Option Informatique en Spé MP et MP*
Devoir surveillé du 19 mars 2003 : le corrigé

Programmation

Question 1 • Voici deux versions, l'une utilisant un parcours explicite de la liste, l'autre faisant appel à la fonction `for_all` :

```
let rec est_constante = function
 | [] | [_] -> true
 | x::y::q -> x=y & est_constante(y::q) ;;

let est_constante = function
 | [] -> true
 | t::q -> for_all (prefix = t) q ;;
```

Question 2 • Pas de remarque particulière.

```
let rec est_croissante = function
 | [] | [_] -> true
 | x::y::q -> x<=y & est_croissante(y::q) ;;
```

Question 3 • On écrit `est_decroissante` sur le même modèle, puis :

```
let est_monotone s = est_croissante s or est_decroissante s ;;
```

Question 4 • Si la liste n'est pas croissante, il faut effectuer un deuxième parcours.

Question 5 • Première méthode : on parcourt le palier de tête ; dès que l'on rencontre deux éléments consécutifs distincts, on vérifie que la queue de la liste est croissante ou décroissante, selon ce que l'on a trouvé.

```
let est_monotone = function
 | [] | [_] -> true
 | x::y::q when x<y -> est_croissante(y::q)
 | x::y::q when x>y -> est_decroissante(y::q)
 | _::q -> est_monotone q ;;
```

Deuxième méthode : on utilise deux drapeaux, initialement levés ; le drapeau `up` reste levé tant que l'on n'a pas rencontré deux éléments consécutifs, en ordre croissant. À la fin du parcours, la liste est monotone, sauf si les deux drapeaux ont été baissés.

```
let est_monotone s =
 let rec aux up dn = function
  | [] | [_] -> up or dn
  | x::y::q when x<y -> aux false dn (y::q)
  | x::y::q when x>y -> aux up false (y::q)
  | _::q -> aux up dn q
 in aux true true s ;;
```

Question 6 • Une fois réglés les cas particuliers où l'une des listes est vide, il suffit de remarquer que la liste $(x_1, \dots, x_p)$ est extraite de la liste $(y_1, \dots, y_q)$ ssi l'une des deux conditions suivantes est satisfaite :

- $x_1 = y_1$ et $(x_2, \dots, x_p)$ est extraite de $(y_2, \dots, y_q)$;
- $x_1 \neq y_1$ et $(x_1, \dots, x_p)$ est extraite de $(y_2, \dots, y_q)$.

```
let rec est_extraite_de = function
  | ([],_) -> true
  | (_,[]) -> false
  | (t::q,t'::q') when t = t' -> est_extraite_de (q,q')
  | (u,_::q') -> est_extraite_de (u,q') ;;
```

Arbre recouvrant minimal

Cette partie est extraite d'un sujet posé aux étudiants de MP1 et MP le 18 mars 1997.*

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 $a_1 < a_2 < \ldots < a_{n-1}$ et $b_1 < b_2 < \ldots < b_{n-1}$ 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 $(x_0 = u, x_1, \ldots, x_p = v)$ 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'arête, 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≫.

L'algorithme de Prim

Question 12 • 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 13 • L'argumentation est la même qu'aux questions 6 et 7.

Question 14 • 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 $\pi(a') \geqslant \pi(a)$. 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'.

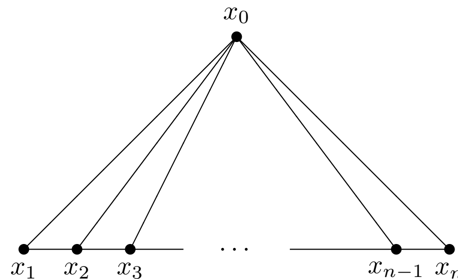
Question 15 • À la k-ième étape de l'algorithme, on part d'un arbre T_k recouvrant minimal à k sommets ; on cherche un sommet v n'appartenant pas à T_k, voisin d'un sommet u de T_k, l'arête reliant u et v étant de poids minimal. La question 12 montre que l'on peut prendre n'importe quel sommet pour T_1 ; les questions 13 et 14 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 16 • L'algorithme retient successivement les arêtes (a, c), (c, b), (b, e), (c, d), (d, f), (e, h) et (h, g).

Question 17 • On réalise un parcours en largeur de l'arbre recouvrant dont la racine est le sommet initial.

Petite question complémentaire

Comptez les arbres recouvants du graphe à $n + 1$ sommets ci-dessous :



FIN