Algorithmique
Partiel n° 4 (P4)

Info-spé (S4) - api
Epita

16 mai 2017 - 10*h*

Consignes (à lire) :

- Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - Aucune réponse au crayon de papier ne sera corrigée.
- La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- **Le code :**
  - Tout code doit être écrit dans le langage `Python` (pas de C, Caml, Algo ou autre).
  - **Tout code `Python` non indenté ne sera pas corrigé.**
  - Tout ce dont vous avez besoin (classes, fonctions, méthodes) est indiqué dans les annexes (dernière page). **Lisez-les** !
  - Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).
    Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.
- Durée : 2h00

Exercice 1 (ARM et PCC ... – *3 points*)

Le graphe de la figure 1 représente les possibilités d'alimentation en électricité de 6 villes par une centrale ainsi que le coût d'exploitation de ces différentes connexions.

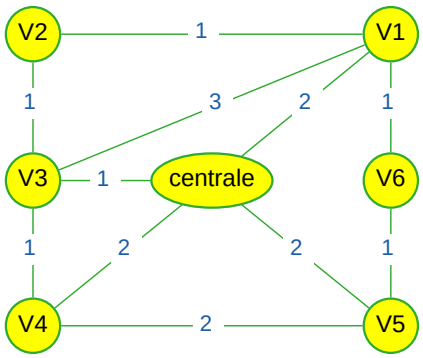

FIGURE 1 – Graphe non orienté valué.

1. Sur quel type de graphes orientés peut-on exécuter l'algorithme de Bellman ?
2. Quel algorithme déterminant l'arm d'un graphe non orienté utilise un principe proche de celui de Dijkstra ?
3. Tracer un arm du graphe de la figure 1.
4. En considérant que les sommets sont toujours gérés en ordre croissant et en utilisant le principe de l'algorithme de Dijkstra, tracer l'arbre des plus courts chemins de racine "centrale" du graphe de la figure 1.

Exercice 2 (Graphe réduit – *4 points*)

Soit G un graphe orienté admettant k composantes fortement connexes : $C_0, C_1, \ldots, C_{k-1}$. On définit le *graphe réduit* de G (noté G_R) par $G_R = < S_R, A_R >$ avec :

— $S_R = \{C_0, C_1, \cdots, C_{k-1}\}$

— $C_i \to C_j \in A_R \Leftrightarrow$ Il existe au moins un arc dans G ayant son extrémité initiale dans la composante fortement connexe C_i et son extrémité terminale dans la composante fortement connexe C_j.

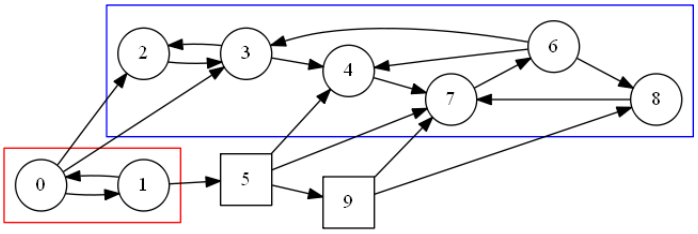

FIGURE 2 – Graphe G_1

On désire construire le graphe réduit d'un graphe G. On dispose de la liste des composantes fortement connexes de G (chaque composante est une liste des sommets qui la constitue).

Par exemple pour le graphe de la figure 2, la liste des composantes est :

```
scc = [[2, 3, 4, 6, 7, 8], [9], [5], [0, 1]]
```

Écrire la fonction `condensation(G, scc)` qui à partir du graphe G et *scc* sa liste de composantes fortement connexes retourne le graphe réduit G_R ainsi que le vecteur des composantes : un vecteur qui pour chaque sommet de G indique à quelle composante il appartient (le numéro du sommet dans G_R).

Par exemple avec le graphe G_1 de la figure 2 :

```
>>> (Gr, comp) = condensation(G1, scc)
>>> comp
[4, 4, 1, 1, 1, 3, 1, 1, 1, 2]
```

Exercice 3 (Graphes et mystère – *3 points*)

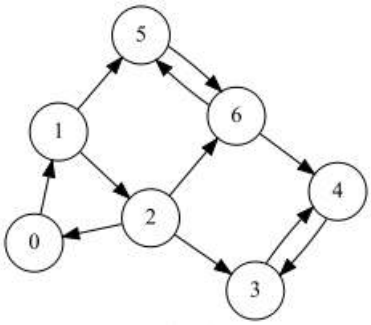

FIGURE 3 – Graphe G_2

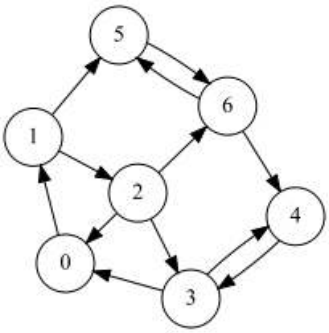

FIGURE 4 – Graphe G_3

```
1        def __test(G, x, p, c):
2            c += 1
3            p[x] = c
4            rx = p[x]
5            for y in G.adjLists[x]:
6                if p[y] == 0:
7                    (ry, c) = __test(G, y, p, c)
8                    if ry == -1:
9                        return (-1, c)
10                   rx = min(rx, ry)
11               else:
12                   rx = min(rx, p[y])
13
14           if rx == p[x]:
15               if p[x] != 1:
16                   return (-1, c)
17
18           return (rx, c)
19
20       def test(G):
21           p = [0] * G.order
22           c = 0
23           (_, c) = __test(G, 0, p, c)
24           return (c == G.order)
```

On suppose que dans le graphe passé en paramètre les listes d'adjacence sont triées en ordre croissant.

1. Pour chacun des appels suivants, avec G_2 et G_3 les graphes des figures 3 et 4 :
   — combien d'appels à `__test` ont été effectués ?
   — quel est le résultat retourné par `test` ?

   (a) `test(`G_2`)`

   (b) `test(`G_3`)`

2. Soit G un graphe orienté. Quelle information est retournée par `test(`G`)` ?

Exercice 4 (T-spanner – *10 points*)

Soit S un ensemble de n points dans $\mathbb{R}^2$ et soit $t \geq 1$ un nombre réel. Un t-spanner est un graphe non orienté G composé de l'ensemble des sommets S, tel que pour toutes paires de points p et q de S, il existe une chaîne dans G entre p et q dont la longueur est plus petite ou égale à $t \times |pq|$ ($|pq|$ est la distance euclidienne entre p et q).

Le *stretch factor (facteur d'élasticité)* de G est défini comme le plus petit nombre réel t tel que G est un *t-spanner* de S.

Construction

Le graphe G est construit par ajouts successifs d'arêtes. Au départ, il ne contient que les sommets. On travaille avec toutes les paires de points. Celles-ci sont prises en ordre croissant de distances euclidiennes. Pour chaque paire de points (p, q), **s'il n'existe pas déjà de plus court chemin dans G entre p et q** de distance inférieure ou égale à $t \times |pq|$, alors une arête $\{p, q\}$ de poids $|pq|$ est ajoutée à G.

L'algorithme pour construire un t-spanner G à partir d'un ensemble de points est donc le suivant :

```
L ← liste des paires de points triées par distances euclidiennes croissantes
S ← ensemble des n points
G ← <S,∅>
pour tout (p, q) ∈ L faire
    δ ← longueur du plus court chemin dans G entre p et q
    w = distance euclidienne entre p et q
    si δ > t * w alors
        ajouter l'arête {p, q} de coût w à G
    fin si
fin pour
```

Un exemple

Soit un ensemble S de 9 points (voir figure 5), dont les coordonnées sont données par la liste suivante :

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Les distances euclidiennes pour chaque paire de points de S :

(p, q)	$\|pq\|$
(0, 1) (0, 3) (1, 2) (1, 4) (2, 5) (3, 4) (3, 6) (4, 5) (4, 7) (5, 8) (6, 7) (7, 8)	1
(0, 2) (0, 6) (1, 7) (2, 8) (3, 5) (6, 8)	2
(0, 4) (1, 3) (1, 5) (2, 4) (3, 7) (4, 6) (4, 8) (5, 7)	$\sqrt{2} = 1,4142...$
(0,8) (2,6)	$2\sqrt{2} = 2,8284...$
(0, 5) (0, 7) (1, 6) (1, 8) (2, 3) (2, 7) (3, 8) (5, 6)	$\sqrt{5} = 2,2360...$

Les figures 6 et 7 sont des t-spanner de S, respectivement de stretch factors 1,5 et 3.

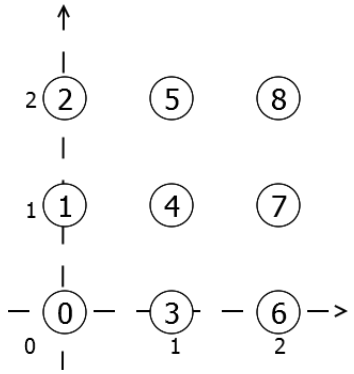

FIGURE 5 – Points

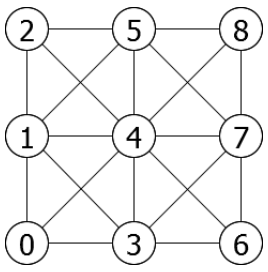

FIGURE 6 – Stretch factor = 1,5

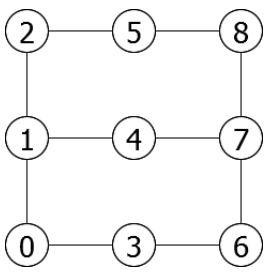

FIGURE 7 – Stretch factor = 3

Pour la construction du t-spanner, les distances euclidiennes ont déjà été calculées. C'est donc directement une liste de triplets $(p, q, |pq|)$ (non triée) qui est fournie.

Pour l'exemple ci-dessus, voici le début de la liste :

```
L = [(0, 1, 1.0),
     (0, 2, 2.0),
     (0, 3, 1.0),
     (0, 4, 1.4142135623730951),
     (0, 5, 2.23606797749979),
     (0, 6, 2.0),
     (0, 7, 2.23606797749979),
     (0, 8, 2.8284271247461903),
     (1, 2, 1.0),
     ...]
```

1. Donner les t-spanners des points de la figure 5 :
   (a) pour un stretch factor de 2
   (b) pour un stretch factor de 5
2. **Fonctions à écrire :**
   (a) Écrire la fonction `Dijkstra(G, src, dst)` qui retourne la longueur du plus court chemin entre *src* et *dst* dans G, $+\infty$ si le chemin n'existe pas.
   (b) Écrire la fonction pathGreedy(n, L, t) qui retourne un t-spanner (de stretch factor = t) pour l'ensemble de n points (numérotés de 0 à $n-1$) avec L la liste des triplets (p, q, $|pq|$) (telle que décrite ci-dessus).

bonus Lorsque le stretch factor est $n-1$ avec n le nombre de points, à quoi correspond le t-spanner ?

Annexes

Graphs

Les graphes manipulés sont non-vides. Ils ne contiennent pas de liaisons multiples.

```
    # new graph
    G = Graph(order, False)
    # new edge (x, y)
    G.addEdge(x, y)

    # new weighted digraph
    G = Graph(order, True, costs = True)
    # new edge (x, y) with cost w
    G.addEdge(x, y, w)
```

Les tas

Les tas de Python

```
from heapq import *
help(heapq)
    Usage:
        heap = []             # creates an empty heap
        heappush(heap, item) # pushes a new item on the heap
        item = heappop(heap) # pops the smallest item from the heap
```

Les tas AlgoPy

Nos tas ne peuvent manipuler que des couples (x, val) avec $x \in [0, n[$ et val la valeur pour le tri.

```
from AlgoPy/heap import *

    Usage:
        H = Heap(size)     # creates an empty heap
        update(H, x, val) # if x not in H, pushes it with val on the heap,
                          # else updates its position according to val
        (x, val) = pop(H) # pops the smallest item from the heap
        isEmpty(H)        # tests if H is empty!
```

Les fonctions que vous pouvez utiliser

— Toutes les fonctions et méthodes sur les listes

— `range`

— `max`, `min`, `abs`

Vos fonctions

Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.