

Option Informatique en Spé MP et MP*

Arbres recouvrants minimaux : le corrigé

Quelques questions d'ordre général

Question 1 • Si le graphe donné est acyclique, c'est fini; sinon, on peut enlever une arête d'un cycle, tout en conservant la connexité. On répète le procédé jusqu'à l'obtention d'un graphe cyclique (ce qui est inévitable puisque le nombre d'arêtes est fini). Ceci assure l'existence d'un arbre recouvrant. Comme le nombre de graphes partiels est fini, le nombre d'arbres recouvrants est lui-même fini, et il en existe donc un de poids minimal.

Question 2 • Raisonnons par l'absurde: soient $a_1 < a_2 < \dots < a_{n-1}$ et $b_1 < b_2 < \dots < b_{n-1}$ les poids des arêtes de deux arbres recouvrants minimaux, classés par valeurs croissantes. Soit i le plus petit indice tel que $a_i \neq b_i$, supposons $a_i < b_i$ pour fixer les idées. Ajoutons l'arête de poids a_i au deuxième arbre: un cycle apparaît. L'une au moins des arêtes de ce cycle est de poids supérieur à a_i (sinon le premier arbre contiendrait ce cycle); en l'enlevant, on obtient un nouvel arbre recouvrant, de poids strictement inférieur aux précédents.

Question 3 • Comme le nombre d'arêtes d'un arbre recouvrant est $n - 1$, le fait d'ajouter une même constante au poids de chaque arête ne change pas le fait, pour un arbre recouvrant, d'être minimal. Il suffit donc d'ajouter comme constante $1 - \min\{\pi(a) \mid a \in A\}$ pour rendre au moins égal à 1 le poids de chaque arête.

Question 4 • Considérons un graphe dont les sommets sont ceux d'un rectangle $ABCD$, les arêtes sont ses côtés, le poids de chaque arête est sa longueur. Supposons $AB = \ell < BC = L$ pour fixer les idées; si Maurice choisit $S_1 = \{B, C\}$ et $S_2 = \{A, D\}$ le poids de l'arbre recouvrant qu'il obtiendra sera $2L + \ell$, alors que le poids d'un arbre recouvrant minimal est $2\ell + L$.

L'algorithme de Kruskal

Question 5 • Considérons un arbre recouvrant minimal T dans lequel l'arête $\{x, y\}$ n'apparaît pas. Ajoutons cette arête à T : un cycle apparaît, auquel l'arête $\{x, y\}$ appartient nécessairement. Ôtons une autre arête de ce cycle: on obtient un arbre recouvrant, dont le poids est à la fois majoré et minoré par celui de T , donc un arbre recouvrant minimal.

Question 6 • Soient $u \in S_1$ et $v \in S \setminus S_1$. Dans l'arbre recouvrant, il existe un chemin $(x_0 = u, x_1, \dots, x_p = v)$ reliant u à v ; soit i le premier indice tel que $x_i \in S_1$ et $x_{i+1} \notin S_1$. Alors $x = x_i$ et $y = x_{i+1}$ conviennent.

Question 7 • Le graphe G étant fini, l'ensemble des arêtes qui peuvent convenir dans la question précédente est lui-même fini. Choisissons a de poids minimal, parmi les arêtes qui conviennent; l'argument utilisé à la question 5 peut être utilisé à nouveau, en considérant un arbre recouvrant minimal dont G_1 est un sous-graphe, mais dont G_2 n'est pas un sous-graphe.

Question 8 • On prend en compte les arêtes par poids croissants; on rejette toute arête qui ferait perdre au graphe en cours de construction son caractère acyclique; on retient les autres arêtes, à concurrence de $n - 1$ (ce qui est possible d'après 5); on a alors obtenu un arbre recouvrant. Il est minimal car, si l'on remplace l'une de ses arêtes par une autre de poids inférieur, on fait apparaître un cycle, en raison même de la méthode de sélection des arêtes.

Question 9 • On retient les arêtes $\{b, c\}$ (de poids 6) et $\{a, c\}$ (de poids 8); on rejette l'arête $\{a, b\}$ qui ferait apparaître un cycle; on retient les arêtes $\{g, h\}$ (de poids 12), $\{b, e\}$ (de poids 23) $\{d, f\}$ (de poids 24) et $\{c, d\}$ (de poids 25); on rejette l'arête $\{a, f\}$; on retient l'arête $\{e, h\}$ (de poids 27). À ce moment, on a un arbre recouvrant minimal, de poids 125.

Question 10 • Multiplier les poids par -1 avant d'appliquer l'algorithme précédent; ou, si l'on préfère, choisir à chaque étape l'arête de poids maximal parmi celles qui ne font pas apparaître de cycle.

Question 11 • G étant connexe, l'algorithme proposé préserve la connexité puisqu'il n'élimine une arête que si elle participe à un cycle. Lorsque l'algorithme s'arrête, le graphe est acyclique: c'est donc un arbre recouvrant, et il est minimal car l'opération effectuée à chaque étape conserve la propriété «le graphe contient un arbre recouvrant minimal».

Mise en œuvre naïve de l'algorithme de Kruskal

Question 12 • Au départ, chaque élément est seul dans sa classe: $n - 1$. D'où:

```
let init repres = for i=0 to vect_length repres - 1 do repres.(i) <- i done;;
```

Question 13 • Notons que le terme «fusion» n'est pas adapté puisque les deux classes ne jouent pas, en fait des rôles symétriques... On décide de faire passer tous les membres de la classe de y dans la classe de x :

```

let fusion repres x y =
  let rx = repres.(x) and ry = repres.(y) in
  for i = 0 to vect_length repres - 1 do
    if repres.(i) = ry then repres.(i) <- rx
  done;;

```

Le coût est au moins de $n + 3$ (initialisations de `rx` et `ry`, examen de chaque élément de `repres`, modification de `repres.(y)`) et au maximum de $2n + 1$ (si la classe de `y` compte $n - 1$ membres).

Question 14 • L'arête $\{x, y\}$ ne doit être ajoutée que si x et y sont dans des classes distinctes. On sait que le nombre d'arêtes à obtenir est $n - 1$. D'où :

```

(* calcule l'ordre de graphe *)
let rec ordre = fonction
  | [(x,y,_)] -> max x y
  | (x,y,_)::q -> max (max x y) (ordre q);;

(* applique l'algorithme de Kruskal *)
let mst_kruskal un_graphe =
  let n = ordre un_graphe in
  let repres = make_vect n 0 in
  init repres;
  let rec mst_aux p liste_ori liste_mst =
    if p = 0 then liste_mst else match liste_ori with
    | [] -> failwith "graphe non connexe !"
    | (x,y,_)::q -> if repres.(x-1) = repres.(y-1)
      then (mst_aux p q liste_mst)
      else (fusion repres (x-1) (y-1); mst_aux (p-1) q ((x,y)::liste_mst))
  in mst_aux (n-1) un_graphe [];;

```

Voici une mise en œuvre avec le graphe de l'énoncé :

```

(* quelques raccourcis commodes *)
let pc = print_char and ps = print_string and pnl = print_newline;;

(* le graphe de l'énoncé *)
let mon_graphe = [(2,3,6);(1,3,8);(1,2,10);(7,8,12);(2,5,23);
  (4,6,24);(3,4,25);(1,6,26);(5,8,27);(6,7,28);(4,5,35)];;

(* impression d'une arête *)
let print_arête x y = let cc u = pc(char_of_int(u + int_of_char 'a' - 1)) in
  ps "("; cc x; ps ","; cc y; ps ")";;

(* impression de la liste des arêtes par poids croissants *)
let rec pliste = fonction
  | [] -> ()
  | (x,y)::q -> pliste q; print_arête x y;;

(* et c'est parti ! *)
pliste (mst_kruskal mon_graphe);;

```

Question 15 • Chaque arête retenue requiert une fusion, de coût $\Omega(n)$; comme il faut $n - 1$ arêtes, le coût est $\Omega(n^2)$; ceci inclut la lecture de la liste des arêtes du graphe original, qui peut en compter jusqu'à $\frac{n(n-1)}{2}$.

Mise en œuvre de l'algorithme de Kruskal avec la méthode *union-find*

Question 16 • Comme à la question 12, chaque élément est initialement seul dans sa classe.

```

let init repres card classe =
  for i=0 to vect_length repres - 1 do
    repres.(i) <- i; card.(i) <- 1; classe.(i) <- [i]
  done;;

```

Question 17 • On détermine d'abord quelle est, des deux classes, celle de plus petit cardinal ; on agit alors sur `repres` pour transférer tous ses membres dans l'autre classe ; il ne reste plus qu'à mettre à jour les vecteurs `card` et `classe`. Noter que, lors de la concaténation des listes, on a mis la plus courte *devant* l'autre.

```
(* fait passer les éléments d'une classe à une autre *)
let rec transfert repres clx y = match clx with
| [] -> ()
| t::q -> repres.(t) <- y; transfert repres q y;;

(* rattache la classe de x à celle de y *)
let rattache repres card classe x y =
  card.(y) <- card.(y) + card.(x);
  transfert repres classe.(x) y;
  classe.(y) <- classe.(x) @ classe.(y);;

(* fusionne les classes de x et y *)
let fusion repres card classe x y =
  let cx = card.(x) and cy = card.(y) in
  if cx < cy then rattache repres card classe x y
  else rattache repres card classe y x;
```

Le coût est proportionnel à l'effectif de la plus petite des deux classes. Voici une mise en œuvre complète :

```
let mst_kruskal un_graphe =
  let n = ordre un_graphe in
  let repres = make_vect n 0 and card = make_vect n 0 and classe = make_vect n [] in
  init repres card classe;
  let rec mst_aux p liste_ori liste_mst =
    if p = 0 then liste_mst else match liste_ori with
    | [] -> failwith "graphe non connexe !"
    | (x,y,_)::q -> if repres.(x-1) = repres.(y-1)
      then (mst_aux p q liste_mst )
      else (fusion repres card classe repres.(x-1) repres.(y-1);
        mst_aux (p-1) q ((x,y)::liste_mst) )
  in mst_aux (n-1) un_graphe [];
```

Question 18 • Observons un élément fixé i : à chaque fois que la taille de sa classe augmente, c'est d'un facteur au moins deux. Donc i constate au plus $\lceil \lg n \rceil$ fusions ; le coût spécifique associé à i lors d'une fusion est compris entre 0 (si i appartient à la liste longue) et 2 (s'il appartient à la liste courte). Donc les n éléments interviennent pour un coût total $O(n \lg n)$; d'autre part, il y a en tout $n - 1$ opérations de fusion, entraînant chacune un coût fixe constant ; la contribution totale $O(n)$ est négligeable devant la précédente.

L'algorithme de Prim

Question 19 • Soit T un arbre recouvrant minimal ne contenant pas l'arête $\{x, y\}$; ajoutons cette arête à T , un cycle apparaît ; on peut supposer que $\{x, y\}$ est la première arête de ce cycle. Alors la dernière, $\{z, x\}$ a un poids au moins égal à celui de $\{x, y\}$; en enlevant $\{z, x\}$ on retrouve un arbre recouvrant, dont le poids est à la fois majoré et minoré par celui de T .

Question 20 • L'argumentation est la même qu'aux questions 6 et 7.

Question 21 • Supposons que a ne fasse pas partie de T ; ajoutons cette arête à T , un cycle apparaît, auquel a appartient nécessairement. Décrivons ce cycle, en partant du sommet u , et en empruntant d'abord l'arête a , laquelle fait «sortir» de S_1 ; on finira par «rentrer» dans S_1 en empruntant une arête $a' = \{v', u'\}$ avec $u' \in S_1$ et $v' \notin S_1$. Le choix de a fait que $\pi(a') \geq \pi(a)$. Enlevons l'arête a' : le cycle disparaît, sans que la connexité soit perdue. On obtient donc un nouvel arbre recouvrant T' , qui contient (S_2, A_2) comme sous-graphe, et dont le poids est au plus égal à celui de T ; comme T était minimal, il en est de même de T' .

Question 22 • À la k -ième étape de l'algorithme, on part d'un arbre T_k recouvrant minimal à k sommets ; on cherche un sommet v n'appartenant pas à T_k , voisin d'un sommet u de T_k , l'arête reliant u et v étant de poids minimal. La question 19 montre que l'on peut prendre n'importe quel sommet pour T_1 ; les questions 20 et 21 montrent que, tant que T_k n'est pas recouvrant, on peut effectivement trouver v et une arête a pour obtenir T_{k+1} . Comme le graphe G est fini (à n sommets), au bout de $n - 1$ étapes le processus se termine.

Question 23 • L'algorithme retient successivement les arêtes (a, c) , (c, b) , (b, e) , (c, d) , (d, f) , (e, h) et (h, g) .

Question 24 • On réalise un parcours en largeur de l'arbre recouvrant dont la racine est le sommet initial.

Matroïdes

Question 25 • Ici, E est l'ensemble des colonnes de M , et les membres de \mathcal{I} sont les familles libres (au sens de l'algèbre linéaire). La propriété **1** est claire; voyons la propriété **2**: soient c_1, \dots, c_p les éléments de X et d_1, \dots, d_q ceux de Y , avec $p < q$. Si chaque élément de Y appartenait à $\text{Vect}(X)$, la famille Y serait liée (cf. « $p + 1$ combinaisons linéaires de p vecteurs sont liées»); soit $d_j \notin \text{Vect}(X)$; la famille $X \cup \{d_j\}$ est libre.

Question 26 • Ici, $E = A$ et les membres de \mathcal{I} sont les parties de A qui définissent (avec S comme ensemble de sommets) un graphe acyclique. La propriété **1** est claire. Soient X et Y deux parties de A définissant des graphes partiels acycliques, avec $|X| < |Y|$. S'il existe un sommet atteint par Y mais pas par X , il suffit de prendre dans Y une arête incidente à ce sommet. Sinon, X compte une composante connexe de plus que Y ; en effet, notant n_s le nombre de sommets et n_a le nombre d'arêtes, chaque composante connexe (qui est un arbre) K vérifie $n_s(K) = n_a(K) + 1$; donc, pour un graphe G ayant $p(G)$ composante connexes: $n_s(G) = n_a(G) + p(G)$. Or $n_a(Y) > n_a(X)$, $n_s(Y) \leq n_s(X)$ si bien que $p(X) = n_s(X) - n_a(X) > n_s(Y) - n_a(Y) = p(Y)$. De par le principe des tiroirs, il existe au moins une composante connexe de Y qui rencontre deux composantes connexes de X ; considérant alors un chemin dans Y reliant deux sommets situés chacun dans une de ces composantes de X , on met en évidence (comme à la question 6) une arête appartenant à Y , reliant un sommet incident à une arête de X à un autre sommet qui n'est incident à aucune arête de X ; cette arête peut donc être ajoutée à X sans lui faire perdre son caractère acyclique.

Question 27 • Soient X et Y deux parties libres, maximales pour l'inclusion, de cardinaux distincts; supposons $|X| < |Y|$ pour fixer les idées. La propriété **2** affirme l'existence de $x \in Y \setminus X$ tel que $X \cup \{x\}$ soit encore libre, contredisant l'hypothèse de maximalité faite sur X .

Question 28 • Comme il existe au moins une partie libre, et que E est fini, il existe au moins une partie F optimale de E . Notons $n = |F|$. Supposons $x \notin F$, et considérons la famille des parties libres de la forme $\{x\} \cup G$, où G est une partie de F . Il en existe au moins une (prendre pour G l'ensemble vide); et, de par la propriété **2**, s'il existe une telle partie de cardinal $p < n$, il en existe aussi une de cardinal $p + 1$. Donc il existe dans cette famille une partie libre H de cardinal n : H contient x , et tous les éléments de F sauf un, y . H ayant même cardinal que F , est maximale pour l'inclusion; comme $\pi(x) \leq \pi(y)$, $\pi(H) \leq \pi(F)$ ce qui montre que H est optimale.

Question 29 • Par construction, la partie énumérée par la liste que rend `glouton` est libre; de plus, elle est maximale: s'il était possible d'ajouter un élément à cette partie en conservant sa liberté, il aurait été possible de le faire lorsque cet élément a été examiné (ceci de par la propriété **1**). Enfin, cette partie est optimale parce que la liste qui énumère E est classée par poids croissants.

Question 30 • Il suffit de prendre pour `est_libre` une fonction qui, appliquée à une liste d'arêtes, détermine si le graphe représenté par cette liste est acyclique.

FIN