Algorithmics
Final Exam #4 (P4)

Undergraduate 2^{nd} year (S4) - API
EPITA

16 May 2017 - 10*h*

Instructions (read it) :

- You must answer on **the answer sheets provided.**
  - No other sheet will be picked up. Keep your rough drafts.
  - Answer within the provided space. **Answers outside will not be marked**: Use your drafts!
  - Do not separate the sheets unless they can be re-stapled before handing in.
  - Penciled answers will not be marked.
- The presentation is negatively marked, which means that you are marked out of 20 points and the presentation points (maximum of 2) are taken off this grade.
- **Code:**
  - All code must be written in the language `Python` (no C, CAML, ALGO or anything else).
  - **Any `Python` code not indented will not be marked.**
  - All that you need (class, types, routines) is indicated in the appendix (last page). **Read it!**
  - You can write your own functions as long as they are documented (we have to know what they do).
    In any case, the last written function should be the one which answers the question.
- Duration : 2h

Exercise 1 (MST and SP ... – *3 points*)

The graph of figure 1 represents the possibilities of power supply of 6 cities by a "power plant" as well as the operating cost of these different connections.

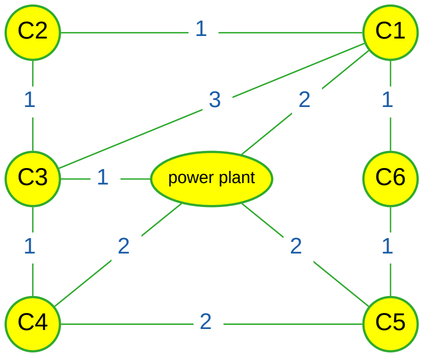

Figure 1: Undirected weighted graph.

1. On which kind of directed graphs can the Bellman algorithm be executed?
2. Which algorithm determines the mst of an undirected graph using a principle close to that of Dijkstra?
3. Draw an mst of the graph of figure 1.
4. Considering that vertices are managed in increasing order and using the principle of the Dijkstra algorithm, draw the tree of the shortest paths from the "power plant" vertex of the graph of figure 1.

Exercise 2 (Condensation – *4 points*)

Let G be a digraph with k strongly connected components: C_0, C_1, ..., C_{k-1}. The *condensation* of G is the digraph $G_R = < S_R, A_R >$ defined by:

- $S_R = \{C_0, C_1, \cdots, C_{k-1}\}$
- $C_i \rightarrow C_j \in A_R \Leftrightarrow$ There exists at least an edge in G with its head in the strongly connected component C_i and its tail in the strongly connected component C_j.

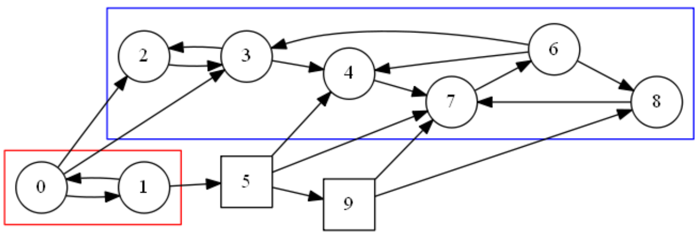

Figure 2: Digraph G_1

The aim here is to build the condensation of a graph G. We have the list of the strongly connected components of G (each component is a list of the vertices it contains).
For instance, the following list is the component list of the graph in figure 2:

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

Write the function `condensation(`G`,` scc`)` that builds the condensation G_R of a digraph G, with scc its component list. The function returns G_r and the vector of components: a vector that give for each vertex, the number of its component (the vertex in G_R).

For instance, with G_1 the graph in figure 2:

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

Exercise 3 (Digraphs and Mystery – *3 points*)

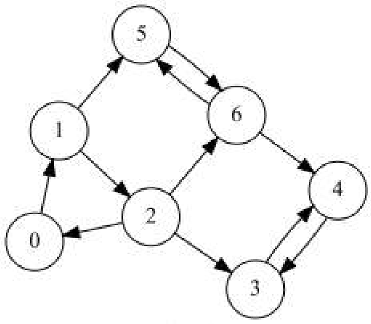

Figure 3: Digraph G_2

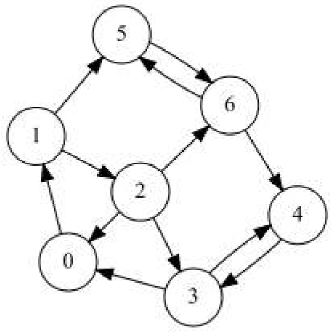

Figure 4: Digraph G_3

```python
def __test(G, x, p, c):
    c += 1
    p[x] = c
    rx = p[x]
    for y in G.adjLists[x]:
        if p[y] == 0:
            (ry, c) = __test(G, y, p, c)
            if ry == -1:
                return (-1, c)
            rx = min(rx, ry)
        else:
            rx = min(rx, p[y])

    if rx == p[x]:
        if p[x] != 1:
            return (-1, c)

    return (rx, c)

def test(G):
    p = [0] * G.order
    c = 0
    (_, c) = __test(G, 0, p, c)
    return (c == G.order)
```

We assume that the adjacency lists are sorted in increasing order in the graph in parameter.

1. Let G_2 and G_3 be the digraphs in figures 3 and 4. For each of the following calls:
   - how many calls of `__test` have been done?
   - what is the result returned by `test`?

   (a) `test(`G_2`)`
   (b) `test(`G_3`)`

2. Let G be a digraph. What is the information returned by `test(`G`)`?

Exercise 4 (T-spanner – *10 points*)

Let S be a set of n points in $\mathbb{R}^2$ et $t \geq 1$ a real number. A t-spanner is a graph G where vertices are the points in S such that for each pair of points p and q of S, there exists a path in G between p and q whose length is smaller or equal to $t \times |pq|$ ($|pq|$ is the spacial distance between p and q).

The *stretch factor* of G is defined as the smallest real t such that G is a t-spanner of S.

Construction

The graph G is built by progressively adding edges. At first, the graph only contains the vertices. We work with all the possible pairs of points. Those are taken with spacial distances in increasing order. For each pair of points $(p,\ q)$, **if there is not already a shortest path in G between p and q** whose distance less or equal to $t \times |pq|$ then we add an edge $\{p, q\}$ of weight $|pq|$ to G.

Thus, the algorithm to build a t-spanner from a set S is the following:

```
L ← point pair list sorted by increasing spacial distance
S ← set of the n points
G ← <S,∅>
for each (p, q) ∈ L do
    δ ← length of the shortest path between p and q in G
    w = spacial distance between p and q
    if δ > t * w then
        add edge {p, q} with weight w in G
    end if
end for
```

An example

Let S a set of 9 points (see figure 5), whose coordinates are given in the following list:

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

Spacial distances for each pair of points of 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)	$\sqrt{5} = 2,2360...$

The graphs in figure 6 and 7 are t-spanner of S, with stretch factors respectively 1,5 and 3.

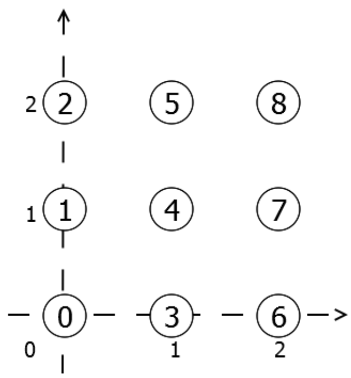

Figure 5: Points

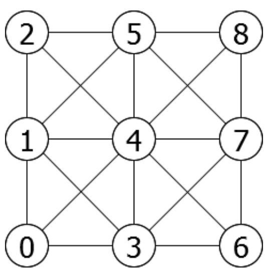

Figure 6: Stretch factor = 1,5

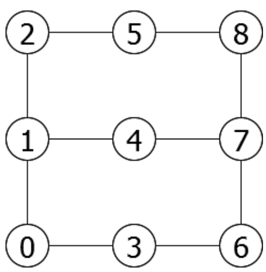

Figure 7: Stretch factor = 3

To build the t-spanner, the spacial distances have already been calculated. An unordered list of triplets $(p, q, |pq|)$ is given.

Here is the beginning of the list for our example:

```
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. Give the t-spanners of points in figure 5:
   (a) for a stretch factor of 2
   (b) for a stretch factor of 5

2. **Functions:**
   (a) Write the function `Dijkstra(G, src, dst)` that returns the length of the shortest path between *src* and *dst* in G, $+\infty$ if there is no path.
   (b) Write the function pathGreedy(n, L, t) that returns a t-spanner (with stretch factor $= t$) for the set of n points (number form 0 to $n-1$) with L the list of triplets (p, q, $|pq|$) (as described above).

bonus When the stretch factor is $n - 1$ with n the number of points, what is the t-spanner?

Appendix

Graphs

The graphs we work on are not empty and do not contain multiples edges.

```
    # 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)
```

Heaps

Python's heaps

```
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
```

AlgoPy's Heaps

Our heaps can only work with pairs (x, val) with $x \in [0, n[$ and val the order value.

```
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!
```

Functions you can use

- Any function or method on lists
- `range`
- `max`, `min`, `abs`

Your functions

You can write your own functions as long as they are documented (we have to know what they do).
In any case, the last written function should be the one which answers the question.