L'accès et la modification d'un l'élément d'index donné est en temps constant (noté $\Theta(1)$) , indépendamment de l'indice et de la taille du tableau.

1.2 Déclaration d'un tableau

La déclaration d'un tableau comprend deux informations :

1. le nombre maximum d'éléments
2. le type de valeurs prises par ses éléments

Tableau de taille fixe

- Java : `int[] tableau` Définit un tableau d'entiers non encore initialisé. Nombre d'éléments pas encore connu.
- C : `int tableau[10]` Définit un tableau prêt à recevoir 10 valeurs entières.
- Python : `tableau = zeros(10)` Définit un tableau contenant 10 zéros avec le package numpy.

Tableau de taille variable : pour simuler un tableau de taille variable, on peut réserver une quantité de mémoire strictement supérieure à celle nécéssaire pour les éléments de départ, et ranger les éléments au début du tableau. Pour les tableaux la bibliothèque principale de Python fournit deux objets natifs de type :

- **list**, équivalent Python d'un tableau, mais cet objet est redimensionnable et peut contenir des éléments de différents types.
- **array** très similaire aux objets de type list mais ne peut contenir des éléments que d'un seul type définit au moment de la déclaration :
  https://docs.python.org/fr/3/library/array.html

1.2.1 Tableau en mémoire

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire. Puisque tous les emplacements sont de même type, ils occupent tous la même taille mémoire d. Connaissant l'adresse a de la première case du tableau, on accède en coût constant à l'adresse de la case d'indice k en calculant $a + kd$.

```
1 from array import *
2 l = array('i', [0]*10)
3 print(f'item\_size = {l.itemsize} octets')
4 print(f'memory\_size = {l.buffer\_info()[1]*l.itemsize} octets')
```

```
item_size = 4 octets
memory_size = 40 octets
```

1.3 Problème de la taille maximum

On essaye d'insérer un élément dans un tableau qui ne contient plus de place disponible. Comportements possibles :

- Erreur (arrêt du programme, exception)
- Allocation d'un nouveau bloc mémoire, puis recopie des éléments de l'ancien tableau dans le nouveau.

Extrait des sources du langage Python, fichier listobject.c, lignes 61-70 :
https://github.com/python/cpython/blob/main/Objects/listobject.c

```
/*This over-allocates proportional to the list size, making room
 * for additional growth.  The over-allocation is mild, but is
 * enough to give linear-time amortized behavior over a long
 * sequence of appends() in the presence of a poorly-performing
 * system realloc().
 * Add padding to make the allocated size multiple of 4.
 * The growth pattern is:  0, 4, 8, 16, 24, 32, 40, 52, 64, 76, ...
 * ...
 */
```

Pour les objets Python de type list l'algorithme permettant d'étendre la taille travaille en temps constant amortis. C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire

1.4 Opérations sur les tableaux

Soient $n \in \mathbb{N}$ le nombre d'éléments dans le tableau et $i \in \mathbb{N}$ l'indice de position d'un élément dans le tableau
Opérations élémentaires

- Initialiser un tableau $\Theta(N)$
- Accéder au premier élément $\Theta(1)$
- Accéder à un élément d'indice i $\Theta(1)$

- Accéder au dernier élément $\Theta(1)$
- Afficher les éléments d'un tableau $\Theta(N)$
- Rechercher si une valeur est dans un tableau ($\Theta(N)$ au pire, $\Theta(1)$ au mieux)
- Échanger le contenu de 2 cases du tableau $\Theta(1)$
- Insérer/supprimer un nouvel élément dans un tableau $\Theta(N-i)$ (insister sur la complexité de l'insertion suivant la place de l'élément à insérer)
- Pour tous les éléments du tableau faire...
- Existe-t-il un élément du tableau qui... ?

1.5 Trier un tableau

Ce sont des algorithmes n'utilisant aucun paradigme ni aucune structure de données élaborée.

- Tri sélection
- Tri insertion
- Tri bulles

1.6 Les tableaux avec Python

Plusieurs possibilités :

- dans les types natifs
  - les objets de type `list`,
  - les objets de type `array`
- package `numpy`
  - les objets de type `ndarray`.

Dans ce cours d'algorithmique nous utiliserons le plus souvent possible les objets de type `array` comme tableau mais nous limiterons les possibilités offertes par Python sur cet objet pour se rapprocher des types tableaux disponibles dans d'autres langages de programmation. L'aspect uniquement déclaratif d'une variable n'existe pas en Python, il faut l'associer à son initialisation.

Remarque : Attention ce cours est un cours d'algorithmique et ne vise pas à être un cours de programmation en Python ; il n'aborde pas les spécificités de ce langage, et les notions présentées s'efforcent de rester valables dans la majorité des autres langages de programmation.

1.7 Initialiser un tableau de type `array`

La doc officielle : https://docs.python.org/fr/3/library/array.html
Ces tableaux ont la particularité d'imposer un type aux éléments du tableau :

```
1 import array as arr
2 tableau = arr.array('l', [1, 2, 3, 4])
3 print(type(tableau))
4 print(tableau.typecode)
5 print(tableau.itemsize)
6 print(tableau.buffer_info())
```

```
<class 'array.array'>
l
8
(140475163192816, 4)
Initialiser un tableau de type array de taille fixe

tab1 = arr.array('l', [0]*10) # avec 10 éléments
print(tableau[0])
0
```

Chapitre 2. Algorithmique sur les tableaux

2.1 Introduction

Pour tous les exercices manipulant des tableaux, on distinguera la taille $nMax$ d'un tableau et le nombre d'éléments n qu'il contient. On entend par taille d'un tableau le nombre de cases de mémoire qui le composent. Un tableau ne peut pas contenir plus d'élements que sa taille. Un tableau pourra par exemple être de taille 10 (c'est à dire qu'il dispose de 10 cases) et contenir moins de 10 éléments. Les éléments seront systématiquement rangés dans les cases d'indices les plus petits possibles, soit de 0 à $n-1$. Par exemple, si un tableau a une taille égale à 10 et contient seulement 4 éléments, ils seront rangés dans les cases d'indice 0 à 3.

Pour connaître la taille $nMax$ d'un tableau on fera appel en Python à la fonction `len` qui prend en paramètre un objet itérable (un tableau est un objet itérable) et renvoie sa taille. Puisque le nombre d'éléments n contenus dans un tableau t est inférieur ou égal à sa taille `len(t)`, il sera nécessaire, chaque fois que l'on passe un tableau en paramètre à une fonction, de passer aussi en paramètre le nombre n de ses éléments.

2.2 Exercices

Exercice 2.2.1 Soit `t` un tableau d'entiers. On appelle monotonie de `t`, une partie de `t` triée dans l'ordre croissant. Si `t` est trié dans l'ordre croissant, sa plus longue monotonie est lui-même. Si T est trié dans l'ordre décroissant, sa plus longue monotonie ne contient qu'un élément.
Écrire une fonction `monotonicity(t, n)` qui renvoie la longueur de la plus longue monotonie d'un tableau d'entiers `t` de `n` éléments ainsi que l'indice auquel commence cette monotonie.

Exercice 2.2.2 Écrire une fonction `search`, prenant en paramètre un tableau non vide `tab` d'entiers, un entier `n` correspondant au nombre d'éléments dans le tableau, un entier `p` et qui renvoie l'indice de la dernière occurrence de `p`. Si l'entier `p` n'est pas présent, la fonction renvoie : `-1`

Exercice 2.2.3 Écrire une fonction `swap(tab, i, j)` qui permute deux éléments d'un tableau non vide `tab`, dont les positions sont `i` et `j`.

Exercice 2.2.4 Soit `t` un tableau contenant `n` nombres rangés dans l'ordre croissant, avec `n < len(t)`. Écrire une fonction `insert_order(t, n, elt)` qui insère un nouvel élément `elt` dans `t` en respectant l'ordre croissant et qui renvoie le nouveau nombre d'éléments dans le tableau.

Exemple : Soit `t =[2, 4, 7, 10, 15, 20, 25, None, None, None]`, l'appel `insert_order(t, 7, 5)` va renvoyer 8 (nombre d'éléments dans t après l'insertion) et va aussi modifier le contenu de `t` en
`[2, 4, 5, 7, 10, 15, 20, 25, None, None]`.

Exercice 2.2.5 Écrire une fonction `separate` qui prend en argument un tableau `t` dont les éléments sont des 0 et des 1 et qui sépare les 0 des 1 en plaçant les 0 au début de tableau et les 1 à la suite.

Exercice 2.2.6 Écrire une fonction `isSection(t, nt, s, ns)` qui prend en paramètre deux tableaux d'entiers `t` et `s` contenant respectivement `nt` et `ns` éléments et qui renvoie True si `s` correspond à une section du tableau `t` et False sinon. Une section est une suite éventuellement vide d'éléments contigus dans un tableau.

Exemple : Soient `t = [5, 1, 2, 3, 1, 2, 1]` et `s = [1, 2]` dans ce cas la fonction renvoie True. Elle renvoie False pour `t = [1, 2, 3, 4, 1]` et `s = [1, 3]`.

Exercice 2.2.7 Soient les variables `student` et `score` qui référencent deux tableaux de même longueur. Ces tableaux contiennent respectivement le nom des élèves de la promo et les notes obtenues au dernier devoir d'informatique. Écrire une fonction `highestScore` qui renvoie les noms des élèves ayant obtenus la note maximale.

Exercice 2.2.8 Soit la fonction `startWithArray` suivante :

```
1 def startWithArray (t, n):
2     m = t[3]
3     for i in range(n-1, 0, -1):
4         if t[i] < m :
5             m = t[i]
6     return m
```

1. En utilisant `t =` | 3 | 5 | 7 | 1 | 10 | 2 | 7 | -3 | comme tableau de valeurs, simulez l'exécution de la fonction `startWithArray` en complétant le tableau ci-dessous qui permet de suivre l'évolution des variables : `m, n, i` et `t[i]`.

	`m`	`n`	`i`	`t[i]`
Avant la boucle				
Dans la boucle				
	...	...	...	...

2. Quel sera en général le résultat de l'exécution de la fonction `startWithArray` pour `n>1` ?

3. Corriger la fonction `startWithArray` pour obtenir le résultat attendu.

4. Quel est le nombre de comparaisons effectuées par la fonction ?

5. Quel est le nombre d'affectations effectuées par la fonction ?

Exercice 2.2.9 Écrire une fonction `insert(t, n, elt, k)` qui, étant donné un tableau `t` contenant `n` éléments (avec `n<len(t)`) insère un élément `elt` à la position `k` avec $k \geqslant 0$. Si $k \geqslant n$, l'élément sera ajouté à l'indice `n`. L'ordre initial des éléments du tableau sera conservé. La fonction doit renvoyer le nouveau nombre d'éléments du tableau.

Exercice 2.2.10 Écrire une fonction `delete(t, n, k)` qui supprime l'élément situé à la position `k` du tableau `t` contenant `n` éléments en supposant $0 \leqslant k < n$. L'ordre initial des éléments du tableau sera conservé. La fonction doit renvoyer le nouveau nombre d'éléments dans le tableau.

Exercice 2.2.11 Écrire une fonction `amplitude(t,n)` qui, en parcourant **une seule fois** le tableau `t`, calcule et renvoie la différence entre le plus grand et le plus petit élément parmi les n premiers éléments du tableau t.

Exemple : Soit `t = [2, 8, 11, 5, 9, 3, 1]` alors `amplitude(t, 7)` renvoie 10 et `amplitude(t, 4)` renvoie 9.

Exercice 2.2.12 On souhaite écrire une fonction `max2(t, n)` qui calcule et renvoie la deuxième plus grande valeur parmi les `n` premiers éléments d'un tableau `t`..

Exemple : Soit `t = [5, 3, 4, 6, 1, 10, 2, 10, 4]`, `max2(t, 5)` doit renvoyer 5, mais `max2(t, 9)` doit renvoyer 10.

1. Proposer un algorithme
2. Quel est le nombre de comparaisons effectuées par votre algorithme ?
3. Implémenter une fonction `max2(t, n)`

Exercice 2.2.13 Écrire une fonction `deleteFirstInstance(t, n, elt)` permettant d'enlever du tableau `t` de taille `n` la première occurrence d'un élément `elt` passé en paramètre (le tableau `t` ne sera pas modifié si `elt` n'appartient pas à `t`). L'ordre initial des éléments du tableau sera conservé. La fonction doit renvoyer le nouveau nombre d'éléments dans le tableau.

1. Quel est le nombre d'éléments du tableau décalés dans le meilleur et dans le pire des cas ?
2. Quel est le nombre de comparaisons concernant des éléments du tableau dans le meilleur et dans le pire des cas ?

Exercice 2.2.14 Écrire une fonction `unduplicated(t, n)` qui retourne True si un tableau d'entiers `t` contenant `n` éléments est sans doublon (c'est à dire sans apparition multiple d'un élément), False sinon.

```
Exemples:
sansDoublon([1, 3, 3, 5, 0], 5) renvoie False
sansDoublon([1, 3, 8, 5, 0], 5) renvoie True
```

Exercice 2.2.15 Soit la fonction Python suivante :

```
1 def myFunction(t, n, x):
2     stop = False
3     while n >0 and not stop :
4         numberOfElement = deleteFirstInstance(t, n, x)
5         if numberOfElement == n :
6             stop = True
7         else :
8             n = numberOfElement
9     return n
```

1. Soit `t = [2, -7, 4, 5, 12, 10, 4, 2, 4, -18]` un tableau contenant 10 éléments. Simulez l'exécution de `myFunction(t, 10, 4)` en complétant le tableau suivant pour montrer l'évolution des variables `n`, `stop`, et `numberOfElement`. Précisez également le contenu de `t` à chaque étape.

`n`		
`stop`		
`numberOfElement`		

2. Quel est le nombre d'éléments du tableau décalés dans le meilleur et dans le pire des cas ?

3. Quel est le nombre de comparaisons concernant des éléments du tableau dans le meilleur et dans le pire des cas ?

Exercice 2.2.16 Écrire une fonction `deleteInstances(t, n, elt)` permettant d'enlever du tableau `t` à `n` éléments toutes les occurrences de l'élément `elt`. La fonction devra renvoyer le nombre d'éléments du tableau à la fin du traitement et ne devra effectuer **qu'un seul parcours** du tableau.
Quel est le nombre de décalages d'éléments du tableau nécessaires dans le meilleur et dans le pire des cas ?

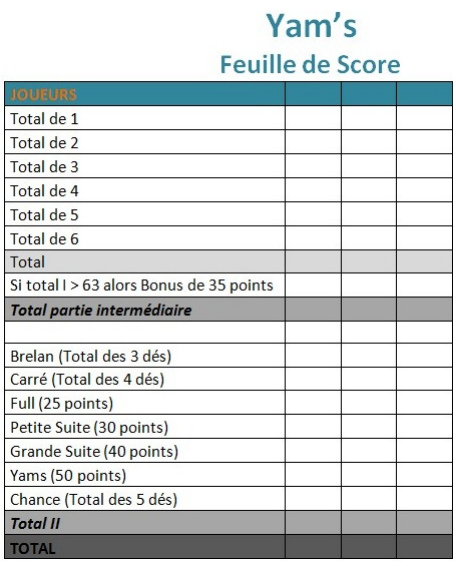
2.3 Exercices complémentaires

En modifiant la fonction `isSection` de l'exercice 2.2.6, écrire une nouvelle fonction `numberSections(t, nt, s, ns)` qui calcule et retourne le nombre de fois où le tableau non vide `s` apparaît comme section du tableau `t`.

Par exemple, le tableau `[1, 2]` correspond à deux sections du tableau `[5, 1, 2, 3, 1, 2, 1]`. Autre exemple, le tableau `[1, 2, 1]` correspond à deux sections du tableau `[5, 1, 2, 1, 2, 1]`. Dans ce dernier cas, les deux sections se chevauchent : `[5, 1, 2, 1, 2, 1]` et `[5, 1, 2, 1, 2, 1]`.

2.4 TP : Le jeu de YAMS

Le Yams se joue avec 5 dés et se finit une fois toutes les cases de la fiche de score remplies. Chaque joueur joue tout à tour et dispose de 3 lancers à chaque coup. L'objectif étant de réaliser des combinaisons qui rapportent des points. Le joueur à le choix de reprendre tout ou une partie des dés à chaque lancer, selon son gré, pour tenter d'obtenir la combinaison voulue. À chaque tour, le joueur doit obligatoirement inscrire son score dans une des cases de la feuille de marque que ce soit par un X ou par les points qu'il a obtenus.
Il peut arriver lors d'un tour que le résultat ne convienne pas au joueur et qu'il se dise qu'il pourrait faire un plus grand score sur un autre tour. Il peut alors choisir de barrer une autre case à la place. Bien entendu, il ne pourra plus faire cette combinaison par la suite.
Lorsque le total intermédiaire est égal ou supérieur à 63 points, un bonus de 37 points supplémentaires est accordé, ce qui peut faire la différence au décompte final. Soyez donc stratégique !


Yam's
Feuille de Score

JOUEURS			
Total de 1			
Total de 2			
Total de 3			
Total de 4			
Total de 5			
Total de 6			
Total			
Si total I > 63 alors Bonus de 35 points			
Total partie intermédiaire			
Brelan (Total des 3 dés)			
Carré (Total des 4 dés)			
Full (25 points)			
Petite Suite (30 points)			
Grande Suite (40 points)			
Yams (50 points)			
Chance (Total des 5 dés)			
Total II			
TOTAL			

Ce TP se décompose en deux parties (A et B), nous utiliserons le type natif `array` proposé par Python et les nombres sont des entiers courts non signés sur 2 octets.

Exemple :

```
from array import *
my_array = array('H', [2, 7, 3])
```

2.4.1 Partie : A

1. Écrire une fonction `rollDice(n)` qui donne le tirage sous la forme d'un tableau contenant des valeurs entières.

   Exemple :

   ```
   dices = rollDice(5)
   print(dices)
   array('H', [3, 6, 4, 3, 2])
   ```

2. Écrire une fonction `sumValues(t, n)` qui renvoie la somme des valeurs affichées par les n dés sans utiliser la fonction Python `sum`.

   Exemple :

   ```
   sumValues(dices, 5)
   18
   ```

3. Écrire une fonction `numberOfDiceWithSameValue(t, n, value)` qui renvoie le nombre de dés affichant la valeur passée en paramètre.

   Exemple :

   ```
   numberOfDiceWithSameValue(dices, 5, 3)
   2
   ```

4. Écrire une fonction `histogram(t, n)` qui renvoie un tableau contenant le nombre de dés identiques pour chaque face. La première case du tableau stocke le nombre de dés.

   Exemple :

   ```
   t = array('H', [3, 6, 4, 3, 2])
   histogram(t, len(t))
   array('H', [5, 0, 1, 2, 1, 0, 1])
   ```

5. Modifier la fonction `histogram` afin d'obtenir l'histogramme pour un nombre de tirage `m`. Que doit-on vérifier si `m` est grand ?

6. Écrire une fonction `maxSubarraySum(t, n)` renvoyant la somme des faces pour le plus grand nombre de dés identiques dans le tirage. S'il y a égalité la fonction renvoie la plus grande somme.

   Exemple :

   ```
   t = array('H', [5, 6, 5, 3, 6])
   maxSubarraySum(t, len(t))
   12
   ```

2.4.2 Partie : B

7. Écire une fonction `moveDice(t, n, value)` qui renvoie un tableau avec les dés correspondants à la valeur passée en paramètre, placés au début du tableau. Que pouvez-vous dire de l'ordre des éléments dans le tableau renvoyé par cette fonction ? **Remarque :** Vous ne devez pas utiliser un tableau auxiliaire.

   Exemple :

   ```
   t = array('H', [..., 6, ..., ..., 6])
   moveDice(t, len(t), 6)
   array('H', [6, 6, ..., ..., ...])
   ```

8. Écrire une fonction `rollDiceAgain(t, nt, s, ns)` permettant de ne relancer qu'une partie des dés. Cette fonction possède 4 paramètres un tableau du tirage, le nombre d'éléments dans tirage, un tableau des indices des dés à relancer, le nombre d'éléments de ce dernier et renvoie le nouveau tirage.

   Exemple :

   ```
   t1 = array('H', [4, 4, 4, 1, 2])
   t2 = array('H', [3, 4])
   rollDiceAgain(t1, len(t1), t2, len(t2))
   # Résultat
   array('H', [4, 4, 4, 5, 3])
   ```

9. Écrire une fonction `searchSequence(t, n)` permettant de détecter la présence d'une suite c'est à dire la combinaison de dés avec des valeurs qui se suivent. La fonction renvoie `0` si la suite est [1, 2, 3, 4, 5], `1` pour [2, 3, 4, 5, 6] et `-1` si aucune des deux suites n'est détectées. Attention les valeurs des dés peuvent être rangées dans n'importe quel ordre au moment du tirage.

   Exemple :

   ```
   t = array('H', [4, 6, 3, 2, 5])
   searchSequence(t, len(t))
   # Résultat :
   1
   ```

10. Écrire une fonction permettant de lancer tout ou une partie des dés trois fois (ou moins si toutes les faces de dés sont identiques) pour obtenir le plus grand nombre de dés identiques possible. En cas d'égalité la fonction choisie les dés avec la valeur de face la plus grande. Cette fonction prendra en paramètre un tableau des cases encore disponibles dans la partie supérieure de la grille. Cette fonction renvoie un tableau du tirage obtenu à chaque lancer.

    Exemple :

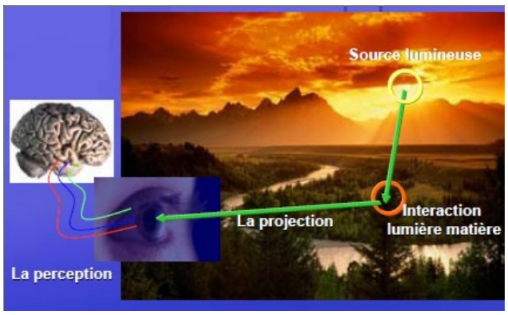
    ```
    #Résultat du script
    jouerTour(array('H', [3, 4, 6]))
    tirage [4 4 5 1 6] tour 1
    tirage [4 4 4 4 1] tour 2
    tirage [4 4 4 4 4] tour 3
    ```

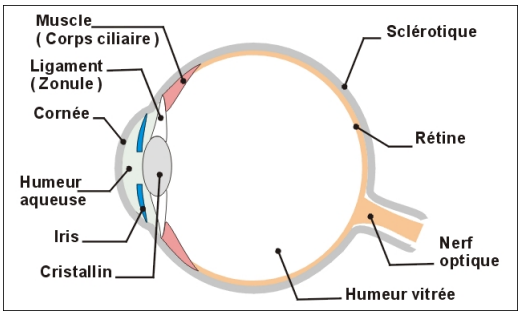
```
jouerTour(array('H', [3, 4, 6]))
tirage [3 3 2 2 1] tour 1
tirage [4 4 3 1 3] tour 2
tirage [4 4 1 2 6] tour 3

jouerTour(array('H', [1, 2, 6]))
tirage [1 1 1 1 4] tour 1
tirage [1 1 1 1 1] tour 2
[1 1 1 1 1]
```

Chapitre 3. Les images

3.1 Formation des images Rappel de physique sur la lumière

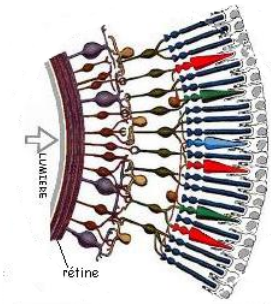
3.1.1 Image et système visuel humain

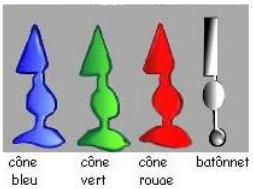




L'œil humain est un globe sphérique d'environ 25 mm. Il comporte de nombreux éléments. L' iris est la membrane colorée qui donne sa couleur à l'œil : en se contractant ou en se dilatant, elle va moduler la quantité de lumière qui traverse le trou percé en son centre appelé pupille. La lumière passe par la cornée traverse l'humeur aqueuse et le cristallin (soutenue par deux muscles) ce qui permet la formation de l'image sur la rétine. Par la suite, la lumière (formée d'ondes électromagnétiques) est convertie, par les photorécepteurs et les neurones (organisés en trois couches) de la rétine, en impulsions électriques. Celles-ci, désormais compréhensibles par le cerveau, lui sont ensuite transmises par l'intermédiaire du nerf optique.

3.1.2 Les cônes et la couleur



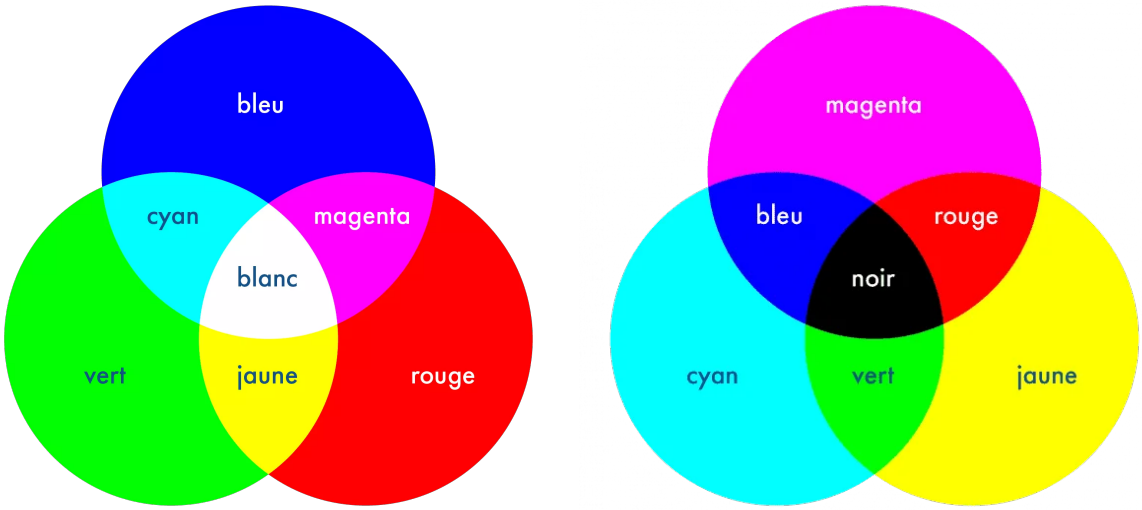


Les cônes sont des photo-récepteurs possédant plusieurs types de pigments (substances chimiques : *l'iodopsine*), caractérisant leur sensibilité à des longueurs d'onde différentes. Les cônes sont au nombre de 6 à 7 millions et leurs trois sortes de pigments permettent la vision de la couleur.

- cyanolabe 420 nm $\longrightarrow$ bleu
- chlorolabe 535 nm $\longrightarrow$ vert
- érythrolabe 570 nm $\longrightarrow$ jaune-rouge

3.1.3 La couleur

La couleur telle que nous la percevons est un phénomène complexe qui lie physique, chimie et système visuel : œil + cerveau.



(a) Synthèse additive : production de lumière (b) Synthèse soustractive : absorption de lumière

FIGURE 3.1 –

3.2 Images numériques

Une image, en informatique, est un simple tableau à deux dimensions de points colorés appelés **pixels**. Les **coordonnées** (x, y) d'un pixel expriment sa position au sein de l'image : x est son abscisse, en partant de la gauche de l'image, et y est son ordonnée, en partant du haut de l'image (attention à l'inverse de l'ordonnée mathématique). Elles partent toutes deux de 0. Le schéma ci-dessous montre un exemple d'image en gris sur fond blanc, de 7 pixels de largeur et 4 pixels de hauteur, les abscisses vont donc de 0 à 6 et les ordonnées vont de 0 à 3.

0,0	1,0	2,0	3,0	4,0	5,0	6,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3

La **couleur RGB** (r, g, b) d'un pixel est la quantité de rouge (r pour *red*), vert (g pour *green*) et de bleu (b pour *blue*) composant la couleur affichée à l'écran. Le mélange se fait comme trois lampes colorées rouge, vert et bleue, et les trois valeurs r, g, b expriment les intensités lumineuses de chacune de ces trois lampes, exprimées entre 0 et 255. Par exemple, $(0, 0, 0)$ correspond à trois lampes éteintes, et produit donc du noir. $(255, 255, 255)$ correspond à trois lampes allumées au maximum, et produit donc du blanc. $(255, 0, 0)$ correspond à seulement une lampe rouge allumée au maximum, et produit donc du rouge. $(255, 255, 0)$ correspond à une lampe rouge et une lampe verte allumées au maximum, et produit donc du jaune. Et ainsi de

suite. Cela correspond très précisément à ce qui se passe sur vos écrans ! Si vous prenez une loupe, vous verrez que l'écran est composé de petits points (appelés *sous-pixels*) verts, rouges et bleus, allumés plus ou moins fortement pour former les couleurs.

3.2.1 Notions de base sur les images Python

Pour manipuler les images avec python nous utiliserons *python Imaging Library* (PIL), qui est préinstallée sur vos machines. Utiliser l'instruction suivante pour importer PIL

```
1 from PIL.Image import *
```

On dispose alors d'un ensemble de fonctions pour manipuler des images. Voici un résumé des fonctions que nous utiliserons dans ce chapitre :

`open(nom)`	Ouvre le fichier *nom* et retourne l'image contenue dedans (par exemple `open("lion.png")`.
`save(nom)`	Sauvegarde l'image *img* dans le fichier *nom*.
`new("RGB",(large,haut))`	Retourne une image de taille $large \times haut$, initialement noire.
`show(img)`	Affiche l'image *img*.
`putpixel((x,y), (r,g,b))`	Peint le pixel (x, y) dans l'image *img* de la couleur (r, g, b)
`getpixel((x,y))`	Retourne la couleur du pixel (x, y) dans l'image *img*
`size`	Retourne un tuple (largeur, hauteur) de l'image

Les bouts de code suivants pourront être testés avec l'image `lion.jpeg` contenue dans le dossier `TD_machine/images` du github.

Exercice 3.2.1 Exécuter les instructions suivantes :

```
1 YELLOW = (255, 255, 0)
2 img1 = Image.new("RGB", (300, 200))
3 j = 100
4 for i in range(img1.size[1]):
5     img1.putpixel((i, j), YELLOW)
6 img1.show()
```

Expliquer ce qu'effectue chaque instruction, et observer l'image produite. Confirmer ainsi le sens dans lequel fonctionnent les coordonnées. Modifiez le code pour que la ligne jaune traverse la fenêtre.

Exercice 3.2.2 Écrire une fonction : `solidRectangle(img, x1, x2, y1, 2, color)` qui prend en paramètres une image, la position du coin supérieur gauche (x1, y1) et du coin inférieur droit (x2, y2) du rectangle, sa couleur et qui dessine un rectangle plein.

Exercice 3.2.3 Écrire une fonction `filtreRouge(img)` qui, pour chaque pixel de l'image, ne conserve que la composante rouge. Testez-la sur la photo `livre.jpeg`. Faites de même pour le vert et le bleu. Que se passe-t-il si on applique successivement un filtre rouge puis un filtre bleu sur la même photo ? Vérifiez votre hypothèse.

3.3 Niveau de gris et modification des couleurs

Il arrive souvent qu'une photo ne corresponde pas à nos souhaits. De nombreux paramètres entrent en jeu lors de la prise d'une photo. Les plus simples à corriger sont les couleurs, la luminosité et le contraste d'une image.

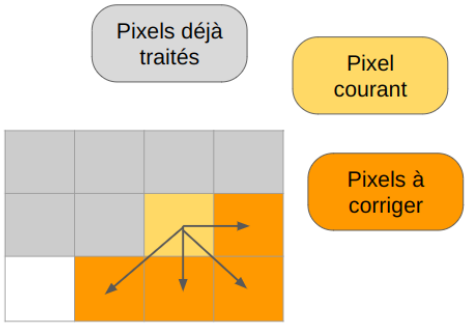
Exercice 3.3.1 Il paraît facile de convertir une photo couleur en photo en noir et blanc. Écrire une fonction `monochrome(img)` qui pour chaque pixel, calcule la moyenne $lum = \frac{r+g+b}{3}$ des composantes r, g, b, et peint le pixel de la couleur (lum, lum, lum). Tester votre fonction sur l'image de la synthèse additive (figure 3.1).

Les éléments verts semblent cependant plus foncés que sur la photo d'origine. L'œil humain est effectivement peu sensible au bleu et beaucoup plus au vert. Une conversion plus ressemblante est d'utiliser plutôt $lum = 0.3 * r + 0.59 * g + 0.11 * b$. Essayez, et constatez que les éléments verts ont une luminosité plus fidèle à la version couleur.

Exercice 3.3.2 Le seuillage est une opération qui permet de transformer une image en niveau de gris en image binaire (noir et blanc), l'image obtenue est alors appelée masque binaire. Il existe 2 méthodes pour seuiller une image, le seuillage manuel et le seuillage automatique. En pratique, le seuil optimal est considéré comme étant celui qui sépare le mieux le fond et les objets présents sur l'image. Écrire une fonction `binarisation(img)` qui calcule la moyenne des intensités des pixels d'une image en niveaux de gris et qui renvoie une image en noir et blanc avec en blanc tous les pixels dont la valeur de l'intensité est en dessous de la moyenne et en noir les pixels dont l'instensite est au dessus de la moyenne.

Exercice 3.3.3 Comme vous l'avez remarqué dans l'exercice précédent, la qualité des images produites par la fonction `binarisation` laisse à désirer. L'algorithme proposé par Floyd et Steinberg permet de limiter la perte d'information. Ecrire une fonction `floydSteinberg(img)` qui convertit une image en niveaux de gris en une image en noir et blanc. L'algorithme est décrit par les transformations d'intensité ci-dessous :

- $I(i+1, j) = I(i+1, j) + \frac{7}{16}e$
- $I(i+1, j+1) = I(i+1, j+1) + \frac{1}{16}e$
- $I(i, j+1) = I(i, j+1) + \frac{5}{16}e$
- $I(i-1, j+1) = I(i-1, j+1) + \frac{3}{16}e$
- $e = I(i, j) - 0$ si $I(i, j) <=$ seuil
- $e = I(i, j) - 1$ si $I(i, j) >$ seuil



Exercice 3.3.4 Le filtre moyenneur est une opération de traitement d'images utilisée pour réduire le bruit dans une image et/ou flouter une image. Le principe est très simple : un pixel est remplacé par la moyenne de lui-même et de ses voisins. C'est dans la définition du voisinage que tout réside. Concrètement, avec un filtre moyenneur de largeur 3, pour calculer la nouvelle valeur du pixel rouge de l'image originale de gauche, on calcule la valeur moyenne (pixel traité compris) des pixels situés dans un carré de dimension 3x3 centré sur ce pixel. Cela donne la nouvelle valeur du pixel sur l'image transformée :

127	127	127	127
127	127	127	127
127	127	127	127

127	183	96	169
162	93	180	85
104	180	93	181

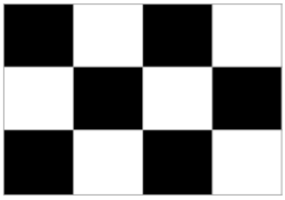


FIGURE 3.2 – transformation de Floyd-Steinberg : [0, 127] —> noir et [128, 255] —> blanc

24	32	234	255	123
44	122	34	200	12
5	167	121	221	202
240	232	128	155	98
124	132	58	167	107

		153		

$$\frac{122 + 34 + 200 + 167 + 121 + 221 + 232 + 128 + 155}{9} = 153$$

Écrire une fonction `blur(img)` qui pour chaque pixel n'étant pas sur les bords, le peint de la couleur moyenne des pixels voisins. Que pensez-vous de votre flou ? Proposer une idée d'amélioration et l'implémenter

3.3.1 Ajustement linéaire de la luminosité et du contraste

On peut ajuster la valeur de la luminosité d'une image en niveau de gris en ajoutant une valeur b à l'intensité du pixel. Si on repère dans le plan la position d'un pixel par ses coordonnées (x, y) on a :

$$J(x, y) = I(x, y) + b$$

- si $b > 0$ la luminosité augmente
- si $b < 0$ la luminosité diminue

(a) Lion trop sombre (b) on ajoute 100 à l'intensité de chaque pixel

FIGURE 3.3 –

La valeur b est ajoutée à chaque pixel de l'image. On augmente dans ce cas l'intensité du niveaux de gris sur tous les points de l'image quelque soit la valeur de départ.

De manière similaire, on peut ajuster le contraste d'une image à l'aide d'un coefficient a sur l'intensité des pixels.

$$g(x, y) = a \times f(x, y)$$

Contrairement à la luminosité tous les pixels ne seront pas affectés de la même façon. Un pixel dont la valeur initiale en intensité de gris est de 0 aura une valeur finale de 0. En fait plus la valeur d'intensité de départ est grande et plus l'écart d'intensité sera grand. En combinant contraste et luminosité on obtient :

$$J(x, y) = a \times I(x, y) + b$$

(a) Image de départ (b) On multiplie par $1,6$ l'intensité de chaque pixel

FIGURE 3.4 –

Exercice 3.3.5 Écrire une fonction `ajustementLinéaire(img, a, b)` qui prend en paramètres une image un coefficient b de luminosité, un coefficient a de contraste et qui renvoie une image ayant subi les modifications de luminosité et/ou de contraste.

Histogramme d'une image

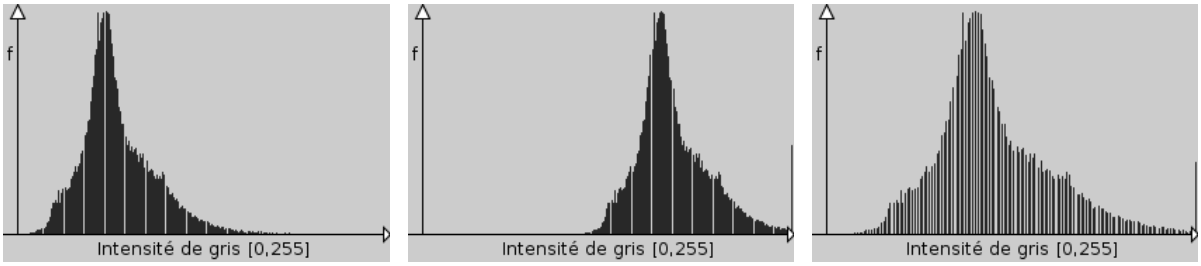
Dans une image en niveaux de gris l'histogramme comptabilise le nombre d'occurence (en ordonnées) de chacune des valeurs d'intensité de 0 à 255 (en abscisses) quand l'image est codée sur 8 bits. Un histogramme permet d'obtenir des informations sur la répartition des intensités comme par exemple la valeur moyenne. Par contre un histogramme ne fournit aucune information de répartion spatiale. Deux images peuvent posséder le même histogramme et être totalement différentes. Certaines modifications d'histogrammes permettent d'améliorer le contraste global d'une image ou encore d'améliorer le contraste pour certaines plages de niveaux de gris. On part d'une image $I(x, y)$ codée sur 8 bits dont on peut calculer un histogramme $h(n)$ où n correspond au niveau de gris allant de 0 à 255 (figure 3.6). On peut également calculer l'histogramme cumulé :

$$h_c(n) = \sum_{i=0}^{n} h(n) \quad \forall \in [0, 255]$$

Exercice 3.3.6 Écrire une fonction `histogramme(img)` qui prend en paramètre une image et qui renvoie l'histogramme de cette image sous la forme d'un tableau.

(a) Image initiale (b) Luminosité modifiée (c) Contraste modifié

FIGURE 3.5 –



(a) Image initiale (b) Luminosité modifiée (c) Contraste modifié

FIGURE 3.6 –

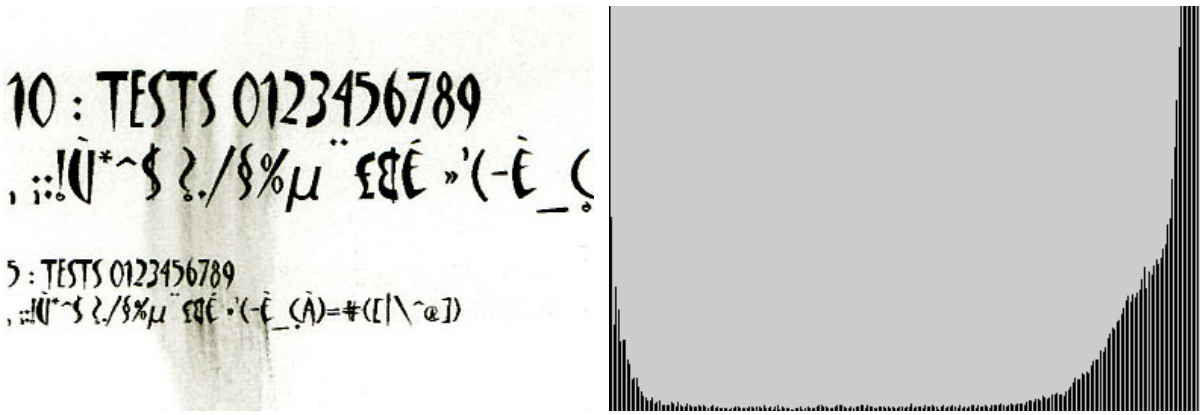
Exercice 3.3.7 La normalisation ou étirement d'histogramme consiste à exploiter les 256 valeurs de niveau d'intensité des canaux (r,g,b). Cela revient à une simple normalisation qui consiste à ramener la valeur max à 255 et la valeur min à 0. Écrire une fonction `normalisationSimple(img)` qui prend en paramètre une image et renvoie une image normalisé.

Exercice 3.3.8 La normalisation de l'exercice précédent est très sensible au bruit. On lui préfère une normalisation qui exploite les statistiques globales de l'image tel que la moyenne d'intensité μ et l'écart-type σ qui sera ensuite suivie d'un nouvel étirement pour ramener les valeurs entre 0 et 255. Écrire une fonction `normalisationGlobale(img)` qui prend en paramètre une image et renvoie une image normalisé.

$$J(x,y) = \frac{I(x,y) - \mu}{\sigma}$$

Exercice 3.3.9 Un exemple d'application est la suppression d'un fond indésirable. Sur l'image de la figure 3.7, l'histogramme permet de mettre en évidence deux zones de niveaux de gris bien distinctes, le texte (niveau proche de 0) et la bavure laissée à la photocopieuse (niveau proche de 255). Écrire une fonction `nettoyerFondIndesirable(img)` L'histogramme permet d'en déduire une valeur seuil de niveau de gris pour supprimer la bavure.

Exercice 3.3.10 La soustraction d'images est utilisée pour mettre en évidence certains détails caractéristiques qui peuvent évoluer au cours du temps (figure 3.8). On effectue alors la différence $g(x,y) = f_2(x,y) - f_1(x,y)$ qui permet d'extraire les détails intéressants. Écrire une fonction `soustraire(img1, img2)` qui met en évidence les différences entre deux images.



(a) Bavure de photocopieuse (b) Histogramme

FIGURE 3.7 –

(a) Image initiale (b) Image initiale avec deux voitures (c) Détection des deux voitures

FIGURE 3.8 –

3.3.2 Exercices complémentaires

Exercice 3.3.11 Une définition de la **droite discrète arithmétique** a été donnée en 1991 par J-P REVEILLES.

Un ensemble de pixels $\mathcal{E}$ appartient à la droite discrète arithmétique(figure 3.9) de pente $\frac{a}{b}$, de borne inférieure μ et d'épaisseur ω (avec a, b, μ et ω dans $\mathbb{Z}, b \neq 0$ et $pgcd(a, b) = 1$) si et seulement si tous les pixels (x, y) de $\mathcal{E}$ vérifient :

$$\mu \leq ax - by < \mu + \omega$$

Cette droite se note $\mathcal{D}(a, b, \mu, \omega)$

3.4 Transformations géométriques 2D

3.4.1 Transformations euclidiennes

Une transformation géométrique permet de déplacer un pixel d'une position (x, y) vers une nouvelle position (x', y') à l'aide d'équations d'écrivant la transormation.

$$\begin{cases} x' & = T_x(x, y) \\ y' & = T_y(x, y) \end{cases}$$

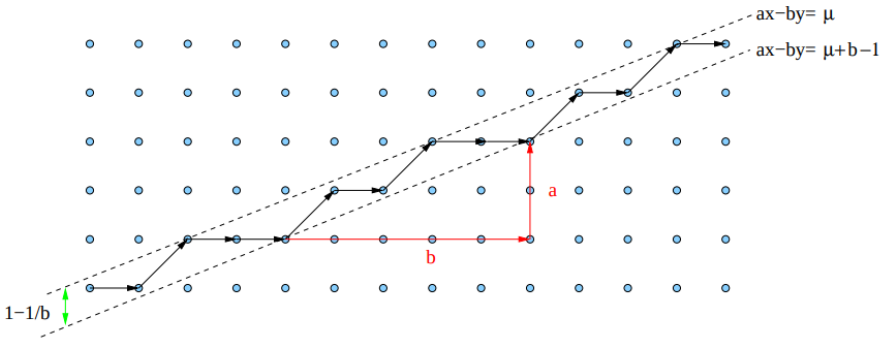


FIGURE 3.9 – Droite de Reveilles de paramètres $\mathcal{D}(2, 5, -1, 5)$

Parmi les transformations géométriques, on peut citer la translation et la rotation. Elles permettent par exemple de stabiliser les images d'une séquences vidéo. Lorsque vous tenez votre caméra à la main de faibles translations et rotations peut donner le mal de mer à quiconque regarde le film non corrigé. Pour corriger ces mouvements parasites, on effectue un ensemble de transformations permettant d'éliminer les mouvements parasites.

Exercice 3.4.1 On souhaite ouvrir la porte gauche d'un placard. Écrire une fonction
`def translatePixelToLeft(img, xi, yi, xf, yf, value)`
qui prend en paramètre une image, le cadre à décaler avec le pixel en haut à gauche (xi, yi) et le pixel en bas à droite (xf, yf) du cadre ainsi que la valeur de décalage. Pour la porte de gauche les coordonnées de ces deux pixels sont (71, 4) à (131, 186)) et le décalage est de 64 pixels. Vous pourrez remplir l'espace libéré avec de la couleur, ou avec l'image `fantome.png`.

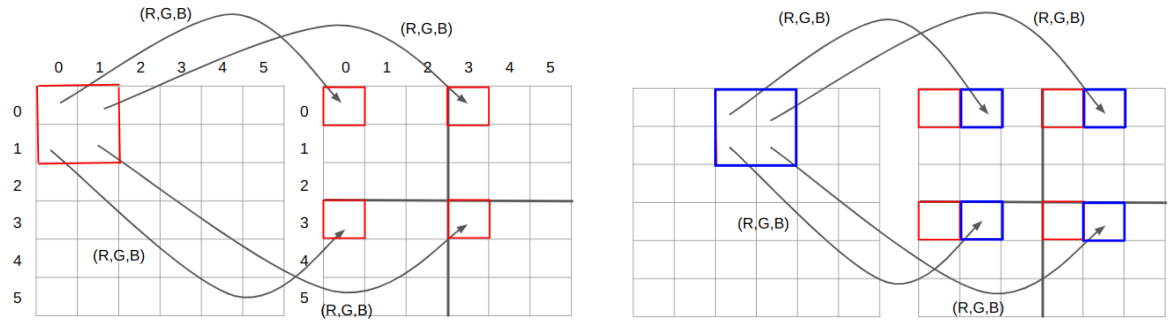
Exercice 3.4.2 Même question en entrouvrant la porte de droite, des pixels $(135, 4)$ à $(196, 186)$, en la décalant de seulement 32 pixels vers la droite.

Exercice 3.4.3 L'objectif de cet exercice est d'écrire le code de deux fonctions Python permettant d'effectuer la transformation miroir (symétrie axiale) d'une image.

1. Écrire la fonction `miroirVertical(img1, img2)` qui place dans `img2` l'image correspondant au miroir vertical de `img1` (`img1` et `img2` sont supposées de même taille).

2. Écrire la fonction `miroirHorizontal(img1, img2)` qui crée et retourne une nouvelle image correspondant au miroir horizontal de `img`.

Exercice 3.4.4 On appelle transformation bijective d'image la transformation d'une image finie de $n \times m$ pixels sur elle-même : chaque pixel est donc déplacé et aucun pixel n'est perdu, ce qu'on appelle en mathématiques une permutation de l'ensemble des pixels. Par exemple, la transformation de l'image $(n \times m)$ qui déplace le pixel (i, j) en $((i + 1)\%n, j)$, correspond à un décalage d'un pixel vers la droite de l'image.
Une transformation connue est celle du **photomaton**. Le principe est le suivant : l'image de départ est découpée en blocs de 4 pixels. Le premier bloc en haut à gauche comporte les pixels encadrés en rouge (figure 3.10). Le pixel en haut à gauche $(0, 0)$ est recopié dans la même position $(0, 0)$, le pixel $(1, 0) \longrightarrow (3, 0)$, le pixel $(0, 1) \longrightarrow (0, 3)$ et le pixel $(1, 1) \longrightarrow (3, 3)$. Puis on recommence avec le bloc de pixels suivants (en bleu sur la figure 3.10), et ainsi de suite jusqu'au dernier bloc en bas à droite de l'image de départ. Écrire une fonction `photomaton(img)` qui prend en paramètre une image et qui renvoie une image *photomaton*



(a) Transformation bijective du bloc rouge

(b) Transformation bijective du bloc bleu

FIGURE 3.10 – Transformation bijective photomaton, l'image doit comporter un nombre pair de lignes et de colonnes

Exercice 3.4.5 Écrire une fonction `forwardRotate(img, angle)` qui prend chaque pixel de l'image de départ et qui calcule par rapport à l'origine la position du pixel sur l'image d'arrivée. Cette fonction prend en argument l'image de départ, l'angle de rotation et renvoie l'image après transformation

$$\begin{cases} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{cases}$$

Exercice 3.4.6 Écrire une fonction `backwardRotate(img, angle)` qui prend chaque pixel de l'image d'arrivée et qui calcule le pixel correspondant sur l'image de départ. Cette fonction prend en argument l'image de départ, l'angle de rotation et renvoie l'image après transformation.

$$\begin{cases} x &= x'\cos\theta + y'\sin\theta \\ y &= -x'\sin\theta + y'\cos\theta \end{cases}$$

Exercice 3.4.7 De manière générale une rotation est déterminée par un centre $C(x_c, y_c)$ et un angle θ. Si un point (x, y) subit une rotation d'angle θ et de centre C. En déduire la vesion `backwardRotateCenter(img, angle, xc, yc)` ou les paramètres `xc` et yc représentent les coordonnées du centre de rotation.

3.5 La segmentation couleur

La segmentation va consister à regrouper les pixels de l'image en régions (composantes connexes) qui vérifient un critère d'homogénéité (par exemple sur la couleur) afin d'obtenir une description de l'image en régions homogènes. Une application simple à réaliser est l'incrustation d'image.

(a) Poisson sur fond vert

(b) Incrustation du poisson

FIGURE 3.11 –

Chapitre 4. Les graphes comme outil de modélisation

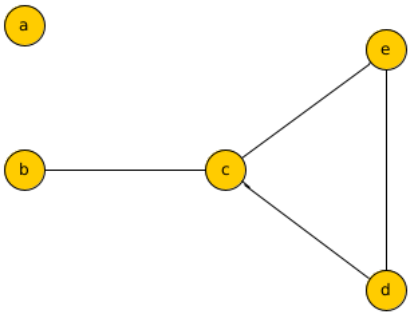
4.1 Ce qu'il faut savoir

Définition 4.1.1 Un graphe G est un couple (X, E) ou X est un ensemble fini et non vide d'éléments appelés sommets et E est un ensemble de paires d'éléments de X appelés arêtes.

- Une arête reliant un sommet à lui-même est une boucle.
- Deux arêtes, ou plus, reliant la même paire de sommets sont des arêtes parallèles (des arêtes multiples).
- Un graphe est dit simple s'il n'a ni boucles ni arêtes multiples.
- Un graphe est dit orienté si ses arêtes sont orientées, c'est à dire qu'il est possible de distinguer son extrémité initiale de son extrémité finale. Dans ce cas on parlera plutôt d'arc que d'arête.

$G = (X = \{a, b, c, d, e\}),\ E = \{bc, cd, de, ec\})$
On peut représenter un tel graphe avec un schéma ou une matrice d'adjacence.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



Aucune des deux représentations n'est unique.

Définition 4.1.2 Deux graphes $G = (X, E)$ et $G'(X', E')$ sont égaux si et seulement si $X = X'$ et $E = E'$

4.1.1 Vocabulaire et notations

Ordre d'un graphe correspond au nombre n de sommets. On le note parfois $|G|$. C'est donc le cardinal de l'ensemble X.

Taille d'un graphe correspond au nombre d'arêtes. On le note parfois $||G||$. C'est aussi le nombre de 1 de la matrice d'adjacence du graphe divisé par 2.

Sommets voisins correspond aux sommets adjacents. Si $e = xy$ est une arête du graphe alors x et y sont des sommets voisins.

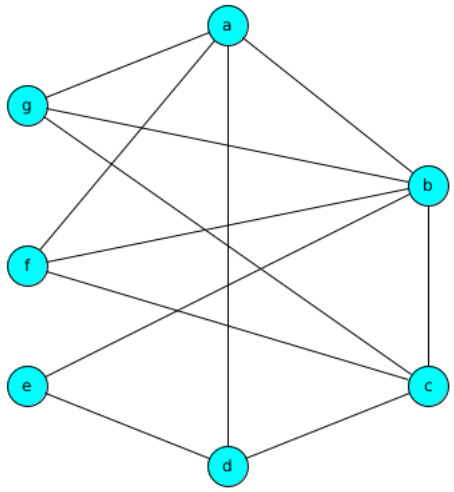
Voisinage d'un sommet x d'un graphe G, noté $V_G(x)$ est l'ensemble de ses voisins.

Degré d'un sommet x d'un graphe G, noté $deg_G(x)$ correspond au nombre de fois où ce sommet apparaît comme extrémité d'une arête. Dans la matrice d'adjacence du graphe G, le degré d'un sommet est égal au nombre de 1 sur la ligne correspondante au sommet.

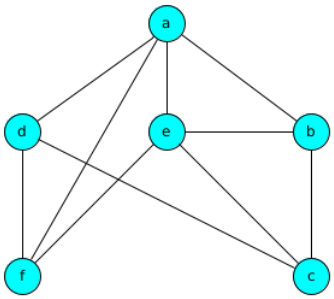
4.1.2 Une autre façon de représenter un graphe

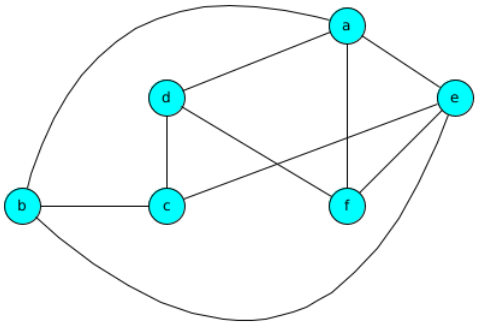
Lorsque l'ordre et la taille d'un graphe devient trop importants, la représentation du graphe par un schéma ou une matrice d'adjacence peut manquer de clarté, il peut alors être pratique de représenter le graphe en donnant le voisinage de chacun des sommets.

Sommet	Voisinage
a	$\{b, d, f, g\}$
b	$\{a, c, e, f, g\}$
c	$\{b, d, f, g\}$
d	$\{a, c, e\}$
e	$\{b, d\}$
f	$\{a, b, c\}$
g	$\{a, b, c\}$



Exercice 4.1.1 Les graphes suivants sont-ils égaux ?





Exercice 4.1.2 Dans une assemblée de dix personnes, chaque membres serre la main de tous les autres. Combien y aura-t-il de poignées de mains ?

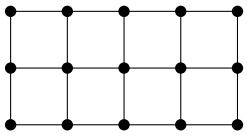
Exercice 4.1.3

1. Dessiner un graphe à 6 sommets tel que la liste des degrés des sommets soit :
   `[2, 1, 4, 0, 2, 1]`.
2. Même question avec `[2, 1, 3, 0, 2, 1]`.

Exercice 4.1.4 Un graphe est dit cubique si tous ses sommets sont de degré 3.

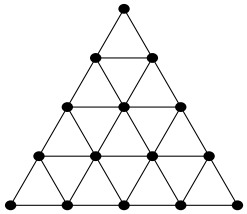
1. Dessiner un graphe cubique ayant 4 sommets ; même question avec 3 sommets, 5 sommets. Que constate-t-on ?
2. Quel est le nombre d'arêtes d'un graphe cubique de n sommets ?
3. En déduire qu'un graphe cubique a un nombre pair de sommets.

Exercice 4.1.5 Une grille (rectangulaire) $m \times n$ est constituée de m lignes horizontales et n lignes verticales, dont les croisements forment les sommets du graphe ; une grille 3×5 est représentée ci-contre.



1. Compter les sommets de degré 2 (respectivement 3 et 4) et en déduire la somme des degrés des sommets en fonction de m et n.
2. Compter les arêtes et comparer avec le résultat de la question précédente.

Exercice 4.1.6 La grille triangulaire T_5 est représentée ci-contre. Adapter les calculs de l'exercice précédent au cas de T_n.



Exercice 4.1.7

1. Essayer de construire un graphe simple ayant 4 sommets, et tel que les degrés des sommets soient tous distincts.
2. On se propose de démontrer par l'absurde qu'il n'existe pas de graphe simple de n sommets ($n \geq 2$) tel que tous ses sommets soient de degrés distincts. Supposons qu'un tel graphe G existe.
   a) Montrer que G ne peut comporter de sommet de degré supérieur ou égal à n.
   b) En déduire les degrés possibles pour les n sommets.
   c) Montrer que ceci entraîne une absurdité.
3. Application : en déduire que dans un groupe de n personnes, il y en a toujours deux qui ont le même nombre d'amis présents.

Exercice 4.1.8 Un crime a eu lieu dans une demeure isolée. En plus de la victime, il y avait 7 autres personnes. Chacune affirme à l'inspecteur, en guise d'alibi, avoir passé une partie de la nuit avec 3 des autres suspects successivement. Après un moment de réflexion, l'inspecteur est convaincu qu'on lui a menti.

1. Pourquoi ?
2. Combien de personnes ont menti sachant que les menteurs n'ont rencontrés personne ce soir là. On sait de plus que parmi ceux qui n'ont pas menti, il y en a au moins deux qui ne se sont pas rencontrés cette nuit là.

4.2 Modélisation de problèmes à l'aide de graphes

4.2.1 Graphes non orientés

Exercice 4.2.1 Le schéma de la figure 4.1 représente le plan d'un musée. Chaque sommet représente une salle et chaque arête un couloir entre deux salles. Un garde placé dans une salle peut surveiller la salle et les couloirs adjacents.

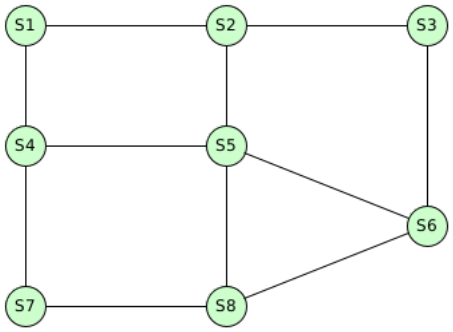


FIGURE 4.1 – Tours de garde

1. Combien faut-il de gardes au minimum pour surveiller tous les couloirs ?

2. Un garde doit effectuer toutes les heures une ronde passant par tous les couloirs au moins une fois et le ramenant à son point de départ, c'est à dire une des salles du musée. Quelle est la longueur mininale, en nombre d'arêtes d'une telle ronde ?

3. Un garde doit effectuer toutes les heures une ronde passant par toutes les salles au moins une fois et le ramenant à son point de départ. Quelle est la longueur mininale, en nombre d'arêtes d'une telle ronde ?

4. Lorsque 2 gardes sont dans des salles voisines, ils peuvent communiquer. On place des gardes en $S1, S2, S3, S5, S7, S8$. Sachant que chaque garde ne peut donner l'information qu'une seule fois, un garde peut-il transmettre une information à tous les autres ?

Exercice 4.2.2 Un centre de vacances cherche à faire un emploi du temps pour 3 groupes de vacanciers A, B, C. Le tableau suivant représente les activités que doivent suivre quotidiennement chaque groupe de vacanciers. La durée d'un cours est d'une heure et chaque cours est donné par un animateur différent.

	Pétanque	Tennis	Yoga	Natation
A	X		X	X
B	X	X	X	
C		X	X	X

1. Donner une représentation et une matrice d'adjacence d'un graphe G permettant de retrouver toutes les informations du tableau.

2. Si une information importante est donnée au hasard à une personne de l'ensemble {vacanciers + animateurs} et sachant que tout vacancier peut transmettre l'information à ses animateurs uniquement et tout animateur peut transmettre l'information à ses vacanciers uniquement, peut-on être sûr que l'information sera transmise à tout le monde ?

3. Sachant qu'il n'y a qu'un animateur par activité, que les animateurs ne sont présents que le matin, qu'ils veulent enchainés tous leurs cours et commencer le plus tard possible, modéliser la situation à l'aide d'un graphe et proposer un emploi du temps pour le directeur du centre.

Exercice 4.2.3 La figure 4.2 représente le plan d'un quartier. Chaque arête est une rue et chaque sommet est un carrefour. Le seul accès au quartier est est le sommet a. Un enfant dont la maison se trouve au sommet b a perdu son chat. Dans chacune des rues du quartier, un témoin a vu avec certitude passer le chat une unique fois à l'exception de la rue qui mène au sommet a ou le témoin est sûr de ne pas avoir vu le chat. Mais où se trouve le chat ?

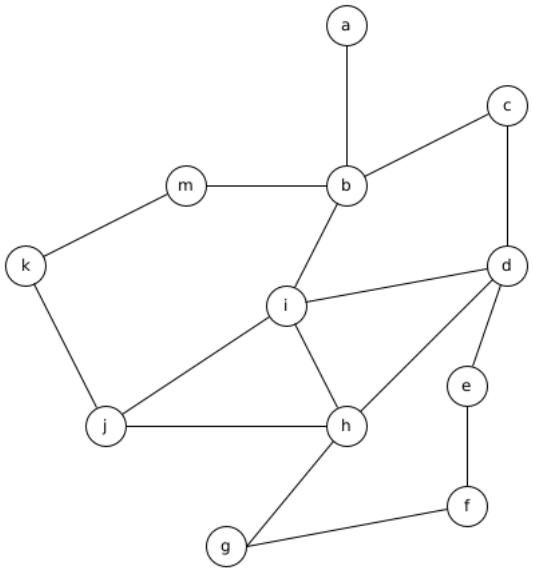


FIGURE 4.2 – Plan du quartier

Exercice 4.2.4 La figure 4.3 représente une carte de pays avec leurs frontières.

1. Représenter cette situation par un graphe dont vous choisirez les informations représentées par les sommets et les arêtes.

2. Peut-on en partant d'un pays franchir toutes les frontières une seule fois et revenir dans le pays d'origine ?

3. Peut-on en partant d'un pays franchir toutes les frontières une seule fois en terminant dans un autre pays ?

4. Peut-on visiter chaque pays une seule fois et revenir dans le pays de départ ?

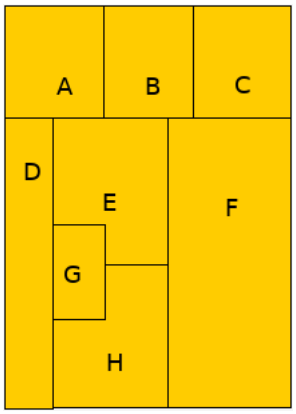


FIGURE 4.3 – Pays et frontières

4.2.2 Graphes orientés

Exercice 4.2.5 On souhaite prélever 4 litres de liquide dans un tonneau. Pour cela on dispose de deux récipients non gradués, l'un de 5 litres, l'autre de 3 litres, et d'une bassine vide de contenance au plus 6 litres. Comment doit-on procéder pour obtenir 4 litres de liquide dans la bassine ? On pourra modéliser le problème à l'aide d'un graphe orienté.

Exercice 4.2.6 (Jeu de Chomp) Chomp se joue avec une tablette de chocolat rectangulaire dont le coin supérieur gauche est empoisonné. Chaque joueur choisit à tour de rôle un carré et le croque ainsi que tous les morceaux situés à sa droite et en dessous. Le joueur qui mange le dernier carré, en haut à gauche, a perdu.

On représente les configurations possibles d'un tel jeu à partir d'une configuration initiale par un graphe. Par exemple, partant d'une tablette de taille 2×2, le graphe est le suivant.

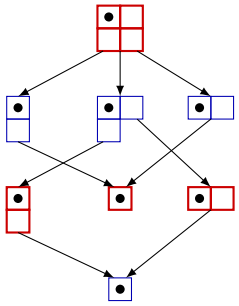


FIGURE 4.4 – Jeu de Chomp

Chaque sommet représente une position du jeu (le carré empoisonné est marqué par un $\bullet$), ainsi que le numéro du joueur qui doit jouer sur cette position (en rouge et gras pour le joueur 0, en bleu pour le joueur 1). Le graphe est orienté, et un arc (l'équivalent d'une arête) entre deux sommets représente la possibilité de jouer un coup. On peut donc voir ce jeu comme joué sur un graphe, avec un jeton se trouvant initialement sur la position 2×2 du jeu. Les joueurs jouent à tour de rôle. Un coup consiste à déplacer le jeton en suivant un arc. Ici, l'objectif du joueur 0 (rouge) est d'atteindre le carré bleu, et l'objectif du joueur 1 (bleu) est d'atteindre le carré rouge.

Le joueur 0 a ici une stratégie pour gagner : il lui suffit de choisir l'arête centrale à partir de la position initiale.

1. Dessiner le graphe correspondant à une tablette 4×2.
2. Montrer qu'à partir d'une tablette carrée, le joueur qui joue en premier a une stratégie pour gagner.

Exercice 4.2.7 (Jeu de Fan Tan) Deux joueurs disposent de deux ou plusieurs tas d'allumettes. À tour de rôle chaque joueur peut enlever un certain nombre d'allumettes de l'un des tas (selon la règle choisie). Le joueur qui retire la dernière allumette a perdu.

1. Modéliser ce jeu à l'aide d'un graphe dans le cas où l'on dispose au départ de deux tas d'allumettes contenant chacun trois allumettes, et où un joueur peut enlever une ou deux allumette à chaque fois de l'un des tas.
2. Que doit jouer le premier joueur pour gagner à coup sûr ?

Exercice 4.2.8 Même questions que dans l'exercice 4.2.7, si l'on dispose initialement de 3 tas, contenant 1, 2, et 4 allumettes, et si chaque joueur peut, à son tour, prendre autant d'allumettes qu'il le veut dans chaque tas (au moins une). On pourra représenter une configuration du jeu par un triplet (x, y, z), où les entiers x, y, z désignent le nombre d'allumettes restant dans chacun des tas.

Chapitre 5. Problèmes et algorithmes de coloration

5.1 Coloration d'un graphe. Nombre chromatique

Un graphe est dit bien colorié si deux sommets voisins ont toujours des couleurs différentes. Un graphe ne peut pas être bien colorié s'il possède une boucle (il existerait alors un sommet voisin de lui-même), et le nombre d'arêtes qui relient deux sommets voisins est sans importance pour la coloration : on ne s'intéressera donc qu'aux graphes *simples*.

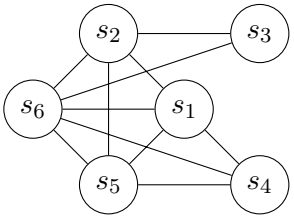
Le nombre minimal de couleurs nécessaires pour bien colorier un graphe G est appelé nombre chromatique (noté χ) de G.

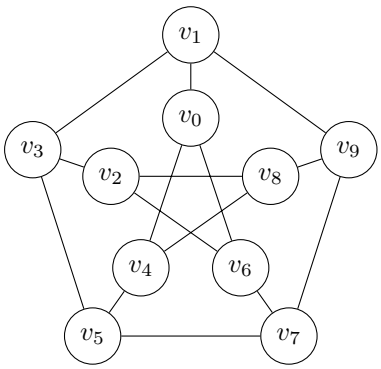
De manière générale, il existe des algorithmes efficaces pour tester si un graphe est connexe et pour construire des chemins dans des graphes. Pour les problèmes de coloration la situation est différente : en général *on ne connaît pas* d'algorithme *efficace*, sauf pour le problème de coloration avec deux couleurs. En particulier on ne connaît pas d'algorithme qui, appliqué à un graphe G quelconque, calculerait rapidement son nombre chromatique ; on ne connait même pas de test efficace qui, appliqué à un graphe G quelconque, déterminerait si trois couleurs suffisent pour colorier G.

On conjecture depuis plusieurs dizaines d'années, sans savoir le démontrer, qu'en fait *il n'existe pas* d'algorithme efficace pour des problèmes tels que la coloration avec trois couleurs.

Exercice 5.1.1 Un graphe complet d'ordre n est un graphe de n sommets tel que chaque sommet est relié aux autres sommets par une arête ; on appelle K_n un tel graphe. Dessiner K_2, K_3, K_4 et K_5. Quel est le nombre chromatique de K_n ?

Exercice 5.1.2 En remarquant que le graphe ci-contre contient des sous-graphes complets (lesquels ?) montrer que son nombre chromatique est supérieur ou égal à 4. Vérifier que l'on peut effectivement le colorier avec 4 couleurs.





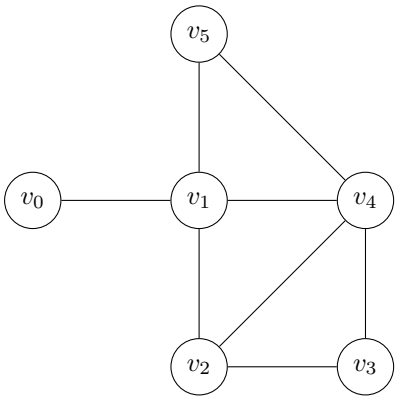
Exercice 5.1.3 Calculer le nombre chromatique du graphe de Petersen, représenté ci-contre à gauche.

5.2 Coloration gloutonne

Soit un graphe $G = (X, E)$ et une $k-$coloration (coloration qui utilise k couleurs) de ce graphe. On peut renommer les couleurs à l'aide des entiers de l'ensemble $\mathcal{C} = \{1, 2, 3, \ldots, k\}$.

La $k-$coloration sera donc entièrement définie par une application *color* de X dans $\mathcal{C}$.
Soit $G = (X = \{a, b, c\}, E = \{ac, cb\})$, on peut définir la 2−coloration de la manière suivante : $color(a) = color(b) = 1$ et $color(c) = 2$.
On peut facilement écrire un algorithme glouton de coloration valide d'un graphe. Mais la coloration obtenue n'est pas forcement optimale.

```
1  def colorationGloutonne(G:graphe):
2  """
3  tout colorier en blanc (0)
4  pour chaque sommet xi du graphe
5      color(xi) est la plus petite couleur non utilisée par
6      les voisins de xi déjà colorés.
7  """
```

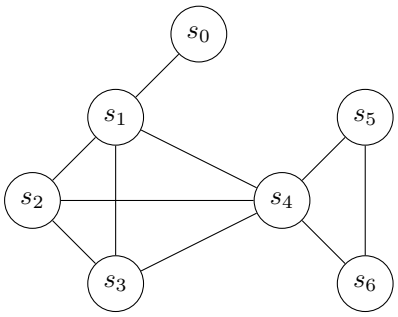


Exercice 5.2.1 Quel est le résultat obtenue pour le graphe G ci-contre avec l'appel `colorationGloutonne(G)` et les ordres de sommets suivants :

- Ordre 1 : $(v_0, v_1, v_5, v_2, v_3, v_4)$
- Ordre 2 : $(v_0, v_5, v_3, v_2, v_1, v_4)$

Une de ces deux $k-$colorations est optimale ? justifier.

Théorème 5.1 *Étant donné un graphe G, il existe au moins un ordre sur les sommets de G tel que la coloration gloutonne calculée à partir de cet ordre soit optimale.*



Exercice 5.2.2 Soit le graphe G ci-contre. Proposer un ordre sur les sommets pour obtenir une coloration gloutonne optimale.

Exercice 5.2.3 Donner un ordre de grandeur du temps nécessaire pour obtenir une coloration gloutonne optimale en prenant la valeur 10^{-9} seconde pour une coloration complète d'un graphe d'ordre $n = 10$, $n = 20$ et $n = 30$. Que peut-on conclure ?

Théorème 5.2 *Étant donné un graphe $G = (X, E)$, $deg_{max}(G)$ le degré maximum des sommets de G et $|K|$ l'ordre du plus grand sous graphe complet K de G. Alors*

$$|K| \leq \chi \leq deg_{max}(G) + 1$$

Exercice 5.2.4 Faire la preuve du théorème (theor. 5.2)

Exercice 5.2.5 Écrire une fonction `colorationGloutonne(G:graphe)` qui prend en paramètre un graphe et qui colorie les noeuds du graphe G en suivant l'algorithme (algo. 5.2) de coloration gloutonne.

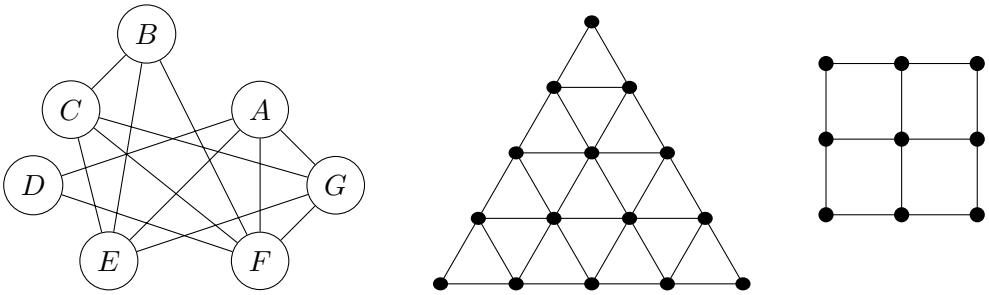
5.3 Graphes 2-coloriables

On dit qu'un graphe est 2-coloriable si son nombre chromatique vaut au plus 2, autrement dit s'il est possible de le colorier avec deux couleurs. Un tel graphe G est aussi appelé *biparti*, ce qui signifie que l'on peut partitionner les sommets en deux ensembles S_1 et S_2 tels que les arêtes de G relient toujours un sommet de S_1 à un sommet de S_2 — on peut donc attribuer la couleur c_1 à tous les sommets de S_1, et la couleur c_2 à tous les sommets de S_2.

Un cycle impair est un cycle de longueur impaire, autrement dit un cycle avec un nombre impair d'arêtes (et donc un nombre impair de sommets si l'on ne compte pas deux fois le sommet de départ qui est aussi le sommet d'arrivée) ; le cycle impair le plus trivial est le triangle.

Théorème 5.3 *Un graphe G est 2-coloriable si et seulement si G ne possède pas de cycle impair.*

Exercice 5.3.1 Parmi les graphes suivant :



1. lesquels sont 2-coloriables ?
2. pour le ou les autres, calculer le nombre chromatique.

5.3.1 Algorithme

Pour tester si un graphe connexe G est coloriable avec deux couleurs données c_1 et c_2, on dispose d'un algorithme qui ressemble au test de connexité :

1. choisir un sommet initial s et le colorier avec la couleur c_1 ;
2. tant qu'il existe un sommet t non coloré avec un voisin coloré :
   a) tester si les voisins de t déjà coloriés ont tous la même couleur ;
   b) si c'est le cas, colorier t avec l'autre couleur ;
   c) sinon, arrêter l'algorithme.

Cet algorithme prouve en même temps le théorème 5.3, c'est l'objet de l'exercice suivant.

Exercice 5.3.2

1. Montrer que lorsque cet algorithme colorie tous les sommets du graphe, celui-ci est bien colorié.
2. Montrer que lorsque l'exécution de l'algorithme se termine prématurément (étape 2.c), un cycle impair a été détecté — le graphe n'est donc pas 2-coloriable.

3. En déduire le théorème.

Exercice 5.3.3

1. Écrire une fonction `effacerCouleurs(G)` qui colorie en blanc tous les sommets de G.
2. Écrire une fonction `sommetColoriable(G)` qui renvoie un sommet de G blanc ayant au moins un voisin non blanc et `None` si un tel sommet n'existe pas.
3. Écrire une fonction `monoCouleurVoisins(s)` qui, si les voisins non blancs du sommet s sont tous de la même couleur, renvoie cette couleur et dans le cas contraire (s a deux voisins de couleurs non blanches différentes) renvoie `None`.
4. Écrire une fonction `deuxColoration(G,c1,c2)` qui tente de colorier le graphe G avec les deux couleurs $c1$ et $c2$ en utilisant l'algorithme détaillé plus haut, et renvoie `True` en cas de succès, `False` sinon.
5. Tester la fonction sur :
   a) des grilles — utiliser la fonction `construireGrille(m,n)`,
   b) des arbres — fonction `construireArbre(degre,hauteur)`,
   c) des graphes bipartis complets — fonction `construireBipartiComplet(m,n)` ; un graphe G est biparti complet, s'il existe une partition de son ensemble de sommets en deux sous-ensembles U et V telle que chaque sommet de U est uniquement relié à chaque sommet de V. Si U est de cardinal m et V est de cardinal n le graphe biparti complet est noté $K_{m,n}$;
   d) le graphe de Petersen,
   e) le cube, l'octaèdre et le dodécaèdre.

   Après chaque essai dessiner le graphe pour vérifier soit qu'il est bien colorié, soit qu'un sommet non coloré t possède deux voisins de couleurs distinctes ; dans ce dernier cas vérifier qu'un cycle impair relie t et certains des sommets déjà coloriés.

Conseils : lorsque la fonction `deuxColoration` découvre un sommet t non coloré avec deux voisins de couleurs différentes (cas d'un graphe non 2-coloriable), afficher ce sommet en ajoutant simplement une instruction `print(t)`.

Les graphes de cette section sont pour la plupart mieux dessinés en spécifiant l'algorithme de dessin comme suit : `dessiner(G,algo='neato')`. Voir http://www.graphviz.org pour une description rapide des algorithmes disponibles avec ce logiciel de dessin de graphes : *dot, neato, fdp, twopi* et *circo*.

Remarque 5.3.1
Pour traiter les arêtes le module de manipulation de graphes contient les fonctions suivantes :

`listeAretesIncidentes(s)`	retourne la liste des arêtes issues du sommet `s`
`sommetVoisin(s,a)`	retourne le voisin du sommet `s` en suivant l'arête `a`
`marquerArete(a)`	marque l'arête `a`
`demarquerArete(a)`	démarque l'arête `a`

Ainsi le fragment de code

```
1 voisins = listeVoisins(s)
2 for i in range(len(voisins)):
3     t = voisins[i]
```

peut être remplacé par :

```
aretesIncidentes = listeAretesIncidentes(s)
for i in range(len(aretesIncidentes)):
    t = sommetVoisin(s,aretesIncidentes[i])
```

ce qui est un peu plus compliqué, mais permet de traiter l'arête a qui relie les sommets s et t. *Note* : si le graphe n'est pas simple plusieurs arêtes incidentes à s peuvent mener au même voisin t, qui dans ce cas apparaît aussi plusieurs fois dans la liste des voisins de s.

Exercice 5.3.4 Écrire une fonction `areteEntre(s1,s2)` qui retourne une arête entre les sommets $s1$ et $s2$ s'il en existe une, `None` sinon. Modifier la fonction `sommetColoriable` pour marquer l'arête qui relie ce sommet non coloré à son voisin coloré ; ne pas oublier de modifier aussi la fonction `effacerCouleurs` pour démarquer les arêtes en même temps.

Tester à nouveau la fonction `deuxColoration` comme dans l'exercice précédent, et dessiner à chaque fois le graphe : que remarque-t-on à propos du sous-graphe formé par les arêtes marquées (et leurs extrémités) ? Lorsque le graphe n'est pas 2-coloriable vérifier qu'il existe toujours un cycle impair dont toutes les arêtes, sauf une, sont marquées.

Exercice 5.3.5 Utiliser la fonction `melange` pour modifier la fonction `sommetColoriable` afin que le résultat ne dépende plus de l'ordre dans lequel sont rangés les sommets du graphe, ni de l'ordre des arêtes incidentes à un sommet.

Une fois cette modification effectuée, appliquer plusieurs fois la fonction `deuxColoration` au dodécaèdre ; après chaque calcul dessiner le graphe et observer quel est le sommet non coloré t avec deux voisins de couleurs différentes qui a stoppé l'exécution de l'algorithme ; observer que le sommet t est situé sur un cycle impair dont toutes les arêtes, sauf une, sont marquées, et que ce cycle n'est pas toujours un pentagone (bonne chance).