Introduction à l'algorithmique **Hiver 2018**

IFT2125-6001 TA: Stéphanie Larocque

Démonstration 5

À partir des corrigés de Maelle Zimmermann

1

Question: En théorie des graphes, un isomorphisme de graphes G et H est une bijection f entre les sommets de G et de H

$$f : V(G) \to V(H)$$

tels que deux somments u et v sont adjascentes dans G si et seulement si $f(u)$ et $f(v)$ sont adjascents dans H. Donner un algorithme qui génère aléatoirement deux graphes et les compare par nombres d'arêtes, nombre de sommets, et séquence de degrés. Cela suffit-il à vérifier si les deux graphes sont isomorphes?

Solution: Voici les algorithmes:

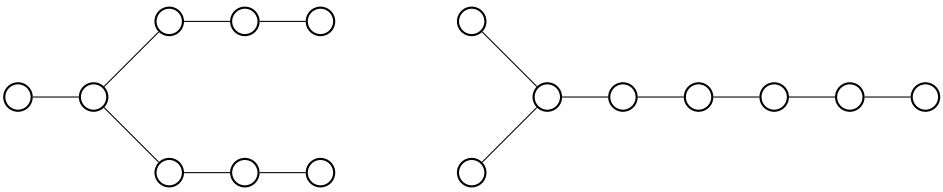
```
import random

# genere une matrice d'incidences de graphe aleatoire
def random_graph(n):
   g = [[random.randint(0, 1) for i in range(n)] for j in range(n)]
   for i in range(n):
      g[i][i] = 0
   for i in range(n):
      for j in range(i,n):
         g[j][i] = g[i][j]
   return g

# compare deux graphes
def graph_iso(g1,g2):
   if len(g1) != len(g2):
      return False
   if sum(sum(x) for x in g1) != sum(sum(x) for x in g2):
      return False
   seq_deg_g1 = [sum(x) for x in g1]
   seq_deg_g2 = [sum(x) for x in g2]
```

```
if sorted(seq_deg_g1) != sorted(seq_deg_g2):
   return False
return True
```

Cet algorithme ne suffit pas à vérifier si deux graphes sont isomorphes, même s'il permet de détecter certains cas où ils ne le sont pas. Voici un contre-exemple:



Ces deux graphes ont le même nombre d'arêtes, de sommets et la même séquence de degrés $[3, 2, 2, 2, 2, 1, 1, 1]$ mais ne sont pas isomorphes.

Par contre si on compte pour chaque sommet les degrés de ses voisins, on peut voir que les graphes ne sont pas identiques. En effet, deux des sommets à degré 1 du premier graphe ont un voisin de degré 2, tandis que dans le deuxième graphe, seul un sommet à degré 1 a un voisin de degré 2. C'est l'idée derrière l'algorithme Weisfeiler-Lehman qui calcule pour chaque sommet les degrés des sommets à distance 1, puis à distance 2, etc. Bien que cet algorithme marche mieux, il échoue tout de même sur certains types de graphes. En réalité, il n'y a pas d'algorithme en temps polynomial connu pour ce problème.

2

Question: Donner une implémentation efficace d'ensembles disjoints et analyser la complexité en temps. Nous sommes intéressés à implémenter et analyser les opérations suivantes:

- `find(x)`: retourne l'étiquette de l'ensemble qui contient x,
- `merge(a,b)`: fusionne les ensemble étiquetés par a et b.

Solution: Supposons que nous avons k objets numérotés de 1 à k et que nous voulons les regrouper dans des ensembles disjoints, c'est-à-dire qu'à tout moment, chaque objet est dans exactement un ensemble. Par exemple, si $k = 10$, les objets peuvent être partitionnés ainsi.

$$\{1, 3, 7\}, \{2, 5, 6, 10\}, \{4, 9\}, \{8\}.$$

Chaque ensemble est associé à une étiquette. Par exemple on peut choisir par convention de dénoter chaque ensemble par son plus petit élément, auquel cas $\{2, 5, 6, 10\}$ est étiqueté par 2.

Par exemple, ci-dessus `find(6)` retourne 2, et après `merge(1,4)` on obtient la partition

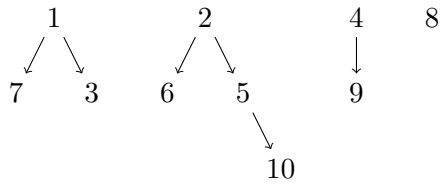
$$\{1, 3, 4, 7, 9\}, \{2, 5, 6, 10\}, \{8\}.$$

Nous voulons représenter ce problème efficacement sur un ordinateur.

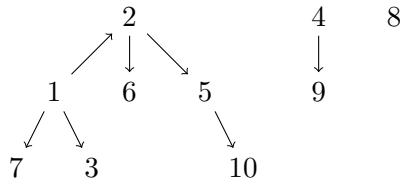
On pourrait considérer une implémentation où chaque élément est associé à l'étiquette de son ensemble, par exemple ici la partition initiale serait représentée par la liste $S = [1, 2, 1, 4, 2, 2, 1, 8, 4, 2]$. Dans ce cas là l'opération `find(x)` prendrait un temps constant, mais l'opération `merge(a,b)` ne serait pas efficace et prendrait un temps dans $\Theta(k)$ en pire cas. Par exemple en fusionnant deux ensemble de taille $k/2$, il faudrait changer $k/2$ étiquettes.

Considérons plutôt une implémentation où les ensembles sont représentés par des arborescences et définissons que l'étiquette d'un ensemble est le nombre à sa racine (pas nécessairement le plus petit nombre). En mémoire, chaque élément est associé à son parent à l'exception des racines qui sont associées à elles mêmes. De plus, nous stockerons également, pour chaque x, la hauteur de l'arborescence de la racine de x. La fusion de deux arborescences se fait en fusionnant la plus petite à la plus grande. Les ensembles disjoints seraient donc représentés par $D = (S, H)$, où $S[i]$ redonne le parent de l'élément i, et $H[racine]$ redonne la hauteur de l'arbre dans lequel se trouve l'élément "racine".

Par exemple, la partition ci-dessus pourrait être représentée par l'arborescence suivante ce qui donnerait la liste $S = [1, 2, 1, 4, 2, 2, 1, 8, 4, 5]$:



Après l'opération `merge(1,2)`, on obtiendrait $S = [2, 2, 1, 4, 2, 2, 1, 8, 4, 5]$:



Ainsi les ensembles disjoints seraient représentés par un tuple $D = (S, H)$ où H est la

liste qui contient les hauteurs des arborescences. Voici une implémentation python des opérations `find(x)` et `merge(a,b)` selon cette idée:

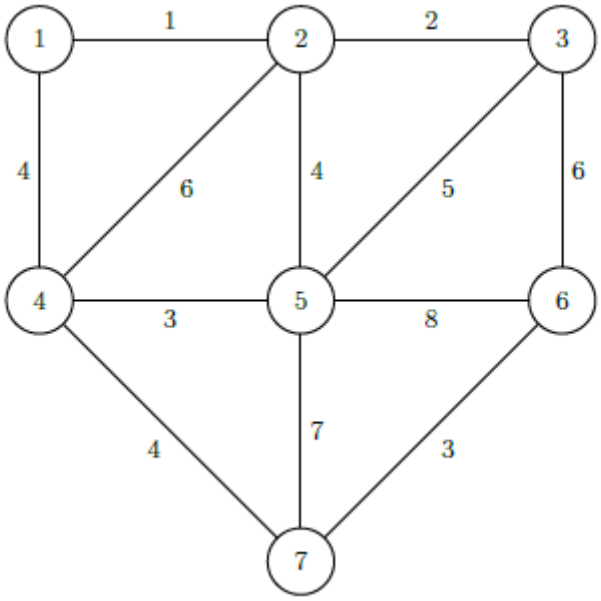
```
def find(D, x):
    S, _ = D
    r = x
    while S[r] != r:
        r = S[r]
    return r

def merge(D, a, b):
    S, H = D
    if H[a] == H[b]:
        H[a] = H[a] + 1
        S[b] = a
    elif H[a] > H[b]:
        S[b] = a
    else:
        S[a] = b
```

On peut montrer que dans ce cas la complexité de find est dans $\Theta(1)$ dans le meilleur cas (si x est une racine lui-même) et dans $\Theta(\log k)$ dans le pire cas (si x est une feuille d'un arbre), car la hauteur des arborescences ne dépasse jamais $\log k$ en prenant soin de fusionner la plus petite à la plus haute à chaque fusion. La complexité de merge est dans $\Theta(1)$ dans le meilleur et pire cas. Pour plus de détails, lire la section 5.9 de Brassard et Bratley (pp. 175-180).

3

Question: Appliquer l'algorithme de Kruskal pour trouver un arbre sous-tendant de poids minimal sur le graphe suivant, où le poids est indiqué à côté de chaque arête.



Solution: L'algorithme de Kruskal permet de trouver un arbre sous-tendant de poids minimal d'un graphe connexe. L'algorithme initialise un ensemble T vide et parcourt les arêtes (u, v) par ordre croissant de poids. Si les somments u et v reliés par l'arête sont dans deux composantes connexes différentes, l'arête est ajoutée à T. En exécutant l'algorithme, nous obtenons:

Itération	(u, v)	Ensemble d'arêtes T	Composantes connexes D
0	-	-	$\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}$
1	$(1, 2)$	$\{(1, 2)\}$	$\{1, 2\}\{3\}\{4\}\{5\}\{6\}\{7\}$
2	$(2, 3)$	$\{(1, 2), (2, 3)\}$	$\{1, 2, 3\}\{4\}\{5\}\{6\}\{7\}$
3	$(4, 5)$	$\{(1, 2), (2, 3), (4, 5)\}$	$\{1, 2, 3\}\{4, 5\}\{6\}\{7\}$
4	$(7, 6)$	$\{(1, 2), (2, 3), (4, 5), (7, 6)\}$	$\{1, 2, 3\}\{4, 5\}\{6, 7\}$
5	$(1, 4)$	$\{(1, 2), (2, 3), (4, 5), (7, 6), (1, 4)\}$	$\{1, 2, 3, 4, 5\}\{6, 7\}$
6	$(2, 5)$	$\{(1, 2), (2, 3), (4, 5), (7, 6), (1, 4)\}$	$\{1, 2, 3, 4, 5\}\{6, 7\}$
7	$(4, 7)$	$\{(1, 2), (2, 3), (4, 5), (7, 6), (1, 4), (4, 7)\}$	$\{1, 2, 3, 4, 5, 6, 7\}$
$\vdots$			
12	$(5, 6)$	$\{(1, 2), (2, 3), (4, 5), (7, 6), (1, 4), (4, 7)\}$	$\{1, 2, 3, 4, 5, 6, 7\}$

Ainsi l'arbre sous-tendant de poids minimal contient les arêtes dans T et est de poids 17.

4

Question: Implémenter l'algorithme de Kruskal et analyser son temps d'exécution dans le pire cas.

Solution: Voici une implémentation python basée sur l'implémentation d'ensembles disjoints de l'exercice précédent:

```
def Kruskal(V, E):
T=[]
D = (range(k), [0] * k) # initie ensemble disjoints (composantes
    connexes)
def poids((u, v, c)):
   return c

# boucle sur les aretes triees en ordre croissant
for (u, v, c) in sorted(E, key = poids): #Trier en ordre des poids
   i = find(D, u) # trouve composante connexe de u
   j = find(D, v) # trouve composante connexe de v
   if i != j: # si u et v ne sont pas dans la meme composante connexe
      merge(D, i, j) # fusionner les ensembles
      T.append((u, v)) # ajouter l'arete
return T
```

Posons $n = |V|$ et $a = |E|$, alors:

- L'initialisation des ensembles disjoints de sommets se fait en temps $\Theta(n)$,
- Le tri de a arêtes se fait en temps $\Theta(a \log a)$,
- Il y a a tours de boucle for, donc $2a$ appels à find,
- Après $n-1$ fusions il ne reste qu'un seul ensemble, donc il y a $n-1$ appels à merge.

Comme chaque appel à find prend un temps dans $\Theta(\log n)$, et chaque appel à merge prend un temps dans $\Theta(1)$, le temps total d'exécution de l'algorithme de Kruskal est dans $\Theta(n + a \log a + 2a \log n + n - 1) = \Theta(\max\{n, a \log a, 2a \log n, n-1\})$. Or comme le graphe considéré est connexe, nous avons que

- $n - 1 \leq a$ car il faut au moins $n-1$ arêtes pour connecter tous les sommets,
- $a \leq n(n-1)/2$ car c'est le nombre maximal d'arêtes que peut comporter un graphe à n sommets.

Ainsi le temps d'exécution de l'algorithme est dans $\Theta(\max\{a \log a, a \log n\}) = \Theta(a \log n)$ car $\log a \leq \log n^2 = 2 \log n$.

5

Question: k-clustering : Implémenter une variante de l'algorithme de Kruskal tel que, pour un k donné, l'algorithme retourne une partition des n sommets du graphe en k ensembles (disjoints) de manière à maximiser la distance entre toute paire de points provenant d'ensembles différents.

Solution: Si on veut obtenir une partition des n sommets du graphe en k ensembles disjoints de poids minimal, alors on peut utiliser l'algorithme de Kruskal, avec une condition d'arrêt différente : on arrête lorsque

$$|T| = n - k,$$

car le graphe trouvé à ce moment sera composé d'exactement k composantes connexes et de poids minimal (ou, de la même façon, aura maximisé la distance entre toute paire de points provenant d'ensembles différents).

On peut remarquer que lorsque $k = 1$, alors on retrouve l'algorithme de Kruskal (voir fichier de code joint).

6

Question: Un serveur a n clients à servir et ne peut en servir qu'un à la fois. Le temps de service requis par chaque client est connu à l'avance: le client i prend un temps t_i. Nous cherchons à minimiser le temps moyen d'attente des clients dans le système. Donner un algorithme pour ce problème et montrer qu'il fonctionne.

Solution: Cela est équivalent à minimiser

$$T = \sum_{i=1}^{n} \text{temps dans le système pour client } i$$

Idée: Servir les clients par ordre croissant de temps de service, c'est-à-dire d'abord ceux qui requièrent le moins de temps.

Si on formule cette idée sous forme d'algorithme, cela correspond à formuler un algorithme vorace qui agende les clients l'un après l'autre sans jamais revenir sur ses choix

précédents. A chaque étape, l'algorithme choisit simplement le client suivant comme ayant le plus petit temps de service requis parmi les clients restants. L'algorithme se termine lorsque tous les clients sont agendés, c'est-à-dire lorsqu'on a obtenu une permutation. Par exemple on peut formuler l'algorithme ainsi:

```
def ord(t):
    def temps(i): return t[i]
    clients = range(len(t))
    return sorted(clients, key = temps)
```

Montrons que cet algorithme donne la solution optimale. Soit $P = p_1 p_2 ... p_n$ un ordonnancement optimal des n clients.

Posons $s_i = t_1 + t_2 + \cdots + t_i$ le temps d'attente total du $i^{\text{ème}}$ client (incluant le temps d'attente pour les clients le précédent, ainsi que son propre temps au comptoir) servi selon l'ordre défini par P. Nous voulons donc minimiser le temps total passé dans le système par l'ensemble des clients (et pas uniquement le temps total de service requis qui, lui, restara toujours le même : $t_1 + t_2 + \cdots + t_n$). Si les clients sont servi dans l'ordre défini par P, alors le temps total requis par l'ensemble des clients est de

$$\begin{aligned} T(P) &= s_1 + s_2 + ... + s_n \\ &= (t_1) + (t_1 + t_2) + (t_1 + t_2 + t_3) + \ldots (t_1 + t_2 + \cdots + t_n) \\ &= nt_1 + (n-1)t_2 + (n-2)t_3 + ... + t_n \\ &= \sum_{i=1}^{n} (n - i + 1)t_i \end{aligned}$$

Supposons par l'absurde que P (une solution optimale) n'ordonne pas les clients en ordre croissant de temps de traitement. Alors il existe

$$a < b \text{ tel que } t_a > t_b$$

(il y a au moins une paire de clients mal classés - sinon l'ordonnancement serait en ordre croissant- , i.e. 2 clients a et b tels que a vient avant b mais prend un temps supérieur de service). Soit P' l'ordonnancement où ces clients sont interchangés dans P. Le temps total requis selon l'ordre défini par P' est:

$$T(P') = (n - a + 1)t_b + (n - b + 1)t_a + \sum_{i=1, i \neq a,b}^{n} (n - i + 1)t_i.$$

(puisque les autres clients $i \neq a, b$ restent au même endroit, alors leur temps de service affectera encore le même nombre de personnes ($n - i + 1$ personnes), tandis que le temps de service t_a affectera maintenant $n - b + 1$ personnes, car le client a se retrouvera à la

position b Ainsi,

$$\begin{aligned} T(P) - T(P') &= (n-a+1)t_a + (n-b+1)t_b - (n-a+1)t_b - (n-b+1)t_a \\ &= (n-a+1)(t_a - t_b) + (n-b+1)(t_b - t_a) \\ &= (n-a+1)(t_a - t_b) - (n-b+1)(t_a - t_b) \\ &= (b-a)(t_a - t_b) > 0, \end{aligned}$$

où l'inégalité provient du fait que $a < b$ mais $t_a > t_b$. Donc le temps total en utilisant l'ordonnancement P' est plus petit que celui obtenu en utilisant l'ordonnancement P. Cela contredit le fait que P est l'ordonnancement optimal. Ainsi si on suppose que P n'ordonne pas les clients en ordre croissant, on aboutit à une contradiction. Donc, l'ordonnancement optimal P ordonne bien les clients par ordre croissant de temps.