

# Feuille de TP Python 04

Graphes, Chaînes de MARKOV, parcours et algorithme de DIJKSTRA

L'objet de ce TP est d'une part de revenir sur les aspects les plus élémentaires de traitement des graphes via la construction d'une couche d'abstraction et d'un module Python et d'autre part de montrer quelques exemples d'utilisation des parcours de graphes.

## 1 Qu'est-ce qu'un graphe ?

Si l'on dispose de  $n$  objets, un graphe (dans sa version la plus simple)  $G$  sur ces  $n$  objets est une structure reflétant le fait que deux de ces objets sont reliés ou pas.

On représente graphiquement ( cf. figure 1) un tel objet dans le plan en plaçant un point pour chaque objet et une ligne (une arête) entre deux objets reliés. Un graphe est donc décrit par l'ensemble de ses sommets et l'ensemble des arêtes.

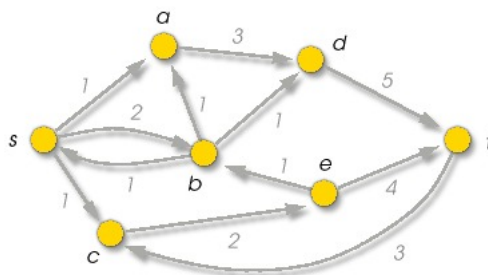


FIGURE 1 – Une représentation graphique de graphe

On peut enrichir cette structure en

- plaçant un *poids* sur chaque arête entre deux sommets <sup>1</sup>,
- en considérant que la relation n'est pas symétrique et donc, pour chaque paire de sommets  $A$  et  $B$  considérer deux arêtes, celle allant de  $A$  vers  $B$  et celle allant de  $B$  vers  $A$  <sup>2</sup>.

La structure de graphe est une des structures les plus fondamentales de l'informatique. Elle est très utilisée par exemple en bio-informatique dans l'analyse de génomes.

On rencontre des graphes de manière évidente et explicite lorsque l'on traite de réseaux routiers, de réseaux électriques mais aussi de manière cachée et implicite lorsque l'on traite de jeux. Dans ce dernier cas, les configurations du jeu sont les sommets du graphe et on relie un sommet à un autre si les règles du jeu permettent de passer de la première configuration à l'autre.

### 1.1 Structure de dictionnaire dict

Le moyen le plus efficace en Python de faire correspondre les noms/labels des sommets du graphe avec des nombres entiers (indices de listes) est d'utiliser la structure de dictionnaire.

Un dictionnaire est une liste d'objets de type quelconque dont l'indichage se fait à l'aide d'identifiants (les « clés »).

1. Un dictionnaire vide se déclare par `D=dict()` ou `D={}`,
2. On peut ajouter des entrées à un dictionnaire à l'initialisation par des syntaxes proches de celles de liste où la paire `[]` est remplacée par la paire `{}`, par exemple,
  - un mini-dictionnaire des verbes anglais et de leurs traductions françaises pourrait se déclarer par :

```
D = {"eat":"manger", "sleep":"dormir", "drink":"boire"}
```
  - Une liste faisant correspondre les carrés de nombres entiers avec leurs racines carrées peut se déclarer en extension par :

```
SQ = { i**2 : i for i in range(N) }
```

1. On ajoute donc une fonction poids qui à chaque arête associe un nombre réel, on peut penser à un réseau électrique en courant continu où les arêtes sont les fils, les sommets, les points de jonction de ces fils, les poids, les résistances de chacun de ces fils

2. Je connais Barack OBAMA mais il ne me connaît pas. Par ailleurs, je le connais moins bien que l'un d'entre vous, on peut mettre un poids sur la relation caractérisant le degré de connaissance

3. On peut ajouter des entrées à un dictionnaire existant par : `D["shout"]="crier"`
4. On peut récupérer la valeur correspondant à une clé suivant une syntaxe similaire : `verbe = D["shout"]`
5. La liste (ce n'est pas une liste Python, mais on peut boucler dessus) des clés du dictionnaire `D` est `D.keys()` :  
 p.ex. `Cles=[k for k in D.keys() if D[k]!="crier"]` fabrique une liste contenant toutes les clés du dictionnaire `D` dont la valeur correspondante n'est pas "crier".  
 On peut tester qu'une clé est présente dans le dictionnaire par l'expressio à valeurs booléennes `k in D.keys()`
6. On peut éliminer une entrée du dictionnaire avec `del`, p.ex. `del(D["shout"])`
7. Finalement, les clés peuvent être de tout type d'objet Python dit « immutable ». Les entiers `int`, les chaînes de caractères `str`, les `t-uples` de tels objets sont des clés de dictionnaire valide. Une liste `list` n'est pas une clé de dictionnaire valide. Pour pouvoir utiliser une liste en tant que clé, on la transforme d'abord en `t-uple` d'objets eux-même « immutables »

**Exercice 1.**—On dispose d'une carte ferroviaire de villes européennes ( cf. fig 2); On dit que deux de ces villes sont reliées s'il existe une liaison train les reliant directement.

1. Quelle(s) structure(s) de données<sup>3</sup> Python peuve(nt) permettre d'encoder le graphe  $G$  de la relation « reliées directement par le train » ?

2. Comment affiner cette représentation pour y inclure les temps de trajet ?



FIGURE 2 – Carte Interrail de liaisons européennes.

Voir le script `python/corrections/interrail-ebauche.py` pour des idées d'implémentation.

## Travail à faire

**Exercice 2.**—On cherche à développer un module Python, nommé `graphes.py` comportant les fonctions nécessaires à l'implémentation de graphes à poids. On suppose que l'on utilise un codage de graphe par *matrice d'adjacence*. Un graphe  $G$  dans ce module est une liste `[M,D]` de deux termes où  $M$  est la matrice d'adjacence du graphe et  $D$  est le dictionnaire établissant la correspondance "vrai nom de sommet"  $\mapsto$  "indice valable pour la matrice".

1. Charger le fichier `graphes.py` dans votre IDE (Spyder ou Pyzo/IEP) et écrire les corps des fonctions `NewGraphe()`, `CreerArete()`, `ValeurArete()` et `ListeVoisins()` selon les spécifications indiquées dans leur `docstring`.

Le but est ensuite de pouvoir utiliser un graphe uniquement via ces fonctions, sans se préoccuper de la structure interne du graphe (ici une matrice d'adjacence).

2. Sauver le fichier ainsi complété dans votre dossier de travail et exécuter.

**Exercice 3.**—Construction d'un graphe aléatoire.

1. On se donne  $p \in ]0, 1[$ . Décrire comment construire un graphe sur un ensemble de sommets  $S = \{s_1, \dots, s_n\}$  de sorte que chaque paire de sommets (distincts) soit reliée avec probabilité  $p$ , indépendamment des autres paires.

2. Ecrire dans `graphes.py`, une fonction `GrapheAleatoire(G,p,sym=False)` qui retourne le graphe obtenu à partir de  $G$  en éliminant chacune des arêtes déjà présentes avec probabilité  $1 - p$ . Si `sym` est `True`, le faire de manière symétrique.

3. Il y a plusieurs possibilités : On va travailler la représentation en matrice d'adjacence mais on peut utiliser aussi une représentation en dictionnaire dont les clés sont les arêtes, *i.e.* les couples de sommets, et les valeurs sont les poids.

**Exercice 4.**—Chaînes de MARKOV : un exemple.

1. Dans le script `graphes.py`, écrire le corps de la fonction `VoisinAuHasard` de façon à respecter sa `docstring`.

On pourra utiliser la fonction `choice` du sous-module `numpy.random`.

Pour les questions suivantes, on utilise un nouveau script nommé `madknight.py`, dans le même répertoire qui importe le module `graphes.py` à l'aide de l'instruction d'importation `import graphes as gr`.

2. Commenter la fonction de construction de graphe `K` qui retourne un graphe :

— ayant comme sommets les cases d'un échiquier;

— on place une arête entre deux cases de cet échiquier si on peut passer de l'une à l'autre par un mouvement de cavalier.

On devra utiliser `gr.NewGraphe` et `gr.CreeArete` pour créer le graphe puis les arêtes.

3. Donner une estimation —en construisant une fonction adéquate— du temps moyen (nombre de coups) que met un cavalier noir partant de sa position initiale pour arriver à une case sur le bord blanc. Ce cavalier est fou au sens où à chaque mouvement, il se déplace sur l'une quelconque des cases accessibles au coup suivant.

## 2 Recherches de chemins optimaux

### 2.1 Problématique

15 Etant donné un graphe  $G = (V, E)$  à poids, où  $V$  est l'ensemble des sommets et  $E$  l'ensemble des arêtes, un problème fondamental est de trouver un chemin de poids minimal reliant deux sommets.

On peut penser à un calcul d'itinéraire entre deux villes de longueur minimale (le poids d'une arête est alors sa longueur) ou engendrant une consommation moindre (le poids d'une arête est alors la quantité de carburant consommée lors du parcours de cette arête) mais aussi au calcul de l'itinéraire optimal d'un paquet de données sur le réseau internet de façon à éviter la congestion du réseau.

Résumons :

1. Le graphe  $G = (V, E)$  est une donnée du problème, de même que les poids<sup>4</sup>  $w(e) \in [0, +\infty[$  de chaque arête  $e \in E$ .

2. Un chemin<sup>5</sup>  $P$  est une suite d'arêtes  $e_1, \dots, e_n$  consécutives, *i.e.* telle que pour tout  $k$ , la source de  $e_{k+1}$  soit le but de  $e_k$ .

3. Le poids d'un tel chemin est la somme des poids des arêtes le constituant  $w(P) = \sum_{i=1}^n w(e_i)$ .

25 4. Problème : étant donné un sommet source  $v_s$  et un sommet final  $v_b$ , trouver un chemin  $P = (e_1, \dots, e_n)$  tel que  $v_s$  est la source de  $e_1$ ,  $v_b$  est le but de  $e_n$  et tel que  $w(P)$  soit minimal parmi tous les chemins vérifiant ces propriétés.

### 2.2 Un cas « simple » : Parcours en largeur

Le cas général, avec un système de poids quelconques, se résout à l'aide d'un algorithme complexe, l'algorithme de DIJKSTRA<sup>6</sup>. Le cas où tous les poids placés entre sommets reliés sont égaux est plus simple. Il s'agit de trouver un chemin et comme par hasard, le chemin trouvé par la méthode proposée ci-dessous est de poids minimal.

Si on se contente de trouver un chemin ou, de manière presque équivalente de trouver tous les sommets reliés à un sommet source, on peut procéder par « cercles concentriques » à partir de la source jusqu'à trouver le but.

C'est ce que permet de faire l'*algorithme de parcours en largeur* qui utilise une structure de file<sup>7</sup>.

En voici une description implémentable directement.

35 1. Structures :FIFO une file,  $R$  un dictionnaire, valeurs et clés sont des sommets du graphe.  $R[v] = w$  signifie que dans le parcours partant de  $v_s$ , le prédécesseur de  $v$  sur le chemin menant de  $v_s$  à  $v$  est  $w$ .  $v$  est une clé de  $R$  signifie que l'on dispose d'un chemin menant de  $v_s$  à  $v$ .

2. Initialisation.  $R[v_s] = v_s$  (Convention : sauf problème, le sommet est le seul dont le prédécesseur est lui-même) et enfiler  $v_s$  sur FIFO

40 3. Boucle. Tant que  $v_b$  n'est pas une clé de  $R$  :

(a) défiler  $v$  de FIFO

(b) Pour chaque voisin  $v'$  de  $v$  qui n'est pas encore une clé de  $R$ ,

i. enfiler  $v'$  sur FIFO;

ii.  $R[v'] = v$

45 4. Conclusion. On calcule l'itinéraire en remontant le dictionnaire  $R$  à partir du but.

Etant donné un graphe, pour calculer le plus court chemin d'un sommet source  $v_s$  à un sommet but  $v_b$  si les poids sont tous à 1 on peut utiliser un parcours en largeur à partir de  $v_s$  et s'arrêter dès que l'on rencontre  $v_b$ .

4. en anglais : weight

5. en anglais : path

6. dont le nom, avec sa prononciation correcte, est au programme

7. Une file (*queue* en anglais, à ne pas confondre avec l'anglais *file* qui signifie fichier) FIFO (first in, first out) est une structure sur laquelle deux opérations sont permises : mise en attente au bout de la file (enfiler) et passage de celui qui attendu le plus longtemps (défiler). On peut la comparer avec une pile LIFO (last in, first out), pour laquelle deux opérations sont permises : mise en attente au sommet de la pile (empiler) et passage de celui en haut de la pile (dépiler). Ces deux structures se codent facilement à l'aide de listes en Python et des méthodes `.append()` et `.pop(0)/.pop()`

## 2.3 Travail à faire

### Exercice 5.—

1. Dans le script `graphes.py`, comprendre et commenter le corps de la fonction `ParcoursLargeur`.
  2. Copier le script `madknight.py` en un script `smartknight.py`.
  - 5 **2.a.** écrire une fonction `NbreDeCoupsSmart` permettant, étant données une position de départ du cavalier et une position d'arrivée, de déterminer le nombre minimal de coups qu'il doit faire pour aller du départ à l'arrivée.
  - 2.b.** adapter une fonction de `madknight.py` permettant, étant données une position de départ du cavalier et une position d'arrivée, de déterminer le nombre moyen de coups que fait un cavalier fou pour aller du départ à l'arrivée.
  - 2.c.** Comparer (par exemple pour traverser l'échiquier de part en part en diagonale) les deux méthodes.
- 10 **Exercice 6.**—On dispose d'une image numérisée comportant  $\ell$  lignes et  $c$  colonnes et on considère le graphe  $G$  tel que
- les sommets sont les pixels  $P$  de l'image repérés par leurs coordonnées
  - deux sommets  $P$  et  $P'$  sont adjacents si
    1. Ils sont géométriquement côte à côte ;
    2. Ils sont de la même couleur.
- 15 **1.** Charger le fichier `graphe-image.py` dans votre IDE et écrire les corps des fonctions `NewGraphe()`, `ValeurArete()` et `ListeVoisins()` selon les spécifications indiquées dans leur `docstring`.
- 2.** Sauver le fichier ainsi complété dans votre dossier de travail et exécuter.
  - 2.a.** Pourquoi l'Australie n'est-elle pas complètement bleue ?
  - 2.b.** Décrire comment dans quel ordre se fait la construction du dictionnaire permettant le coloriage.
- 20 **3.** Décommenter le dernier bloc de commentaires et exécuter. Pourquoi le chemin de Tanger à Gibraltar ne prend-il pas vraiment de diagonale ?

## 2.4 Dijkstra

L'algorithme de DIJKSTRA calcule un plus « léger<sup>8</sup> » chemin en prenant en compte les poids sur les arêtes. Il peut se décrire de la façon suivante. On suppose qu'il y a  $N \geq 2$  sommets et que  $v_s$  et  $v_b$  sont dans la même composante connexe<sup>9</sup> du graphe  $G$ .

25 L'algorithme comporte une initialisation et au maximum  $N - 1$  étapes. Tout au long du calcul on maintient deux ensembles et deux tableaux<sup>10</sup> :

1.  $NT$  l'ensemble des sommets non encore traités. A l'initialisation  $NT = V \setminus \{v_s\}$ .
2.  $T$  l'ensemble des sommets dont un chemin de poids minimal à la source est connu. A l'initialisation  $T = \{v_s\}$ . L'algorithme s'arrête dès que  $v_b \in T$ .
- 30 **3.**  $W$  indexé par tous les sommets (sauf  $v_s$ ).  $W[v]$  contient au cours du calcul le poids d'un plus léger chemin de  $v_s$  à  $v$  connu à ce moment. En fin d'algorithme  $W(v_b)$  contient donc le poids du chemin le plus léger de  $v_s$  à  $v_b$ . A l'initialisation  $W[v]$  vaut  $w(v_s, v)$  si  $v_s$  et  $v$  sont voisins,  $+\infty$  sinon.
- 4.**  $R$  indexé par tous les sommets (sauf  $v_s$ ).  $R[v]$  contient le sommet qui le précède dans un plus léger chemin menant de  $v_s$  à  $v$  connu à ce moment. En fin d'algorithme, partant de  $R[v_b]$  et remontant la chaîne, on construit un itinéraire optimal. **A l'initialisation,  $R[v]$  contient  $v_s$  pour tout  $v$ .**

35

A l'étape  $n$ .

1. On choisit un sommet  $v$  dans  $NT$  qui réalise le minimum de  $W$  parmi les sommets de  $NT$ .
2. Ce sommet  $v$  est retranché de  $NT$  et ajouté à  $T$ .
3. On met à jour les tableaux  $W$  et  $R$  de la façon suivante. On parcourt tous les sommets  $v'$  voisins de  $v$ , si  $W[v] + w(v, v') < W[v']$  alors
  - (a)  $W[v']$  devient  $W[v] + w(v, v')$
  - (b)  $R[v']$  devient  $v$ .

40

On conclut comme dans le parcours en largeur en remontant  $R$  à partir de  $v_b$  pour reconstituer l'itinéraire « à l'envers ».

---

8. Habituellement on parle du chemin le plus « court ». Pour que cette interprétation en termes de longueur soit valide, la matrice des poids doit satisfaire certaines inégalités (inégalité triangulaire !). Comme on place des « poids » sur les arêtes, l'interprétation en termes de masse semble plus appropriée.

9. Deux points d'un graphe sont dans une même composante connexe s'il existe un chemin dans le graphe les reliant

10. Dans l'implémentation écrite dans `graphes.py`, on utilise trois dictionnaires

## 2.5 Travail à faire

**Exercice 7.**—Comprendre l’application à la main l’algorithme de DIJKSTRA faite dans le tableau suivant pour calculer un plus léger chemin de  $s$  à  $t$  dans le graphe décrit par la figure 1.

étape	NT	T	W[a]	W[b]	W[c]	W[d]	W[e]	W[t]	R[a]	R[b]	R[c]	R[d]	R[e]	R[t]
0	abcdet	s	1.0	2.0	1.0	$\infty$	$\infty$	$\infty$	s	s	s	s	s	s
1	abdet	sc	1.0	2.0	1.0	$\infty$	3.0	$\infty$	s	s	s	s	c	s
2	bdet	sac	1.0	2.0	1.0	4.0	3.0	$\infty$	s	s	s	a	c	s
3	det	sabc	1.0	2.0	1.0	3.0	3.0	$\infty$	s	s	s	b	c	s
4	dt	sabce	1.0	2.0	1.0	3.0	3.0	7.0	s	s	s	b	c	e
5	t	sabcde	1.0	2.0	1.0	3.0	3.0	7.0	s	s	s	b	c	e

5 **Exercice 8.**—On continue le développement du module `graphes` entamé dans la partie précédente en analysant le code de la fonction `Dijkstra`.

Lire et surtout commenter abondamment la fonction Python `Dijkstra` du module `graphes.py`. Spécifier les objets passés en paramètres et décrire la façon de coder les différents tableaux et ensembles de l’algorithme.

**Exercice 9.**—On s’intéresse à l’utilisation du module `graphes` afin de construire un petit logiciel permettant de calculer le plus court (en durée) itinéraire reliant deux stations de métro.

1. Charger dans votre IDE les modules `graphes.py` et `metrodat.py` et le programme `metro.py` et les sauver dans votre dossier de travail.

1.a. Que fait le programme ? Exécutez le <sup>11</sup>.

1.b. Comment est déclaré le graphe du métro dans le module `metrodat.py` ?

15 2. Rendre le programme `metro.py` interactif.

3. Faire un site web avec et gagner des millions.

## 3 Annexes

### 3.1 Dictionnaires

<code>D=dict()</code>	fabrique un dictionnaire vide à remplir
<code>D = {'a': 65, 'b': 66}</code>	fabrique un dictionnaire avec un préremplissage du type clé : valeur
<code>len(D)</code>	donne le nombre d’éléments du dictionnaire
<code>D[k] = v</code>	affecte la valeur $v$ à l’élément de $D$ de clé $k$ , le crée s’il n’existe pas encore
<code>del(D[k])</code>	élimine l’entrée de clé $k$ dans le dictionnaire $D$
<code>D[k]</code> (dans une expression)	s’évalue à la valeur de l’élément de $D$ de clé $k$
<code>E = D</code>	Deux étiquettes $E$ et $D$ sur le même dictionnaire en mémoire
<code>E = D.copy()</code>	l’étiquette $E$ est placée sur un clone de $D$
<code>D.keys()</code>	retourne un itérable contenant les clés du dictionnaire $D$ , pas d’ordre <i>a priori</i>
<code>sorted(list(D.keys()))</code>	retourne une liste <i>triée</i> des clés du dictionnaire $D$
<code>D.items()</code>	retourne un itérable contenant les couples (clés, valeurs) du dictionnaire $D$ , pas d’ordre <i>a priori</i>
<code>for k in D.keys() :</code> ou <code>for k in D :</code>	boucle sur les clés de $D$
<code>for k,v in D.items() :</code>	boucle sur les couples clé, valeur de $D$
<code>D = { f(i) : g(i) for i in L }</code>	Compose un dictionnaire par une formule en bouclant sur une liste
<code>D = { g(i) : f(i) for i in L }</code>	Réciproque du dictionnaire précédent

20 11. Si votre module `graphes.py` est fonctionnel, ceci ne doit pas poser de problèmes.

## 3.2 Listes

<code>L=[]</code> ou <code>L=list()</code>	fabrique une liste vide à remplir
<code>L=[1.0, 'aa']</code>	fabrique une liste avec un préremplissage
<code>len(L)</code>	donne le nombre d'éléments de la liste, ils sont indicés de 0 à <code>len(L) - 1</code>
<code>L[i]</code>	donne la valeur de l'élément de L d'indice $i$ si $0 \leq i \leq \text{len}(L) - 1$
<code>M=L[i:j]</code>	extraite une sous-liste $M$ dont les éléments ceux de L sont indicés de $i$ à $j - 1$
<code>L[i:j]=M</code>	remplace la sous-liste des éléments de L indicés de $i$ à $j - 1$ par la liste M
<code>M=L</code>	Les deux étiquettes M et L sont sur la même liste en mémoire
<code>M=L[:]</code> ou <code>M = L.copy()</code>	créé une copie de L et lui affecte l'étiquette M
<code>L[-1]</code>	donne le dernier élément de la liste
<code>L.append(obj)</code>	ajoute l'objet <code>obj</code> en fin de liste
<code>L.extend(M)</code>	fusionne les listes L et M en plaçant M à la fin de L
<code>L.insert(index,obj)</code>	insère l'objet <code>obj</code> avant l'indice $i$ , décale le reste de L
<code>L[i:i] = M</code>	insère la liste M à l'indice $i$ de la liste L, décale le reste de L
<code>L.pop()</code>	efface le dernier élément de la liste L et le retourne
<code>L.pop(i)</code>	efface l'élément $i$ de la liste L et le retourne
<code>L.reverse()</code>	inverse <i>en place</i> l'ordre des éléments de la liste
<code>L.sort()</code>	trie <i>en place</i> les éléments de la liste, pourvu que l'on puisse comparer les éléments
<code>M=sorted(L)</code>	M devient une copie triée de la liste L
<code>V=np.array(L)</code> ou <code>V=np.asarray(L)</code>	V devient un vecteur numpy composé avec les éléments de la liste de nombres L. Si L est une liste de listes de nombres, compose une matrice numpy.
<pre>for obj in L :</pre> <p>Exemple :</p> <pre>for k in L:     print(k**2)</pre>	<p>boucle sur les éléments de L, ne modifie pas L. Modifier L dans la boucle peut avoir des conséquences inattendues.</p> <p>Si on prend <code>L=[1,2,4]</code> affiche successivement 1, 4, 16.</p>
<pre>M = [expression(obj) for obj in L]</pre> <p>Exemple :</p> <pre>M=[k+1 for k in L]</pre>	<p>fabrique une liste en bouclant sur les éléments de L et en leur appliquant <code>expression()</code>, ne modifie pas L.</p> <p>Si on prend <code>L=[1,2,4]</code> fabrique la nouvelle liste <code>[2,3,5]</code> et l'affecte à M.</p>
<pre>M = [expr(obj) for obj in L if cond(obj)]</pre> <p>Exemple :</p> <pre>M=[k**2 for k in L if k&gt;1]</pre>	<p>fabrique une liste en bouclant sur les éléments de L, ne gardant que ceux vérifiant <code>cond()</code> et en leur appliquant <code>expr()</code>, ne modifie pas L.</p> <p>Si on prend <code>L=[1,2,4]</code> fabrique la nouvelle liste <code>[4,16]</code> et l'affecte à M.</p>