ÉCOLE POLYTECHNIQUE – ÉCOLES NORMALES SUPÉRIEURES

SUJET ZERO — NOUVEAUX PROGRAMMES **FILIÈRE MP OPTION INFO**

COMPOSITION D'INFORMATIQUE – A – (XULCR)

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation est Caml Light.

⋆ ⋆ ⋆

Commentaire. Ce sujet n'a pas vocation à tomber au concours. Il s'agit d'un "sujet zéro" dont l'objectif est d'illustrer les programmes entrant en vigueur en classe de mathématiques spéciales à la rentrée 2014. À ce titre, l'accent a été mis sur les aspects les plus nouveaux du programme, à savoir la théorie des graphes en informatique, et incidemment les probabilités en mathématiques. On insiste sur le fait que ce sujet est significativement plus long et moins progressif qu'un sujet susceptible de tomber au concours.

Arbres couvrants de poids minimum

L'objet de ce sujet est l'étude du problème de l'arbre couvrant de poids minimum, un problème fondamental d'optimisation combinatoire apparaissant dans de multiples contextes, en particulier dans l'étude et la réalisation de réseaux.

Voici un exemple d'application qui donne une idée des problèmes concernés. On souhaite relier les villes de Bordeaux, Lille, Lyon, Marseille, Paris, et Toulouse par un nouveau type de fibre optique. Les nouveaux cables ne peuvent être enterrés que dans des tranchées déjà creusées entre les différentes villes (voir Figure 1, où les nombres sur les arêtes représentent la longueur des tranchées en kilomètres). Le problème est de déterminer dans quelles tranchées enterrer les cables de manière à ce que toutes les villes soient reliées, tout en minimisant le coût de l'installation (qui dépend linéairement de la longueur de cable utilisée). Le réseau des tranchées déjà creusées se modélise naturellement par un graphe non-orienté avec des poids sur les arêtes (égaux à la longueur des tranchées). L'objet que l'on recherche est un sous-graphe connexe dont le poids (égal à la somme des poids des arêtes) est minimum.

⋆ ⋆ ⋆

La **complexité**, ou le **coût**, d'un programme P (fonction ou procédure) est le nombre

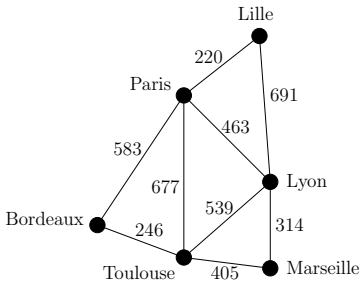


Figure 1: Les liaisons possibles entre les différentes villes.

d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, comparaison, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend d'un paramètre n, on dira que P **a une complexité en** $\mathcal{O}(f(n))$, s'il existe $K > 0$ tel que la complexité de P est au plus $Kf(n)$, pour tout n. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière.

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes.

Partie I. Implémentation des graphes

Quand V est un ensemble, on note $\binom{V}{2}$ l'ensemble de tous les sous-ensembles de V de cardinal 2. Formellement, $\binom{V}{2} = \{\{x, y\} \mid x, y \in V \text{ et } x \neq y\}$. Un **graphe** (non-orienté) est une paire $G = (V, E)$ telle que V est un ensemble fini d'éléments appelés **sommets de** G, et E est un sous-ensemble de $\binom{V}{2}$. Les éléments de E sont appelés **arêtes de** G. Pour alléger les notations, on notera xy au lieu de $\{x, y\}$.

Dans ce sujet, on considère qu'à chaque arête est associé un poids entier strictement positif. En Caml, on prendra comme ensemble de sommets $\{0, 1, \ldots, n-1\}$, où $n = |V|$, et on représentera le graphe sous la forme de sa matrice d'adjacence, dans laquelle la case i, j contient 0 si i et j ne sont pas adjacents, et le poids de l'arête ij sinon. Voici par exemple le graphe de la Figure 1 sous cette forme (après renommage de Lille, Paris, Lyon, Bordeaux, Toulouse, et Marseille en 0, 1, 2, 3, 4, et 5, respectivement).

```
let grapheEx=
[|[|0;220;691;0;0;0|];
  [|220;0;463;583;677;0|];
  [|691;463;0;0;458;314|];
  [|0;583;0;0;246;0|];
  [|0;677;458;246;0;405|];
  [|0;0;314;0;405;0|]|];;
```

Il sera également pratique de considérer chaque arête ij d'un graphe pondéré comme un triplet (i, j, p_{ij}), où p_{ij} est le poids de l'arête ij. Voici la liste des arêtes du graphe de la Figure 1 sous cette forme.

```
[ (0, 1, 220); (3, 4, 246); (2, 5, 314); (4, 5, 405);
(2, 4, 458);  (1, 2, 463); (1, 3, 583); (1, 4, 677); (0, 2, 691) ];
```

Question 1 Implémenter en Caml une fonction qui prend un graphe pondéré sous forme matricielle, et renvoie la liste de ses arêtes sous forme de triplets (i, j, p_{ij}), où i et j sont deux sommets adjacents et p_{ij} est le poids de l'arête ij.

```
(* Caml *)  liste_aretes :  int vect vect -> (int * int * int) list
```

Solution à la question 1.

```
let liste_aretes g=
  let l=ref [] in
  for i=0 to vect_length g-1 do
    for j=i+1 to vect_length g-1 do
      if g.(i).(j) <> 0 then l:=(i,j,g.(i).(j))::(!l) done done;
!l;;
```

□

Question 2 En utilisant le tri fusion, implémenter en Caml une fonction qui prend un graphe pondéré sous forme matricielle, et renvoie la liste de ses arêtes (sous la même forme que dans la question précédente) triées par ordre croissant de poids.

```
(* Caml *)  liste_aretes_triees :  int vect vect -> (int * int * int) list
```

Solution à la question 2. C'est une question de cours. Il faut juste faire attention à ce que la comparaison entre deux triplets se fasse en regardant le troisième élément de chaque triplet (i.e. le poids de l'arête correspondante).

```
let rec division=function
  | [] -> [],[];
  | [x] -> [x],[];
  | x::y::l -> let (a,b)=division l in (x::a),(y::b);;


let rec fusion=function
  | [], l -> l;
  | l, [] -> l;
  | (x::l1), (y::l2) -> let (_,_,p)=x and (_,_,q)=y in
                        if p < q then x::(fusion (l1, (y::l2)))
```

```
                    else y::(fusion ((x::l1), l2));;

let rec tri_fusion=function
  | [] -> [];
  | [x] -> [x];
  | l -> let (l1,l2)=division l in fusion (tri_fusion l1,tri_fusion l2);;

let liste_aretes_triees g=
  tri_fusion (liste_aretes g);;
```

□

Partie II. Préliminaires sur les arbres

Un graphe $G' = (V', E')$ est un **sous-graphe** de $G = (V, E)$ si $V' \subseteq V$ et $E' \subseteq E$. Quand $xy \in E$, on dit que x et y sont **adjacents**. Pour $k \geq 1$, un **chemin** de G est une suite de sommets deux à deux distincts $x_1 \dots x_k$ telle que pour tout $1 \leq i < k$ on a $x_i x_{i+1} \in E$. Les sommets x_1 et x_k sont appelés les **extrémités** du chemin. On dit que le chemin **relie** x_1 à x_k. Le chemin **passe** par les sommets x_1, …, x_k, et par les arêtes x_1x_2, …, $x_{k-1}x_k$.

Pour $k \geq 2$, un **cycle** de G est une suite de $k+1$ sommets $x_1 \dots x_k x_{k+1}$ telle que $x_1 = x_{k+1}$, les sommets $x_1, \dots, x_k$ sont deux à deux distincts, pour tout $1 \leq i < k$ on a $x_i x_{i+1} \in E$ et $x_1 x_k \in E$. Le cycle **passe** par les sommets x_1, …, x_k, et par les arêtes x_1x_2, …, $x_{k-1}x_k$, x_kx_1.

Un graphe $G = (V, E)$ est **connexe** si pour toute paire de sommets x, y de G, il existe un chemin reliant x à y. Un sous-ensemble X de V est une composante connexe de G si le sous-graphe

$$(X, \{xy \,|\, x, y \in X \text{ et } xy \in E\})$$

est connexe, et si X est maximal pour l'inclusion avec cette propriété. On rappelle un résultat classique : pour tout graphe $G = (V, E)$, les composantes connexes de G forment une partition de V. En effet, chaque composante connexe est une classe d'équivalence de la relation d'équivalence $\sim$ définie par : $u \sim v$ si et seulement s'il existe un chemin de u à v dans G.

On appelle **arbre au sens des graphes** tout graphe connexe qui ne contient aucun cycle. Nous attirons l'attention sur le fait que dans un arbre au sens des graphes, aucun sommet n'est privilégié : il n'y a pas de racine, ou de notion d'ancêtre, de père, de fils, etc. Dans la suite du sujet, on écrira simplement **arbre** par souci de brièveté.

Commentaire. Les notations en théorie des graphes ne sont pas toujours aussi standardisées que dans d'autres domaines des mathématiques, surtout en français. Plusieurs ouvrages utilisent "chaîne élémentaire" là où nous disons "chemin" et "graphe partiel" là où nous disons "sous-graphe" par exemple.

Le vocabulaire présenté ici et la définition précise des graphes donnée ici, n'ont pas vocation à être standards. Il est attendu des candidats qu'ils s'adaptent rapidement aux notations convenant au problème particulier étudié dans le sujet.

Étant donné un graphe $G = (V, E)$, et deux sommets u, v non adjacents dans G, on note $G + uv$ le graphe $G' = (V, E \cup \{uv\})$, c'est-à-dire le graphe obtenu à partir de G en ajoutant une arête entre u et v. De manière similaire, si u et v sont adjacents dans G, on note $G - uv$ le graphe $G' = (V, E \setminus \{uv\})$, c'est-à-dire le graphe obtenu à partir de G en supprimant l'arête entre u et v.

Question 3 Soit G un graphe, et $u \neq v$ deux sommets non adjacents de G. Montrer que l'un et seulement l'un des deux cas suivants se produit :

(i) u et v sont dans deux composantes connexes distinctes de G, $G+uv$ possède une composante connexe de moins que G, et aucun cycle de $G + uv$ ne passe par l'arête uv.

(ii) u et v sont dans la même composante connexe de G, $G + uv$ et G ont le même nombre de composantes connexes, et $G + uv$ possède un cycle qui passe par l'arête uv.

Solution à la question 3. Soient $C_1, \ldots, C_k$ les composantes connexes de G. Supposons que u et v sont dans deux composantes connexes distinctes C_i et C_j, alors, à cause de l'arête uv, $C_i \cup C_j$ est une composante connexe de $G + uv$. En effet, soient x et y deux sommets de $C_i \cup C_j$. Si $x, y \in C_i$ alors, comme C_i est une composante connexe, il existe un chemin reliant x à y. De même si $x, y \in C_j$. Pour le cas où $x \in C_i$ et $y \in C_j$ alors il existe, dans $G + uv$ un chemin de x à y qui passe de x à u (car C_i est une composant connexe de G) puis de u à v via la nouvelle arête uv puis de v à y (car C_j est une composant connexe de G). Pour prouver que $C_i \cup C_j$ est une composante connexe de $G + uv$ il reste donc à prouver sa maximalité pour la relation d'inclusion. Pour cela on raisonne par l'absurde et on suppose qu'il existe un sommet dans $V \setminus (C_i \cup C_j)$ qui est dans la même composante connexe que $C_i \cup C_j$ dans $G + uv$. Alors en particulier il existe un sommet $z \in V \setminus (C_i \cup C_j)$ qui est adjacent à un sommet de $C_i \cup C_j$ dans $G + uv$. Cela implique que z est adjacent à un sommet de $C_i \cup C_j$ dans G et contredit la maximalité de C_i et C_j. Comme pour tout $i' \in \{1, \ldots, k\}$ distinct de i et j, $C_{i'}$ est une composante connexe de $G + uv$, le nombre de composantes connexes de $G + uv$ est le nombre de composantes de G diminué de 1. En revanche si u et v sont dans la même composante connexe C_i de G, alors les composantes connexes de G sont les mêmes que celles de $G + uv$.

Il reste à remarquer qu'un cycle de $G + uv$ passe par l'arête uv si et seulement s'il existe un chemin entre u et v dans G. C'est équivalent au fait que les sommets u et v sont dans la même composante connexe de G. □

Question 4 Soit $G = (V, E)$ un graphe à n sommets. Montrer que les six conditions suivantes sont équivalentes *(indication : il est conseillé de considérer les m arêtes $e_1, \ldots, e_m$ de G, puis de construire G à partir du graphe $G_0 = (V, \emptyset)$ en posant pour tout $i \in \{1, \ldots, m\}$, $G_i = G_{i-1} + e_i$ et en posant $G = G_m$).*

(i) G est un arbre (autrement dit, G est connexe et sans cycle).

(ii) G est sans cycle et possède $n - 1$ arêtes.

(iii) G est connexe et possède $n - 1$ arêtes.

(iv) G est minimalement connexe, c'est-à-dire que G est connexe et que pour toute arête e de G, $G - e$ n'est pas connexe.

(v) Pour toute paire de sommets x, y de G, il existe un unique chemin de G d'extrémités x et y.

(vi) G est maximalement sans cycle, c'est-à-dire que G est sans cycle, et que pour toute paire de sommets non adjacents u, v de G, le graphe $G + uv$ contient un cycle.

Solution à la question 4. Notons que $G_0 = (V, \emptyset)$ possède n composantes connexes. On montre les implications suivantes.

(i)⇒(ii) Supposons que (i) est vérifiée. En appliquant la question 3 on voit qu'à chaque ajout d'arête dans la suite $G_0, \dots, G_m$, le nombre de composantes connexes diminue de 1 puisque G ne contient pas de cycle. Or G possède une seule composante connexe. On en déduit que $m = n - 1$, et que G possède $n - 1$ arêtes.

(ii)⇒(iii) Supposons que (ii) est vérifiée. En appliquant la question 3 on voit qu'à chaque ajout d'arête dans la suite $G_0, \dots, G_m$, le nombre de composantes connexes diminue de 1 puisque G est sans cycle. Comme il y a $n - 1$ étapes et G_0 possède n composantes connexes, le nombre de composantes connexes de G est 1, donc G est connexe.

(iii)⇒(iv) Supposons que (iii) est vérifiée. Quitte à renuméroter les arêtes de G, on suppose que $e = e_m$. En appliquant la question 3, on voit qu'à chaque ajout d'arête dans la suite $G_0, \dots, G_m$, le nombre de composantes connexes doit diminuer de 1 puisqu'il y a $n - 1$ étapes, et que le nombre de composantes connexes de G est 1. On en déduit que $G - e$ possède deux composantes connexes, et donc n'est pas connexe.

(iv)⇒(v) Soient x, y deux sommets de G. Si (iv) est vérifiée, alors comme G est connexe, il existe un chemin de x vers y. Supposons par contradiction qu'il existe deux chemins P et P' distincts ayant pour extrémités x et y. Alors il existe une arête uv dans la différence symétrique de P et P' (vus comme des ensembles d'arêtes), mettons dans $P \setminus P'$. Quitte à intervertir u et v, il existe un chemin de u à x dans $P - uv$, un chemin de v à y dans $P - uv$ et un chemin de x à y dans $P' - uv = P'$. Par conséquent, u et v sont dans la même composante connexe de $G - uv$ et donc d'après la question 3, $G - uv$ est connexe. Cela contredit le fait que G est minimalement connexe.

(v)⇒(vi) Si (v) est vérifiée, alors G est sans cycle car pour toute paire de sommets $x \neq y$ d'un cycle, il existe deux chemins arêtes-disjoints (et en particulier, distincts) de x à y. Soient $u \neq v$ deux sommets non adjacents de G. Il existe un chemin de u à v dans G, et si l'on ajoute l'arête uv à G, on obtient un cycle.

(vi)⇒(i) Supposons que (vi) est vérifiée. Si G n'est pas connexe, soient u et v deux sommets dans deux composantes distinctes de G. D'après la question 3, on sait qu'aucun cycle de $G + uv$ ne passe par l'arête uv, et donc que tout cycle de $G + uv$ est en fait un cycle de G. Il s'ensuit que $G + uv$ est sans cycle, ce qui contredit la propriété (vi). □

Commentaire. On notera l'analogie frappante avec l'algèbre linéaire. Si on voit un arbre comme un ensemble d'arêtes, et que l'on remplace "arête" par "vecteur", "sans cycle" par

"libre", "connexe" par "générateur", et "arbre" par "base", on obtient un théorème classique de caractérisation des bases d'un espace vectoriel. La condition (v) est analogue du fait que tout vecteur s'exprime de manière unique comme combinaison linéaire des éléments d'une base.

Soit $G = (V, E)$ un graphe connexe. Un arbre $T = (V, E')$ tel que $E' \subseteq E$ est appelé un **arbre couvrant de** G.

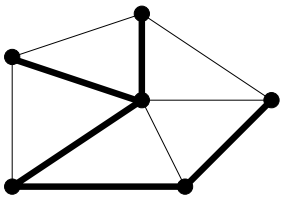


Figure 2: Un graphe et un arbre couvrant (en gras) de ce graphe.

Question 5 Montrer que tout graphe connexe $G = (V, E)$ possède un arbre couvrant.

Solution à la question 5. On considère $E' \subseteq E$ tel que (V, E') est connexe (E' existe puisque G est connexe), et on suppose que E' est minimal pour l'inclusion avec cette propriété. Par définition, (V, E') est minimalement connexe (au sens de la question 4), c'est donc bien un arbre. □

Commentaire. On peut continuer l'analogie avec l'algèbre linéaire : tout espace vectoriel possède au moins une base.

Question 6 Soit $G = (V, E)$ un graphe connexe, et $e_1, \ldots, e_m$ les arêtes de G triées dans un ordre quelconque. On considère la suite de graphes $(V, E_0), \ldots, (V, E_k)$ définie par l'algorithme suivant :

$k \leftarrow 0$ and $E_0 \leftarrow \emptyset$;
pour i de 1 à m **faire**
  si le graphe $(V, E_k \cup \{e_i\})$ est sans cycle **alors**
    $E_{k+1} \leftarrow E_k \cup \{e_i\}$;
    $k \leftarrow k + 1$;
  fin si
fin pour

Montrer que (V, E_k) est un arbre couvrant de G (ce qui implique $k = n - 1$).

Solution à la question 6. L'algorithme proposé va fournir par construction un graphe $G_k = (V, E_k)$ sans cycle. Supposons que G_k n'est pas connexe. Comme G est connexe, il s'ensuit que G_k contient deux composantes connexes distinctes X et Y telles qu'il existe une arête e_j de G ayant une extrémité dans X et l'autre dans Y. D'après la question 3, pour tout $i \leq k$ l'ajout de e_j à E_i ne crée aucun cycle. En particulier à l'étape j de l'algorithme, l'arête e_j est ajoutée à l'ensemble d'arêtes courant et donc $e_j \in E_k$, ce qui est une contradiction. On a montré que l'algorithme produit un graphe connexe sans cycle, donc un arbre couvrant de G. □

Commentaire. L'algorithme est glouton. C'est-à-dire qu'il ne revient jamais sur le choix d'inclure une arête dans la solution. L'existence d'un tel algorithme est tout à fait exceptionnelle en théorie des graphes. Par exemple, si l'on cherche un chemin reliant deux sommets x et y, commencer par une arête arbitraire partant de x peut échouer.

On peut poursuivre l'analogie avec l'algèbre linéaire : une base peut être construite en prenant tant que l'on peut des vecteurs ne créant aucune dépendance linéaire.

Question 7 Que renvoie l'algorithme de la question précédente si le graphe G donné en entrée n'est pas connexe?

Solution à la question 7. Supposons que $G = (V, E)$ n'est pas connexe et appelons $C_1, \dots, C_k$ ses composantes connexes. Alors l'algorithme va retourner un graphe $(V, E_1 \cup \cdots \cup E_k)$ où E_i est un arbre couvrant de $(C_i, E \cap \binom{C_i}{2})$ pour tout $1 \leq i \leq k$. □

Question 8 Soit $G = (V, E)$ un graphe et $E_1, E_2 \subseteq E$ tels que (V, E_1) et (V, E_2) sont des graphes sans cycle. Montrer que si $|E_2| > |E_1|$, alors il existe $e \in E_2 \setminus E_1$ tel que $(V, E_1) + e$ est sans cycle.

Solution à la question 8. Si une arête e de E_2 relie deux composantes connexes de (V, E_1), alors $e \in E_2 \setminus E_1$ et d'après la question 3, $(V, E_1) + e$ est sans cycle. On peut donc supposer que toute arête de E_2 relie deux sommets qui sont dans une même composante de (V, E_1). Comme $|E_2| > |E_1|$, il existe une composante connexe C de (V, E_1) telle que le nombre m_1 d'arêtes de (V, E_2) ayant leurs deux extrémités dans C est strictement supérieur au nombre m_2 d'arêtes de (V, E_1) ayant leurs deux extrémités dans C. Comme (V, E_1) est sans cycle, C est un arbre d'après la question 4, et m_1 est égal au nombre de sommets de C moins 1. Donc m_2 est supérieur ou égal au nombre de sommets de C, ce qui implique que (V, E_2) contient un cycle d'après la question 4, une contradiction. □

Commentaire. Là encore, il y a une analogie avec l'algèbre linéaire : si X et Y sont deux familles libres d'un espace vectoriel telles que $|Y| > |X|$, alors il existe $y \in Y \setminus X$ tel que $X \cup \{y\}$ est libre.

Partie III. Algorithme de Kruskal

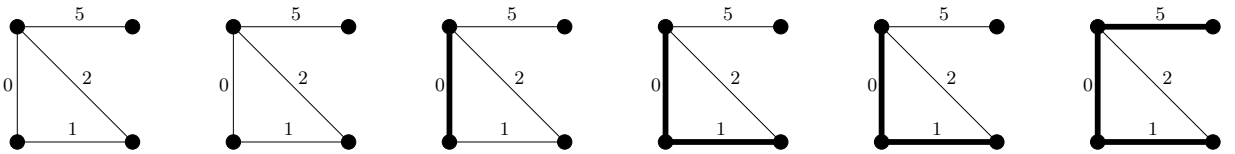
Comme on l'a vu dans le préambule, l'objectif est ici la recherche d'un sous-graphe connexe minimisant la somme des poids attribués aux arêtes. On a vu à la question 4 que les arbres sont précisemment les graphes minimalement connexes. La recherche d'un sous-graphe connexe minimisant la somme des poids attribués aux arêtes se ramène donc à celle d'un arbre couvrant de poids minimum. Le problème est défini rigoureusement comme suit :

*Soit $G = (V, E)$ un graphe connexe et $(w_e)_{e \in E} \in \mathbb{N}^E$ des poids sur les arêtes de G. Trouver un **arbre couvrant de poids minimum de** G, où le poids d'un arbre couvrant $T = (V, E')$ de G est défini comme $\sum_{e \in E'} w_e$.*

L'algorithme de Kruskal (1956) fonctionne de la manière suivante. Le graphe G donné en

entrée possède n sommets et m arêtes. On commence par trier les m arêtes de G par ordre de poids croissant (les algorithmes de tris vu en cours permettent de trier les m arêtes en temps $O(m \log m) = O(m \log \frac{n(n-1)}{2}) = O(m \log n)$). Puis on applique simplement l'algorithme de la question 6.

L'exécution de l'algorithme de Kruskal sur un petit exemple est donnée sur la Figure 3.



G | T à l'étape 0 | T à l'étape 1 | T à l'étape 2 | T à l'étape 3 | T à l'étape 4

Figure 3: Exécution de l'algorithme de Kruskal. Les arêtes de T sont en gras.

Question 9 Exécuter l'algorithme de Kruskal sur le graphe de la Figure 4. Inutile de détailler les étapes du calcul, dessiner simplement le graphe de départ avec en gras les arêtes de l'arbre couvrant obtenu *(note : il y a plusieurs exécutions possibles en fonction de la manière dont les arêtes de même poids sont ordonnées).*

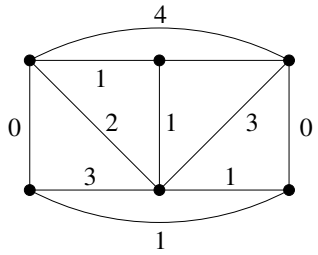


Figure 4: Le graphe de la question 9

Solution à la question 9. Chacun des quatre arbres de la Figure 5 correspond à une exécution possible de l'algorithme de Kruskal. □

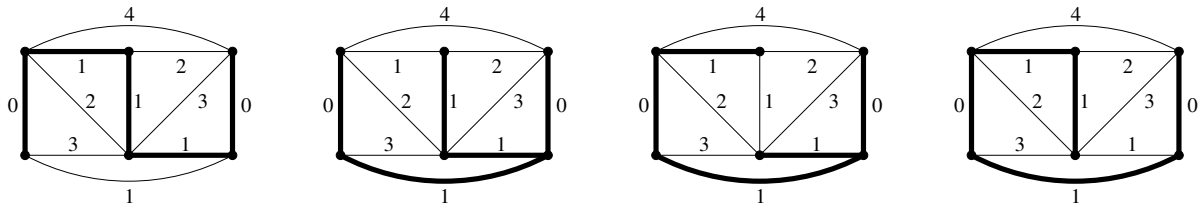


Figure 5: Les 4 arbres couvrant de poids minimum du graphe de la question 9

Question 10 Montrer en utilisant la question 8 que pour tout graphe $G = (V, E)$ connexe et pour tous poids $(w_e)_{e \in E}$ sur les arêtes de G, l'algorithme de Kruskal décrit plus haut retourne un arbre couvrant de poids minimum de G *(indication : montrer par récurrence sur k que pour tout k et pour tout arbre couvrant T de poids minimum de G, la somme des poids des k arêtes de poids minimum de l'arbre trouvé par l'algorithme de Kruskal est inférieure ou égale à la somme des poids des k arêtes de poids minimum de T).*

Solution à la question 10. D'après la question 6, on sait que l'algorithme retourne un arbre couvrant de G. Reste à voir que cet arbre est bien de poids minimum. À cette fin, on note $e_1, \ldots, e_m$ les arêtes de G triées par ordre de poids croissant. On note $i_1 \leq \cdots \leq i_{n-1}$ les indices des arêtes de l'arbre couvrant retourné par l'algorithme (qui sont donc $e_{i_1}, \ldots, e_{i_{n-1}}$). On note $j_1 \leq \cdots \leq j_{n-1}$ les indices des arêtes d'un arbre couvrant de poids minimum (qui sont donc $e_{j_1}, \ldots, e_{j_{n-1}}$). Comme les arêtes sont triées en ordre croissant, on a :

$$w(e_{j_1}) \leq \cdots \leq w(e_{j_{n-1}}) \tag{1}$$

Il suffit de montrer par récurrence que pour tout $k = 0, \ldots, n-1$, on a

$$w(e_{i_1}) + \cdots + w(e_{i_k}) \leq w(e_{j_1}) + \cdots + w(e_{j_k}).$$

Pour $k = 0$, c'est évident. Supposons l'inégalité vraie pour $k < n-1$ fixé. D'après la question 8, il existe une arête e_j dans

$$\{e_{j_1}, \ldots, e_{j_{k+1}}\} \setminus \{e_{i_1}, \ldots, e_{i_k}\}$$

telle que $(V, \{e_{i_1}, \ldots, e_{i_k}, e_j\})$ est sans cycle. Ceci implique $j > i_k$ (car sinon, l'algorithme aurait inclus e_j dans la solution) et d'après la façon dont l'algorithme choisi $e_{i_{k+1}}$, on a $i_{k+1} \leq j$ et donc $w(e_{i_{k+1}}) \leq w(e_j)$. On a donc :

$$\begin{array}{rcll} w(e_{i_1}) + \cdots + w(e_{i_k}) + w(e_{i_{k+1}}) & \leq & w(e_{i_1}) + \cdots + w(e_{i_k}) + w(e_j) & \\ & \leq & w(e_{j_1}) + \cdots + w(e_{j_k}) + w(e_j) & \text{par hypothèse de récurrence} \\ & \leq & w(e_{j_1}) + \cdots + w(e_{j_k}) + w(e_{j_{k+1}}) & \text{par (1) et } e_j \in \{e_{j_1}, \ldots, e_{j_{k+1}}\} \end{array}$$

□

Question 11 Résoudre le problème posé dans le préambule (relier les villes par de la fibre optique).

Solution à la question 11. Soit G le graphe de la Figure 1, et soit T le graphe dont les sommets sont les villes, et dans lequel on met une arête entre deux villes si l'on a posé un cable dans la tranchée reliant ces deux villes. On recherche donc un arbre couvrant de G de poids minimum. En appliquant l'algorithme de Kruskal au graphe G on obtient l'arbre couvrant de la Figure 6, ayant un poids de 1648.

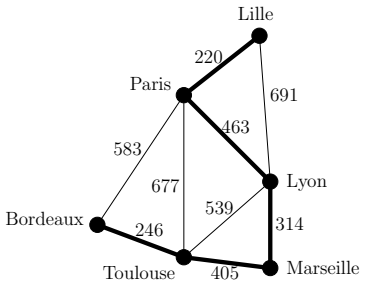


Figure 6: L'arbre couvrant de poids minimum de G (en gras).

□

Question 12 Un document produit en anglais doit être traduit en français, allemand, espagnol et italien. Les coûts de traduction (qui sont symétriques) sont indiqués dans la Table 1, en

milliers d'euros. On cherche à minimiser le coût de traduction. Pour cela on s'autorise à traduire de manière séquentielle le document (par exemple de l'anglais à l'italien et au français, puis de l'italien à l'allemand et l'espagnol). Modéliser le problème par un problème d'arbre couvrant de poids minimum (expliquer comment ramener le problème à une instance du problème de l'arbre couvrant de poids minimum), et résoudre le problème en utilisant l'algorithme de Kruskal.

	Allemand	Anglais	Espagnol	Français	Italien
Allemand	0	3	8	6	7
Anglais	3	0	4	3	6
Espagnol	8	3	0	2	3
Français	6	3	2	0	2
Italien	7	6	3	2	0

Table 1: Coûts de traduction

Solution à la question 12. On construit un graphe dont les sommets sont les 5 langues. Entre chaque paire de langues distinctes, on ajoute une arête dont le poids est le coût de traduction. On recherche donc un ensemble d'arête de coût minimal qui assure la connexité, et donc un arbre couvrant de poids minimum. En appliquant l'algorithme de Kruskal, on obtient un coût total de 10 000 euros, en traduisant les documents en français et en allemand (depuis l'anglais), puis en italien et en espagnol (depuis le français). □

On implémente maintenant en Caml l'algorithme de Kruskal. Il peut être vu de la manière suivante. L'entrée est un graphe $G = (V, E)$ avec n sommets et m arêtes, et chaque arête possède un poids. Au lieu de garder en mémoire au cours de l'exécution de l'algorithme quelles arêtes sont ajoutées à l'arbre couvrant T, on numérote les composantes connexes de T et on garde en mémoire, pour chaque sommet de G, le numéro de la composante de T à laquelle il appartient. Cela permet de tester efficacement si l'ajout d'une arête uv à T crée un cycle ou non : il suffit pour cela de tester si u et v appartiennent à la même composante. Si c'est le cas, on ne fait rien, et si ça n'est pas le cas on ajoute l'arête uv à T, ce qui revient à "fusionner" la composante de u et la composante de v en une seule et même composante. Les implémentations proposées ici reposent donc sur trois fonctions : `creer`, `composante` et `fusionner` (les prototypes des fonctions seront donnés plus tard). La fonction `creer` prend en entrée un graphe $G = (V, E)$ et initialise une représentation des composantes connexes de $(V, \emptyset)$ (qui sont au nombre de n puisqu'au début de l'algorithme T ne contient pas d'arêtes). La fonction `composante` permet de savoir à quelle composante appartient un sommet v. La fonction `fusionner` prend en paramètres deux composantes connexes i et j de T et fusionne ces deux composantes.

On passe maintenant aux détails de l'implémentation. Comme expliqué dans le préambule, les n sommets du graphe sont représentés par des nombres entiers de 0 à $n-1$, et le graphe est représenté sous forme matricielle. Grâce à la fonction `liste_aretes_triees` de la question 2, on dispose de la liste des arêtes du graphe (sous la forme de triplets où les deux premiers éléments représentent les sommets reliés par l'arête, et le troisième est le poids de l'arête) triées par ordre de poids croissant. Par exemple, voici la liste des arêtes du graphe du préambule triées par ordre croissant de poids :

```
[ (0, 1, 220); (3, 4, 246); (2, 5, 314); (4, 5, 405);
(2, 4, 458);  (1, 2, 463); (1, 3, 583); (1, 4, 677); (0, 2, 691) ]
```

Pour conserver en mémoire les numéros des composantes contenant chaque sommet, la fonction `creer n` initialise un tableau à n cases dans lequel chaque case i va contenir, tout au long de l'exécution de l'algorithme, le numéro de la composante contenant le sommet i. Initialement, chaque case i contient l'entier i (toutes les composantes connexes de T sont réduites à un sommet), et à la fin de l'algorithme toutes les cases du tableau contiennent le même entier (puisque l'algorithme s'arête lorsque T est connexe). Voici un exemple de l'évolution du tableau des composantes lorsqu'on applique l'algorithme de Kruskal au graphe du préambule :

```
[| 0; 1; 2; 3; 4; 5 |]
[| 0; 0; 2; 3; 4; 5 |] (* ajout de l'arête 01, fusion des composantes 0 et 1 *)
[| 0; 0; 2; 3; 3; 5 |] (* ajout de l'arête 34, fusion des composantes 3 et 4 *)
[| 0; 0; 2; 3; 3; 2 |] (* ajout de l'arête 25, fusion des composantes 2 et 5 *)
[| 0; 0; 2; 2; 2; 2 |] (* ajout de l'arête 45, fusion des composantes 2 et 3 *)
[| 0; 0; 0; 0; 0; 0 |] (* ajout de l'arête 12, fusion des composantes 0 et 2 *)
```

Question 13 Programmer en Caml la fonction `creer n`.

```
(* Caml *)  creer :  int -> int vect
```

Solution à la question 13.

```
let creer n =
  let comp = make_vect n 0 in
  for i=0 to n-1 do
    comp.(i) <- i
    done;
comp;;
```

□

La fonction `composante` prend en entrée le tableau des composantes et un entier (représentant un sommet), et renvoie le numéro de la composante contenant ce sommet.

Question 14 Programmer en Caml la fonction `composante c i`.

```
(* Caml *)  composante :  int vect -> int -> int
```

Solution à la question 14.

```
let composante c i =
  c.(i);;
```

□

La fonction `fusionner` prend en entrée un tableau de n entiers représentant les composantes, et deux entiers i et j représentant les numéros des composantes à fusionner, et modifie le tableau pour prendre en compte la fusion des composantes i et j.

Question 15 Programmer en Caml la fonction `fusionner c i j`. Noter que la fonction ne renvoie rien, elle modifie simplement le tableau `c`.

```
(* Caml *)  fusionner :  int vect -> int -> int -> unit
```

Solution à la question 15. La fonction remplace toutes les occurrences de j par i dans le tableau `c`.

```
let fusionner c i j =
  for k=0 to vect_length c - 1 do
    if c.(k) == j then c.(k) <- i
  done;;
```

□

Question 16 Donner la complexité des fonctions `créer`, `composante`, et de la fonction `fusionner`.

Solution à la question 16. La fonction `créer` initialise un tableau de taille n, sa complexité est donc $O(n)$. La fonction `composante` accède à une case d'un tableau, le coût est donc $O(1)$. La fonction `fusionner` parcourt tout le tableau en une passe. Son coût est donc $O(n)$. □

Question 17 À l'aide des fonctions déjà programmées, écrire en Caml la fonction `kruskal g`. Cette fonction prend en entrée un graphe G sous forme matricielle, applique l'algorithme de Kruskal, et renvoie la liste des arêtes faisant partie d'un arbre couvrant de poids minimum de G (chaque arête est représentée par un triplet, comme précédemment). Combien de fois les fonctions `créer`, `composante`, et `fusionner` sont-elles appelées au cours de l'algorithme ? En déduire une borne supérieure sur la complexité de cet algorithme en fonction de n et m (on pensera à prendre en compte le temps nécessaire au tri des arêtes, $O(m \log n)$).

```
(* Caml *)  kruskal :  int vect vect -> (int * int * int) list
```

Solution à la question 17.

```
let kruskal g =
```

```
  let comp = creer (vect_length g) in
  let aretes = ref (liste_aretes_triees g) and arbre = ref [] in
  while !aretes <> [] do
    let ((u,v,p)::l) = !aretes in
    aretes:=l;
    let i = composante comp u and j = composante comp v in
    if i <> j then begin
      fusionner comp i j;
      arbre:= (u,v,p):: !arbre;
    end
  done;
!arbre;;
```

L'algorithme exécute une fois la fonction `créer`, et $2m$ fois la fonction `composante`. La fonction `fusionner` est exécutée à chaque fois qu'une arête est ajoutée à T. Comme T contient $n-1$ arêtes, la fonction `fusionner` est exécutée $n-1$ fois. En ajoutant le temps du tri, la complexité de l'algorithme de Kruskal avec cette implémentation est donc $O(m \log n + n + 2m + n^2) = O(m \log n + n^2)$. □

Partie IV. Implémentation efficace

L'implémentation donnée dans la partie précédente permet de décider en temps constant si deux sommets sont dans une même composante : il suffit de comparer le contenu des deux cases indexées par les sommets dans le tableau des composantes. Par contre, la fonction `fusionner` est assez coûteuse. Dans cette partie, on cherche une implémentation où ces deux coûts sont plus équilibrés, ce qui se traduira par une meilleure complexité globale.

On appelle **arbre enraciné** tout couple (T, r), où T est un arbre (au sens des graphes) et r un sommet de T. Le sommet r est appelé la **racine** de l'arbre enraciné. Dans un arbre enraciné on peut définir naturellement une notion de parent : on rappelle que d'après la question 4, pour tout sommet v de T il existe un unique chemin de v à r dans T. Si $v \neq r$, l'unique sommet de ce chemin qui est adjacent à v est appelé le **parent** de v dans l'arbre enraciné. On définit le parent de r comme étant r lui-même.

La notion de parent permet de stocker de manière compacte un ensemble disjoint d'arbres enracinés. Les sommets sont étiquetés de 0 à $n-1$, et on représente l'ensemble d'arbres enracinés par un tableau de taille n : la case i du tableau contient le numéro du parent de i. Un exemple est donné sur la Figure 7.

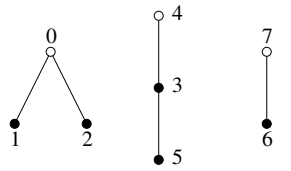


Figure 7: L'ensemble d'arbres enracinés associé au tableau `[|0;0;0;4;4;3;7;7|]`. Les sommets blancs sont les racines de chaque composante.

On rappelle que dans la partie III, on gardait en mémoire le numéro de la composante contenant chaque sommet à l'aide d'un tableau de taille n (la case i contenait le numéro de la composante contenant le sommet i). L'idée est maintenant la suivante : chaque composante connexe C de T va être représentée comme un arbre enraciné ayant le même ensemble de sommets que C. **Attention, et c'est primordial, cet arbre enraciné n'a pas nécessairement de rapport avec le sous-arbre induit par les arêtes de la composante C dans G).** Un exemple est donné sur la Figure 8 pour éviter toute confusion.

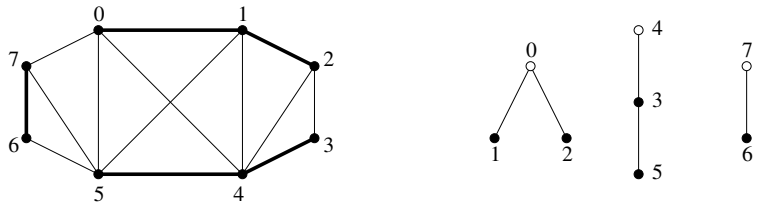


Figure 8: Un ensemble d'arbres enracinés associé aux composantes connexes de T pendant l'algorithme (les arêtes de T sont en gras). Le seul lien entre une composante de T et l'arbre enraciné associé est qu'ils partagent le même ensemble de sommets.

On commence donc par créer un tableau `parent` de n cases. Au départ, la case i du tableau contient l'entier i. Initialement, `parent` représente donc un ensemble de n arbres enracinés composés chacun d'un unique sommet (la racine). La fonction `creer` est donc identique à celle de la partie précédente.

La fonction `composante` permet de calculer à quelle composante appartient un sommet. Pour cela, il suffit de remonter de parent en parent jusqu'à la racine de l'arbre. Le numéro d'une composante est donc identifié au numéro du sommet racine de la composante.

Question 18 Implémenter en Caml la fonction `composante parent i`, qui prend en entrée un sommet $i \geq 0$ et un ensemble d'arbres enracinés sous la forme du tableau `parent`, et qui renvoie la racine de l'arbre contenant i.

```
(* Caml *)  composante :  int vect -> int -> int
```

Solution à la question 18. On renvoie i si i est la racine, et sinon on renvoie la racine de l'arbre contenant le parent de i.

```
let rec composante parent i=
  let j = parent.(i) in
  if j == i then i
  else composante parent j;;
```

□

La **hauteur** d'un arbre enraciné (T, r) est le nombre maximum d'arêtes sur un chemin entre r et un sommet de $\mathcal{T}$. Les arbres de la Figure 7 sont de hauteur 1, 2, et 1, respectivement.

La complexité de la fonction `composante` ci-dessus dépend de la hauteur de l'arbre enraciné contenant i. On cherche donc à implémenter la fonction `fusionner` de manière à obtenir des arbres enracinés **les moins profonds possible**. Pour cela on maintient également un tableau de taille n, `hauteur`, dont toutes les cases sont initialisées à 0 au début de l'algorithme. À chaque étape, pour tout arbre enraciné (T, i) du tableau `parent`, `hauteur.(i)` va contenir la hauteur de l'arbre enraciné (T, i). Par exemple sur la Figure 7, on aurait `hauteur.(0)`$= 1$, `hauteur.(4)`$= 2$, et `hauteur.(7)`$= 1$, et les autres cases du tableau `hauteur` peuvent *a priori* contenir des valeurs arbitraires (ces valeurs n'importent pas). On pourrait aisément, étant donné le tableau `parent` et une racine i, calculer la hauteur de l'arbre enraciné dont i est la racine. Mais il est plus efficace de maintenir ce tableau `hauteur` tout au long de l'algorithme et de le mettre à jour uniquement lorsque c'est nécessaire.

Question 19 Pour fusionner deux arbres enracinés dont les racines sont i et j, respectivement, on modifie juste une case du tableau `parent` de manière à ce que j devienne le parent de i (si `hauteur.(i) < hauteur.(j)`) ou i le parent de j (sinon). Dans quel cas doit-on aussi mettre à jour le tableau `hauteur`? Implémenter en Caml la fonction `fusionner parent hauteur i j` qui effectue ces opérations. Quelle est la complexité de cette fonction?

```
(* Caml *)  fusionner :  int vect -> int vect -> int -> int -> unit
```

Solution à la question 19. On appelle (T, i) l'arbre enraciné dont la racine est i, et (T', j) l'arbre enraciné dont la racine est j. Si la hauteur de (T, i) est strictement inférieure à la hauteur de (T', j), comme le nouveau père de i est j, la hauteur du nouvel arbre enraciné ne croît pas. De même si la hauteur de (T', j) est strictement inférieure à la hauteur de (T, i). En revanche si les hauteurs de $(\mathcal{T}, i)$ et (T', j) sont égales, i devient le père de j et il faut incrémenter `hauteur.(i)` (la hauteur du nouvel arbre enraciné).

```
let fusionner parent hauteur i j=
  if hauteur.(i) < hauteur.(j) then parent.(i) <- j
  else parent.(j) <- i;
  if hauteur.(i) == hauteur.(j) then hauteur.(i) <- hauteur.(i) + 1;;
```

La complexité de cette fonction est $O(1)$. □

Question 20 Montrer que tout arbre enraciné dont la racine est i contient au moins $2^{\texttt{hauteur.(i)}}$ sommets. En déduire que chacun des arbres enracinés est de taille $O(\log n)$, et conclure en donnant la complexité de l'algorithme de Kruskal avec cette implémentation.

Solution à la question 20. On montre la propriété par récurrence sur `hauteur.(i)`. Si `hauteur.(i)` vaut 0, l'arbre enraciné dont la racine est i contient uniquement i, c'est-à-dire $1 = 2^0$ sommet. Sinon, regardons la dernière fusion où la valeur de `hauteur.(i)` a changé. À cette étape, la valeur a été incrémentée du fait de la fusion de deux arbres de hauteur `hauteur.(i)`-1. Par récurrence, ces deux arbres contenaient chacun au moins $2^{\texttt{hauteur.(i)}-1}$ sommets, et l'arbre enraciné resultant de cette fusion contient donc au moins $2^{\texttt{hauteur.(i)}}$ sommets.

Comme chaque arbre enraciné contient au plus n sommet, tous les arbres enracinés sont de hauteur $O(\log n)$. Il s'ensuit que la complexité de cette implémentation de l'algorithme de Kruskal est $O(m \log n + n + 2m \log n + n) = O(m \log n)$. □

Il semble qu'à ce point, la complexité du tri des arêtes (en $O(m \log n)$) soit un obstacle pour obtenir un algorithme plus rapide. Mais il existe de nombreuses situations où le tri peut se faire de manière plus efficace.

Question 21 Implémenter en Caml une fonction qui prend en entrée un entier k et un tableau t d'entiers qui sont tous dans l'intervalle $\{0, 1, \ldots, k-1\}$, et qui trie le tableau (en place) en temps $O(m + k)$.

```
(* Caml *)  tri_lineaire :  int -> int vect -> unit
```

Solution à la question 21. On crée un tableau `compter` de taille k, en initialisant toutes les cases à 0. Ensuite on lit le tableau à trier case par case, et à chaque fois que l'on lit un élément i, on incrémente `compter.(i)`. Finalement, on lit le tableau `compter` de la gauche vers la droite, et pour toute case i on écrit `compter.(i)` fois l'élément i. La suite d'entiers obtenue correspond au tableau initial trié, et la complexité en temps est $O(m + k)$.

```
let tri_lineaire k t=
  let compter=make_vect k 0 and c=ref 0 and j=ref 0 in
  for i=0 to vect_length t-1 do
    compter.(t.(i)) <- compter.(t.(i)) + 1 done;
  while !c < k do
    if compter.(!c) == 0 then incr c
    else begin t.(!j) <- !c; incr j; compter.(!c) <- compter.(!c) - 1 end done;;
```

□

Lorsque la complexité du tri n'est plus une limitation pour l'algorithme de Kruskal, on cherche à améliorer la complexité du reste de l'algorithme. En particulier, dans l'implémentation précédente de la fonction `composante`, la complexité dépend uniquement de la hauteur des arbres enracinés et il est donc naturel d'essayer de raccourcir ceux-ci. La technique dite de "compression de chemin" revient à implémenter la fonction `composante` comme suit.

```
let rec composante i parent=
  if i <> parent.(i)
    then parent.(i) <- composante parent.(i) parent;
  parent.(i);;
```

Question 22 Expliquer ce que fait cette fonction `composante`.

Solution à la question 22. Lorsque l'on exécute `composante i parent`, la fonction renvoie l'indice j de la racine de l'arbre enraciné contenant i (comme précédemment). De plus, j devient le parent de i et de tous ses ancêtres dans l'arbre. Cela permet de raccourcir de manière significative la hauteur de l'arbre, sans changer la complexité de la fonction `composante`. □

Il se trouve que cette implémentation de l'algorithme de Kruskal est très efficace à la fois en pratique et en théorie. On peut montrer que si le tri des arêtes peut se faire en temps linéaire, la complexité de cette version de l'algorithme de Kruskal (utilisant la compression de chemins) est $O(m\,\alpha(n))$, où $\alpha(n)$ est une fonction tendant vers l'infini mais avec une croissance extrêmement lente (plus lente que le nombre de fois qu'il faut itérer la fonction logarithme en partant de n pour obtenir une valeur inférieure ou égale à 1).

Partie V. Coupe minimum

Étant donné un graphe $G = (V, E)$ et un ensemble X de sommets de G, la **coupe** associée à X, notée $\delta(X)$, est l'ensemble des arêtes ayant exactement une extrémité dans X. La **taille** de la coupe associée à X est le cardinal de $\delta(X)$. Dans cette partie, on utilise l'algorithme de Kruskal pour trouver de manière efficace une coupe de taille minimum dans un graphe G. Il s'agit d'un problème important d'optimisation combinatoire, qui revient à trouver la partie la plus isolée du graphe, et qui a des applications en logistique et en transport.

Au lieu de concevoir un algorithme déterministe, nous allons concevoir un algorithme probabiliste, qui peut renvoyer une réponse erronée avec une probabilité non-nulle (mais très faible). Un tel algorithme est appelé **algorithme de Monte-Carlo**.

On rappelle que dans l'algorithme de la question 6 (et donc dans l'algorithme de Kruskal), T ne contient au départ aucune arête (T consiste alors en n composantes connexes réduites chacune à un sommet), et qu'à chaque fois que l'on ajoute une arête à T le nombre de composantes connexes de T diminue de un.

Dans la suite de cette partie, on considère un graphe G à n sommets. L'idée de l'algorithme pour trouver une coupe de taille minimum dans G est de prendre un ordre aléatoire (tiré uniformément) sur les arêtes de G et d'exécuter l'algorithme de la question 6 en traitant les arêtes dans l'ordre choisi. La seule différence est qu'au lieu d'exécuter l'algorithme jusqu'à la fin, on s'arête lorsque T contient exactement deux composantes, appelons-les X et $V \setminus X$. L'objectif de cette partie est de montrer qu'avec une probabilité d'au moins $\frac{2}{n(n-1)}$, $\delta(X)$ est une coupe de taille minimum dans G.

Soit k la taille minimum d'une coupe de G.

Question 23 Montrer qu'à chaque étape de l'algorithme de la question 6, si T a i composantes connexes, alors G contient au moins $\frac{ik}{2}$ arêtes e telles que les extrémités de e sont dans des composantes connexes distinctes de T.

Solution à la question 23. Comme la taille minimum d'une coupe de G est k, pour toute composante connexe C de T, $|\delta(C)| \geq k$. Chaque arête e telle que les extrémités de e sont dans des composantes connexes distinctes de T est comptée exactement deux fois dans l'union des

$\delta(C_j)$, où les C_j sont les composantes connexes de T. Le graphe G contient donc au moins $\frac{ik}{2}$ arêtes e telles que les extrémités de e sont dans des composantes connexes distinctes de T. □

Soit Y un ensemble de sommets de G tels que $|\delta(Y)| = k$.

Question 24 Soit i une étape de l'algorithme. On considère T au moment précis où il passe de i composantes connexes à $i-1$ composantes connexes. Montrer que la probabilité que l'arête que l'on ajoute à T (de manière à avoir $i-1$ composantes connexes) soit dans $\delta(Y)$ est au plus $\frac{2}{i}$.

Solution à la question 24. L'ordre des arêtes étant aléatoire uniforme, l'arête ajoutée à T est sélectionnée de manière aléatoire uniforme entre toutes les arêtes dont les deux extrémités sont dans des composantes différentes de T. D'après la question 23, il y a $\frac{ik}{2}$ arêtes de ce type, donc la probabilité de sélectionner une arête de $\delta(Y)$ est au plus $|\delta(Y)|/\frac{ik}{2} = \frac{2}{i}$. □

Question 25 Montrer qu'à la fin de l'algorithme, lorsque T a exactement deux composantes connexes (appellons-les X et $V - X$), la probabilité que $\delta(X) = \delta(Y)$ est au moins $\frac{2}{n(n-1)}$.

Solution à la question 25. On remarque que lorsque T a exactement deux composantes connexes X et $V \setminus X$, on a $\delta(X) = \delta(Y)$ si et seulement si aucune des arêtes de T n'est dans $\delta(Y)$. Notons e_i l'arête dont l'ajout à T fait passer T de i composantes connexes à $i-1$ composantes connexes. D'après la question 24, la probabilité que $e_i \in \delta(Y)$ est au plus $\frac{2}{i}$. Donc la probabilité qu'aucune des arêtes de T ne soit dans $\delta(Y)$ est au moins $\prod_{i=3}^{n}(1 - \frac{2}{i}) = \prod_{i=3}^{n}(\frac{i-2}{i}) = \frac{2}{n(n-1)}$. □

On peut estimer que $\frac{2}{n(n-1)}$ est une probabilité trop faible pour que l'algorithme soit intéressant en pratique. On décide donc d'exécuter l'algorithme t fois, pour un certain entier $t \geq 1$ dépendant de n, et de renvoyer la plus petite coupe trouvée au cours de ces t exécutions.

Question 26 Montrer que la probabilité que la plus petite coupe trouvée au cours de ces t exécutions ne soit pas une coupe minimum est au plus $\exp(-\frac{2t}{n(n-1)})$. En utilisant la partie précédente et en admettant qu'une permutation aléatoire uniforme d'un ensemble à m éléments peut être générée en temps $O(m)$, donner la complexité d'un algorithme probabiliste qui renvoie une coupe minimale de G avec probabilité au moins $1 - e^{-10} \approx 0.999955$.

Solution à la question 26. D'après la question 25, à chaque exécution de l'algorithme, la probabilité que la coupe obtenue ne soit pas de taille minimum est au plus $1 - \frac{2}{n(n-1)}$. Donc la probabilité qu'aucune des coupes obtenues pendant les t exécutions de l'algorithme soit de taille minimum est au plus $(1 - \frac{2}{n(n-1)})^t \leq \exp(-\frac{2t}{n(n-1)})$.

En exécutant l'algorithme $t = 5n^2 = O(n^2)$ fois, cette probabilité est au plus e^{-10}. Donc la probabilité que la plus petite coupe obtenue au cours des t exécutions soit une coupe minimale est au moins $1 - e^{-10}$. On a montré dans la partie précédente que la complexité de l'algorithme de Kruskal est $O(m \log n)$, et selon les hypothèses de la question 26, chaque génération d'un ordre aléatoire uniforme sur les arêtes de G coûte $O(m)$. On obtient donc un algorithme en $O(mn^2 \log n)$. □

Commentaire. Le meilleur algorithme connu pour ce problème est en $O(n^2 \log^3 n)$, cf. D. Karger, C. Stein, A new approach to the minimum cut problem*, J. ACM* **43(4)** *(1996), 601–640.*

* *
*