Tutoriels

Initiation à la Programmation

ENSAE

Xavier Dupré

Table des matières

Chapitre 1

Commencer à programmer

1.1 Graphe

Ce tutoriel mélange des questions dont la réponse attendue est un extrait de programme et d'autres dont la réponse doit être rédigée. Le rendu doit contenir deux choses :

1. un (ou plusieurs) programme informatique ne faisant aucune référence à des fichiers ou ressources liées à l'ENSAE (pas de fichier W :..... par exemple),
2. un rapport imprimé (non manuscrit) regroupant les réponses à chaque question incluant les extraits de programmes.

L'élève devra envoyer par email à son chargé de TD dans un seul fichier ZIP l'ensemble des programmes et le rapport. Il devra également imprimer son rapport et le rendre à son chargé de TD.

Ce tutoriel s'intéresse à des algorithmes non abordés lors des travaux pratiques. On se propose de manipuler des graphes. Les premières questions ont pour but de personnaliser l'énoncé pour chaque élève afin que chacun travaille sur des données différentes. Par la suite de l'énoncé, il faudra répondre avec le jeu de données que les premières questions vous attribueront.

Personnalisation des données .

1) Dans l'exemple qui suit, la première ligne permet d'ajouter des accents dans le programme. Autrement, le moindre accent provoque une erreur. Tous les langages de programmation manipulent les caractères comme s'ils étaient des nombres. Ainsi, si on crée la chaîne de caractères suivante :

```
# coding : latin-1
nom_prenom = "Xavier Dupré"
```

On peut récupérer la valeur numérique du caractère à la position `i` on écrivant :

```
i      = 3                      # i=3 mais on fait référence au 4ème caractère
valeur = ord ( nom_prenom [i] )
```

En remplaçant le nom cité en exemple par votre prénom suivi de votre nom, quel résultat cela vous donne-t-il ? (0,5 point)

2) La fonction `len` retourne le nombre d'éléments d'un tableau ou la longueur d'une chaîne de caractères par exemple. En utilisant une boucle `for`[1], écrire un mini-programme qui calcule la somme des valeurs des caractères de la chaîne de caractères utilisée à la question précédente. (1 point)

3) Transformer le mini-programme précédent pour en faire une fonction qui retourne un entier : (1 point)

1. On pourra aussi utiliser les fonctions `len`, `range` ou `xrange`.

```
def somme_caractere (nom_prenom) :
    ...
    return ...
```

On suppose que la variable `s` contient le résultat de la fonction précédente appliquée à votre nom prénom, les lignes suivantes vous donneront accès aux données à insérer dans votre programme pour la suite du tutoriel :

```
s   = somme_caractere ("Xavier Dupre") % 200
url = "http://www.xavierdupre.fr/enseignement/tutoriels_data/tutoriel_%d.py" % s
print url
```

On commence maintenant un nouveau programme en laissant celui-ci de côté. Le fichier téléchargé sera copié à côté de ce nouveau programme. En supposant que 99 soit le nombre sur lequel vous soyez tombé, le nouveau programme commencera par la ligne :

```
from tutoriel_99.py import *
```

Cette ligne permettra d'inclure les données utilisées.

Les données contiennent deux dictionnaires :

`noeuds`[`i`]	contient la valeur associée au nœud `i`
`arcs`	si le dictionnaire contient le tuple (`i`, `j`), alors les nœuds `i` et `j` sont reliés.

Quelque soit les données qui correspondent à votre nom, ce graphe sera semblable à celui de la figure **??**. L'énoncé propose dans la troisième partie une fonction qui permet de dessiner ce graphe et ainsi de vérifier les manipulations opérées sur ce genre de structure.

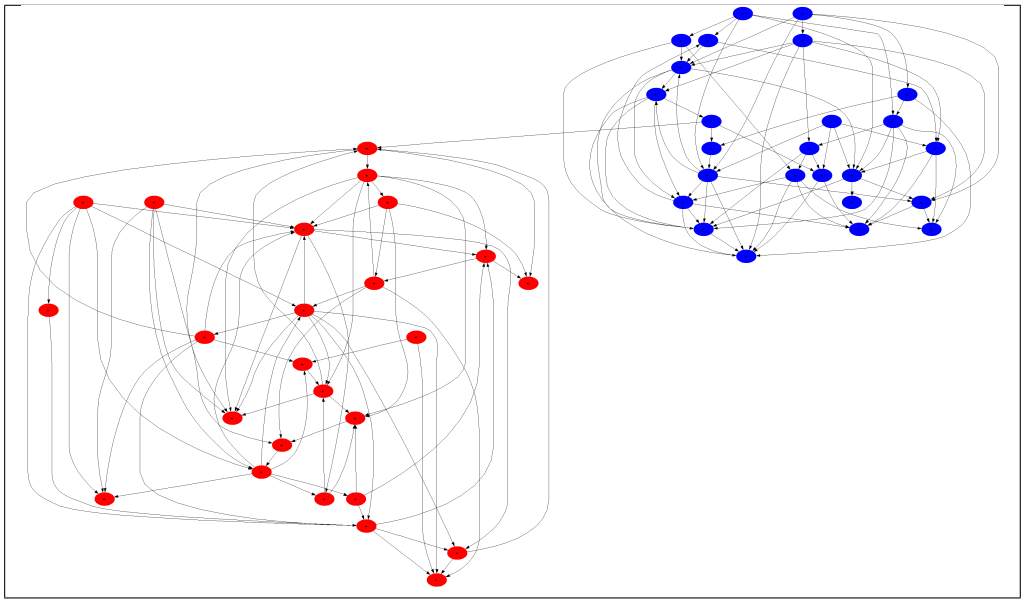

Figure 1.1 : *Exemple de graphe défini par les variables* `noeuds` *et* `arcs`. *Le graphe est connexe : il existe toujours un chemin pour passer d'un nœud à un autre. En coupant un arc, le graphe devient composé de deux composantes connexes.*

Partie 1

Pour la première partie, on utilise une représentation différente d'un graphe en utilisant une classe :

```
class MonGraphe :
    def __init__ (self, num, valeur) :
        self.valeur = valeur
        self.num    = num
        self.arcs   = [] # liste dŠéléments MonGraphe
```

Du graphe (celui de la figure **??** par exemple), on ne connaît alors qu'un seul nœud appelé *racine*. Pour trouver les autres nœuds, on part de la racine et on explore de proche en proche les voisins. Cette représentation est inévitable lorsqu'il s'agit d'un grand graphe comme Internet. Chaque page ou nœud ne donne accès qu'à ses voisins et il est impossible de connaître à un instant l'ensemble d'un graphe en constante évolution. L'inconvénient est qu'il n'est plus aussi évident de compter le nombre de nœuds que contient le graphe.

4) Ecrire une fonction qui transforme les dictionnaires `noeuds` et `arcs` en une liste d'éléments `MonGraphe`. Il peut y avoir des arcs redondants. (1,5 point)

```
def ConstruireMonGraphe (noeuds, arcs) :
    ...
    return mongraphe
```

On appellera le premier nœud retourné par la fonction le nœud racine. On pensera tout particulièrement au fait que le graphe est symétrique.

5) On ajoute la méthode :

```
class MonGraphe :
    def NombreTotalNoeud (self) :
        r = 1
        for a in self.arcs :
            r += a.NombreTotalNoeud()
        return r
```

En principe, cette fonction devrait provoquer une erreur. Expliquez pourquoi ? (1 point) On pourra s'intéresser au cas où il existe des cycles (il existe un chemin d'un nœud du graphe vers lui-même)

6) Il existe des graphes pour lesquelles cette fonction retourne le bon résultat (à savoir le nombre de nœuds), quelle particularité ont ces graphes ? (2 points)

7) Adapter (ou réécrire) la fonction précédente pour qu'elle retourne le bon résultat quelque soit le graphe. (3 points)

Partie 2

Pour cette seconde partie, on revient à la représentation du graphe avec les deux dictionnaires `noeuds` et `arcs`.

8) On souhaite obtenir deux composantes connexes en enlevant le minimum d'arcs. Pour cela, on pense à un algorithme appelé *Minimum Spanning Tree* (avec `Wikipedia` par exemple). Est-ce que cet algorithme vous satisfait ? Expliquer. (1 point)

9) On définit le degré d'un nœud comme étant le nombre d'arcs reliés à ce nœud. Ecrire une fonction qui calcule le degré pour un nœud donné. (1 point)

10) On construit le Laplacien du graphe. C'est une matrice carrée $n \times n$ où n est le nombre de nœuds du graphe. Soit $M = (m_{ij})$ ce laplacien :

$$m_{ij} = \begin{cases} d_i & \text{si } i = j \\ -1 & \text{s'il existe un arc reliant } i \text{ et } j \\ 0 & \text{sinon} \end{cases} \quad (1.1)$$

Construisez cette matrice. On rappelle que le graphe est toujours non-directionnel. Montrez que la somme des valeurs de chaque colonne est 0. (2 points)

11) Montrez que l'une des valeurs propres [2] de cette matrice est 0. Quel est le vecteur propre associé ? (1 point)

12) Calculez les valeurs propres à l'aide du module `numpy`. Vous vous assurerez que le résultat théorique précédent est vérifié numériquement. (1 point)

13) Toutes les valeurs propres sont positives ou nulles, quel est le vecteur propre associé à la plus petite des valeurs propres non nulles ? Ces deux classes sont également décrites par les expressions (**??**) et (**??**). (1 point)

14) On classe les nœuds en deux classes selon qu'ils sont associés à une valeur positive ou négative d'après ce vecteur propre. On peut maintenant déterminer quel nœud appartient à la première composante, quel nœud appartient à la seconde. Ecrire une fonction qui calcule le nombre d'arcs qui relient un nœud de la première composante à un nœud de la seconde. Quel est le résultat ? (3 points)

Dessin du graphe .

Pour vérifier, on souhaite dessiner le graphe dans une image. Pour cela on utilise le programme [3] suivant qui s'appuie sur le logiciel *Graphviz* [4].

```
# coding:latin-1
def drawGraph (edges, image) :
    """
    dessine un graph en utilisant Graphviz (http://www.graphviz.org/
    edges = [ (1,2), (3,4), (1,3), ... ]  , liste d'arcs
    image = bom d'image (format png)
    """
    import os, urllib,struct
    files = [ "_graphviz_draw.exe"]
    if not os.path.exists (files[-1]) :
        # on télécharge les fichiers nécessaires d'abord
        for f in files :
            print "téléchargement de ", f
            url = "http://www.xavierdupre.fr/enseignement/tutoriel_python/graphviz/" + f
            u = urllib.urlopen (url, "rb")
            all = u.read ()
            if "404 Not Found" in all :
                raise Exception ("fichier introuvable")
            u.close ()
            u = open (f, "wb")
            u.write ( struct.pack ("c"*len(all), *all))
            u.close()
    if not os.path.exists (files[-1]) :
        raise Exception ("mauvais téléchargement")

    li = [ "digraph{" ]
    for i,j in edges :
        li.append ( "%d -> %d ;" % (i,j) )
    li.append (";")
```

2. En anglais, valeurs propres et vecteurs propres se traduisent par *eigen values* et *eigen vectors*.
3. voir http://www.xavierdupre.fr/enseignement/tutoriel_python/graphviz/use_graphivz.py
4. http://www.graphviz.org/

```
    f = open ("graph.gv", "w")
    f.write ( "\n".join(li) )
    f.close ()

    cmd = "%s . graph.gv %s png neato" % (files[-1], image)
    os.system (cmd)

drawGraph ([(1,2),(3,4), (1,3)], "image.png")
```

Le système informatique de l'école ne permet pas l'installation de logiciels, le programme utilisé n'est pas la version officiel de *Graphivz* mais une version recompilée à partir des sources. Les graphes seront moins jolis que ceux que vous pourriez obtenir en installant la dernière version du logiciel.

Partie facultative : intermède théorique

Les résultats suivants sont extraits du livre *Statistical Analysis of Network Data* de Eric D. Kolaczyk. G désigne un graphe connexe, avec une seule composante connexe. On définit S un ensemble de sommets du graphe, $\bar{S}$ est son complémentaire. $E(S,\bar{S})$ désigne l'ensemble des arcs qui relient un sommet de S à un sommet de son complémentaire. $|S|$ désigne le nombre de sommets inclus dans S. Enfin :

$$\Phi(G) = \min_{|S| \leqslant \frac{|G|}{2}} \frac{\left|E(S,\bar{S})\right|}{|S|} \tag{1.2}$$

On cherche toujours à couper un graphe connexe en deux composantes connexes en supprimant le moins d'arcs possibles. Cela dit, on souhaite aussi éviter couper un seul sommet du reste du graphe tout simplement parce qu'il est relié par un seul arc. La fonction $\Phi(G)$ exprime ce compromis.

Résoudre ce problème de minimisation est un problème NP-complet [5]. λ_2 désigne la plus petite des valeurs propres non nulles et d_{max} le plus grand nombre d'arcs reliés au même sommet (degré maximal du graphe). On peut montrer dans ce cas :

$$\frac{\lambda_2}{2} \leqslant \Phi(G) \leqslant \sqrt{\lambda_2(2d_{max} - \lambda_2)} \tag{1.3}$$

De plus, si $V = (x_1, ...x_n)$ le vecteur propre associé à cette valeur propre, on définit S et $\bar{S}$ comme suit :

$$S = \{v \in V | x_v > 0\} \tag{1.4}$$
$$\bar{S} = \{v \in V | x_v < 0\} \tag{1.5}$$

On s'assure que $|S| \leqslant \left|\bar{S}\right|$. Alors, on peut montrer que :

$$\Phi(G) \leqslant \frac{\left|E(S,\bar{S})\right|}{|S|} \leqslant \frac{\Phi^2(G)}{d_{max}} \leqslant \lambda_2 \tag{1.6}$$

Le vecteur V est donc une réponse approchée au problème de minimisation (**??**).

Partie facultative : si vous êtes curieux

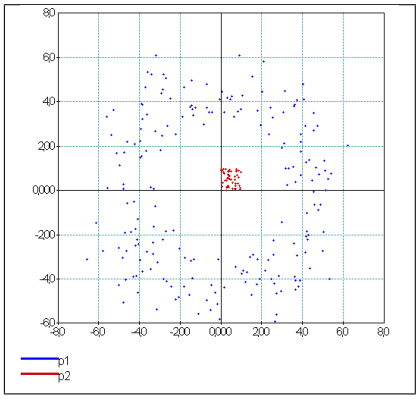

Figure 1.2 : *Problème de clusterisation. On souhaite automatiquement classer les points du centre et ceux de la périphérie en deux classes différentes.*

On applique cette méthode à un problème de *clusterisation*. Prenons l'exemple de la figure **??**.

On note $d(X_1, X_2)$ la distance euclidienne entre deux points X_1 et X_2. On construit le Laplacien suivant à partir d'un ensemble de points du plan $(X_i)_i$.

$$m_{ij} = \begin{cases} -e^{-d(X_i,X_j)^2} & \text{si } i \neq j \\ \sum_{i \neq j} e^{-d(X_i,X_j)^2} & \text{si } i = j \end{cases} \tag{1.7}$$

15) Créer une fonction qui génère un tel ensemble de points en deux dimensions de façon aléatoires ? (pas de point)

16) Implémentez la méthode suggérée et dessiner le résultat. (A l'aide du module `matplotlib` par exemple.) (pas de point)

5. Il n'existe pas d'algorithme capable de résoudre ce problème en un temps polynomiale.

Index

Table des matières