Feuille de TP Python 05

Graphes et algorithme de DIJKSTRA

L'objet de ce TP est essentiellement de mettre en place les aspects les plus élémentaires de traitement des 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 (*c.f.* 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.

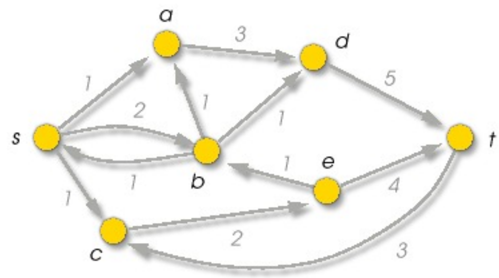

FIGURE 1 – Une représentation graphique de graphe

On peut enrichir cette structure en

— mettant un *poids* sur chaque arête entre deux sommets [1],

— 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 [2].

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 numéroter les noms/labels des sommets du graphe avec des nombres entiers (ce qui peut-être utile, par exemple en calculant en nombres entiers puis en utilisant la correspondance pour revenir aux "vrais" noms, qui ont du sens) est d'utiliser la structure de dictionnaire.

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

1. Un dictionnaire vide se déclare par `D=dict()` ou `D={}`,
2. On peut remplir un dictionnaire à l'utilisation par des syntaxes proches de celles de liste où la paire `[]` et 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"}`
3. On peut ajouter des entrées à un dictionnaire existant par : `D["shout"]="crier"`
4. On peut récupère la valeur correspondant à une clé suivant une syntaxe similaire : `verbe = D["shout"]`

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 connait 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

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 `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 chaines de caractères `str`, les t-uples de tels objets sont des clés de dictionnaire valide.

Exercice 1.—On dispose d'une carte ferroviaire de villes européennes (*c.f.* 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[3] 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()`, `CreeArete()`, `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`, avant la fonction `Dijkstra`, 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 `sim` 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.

2 Plus court chemin avec poids : DIJKSTRA

2.1 Problématique

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 poids [4] d'une arête $e \in E$ est un nombre réel positif $w(e)$. C'est une donnée du problème, de même que le graphe $G = (V,E)$.
2. Un chemin [5] P est une suite d'arêtes $e_1,\ldots,e_n$ telle que deux arêtes consécutives ont un sommet commun.
3. Le poids d'un tel chemin est la somme des poids des arêtes le constituant
$$w(P) = \sum_{i=1}^{n} w(e_i)$$
4. Le problème se reformule ainsi : étant donnés un sommet source v_s et un sommet final v_f, trouver un chemin $P = (e_1,\ldots,e_n)$ tel que v_b est une extrémité de e_1, v_f est une extrémité de e_n et tel que $w(P)$ soit minimal parmi tous les chemins vérifiant ces propriétés.

2.2 Un cas plus simple : Parcours en largeur

Avant de trouver le chemin le plus court, on peut déjà se contenter 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 » et c'est ce permet de faire l'algorithme de parcours en largeur qui utilise une structure de file [6].

Etant donné un graphe, pour calculer le plus court chemin d'un sommet source v_s à un sommet but v_f 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_f.

En voici une description implémentable directement.

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 prédecesseur de v sur le chemin menant de v_s à v est w. v est une clé de R signfie 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 prédecesseur est lui-même) et enfiler v_s sur `FIFO`
3. Boucle. Tant que v_f 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. $R[v'] = v$
4. Conclusion. On calcule l'itinéraire en remontant le dictionnaire R à partir du but.

2.3 Travail à faire

Exercice 4.—On dispose d'une image numérisée comportant ℓ 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.

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`.

4. en anglais : weight
5. en anglais : path
6. 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 atente 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. 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.
3. Décommenter le dernier bloc de commentaires et exécuter. Pourquoi le chemin de Tanger à Gibraltar ne prend-il pas vraiment de diagonale ?

Exercice 5.—Dans le script `graphes.py`, comprendre et commenter le corps de la fonction `ParcoursLargeur`.

2.4 Dijkstra

L'algorithme de DIJKSTRA calcule un plus léger chemin en prenant en compte les poids su rles 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_f sont dans la même composante connexe [7] du graphe G.

L'algorithme comporte une initialisation et au maximum $N-1$ étapes. Tout au long du calcul on maintient deux ensembles et deux tableaux [8] :

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 $v_f \in T$.
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 $W(v_f)$ contient donc le poids du chemin le plus léger de v_s à v_f. 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 $R[v_f]$ et remontant la chaîne, on construit un itinéraire optimal. **A l'initialisation,** $R[v]$ **contient** v_s **pour tout** v.

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.

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

2.5 Travail à faire

Exercice 6.—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	ab**c**det	s	1.0	2.0	1.0	∞	∞	∞	s	s	s	s	s	s	
1	**a**bdet	sc	1.0	2.0	1.0	∞	3.0	∞	s	s	s	s	c	s	
2	**b**det	sac	1.0	2.0	1.0	4.0	3.0	∞	s	s	s	a	c	s	
3	d**e**t	sabc	1.0	2.0	1.0	3.0	3.0	∞	s	s	s	b	c	s	
4	**d**t	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	

Exercice 7.—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 8.—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 [9].
1.b. Comment est déclaré le graphe du métro dans le module `metrodat.py` ?
2. Rendre le programme `metro.py` interactif.

7. Deux points d'un graphe sont dans une même composante connexe s'il existe un chemin dans le graphe les reliant
8. Dans l'implémentation écrite dans `graphes.py`, on utilise trois dictionnaires
9. Si votre module `graphes.py` est fonctionnel, ceci ne doit pas poser de problèmes.

3 Annexes

3.1 Listes

`L=[]` ou `L=list()`	fabrique une liste vide à remplir
`L=[1.0,'aa']`	fabrique une liste avec un préremplissage
`len(L)`	donne le nombre d'éléments de la liste, ils sont indicés de 0 à `len(L)-1`
`L[i]`	donne la valeur de l'élément de `L` d'indice i si $0 \leq i \leq$`len(L)-1`
`M=L[i:j]`	extrait une sous-liste M dont les éléments ceux de `L` sont indicés de i à $j-1$
`L[i:j]=M`	remplace la sous-liste des éléments de `L` indicés de i à $j-1$ par la liste `M`
`M=L`	Les deux étiquettes `M` et `L` sont sur la même liste en mémoire
`M=L[:]` ou `M = L.copy()`	créé une copie de `L` et lui affecte l'étiquette `M`
`L[-1]`	donne le dernier élément de la liste
`L.append(obj)`	ajoute l'objet `obj` en fin de liste
`L.extend(M)`	fusionne les listes `L` et `M` en plaçant `M` à la fin de `L`
`L.insert(index,obj)`	insère l'objet `obj` avant l'indice i, décale le reste de `L`
`L[i:i] = M`	insère la liste `M` à l'indice i de la liste `L`, décale le reste de `L`
`L.pop()`	efface le dernier élément de la liste `L` et le retourne
`L.pop(i)`	efface l'élément i de la liste `L` et le retourne
`L.reverse()`	inverse *en place* l'ordre des éléments de la liste
`L.sort()`	trie *en place* les éléments de la liste, pourvu que l'on puisse comparer les éléments
`M=sorted(L)`	`M` devient une copie triée de la liste `L`
`V=np.array(L)` ou `V=np.asarray(L)`	`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`.

`for obj in L :`
 	
Exemple :
`for k in L:`
`    print(k**2)`
	boucle sur les éléments de `L`, ne modifie pas `L`. Modifier `L` dans la boucle peut avoir des conséquences inattendues.
	Si on prend `L=[1,2,4]` affiche successivement 1, 4, 16.

`M = [expression(obj) for obj in L]`
 	
Exemple :
`M=[k+1 for k in L]`
	fabrique une liste en bouclant sur les éléments de `L` et en leur appliquant `expression()`, ne modifie pas `L`.
	Si on prend `L=[1,2,4]` fabrique la nouvelle liste `[2,3,5]` et l'affecte à `M`.

`M = [expr(obj) for obj in L if cond(obj)]`
 	
Exemple :
`M=[k**2 for k in L if k>1]`
	fabrique une liste en bouclant sur les éléments de `L`, ne gardant que ceux vérifiant `cond()` et en leur appliquant `expr()`, ne modifie pas `L`.
	Si on prend `L=[1,2,4]` fabrique la nouvelle liste `[4,16]` et l'affecte à `M`.

3.2 Dictionnaires

`D=dict()`	fabrique un dictionnaire vide à remplir
`D={'a': 65, 'b': 66}`	fabrique un dictionnaire avec un préremplissage du type clé : valeur
`len(D)`	donne le nombre d'éléments du dictionnaire
`D[k] = v`	affecte la valeur `v` à l'élément de `D` de clé `k`, le créé s'il n'existe pas encore
`del(D[k])`	élimine l'entrée de clé `k` dans le dictionnaire `D`
`D[k]` (dans une expression)	s'évalue à la valeur de l'élément de `D` de clé `k`
`E=D`	Les deux étiquettes `E` et `D` sont sur le même dictionnaire en mémoire
`E=D.copy()`	créé une copie de `D` et lui affecte l'étiquette `E`
`D.keys()`	retourne un itérable contenant les clés du dictionnaire `D`, pas d'ordre *a priori*
`sorted(list(D.keys()))`	retourne une liste *triée* des clés du dictionnaire `D`
`D.items()`	retourne un itérable contenant les couples (clés,valeurs) du dictionnaire `D`, pas d'ordre *a priori*