

DS PCE : union-find et Kruskal

Démonstration.

□

Présentation

Ce devoir est décomposé en 2 parties principales et une dernière partie applicative. Dans la première nous présentons une structure de donnée, appelée *union-find*, pour le problème des *classes disjointes*. Dans la seconde partie, nous utilisons cette structure de donnée pour résoudre le problème de l'*arbre couvrant de poids minimal* dans un graphe non orienté sans boucle (pas d'arête $x - x$) connexe¹ et pondéré.

Pour les listes, les méthodes `append` et `pop` sans argument sont autorisées mais aucune autre méthode de liste. L'appel `x=l.pop()` retire le dernier élément de `l` et le met dans `x` (`l` est donc modifié).

On n'autorise pas l'application des fonctions `max,min,sum` aux listes, dictionnaires et tuples.

La concaténation de deux listes `l1+l2` ainsi que le slicing sont autorisés. On peut aussi utiliser la construction de listes ou dictionnaires par compréhension. On rappelle que La commande `x in S` renvoie un booléen indiquant si `x` est dans la structure `S`. La complexité est linéaire en $|S|$ si `S` est une liste et de coût amorti constant si `S` est un dictionnaire.

Enfin, pour parcourir toutes les clés d'un dictionnaire ou tous les éléments d'une liste, on peut utiliser `for k in S` si `S` est un dictionnaire ou une liste.

On utilise la convention classique qui consiste à noter x un objet mathématique et `x` le même objet dans un contexte de code Python.

1 Structure union-find

Étant donné une relation d'équivalence d'un ensemble fini E connue par l'ensemble \mathcal{C} de ses classes d'équivalences. On souhaite faire deux actions :

- Déterminer si deux éléments sont en relation (d'équivalence). Cela revient à vérifier si ils sont dans la même classe ;
- Fusionner deux classes d'équivalence (et donc modifier la relation d'équivalence). Cela revient à mettre en relation des éléments qui n'y étaient pas.

L'objectif est d'obtenir une complexité minimale pour ces deux opérations. on essaye plusieurs méthodes dans ce but.

On note E l'ensemble sur lequel on travaille et on convient que $E = \llbracket 0, n-1 \rrbracket$ pour un $n \in \mathbb{N}^*$.

1.1 Solution naïve

Dans cette partie, l'ensemble \mathcal{C} des classes d'équivalences est représenté par une liste de listes `C`. Par exemple :

```
1 C = [[0,3],[1,5],[2,4]]
```

Q.1 Outils pour les dictionnaires :

- Écrire la fonction `some_element(d)` qui renvoie UNE clé du dictionnaire `d`, n'importe laquelle. Cette fonction doit avoir une complexité $O(1)$. On suppose `d` non vide.
- Écrire la fonction `maxd(d)` qui renvoie la plus grande clé du dictionnaire. Cette fonction doit avoir une complexité linéaire en le nombre de clefs. On suppose `d` non vide.

1. Il existe un chemin entre tout couple de sommets.

- (c) Écrire la fonction `check(C)` qui prend en paramètres une liste de listes d'entiers et renvoie le booléen `True` si `C` représente une partition d'un intervalle d'entiers de borne inférieure 0. La difficulté est bien sûr qu'on ne connaît pas cet intervalle d'entier, c'est l'exploration de `C` qui permet de le déterminer.

La complexité doit être linéaire en la somme des longueurs de listes de `C`.

```
1 d = {1:6,3:2,15:-1,12:13}
2 some_element(d), maxd(d), check([[0,3],[1,5],[2,4]]),\
3 check([[0,3],[1,5],[6,4]])
```

```
(1, 15, True, False)
```

On suppose pour toute la suite que `C` représente bien une partition de E sans qu'il soit demandé de le vérifier. Les classes d'équivalences elle-mêmes sont supposées sans doublon.

On note N le nombre de classes d'équivalences, m le cardinal de la plus grande classe d'équivalence et n le cardinal de E .

- Q.2** Écrire la fonction `find_classe(x,C)` qui renvoie un tuple formé de la classe d'équivalence de `x` et de son indice ou `[]` et `-1` si $x \notin E$.

```
1 C = [[0,3],[1,5],[2,4]]
2 print(find_classe(0,C), print(find_classe(6,C))
```

```
([0, 3], 0)
([], -1)
```

Cette fonction doit avoir une complexité $O(n)$.

- Q.3** Écrire la fonction `en_relation(x,y,C)` qui renvoie un booléen indiquant si x et y (supposés être deux éléments de E) sont en relation d'équivalence.

```
1 C = [[0,3],[1,5],[2,4]]
2 en_relation(2,4,C), en_relation(0,1,C)
```

```
(True, False)
```

Cette fonction doit avoir une complexité $O(n + m)$.

Dans la question qui suit, on suppose que les classes d'équivalences sont données sous la forme de listes triées sans doublon.

- Q.4** (a) Écrire la fonction `fusion(l1,l2)` qui prend en paramètres deux listes triées et sans doublon et renvoie la classe fusionnée (donc une liste triée et sans doublon).

La complexité doit être linéaire en la somme des longueurs des deux listes paramètres.

```
1 l1 = [10,15,20,30]
2 l2 = [5,17,19,20,26,28,30]
3 fusion(l1,l2)
```

```
[5, 10, 15, 17, 19, 20, 26, 28, 30]
```

Il s'agit de la même fonction que celle utilisée pour le tri fusion.

- (b) Écrire la fonction `union(x,y,C)` qui fusionne les deux classes de x et y . Elle retourne une nouvelle liste qui est `C` privée des classes de x et y mais avec une nouvelle classe résultant de la fusion des classes supprimées. On ne cherche pas à faire un code trop élaboré et on se contente d'utiliser les fonctions précédentes.

```
1 C = [[0,3,7],[1,5,6],[2,4]]
2 union(3,5,C)
```

```
[[2, 4], [0, 1, 3, 5, 6, 7]]
```

1.2 La structure de données

L'ensemble des classes d'équivalences \mathcal{C} de E est maintenant représenté comme une liste `C` de n éléments. La valeur `C[i]` indique un unique *représentant* de la classe de i , c'est à dire un élément de E qui est dans la même classe que i .

Si `C[i]==i`, nous disons dans ce devoir que i est un *représentant ultime* de sa classe d'équivalence.

L'idée principale est de relier entre eux les éléments d'une même classe. Dans chaque classe, ces liaisons forment des chemins qui mènent tous à un unique représentant, lequel est le seul relié à lui-même.

La figure 1 montre un exemple où l'ensemble $\{0, \dots, 7\}$ est partitionné en deux classes dont les représentants ultimes sont respectivement 3 et 4.

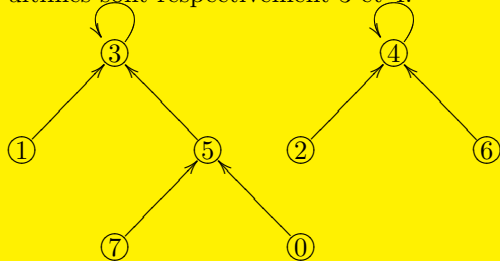


FIGURE 1 – Une partition en deux classes de $\{0, \dots, 7\}$

Cette situation est représentée en Python par

```
C = [5, 3, 4, 3, 4, 3, 4, 5]
```

On appelle *partition triviale d'un ensemble*, une partition dans laquelle chaque élément est en relation avec lui-même uniquement. Les classes d'équivalences sont donc des singletons.

Q.5 Écrire la fonction `create(n)` qui construit une partition triviale de $\llbracket 0, n-1 \rrbracket$. La complexité doit être $O(n)$.

```
1 create(7)
```

```
[0, 1, 2, 3, 4, 5, 6]
```

Q.6 Écrire la fonction `find(i,C)` qui donne le représentant ultime de la classe de i .

```
1 C = [5, 3, 4, 3, 4, 3, 4, 5, 7]
2 find(3, C), find(7, C)
```

```
(3, 3)
```

Q.7 Pour faire l'union des classes de x et y , on commence par trouver les représentants ultimes des deux éléments puis on lie l'un des deux représentants à l'autre de sorte que le plus grand reste ultime. En particulier, si x, y sont dans la même classe, alors `union` est sans effet.

```
1 C = [5, 3, 4, 3, 4, 3, 4, 5]
2 union(7, 6, C)
3 C
```

```
[5, 3, 4, 4, 3, 3, 4, 5]
```

On observe sur cet exemple que 3 n'est désormais plus ultime et pointe sur 4. La figure 2 représente ce qu'il s'est passé :

Malheureusement, cette méthode n'est pas encore optimale. On peut en effet se retrouver avec une chaîne qui contient tous les éléments de la classe comme dans la situation de la figure 3.

La complexité pour trouver avec `find` le représentant ultime de l'élément en bas de la chaîne est alors en $O(n)$ ce que nous allons améliorer dans la sous-section suivante.

Q.8 Écrire une fonction `chaîne(n)` qui renvoie une liste chaînée en effectuant uniquement la fonction de création de partition et une succession d'union de classes. L'appel `chaîne(4)` renvoie la liste `[1, 2, 3, 3]` qui représente la situation exposée en figure 3.

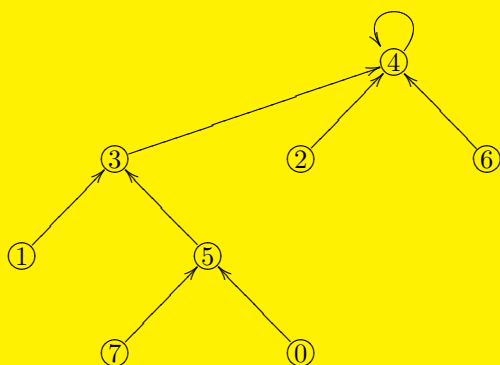


FIGURE 2 – La partition obtenue après union des classes de 3 et 4. La classe de 4 a « absorbé » celle de 3

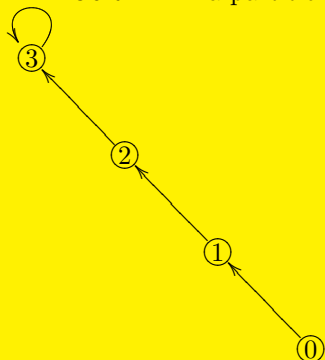


FIGURE 3 – Une chaîne de 0 à 3 de longueur 3

1.3 Une première amélioration

Rappelons que, dans un graphe, la *longueur d'un chemin* est le nombre d'arêtes qui le constituent.

Nous allons améliorer nos primitives en faisant en sorte que la longueur d'un chemin depuis un sommet vers son représentant ultime soit négligeable devant le cardinal de sa classe d'équivalence. En plus du tableau représentant la partition, nous maintenons un tableau dit *des rangs* qui associe à chaque représentant ultime la longueur maximale que pourrait avoir un chemin dans cette classe.

La partition \mathcal{C} est représentée par une structure appelée *union-find* que l'on implante par un dictionnaire \mathcal{C} à deux clés :

- `"link"` qui représente la partition proprement dite (comme à la sous-section précédente)
- `"rank"` qui représente le tableau des rangs.

Par convention, on décide que seule l'information du rang pour un représentant ultime est pertinente. On peut donc laisser n'importe quoi dans $\mathcal{C}["rank"][j]$ si j n'est pas un représentant ultime de sa classe.

Voici un exemple de dictionnaire qui modélise la situation de la figure 1.

```
d = {"link": [5, 3, 4, 3, 4, 3, 4, 5], "rank": [10, 5, 40, 2, 1, 7, 14, 12]}
```

Seules les informations des rangs de 3 et 4 sont pertinentes : elles correspondent exactement aux longueurs des plus longs chemins entre un sommet et son représentant ultime dans les arbres correspondants.

Dans ce qui suit, nous modifions les fonctions `create`, `find`, `union` de la sous-section 1.2.

Q.9 Écrire la fonction `create(n)` qui retourne le dictionnaire correspondant à la partition initiale :

```
1 create(8)
```

```
{'link': [0, 1, 2, 3, 4, 5, 6, 7], 'rank': [0, 0, 0, 0, 0, 0, 0, 0]}
```

Q.10 Modifier `find(x,C)`. Il n'y a pas beaucoup de changement à faire pour cette fonction mais il faut prendre en compte le fait que le second paramètre est un dictionnaire et pas une liste.

```
1 C = [5,3,4,3,4,3,4,5,7]
2 R = [100, 100, 100, 3, 1, 100, 100, 100]
3 d= {"link":C,"rank":R}
4 find(3,d), find(7,d)
```

(3, 3)

L'algorithme est le suivant pour l'appel `union(x,y,d)` :

- on récupère les représentants ultimes r_x et r_y de x et y ;
- si $r_x = r_y$, on ne fait rien, l'union est terminée.
- Si $r_x \neq r_y$, on compare les rangs R_x et R_y de r_x et r_y (souvenons-nous que comme r_x, r_y sont ultimes, l'indications de leur rang est pertinente).
 - si $R_x < R_y$, alors on fait de r_y le représentant de l'union.
 - Si $R_x > R_y$, on réalise l'opération symétrique de la précédente.
 - Si $R_x = R_y$, on met r_x comme représentant de l'union et on incrémente de 1 le rang de r_x .

On Justifie certains points de cet algorithme :

Q.11 Dans le cas où $R_x < R_y$, montrer que l'information sur les rangs n'a pas besoin d'être modifiée.

Démonstration. L'information sur les rangs n'a pas besoin d'être modifiée parce que :

- r_x n'est plus ultime donc l'information sur son rang n'a pas besoin d'être renseignée ;
- Seuls les chemins sommets-racine de l'ancienne classe de r_x ont vu leur longueur maximale (R_x) augmentée de 1. Mais comme $R_x < R_y$, on a encore $R_x + 1 \leq R_y$, ce qui fait qu'on ne touche pas au rang de r_y .

□

Q.12 Si $R_x = R_y$, montrer que seul le rang de r_x doit être modifié.

Démonstration. C'est le seul cas où il faut mettre à jour les rangs. En effet, le plus long chemin dans la classe de r_y joignait un élément de cette classe à r_y (qui était ultime). Ce plus long chemin, de longueur initiale R_y , doit maintenant être augmenté de 1 (pour joindre r_y à son nouveau représentant ultime r_x). Bref il faut incrémenter de 1 le rang de r_x .

Les chemins sommets-racine dans r_x ne changent pas.

□

On présente maintenant l'invariant de boucle respecté par la fonction d'union.

Pour toute classe C de rang k :

- le plus long chemin sommet-racine dans la représentation arborescente de C est de longueur k .
- C possède au moins 2^k éléments.

Cela signifie que si l'invariant est vérifié avant l'appel à `union`, alors il l'est encore après l'appel (en tenant compte qu'une classe a éventuellement vu son rang incrémenté de 1).

Q.13 Montrer que si la partition vérifie l'invariant, alors, après l'appel `union(x,y,C)`, la partition obtenue après l'appel `union(x,y,C)` le respecte encore.

Démonstration. Si x, y ont même représentant ultime alors aucune classe n'est modifiée et l'invariant reste valable.

Supposons les représentants ultimes différents. Les classes autres que celles de x, y ne sont pas modifiées donc vérifient l'invariant.

Notons C_x, C_y les classes de x, y avant l'appel à `union`.

- Si $R_x < R_y$, alors C_x possède au moins 2^{R_x} éléments, C_y possède au moins 2^{R_y} éléments.

Après union, la classe obtenue possède $2^{R_x} + 2^{R_y}$ éléments donc au moins 2^{R_y} . Le plus long chemin est toujours de longueur R_y . l'invariant est vérifié.

— Si $R_x = R_y$, alors C_x possède au moins 2^{R_x} élément, comme C_y .

Après union, la classe obtenue possède au moins $2^{R_x} + 2^{R_x}$ éléments donc au moins 2^{R_x+1} . Le plus long chemin est de longueur $R_x + 1$. Donc l'invariant est vérifié

□

Q.14 Établir que les appels `find(x,C)` et `union(x,y,C)` sont de complexité $O(\log_2(n))$.

Démonstration. La question précédente établit que la longueur d'un plus long chemin sommet-racine dans la représentation arborescente d'une classe C est majoré par $\log_2(|C|)$ (puisque $2^{\log_2|C|} = |C|$).

Comme la classe contient moins de n éléments, alors la longueur d'un plus long chemin est majoré par $\log_2 n$.

Comme la recherche du représentant ultime a une complexité proportionnelle à la longueur de la plus grande branche de la classe, sa complexité est un $O(\log_2 n)$.

L'union ne réalise que des opérations en $O(1)$ en plus des deux recherches en $O(\log_2 n)$. D'où sa complexité.

□

Q.15 Écrire la fonction `union(x,y,C)` où `C` est un dictionnaire.

Par exemple, en faisant l'union des classes de 3 et 4 dans la situation de la figure 1

```
1 C = [5,3,4,3,4,3,4,5]
2 R = [100, 100, 100, 2, 1, 100, 100]
3 d= {"link":C,"rank":R}
4 union(3,4,d)
5 d
```

```
{'link': [5,3,4,3,3,3,4,5], 'rank': [100,100,100,2,1,100,100]}
```

La figure 4 résume ce qu'il vient de se passer. On observe que le rang de la plus grosse classe n'a pas évolué. Les autres rangs ne sont de toute façon pas pertinents (ici, ils valent 100 mais ce pourrait être n'importe quoi).

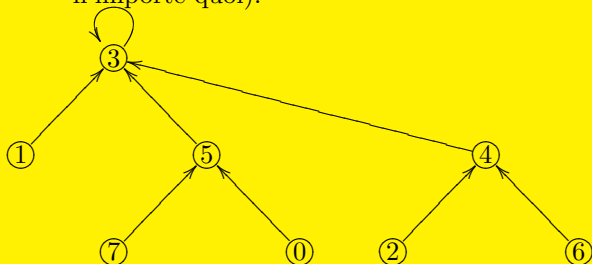


FIGURE 4 – La partition obtenue après union des classes de 3 et 4 : la classe de rang le plus élevé a absorbé l'autre ; la longueur des plus longs chemins n'est pas modifiée.

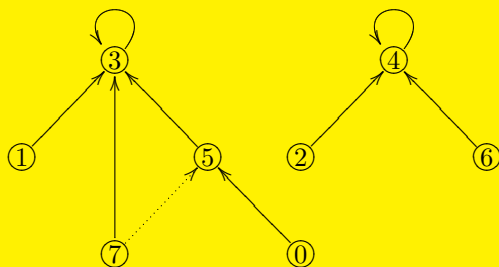
1.4 Compression de chemin

La méthode précédente précédente a une bonne complexité pour `union` et `find`. Mais on peut encore l'améliorer grâce à une dernière astuce : chaque fois qu'un appel `find(x,C)` est fait, on relie directement x à son représentant ultime. De ce fait, on réalise ce que nous appelons une *compression de chemin*.

La figure 5 explique ce qu'il se passe après un appel à `find(7,d)` dans la situation initiale de la figure 2. Alors que 7 pointait sur 5, l'appel à `find` le fait maintenant pointer directement sur son représentant ultime : 3. On a laissé en pointillé la liaison supprimée.

Q.16 Modifier la fonction `find(x,C)`. Elle renvoie toujours le représentant ultime de x mais elle agit de sorte que `C["link"][x]` désigne maintenant ce représentant.

Voici une séquence d'exécution :

FIGURE 5 – Après un appel à `find`, 7 pointe directement sur 3 ; l'ancienne liaison (en pointillés) a été supprimée

```

1 C=create(6)# C est la partition
2 print(C)
3 union(0,1,C)
4 print(C)
5 union(2,3,C)
6 print(C)
7 union(0,2,C)
8 print(C)
9 union(1,4,C)
10 print(C)

```

```

{'link': [0, 1, 2, 3, 4, 5], 'rank': [0, 0, 0, 0, 0, 0]}
{'link': [0, 0, 2, 3, 4, 5], 'rank': [1, 0, 0, 0, 0, 0]}
{'link': [0, 0, 2, 2, 4, 5], 'rank': [1, 0, 1, 0, 0, 0]}
{'link': [0, 0, 0, 2, 4, 5], 'rank': [2, 0, 1, 0, 0, 0]}
{'link': [0, 0, 0, 2, 0, 5], 'rank': [2, 0, 1, 0, 0, 0]}

```


Remarque. On observe que le rang, qui documentait la longueur du plus long chemin dans la classe d'un représentant ultime à la sous-section précédente, n'est plus maintenant qu'un majorant de cette longueur.

Avec l'astuce appliquée à `find` dans cette sous-section, la complexité devient encore meilleure la plupart du temps. On peut montrer que la complexité *amortie*² est en $O(\alpha(n))$ où α est réciproque de la fonction d'Ackerman. Cette fonction α tend si lentement vers l'infini que pour toute application pratique (avec les limites actuelles de mémoire et de rapidité des ordinateurs) on peut considérer que la recherche et la fusion sont à une écrasante majorité en temps constant.

Q.17 Faut-il adapter la fonction `union` ? Si oui, écrire le code de `union`.

Dans toute la suite, on considère (abusivement) qu'une opération de recherche ou d'union dans une structure *union-find* est de coût $O(1)$.

2 Algorithme de Kruskal

Dans cette section on s'intéresse uniquement à des graphes pondérés non orientés sans boucle (c'est à dire sans liaison de la forme : 

Dans toute la suite, les sommets sont numérotés de 0 à $n - 1$. Le graphe générique étudié possède donc n sommets. On note p le nombre d'arcs.

Un tel graphe est noté $G = (S, A, W)$ où S est l'ensemble des sommets, A est l'ensemble des arcs et W est la fonction de poids (à toute arête, elle associe un poids).

On appelle *arbre* tout graphe non orienté sans boucle *connexe* et *acyclique* :

- connexe : il y a un chemin entre toute paire de sommets ;
- acyclique : il n'y a pas de cycle, c'est à dire de chemin fermé ne passant pas plus d'une fois par arc.

Un théorème admis indique que tout graphe connexe vérifie $p \geq n - 1$ et pour tout graphe acyclique $p \leq n - 1$. Il vient qu'un arbre à n sommets possède exactement $p = n - 1$ arcs. La réciproque du théorème est également intéressante : tout graphe acyclique (resp. connexe) à $n - 1$ arcs est connexe (resp. acyclique).

². On dit *amortie* parce que, de temps en temps (mais rarement) la complexité est logarithmique, mais la plupart du temps elle est en $O(\alpha(n))$. Et les (rares) fois où ce n'est pas le cas, les opérations alors réalisées permettent de gagner beaucoup de temps dans les appels ultérieurs aux primitives.

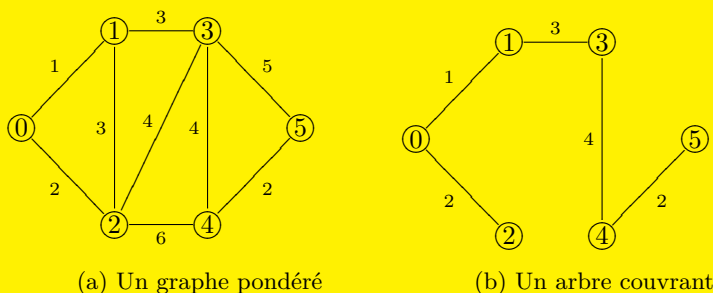


FIGURE 6 – Un graphe et un arbre couvrant

Définition 1. Étant donné un graphe $G = (S, A)$, un *arbre couvrant* de G est un sous-ensemble T d'arcs de G tel que :

- T est un arbre ;
- chaque sommet de G est l'extrémité d'au moins un arc de T (on dit que T *couvre* tous les sommets de G).

Un arbre couvrant n'a aucune raison d'être unique.

Définition 2. Étant donné un graphe pondéré $G = (S, A, W)$, un *arbre couvrant de poids minimal* de G est un arbre couvrant de G dont la somme des poids des arcs est minimale.

Un arbre couvrant de poids minimal n'a aucune raison d'être unique.

On donne figure 6 un exemple de graphe pondéré (6a) et d'un arbre couvrant de poids minimal (6b) pour ce graphe.

Algorithme de Kruskal (1956) Étant donné un graphe pondéré G orienté connexe sans boucle à n sommets, l'algorithme de Kruskal construit un arbre couvrant de poids minimal pour G .

1. On construit U , une structure de type union-find représentant une partition triviale pour les sommets ;
2. On construit P une pile qui contient tous les arcs par ordre décroissant de poids (le sommet de la pile est l'arc le plus léger, la base, le plus lourd) ;
3. On considère T une liste d'arcs initialement vide ;
4. Tant que T contient strictement moins de $n - 1$ arcs :
 - (a) retirer l'arc $\{x, y\}$ au sommet de P ;
 - (b) si x, y ne sont pas dans la même classe pour U alors :
 - i. ajouter l'arc $\{x, y\}$ à T ;
 - ii. fusionner dans U les classes de x et y .

Pour le graphe de la figure 6a, on donne les premières étapes de l'algorithme :

arc $\{x, y\}$	poids	action (ajouté ou ignoré)	U
			$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}$
$\{0, 1\}$	1	ajouté	$\{0, 1\}, \{2\}, \{3\}, \{4\}, \{5\}$
...			

Démonstration. L'algorithme complet :

arc $\{x, y\}$	poids	action	U
			$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}$
$\{0, 1\}$	1	ajouté	$\{0, 1\}, \{2\}, \{3\}, \{4\}, \{5\}$
$\{0, 2\}$	2	ajouté	$\{0, 1, 2\}, \{3\}, \{4\}, \{5\}$
$\{4, 5\}$	2	ajouté	$\{0, 1, 2\}, \{3\}, \{4, 5\}$
$\{1, 2\}$	3	ignoré	
$\{1, 3\}$	3	ajouté	$\{0, 1, 2, 3\}, \{4, 5\}$
$\{2, 3\}$	4	ignoré	
$\{3, 4\}$	4	ajouté	$\{0, 1, 2, 3, 4, 5\}$
$n - 1$ arcs sélectionnés : l'algorithme s'arrête			

□

Q.18 Compléter le tableau précédent de façon à simuler le déroulement complet de l'algorithme.

Q.19 Donner la complexité temporelle en fonction de n et p de l'algorithme de Kruskal.

Démonstration. Le union-find initial est en $O(n)$.

L'étape de remplissage de la pile de tri des arcs se fait en $O(p \log_2 p)$.

Il y a au plus p passages dans la boucle et pas moins de $n - 1$. L'opération de dépilement de $\{x, y\}$ se fait en $O(1)$. Vérifier si x, y sont dans la même classe se fait avec `find` et coûte $O(1)$. La fusion éventuelle (`union`) est aussi en $O(1)$ (voir remarque à la fin de la section précédente).

Bref, la boucle coûte un $O(p)$. Au total $O(p + n + p \log_2 p)$ qu'on peut résumer en $O(n + p \log_2 p)$. Puisque le graphe est connexe $p \geq n - 1$ et $O(p \log_2 p)$ décrit la complexité. \square

Q.20 Quelle est, au début de l'algorithme et en fonction de n , la taille maximale de la pile P ? Sa taille minimale?

Démonstration. $\binom{n}{2} = \frac{n(n-1)}{2}$ et $n - 1$ (puisque G est connexe). \square

La *correction* de l'algorithme s'établit en deux parties :

Arbre couvrant On montre que le graphe G' formé par l'ensemble d'arêtes T est un arbre couvrant (c.a.d. acyclique, connexe et tous les sommets de G sont dans G'). Cela fait l'objet de la question suivante.

Poids minimum On montre que le poids de l'arbre obtenu est le plus petit possible parmi les arbres couvrants du graphe. On ne fait pas la preuve de cette partie dans ce devoir.

On pose $G = (S, A, W)$.

Q.21 On numérote les passages dans la boucle de 0 (avant l'entrée dans la boucle) à f (tour final).

On utilise l'invariant : *À la fin du tour de boucle i , le graphe $G'_i = (S, T_i, W'_i)$ (où T_i désigne l'ensemble d'arête T au tour i et W'_i , la restriction de W à T_i) est acyclique.*

Montrer que cet invariant est vérifié à la fin du tour 0 et que s'il est vrai à la fin du tour i il est vrai à la fin du tour $i + 1$.

Q.22 Montrer que si l'invariant est vérifié à la fin du dernier tour, alors le graphe obtenu est un arbre couvrant.

Démonstration. A la fin du tour 0, il n'y a pas d'arc donc pas de cycle.

Si $H(i)$, soit $\{x, y\}$ sélectionné à la fin du tour $i + 1$.

— Si x, y sont dans la même classe d'équivalence, alors $T_i = T_{i+1}$ et donc l'invariant est vérifié.

— Sinon, x et y ne sont pas dans la même classe pour U_i : il n'y a pas de chemin qui les relie dans G'_i .

Aucune des classes autres que celles de x et y n'a été modifiée donc elles ne contiennent pas de cycle.

Supposons que G_{i+1} un cycle. Comme il n'y avait pas de cycle dans U_i , ce nouveau cycle contient forcément l'arc $\{x, y\}$. C'est donc qu'il existe une autre façon d'aller de x à y dans G_{i+1} que d'utiliser l'arête $\{x, y\}$.

Les autres arêtes utilisées sont dans G'_i puisque seule $\{x, y\}$ a été ajoutée à G'_i . Cela signifie que x et y sont déjà dans la même classe dans G'_i , ce qui est absurde.

Les graphes aux différents tours de boucles sont tous couvrants puisqu'ils contiennent le même ensemble de sommets que G . Par ailleurs ils sont acycliques comme on l'a vu. Or, au dernier tour, il y a $n - 1$ arcs. Par théorème un graphe acyclique à n sommets et $n - 1$ arcs est connexe.

Donc le graphe obtenu est un arbre couvrant. \square

Dans le cours, les graphes sont représentés comme des listes de listes de voisins. On note ici LLV ce type de représentation. Par exemple le graphe de la figure 6a est implémenté ainsi au format LLV :

```
1 g = [[(2,2),(1,1)], [(1,0),(3,2),(3,3)], [(2,0),(3,1),(4,3),(6,4)],
2 [(3,1),(4,2),(4,4),(5,5)], [(6,2),(4,3),(2,5)], [(5,3),(2,4)]]
```

La représentation du cours n'est pas bien adaptée à l'algorithme de Kruskal. On décide de représenter les graphes par un dictionnaire à deux clés. La clé `"n"` désigne le nombre de sommets (les sommets sont alors implicitement $0, 1, \dots, n - 1$) et `"arcs"` a pour valeur une liste de triplets (w, x, y) (poids, 1er sommet de l'arc, 2nd sommet de l'arc) non triée. Par convention, dans le triplet (w, x, y) , on impose $x < y$. On note ici K ce type de représentation.

Le graphe de la figure 6a peut donc être représenté au format K par :

```
1 g4kruskal1 = {"n":6, "arcs":[(2,0,2),(6,2,4),(2,4,5),(3,1,2),
2 (4,2,3),(4,4,3),(1,0,1),(3,1,3),(5,3,5)]}
```

Q.23 Écrire la fonction `llv2k(g)` qui prend en paramètre un graphe `g` représenté sous forme LLV et retourne le même graphe représenté sous la forme K.

On rappelle l'algorithme du tri fusion :

Algorithme 1 : Tri fusion (pseudocode)

Fonction `TriFusion(T)` :

```

  if  $|T| \leq 1$  then
    return  $T$ 
  diviser  $T$  en deux moitiés  $G$  (gauche) et  $D$  (droite)
   $G \leftarrow \text{TriFusion}(G)$ 
   $D \leftarrow \text{TriFusion}(D)$ 
  return Fusion( $G, D$ )

```

Fonction `Fusion(G, D)` :

```

   $R \leftarrow []$ ,  $i \leftarrow 0$ ,  $j \leftarrow 0$ 
  while  $i < |G|$  et  $j < |D|$  do
    if  $G[i] \leq D[j]$  then
      ajouter  $G[i]$  à  $R$ ;  $i \leftarrow i + 1$ 
    else
      ajouter  $D[j]$  à  $R$ ;  $j \leftarrow j + 1$ 
  ajouter le reste de  $G$  puis de  $D$  à  $R$ 
  return  $R$ 

```

Q.24 La fonction `fusion(gauche, droite)` qui fusionne deux listes croissantes et renvoie une 3ème liste croissante formée du contenu des deux premières a déjà été écrite dans la partie 1.1.

Écrire la fonction `tri_fusion(tab)` qui trie le tableau `tab` selon l'algorithme du tri fusion.

Q.25 Écrire la fonction `kruskal(g)` qui prend en paramètre un graphe donné sous la forme K. La fonction applique l'algorithme de Kruskal et retourne un arbre couvrant de poids minimal sous la forme K. Le graphe `g` ne doit pas être modifié durant l'opération.

```

1 g = {"n":6, "arcs":[(2,0,2),(6,2,4),(2,4,5),(3,1,2),(4,2,3),\
2               (4,4,3),(1,0,1),(3,1,3),(5,3,5)]}
3 kruskal(g)

```

```
{'n': 6, 'arcs': [(1,0,1),(2,4,5),(2,0,2),(3,1,3),(4,4,3)]}
```

Q.26 Écrire la fonction `poids(g)` qui prends en paramètre un graphe et retourne son poids total (la somme des poids de ses arcs).

```

1 tree = kruskal(g)
2 poids(tree)

```

```
12
```