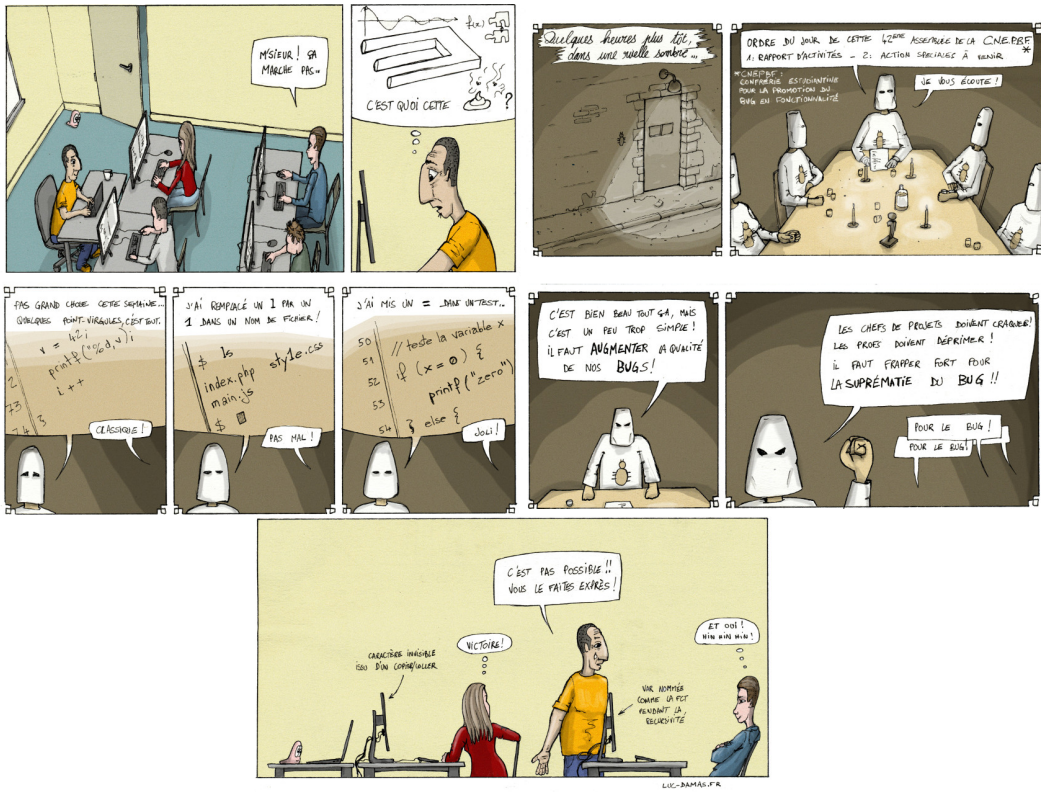
Informatique commune

MP



Lycée Sainte Geneviève

2023-2024

Table des matières

ITC 1

Révisions sur les graphes

1 Notions théoriques sur les graphes

1.1 Vocabulaire général

Graphe non-orienté

Définition

Un graphe non-orienté $G(S, A)$ est la donnée d'un ensemble fini S et d'un ensemble A de parties de S à 2 éléments. Les éléments de S sont appelés **sommets** (ou nœuds) du graphe. Les éléments de A sont appelés **arêtes** du graphe.

Graphe orienté

Définition

Un graphe orienté $G(S, A)$ est la donnée d'un ensemble fini S et d'une partie A de $S \times S$. Les éléments de S sont appelés **sommets** (ou nœuds) du graphe. Les éléments de A sont appelés **arcs** du graphe.

Degré

Définition

Si $G(S, A)$ est un graphe orienté, pour tout arc $(s, s') \in A$, on dit que s est un **prédécesseur** de s' et que s' est un **successeur** de s. Pour tout sommet $s \in S$ on note :
- $d_+(s)$ le nombre de successeurs de s : c'est le **degré sortant** de s ;
- $d_-(s)$ le nombre de successeurs de s : c' est le **degré entrant** de s.

Si $G(S, A)$ est un graphe non orienté, pour tout sommet $s \in S$, on dit que s' est un **voisin** de s si $\{s, s'\} \in A$. On note $d(s)$ le nombre de voisins de s : c'est le **degré** de s.

Chemin et chaîne

Définition

Un arc qui part et arrive au même sommet est une **boucle**.
Un **chemin** (respectivement une **chaîne**) est une suite finie $(s_0, \dots, s_k)$ d'éléments de S telle que pour tout $i \in [\![0, k-1]\!]$, $(s_i, s_{i+1}) \in A$ (resp. $\{s_i, s_{i+1}\} \in A$)

La **longueur** du chemin est le nombre k d'arcs utilisés pour relier s_0 à s_k.
Le chemin (resp. une chaine) est dit **simple** s'il ne passe pas deux fois par un même arc (resp. arête).
Un **circuit** (resp. **cycle**) est un chemin (resp. chaîne) $(s_0, \dots, s_k)$ tel que $s_0 = s_k$. On le qualifie d'élémentaire s'il est simple.

Connexité

Définition

Une **composante connexe** d'un graphe non orienté (S, A) est un sous-ensemble S' de S tel que pour tous $s, t \in S'$ il existe une chaîne reliant s à t dans le graphe en ne passant que par des sommets de S'.
Une **composante fortement connexe** d'un graphe orienté (S, A) est un sous-ensemble S' de S tel que pour tout couple $(s, t) \in S' \times S'$ il existe un chemin de s à t dans le graphe en ne passant que par des sommets de S'.

Un graphe non orienté est dit **connexe** si l'ensemble de ses sommets forme une composante connexe.
Un graphe orienté est dit **fortement connexe** si l'ensemble de ses sommets forme une

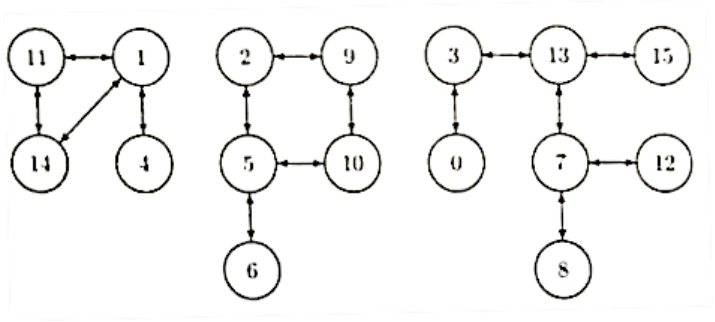
Définition

composante fortement connexe.

Application

ITC1.1 : Graphes non orientés

Considérons le graphe ci-dessous.

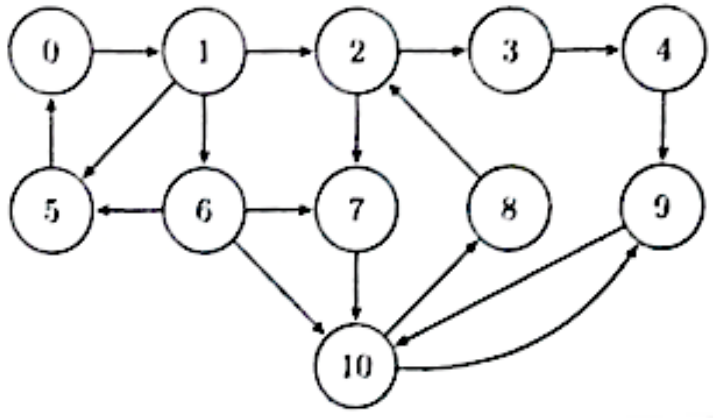


1. Donner une chaîne de longueur 3 reliant le sommet 3 au sommet 12.
2. Donner les degrés maximal et minimal de sommets que l'on peut trouver dans ce graphe.
3. S'il en contient, donner les composantes connexes de ce graphe.

Application

ITC1.2 : Graphes orientés

Considérons le graphe ci-dessous.



Q.1. S'il en contient, donner les composantes fortement connexes de ce graphe.

Q.2. Répondre par vrai ou faux aux questions suivantes :

Q.2.1. Si a et b sont dans une même composante fortement connexe de G, alors il existe un chemin de b vers a et un chemin de a vers b.

Q.2.2. Si a et b sont dans une même composante fortement connexe alors il existe un circuit qui passe par a et b.

Q.2.3. Si a et b sont dans une même composante fortement connexe alors il existe un circuit élémentaire qui passe par a et b.

Q.2.4. S'il existe un circuit qui passe par a et b alors, a et b sont dans une mème composante fortement connexe.

1.2 Listes et matrices d'adjacence

Définition — *Liste d'adjacence*

Soit $G = (S, A)$ un graphe éventuellement orienté et pondéré. Une représentation par **listes d'adjacence** consiste à associer à chaque sommet du graphe la liste de ses successeurs (ou voisins).

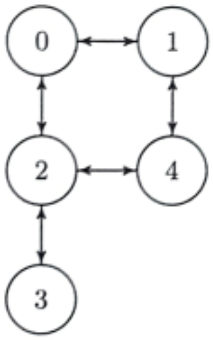
Définition — *Matrice d'adjacence*

La **matrice d'adjacence** associée au graphe non orienté $G = (S, A)$ est la matrice (symétrique) $M_G \in \mathcal{M}_n(\mathbb{R})$ définie par $\begin{cases} m_{ij} = m_{ji} & = 1 \text{ si } \{i, j\} \in A \\ m_{ij} & = 0 \text{ sinon.} \end{cases}$.

La définition s'étend naturellement aux matrices orientées.

Application — *ITC1.3 : Listes et matrices d'adjacence*

On considère le graphe non orienté suivant :



Q.1. Donner une représentation par listes d'adjacence de ce graphe.

Q.2. Donner une représentation par matrice d'adjacence de ce graphe.

On considère le graphe orienté défini par les commandes suivantes :

```
S = [k for k in range(0,5)]
A = [(0,1], [1,4] ,[0,2], [4,2], [2,3]]
```

Q.3. Donner une représentation par liste d'adjacence de ce graphe.

Q.4. Donner une représentation par matrice d'adjacence de ce graphe.

Q.5. Représenter graphiquement cet objet.

Nombre de chemins entre i et j

Soit M^n la puissance usuelle n-ième (avec $n > 1$) de la matrice d'adjacence M d'un graphe (S, A). Alors le coefficient $m_{i,j}^{(n)}$ de M^n contient le nombre de chemins distincts de longueur n du i-ième sommet du graphe au j-ième sommet du graphe.

Démonstration :

Application

ITC1.4 : Décompte de chemins

On reprend le graphe de l'application **??**.

Q.1. Déterminer le nombre de chemins de longueur 1,2 et 3 permettent de relier 2 à 0.

Q.2. En calculant les puissances n-ième de la matrice d'adjacence, vérifier le résultat précédent (on pourra se servir de sa calculatrice ou de Python).

2 Parcours d'un graphe

Différents parcours d'un graphe

Définition

Un **parcours en profondeur** consiste à partir d'un sommet et à explorer d'abord un chemin en ne visitant que des sommets non encore visités, jusqu'à être bloqué et à revenir au sommet précédent pour tenter de découvrir d'autres sommets non encore visités, explorant ainsi un nouveau chemin.
Un **parcours en largeur** consiste à partir d'un sommet et à explorer d'abord tous ses successeurs, avant de visiter tous les successeurs (non encore visités) des successeurs visités lors de la première étape, et ainsi de suite.

Application

ITC1.5 : Parcours de graphe

On considère la situation suivante où vous souhaitez acheter des mangues en utilisant un réseau de connaissances (vos amis, les amis de vos amis, etc.). Suivant les jours, seuls certains d'entre eux sont en mesure de vous vendre le produit désiré.

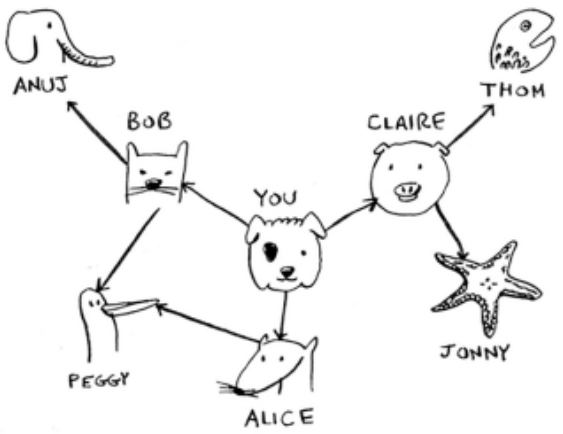


Figure 1.1 – Représentation du réseau relationnel

On considère les 3 situations suivantes :

1. Alice est la seule vendeuse de mangues ;
2. Peggy est la seule vendeuse de mangues ;
3. Anuj et Claire sont les deux seuls vendeurs de manques.

Q.1. Donner, dans le cas d'un parcours en profondeur, le nom du vendeur sélectionné dans chacun des cas ainsi que le nombre de tests effectués. On considère que les parcours sont faits par ordre lexicographique dans un niveau donné.

Q.2. Donner, dans le cas d'un parcours en largeur, le nom du vendeur sélectionné dans chacun des cas ainsi que le nombre de tests effectués.. On considère que les parcours sont faits par ordre lexicographique dans un niveau donné.

2.1 Structures de données adaptées

Pile et file

Définition

Une **pile** est une structure de données dans laquelle des éléments figurent dans un ordre précis, de sorte que seul le dernier élément ajouté, nommé le **sommet** de la pile, est accessible en le retirant de la pile.
Une **file** est, à l'instar d'une pile, une structure de données dans laquelle des éléments figurent également dans un ordre précis, mais cette fois-ci seul le premier élément ajouté, nommé la **tête** de la file, est accessible en le retirant de la file. Le dernier élément mis en place est nommé la **queue** de la file.

En `Python`, on peut réaliser une structure de pile en utilisant des listes qu'on s'interdit de manipuler autrement que par l'initialisation à `[]`. L'ajout se fait en écrivant : `l=[]+l` et nécessite une recopie en $\mathcal{O}(n)$ tandis que la lecture est à coût constant.
Pour les files, l'ajout en fin de file se fait en $\mathcal{O}(1)$ à l'aide de la méthode `append` ou `l=l+[]`. En revanche, nous verrons que l'accès en lecture au dernier élément a un coût linéaire.

Un objet (au programme) présent dans la bibliothèque `collections` est bien plus intéressant et permet une manipulation aisée des files et pile : les **`deque`** (qu'on traduit en français par dèque ou file à double entrée) dont le nom vient de *double-ended queue*, c'est-à-dire une file où les opérations d'ajout et de retrait sont disponibles (et optimisées) à la fois en tête et en queue. On utilisera les commandes suivantes :

Code Python

Code éclu

?? PythonTeX ??

2.2 Parcours en profondeur

On souhaite maintenant écrire un code Python permettant d'afficher tous les sommets d'un graphe représenté par une matrice d'adjacence M (codée par une liste de listes) en partant d'un sommet arbitraire [a].
En premier lieu, il faut réfléchir à la structure de données appropriée :

Complétons le code Python suivant :

```
def parcours_profondeur(M,s0):
    """
    M: matrice d'adjacence sous forme d'une liste de
   listes
    s: sommet de départ
    sortie: aucune
    affiche tous les sommets de M, en partant du sommet i
    """
    n=len(M)
```

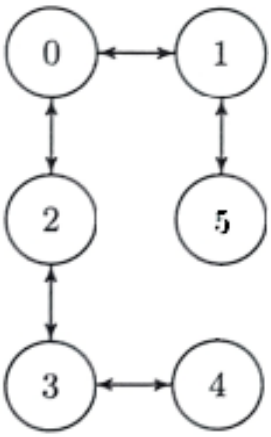
a. on considère que les n sommets du graphe sont les entiers $0, \dots, n-1$; l'utilisation d'un dictionnaire permet de facilement se rapporter à ce cas.

2.3 Parcours en largeur

Le code à utiliser est ici très proche du précédent :

```
def parcours_largeur(M,s0):
    """
    M: matrice d'adjacence sous forme d'une liste de listes
    s: sommet de départ
    sortie: aucune
    affiche tous les sommets de M, en partant du sommet i
    """
    n=len(M)
```

On peut valider les résultats des deux parcours sur l'exemple suivant :



Deux propriétés importantes découlent des parcours précédents.

Connexité d'un graphe non orienté .

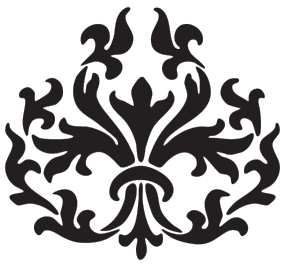
Remarque importante

Un graphe non orienté est connexe si, et seulement si, un algorithme de parcours depuis n'importe quel sommet renvoie un objet dont la taille est le nombre de sommets.

Présence d'un circuit dans un graphe orienté .

Remarque importante

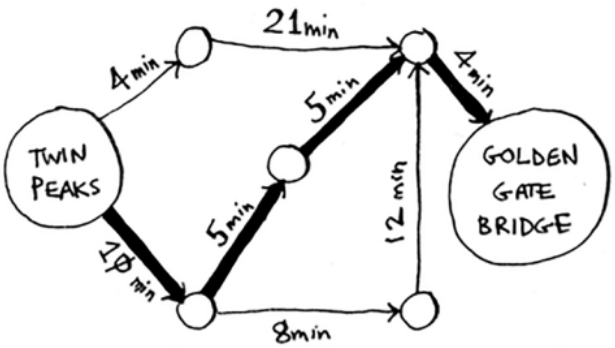
Un graphe orienté possède un circuit, si et seulement si, un algorithme de parcours depuis au moins un sommet renvoie un objet contenant ce sommet.



3 Recherche d'un plus court chemin

3.1 Présentation du problème

On considère le plan de transports en commun très simplifié suivant, indiquant de façon sommaire les durées entre les arrêts.



Deux situations peuvent se présenter :

- on souhaite minimiser le nombre de correspondance
- on souhaite minimiser le temps de parcours (on considérant que le temps de correspondance est pris en compte dans les durées présentées)

3.2 Cas d'un graphe non pondéré

Minimiser le nombre d'étapes pour aller d'un point A à un point B est une légère variation du parcours en largeur mené précédemment : il suffit de tester pour chaque sommet rencontré si c'est celui que l'on recherche.
On donnera deux versions du code : la première se contente de tester si un chemin existe entre deux sommets passés en arguments, la seconde retourne le nombre d'étapes minimal entre ces deux sommets.

```
def minimise_etapes(M,s_a,s_b):
    """
    M: matrice d'adjacence sous forme d'une liste de
   listes
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: bool. qui détermine si un chemin existe de A à
    B
    """
    n=len(M)
```

```
def minimise_etapes_compte(M,s_a,s_b):
    """
    M: matrice d'adjacence sous forme d'une liste de
   listes
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: nombre d'étapes pour aller de A à B
    """
    assert minimise_etapes(M,s_a,s_b)
    n=len(M)
```

3.3 Cas d'un graphe pondéré : algorithme de Dijkstra

3.3.1 Notion de graphe pondéré

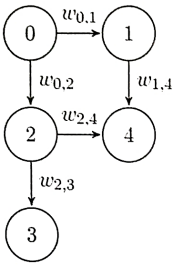
Avant de détailler l'algorithme proprement dit, il nous reste à introduire la notion de graphe pondéré.

Graphe pondéré

Définition

Un graphe $G(S, A)$ **pondéré** est un graphe dont les arcs (ou les arêtes) sont étiquetés par un nombre entier appelé **poids**. Cette donnée supplémentaire peut se représenter par une fonction de A vers $\mathbb{Z}$.
Le **poids d'un chemin** dans un graphe pondéré est la somme des poids des arcs qui forment ce chemin.

La représentation par matrice d'adjacence permet très facilement d'indiquer les poids de chaque arc :



$\leftrightarrow$

$$\begin{bmatrix} \star & w_{0,1} & w_{0,2} & \star & \star \\ \star & \star & \star & \star & w_{1,4} \\ \star & \star & \star & w_{2,3} & w_{2,4} \\ \star & \star & \star & \star & \star \\ \star & \star & \star & \star & \star \end{bmatrix}$$

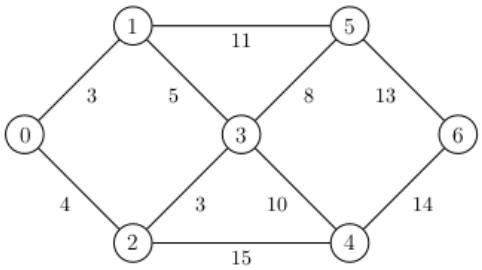
3.3.2 Principe de la méthode

L'algorithme de Dijkstra permet de trouver le chemin le plus court entre deux sommets d'un graphe non orienté pondéré lorsque **tous les poids sont positifs**. Il suit le principe d'un parcours en largeur, mais au lieu de prendre le sommet suivant dans la file, on traite d'abord celui dont la distance à l'origine est la plus petite. Ainsi, une fois qu'un sommet est traité, sa distance à l'origine ne peut plus diminuer (car les poids sont positifs). Dit autrement : si pour aller de A à B le chemin le plus court passe par I, alors la partie du chemin entre A et I est le plus court chemin entre A et I et la partie du chemin entre B et I est le plus court chemin entre B et I. On optimise donc le choix à chaque étape : c'est un algorithme glouton.

Application

ITC1.6 : Mise en œuvre à la main de l'algorithme de Dijkstra

On considère le graphe pondéré suivant :



Étape	0	1	2	3	4	5	6
0	0	∞	∞	∞	∞	∞	∞
1							
2							
3							
4							
5							
6							
7							

Remplir le tableau suivant correspondant aux 6 itérations de la méthode en considérant que le sommet d'origine est le 0.

3.3.3 Code

Nous allons ici nous contenter d'une version naïve qui utilise des listes.
Dans la première version du code, on veut simplement retourner le poids d'un chemin reliant deux sommets.

```
def dijkstra(M,s_a,s_b):
    """
    M: matrice d'adjacence pondérée (liste de listes)
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: coût minimal pour aller de s_a à s_b
    """
    assert minimise_etapes(M,s_a,s_b)
    n=len(M)
    poids_sommets={i:float('inf') for i in range(n)}
    dic_sommets_minimises={i:0 for i in range(n)}
```

Dans une seconde version, on veut retourner le chemin suivi pour aller de A à B. Pour cela il suffit de modifier légèrement le programme précédent, en construisant au fur et à mesure une liste des antécédants des points traités.

```
def dijkstra_chemin(M,s_a,s_b):
    """
    M: matrice d'adjacence pondérée (liste de listes)
    s_a: sommet de départ
    s_b: sommet d'arrivée
    sortie: coût minimal pour aller de s_a à s_b,chemin suivi
    """
    assert minimise_etapes(M,s_a,s_b)
    n=len(M)
    poids_sommets={i:float('inf') for i in range(n)}
    dic_sommets_minimises={i:0 for i in range(n)}
    dic_antecedants={i:0 for i in range(n)}
```

Remarque : On peut améliorer le code en utilisant une variante des files : les **files de priorité**. Les défilements des éléments ne sont plus décidés par leur ordre d'enfilement, mais par un entier naturel associé : leur priorité.
Pour nous, cette priorité est la distance du sommet à l'origine : on veut traiter en priorité le sommet restant dont la distance à l'origine est la plus faible. On peut aussi mettre à jour la priorité des éléments au fur et à mesure de l'avancement.
La fonction `remove` du type `deque` permet ce type de manipulation facilement.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it. If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.