

I Graphes d'intervalles

I.A – Représentation du problème

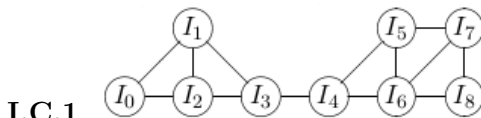
I.A.1

```
1 let conflit (a,b) (c,d) = not (b<c || d<a) ;;
```

I.B – Graphe simple non orienté

Remarquons que la représentation d'un graphe par un tableau de listes n'est pas unique si on n'impose aucun ordre sur ces listes. L'énoncé, ne suppose pas que les listes sont triées (dans l'exemple donné par l'énoncé, elles le sont cependant) et nous ne ferons aucune hypothèse de ce type. Dans la construction d'un graphe, on n'a donc pas à se préoccuper de l'ordre dans lequel on "ajoute" les arêtes.

I.C – Graphe d'intervalles



I.C.2

```
1 let construit_graphe t =
2   let n = Array.length t in
3   let g = Array.make n [] in
4   for i=0 to n-2 do
5     for j=i+1 to n-1 do
6       if conflit t.(i) t.(j) then
7         begin
8           g.(i) <- j::g.(i);
9           g.(j) <- i::g.(j)
10        end
11    done
12  done;
13  g
14  ;;
```

I.D – Coloration

I.D.1 Le graphe associé au problème (a) ne peut être colorié avec moins de trois couleurs à cause du triangle $I_0I_1I_2$. Une 3-coloration sera donc optimale si on en trouve une. On peut choisir : (0, 1, 2, 0, 1, 0, 0).

De même, on a besoin d'au moins trois couleurs pour le graphe du problème (b) (triangle $I_0I_1I_2$). Une coloration convenable est : (0, 1, 2, 0, 1, 2, 0, 1, 2).

I.D.2

```
a. 1 let rec appartient l x = match l with
2   | [] -> false
3   | t::q -> t=x || appartient q x
4   ;;
```

```

b. 1 let plus_petit_absent l =
    2   let i = ref 0 in
    3   while appartient l !i do
    4     incr i
    5   done;
    6   !i
    7 ;;

```

c. La fonction `aux : int list -> int list` prend en argument une liste de sommets et renvoie la liste des couleurs de ces sommets (s'ils sont coloriés).

```

1 let couleurs_voisins aretes couleurs i =
2   let rec aux l = match l with
3     | [] -> []
4     | j::q when couleurs.(j) = -1 -> aux q
5     | j::q -> let l_couleurs = aux q in
6                 if appartient l_couleurs couleurs.(j)
7                 then l_couleurs
8                 else couleurs.(j)::l_couleurs
9   in aux aretes.(i)
10 ;;

```

```

d. 1 let couleur_disponible aretes couleurs i =
    2   plus_petit_absent (couleurs_voisins aretes couleurs i)
    3 ;;

```

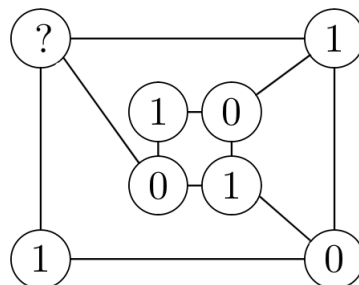
I.E – Cliques

I.E.1

- Si G ne possède pas d'arête, alors $\chi(G) = 1$ et $\omega(G) = 1$.
- Si G est complet à n sommets, alors $\chi(G) = n$ et $\omega(G) = n$.

I.E.2 Une bonne coloration de G induit une bonne coloration de tout sous-graphe de G . Ainsi, d'après la question précédente : $\omega(G) \leq \chi(G)$.

L'égalité est fautive en général. Dans le graphe suivant, il n'y a pas de triangles et donc pas de clique de taille 3. Cependant, on vérifie aisément que trois couleurs sont indispensables pour le colorier correctement. Pour le voir, on donne la couleur 0 à un sommet et on complète de manière obligatoire en n'utilisant que 0 ou 1 pour tomber sur une contradiction.



I.E.3

- `tous_voisins : int list -> int -> bool` : `tous_voisins l i` vérifie si i est voisin de tous les sommets de l ;
- `aux : int list -> bool` : vérifie que `tous_voisins xs i` est vrai pour tous les sommets i de la liste prise en entrée.

```

1  let est_clique aretes xs =
2    let rec tous_voisins l i = match l with
3      | [] -> true
4      | j::q when j = i -> tous_voisins q i
5      | j::q -> appartient aretes.(i) j && tous_voisins q i
6    in
7    let rec aux l = match l with
8      | [] -> true
9      | i::q -> tous_voisins xs i && aux q
10   in aux xs
11  ;;

```

II Graphes munis d'un ordre d'élimination parfait

II.A – Un exemple

Dans le tableau ci-dessous, on donne l'ordre dans lequel on sélectionne les sommets et, en seconde colonne, l'ensemble composé des voisins déjà choisis du dernier sommet sélectionné.

x_5	\emptyset
x_2	$\{x_5\}$
x_1	$\{x_2, x_5\}$
x_4	$\{x_1, x_2, x_5\}$
x_7	$\{x_4, x_5\}$
x_0	$\{x_1, x_4\}$
x_6	$\{x_2, x_5\}$
x_3	$\{x_2, x_6\}$

Il reste à vérifier que les ensembles de la seconde colonne forment des cliques, ce qui est le cas. Un ordre d'élimination parfait est donc : $(x_5, x_2, x_1, x_4, x_7, x_0, x_6, x_3)$.

II.B – Vérification

II.B.1 La fonction `aux` : `int list -> int list` prend en argument une liste et renvoie la liste de ces éléments qui sont $< x$.

```

1  let voisins_inferieurs aretes x =
2    let rec aux l = match l with
3      | [] -> []
4      | i::q when i < x -> i::aux q
5      | _::q -> aux q
6    in aux aretes.(x)
7  ;;

```

II.B.2

```

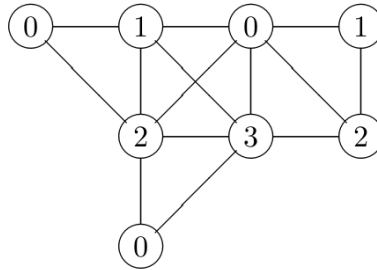
1  let est_ordre_parfait aretes =
2    let n = Array.length aretes in
3    let x = ref 1 in
4    while !x < n && est_clique aretes (voisins_inferieurs aretes !x) do
5      incr x
6    done;
7    !x = n
8  ;;

```

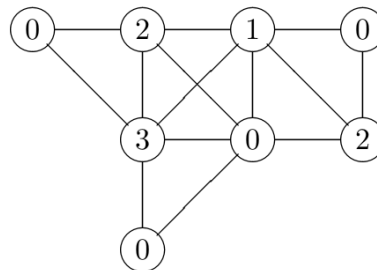
II.C – Coloration

II.C.1

a. Pour l'ordre (x_0, \dots, x_7) , on obtient la coloration suivante :



b. Pour l'ordre $(x_5, x_2, x_1, x_4, x_7, x_0, x_6, x_3)$ de la question II.A, on obtient la coloration suivante :



III Algorithme glouton pour la coloration

III.A – L'algorithme sur un exemple

En choisissant la numérotation proposée par l'énoncé, on obtient la coloration : $(0, 1, 2, 0, 1, 0, 2, 1, 0)$.

III.B – Coloration

```

1  let coloration segments =
2    let aretes = construit_graphe segments in
3    let n = Array.length segments in
4    let couleurs = Array.make n (-1) in
5    couleurs.(0) <- 0;
6    for i=1 to n-1 do
7      couleurs.(i) <- couleur_disponible aretes couleurs i
8    done;
9    couleurs
10 ;;
```

III.C – Preuve de l'algorithme

III.C.1 Puisque I_k s'est vu attribuer la couleur c , il est en conflit avec des intervalles ayant reçu les couleurs $0, \dots, c-1$. Avec l'ordre choisi sur les extrémités gauches des segments, ceci signifie que a_k est inférieur ou égal à au moins c des entiers b_0, \dots, b_{k-1} . a_k appartient donc à coup sûr à au moins c des segments I_0, \dots, I_{k-1} .

III.C.2 Considérons l'ensemble C formé des intervalles I_i tels que $i \leq k$ et $a_k \in I_i$. On vient de voir que cet ensemble est au moins de cardinal $c+1$ (il y a I_k et c autres des précédents intervalles). Il forme une clique. En effet, considérons $i < j$ des éléments tels que $I_i, I_j \in C$. On a $a_i \leq a_j \leq a_k$ puisque les intervalles sont ordonnés selon la borne inférieure. De plus $a_k \leq b_i$ (par définition de C) et donc $a_i \leq a_j \leq b_i$. Ainsi, I_i et I_j sont en conflit et donc reliés dans le graphe.

III.C.3 La question I.E.2 indique alors que le nombre chromatique est plus grand que $c+1$.

III.C.4 Quand on introduit une couleur, son numéro est toujours inférieur au nombre chromatique. Par ailleurs, la coloration partielle est toujours une bonne coloration (c'est un invariant d'itération évident puisqu'une couleur ajoutée ne contredit pas la bonne coloration). On obtient donc finalement une coloration avec un nombre de couleurs inférieur au nombre chromatique. Il est égal à $\chi(G)$ par minimalité de $\chi(G)$ et c'est une coloration optimale.

III.D – Complexité

- La fonction `appartient` est de complexité $O(p)$ où p est la longueur de la liste argument.
- La fonction `plus_petit_absent` met en œuvre une boucle effectuée au plus $p + 1$ fois où p est la longueur de la liste argument (en effet, par lemme de tiroirs, un des entiers $0, \dots, p$ est absent de la liste). Chaque itération est de complexité $O(p)$. Le coût total est donc $O(p^2)$.
- La fonction `couleurs_voisins` parcourt la liste des voisins d'un sommet. À chaque élément rencontré, on effectue $O(n)$ opérations où n est un majorant du nombre de couleurs qu'on utilisera, par exemple le nombre de sommets (c'est l'appel à `appartient` qui induit ce coût). On a donc une complexité $O(m_i n)$ où m_i est le nombre de voisins de i et n le nombre total de sommets.
- La fonction `couleur_disponible` a donc un coût $O(m_i n) + O(n)$ (comme on a formé une liste de couleurs sans doublon, on appelle `plus_petit_absent` avec une liste d'au plus n éléments) et donc $O((m_i + 1)n)$.
- La fonction `coloration` initialise un tableau pour un coût $O(n)$ où n est le nombre de sommets. On fait un appel à `couleur_disponible` pour un coût global $\sum_{i=0}^{n-1} O((m_i + 1)n) = O(mn) + O(n)$.

Le coût global est donc $O(mn) + O(n)$.

IV Ordre d'élimination parfait pour un graphe cordal

IV.A – Cycles de longueur 4 dans un graphe d'intervalles

IV.A.1 Supposons, par l'absurde, que l'un des segments soit inclus dans un autre. Deux cas se présentent.

- Il s'agit de deux intervalles consécutifs dans le cycle et, quitte à renuméroter, on peut supposer que $I_0 \subset I_1$. I_0 et I_3 étant en conflit, il en sera a fortiori de même pour I_1 et I_3 et le cycle possédera la corde $\{I_2, I_3\}$.
- Sinon, quitte à renuméroter, on peut supposer que $I_2 \subset I_3$ et on obtient encore une corde.

Dans tous les cas, il y a contradiction.

IV.A.2 Comme I_1 et I_2 sont en conflit, on a soit $a_1 \in]a_2, b_2[$ soit $a_2 \in]a_1, b_1[$.

Dans le premier cas, comme I_1 n'est pas inclus dans I_2 , on a $a_2 < a_1 < b_2 < b_1$. De plus, I_0 et I_2 ne sont pas en conflit (pas de corde) et comme $b_2 > a_1 > a_0$, I_2 est à droite de I_0 c'est à dire $b_0 < a_2$. Les conditions $b_0 < a_2 < a_1$ contredisent l'hypothèse $a_1 < b_0$ de l'énoncé.

On est donc dans le second cas $a_1 < a_2 < b_1$ et, comme aucun intervalle n'est inclus dans un autre : $a_1 < a_2 < b_1 < b_2$.

On prouve de même que $a_2 < a_3 < b_2 < b_3$.

IV.A.3 On a alors $b_0 < a_1 < a_2 < a_3$ et I_3 est strictement à droite de I_0 et pas en conflit avec lui.

Ceci contredit l'existence de l'arête $\{I_0, I_3\}$.

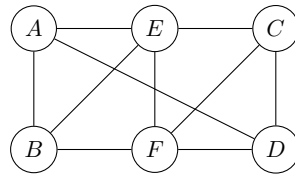
IV.B – Cordalité des graphes d'intervalles

Supposons, par l'absurde, que l'on dispose d'un graphes d'intervalles non cordal. L'ensemble des longueurs des cycles de taille ≥ 4 qui possèdent une corde est alors non vide et possède un minimum m et la question précédente donne $m \geq 5$. Soit $(v_0, v_1, \dots, v_{m-1}, v_0)$ un cycle de longueur minimale qui possède une corde. Comme $m \geq 5$ cette corde découpe le cycle en deux autres et l'un des deux est de longueur ≥ 4 et $\leq m - 1$ ce qui contredit la minimalité de m . On conclut que tout graphe d'intervalles est cordal.

IV.C – Une enquête policière

On a bien sûr envie d'introduire un graphe d'intervalles. Notons A, B, C, D, E, F les intervalles de temps de présence des différents protagonistes. Les affirmations des suspects se modélisent sous forme d'un

graphe d'intervalles puisqu'une personne peut en avoir vu une autre seulement si les intervalles associés sont en conflit. Il a la forme suivante :



(A, E, F, D) est un cycle de longueur 4 sans corde, donc l'une des personnes dont la déclaration a donné naissance à ce cycle a menti. Il s'agit de A , F , ou D (les arêtes $A-E$ et $E-F$ ne proviennent pas des déclarations d'Edouard).

(A, B, F, D) montre de même que A , B , ou D a menti. Ici, même si A a menti, le cycle demeure (c'est D qui a vu A et B a vu A aussi). Ainsi, le menteur est B ou D .

Finalement, en recoupant les informations : **Didier est le coupable.**

IV.D – Ordre d'élimination parfait

IV.D.1

- a. La fonction `bon_sommets` : `int list -> int list` prend en arguments une liste `l` de sommets, et renvoie la liste des éléments de `l` qui sont éléments de l'ensemble décrit par `sg`.

```

1 let simplicial (aretes, sg) k =
2   letrec bon_sommets l = match l with
3     | [] -> []
4     | i::q when sg.(i) -> i::bon_sommets q
5     | _::q -> bon_sommets q
6   in est_clique aretes (bon_sommets aretes.(k))
7 ;;

```

- b.
- Trouver les "bon sommets" se fait en un parcours de liste de taille $\leq n$ (nombre de sommets du graphe) pour un coût $O(n)$.
 - Dans l'appel `est_clique aretes xs`, pour chaque élément i de `xs` (il y en a au plus n), on teste si les éléments de `xs` différents de i sont voisins de i .
 - Tester si un élément est voisin de i a un coût $O(m_i)$ où m_i est le nombre des voisins de i .
 - Tester si tous les éléments de `xs` sont voisins de i a donc un coût $O(nm_i)$.
 - Le coût global des tests est donc $\sum_i O(nm_i) = O(mn)$ (m est le nombre des arêtes).

La complexité de `simplicial` est donc $O(mn) + O(n)$, n étant le nombre de sommets et m le nombre d'arêtes.

IV.D.2

```

a. 1 exception NoSimplicial ;;
2
3 let trouver_simplicial (aretes, sg) =
4   let n = Array.length aretes in
5   let i = ref 0 in
6   while !i < n && ((not sg.(!i)) || not (simplicial (aretes,sg) !i)) do
7     incr i
8   done;
9   if !i = n
10  then raise NoSimplicial
11  else !i
12 ;;

```

- b. Dans le pire des cas, le coût de la fonction `trouver_simplicial` est $O(mn^2) + O(n^2)$ (n appels à la fonction `simplicial`).

IV.D.3

- a. (x_0, \dots, x_{p-1}) est un ordre d'élimination parfait si et seulement pour tout $i \geq 1$, x_i est simplicial dans le sous-graphe induit par $\{x_0, \dots, x_{i-1}\}$. On a donc envie d'utiliser l'algorithme suivant :
- Si il n'y a qu'un seul sommet, la liste formée par ce sommet convient.
 - Sinon, trouver un sommet simplicial x , former le sous-graphe induit en supprimant x , chercher un ordre l pour ce sous graphe et renvoyer $l@[x]$ si on le trouve.

Cependant, dans la seconde étape il peut y avoir plusieurs sommets simpliciaux x et selon le choix du sommet, l'appel récursif pourrait aboutir ou non. Il faudrait commencer par montrer que pour tout graphe G (de taille au moins 2) et pour tout sommet simplicial x de G , G possède un ordre d'élimination parfait si et seulement si c'est le cas pour le sous-graphe H obtenu en supprimant x . Nous admettons ce résultat dans cette question.

Pour éviter d'utiliser une référence de liste, on écrit une fonction auxiliaire récursive. De plus, pour éviter une concaténation (voir la description ci-dessus) on utilise un accumulateur pour construire la liste "à l'envers".

On écrit ainsi une fonction `recherche : int -> int list -> int list`.

Dans l'appel `recherche j ordre`, `j` correspond au nombre de sommets déjà trouvés (et donc enlevés du graphes) et `ordre` aux sommets enlevés (dans l'ordre inverse d'enlèvement, le dernier élément de la liste étant le premier enlevé). Il faut globalement gérer un tableau `sg` donnant les sommets qui n'ont pas été enlevés.

```

1  let ordre_parfait aretes =
2    let n = Array.length aretes in
3    let sg = Array.make n true in
4    let rec recherche j ordre = match j = n with
5      | true -> ordre
6      | _ -> let k = trouver_simplicial (aretes,sg) in
7              sg.(k) <- false;
8              recherche (j+1) (k::ordre)
9    in recherche 0 []
10 ;;
```

- b. Dans le pire des cas, le coût de la fonction `ordre_parfait` est $O(mn^3) + O(n^3)$ (n appels à la fonction `trouver_simplicial`).

IV.E – Coupures minimales dans un graphe cordal

IV.E.1 Dans G , il existe des chemins de a à b et ils passent tous par au moins un élément de C . Notons C_1 l'ensemble des éléments de C qui sont directement voisins d'un sommet de G_1 . C_1 déconnecte donc a et b et $C_1 \subset C$. Par minimalité, cette inclusion est une égalité. Tout élément de C est donc directement relié à un élément de G_1 . On montre de même que tout élément de C est donc directement relié à un élément de G_2 .

IV.E.2 x est relié à un élément a_1 de G_1 . y est relié à un élément a_p de G_1 . Comme G_1 est une composante connexe, a_1 et a_2 sont reliés dans G_1 . On trouve donc un chemin P_1 convenable. De même pour P_2 .

IV.E.3 $(x, a_1, \dots, a_p, y, b_1, \dots, b_q, x)$ est un cycle de G . Comme $p, q \geq 1$, ce cycle est au moins de longueur 4. Comme le graphe est cordal, ce cycle possède une corde. Par défaut de connexité, une telle corde relie pas un a_i à un b_j . Par minimalité des longueurs des chemins P_1 et P_2 , elle ne relie pas non plus deux a_i ou deux b_j . Elle doit donc relier x et y . Ainsi, x et y sont voisins dans G .

IV.E.4 On a montré que deux sommets quelconques de C sont voisins dans G . C est donc une clique de G .

IV.F – Sommets simpliciaux dans un graphe cordal

IV.F.1 Si G est complet alors l'ensemble des voisins d'un sommet x est $S_x = S \setminus \{x\}$ et le graphe induit par S_x est encore complet. Le sommet x est donc simplicial.

IV.F.2

- Si G possède un sommet, il est complet et possède un sommet simplicial unique.
- Si G possède deux sommets, soit il est complet possède deux sommets simpliciaux soit il ne l'est pas et il a deux sommets isolés qui sont simpliciaux et non adjacents.
- Si G possède trois sommets, on peut envisager trois cas.
 - Le graphe est complet et possède trois sommets simpliciaux.
 - Les trois sommets sont isolés et il y a encore trois sommets simpliciaux qui ne sont pas voisins.
 - Le graphe a deux composantes connexes, l'une de taille 1 et l'autre de taille 2. En prenant un sommet dans chaque composante, on obtient deux sommets simpliciaux non voisins.

IV.F.3

- a. Soit $(u_0, u_1, \dots, u_{p-1} = u_p)$ un cycle de H_1 de longueur ≥ 4 . C'est a fortiori un cycle dans G et il possède une corde dans G . Cette corde étant composée de sommets de $S_1 \cup C$ est aussi une corde de H_1 . H_1 est donc cordal.
- b. Si H_1 est complet alors tout sommet de S_1 est simplicial pour H_1 . Par exemple, a l'est. Mais les voisins de a dans G sont tous soit dans S_1 soit dans C . a est donc simplicial dans G .
- c. Sinon, par hypothèse de récurrence $\mathcal{P}(H_1)$ est vraie et on peut trouver deux éléments de $S_1 \cup C$ simpliciaux pour H_1 et non voisins dans H_1 . Comme ils ne sont pas voisins et que C est une clique, l'un des deux est dans S_1 . Comme ci-dessus, c'est alors un sommet simplicial de G .
- d. Dans tous les cas on trouve un sommet simplicial de G dans S_1 et de même on en trouve un dans S_2 . Ces deux sommets ne sont pas voisins dans G (sinon, ils seraient dans la même composante connexe de $G \setminus C$). On a donc montré que $\mathcal{P}(G)$ est vérifiée. Ceci clôt la récurrence entamée (sans le dire) par l'énoncé (on mène une récurrence sur la taille du graphe cordal).

IV.G – Ordre d'élimination parfait pour un graphe cordal

Soit G un graphe cordal. On peut trouver un sommet simplicial x . Soit H le graphe induit par $S \setminus \{x\}$. Ce graphe est encore cordal (tout cycle dans H est un cycle dans G et possède une corde qui est dans H puisque ses extrémités ne sont pas égales à x). On peut ainsi appliquer de nouveau la procédure à H . On obtient ainsi tous les sommets dans un ordre x_0, \dots, x_{p-1} . Par construction, x_{p-1}, \dots, x_0 est un ordre d'élimination parfait.