Centrale 2015 - Graphes

I. Graphes d'intervalles

I.A. Représentation du problème

On suppose ici, et dans la suite, que si un couple (a, b) représente un intervalle alors $a \leq b$.
Il n'y a pas de conflit entre $[a, b]$ et $[c, d]$ quand $[a, b]$ est strictement à droite ou strictement à gauche de $[b, d]$. La condition de conflit est donc la négation de cette disjonction.

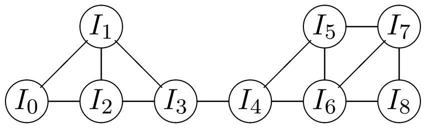
```
let conflit (a,b) (c,d) = not((b<c)||(d<a));;
```

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.1 Voici une représentation du graphe associé au problème b.



I.C.2 On initialise un tableau g de bonne taille avec des listes vides. A l'aide d'une double boucle, on considère tous les couples (I_i, I_j) d'intervalles avec $i < j$. Quand on détecte un conflit, on construit une arête, ce qui revient à ajouter i à la liste numéro j et j à la liste numéro i (ce qui est possible puisque la structure de tableau est mutable).

```
let construit_graphe t =
   let n=vect_length t in
   let g=make_vect n [] in
   for i=0 to n-2 do
     for j=i+1 to n-1 do
       if conflit t.(i) t.(j) then
          begin
          g.(i) <- j::g.(i);
          g.(j) <- i::g.(j)
          end ;
       done;
     done;
   g;;
```

I.D. Coloration

ID.1 Le graphe associé au problème a ne peut être colorié avec moins de trois couleurs du fait de la présence du triangle reliant I_0, I_1, I_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 3 couleurs pour le graphe du problème b (triangle I_0, I_1, I_2). Une coloration optimale convenable est

$$(0, 1, 2, 0, 1, 2, 0, 1, 2)$$

ID.2a Il s'agit de la fonction `mem` de Caml. Ci-dessous, on tire partie de l'évaluation paresseuse (si le test `x=y` vaut `true`, l'appel récursif n'est pas effectué).

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

ID.2b On teste successivement tous les entiers en incrémentant une référence tant que sa valeur est présente.

```
let plus_petit_absent l =
   let i=ref 0 in
   while (appartient l !i) do incr i done;
   !i;;
```

ID.2c J'écris une fonction auxiliaire locale `construit : int list` $\to$ `int list` qui prend en argument une liste de sommets et renvoie la liste des couleurs de ceux-ci (ceux qui sont coloriés). Il suffit de l'appeler avec la liste des voisins de i. Cette fonction étant locale, elle connaît le tableau `couleurs`. L'énoncé ne précise pas si la liste résultat peut comporter des doublons (rien n'empêche un sommet d'avoir plusieurs voisins de la même couleur). J'ai choisi d'imposer une liste résultat sans doublon (d'où le test d'appartenance).

```
let couleurs_voisins aretes couleurs i =
   let rec construit l =
      match l with
      [] -> []
     |j::q ->
        let liscoul=construit q in
        if couleurs.(j) <> (-1)
           & not (appartient liscoul couleurs.(j))
        then couleurs.(j)::liscoul
        else liscoul
    in construit aretes.(i);;
```

ID.2d Il suffit de combiner les deux fonctions précédentes.

```
let couleur_disponible aretes couleurs i =
   plus_petit_absent (couleurs_voisins aretes couleurs i);;
```

I.E. Cliques

IE.1a Si G ne possède pas d'arêtes alors (de façon immédiate)

$$\chi(G) = 1 \quad \text{et} \quad \omega(G) = 1$$

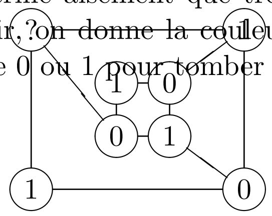
IE.1b Si G est complet à n sommets alors (de façon immédiate)

$$\chi(G) = n \quad \text{et} \quad \omega(G) = n$$

IE.2 Une bonne coloration d'un graphe G induit une bonne coloration de tout sous-graphe de G. Avec la question précédente, on en déduit que

$$\omega(G) \leq \chi(G)$$

L'égalité est fausse 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.



IE.3 Il s'agit de voir si pour tout élément `i` de `xs`, la liste `aretes.(i)` contient bien tous les autres éléments de `xs`. Je propose donc d'écrire deux fonctions auxiliaires locales (qui connaîtront donc `aretes`).

- `tester : int list` $\rightarrow$ `int` $\rightarrow$ `bool`. Dans l'appel `tester l i` on regarde si tous les éléments de `l` différents de `i` sont voisins de `i`.
- `parcourir : int list` $\rightarrow$ `bool`. Dans l'appel `parcourir xs`, on teste si chaque élément de `xs` est relié à tous les autres.

```
let est_clique aretes xs =
    let rec tester l i =
       match l with
       [] -> true
      |j::q -> if j=i then tester q i
               else (appartient aretes.(i) j)&&(tester q i)
  in
    let rec parcourir xs =
       match xs with
       [] -> true
      |i::q -> tester xs i
  in parcourir xs ;;
```

II. Algorithme glouton pour la coloration

II.A L'algorithme sur un exemple

Dans le problème b, certaines extrémités gauches de segments sont égales. On voit sur cet exemple que l'ordre des intervalles n'est donc pas parfaitement défini. En choisissant la numérotation proposée par l'énoncé, on obtient la coloration

$$(0, 1, 2, 0, 1, 0, 2, 1, 0)$$

II.B. Coloration

Notons que dans la fonction demandée, l'argument `segments` ne sert à rien puisque tous les renseignements sont contenus dans le graphe (alternativement, on pourrait se contenter de `segments` à partir de qui on sait construire le graphe).
Il nous suffit de créer un tableau de couleurs de bonne taille (initialisé avec l'absence de couleur -1) et de le remplir petit à petit grâce aux fonctions de **I.D**.

```
let coloration segments aretes =
    let n=vect_length aretes in
    let couleurs=make_vect n (-1) in
    couleurs.(0) <- 0 ;
    for i=1 to n-1 do
       let c=couleur_disponible aretes couleurs i in
       couleurs.(i) <- c
       done;
    couleurs;;
```

II.C. Preuve de l'algorithme

IIC.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 $I_0, \dots, I_{k-1}$.

IIC.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.

IIC.3 La question **IE.2** indique alors que le nombre chromatique est plus grand que $c+1$.

IIC.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.

II.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 oeuvre 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. A 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 $O(mn) + O(n)$.
Remarque : il me semble difficile de donner une complexité ne faisant pas intervenir le nombre d'intervalles.

III. Graphes munis d'un ordre d'élimination parfait

III.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 choisi 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

$$(x_5, x_2, x_1, x_4, x_7, x_0, x_6, x_3)$$

III.B. Vérification

IIIB.1 On écrit une fonction auxiliaire locale `construit : int list` $\rightarrow$ `int list` qui prend en argument une liste `l` et renvoie la liste de ses éléments qui sont $< x$ (x est connu puisque la fonction est locale). Il suffit de l'appeler avec la liste des voisins de x.

```
let voisins_inferieurs aretes x =
   let rec construit l =
      match l with
      [] -> []
     |i::q -> if i<x then i::(construit q)
                else (construit q)
   in construit aretes.(x);;
```

IIIB.2 Pour chaque sommet (dans l'ordre) on regarde si la liste de ses voisins inférieurs forme une clique. Une boucle conditionnelle s'impose puisque l'on peut s'interrompre dès qu'une anomalie est détectée.

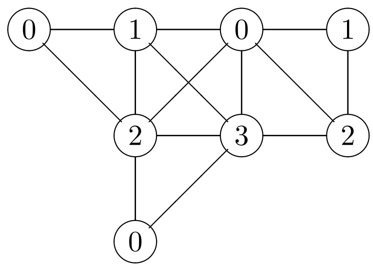
```
let est_ordre_parfait aretes =
   let n=vect_length aretes in
   let x=ref 1 in
   while !x<n && (est_clique aretes (voisins_inferieurs aretes !x)) do
      incr x
      done;
   !x=n;;
```

III.C. Ordre d'élimination parfait pour un graphe d'intervalles

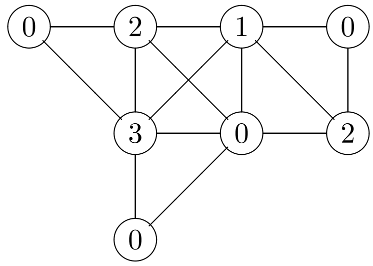
L'ensemble C formé des intervalles I_i tels que $i < k$ et I_i est relié à I_k (c'est à dire tels que I_i et I_k sont en conflits) est exactement celui des intervalles I_i tels que $i < k$ et $a_k \in I_i$ (c'est une conséquence directe de l'ordre choisi pour les intervalles). On peut alors reprendre la preuve de la question **II.C2** pour voir que cet ensemble est une clique. Ceci est vrai pour tout k et on a bien un ordre d'élimination parfait.

III.D. Coloration

IIID.1a Pour l'ordre initial, on obtient la coloration suivante



IIID.1b Pour l'ordre $(x_5, x_2, x_1, x_4, x_7, x_0, x_6, x_3)$, la coloration est



IIID.2 Je ne vois pas la différence avec la question **II.B** (où je n'utilisais pas la liste des segments). Je fournis donc la même fonction.

```
let colore aretes =
    let n=vect_length aretes in
    let couleurs=make_vect n (-1) in
    couleurs.(0) <- 0 ;
    for i=1 to n-1 do
       let c=couleur_disponible aretes couleurs i in
       couleurs.(i) <- c
       done;
    couleurs;;
```

IIID.3a A nouveau, on est dans la même situation qu'au **IIC.3**. Quand on attribue la couleur c_i, on a une clique de taille $1 + c_i$ et donc le nombre chromatique est plus grand que $1 + c_i$.

IIID.3b Cette fois, c'est la même chose qu'en **IIC.4** me semble-t-il.

IV. Ordre d'élimination parfait pour un graphe cordal

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

IV.A1 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.A2 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 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

$$a_2 < a_3 < b_2 < b_3$$

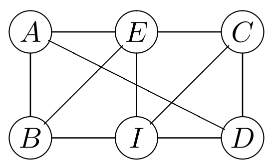
IV.A3 On a alors $b_0 < a_1 < a_2 < a_3$ et I_3 est strictement à droite de I_0 et pas en conflit avec lui. Ceci contredit l'existence de l'arête $\{I_0, I_3\}$.

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 $(v_0, v_1, \ldots, v_{m-1}, v_0)$ 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 bon graphe d'intervalles. Notons A, B, C, D, E, I les intervalles de temps de présence des différents protagonistes. *Une première question angoissante est : pourquoi Isabelle ne s'appelle-t-elle pas Fiona afin d'avoir des lettres qui se suivent ?* Les affirmations des suspects se modélisent sous forme d'un graphe puisqu'une personne peut en avoir vu une autre seulement si les intervalles associés sont en conflit. Il a la forme suivante :



(A, E, I, D) est un cycle de longueur 4 sans corde et l'une des personnes dont la déclaration a donné naissance à ce cycle a menti. Il s'agit de A, I ou D (les arêtes $\{A, E\}$ et $\{E, I\}$ ne proviennent pas des déclarations d'Édouard.
(A, B, I, 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, c'est B ou D le menteur.
Finalement, en recoupant les informations

DIDIER EST LE COUPABLE

IV.D. Ordre d'élimination parfait

IV.D1 J'écris une fonction auxiliaire locale `bon_sommets : int list` $\rightarrow$ `int` telle que l'appel `bon_sommets l` renvoie la liste des éléments de `l` qui sont éléments de l'ensemble décrit par `sg`. Il nous suffit alors de l'appeler avec l'ensemble des voisins de `k` et de voir si la liste de sommets obtenue est une clique de G (pour une liste de sommets du sous-graphe H, être une clique de G ou de H sont des phrases équivalentes).

```
let simplicial (aretes,sg) k =
    let rec bon_sommets l=
      match l with
       [] -> []
     |i::q -> if sg.(i) then i::(bon_sommets q)
                else bon_sommets q
    in est_clique aretes (bon_sommets aretes.(k));;
```

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 $m_i i$ 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.D2 Il suffit de tester un à un tous les sommets. On s'arrête quand on a testé tous les sommets ou quand on trouve un élément de `sg` qui est simplicial (le test de continuation est ainsi la négation de cette condition). En fin d'itération, on n'a pas trouvé de sommet simplicial si tous les sommets ont été testés ($i = n$). Dans ce cas, on lève une erreur. Dans le cas contraire, i est le numéro d'un sommet simplicial du sous-graphe.

```
let trouver_simplicial (aretes,sg) =
   let n=vect_length aretes in
   let i=ref 0 in
   while !i<n && ((not sg.(!i)) || not (simplicial (aretes,sg) !i)) do
         incr i
         done;
   if !i=n then failwith "Pas de simplicial"
   else !i ;;
```

Le coût de la fonction `trouver_simplicial` est alors immédiatement $O(mn^2)+O(n^2)$ (n appels au pire à `simplicial`).

IV.D3 $(x_0, \ldots, x_{p-1})$ est un ordre d'élimination parfait si et seulement pour tout $i \geq 1$, x_i est simplicial dans le sous-graphe induit par $\{x_0, \ldots, x_{i-1}\}$. 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 (vois la description ci-dessus) on utilise une technique accumulative. On écrit ainsi une fonction `recherche : int` $\rightarrow$ `int list` $\rightarrow$ `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.

```
let ordre_parfait aretes =
   let n=vect_length aretes in
   let sg=make_vect n true in
   let rec recherche j ordre=
      if j=n then ordre
      else begin
         let k=trouver_simplicial (aretes,sg) in
         sg.(k) <- false ;
         recherche (j+1) (k::ordre)
         end
   in recherche 0 [];;
```

IV.E. Coupures minimales dans un graphe cordal

IV.E1 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.E2 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.E3 $(x, a_1, \ldots, a_p, y, b_1, \ldots, b_q, x)$ 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 de 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.E4 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.F1 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.F2 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.

IVF.3a 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.

IVF.3b 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.

IVF.3c 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.

IVF.3d 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 serait 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.