Centrale Informatique optionnelle MP 2015
Corrigé

Ce corrigé est proposé par Benjamin Monmege (Enseignant-chercheur à l'université) ; il a été relu par Charles-Pierre Astolfi (ENS Cachan) et Guillaume Batog (Professeur en CPGE).

Ce sujet consiste en l'étude d'un problème d'allocation de ressources. Un ensemble de tâches (cours, avion en phase d'atterrissage, requête d'un client informatique, etc.) est modélisé par un ensemble d'intervalles correspondant au temps nécessaire à la réalisation de chacune d'elles. Il s'agit alors d'allouer une ressource (salle, piste d'atterrissage, serveur informatique, etc.) à chaque tâche de sorte qu'une ressource ne soit jamais utilisée au même instant par deux tâches différentes. On cherche par ailleurs à minimiser le nombre de ressources nécessaires. Au long de quatre parties fortement dépendantes, le sujet propose une solution grâce à l'étude de colorations de graphes d'intervalles.

- La partie I introduit les graphes d'intervalles, qui permettent de représenter à l'aide de graphes non orientés les contraintes d'intervalles d'un problème d'allocation de ressources. Une allocation de ressources consiste alors en une coloration du graphe, c'est-à-dire à l'étiquetage de chaque sommet par une couleur de telle sorte que les sommets voisins ne soient jamais étiquetés avec la même couleur. Un lien est établi entre le nombre chromatique d'un graphe, c'est-à-dire le nombre minimal de couleurs nécessaire pour le colorier, et la taille de la plus grande clique (sous-graphe complet) du graphe. Certaines fonctions utiles pour la suite du sujet sont également implémentées.
- La partie II propose un algorithme glouton pour colorier un graphe d'intervalles. Son implémentation, la preuve de sa correction et sa complexité sont étudiées.
- La partie III démontre que l'algorithme glouton précédent reste correct dans le cas plus général de graphes munis d'un ordre d'élimination parfait, c'est-à-dire une énumération de leurs sommets telle que, pour chaque sommet, ses voisins énumérés avant lui forment une clique.
- La partie IV étudie finalement une condition suffisante pour qu'un graphe possède un ordre d'élimination parfait, à savoir que le graphe soit cordal : un graphe est cordal si tout cycle de longueur supérieure à quatre possède une corde. Les graphes cordaux sont couramment appelés graphes triangulés. Après avoir montré que tout graphe d'intervalles est cordal, le sujet demande de résoudre une énigme policière à l'aide de cet outil. Le reste du sujet implémente la recherche d'ordres d'élimination parfaits lorsqu'ils existent, puis prouve que tout graphe cordal possède un ordre d'élimination parfait.

Les parties I à III du sujet sont des applications directes du cours, où l'on manipule des graphes représentés par des listes d'adjacence. L'énigme policière, ainsi que les questions d'implémentation et de complexité de la partie IV, sont nettement plus difficiles et discriminantes : elles permettent d'évaluer la créativité et la compréhension globale des parties précédentes. Enfin, les trois dernières sections de la partie IV proposent une preuve élégante de théorie des graphes.

Indications

I.C.2 Utiliser la fonction `conflit` de la question I.A.1 afin de construire itérativement le tableau de listes de voisins de $G(I_0, \dots, I_{n-1})$.

I.E.2 Montrer que $\omega(G) \leqslant \chi(G)$.

I.E.3 Remarquer que tester si un ensemble $\{x_0, \dots, x_{p-1}\} \subset S$ de sommets est une clique de G consiste à vérifier que, pour tout $0 \leqslant q < p$, tous les sommets de $\{x_{q+1}, \dots, x_{p-1}\}$ sont voisins avec x_q. Utiliser la fonction `appartient` de la question I.D.2.a.

II.B Employer la fonction `couleur_disponible` de la question I.D.2.d.

II.C.1 Les intervalles sont énumérés dans l'ordre croissant de leurs extrémités gauches.

II.C.3 Utiliser l'inégalité trouvée à la question I.E.2.

II.D Déterminer dans un premier temps la complexité des fonctions des questions de la partie I.D.2.

III.B.2 Utiliser la fonction `est_clique` de la question I.E.3 ainsi que la fonction `voisins_inferieurs` de la question III.B.1.

III.C Appliquer le résultat de la question II.C.2.

III.D.3 Remarquer la ressemblance de cette partie avec la partie II.C.

IV.A.2 Grâce à la question I.A.1, expliciter le fait que I_1 et I_2 ont une intersection non vide et que I_0 et I_2 ont une intersection vide.

IV.A.3 Utiliser la caractérisation de la question I.A.1 pour contredire le fait que $I_0 \cap I_3 \neq \varnothing$.

IV.B Procéder par récurrence, en remarquant qu'un cycle de longueur $n+1 \geqslant 5$ dans un graphe d'intervalles peut se réduire en un cycle de longueur n dans un graphe d'intervalles obtenu en fusionnant deux sommets.

IV.C Construire le graphe d'intervalles G résumant les informations données et noter qu'il n'est pas cordal. En sachant que seul le coupable a menti, la suppression d'un seul sommet de G le rend cordal.

IV.D.1 Utiliser la fonction `est_clique` de la question I.E.3, puis montrer qu'elle teste si l'ensemble représenté par la liste `xs` de longueur k est une clique d'un graphe à m arêtes avec une complexité $O(1 + km)$.

IV.D.3 Employer une approche gloutonne qui consiste à éliminer itérativement des sommets simpliciaux en utilisant la fonction `trouver_simplicial` de la question IV.D.2. Justifier que l'algorithme est correct en démontrant que si un graphe $G = (S, A)$ possède un ordre d'élimination parfait, alors pour tout sommet simplicial x dans G, le graphe induit par $S \setminus \{x\}$ possède un ordre d'élimination parfait.

IV.E.1 En notant C_1 l'ensemble des sommets de C qui n'ont pas de voisin dans S_1, montrer que $C \setminus C_1$ est une coupure. Déduire de la minimalité de C que $C_1 = \varnothing$.

IV.E.2 Profiter du fait que G_1 et G_2 sont des composantes connexes de G.

IV.E.3 Dans le cycle formé à partir des chemins P_1 et P_2, où se trouvent les cordes ?

IV.F.3.c Utiliser le résultat de la question IV.F.3.a pour appliquer l'hypothèse $\mathcal{P}(H_1)$. Conclure grâce à la question IV.E.4.

IV.G S'inspirer du raisonnement de la question IV.D.3 et utiliser le résultat de la partie IV.F.

I. Graphes d'intervalles

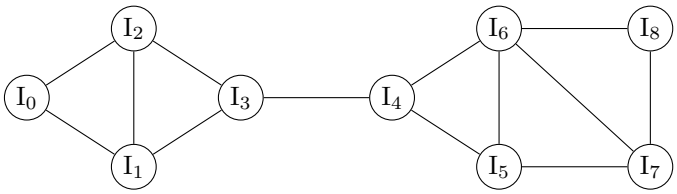
I.A.1 Soient $I = [a\,;b]$ et $J = [c\,;d]$ deux intervalles. S'ils sont en conflit, il existe un réel x appartenant à $I \cap J$. En particulier, $a \leqslant x \leqslant b$ et $c \leqslant x \leqslant d$, ce qui implique que $a \leqslant d$ et $c \leqslant b$.

Réciproquement, supposons que $a \leqslant d$ et $c \leqslant b$. Deux cas sont alors possibles : soit $a \leqslant c$, auquel cas $c \in I \cap J$, soit $a > c$, auquel cas $a \in I \cap J$. Dans tous les cas, les intervalles I et J sont en conflit.

Les intervalles $[a\,;b]$ et $[c\,;d]$ sont donc en conflit si et seulement si $a \leqslant d$ et $c \leqslant b$. Ainsi, la fonction `conflit` se contente d'exécuter ce test.

```
let conflit (a,b) (c,d) = (a <= d) && (c <= b);;
```

I.C.1 Le graphe d'intervalles associé au problème b de la figure 1 peut se représenter de la manière suivante :

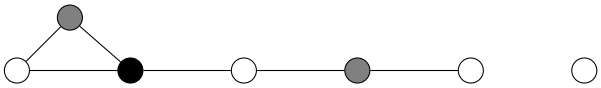


I.C.2 Afin de construire la représentation des arêtes du graphe $G(I_0, \ldots, I_{n-1})$ associé au tableau de segments $(I_0, \ldots, I_{n-1})$, parcourons les paires d'intervalles itérativement afin d'insérer une arête (i, j) si I_i et I_j sont en conflit. Un tableau `aretes` de n listes vides est initialement créé pour recevoir les listes de voisins du graphe. Une boucle sur les indices i parcourt ensuite les intervalles I_i et une boucle interne sur les indices $j \in \{0, \ldots, i-1\}$ parcourt les intervalles I_j pour concaténer j en tête de la liste courante des voisins de i, et i en tête de la liste courante des voisins de j, si I_i et I_j sont en conflit : le test de conflit est réalisé à l'aide de la fonction `conflit` décrite à la question I.A.1.

```
let construit_graphe segments =
  let n = vect_length segments in
  let aretes = make_vect n [] in
  for i=n-1 downto 0 do
    for j=i-1 downto 0 do
      if (conflit segments.(i) segments.(j))
      then begin
        aretes.(i) <- j::aretes.(i);
        aretes.(j) <- i::aretes.(j)
      end
    done;
  done;
  aretes;;
```

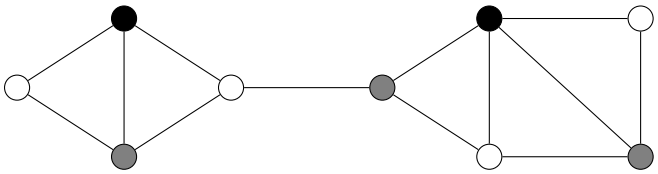
> Par souci esthétique, les indices i et j sont parcourus dans l'ordre décroissant, afin de renvoyer une liste des voisins de chaque sommet dans l'ordre croissant.

I.D.1 Une coloration du graphe d'intervalles de la figure 3, associé au problème a de la figure 1, est donnée par la suite $(0, 1, 2, 0, 1, 0, 0)$:



Toute coloration de ce graphe nécessite au moins trois couleurs puisque les sommets 0, 1 et 2 sont reliés deux à deux et ne peuvent donc pas avoir une couleur en commun.

Pour les mêmes raisons, la suite $(0, 1, 2, 0, 1, 0, 2, 1, 0)$ est une coloration optimale du graphe d'intervalles trouvé en question I.C.1, associé au problème b de la figure 1, comme reproduite ci-dessous :



I.D.2.a Le test d'appartenance d'un élément x dans une liste est réalisé à l'aide d'une fonction récursive qui parcourt la liste jusqu'à trouver x et qui conclut négativement si la fin de la liste est atteinte.

```
let rec appartient l x =
  match l with
  |[] -> false
  |a::q -> (a=x) || (appartient q x);;
```

> Il est sans doute maladroit dans cette question de répondre en utilisant la fonction de signature `mem : 'a -> 'a list -> bool` de la librairie standard de Caml telle que `mem x l` teste l'appartenance de l'élément `x` dans la liste `l`.

I.D.2.b Notons que le plus petit entier non présent dans une liste `l` de taille n est nécessairement inférieur à $n + 1$. Créons donc un vecteur `v` de taille n, initialisé à `false`, dont la case d'indice c a pour vocation de contenir `true` si et seulement si c appartient à `l`. À l'aide d'un tel vecteur, trouver le plus petit entier non présent revient à trouver l'indice de la première case du vecteur ne contenant pas `true`, donc $n + 1$ si toutes les cases contiennent `true`, ce que réalise la boucle `while` finale du programme. Construisons ainsi ce vecteur à l'aide d'une fonction auxiliaire récursive de signature `parcours : int list -> unit` telle que `parcours l` met à jour le vecteur `v` en fonction des entiers contenus dans `l`.

```
let plus_petit_absent l =
  let n = list_length l in
  let v = make_vect n false in
  let rec parcours = function
    |[] -> ()
    |c::q -> if (c<n) then v.(c) <- true; parcours q
  in parcours l;
     let c = ref 0 in
     while (!c<n && v.(!c)) do incr c done;
     !c;;
```