TD 8 : Lecture écriture dans un fichier - Graphes

Sources :

* <http://michel.stainer.pagesperso-orange.fr/PSIx/Informatique/Themes/2016-2017/Theme4-Graphes.pdf>

# Lecture et écriture dans un fichier

Pour lire ou écrire dans un fichier, on doit d’abord créer un « objet-fichier » par la commande open, dont le type est \_io.TextIOWrapper .

>>> f=open(os.listdir()[0])

>>> type(f)

<class '\_io.TextIOWrapper'>

Bien noter qu’il est **indispensable** de refermer lorsque la lecture ou la transformation en écriture est terminée, comme on le fait dans les exemples au-dessous.

La fonction open a deux arguments : le nom du fichier (ou le chemin menant au fichier, chemin absolu ou relatif), le mode d’ouverture du fichier (lecture 'r', écriture 'w', ajout en écriture 'a' – pour append, ou lecture et écriture 'r+') dont la valeur par défaut est 'r' (read).

* Ouverture/fermeture et lecture complète :

>>> f=open('Exemple.txt')

>>> f.read()

'riri\nfifi loulou\ndonald\n'

>>> f.close()

* Ouverture/fermeture et lecture ligne par ligne :

|  |  |
| --- | --- |
| >>> f=open('exemple.txt','r')  >>> f.readline()  'riri\n'  >>> f.readline()  'fifi loulou\n' | >>> f.readline() ##la fin de fichier se signale ##par une ligne vide  ''  >>> f.close() #note on a omis de lire la dernière ligne |

* Création/fermeture et écriture :

>>> f=open('nouveau.txt','w') #paramètre 'w’ nécessaire car égal à 'r' par défaut

>>> f.write('titi')

>>> f.write('rominet\n') #insérer \n si on souhaite un changement de ligne

>>> f.write('Fin')

>>> f.close()

Note : pour stocker des objets créés dans un programme Python, il existe le module pickle qui permet d’enregistrer les objets, ainsi que leur attributs, et non pas seulement leurs valeurs, comme on peut le faire avec les méthodes ci-dessus [<https://docs.python.org/3.3/library/pickle.html#module-pickle>]

**Q6.1.** Créer à l’aide d’un tableur, un tableau de valeurs quelconques, flottantes ou entières, en utilisant éventuellement des formules. Enregistrer ce tableau dans un fichier au format .csv (« comma separated values »). Ouvrir à l’aide de Python le fichier créé et expliciter ce qui code la structure de tableau.

Inversement, écrire un script Python qui génère une liste de 101 abscisses (entières, de 0 à 100) et une liste de 101 ordonnées, choisies aléatoirement dans l’intervalle et qui écrit ces valeurs dans un fichier, d’une façon compatible avec le format .csv, en indiquant des titres de colonnes (« abscisses » et « ordonnées »).

Vérifier la lisibilité du fichier créé en demandant son ouverture à partir d’un tableur.

Ouvrir à l’aide du bloc-notes le fichier TP16\_titrage.txt. Ce fichier correspond aux données utilisées dans l’exercice 1 du DS3 et contient des informations sur le titrage d’un acide. La première ligne de ce fichier contient la description du titrage, la seconde ligne indique le nombre de mesures effectuées, les lignes suivantes indiquent, pour chaque prise de mesure, le volume de soude versé en mL et la mesure de pH correspondante.

Écrire un script Python qui lit les informations dans ce fichier et qui écrit, dans un fichier .csv ayant pour nom le descriptif de l’expérience, ces informations sous la forme deux intitulées « Volume de soude versé (mL) » et « pH mesuré », la virgule étant codée par une virgule et non un point.

Écrire un script Python prenant en argument le nom d’un fichier texte et renvoyant le nombre d’occurrences de chaque lettre de l’alphabet dans ce texte. On pourra utiliser le fichier Notre-Dame de Paris.txt et comparer les fréquences obtenues aux fréquences données sur la page <https://fr.wikipedia.org/wiki/Fr%C3%A9quence_d'apparition_des_lettres_en_fran%C3%A7ais> .

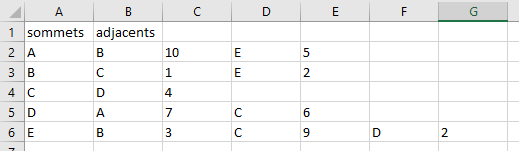
# Graphes

## Représentations d’un graphe

### Par liste d’adjacence

On prend comme modèle les graphes déjà étudiés en cours. Pour le premier (qui est un graphe orienté pondéré), on donne un fichier .csv (comma separated values) qui donne pour chaque sommet, la liste des sommets qui lui sont adjacents, avec le poids associé à l’arc qui relie ces sommets.

En ouvrant ce fichier (graphe1\_listesadjacence.csv) à l’aide d’un tableur, on a l’affichage suivant :



Ci-dessous, on rappelle la représentation « graphique » du graphe, ainsi que le contenu du fichier .csv :

|  |  |
| --- | --- |
| Graphe 1 | graphe1\_listesadjacence.csv |
|  | En ouvrant ce fichier à l’aide de l’application « bloc notes », on découvre la structure du fichier et son contenu :  sommets,adjacents,,,,,  A,B,10,E,5,,  B,C,1,E,2,,  C,D,4,,,,  D,A,7,C,6,,  E,B,3,C,9,D,2 |

Écrire une fonction lectureGraphe, utilisant les fonctions de lecture de fichiers, prenant en argument le chemin vers le fichier .csv représentant le graphe et renvoyant sous la forme d’une liste le nom de tous les sommets et, sous la forme d’une liste de listes, les listes d’adjacence des sommets du graphe, en respectant un ordre cohérent avec la première liste.

On rappelle que la suite d’instructions élémentaires permettant de lire un fichier :

>>> f=open('graphe1\_listesadjacence.csv')

>>> u=f.readlines()

>>> f.close()

On obtient par cette méthode, une variable u de type list, dont les éléments correspondent chacun à une ligne du fichier lu. Chaque élément a la forme d’une chaîne de caractères.

Il s’agit ensuite, à l’aide des méthodes strip et split, propres aux chaînes de caractères, d’extraire de chaque ligne les informations pertinentes.

On rappelle ce que font ces deux fonctions :

>>> ch="il fait beau\n"

>>> ch.strip()

'il fait beau'

>>> ch.split()

['il', 'fait', 'beau']

Appliquée au graphe considéré, la fonction lectureGraphe doit renvoyer :

>>> lectureGrapheCSV('graphe1\_listesadjacence.csv' )

>>> ['A','B','C','D','E'],[[('B',10),('E',5)],[('C',1),('E',2)],[('D',4)],

[('A',7),('C',6)],[('B',3),('C',9),('D',2)]]

### Matrice d’adjacence

**Q4.1** Écrire une fonction LadjVersMatAdj, qui prend en argument la liste des listes d’adjacences d’un graphe et renvoie la matrice d’adjacence de ce graphe.

**[Bonus]** Écrire une fonction réciproque MatAdjVersLAdj.

## Parcours en largeur d’un graphe

Le parcours en largeur d’un graphe permet d’explorer un graphe à partir d’un sommet, L’idée est ici de partir d’un sommet donné s et de visiter les sommets du graphe en commençant par  
les sommets adjacents à s, puis en s’éloignant progressivement de s (sommets accessibles via un chemin  
de longueur(\*) , puis , *etc.*).

Par essence même, ce parcours n’atteindra que les sommets accessibles à  
partir de s.

(\*) *Note :* dans cet algorithme, on ne tient pas compte des poids éventuellement attachés aux arêtes : la distance entre deux sommets reliés par une arête est de 1 de façon systématique (la valeur entière 1 étant à interpréter comme le fait qu’une arête sépare les deux sommets).

Une distance égale à un entier entre deux sommets signifie qu’il faut parcourir (au minimum) deux arêtes pour aller de l’un de ces sommets à l’autre.

### Principe de l’algorithme

On décide du sommet s à partir duquel va commencer l’exploration du graphe.

On définit un tableau c (une liste Python) dans lequel on maintiendra à jour une couleur affectée à chaque sommet, selon que ce sommet n’a pas encore été découvert (couleur « blanc »), a été découvert et est en cours de traitement (couleur « gris »), a été découvert et complètement traité (couleur « noire »).

Initialement, aucun sommet n’a été découvert, et donc la couleur de tous les sommets est à « blanc », seul le sommet, s0, à partir duquel on décide, arbitrairement, de commencer l’exploration est « gris ».

On associe également à chaque sommet, deux valeurs, inscrites respectivement dans deux tableaux d et p (listes Python), dont la première, d[u], associera à chaque sommet u, sa distance au sommet de départ, s0, et la deuxième, p[u], associera à chaque sommet u, un père de u, à savoir un sommet précédant u, le long d’un plus court chemin de s0 à u.

L’idée du parcours en largeur d’un graphe est que les sommets sont découverts dans l’ordre de leur distance au sommet de départ, s0.

Dès qu’un sommet v est découvert à partir d’un sommet u pour lequel on connaît d[u], on connaît d[v] qui vaut d[u] + 1 et devient « gris ». Pour ne pas oublier de revenir à v pour explorer sa liste d’adjacence, on le stocke dans une file d’attente F qui contiendra en permanence les sommets en cours de traitement (« gris »).

On doit utiliser une file d’attente (structure FIFO (*First In First Out*)) pour être sûr de repartir d’abord des sommets les moins éloignés, qui auront été les premiers ajoutés à la file d’attente.

On peut résumer comme suit, l’algorithme à mettre en œuvre :

* Tous les c[u] sont initialisés à « blanc », les d[u] à «  » et les p[u] à «  », F est initialisée à la file vide ;
* Au début du traitement, c[s] devient « gris », s est ajouté à F, d[s] prend la valeur , tandis que p[s] reste à .
* On traite les sommets gris présents dans F en explorant leurs sommets adjacents.
* Exemple : <http://math.univ-lyon1.fr/irem/IMG/pdf/parcours.pdf>

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Matrice d’adjacence   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | a | b | c | d | e | f | g | h | | a | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | | b | 1 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | | c | 1 | -1 | -1 | 1 | -1 | -1 | -1 | -1 | | d | -1 | 1 | 1 | -1 | 1 | -1 | -1 | -1 | | e | -1 | 1 | -1 | 1 | -1 | 1 | 1 | -1 | | f | -1 | -1 | -1 | -1 | 1 | -1 | 1 | -1 | | g | -1 | -1 | -1 | -1 | 1 | 1 | -1 | 1 | | h | -1 | -1 | -1 | -1 | -1 | -1 | 1 | -1 | |

### Implémentation

L’algorithme présenté ci-dessus s’écrit en pseudo-code, de la façon suivante :

|  |
| --- |
| initialiser les couleurs à « blanc », les distances à «  », les pères à «  », et F à la file vide  c[s] ← « gris » ; d[s] ← 0 ; ajouter s à F  tant que F n’est pas vide faire  u ← suivant(F)  pour chaque v adjacent à u faire  si c[v] = « blanc » alors faire  c[v] ← « gris » ; ajouter v à F  d[v] ← d[u] + 1 ; p[v] ← u  fin faire  fin faire  c[u] ← « noir »  fin faire |

**Q5.1** Tester cet algorithme sur le graphe de l’exercice 3.

Implémenter cet algorithme sous la forme d’une fonction PL(L,s0), prenant en argument une liste de listes d’adjacence représentant un graphe et le numéro du sommet à partir duquel on explore le graphe. On rappelle que la valeur «  » peut être modélisée par la valeur flottante float('inf').

Les sommets du graphe seront numérotés de à et la liste d’adjacence modifiée en conséquence sur les exemples.

def PL(L,s0):

n=len(L)#nombre de sommets du graphe

c=n\*["b"]#initialisation des couleurs à « blanc »

d=n\*[float('inf')]#initialisation des distances à s0 à «  »

p=n\*[-1]#initialisation des couleurs à « blanc »

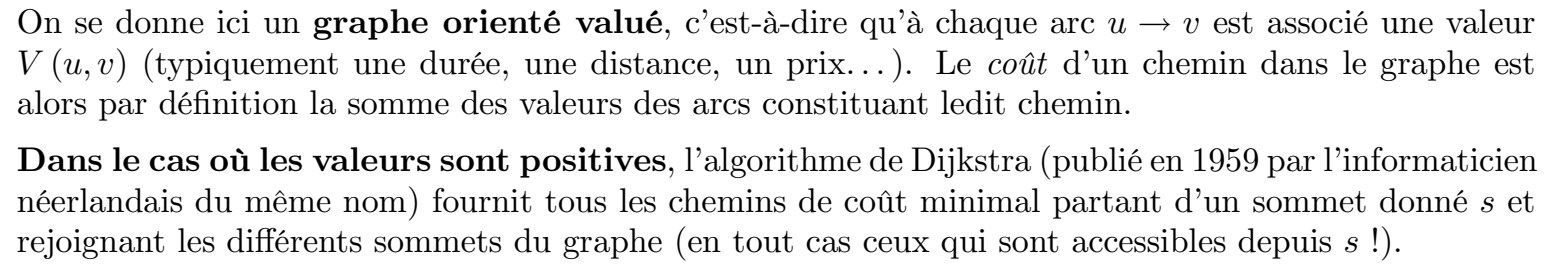
F= file\_vide()#on aura importé les primitives sur les files (exercice 2)

c[s0],d[s0]="g",0 ; ajouter(x,f)#initiation de l’exploration en s0

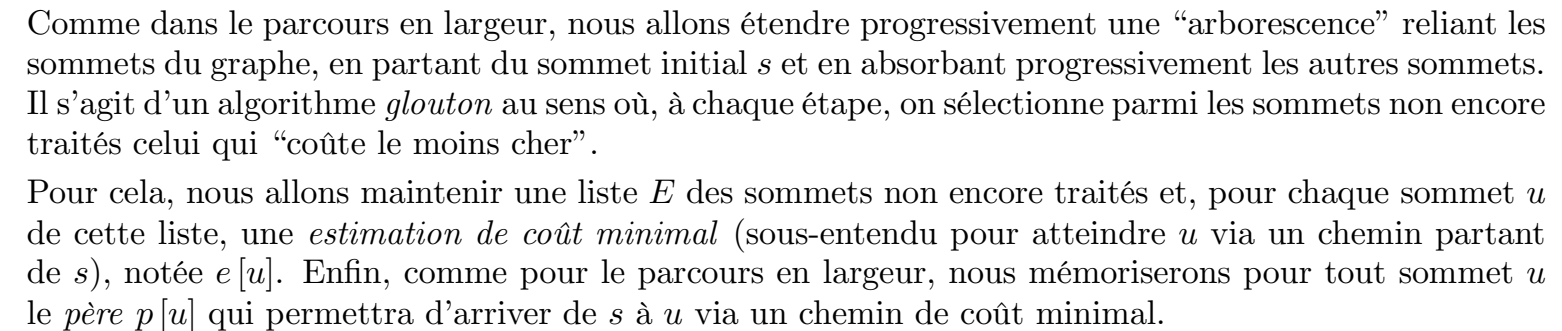
while not est\_file\_vide(f):

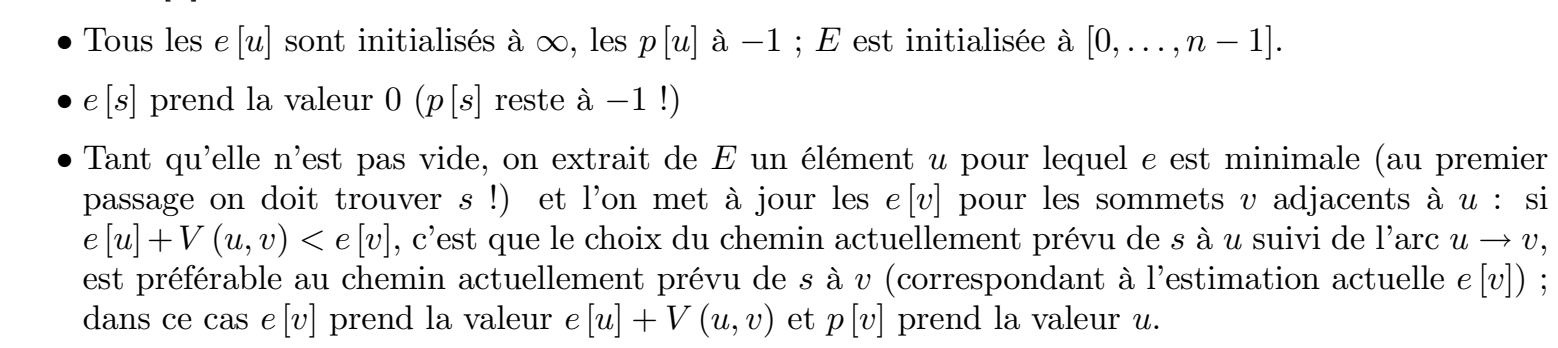
...

### Application : algorithme de Dijkstra

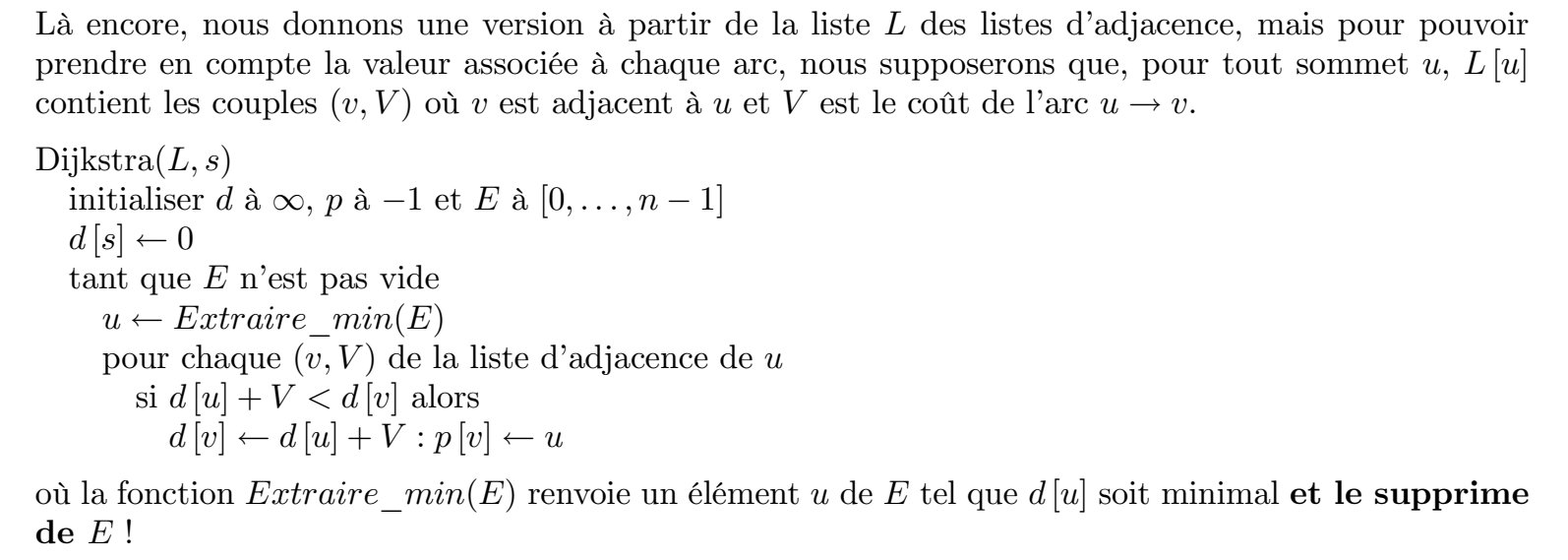


#### Principe de l’algorithme





#### Implémentation en pseudo-code



**Q7.1** Tester l’algorithme sur le graphe de l’exercice 3.

Écrire une fonction Dijkstra(L,s0), qui prend en argument la liste des listes d’adjacences d’un graphe et un sommet de départ et renvoie les chemins de coût minimal vers les sommets accessibles depuis s0.

Tester la fonction Dijkstra sur l’exemple suivant, où le graphe est exploré depuis le sommet « E » :

|  |  |
| --- | --- |
|  |  |