TD 9 : Piles - Graphes

Sources :

* <http://michel.stainer.pagesperso-orange.fr/PSIx/Informatique/Exos/2017-2018/17tdi3.pdf>
* <http://michel.stainer.pagesperso-orange.fr/PSIx/Informatique/Themes/2016-2017/Theme4-Graphes.pdf>

# Piles

Dans de nombreux cas, la première étape de l’analyse syntaxique d’une expression est la vérification du parenthésage. Une expression bien parenthésée (représentée par une chaîne de caractères) est définie (récursivement) comme étant :

* soit une chaîne ne contenant aucune parenthèse (éventuellement vide !) ;
* soit une expression bien parenthésée entre parenthèses
* soit la concaténation de deux expressions bien parenthésées.

Pour contrôler le parenthésage d’une expression, il s’agit de vérifier que chaque parenthèse fermante est associée à une parenthèse ouvrante située à sa gauche et que toutes les parenthèses ouvrantes sont refermées, d’où l’idée d’utiliser une pile.

Le principe de l’algorithme à mettre en œuvre est le suivant :

on parcourt de gauche à droite la chaîne représentant l’expression et on empile les positions des parenthèses ouvrantes. Lorsqu’on rencontre une parenthèse fermante, on dépile, le sommet de la pile étant alors la position de la parenthèse ouvrante associée (en option, on peut afficher le couple des positions des deux parenthèses associées).

**Q1.1.** Rappeler l’implémentation en Python des opérations élémentaires sur les files, que l’on notera creer\_pile, empiler, depiler, est\_vide. Une pile p sera modélisée par une liste Python, dont le dernier élément correspond au sommet de la pile p.

Implémenter le contrôle de parenthésage présenté en introduction, en Python, sous la forme d’une fonction verifp, prenant en argument la liste des caractères composant l’expression dont on veut vérifier le parenthésage, implémentant l’algorithme dont le principe est décrit ci-dessus, et renvoyant un booléen indiquant si l’expression est bien ou mal parenthésée.

On notera ch la chaîne en entrée, et p la pile contenant les positions des parenthèses ouvrantes au sein de la chaîne ch.

La fonction verifp affichera les couples des positions des paires de parenthèses associées jusqu’à détection d’un éventuel mauvais parenthésage, et renverra un booléen True ou False, selon que l’expression est bien parenthésée ou non.

Le parcours de la chaîne ch s’interrompra dès qu’un mauvais parenthésage sera détecté.

**Files**

Le type abstrait de données “files d’attente” est une structure linéaire de stockage munie des primitives  
suivantes :

* est\_file\_vide(f) : renvoie True si la file f est vide, Faux sinon ;
* file\_vide() : renvoie la file vide ;
* ajouter(x,f) : ajoute le client x à la fin de la file ;
* suivant(f) : renvoie le client en tête de la file non vide f et le supprime de f.

**Q2.1.** Proposer une implémentation en Python où la file est stockée dans une simple liste. Évaluer la  
complexité des fonctions ajouter et suivant.

Pour obtenir une complexité en temps constant (ou temps constant amorti) des deux fonctions  
ajouter et suivant, on se propose de stocker une file à l’aide de deux piles :

* la première contient le début de la file, le premier client à servir se trouvant au sommet ;
* la deuxième contient la fin de la file, le dernier client entré se trouvant au sommet.

Mettre en œuvre ce principe en Python ; concrètement, la file f pourra être définie comme une liste  
de deux piles [p0,p1] ; expliquer son intérêt du point de vue de la complexité. Outre les primitives  
ci-dessus, on programmera l’affichage d’une file.

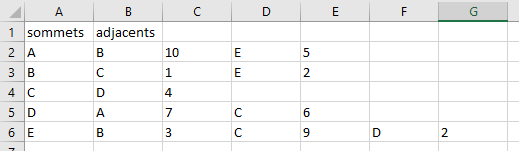
# Graphes

## 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 »

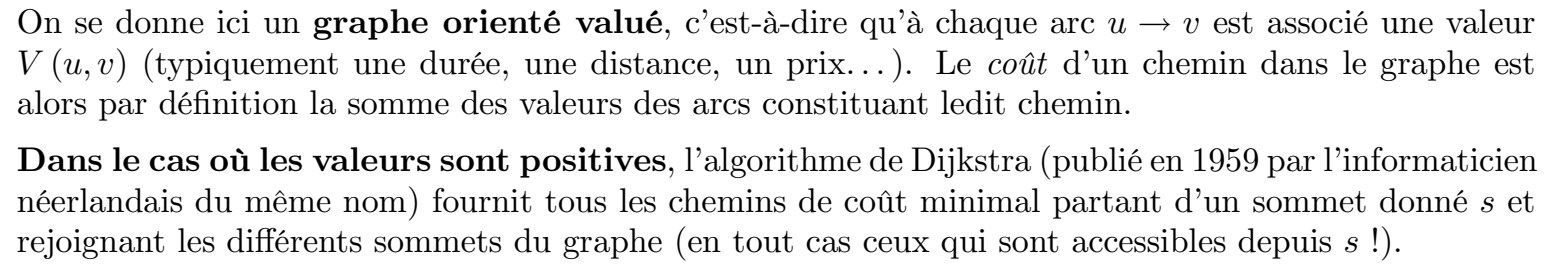
F= file\_vide()#on aura importé les primitives sur les files (exercice 2)

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

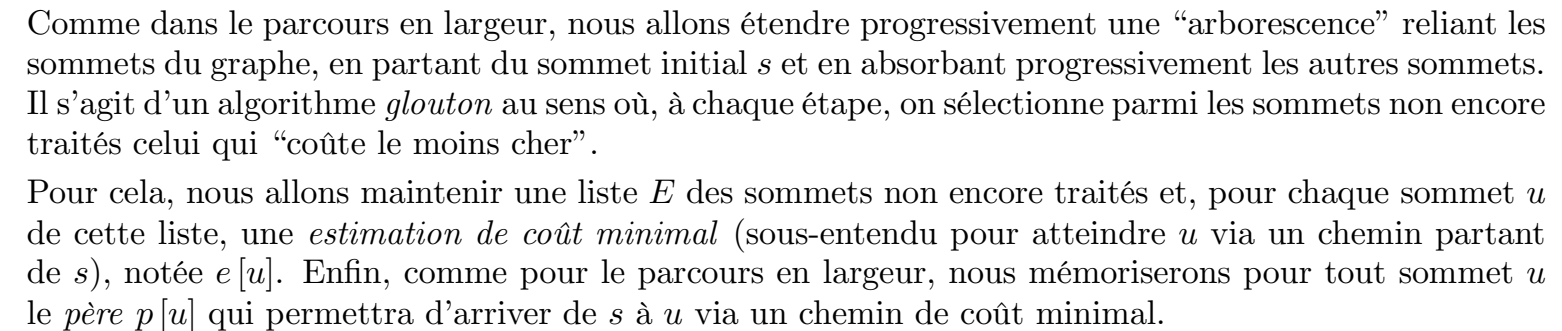
while not est\_file\_vide(f):

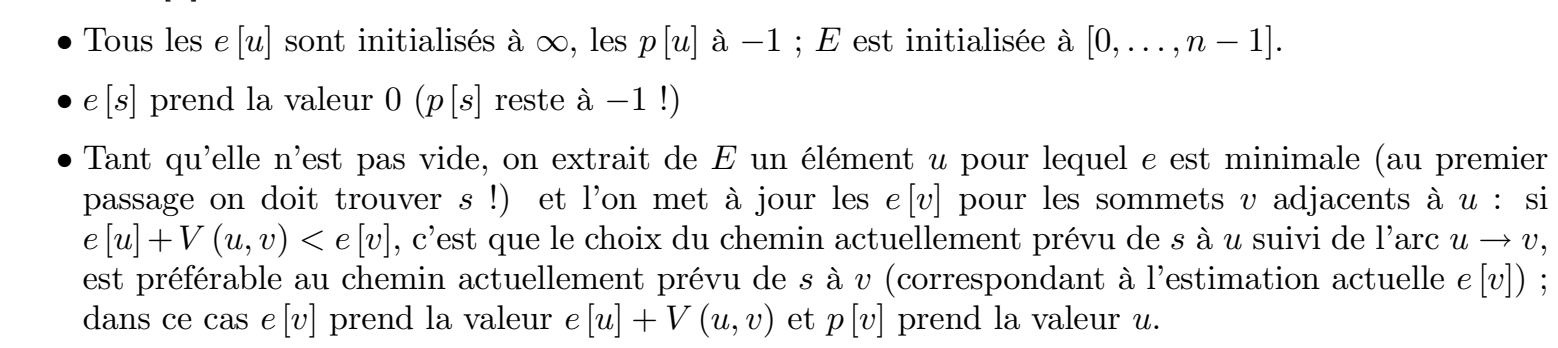
...

### Application : algorithme de Dijkstra

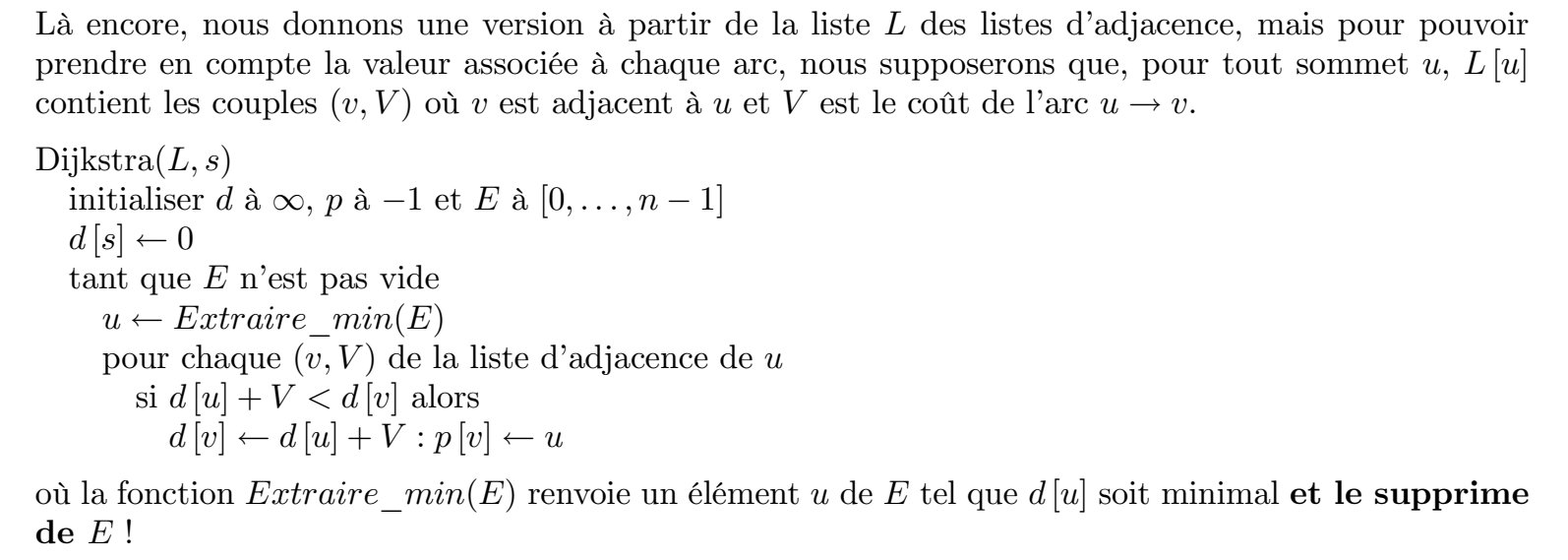


#### Principe de l’algorithme





#### 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 » :

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