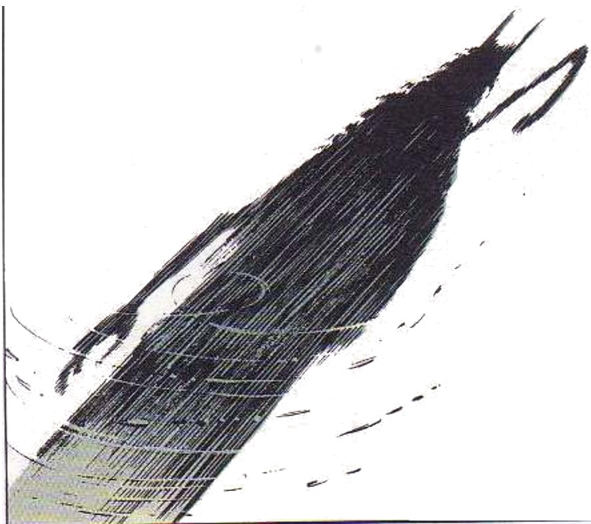
Algorithmics
Final Exam #4 (P4)

Undergraduate 2nd year S4
Epita

15 May 2018 - 10 : 00

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)!
  - You can write your own functions as long as they are documented (we have to known what they do).
    In any case, the last written function should be the one which answers the question.
- Duration : 2h



Exercise 1 (Exhausting deposit. . . – *5 points*)

Miners want to secure the traffic between the different extraction points (represented by vertices) connected by galeries. All extraction points are accessible from the outside. The network is sufficiently complex to allow several paths from one extraction point to another. It is therefore not necessary for each gallery to be secured as long as there is always at least one safe route between two extraction points.

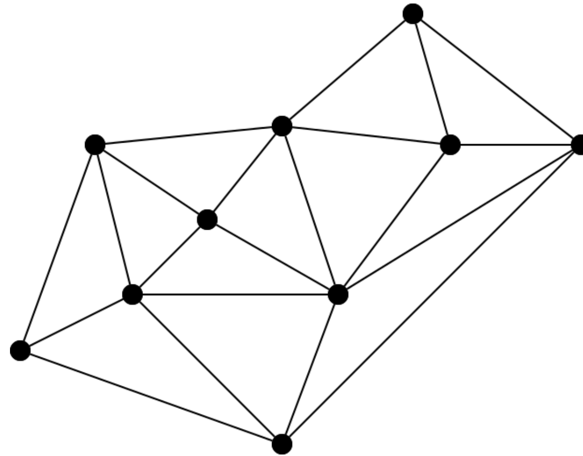


Figure 1: Graph of the gallery network

1. We want to determine the smallest number of galleries to secure:
   (a) to which graph model corresponds the solution?
   (b) In the case on figure 1, how many galleries must be secured?
   (c) Suggest a graphic solution.
   (d) By generalizing to a N extraction point network, how many galleries should be secured?
2. We detail the problem analysis:
   For each gallery, the cost of securing work has been calculated (see figure 2).

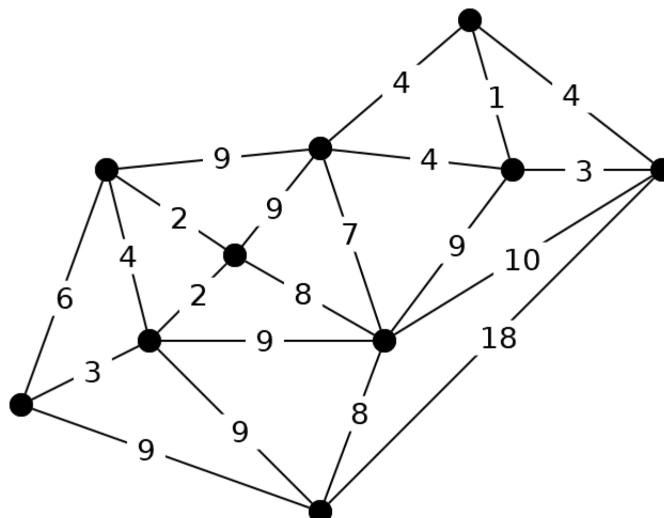


Figure 2: Weighed graph: securing the gallery network

   (a) Suggest a graphic solution that minimises the costs of securing work.
   (b) Does this solution be unique?
   Why?

Exercise 2 (Asterix and the Soothsayer – *13 points*)

The aim is to find the best path between a `source` and a `destination` in a simple digraph weighted with positive costs treating a minimum of uninteresting vertices. To do this, we need the help of the diviner *Heuristix*. The diviner *Heuristix* is able to tell us if a vertex is near the `destination` or not. Like any diviner, his response is not accurate.

If we give *Heuristix* a graph, a vertex `s` and a `destination` in the graph, it returns a positive real estimating the distance to travel from the vertex to the `destination` (the smaller the value, the closer the vertex is to `destination`). The estimation does not take into account the distance from the `source` to reach the vertex `s`.

1. **The algorithm:**

   We want to build an algorithm using the informations of the diviner, estimations given by the function `heuristix` (see appendices). The algorithm to write must give the best path between a `source` and a `destination`. The difference with the classical algorithms of shortest paths is that we use the diviner's guess for the choice of vertices to be processed: at each iteration we choose the vertex whose **sum** of the computed distance from the `source` **with** the guessing of the diviner is the **smaller**. It is also necessary to avoid the circuits, so **we will not treat again the vertices already treated**: in the diviner's language, these vertices are said "closed", those not yet treated are "open" .

   (a) What is the name of this algorithm?

   (b) **Use of additional data structures:** The class to use for the graph is given in appendix (implementation by adjacency lists). You can also use all types / classes (and associated functions and methods) given in the appendix. Optimization is of course required.
   How to represent the "open" vertices?
   How to represent the "closed" vertices?

   (c) **What about the complexity?** Assuming the cost of the function `heuristix` is constant, give an order of growth of the algorithm complexity (in the worst case: all vertices are used) with the choices made in the previous question (with n the vertex number, p the edge number).

   (d) **Implement the algorithm:** it is a function that takes the graph (with positive costs), the two vertices `source` and `destination` as parameters ; it gives as result the path from the `source` to the `destination` obtained as a list of vertices (from the source to the destination).
   At first, it will be assumed that the path always exists.
   Bonus if the function raises an exception when the path does not exist!

2. **Deviners:**

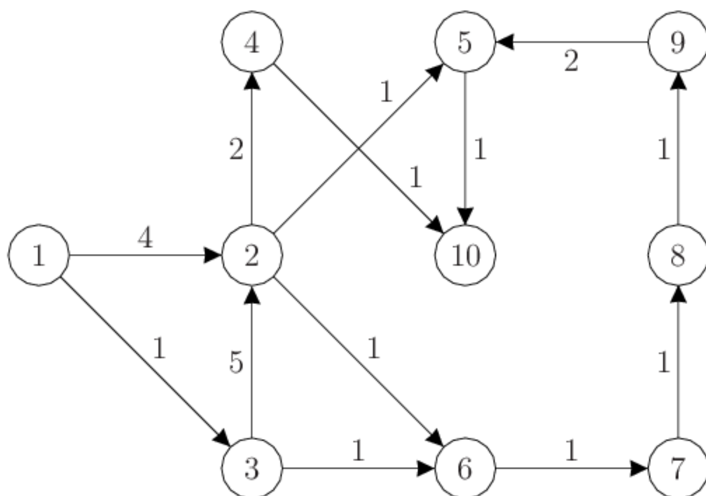


Figure 3: Digraph for the diviners

The shortest path from 1 to 10?

When we call the diviner Heuristix, three people present themselves:

⋆ **Heuristix the Dutchman** (called *HeuristixD*): very (too?) optimistic, he thinks that all the vertices are close to the destination.
In this case, he systematically returns 0.

⋆ **Heuristix of the New World** (called *HeuristixM*): Germain (it is an Angle) having traveled a lot (hence his nickname) he gives us an estimation corresponding to the minimum number of edges to reach the destination, he likes square things.
In this case, he thus gives us:

s	1	2	3	4	5	6	7	8	9	10
HeuristixM (s)	3	2	3	1	1	5	4	3	2	0

⋆ **Heuristix the Byzantine** (called *HeuristixB*): Like all Byzantine generals, HeuristixB often answers anything.
In this case, he gives us the sum of incoming edges:

s	1	2	3	4	5	6	7	8	9	10
HeuristixB (s)	0	9	1	1	3	2	2	1	1	2

(a) Apply the algorithm on the graph in figure 3 with the heuristiques *HeuristixD* and *HeuristixM* to find the best path between vertices 1 and 10: fill the vectors *dist* (the distances from the source, without the heuristic . . .) and *parent* and list the processed vertices (in order). As usual, if several choices are possible we will choose the vertex of lower number.

(b) To which algorithm does the solution with *HeuristixD*'s estimation correspond (warning, spelling counts . . .)?

(c) **Bonus** What to think of *HeuristixB*'s estimation? Is it better than *HeuristixM*'s?

Exercise 3 (What is this? – *4 points*)

```
def __dfs(G, s, x, y, M):
    M[s] = True
    for adj in G.adjlists[s]:
        if adj == y:
            if s != x:
                return True
        elif not M[adj]:
            if __dfs(G, adj, x, y, M):
                return True
    return False

def dfs(G, x, y):
    M = [False] * G.order
    return __dfs(G, x, x, y, M)
```

```
def what(G):
    n = G.order
    L = []
    for x in range(n):
        for y in G.adjlists[x]:
            if x < y:
                L.append((G.costs[(x,y)], x, y))
    L.sort()  # sorts L in increasing order
    while L != []:
        (_, x, y) = L.pop()
        if dfs(G, x, y):
            G.removeedge(x, y)
```

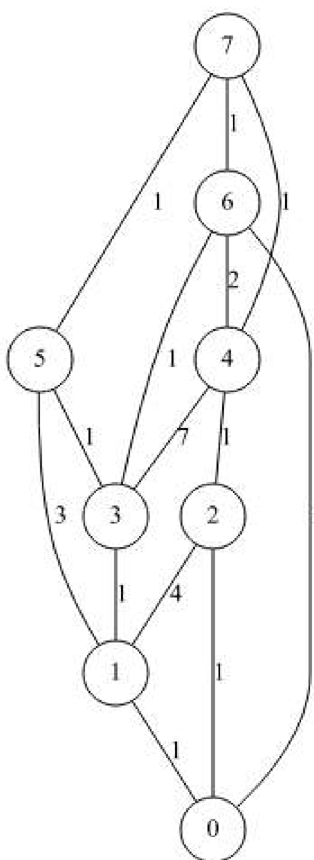


Figure 4: Graph for `what`

1. What does the function `dfs(G)` test?

2. The function `what`

   (a) Give the result of the application of the function `what` to the graph in figure 4 (draw the graph).

   (b) What property has the graph after application of the function?

   (c) How this function can be optimised?

Appendix

Classes `Graph` and `Heap` are supposed imported. Graphs we manage cannot be empty.

Graphs

```
class Graph:
    def __init__(self, order, directed = False, costs = False):
        self.order = order
        self.directed = directed
        self.adjLists = []
        for i in range(order):
            self.adjLists.append([])
        if costs:
            self.costs = {}
        else:
            self.costs = None

    def addedge(self, src, dst):
        self.adjlists[src].append(dst)
        if not self.directed and dst != src:
            self.adjlists[dst].append(src)
        if cost:
            self.costs[(src, dst)] = cost
            if not self.directed:
                self.costs[(dst, src)] = cost

    def removeedge(self, src, dst):
        if dst in self.adjlists[src]:
            self.adjlists[src].remove(dst)
            if self.costs:
                self.costs.pop((src, dst))
            if not self.directed and dst != src:
                self.adjlists[dst].remove(src)
                if self.costs:
                    self.costs.pop((dst, src))
```

Heap

- `Heap(`n`)`
  returns a new heap with size n
- H`.push(`s`, `val`)`
  add s of value val in H
- H`.update(`s`, `$newval$`)`
  if s not in H same as `H.push(`s`, `$newVal$`)`
  else updates the heap after minimization of s's value with $newVal$
- `(`val`, `s`) = `H`.pop()`
  returns and deletes the element (s) of smallest value (val) in heap
- H`.isempty()`
  tests whether H is empty

List

- `len(L)`
- `L.append(x)`
- `L.pop() L.pop(i)`
- `L.insert(i, x)`

Others

- `range`.

The diviner

The function `heuristix(`G`, `s`, `dst`)` gives an estimation of the distance between s and dst in G.

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.