Nom	
Prénom	
Groupe	

Note	

Algorithmique
Plus courts chemins et ARPM

Spé S4 Epita

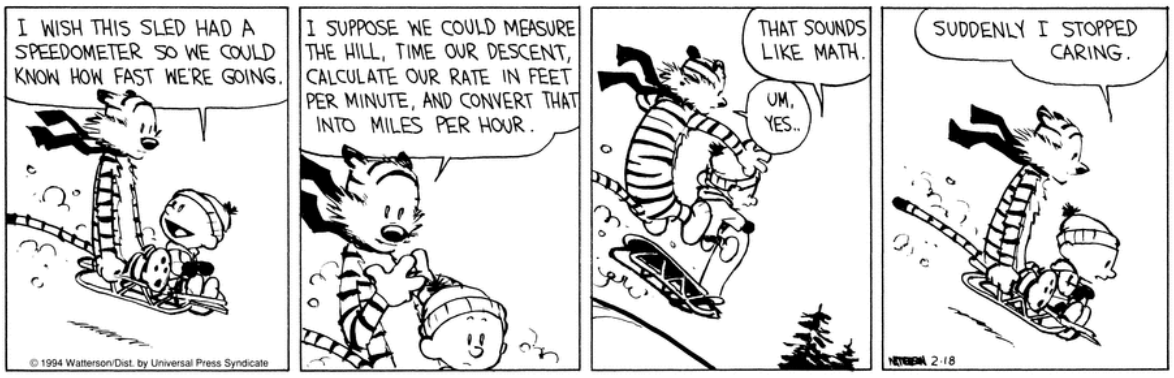
Examen B8

14 mai 2025

1	
2	
3	
4	

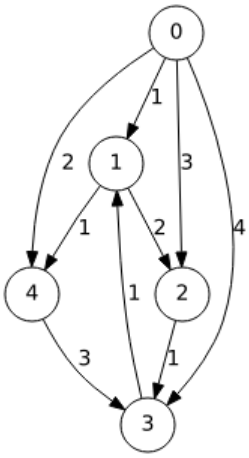
Consignes (à lire) :

- Vous devez répondre directement **sur ce sujet**.
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées**.
  - 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.
- **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.**
  - Les seules classes, fonctions, méthodes que vous pouvez utiliser sont données en **annexe**.
  - Vos fonctions doivent impérativement respecter les exemples d'applications donnés.
  - 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.
  - Comme d'habitude l'optimisation est notée. Si vous écrivez des fonctions non optimisées, vous serez notés sur moins de points. [1]
- Durée : 2h00



1. Des fois, il vaut mieux moins de points que pas de points.

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



Dans un graphe orienté à coûts strictement positifs, on appelle *longest-shortest path* un plus court chemin passant par le plus d'arcs possibles.

Par exemple sur le graphe `G1` de la figure 1, il y a trois plus courts chemins de coût 4 du sommet 0 au sommet 3 :
- $0 \to 3$ de longueur 1 arc
- $0 \to 2 \to 3$ de longueur 2 arcs
- $0 \to 1 \to 2 \to 3$ de longueur 3 arcs

Le troisième est un *longest-shortest path*.

Figure 1 – `G1`

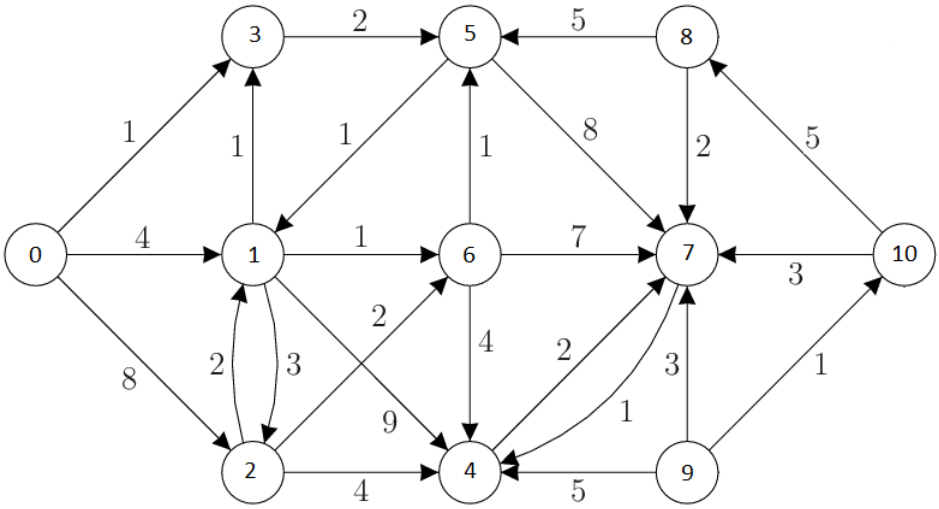


Figure 2 – `G2`

1. Donner 3 plus courts chemins (tous de coût minimum) entre les sommets 0 et 7 dans le graphe `G2` de la figure 2.
   - Path 1 = _____
   - Path 2 = _____
   - Path 3 (longest-shortest) = _____

2. Écrire (page suivante) la fonction `longest_shortest(G, src, dst)` qui cherche un *longest-shortest path* entre les deux sommets distincts `src` et `dst` dans le graphe orienté `G` à coûts strictement positifs. La fonction retourne :
   - le *longest-shortest path* trouvé (plus court chemin contenant le plus d'arcs) ;
   - le coût de ce chemin ;
   - sa longueur (en nombre d'arcs).

   Si le chemin entre `src` et `dst` n'existe pas dans `G` la fonction retourne `None`.

   Exemples d'applications :

```
>>> 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
```

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

```
1  def __aux(G, x, M, L):
2      M[x] = True
3      for y in G.adjlists[x]:
4          if not M[y]:
5              __aux(G, y, M, L)
6      L.append(x)
7  def aux(G, src):
8      M = [False] * G.order
9      L = []
10     __aux(G, src, M, L)
11     return L
12
13 def shortestpath(G, src, dst, L, P1):
14     dist = [infty] * G.order
15     dist[src] = 0
16     P = [None] * G.order
17     P[src] = -1
18     i = len(L)-1
19     while i >= 0 and L[i] != dst:
20         x = L[i]
21         for y in G.adjlists[x]:
22             if P1[y] != x:
23                 if dist[x] + G.costs[(x, y)] < dist[y]:
24                     dist[y] = dist[x] + G.costs[(x, y)]
25                     P[y] = x
26         i -= 1
27     return (dist, P)
28
29 def mystery(G, src, dst):
30     L = aux(G, src)                                    # 1.
31     P0 = [None] * G.order
32     (dist1, P1) = shortestpath(G, src, dst, L, P0)     # 2.
33     if dist1[dst] == infty:
34        return (None, None)
35     else:
36        (dist2, P2) = shortestpath(G, src, dst, L, P1) # 3.
37        return (dist1[dst], dist2[dst])                # 4.
```

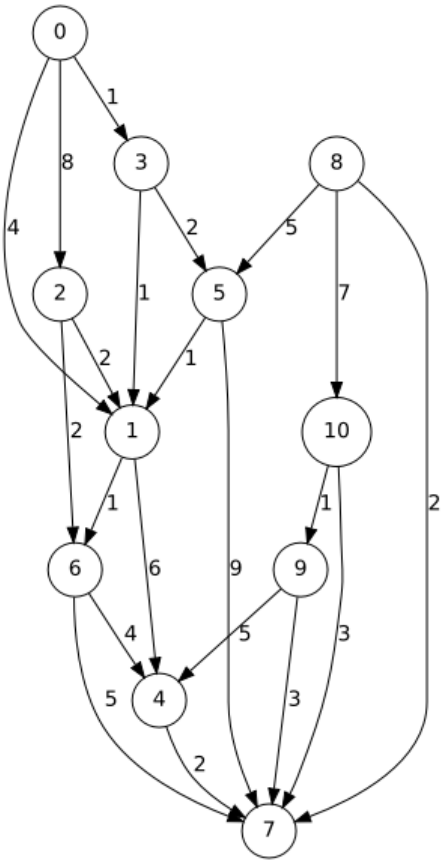


FIGURE 3 - Graphe G_3

Lors de l'appel `mystery(`G_3`, 0, 7)` avec G_3 le graphe de la figure 3 :

1. Donner la liste `L` construite par l'appel `aux(G, src)`, ligne 30 :

2. Donner les vecteurs `dist1` et `P1` résultats du premier appel à `shortestpath` ligne 32 :

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

3. Donner les vecteurs `dist2` et `P2` résultats du deuxième appel à `shortestpath` ligne 36 :

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

4. Résultat final retourné par `mystery(`G_3`, 0, 7)` :

Exercice 3 (ARM ou pas ? – *3 points*)

Examiner si le graphe partiel T obtenu dans la construction suivante est un arbre couvrant de poids minimum (ARM) de G ; si oui, expliquer pourquoi, sinon justifier et construire un contre-exemple.

Soit $G =< S, A, C >$ un graphe non orienté connexe valué d'ordre n. Initialement T est vide. Soit s un sommet quelconque. On choisit une arête v incidente à s, de coût minimum, qu'on ajoute à T. Soit x l'autre extrémité de l'arête. On recommence avec ce sommet x : on cherche une arête incidente à x , de coût minimum qui n'est pas dans T et on l'ajoute à T, et ainsi de suite. On s'arrête lorsque T a $n - 1$ arêtes.

Le graphe partiel T est-il un ARM de G ? ☐ OUI ☐ NON

Justification :

Exercice 4 (ARM via Kruskal – *2 points*)

Soit le graphe `G=<S,A,C>` non orienté valué suivant :

```
  S = {A, B, C, D, E, F, G, H}
et 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}}
```

Les arêtes sont données, en ordre alphabétique croissant, sous la forme `{sommet,sommet2,coût de l'arête}`.

Pour ce graphe, utilisez l'algorithme de Kruskal pour trouver un arbre couvrant de poids minimum. Votre réponse devra inclure une liste complète des **arêtes examinées**, indiquant quelles arêtes vous avez choisies pour votre arbre et lesquelles (s'il y en a) vous avez rejetées au cours de l'exécution de l'algorithme.

Dans ces deux listes, les arêtes doivent être données dans l'ordre de traitement.

1. Les arêtes retenues sont :

   _____

   _____

2. Les arêtes rejetées sont :

   _____

   _____

Annexes

Toutes les classes utilisées sont supposées importées.

Les graphes

Tous les exercices utilisent l'implémentation par listes d'adjacence des graphes. Les listes d'adjacence sont triées en ordre croissant.

Les graphes manipulés ne peuvent pas être vides (ni l'ensemble des sommets, ni celui des liaisons ne sont vides). Il n'y a pas de liaisons multiples ni boucles.

Rappel des attributs et méthodes sur les graphes :

```
# 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
```

Les tas de `algo_py`

Ici, un tas de "taille maximum" n contient des couples (élément, valeur) $= (s, val)$ avec $s \in [0, n-1[$, et val la valeur utilisée pour la priorité.

- `Heap(`n`)` retourne un nouveau tas de taille maximum n
- H`.push(`s`, `val`)` ajoute s de valeur val dans H ($s \in [0, n[$)
- H`.update(`s`, `$newval$`)` si s ($s \in [0, n[$) n'est pas présent, équivalent à `H.push(`s`, `$newVal$`)`, sinon met à jour le tas après minimisation de la valeur de s à $newVal$
- `(`val`, `s`) = `H`.pop()` supprime et retourne l'élément (s) de plus petite valeur (val)
- H`.isempty()` teste si H est vide

Les files

- `Queue()` retourne une nouvelle file
- q`.enqueue(`e`)` enfile e dans q
- q`.dequeue()` retourne le premier élément de q, défilé
- q`.isempty()` teste si q est vide

`List`

- `len(L)`
- `L.append(x)`
- `L.pop()` `L.pop(i)`
- `L.sort()` trie L en ordre croissant en place ($O(n\ log\ n)$, $n =$ `len(L)`)
- `L.reverse()` inverse L en place

Autres

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

Vos fonctions

Vous pouvez également écrire vos propres fonctions supplémentaires, dans ce cas vous devez donner leurs **spécifications** (on doit savoir ce qu'elles font).

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