Last name	
First name	
Group	

Grade	

Algorithms
Shortest Paths and MST

Undergraduate 2nd year S4 Epita

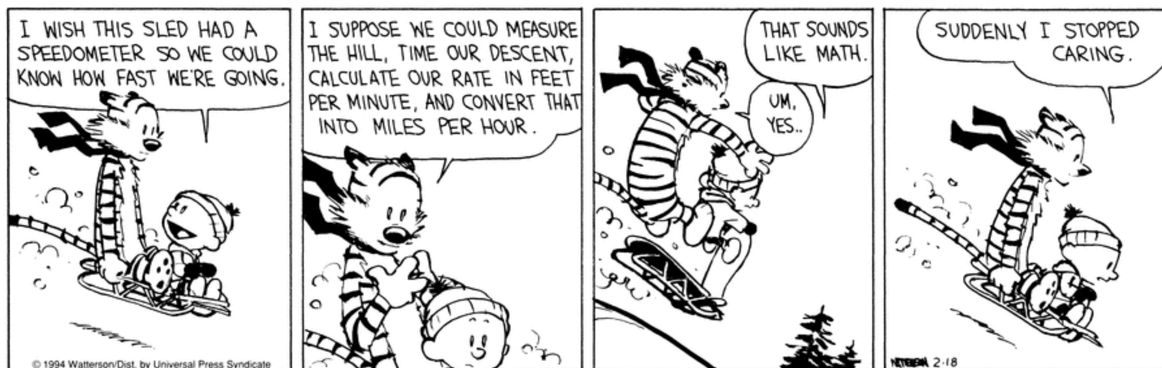
Exam B8

May, the 14th 2025

1	
2	
3	
4	

Instructions (read it):

- ☐ You must answer on **this subject.**
  - Answer within the provided space. **Answers outside will not be marked.**
  - 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.**
  - The only classes, functions, methods that can be used are given in the **appendix**.
  - Your functions must follow the given examples of application.
  - 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.
  - As usual, optimization is graded. If you write non-optimized functions, you will be graded on fewer points.[1]
- ☐ Duration: 2h00



[1]Sometimes a few points are better than none.

Exercise 1 (Longest-shortest – *9 points*)

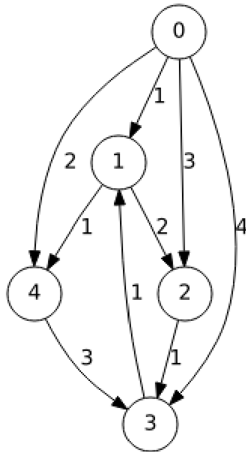


Figure 1: G1

In a digraph with strictly positive costs, a shortest path with as many arcs as possible is called *longest-shortest path*.

For example, on the graph G1 in figure 1, there are three shortest paths of cost 4 from vertex 0 to vertex 3:

- $0 \rightarrow 3$ of length 1 arc
- $0 \rightarrow 2 \rightarrow 3$ of length 2 arcs
- $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ of length 3 arcs

The third one is a *longest-shortest path*.

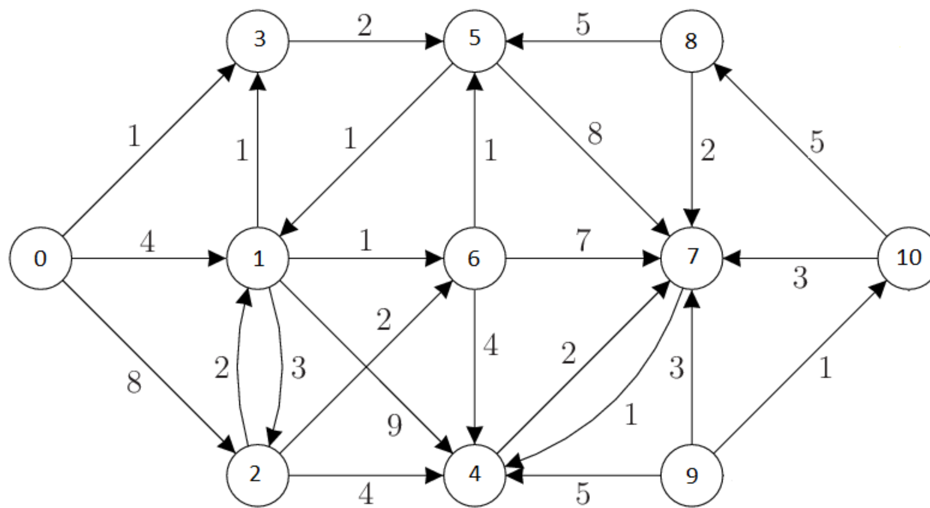


Figure 2: G2

1. Give 3 shortest paths (all with minimum cost) between vertices 0 and 7 in the graph G2 in figure 2.

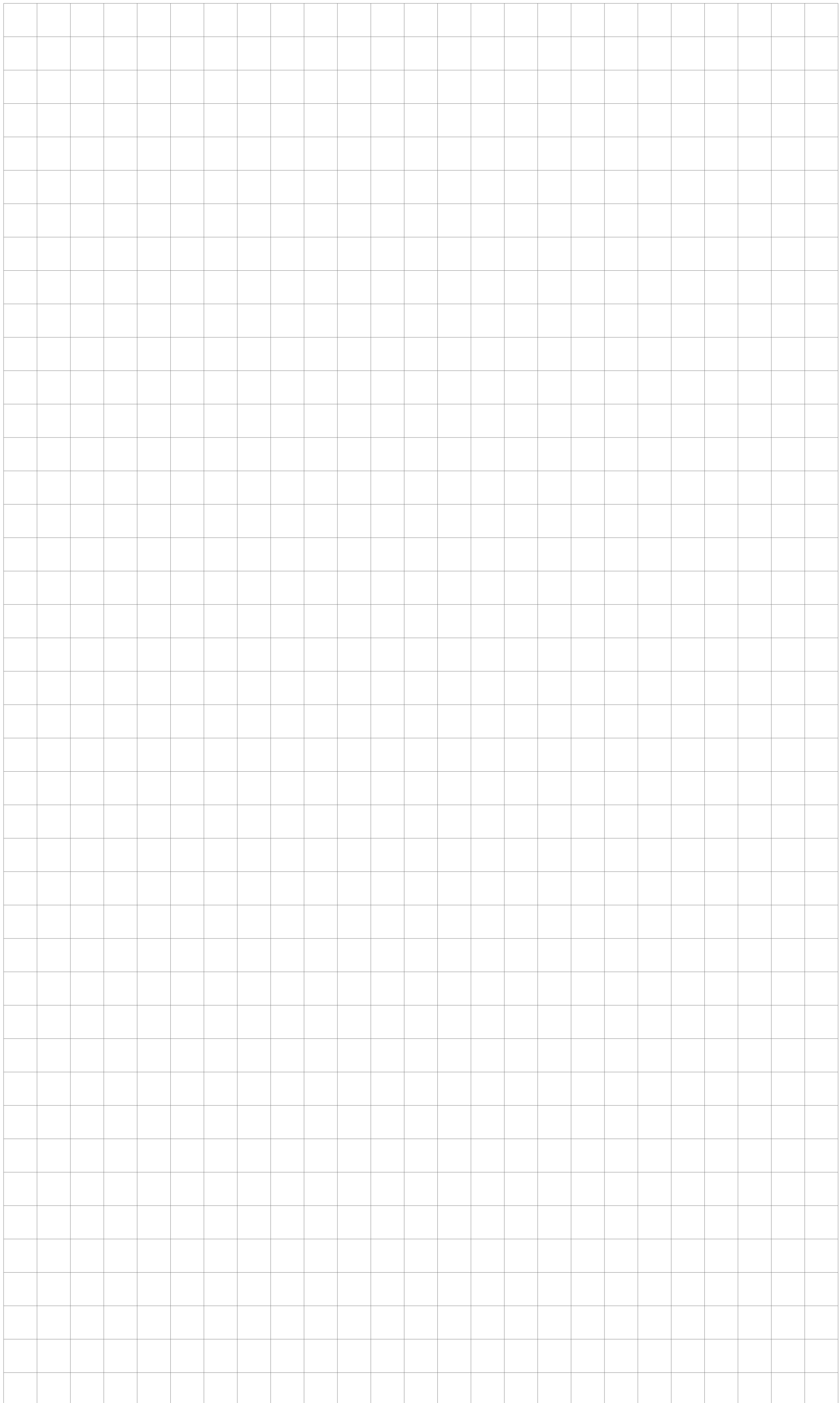
   - Path 1 = ____________________
   - Path 2 = ____________________
   - Path 3 (longest-shortest) = ____________________

2. Write (next page) the function `longest-shortest(G, src, dst)` that searches for a *longest-shortest path* between the two distinct vertices `src` and `dst` in the directed graph `G` with strictly positive costs. The function returns:
   - the *longest-shortest path* found (shortest path containing the most arcs) ;
   - the cost of this path ;
   - its length (number of arcs).

   If the path between `src` and `dst` does not exist in `G`, the function returns `None`.

   Application examples:

```
>>> longest_shortest(G1, 0, 3)
([0, 1, 2, 3], 4, 3)
>>> longest_shortest(G1, 0, 4)
([0, 1, 4], 2, 2)
>>> print(longest_shortest(G2, 0, 9))
None
```



Exercise 2 (What is this? – *6 points*)

```python
def __aux(G, x, M, L):
    M[x] = True
    for y in G.adjlists[x]:
        if not M[y]:
            __aux(G, y, M, L)
    L.append(x)
def aux(G, src):
    M = [False] * G.order
    L = []
    __aux(G, src, M, L)
    return L

def shortestpath(G, src, dst, L, P1):
    dist = [infty] * G.order
    dist[src] = 0
    P = [None] * G.order
    P[src] = -1
    i = len(L)-1
    while i >= 0 and L[i] != dst:
        x = L[i]
        for y in G.adjlists[x]:
            if P1[y] != x:
                if dist[x] + G.costs[(x, y)] < dist[y]:
                    dist[y] = dist[x] + G.costs[(x, y)]
                    P[y] = x
        i -= 1
    return (dist, P)

def mystery(G, src, dst):
    L = aux(G, src)                                    # 1.
    P0 = [None] * G.order
    (dist1, P1) = shortestpath(G, src, dst, L, P0)     # 2.
    if dist1[dst] == infty:
        return (None, None)
    else:
        (dist2, P2) = shortestpath(G, src, dst, L, P1) # 3.
        return (dist1[dst], dist2[dst])                # 4.
```

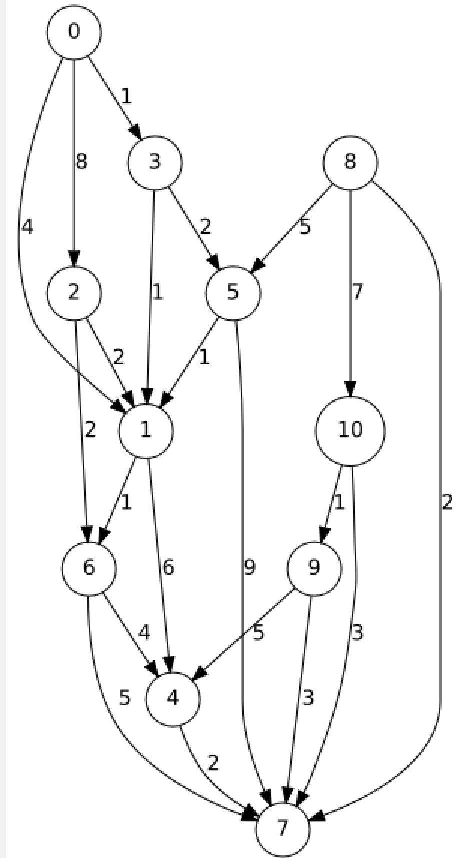


Figure 3 - Graph G_3

When calling `mystery(`G_3`, 0, 7)` with G_3 the graph in figure 3:

1. Give the list `L` built by the call `aux(G, src)`, line 30:

2. Give the vectors `dist1` and `P1` results of the first call to `shortestpath` line 32:

	0	1	2	3	4	5	6	7	8	9	10
dist1											
P1											

3. Give the vectors `dist2` and `P2` results of the second call to `shortestpath` line 36:

	0	1	2	3	4	5	6	7	8	9	10
dist2											
P2											

4. Result returned by `mystery(`G_3`, 0, 7)`

Exercise 3 (MST or not? – *3 points*)

Check whether the spanning subgraph T built by the following process is a minimum spanning tree (MST) of G or not. If it is, explain why, otherwise justify and give a counterexample.

Let $G =< S, A, C >$ be an undirected weighted connected graph of order n. Initially T is empty. Let s be any vertex. An edge v incident to s of minimum cost is chosen and added to T. Let x be the other endpoint of the edge. The procedure starts again with this vertex x, looks for an incident edge to x of minimum cost that is not in T, adds it to T and so on. The procedure stops when T has $n - 1$ edges.

Is the spanning subgraph T a MST of G? ☐ YES ☐ NO

Justification:

Exercise 4 (MST via Kruskal – *2 points*)

Consider the following undirected weighted graph `G=<S,A,C>`:

```
    S = {A, B, C, D, E, F, G, H}
and A = {{A,B,10}, {A,C,16}, {A,D,7}, {A,E,6}, {B,C,4}, {B,F,1}, {B,G,2}, {C,E,14}, {C,F,8},
      {D,E,9}, {D,H,11}, {E,F,12}, {E,H,13}, {F,G,3}, {F,H,15}, {G,H,5}}
```

Edges are given, in ascending alphabetical order, as `{vertex,vertex2,edge cost}`.

For this graph, use Kruskal's algorithm to find a spanning tree of minimum weight. Your answer should include a complete list of **processed edges**, indicating which edges you chose for your tree and which (if any) you rejected during the execution of the algorithm.

In these two lists, the edges must be given in the order of processing.

1. The chosen edges are:

2. The rejected edges are:

Appendix

All used classes are assumed imported.

Graphs

All exercises use the implementation with adjacency lists of graphs. Adjacency lists are sorted in increasing order.

Graphs we manage cannot be empty (neither the set of vertices nor the set of links are empty). There are neither multiple edges nor loops.
Reminder of graph attributes and methods:

```
# G: Graph
G.order
G.adjlists

# cost of the edge (x, y)
G.costs[(x, y)]

# add an edge:
G.addedge(x, y, cost)

# remove an edge
G.removeedge(x, y)  # raises an exception if the edge (x, y) does not exist
```

Heap from `algo_py`

Here, a heap of "maximum size" n contains pairs (element, value) $= (s, val)$ with $s \in [0, n-1[$, and val the value used for priority.

- `Heap(`n`)` returns a new heap with maximum size n
- H`.push(`s`, `val`)` adds s of value val in H ($s \in [0, n[$)
- 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

Queues

- `Queue()` returns a new queue
- q`.enqueue(`e`)` enqueues e in q
- q`.dequeue()` returns the first element of q, dequeued
- q`.isempty()` tests whether q is empty

`List`

- `len(L)`
- `L.append(x)`
- `L.pop()` `L.pop(i)`
- `L.sort()` sorts `L` in place ($O(n\ log\ n)$, $n =$ `len(L)`)
- `L.reverse()` reverses `L` in place

Others

- `range`
- `min`, `max`, `abs` ...
- value *infinity*: `infty` or ∞

Your functions

You can write your own additional functions as long as you give their **specifications** (we have to know what they do).

In any case, the last written function should be the one which answers the question.