Correction TD d'algorithmique

Cours d'algorithmique

—Master 1 / ISD—

Les documents manuscrits, sujets de travaux pratiques et dirigés ainsi que les supports de cours sont autorisés. Tous les autres documents tels que livres, calculatrices, téléphones portables et ordinateurs sont interdits.

Pour toutes les questions de complexité, bien préciser le *modèle de calcul* : taille des instances, opérations élémentaires.

Durée : 1h30 heures

▸ Exercice 1. (Application de UnionFind)

On suppose écrite une classe `UnionFind` avec l'interface suivante

- `UnionFind(S : ensemble)` : Constructeur avec pour support l'ensemble `S` ;
- `UnionFind.find(x : S) : S` : Retourne l'élément canonique associé à `x` ;
- `UnionFind.union(x, y : S)` : Réuni les classes de `x` et `y`. On suppose que `x` et `y` sont canoniques.

On ne demande pas d'écrire la classe `UnionFind`.

1. Étant donné un graphe $G = (E, V)$, écrire un algorithme basé sur `UnionFind` qui répond si G est connexe ou non.

✂ ----------

```
def is_conected((V, E)):
    UF = UnionFind(V)
    nbcomp = len(V)
    for (i,j) in E:
        i0 = UF.find(i)
        j0 = UF.find(j)
        if i0 != j0:
            UF.union(i0, j0)
            nbcomp -= 1
            if nbcomp == 1:
                return True
    return False
```

---------- ✂

2. Quelle est la complexité de l'appel au constructeur de `UnionFind` ?

✂ ----------

L'initialisation est en $O(|V|)$.

---------- ✂

3. En supposant que les appels aux méthodes `union` et `find` se font en temps constant (ce qui est vrai pour `union` et presque vrai en pratique pour `find`), quelle est la complexité de votre algorithme ?

✂ ----------

On a deux appels à `find` par arête et au plus un appel `union`. Donc la complexité est $O(|V|+|E|)$.

---------- ✂

On considère une liste l de nombres stockés sous la forme de chaînes de caractères par leur écriture en base 10. Deux nombres sont **amis directs**, s'il est possible de passer de l'un à l'autre en **échangeant deux chiffres consécutifs qui diffèrent d'au moins deux**. Par exemple :

- les deux nombres 14233 et 12433 sont amis directs ;
- les deux nombres 12433 et 12343 ne sont pas amis directs car on a échangé 3 et 4 qui diffèrent seulement de 1 ;
- les deux nombres 1233 et 3211 ne sont pas amis directs car on a échangé 1 et 3 qui ne sont pas consécutifs ;
- le nombre 1123 n'a pas d'amis.

On applique de plus la règle «les amis de mes amis sont mes amis». Ainsi le nombre 5123 et 1235 sont amis car il sont reliés par la chaîne d'amis $5123 \leftrightarrow 1523 \leftrightarrow 1253 \leftrightarrow 1235$.

Étant donné une liste L de nombres distincts, on cherche un sous ensemble S des nombres de la liste qui vérifie la condition suivante : chaque nombre de L est ami avec exactement un et un seul nombre de S.

4. Expliquer brièvement comment le problème se traduit en un problème de graphe.

✂ ----------

La relation amis directs est un graphe. On cherche donc a sélectionner un élément par composante connexe.

---------- ✂

5. Écrire un algorithme basé sur `UnionFind` qui calcule un tel ensemble S.

✂ ----------

```
def amis_directs(st):
    for i in range(len(st)-1):
        if abs(int(st[i]) - int(st[i+1])) >= 2:
            yield st[:i]+st[i+1]+st[i]+st[i+2:]

def un_ami(L):
    UF = UnionFind(L)
    for i in L:
        for a in amis_directs(i):
            if a in L:
                i0 = UF.find(i)
                a0 = UF.find(a)
```

```
                if i0 != a0:
                    UF.union(i0, a0)
    res = []
    for i in L:
        if UF.find(i) == i:
            res.append(i)
    return res
```

▸ **Exercice 2.**[Le jeu des sept différences]

On considère les deux fonctions suivantes :

```
def f(data, x):
    r    = []
    v    = {x}
    pile = [x]
    while pile:
        x = pile.pop()
        r.append(x)
        for y in data[x]:
            if y not in v:
                v.add(y)
                pile.append(y)
    return r
```

```
def g(data, x):
    r    = []
    v    = {x}
    file = [x]
    while file:
        x = file[0]; file = file[1:]
        r.append(x)
        for y in data[x]:
            if y not in v:
                v.add(y)
                file.append(y)
    return r
```

1. Comparer le code des deux fonctions, et souligner les différences sur le sujet.
2. On suppose que d est défini comme suit :

```
d = {'V': ['␣'], 'I': ['N', '␣'], 'U': [], 'E': ['␣'],
     '␣': ['U'], 'B': ['I', 'E'], 'N': ['E', 'V']}
```

Exécuter pas-à-pas les appels de fonction f(d,'B') et g(d,'B') et en donner le résultat.

On obtient respectivement :

```
['B', 'E', '␣', 'U', 'I', 'N', 'V']
```

```
['B', 'I', 'E', 'N', '␣', 'V', 'U']
```

3. Que représente d ? Faire un dessin.

d représente un graphe.

4. Quel algorithmes implantent respectivement f et g ?

- f implante un parcours en profondeur
- g implante un parcours en largeur

5. Quels noms suggériez vous pour les fonctions et variables f,g,r,v,x,y ?

- f : parcours_profondeur
- g : parcours_largeur
- r : parcours
- v : sommets_vus
- pile / **file** : sommets_a_traiter
- x, y : x, y

6. Lequel de ces deux algorithmes faudrait-il utiliser pour calculer la distance entre B et V ?

Il faudrait utiliser un parcours en largeur, pour lequel par construction on parcoure les sommets à distance 0, puis 1, puis 2, ...

7. Quel est, en théorie, la complexité des opérations suivantes : ajouter / enlever un élément à une pile ? À une file ?

Ces opérations sont en temps constant.

8. Que deviennent ces complexités lorsque l'on émule respectivement ces structures de données en utilisant une liste Python ?

✂ ----------

– Enlever un élément au début (comme dans une file) ou à la fin (comme dans une pile) d'une liste est en temps constant (mais ici on utilise **file** [1:] qui fait une copie et est donc de complexité linéaire en la longueur de la file).
Ajouter un élément à la fin d'une liste (comme dans une pile ou une file) est en temps constant amorti.

---------- ✂

9. En faisant l'hypothèse d'une pile ou file de complexité théorique, et en supposant que le test d'appartenance à un ensemble est en temps constant, donner la complexité de f et de g.

✂ ----------

Ces deux algorithmes traversent chaque sommet et chaque arête exactement une fois. Les opérations effectuées dans chaque cas sont en temps constant. La complexité globale est donc en $O(n+m)$ où n est le nombre de sommets et m le nombre d'arêtes.

---------- ✂

10. On considère la fonction h suivante qui est une version simplifiée de la fonction g :

```
def h(data, x):
    v    = [x]
    file = [x]
    while file:
        x = file[0]; file = file[1:]
        for y in data[x]:
            if y not in v:
                v.append(y)
                file.append(y)
```

Est-ce que cette simplification a un impact sur le résultat de la fonction ? Sur sa complexité ?

✂ ----------

Le résultat ne change pas. La seule différence est le test d'appartenance qui se fait avec une liste et non un ensemble, et qui a donc un coût linéaire en la taille de la liste ; celle-ci étant de taille bornée par n, cela donne une complexité totale en $O(n+nm)$.

---------- ✂

▸ Exercice 3. (Algorithme de Kruskal)

Appliquer l'algorithme de Kruskal, pour calculer l'arbre couvrant de poids **maximum** du graphe donnée en annexe.

▸ **Exercice 4. (Ford-Fulkerson)**

1. Pour tous les graphes proposés dans l'annexe "est-ce des flots", répondez aux questions suivantes (directement sur la feuille) sachant que **le premier nombre** sur les arêtes indique la capacité de l'arête dans le réseau et le **second nombre**, en gras, indique le flot sur l'arête :
   (a) Est-ce bien un flot ? Si ce n'est pas le cas, justifiez-le sur le graphe.
   (b) Si oui, quelle est sa valeur ?
   (c) Est-ce un flot maximal ? Si ce n'est pas le cas, justifiez-le sur le graphe.
   (d) Si le flot est maximal, **indiquez sur le graphe la coupe minimale correspondante**
2. Complétez l'algorithme de Ford Fulkerson sur le réseau donné dans l'annexe "appliquer Ford Fulkerson" dont la première étape est donnée (Remarque : le nombre de réseaux dessinés ne correspond pas forcément au nombre d'étapes de l'algorithme).
3. Une compagnie de production de panneaux solaires cherche à organiser ses livraisons. Elle possède 3 usines de production (New York, Buenos Aires et Le Cap) et doit livrer dans 8 villes dans le monde. Ces villes sont reliées par des liaisons aériennes avec une certaine capacité de transport. Utilisez les données ci-dessous pour **modéliser le problème** : on ne vous demande pas de résoudre à la main le problème, simplement de donner le **réseau** avec un **point source** et un **point cible** tel que le problème puisse être résolu par l'algorithme de Ford-Fulkerson.

Usine	**Production**
New York	600
Buenos Aires	1500
Le Cap	1500

Lieu livraison	**Demande**
San Francisco	500
New York	200
Buenos Aires	300
Le Cap	200
Copenhague	200
Paris	100
Casablanca	200
Tokyo	150

Liaison		**Capacité**
San Francisco	Tokyo	300
San Francisco	New York	100
San Francisco	Paris	200
San Francisco	Buenos Aires	200
New York	Copenhague	200
New York	Paris	200
Buenos Aires	Le Cap	200
Le Cap	Casablanca	300
Copenhague	Pairs	200
Paris	Tokyo	300
Paris	Casablanca	100

(Les avions volent dans les deux sens)

1.

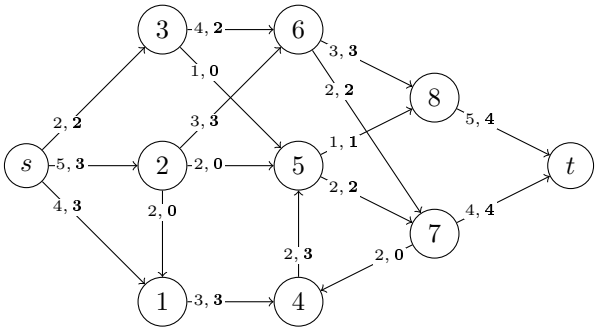
Flot ? non (capacité 4 ->5)	Flot ? oui Valeur : 6 Maximal ? non Chaîne augmentante s-2-5-8-t
Flot ? oui Valeur : 8 Maximal : oui Coupe : s,1,2,4,5	Flot ? non (sommet 4)
Flot ? oui Valeur : 7 Maximal : non Chaîne augmentante s-2-5-3-6-8-t	Flot ? oui Valeur : 8 Maximal : oui Coupe : s,1,2,4,5

2. (a) Chaîne $s \to 2 \to 1 \to t$ (potentiel 2)
   (b) Chaîne $s \to 2 \to t$ (potentiel 1)
   (c) Chaîne $s \to 1 \to t$ (potentiel 1)
3. Créer un sommet source s avec une arête vers chaque usine dont la capacité est la production de l'usine. Ajouter les arêtes entre les villes avec la capacité des avions. Créer un sommet cible t et une arête depuis chaque ville dont la capacité est la demande de la ville.

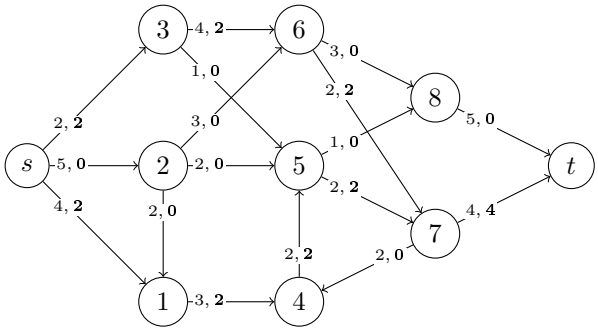
Numéro de copie :

Annexe exercice 4 : "est-ce des flots"

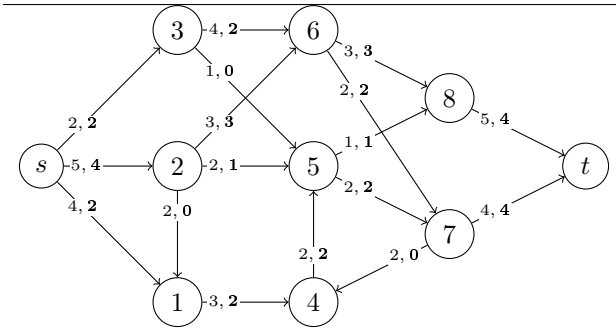
Justifiez vos réponses sur les graphes.



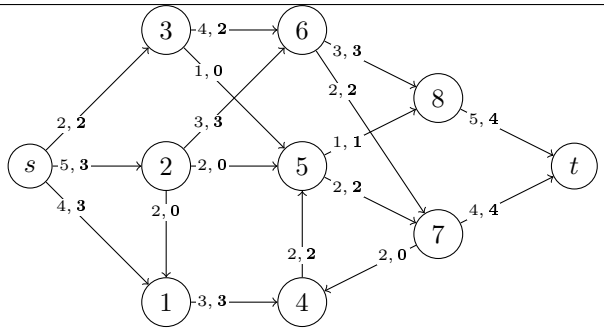
Flot ? oui non
(si oui) Valeur :
(si oui) Maximal ? oui non



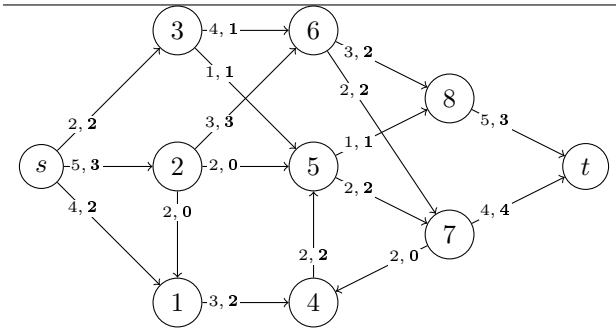
Flot ? oui non
(si oui) Valeur :
(si oui) Maximal ? oui non



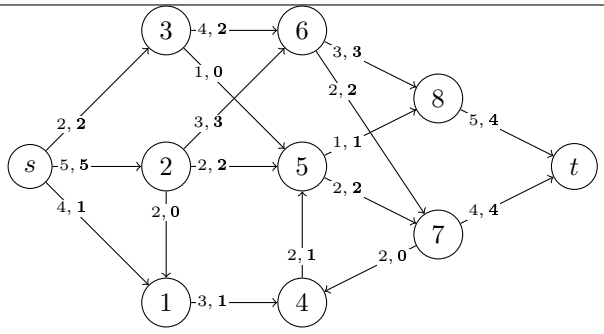
Flot ? oui non
(si oui) Valeur :
(si oui) Maximal ? oui non



Flot ? oui non
(si oui) Valeur :
(si oui) Maximal ? oui non

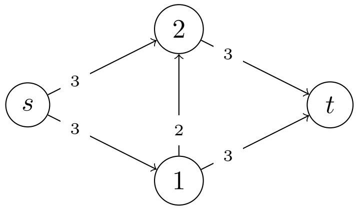


Flot ? oui non
(si oui) Valeur :
(si oui) Maximal ? oui non

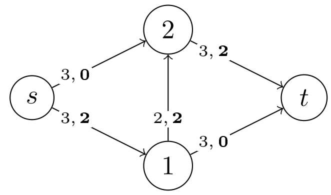


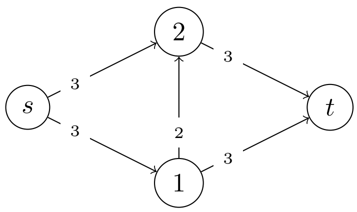
Flot ? oui non
(si oui) Valeur :
(si oui) Maximal ? oui non

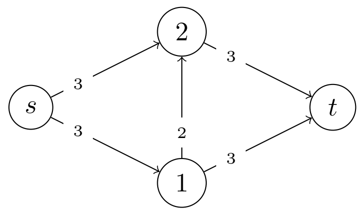
Annexe exercice 4 : "appliquer Ford Fulkerson"

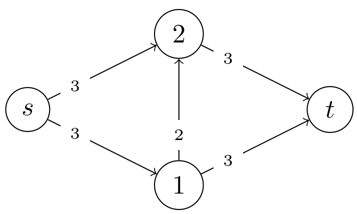


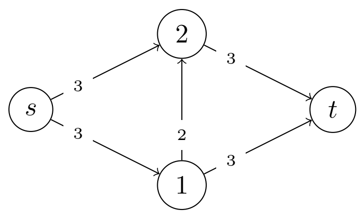
Chaîne augmentante : $s \rightarrow 1 \rightarrow 2 \rightarrow t$

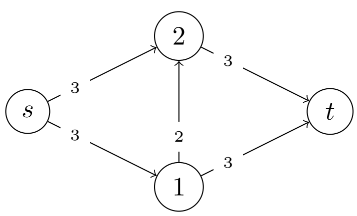


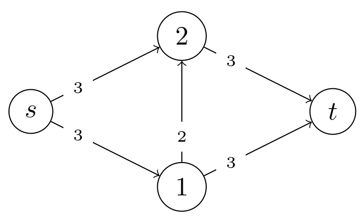




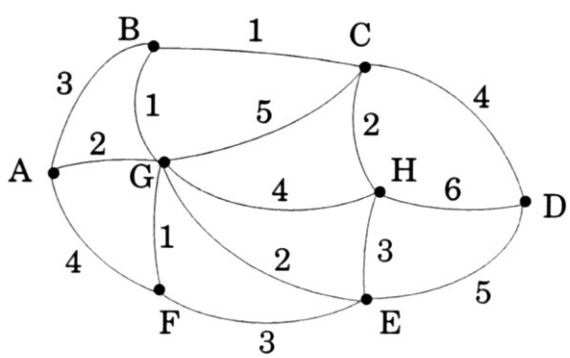








Annexe exercice 3 : Algorithme de Kruskal



Poids maximum =