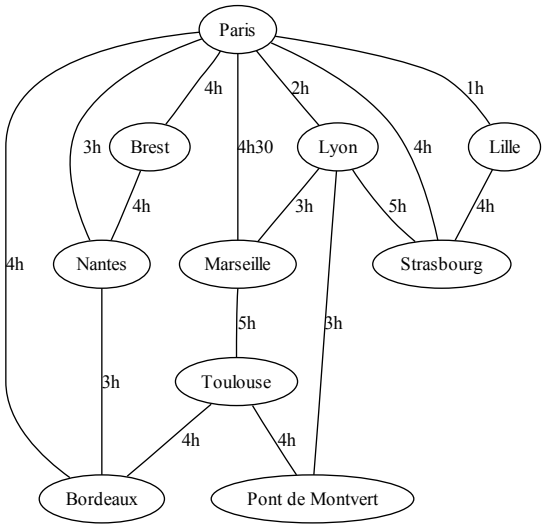
1 TD 12 : Meilleur chemin dans un graphe

(correction page ??)

L'énoncé de cette séance est long, les deux dernières parties peuvent être traitées indépendamment l'une de l'autre.

Abordé lors de cette séance	
programmation	meilleur chemin dans un graphe
algorithme	graphe

L'algorithme du plus court chemin est celui qu'on évoque le plus souvent lorsqu'on parle de programmation dynamique [1]. Plusieurs variantes existent, chacune portant le nom de son inventeur Bellman, Ford [2], Djikstra [3]... Le graphe suivant représente quelques lignes ferroviaires et le temps approximatif que le train met pour relier deux villes.



Lorsque le graphe est aussi simple que celui-là, il est facile de chercher le chemin le plus court pour aller de Nantes à Pont de Montvert. Lorsqu'il ressemble à celui de la figure 1, il devient plus difficile de déterminer le chemin le plus court sans utiliser un algorithme.

Première demi-heure : données .

1) On utilise le programme suivant [4] pour récupérer les données qui représente le graphe dont une partie est représentée par la figure 1.

1. http://en.wikipedia.org/wiki/Dynamic_programming
2. http://fr.wikipedia.org/wiki/Algorithme_de_Bellman-Ford
3. http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra
4. http://www.xavierdupre.fr/enseignement/td_python/python_td_minute/data/court_chemin/load_distance_matrix.py

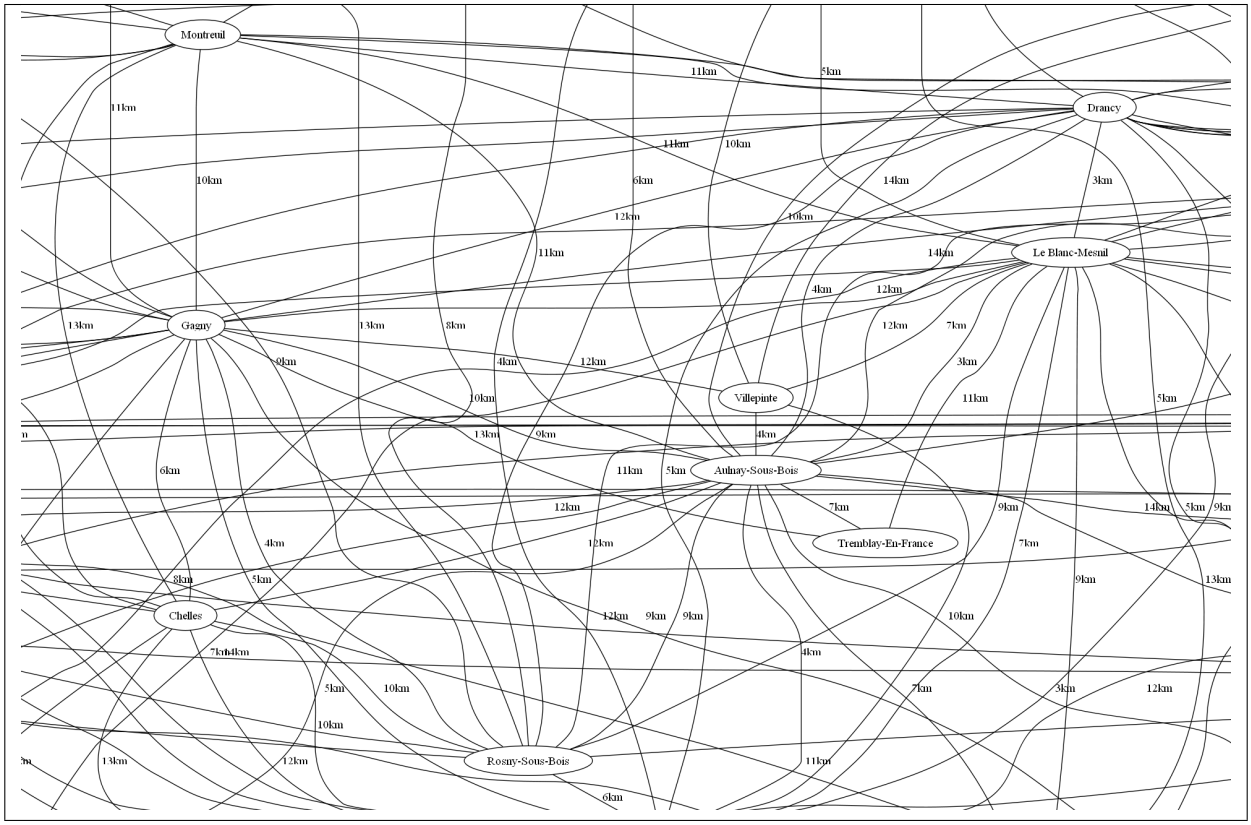


Figure 1 : *Extrait d'un graphe dont chaque arc correspond à une distance entre deux villes. Ces distances ne sont pas les distances à vol d'oiseau mais celles des itinéraires calculés par GoogleMaps.*

```
#coding:latin-1
import urllib, os, os.path
# coding:latin-1
def drawGraph (edges, script, 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")

    if script == None :
        li = [ "digraph{" ]
        for i,j in edges :
            li.append ( "%d -> %d ;" % (i,j) )
        li.append (";")
        f = open ("graph.gv", "w")
        f.write ( "\n".join(li) )
        f.close ()
    else :
        f = open ("graph.gv", "w")
        f.write ( script )
        f.close ()

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

def charge_donnees (file = "matrix_distance_7398.txt") :
    if os.path.exists (file) :
        # si le fichier existe (il a déjà été téléchargé une fois)
        f = open (file, "r")
        text = f.read ()
        f.close ()
    else :
        # si le fichier nŠexiste pas
        link = "http://www.xavierdupre.fr/enseignement/td_python/" + \
        "python_td_minute/data/court_chemin/" + file
        url = urllib.urlopen (link)
        text = url.read ()
        # on enregistre les données pour éviter de les télécharger une seconde fois
        f = open (file, "w")
        f.write (text)
        f.close ()
    lines = text.split ("\n")
    lines = [ l.split("\t") for l in lines if len(l) > 1 ]
    return lines

def conversion_en_dictionnaire (lines) :
    res = { }
    for a,b,c in lines :
        c = int (c)
        res [a,b] = c
        res [b,a] = c
    return res

def graphviz_script (mat_dict) :
    script = ["graph {"]
    vertex = { }
    villes = [ _[0] for _ in mat_dict.keys() ]
    for v in villes :
        if v not in vertex :
            vertex [v] = len(vertex)
    for k,v in vertex.iteritems () :
        script.append ( "%d [label=\"%s\"];" % (v,k) )
    for k,v in mat_dict.iteritems () :
        i1 = vertex[k[0]]
        i2 = vertex[k[1]]
```

```
        if i1 < i2 and v < 50000 :
            #on coupe des arcs car le tracé est trop long sinon
            script.append ( "%d -- %d [label=\"%skm\"];" % (i1,i2,v/1000))
    script.append ("}")
    return "\n".join( script)

if __name__ == "__main__" :
    matrice_line = charge_donnees ()
    mat = conversion_en_dictionnaire (matrice_line)
    script = graphviz_script (mat)
    drawGraph([], script , "im.png")
```

2) En utilisant les résultats retournés par la fonction `conversion_en_dictionnaire` (ou `charge_donnees`), on souhaite évaluer la complexité du problème en calculant deux choses :

1. le nombre de villes du graphe
2. le nombre maximum d'arcs reliés à une ville

Ecrire deux fonctions qui calculent ces deux nombres.

3) Adaptez la première fonction pour qu'elle retourne une liste contenant la liste des villes, chaque ville n'apparaissant qu'une fois.

Seconde demi-heure : approche innocente .

Dans cette partie, on utilise le dictionnaire (d) retourné par la fonction `conversion_en_dictionnaire`.

4) Extraire les trois distances reliant les trois villes *Paris*, *Beauvais*, *Dieppe* entre elles. Vérifier sur *GoogleMaps*[5] les trois itinéraires reliant ces trois villes deux à deux ? Par défaut, quel critère le site optimise-t-il ? Comparer les deux distances suivantes :

– distance(Paris,Dieppe)
– distance(Paris,Beauvais) + distance(Beauvais, Dieppe)

5) On cherche tous les triplets de villes (x, y, z) qui vérifient :

$$d(x, y) + d(y, z) < d(x, z) \tag{1}$$

Combien sont-ils ? (le graphe n'inclut pas forcément un arc (x, z) mais il inclut les deux autres)

Troisième demi-heure : algorithme approchée de Bellman[6] .

6) Ecrire une fonction `misajour` qui met à jour le dictionnaire d pour chaque triplet trouvé vérifiant la condition (1). La nouvelle valeur contient la distance la plus courte.

7) Modifier la fonction précédente pour qu'elle retourne le nombre de valeurs modifiées.

8) Appeler la fonction `misajour` une dizaine de fois en affichant le nombre de valeurs modifiées.

9) Après plusieurs itérations, on remarque que :

5. http://maps.google.fr/
6. http://fr.wikipedia.org/wiki/Algorithme_de_Bellman-Ford

1. Le nombre de valeurs modifiées est croissant ou décroissant ?
2. La somme de chaque distance est croissante ou décroissante ?
3. On suppose que cette somme est constante après un grand nombre d'itérations, que vaut alors `d["Paris","Dieppe"]` ?
4. Est-il alors possible de connaître le chemin le plus court qui relie Paris à Dieppe par la seule connaissance de d ?

La figure 2 vous aidera à trouver quelques réponses.

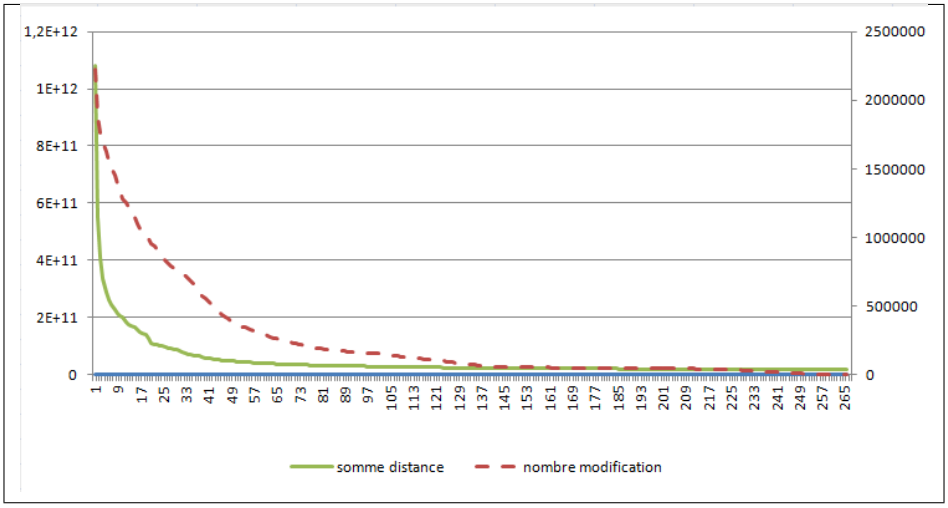


Figure 2 : *Evolution du nombre de modifications et de la somme des distances de chaque case de la matrice en fonction du nombre d'itérations.*

10) L'algorithme suggéré dans cette partie est différent de celui inventé par Bellman-Ford, pourquoi ?

Quatrième demi-heure : algorithme de Djikstra [7]

L'algorithme de Djikstra se concentre sur le chemin le plus court entre deux villes précises et permet de retourner le chemin le plus court.

11) Utiliser le pseudo-code de la page Wikipedia pour implémenter cet algorithme.

```
Fonction Dijkstra (nJuds, fils, distance, début, fin)
     Pour n parcourant nJuds
         n.parcouru = infini   // Peut être implémenté avec -1
         n.précédent = 0
     Fin pour
     début.parcouru = 0
     pasEncoreVu = nJuds
     Tant que pasEncoreVu != liste vide
         n1 = minimum(pasEncoreVu)   // Le nJud dans pasEncoreVu avec parcouru le plus petit
         pasEncoreVu.enlever(n1)
         Pour n2 parcourant fils(n1)                          // Les nJuds reliés à n1 par un arc
             Si n2.parcouru > n1.parcouru + distance(n1, n2) // distance correspond au
```

7. http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

```
                                                        // poids de l'arc reliant n1 et n2
                n2.parcouru = n1.parcouru + distance(n1, n2)
                n2.précédent = n1   // Dit que pour aller à n2, il faut passer par n1
            Fin si
        Fin pour
    Fin tant que
    chemin = liste vide
    n = fin
    Tant que n != début
        chemin.ajouterAvant(n)
        n = n.précédent
    Fin tant que
    chemin.ajouterAvant(debut)
    Retourner chemin
Fin fonction Dijkstra
```

12) Quel est le résultat entre Paris et Dieppe ?