Corrigé de l'épreuve zéro d'informatique 2015
X - ENS MP option informatique

Partie I - Implémentation des graphes

1 Il suffit de parcourir la moitié supérieure de la matrice en ajoutant les arêtes à une liste initialement vide:

```
let liste_aretes g =
   let n = vect_length g in
   let l = ref [ ] in
   for i = 0 to n-2 do
   for j = i+1 to n-1 do
      if g.(i).(j) <> 0 then
         l := (i,j,g.(i).(j)) :: !l
   done;
   done;
   !l;;
```

2 Pour mettre en place un tri fusion de triplet (le tri est fait sur les troisièmes coordonnées), nous avons besoin d'une fonction qui coupe une liste en deux parties de tailles égales (à un près) et d'une fonction qui fusionne deux listes triées. Cela donne:

```
let decoupe l = match l with
   | [] -> [],[]
   | [a] -> [],[a]
   | a::b::q -> let l1,l2 = decoupe q in a::l1,b::l2;;

let rec fusionne l1 l2 = match l1,l2 with
   | [],_ -> l2
   | _,[] -> l1
   | (a,b,c)::q1,(A,B,C)::q2 when c<C -> (a,b,c)::(fusionne q1 l2)
   | (a,b,c)::q1,(A,B,C)::q2 -> (A,B,C)::(fusionne l1 q2);;

let rec tri_fusion l = match l with
   | [] -> l
   | [a] -> l
   | _ -> let l1,l2 = decoupe l in fusionne (tri_fusion l1) (tri_fusion l2);;
```

Partie II - Préliminaires sur les arbres

3 Pour $x \in V$, notons C_x et C'_x les composantes connexes de x dans G et dans $G+uv$. Nous sommes dans l'un (et un seul) des deux cas suivants:

(i) $C_u \neq C_v$: pour $x \in V$, on a alors $C'_x = C_x$ si $x \notin C_u \cup C_v$ et $C'_x = C_u \cup C_v$ sinon; le graphe $G + uv$ possède donc exactement une composante connexe de moins que G. Si un cycle de $G + uv$ passait par l'arête uv, on aurait un chemin $x_1 \dots x_{k+1}$ dans $G+uv$ avec $x_1 = x_{k+1} = u$ et $x_2 = v$, ce qui donnerait un chemin $x_2 \dots x_{k+1}$ de u à v dans G, ce qui contredirait l'hypothèse $C_u \neq C_v$. Aucun cycle de $G+uv$ ne passe donc par uv.

(ii) $C_u = C_v$: pour $x \in V$, on a alors $C'_x = C_x$ et le graphe $G + uv$ a autant de composantes connexes que G. Comme u et v sont connectés dans G, il existe un chemin $x_1 \dots x_k$ dans G qui relie u à v. Comme u et v ne sont pas adjacents dans G, $k \geq 3$ (dans la définition d'un cycle, il y a une petite erreur d'énoncé: il faut avoir $k \geq 3$ pour éviter de dire que xyx est un cycle quand xy est une arête du graphe). Ainsi, $x_1 \dots x_k u$ est un cycle de $G + uv$ qui passe par l'arête uv.

4

$(i) \Longrightarrow (ii)$ Supposons que G est un arbre et reprenons les notations proposées par l'énoncé. Les graphes G_i sont tous acycliques (ce sont des sous-graphes du graphe acyclique G) donc, d'après la question précédente, le nombre de composantes connexes décroit d'une unité à chaque ajout d'une arête. Comme G_0 a n composantes connexes et $G_m = G$ une seule, nous en déduisons que $m = n - 1$: G est sans cycle et possède $n - 1$ arêtes.

$(ii) \Longrightarrow (iii)$ Supposons que le graphe G vérifie (ii) et notons $V_1, \dots, V_k$ ses composantes connexes. La restriction de G a chaque V_i est alors un arbre (V_i, E_i) (E_i est l'ensemble des arêtes de G dont les extrémités appartiennent V_i), puisqu'il est connexe et sans cycle. On a donc, d'après la première application:

$$\forall i \in \{1, \dots, k\},\ \mathrm{Card}(E_i) = \mathrm{Card}(V_i) - 1$$

ce qui donne $n - 1 = \mathrm{Card}(E) = \sum_{1 \leq i \leq k} \mathrm{Card}(E_i) = \sum_{1 \leq i \leq k} (\mathrm{Card}(V_i) - 1) = \mathrm{Card}(V) - k = n - k$. On en déduit donc que G est connexe et possède $n - 1$ arêtes.

$(iii) \Longrightarrow (iv)$ En appliquant la question 3, on montre que le nombre de composantes connexes d'un graphe possédant m arêtes est au moins $n-m$ (puisque l'ajout d'une arête ne modifie le nombre de composantes connexes que d'une unité au plus). On en déduit que si G est connexe et possède $n - 1$ arête, $G - e$ possède au moins deux composantes connexes pour tout arête e de G: G est minimalement connexe.

$(iv) \Longrightarrow (v)$ Supposons que G est minimalement connexe. Comme G est connexe, il existe pour tout couple (x, y) de sommet un chemin de x à y. Si $x = y$, ce chemin est unique : c'est le chemin de longueur nulle (les sommets d'un chemin sont deux à deux distincts). Sinon, notons $x_1 \dots x_k$ et $y_1 \dots y_l$ deux tels chemins (avec $k \geq 2$, $l \geq 2$, $x_1 = y_1 = x$ et $x_k = y_l = y$). Comme $G_1 = G - x_1x_2$ n'est pas connexe, c'est que x_1 et x_2 ne sont pas dans la même composante connexe de G_1 (question 3). Comme y reste connecté à x_2, on en déduit que x n'est pas connecté à y dans G_1. Cela prouve qu'un des arêtes de $y_1 \dots y_l$ est égale à x_1x_2: comme les sommets des chemins sont deux à deux distincts et que $y_1 = x = x_1$, on a nécessairement $x_1 = y_1$. Une récurrence évidente prouve que $k \leq l$ et $x_1 \dots x_k = y_1 \dots y_k$: on peut ensuite conclure en remarquant que $y_k = x_k = y$ impose $k = l$ et l'égalité des deux chemins.

$(v) \Longrightarrow (vi)$ Supposons que G vérifie (iv). Alors G est sans cycle, car s'il existait un cycle c d'un sommet x à lui-même était un cycle, il y aurait deux chemins de x à x: le cycle c et le chemin de longueur nul x. D'autre part, soit (u, v) une paire de sommets non adjacents. Nous pouvons appliquer la question 3: nous sommes dans le cas (ii) puisque G est connexe et $G + uv$ contient un cycle.

$(vi) \Longrightarrow (i)$ Supposons que G est sans cycle et non connexe. Il existe alors deux sommets u et v non connectés dans G. Les sommets u et v sont donc distincts et non adjacents, ce qui nous met dans le cas (i) de la question 3: $G + uv$ ne possède aucun cycle passant par uv, donc $G + uv$ est acyclique puisque G est lui-même acyclique. Ainsi, nous avons démontré par contraposée que tout graphe maximalement sans cycle était un arbre.

5 On peut utiliser la propriété (iv) pour construire un arbre couvrant. On construit la suite (finie) G_i par récurrence: $G_0 = G$ (qui est connexe) et tant que G_i n'est pas minimalement connexe, on choisit une arête e_i de G_i telle que $G_i - e_i$ soit connexe et on pose $G_{i+1} = G_i - e_i$. Comme le nombre d'arête décroit d'une unité à chaque étape, ce processus s'arrête et on atteint un graphe $G_m = (V, E')$ minimalement convexe: ce graphe est un arbre (car $(iv) \Longrightarrow (i)$) et c'est bien un arbre couvrant de G, puisque $E' \subset E$.

6 L'algorithme proposé utilise la propriété (vi), mais en se restreignant aux arêtes uv du graphe. Pour bien détailler les choses, notons k_i la valeur de la variable k à la sortie de la i-ème boucle (avec $k_0 = 0$, valeur initiale de k). La propriété: "$E_{k_i} \subset E$ et (V, E_{k_i}) est sans cycle" est trivialement un invariant de boucle: on en déduit que (V, E_{k_m}) est un sous-graphe sans cycle de G.

La connexité va se démontrer grâce à un second invariant de boucle: "les graphes $G_{i=}(V, \{e_1, \ldots, e_i\})$ et (V, E_{k_i}) ont les mêmes composantes connexes. En effet, cette propriété est vérifiée quand $i = 0$, puisque $E_{k_0} = \emptyset$. Soit $i \in \{0, \ldots, m-1\}$ et supposons la propriété vérifiée au rang i. Notons uv l'arête e_{i+1} et $k = k_i$. D'après la question 3, deux cas sont possibles:

(i) u et v ne sont pas dans la même composante connexe de (V, E_k): comme aucun cycle de $(V, E_k) + uv$ ne passe par uv et comme (V, E_k) est sans cycle, $(V, E_k) + uv$ est également sans cycle et $k_{i+1} = k_i + 1$. On en déduit que $E_{k_{i+1}} = E_{k_i} + uv$ et que le graphe $(V, E_k) + uv$ a une composante connexe de moins que (V, E_k), les composantes connexes de u et v ayant été réunies. Comme u et v ne sont pas non plus dans la même composante connexe de G_i, on regroupe également les composantes connexes de u et de v dans G_i pour obtenir les composantes connexes de G_{i+1}: la propriété est donc vérifiée au rang $i + 1$.

(ii) u et v sont dans la même composante connexe de (V, E_k): alors $(V, E_k) + uv$ possède un cycle, donc $k_{i+1} = k_i$ et $E_{k_{i+1}} = E_{k_i}$. Les composantes connexes de $(V, E_{k_{i+1}})$ sont donc les mêmes que celles de (V, G_i), qui sont également les mêmes que celles de (V, G_{i+1}), puisque u et v sont dans la même composante connexe pour (V, G_i).

Comme $G_m = G$ est connexe, on en déduit que (V, E_{k_m}) l'est également: l'algorithme proposé construit donc un arbre couvrant de G.

7 Si G n'est pas connexe, l'algorithme précédent travaille séparément sur chaque composante connexe de G et construit une forêt d'arbres couvrants: (V, E_k) est la réunion d'une famille d'arbres qui couvrent chaque composante connexe de G.

8 Notons p_1 et p_2 les cardinaux de E_1 et E_2 et supposons qu'il n'existe pas d'arête e de $E_2 \setminus E_1$ telle que $(V, E_1) + e$ soit sans cycle. Nous allons appliquer l'algorithme précédent au graphe $(V, E_1 \cup E_2)$ en modifiant l'ordre des arêtes. La valeur finale k_f du paramètre k ne dépend pas de l'ordre dans lequel on traite les arêtes, puisqu'il ne dépend que du nombre de composantes connexes du graphe $(V, E_1 \cup E_2)$ (plus précisément, $n - k_f$ est le nombre de composantes connexes du graphe puisqu'on part avec n composantes connexes et qu'à chaque incrémentation de k, il y a une composante connexe en moins).

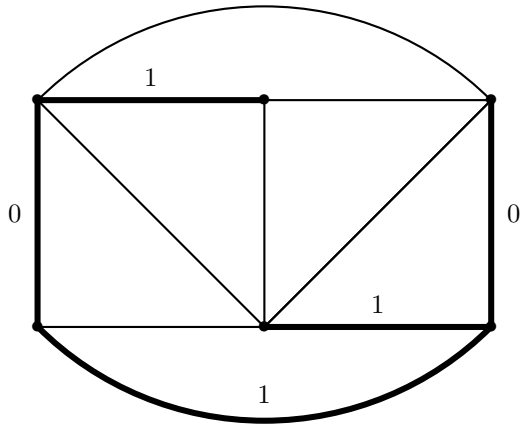
Numérotons les arêtes de $E_1 \cup E_2$ en commençant par les éléments de E_2. Comme (V, E_2) est sans cycle, les p_2 premiers passages dans la boucle (pour i variant de 1 à p_2) conduisent à une incrémentation de k. On en déduit que $k_f \geq p_2$.

Si nous numérotons maintenant les arêtes en commençant par les éléments de E_1, on aura $k = p_1$ et $V_k = E_1$ après les p_1 premiers passage dans la boucle. Les passages suivants ne vont plus modifier k, puisque chaque nouvelle arête étudiée e est élément de $E_2 \setminus E_1$. Nous aurons donc $p_1 = k_f$, d'où $p_1 \geq p_2$.

Nous avons ainsi montré le résultat demandé par contraposée.

Partie III - Algorithme de Kruskal

9 Il manque sur la figure 4 le poids d'une arête. En supposant que ce poids est au moins égal à 1, on peut obtenir l'arbre couvrant de poids minimal 3:



10 Soit $(e_1, e_2, \ldots, e_{n-1})$ la liste des arêtes construite par l'algorithme de Kruskal (par construction, les arêtes e_i sont de poids croissants) et soit $(f_1, f_2, \ldots, f_{n-1})$ les arêtes d'un arbre couvrant. Quitte à réordonner les f_i, nous pouvons également supposer qu'elles sont rangées par ordre croissant de poids. Comme e_1 est une arête de poids minimal, nous avons $p_{e_1} \leq p_{f_1}$, ce qui initialise la récurrence.

Soit maintenant $k \in \{1, \ldots, n-2\}$ et supposons que $p(e_1) + \cdots + p(e_k) \leq p(f_1) + \cdots + p(f_k)$. On peut alors appliquer la question 8 avec $E_1 = \{e_1, \ldots, e_k\}$ et $E_2 = \{f_1, \ldots, f_{k+1}\}$: il existe $i \in \{1, \ldots, k+1\}$ tel que $f_i \notin E_1$ et $(V, E_1) + f_i$ soit sans cycle. Ceci prouve que l'arête f_i est située après e_{k+1} (au sens large) dans la liste traitée par l'algorithme de Kruskal (sinon, elle aurait été ajoutée à l'ensemble E_k au moment de son traitement et serait élément de E_1). On en déduit donc que $p_{e_{k+1}} \leq p_{f_i} \leq p_{f_{k+1}}$, ce qui donne $p(e_1) + \cdots + p(e_k) + p(e_{k+1}) \leq p(f_1) + \cdots + p(f_k) + p(f_{k+1})$.

Nous obtenons ainsi la propriété $p(e_1) + \cdots + p(e_{n-1}) \leq p(f_1) + \cdots + p(f_{n-1})$, ce qui prouve que l'arbre renvoyé par l'algorithme de Kruskal est bien un arbre couvrant minimal.

11 L'algorithme de Kruskal donne l'arbre couvrant minimal:

(Paris, Lille, 220), (Bordeaux, Toulouse, 246), (Lyon, Marseille, 314), (Marseille, Toulouse, 405), (Lyon, Paris, 463)

12 La matrice donnée n'est pas symétrique: en remplaçant la valeur 3 (Espagnol-Anglais) par 4, l'algorithme de Kruskal nous donne l'arbre couvrant minimal:

(Français, Espagnol, 2), (Français, Italien, 2), (Français, Anglais, 3), (Anglais, Allemand, 3)

Le coût minimal sera donc obtenu en traduisant le texte original en allemand et en français, puis en traduisant la version française en espagnol et en italien.

13 à 15 Ces procédures ne posent aucun problème:

```
let creer n = let c = make_vect n 0 in
   for i=1 to n-1 do
```

```
      c.(i) <- i
   done;
   c;;

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

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

16 Les fonctions `créer`, `composante` et `fusionner` sont de complexités respectives $O(n)$, $O(1)$ et $O(n)$.

17 Le code s'écrit une nouvelle fois sans problème:

```
let kruskal g =
   let n = vect_length g in
   let c = creer n in
   let L = ref (liste_aretes_triees g) in
   let A = ref [] in           (* liste des arêtes retenues *)
   while !L <> [] do
      let (i,j,p) = hd(!L) in
      L := tl !L;
      match composante c i,composante c j with
         | a,b when a<>b -> A := (i,j,p):: !A; fusionner c a b
         | _ -> ()
   done;
   !A;;
```

La complexité de la fonction `kruskal` s'obtient en ajoutant:

- le temps de calcul de `créer n`: $O(n)$;
- le temps de calcul de `liste_aretes_triees g`: $O(m\log(n))$;
- le temps mis pour parcourir la liste des arêtes: $O(m) + (n-1)O(n)$ (chaque arête conduit a un calcul de temps constant et les $n-1$ arêtes retenues font un appel à la fonction *fusion*).

ce qui donne une complexité $O(n^2) + O(m\log(n))$.

18 J'ai choisi d'écrire une procédure itérative: une référence j pointe sur le sommet i et on remplace j par son parent tant que celui-ci est différent de j:

```
let composante parent i =
   let j=ref i in
   while parent.(!j) <> !j do
```

```
    j:=parent.(!j)
  done;
  !j;;
```

19 Il faut incrémenter la hauteur de i quand i et j ont la même hauteur. Cela donne:

```
let fusionner parent hauteur i j
  match hauteur.(i),hauteur.(j) with
    |a,b when a<b -> parent.(i) <- j
    |a,b when a=b -> parent.(j) <- i; hauteur.(i) <- a+1
    |_ -> parent.(j) <- i;;
```

20 Il faut comprendre que l'on parle ici d'arbres enracinés construits en utilisant exclusivement la fonction `fusionner` précédente. Pour faire une preuve précise, nous avons besoin de notations: nous sommes partis des vecteurs `parent` et `hauteur`, qui valaient respectivement au début du calcul $P_0 = [|0; 1; \ldots; n-1|]$ et $H_0 = [|0; \ldots; 0|]$, puis nous avons effectué un certain nombre d'appels à la fonction `fusionner`. Notons $P_1, \ldots, P_k$ et $H_1, \ldots, H_k$ les vecteurs `parent` et `hauteur` après les différents appels à `fusionner`. Nous aurons également besoin des vecteurs $T_0, T_1, \ldots, T_k$ qui contiennent les tailles des différents arbres. Ainsi, après la j-ème fusion, pour tout i, l'arbre enraciné en i est de hauteur $H_j.(i)$ et de taille $T_j.(i)$ (ceci est également valable si i n'est plus une racine à l'instant j).

Nous allons montrer par récurrence sur $j \in \{0, \ldots, k\}$ la propriété:

$$\mathcal{P}_j : \quad \forall i \in \{0, \ldots, n-1\},\ T_j.(i) \geq 2^{H_j.(i)}.$$

- $\mathcal{P}_0$ est vérifiée car pour tout i, $T_0.(i) = 1 = 2^0 = 2^{H_0.(i)}$.
- Soit $j \in \{1, \ldots, k\}$ et supposons que $\mathcal{P}_{j-1}$ est vérifiée. L'étape j a consisté à fusionner deux arbres enracinés en i_1 et i_2, i_1 devenant le nouveau père de i_2. Nous avons alors:

  $$\forall i \neq i_1,\ T_j.(i) = T_{j-1}.(i) \geq 2^{H_{j-1}.(i)} = 2^{H_j.(i)}$$

  puisque seul l'arbre enraciné en i_1 a été modifié.

  Pour $i = i_1$, deux cas sont alors à distinguer:

  1er cas: $H_{j-1}.(i_1) > H_{j-1}.(i_2)$.
  On a alors $H_j.(i_1) = H_{j-1}.(i_1)$ et $T_j.(i_1) = T_{j-1}.(i_1) + T_{j-1}.(i_2)$, d'où:

  $$T_j.(i_1) \geq T_{j-1}.(i_1) \geq 2^{H_{j-1}.(i_1)} = 2^{H_j.(i_1)}.$$

  Second cas: $H_{j-1}.(i_1) = H_{j-1}.(i_2)$.
  On a cette fois $H_j.(i_1) = H_{j-1}.(i_1) + 1$ et toujours $T_j.(i_1) = T_{j-1}.(i_1) + T_{j-1}.(i_2)$, d'où:

  $$T_j.(i_1) = T_{j-1}.(i_1) + T_{j-1}.(i_2) \geq 2^{H_{j-1}.(i_1)} + 2^{H_{j-1}.(i_2)} = 2 \times 2^{H_{j-1}.(i_1)} = 2^{H_j.(i_1)}.$$

  ce qui achève la preuve.

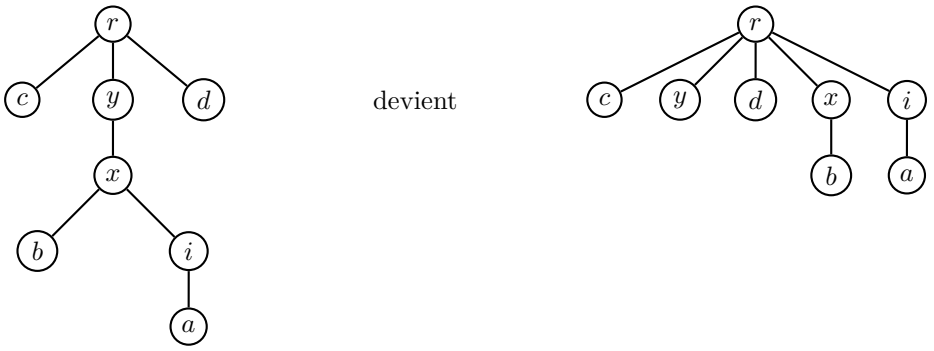
Avec cette implémentation de l'algorithme de Kruskal, les calculs de `composante c i` et `composante c j` se font en temps de l'ordre de `hauteur.(i)` et `hauteur.(j)`, soit en $O(\log n)$ (puisque la taille des arbres ne peut dépasser n). La fusion (dans le cas où les deux composantes connexes sont distinctes) se fait ensuite en temps constant, ce qui donne un temps de calcul total en $O(m \log n)$. Ainsi, le temps de calcul est du même

ordre de grandeur pour le tri des arêtes et pour appliquer l'algorithme de la question 6: comme expliqué dans l'énoncé, on ne pourra espérer améliorer ce temps de calcul que dans des cas où le tri des arêtes pourra être accéléré.

21 On initialise un tableau `compteur` de taille k ne contenant que des 0, qui va permettre de compter le nombre d'occurrence de chaque valeur $j \in \{0, \dots, k-1\}$ dans le vecteur `t`, en effectuant un simple parcours de `t`. Il reste ensuite à recopier dans `t` les valeurs $0, 1, \dots, k-1$ en respectant les nombres d'occurrences.

```
let tri_lineaire k t =
   let compteur = make_vect k 0 in
   let n = vect_length t in
   for i=0 to n-1 do
      compteur.(t.(i)) <- compteur.(t.(i))+1
   done;
   let indice = ref 0 in
   for j=0 to k-1 do
      for i=1 to compteur.(j) do
         t.(!index) <- j;
         incr indice
      done;
   done;;
```

22 Cette fonction renvoie le représentant r de la composante connexe de i, en remontant dans l'arbre de racine r tous les sommets appartenant au chemin qui relie r à i, qui deviennent fils directs de r. Voici un exemple de fonctionnement de cette fonction:



Partie V - Coupe minimum

23 Il ne faut surtout pas faire une preuve par récurrence! Supposons donc qu'à un instant donné, T contienne i composantes connexes $C_1, \dots, C_i$ et notons E_i l'ensemble des arêtes de G dont les extrémités sont dans des composantes connexes différentes. Pour j, k éléments distincts de $\{1, \dots, i\}$, notons $E_{j,k}$ l'ensemble des arêtes de G donc l'une des extrémités est dans C_j et l'autre dans C_k: E_i est donc la réunion disjointe des $E_{j,k}$ pour $\{j, k\}$ décrivant toutes les parties de $\{1, \dots, i\}$ de cardinal 2.

Pour tout $j \in \{1, \dots, i\}$, $\delta(C_j)$ est la réunion disjointe des $E_{j,k}$ pour k décrivant $\{1, \dots, i\} \setminus \{j\}$. Par définition

de k, on en déduit:

$$\forall j \in \{1,\dots,i\},\ k \leq |\delta(C_j)| = \sum_{k\neq j} |E_{j,k}|$$

En sommant ces inégalités, nous obtenons:

$$ik \leq \sum_{j=1}^{i} \left(\sum_{k\neq j} |E_{j,k}| \right) = \sum_{\substack{1\leq j,k\leq i \\ j\neq k}} |E_{j,k}| = 2 \sum_{1\leq j<k\leq i} |E_{j,k}| = 2|E_i|$$

Ceci signifie donc qu'au moins $\dfrac{ik}{2}$ arêtes relient des composantes connexes différentes.

24 L'arête ajoutée pour regrouper deux composantes connexes est l'une des arêtes de E_i. Comme $|\delta(Y)|$ est de cardinal k, la probabilité p_i que l'une de ces arêtes ait été choisie est $\dfrac{|E_i \cap \delta(Y)|}{|E_i|}$. En effet, on peut admettre comme évident qu'en choisissant avec une probabilité uniforme les arêtes les unes après les autres parmi les arêtes non encore traitées, la prochaine arête ajoutée à l'arbre est une variable aléatoire qui suit une loi uniforme sur E_i. Nous obtenons donc:

$$p_i \leq \frac{k}{|E_i|} \leq \frac{k}{ik/2} = \frac{2}{i}.$$

25 Notons $T_0, T_1, \dots, T_{n-2}$ les arbres obtenus par ajouts successifs des arêtes $e_1, \dots, e_{n-2}$. Si aucune des e_i n'est élément de $\delta(Y)$, chaque composante connexe de chaque T_i est soit contenue dans Y, soit contenue dans $V \setminus Y$. En effet, cette propriété est vérifiée par T_0, dont les composantes connexes sont des singletons, et se montre ensuite par récurrence puisque l'ajout de e_i fusionnera deux composantes contenus dans Y ou deux composantes connexes contenus dans $V \setminus Y$. Comme T_{n-2} n'a que deux composantes connexes X et $V \setminus X$, on a nécessairement $X = Y$ ou $X = V \setminus Y$ et $\delta(X) = \delta(Y)$. La probabilité d'obtenir $\delta(X) = \delta(Y)$ est donc au moins égale à la probabilité de ne jamais choisir une arête dans $\delta(Y)$. En n'y regardant pas de trop près, nous obtenons donc:

$$\mathbf{P}\Big([\delta(X) = \delta(Y)]\Big) \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\dots\left(1 - \frac{2}{3}\right) = \frac{2}{n(n-1)}.$$

26 Notons X_i la variable aléatoire qui prend la valeur 0 si la i-ème coupe calculée est minimale et 1 sinon. En supposant que les permutations sont indépendantes, $(X_i)_{1\leq i\leq t}$ est un schéma de Bernoulli de paramètre $p \leq \dfrac{2}{n(n-1)}$. La probabilité p_0 qu'aucune des coupes calculées ne soit minimale est donc:

$$p_0 = p^t \leq \left(\frac{2}{n(n-1)}\right)^t = \exp\left(t \ln \frac{2}{n(n-1)}\right) \leq \exp\left(-\frac{2t}{n(n-1)}\right)$$

en remarquant que $u + \ln u \leq 0$ pour $0 < u < 1/3$ (nous supposerons donc que $n \geq 3$).

Pour obtenir une coupe minimale avec une probabilité au moins égale à $1 - e^{-10}$, il suffit de construire $t = 5n(n-1)$ permutations des arêtes aléatoires (sous-entendu suivant le loi uniforme) et indépendantes $\sigma_1, \dots, \sigma_t$: pour chacun des σ_j, on applique l'algorithme 6 en s'arrêtant après avoir ajouté une $(n-2)$-ième arête: on définit ainsi une partie X_j dont on calcule le cardinal de la coupe $k_j = \delta(X_j)$. Pour cela, il suffit de terminer le parcours de la liste des arêtes ordonnées par σ_j et de compter celles dont les extrémités sont dans des composantes connexes distinctes. On retient ensuite le X_j donnant un k_j minimal: la probabilité d'obtenir une coupe minimale de G est au moins égale à $1 - \exp\left(-\frac{2t}{n(n-1)}\right) = 1 - e^{-10}$.

Nous pouvons écrire l'algorithme en pseudo-code, le vecteur X servant à stocker le X_j minimisant k_j: la case $X.(x)$ contient le booléen `true` si $x \in X_j$ et `false` sinon. Par convention, X_j est la classe du sommet 0.

Algorithme de Monte-Carlo pour la recherche d'une coupe minimale

$K \leftarrow m+1$ et $X = [|true; true; \ldots; true|]$
pour j de 1 à $5n(n-1)$ **faire**
  engendrer une permutation $(e_1, \ldots e_m)$ de E
  $k \leftarrow 0$ et $i \leftarrow 0$
  initialiser $Parent$ et $Hauteur$
  tant que $k \neq n-2$ **faire**
    si les extrémités de e_i sont dans des composantes connexes différentes **alors**
      fusionner les classes des extrémités de e_i
      $k \leftarrow k+1$
    fin si
    $i \leftarrow i+1$
  fin tant que
  $K' \leftarrow 0$ (* on calcule k_j *)
  tant que $i \leq m$ **faire**
    si les extrémités de e_i sont dans des composantes connexes différentes **alors**
      $K' \leftarrow K'+1$
    fin si
    $i \leftarrow i+1$
  fin tant que
  si $K' < K$ **alors**
    $K \leftarrow K'$ (* on a trouvé une meilleure coupe et on copie X_j dans X *)
    $a \leftarrow$ représentant de la classe de 0
    `pour` x variant de 1 à $n-1$ **faire**
      $X.(x) \leftarrow (a =$ représentant de la classe de $x)$
    fin pour
  fin si
fin pour
renvoyer K et X

Le temps de calcul pour chaque permutation est $O(m) + O(m \log n) + O(n \ln n)$ (temps pour créer la permutation, pour parcourir toutes les permutations (avec des appels à `composante` en temps $O(n)$) puis temps de la recopie éventuelle de X_i dans X), soit $O(m \log n)$ puisque $n = O(m)$ car le graphe est connexe.

On obtient donc une coupe minimale avec la probabilité $1 - e^{-10}$ en un temps $O(mn^2 \log n)$.