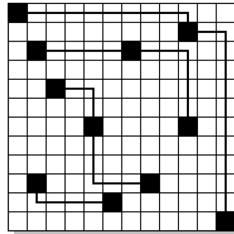
Examen Partiel : Algorithmique des Structure de Données

Ecole normale supérieure
Département d'informatique
td-algo@di.ens.fr

5 Décembre 2011

Tous documents autorisés. Ne paniquez pas. Il n'est pas indispensable d'écrire du pseudo-code pour décrire un algorithme. L'important est de se faire comprendre. La rigueur des preuves sera évaluée. Il est possible d'admettre le résultat d'une question pour traiter les suivantes.

Exercice 1 : Couverture Monotone minimale. On considère une grille $n \times n$ dont les cases sont soit noires soit blanches. La grille est donnée par une matrice de booléens. Un chemin monotone dans la grille part d'une case noire, termine sur une case noire, et ne peut avancer que vers le bas ou vers la droite. Le but du problème est de couvrir l'ensemble des cases marquées avec le moins possible de chemins monotones.



La stratégie générale est de ramener le problème à une situation de flots maximaux. Comme c'est un peu compliqué, on va s'y prendre par étapes. D'abord, il est assez clair que la grille définit un réseau de transport de flot où chaque case est reliée en à celle du bas et à celle de droite.

($\star$) 1. Expliquez comment on peut forcer un flot à atteindre chacune des cases noires. Dessinez un flot maximal pour la grille d'exemple. On ne cherche pas à minimiser le nombre de chemins.

Un des problèmes de ce flot est qu'il peut "brancher" : si plusieurs unités de flot entrent dans une case, rien n'interdit qu'une partie aille dans la case du bas et que l'autre partie aille dans la case de droite. Ceci ne correspond pas très bien à notre notion intuitive de ce qu'est un "chemin". Une solution tentante à ce problème consisterait à imposer une capacité de 1 sur toutes les arêtes, mais cela pose des difficultés techniques. Il faudrait en particulier s'assurer que ça ne change pas la valeur du flot maximal.

($\star\star$) 2. Montrez comment, en remplaçant chacun des sommets associé à une case noire par un gadget composé de deux sommets, on peut trouver un flot qui traverse toutes les cases noire, en imposant que la capacité de toutes les arêtes soit 1. On ne cherche toujours pas à minimiser le nombre de chemins. Dessinez un flot maximal possible pour la grille d'exemple.

($\star$) 3. Expliquez comment on peut transformer la solution du problème de flot (avec capacité un) en solution du problème de couverture monotone, et vice-versa.

Maintenant, le moment que vous attendiez tous, on cherche à minimiser le nombre de chemins. Pour cela, on va dire que les arêtes ont non seulement une capacité $c(u, v)$, qui limite la quantité de flot qui peut passer, mais aussi un coût $cost(u, v)$. Si $f(u, v)$ unités de flots circulent le long d'une arête, alors on "paye" $f(u, v) \cdot cost(u, v)$, et donc l'ensemble du flot coûte :

$$Cost = \sum_{(u,v)\in E} f(u, v) \cdot cost(u, v)$$

On a déjà restreint les capacités à être unitaires. Les coûts, eux, peuvent prendre n'importe quelle valeur dans $\mathbb{Z}$. Ils peuvent en particulier être négatifs, ce qui signifie que pousser du flot dans l'arête en question rapporte un profit. On impose que les coûts soient symmétriques : $cost(u, v) = -cost(v, u)$. Sinon on pourrait réaliser un profit infini (ou une perte infinie...) en poussat du flot le long de l'arête dans un sens puis dans l'autre de façon répétitive.

Le problème du flot maximum de coût minimum, comme son nom l'indique, consiste à trouver la manière de pousser le plus possible de flot dans le réseau, et parmi celles-là trouver une de celles qui coûtent le moins (voire même qui rapporte le plus). Pour notre problème de couverture monotone, il est logique de chercher une modélisation où le coût du flot est le nombre de chemins.

($\star\star$) 4. Montrez qu'on peut résoudre le problème de couverture monotone minimale en résolvant une instance de flot maximum de coût minimum. N'oubliez pas de démontrer que cette procédure fournit bien une solution optimale au problème de départ.

Il reste à résoudre le problème de flot maximum de coût minimum. Pour ça, on calcule dans un premier temps un flot maximal en ignorant les coûts, et une fois que c'est fait on se penche sur le graphe résiduel.

($\star$) 5. Comment définir le graphe résiduel en présence des coûts ?

($\star$) 6. Que se passe-t-il s'il existe un cycle dans le graphe résiduel tel que la somme des coûts le long de ses arrêtes est négative ? Dessinez un exemple.

($\star\star$) 7. Comment trouver un cycle de coût négatif ? Quelle est la complexité de la procédure ?

On veut démontrer que si un flot maximum n'est *pas* de coût minimal, alors le graphe résiduel contient (au moins) un cycle de coût négatif. Comme ça n'est pas très facile, les questions 8, 9 et 10 servent à vous diriger dans la bonne direction. Les questions 11 et 12 sont de nouveau consacrées au problème de trouver un flot de coût minimum.

Dans un graphe muni de capacités, une circulation est un flot "qui tourne en boucle", c'est-à-dire qui n'est pas alimenté par une source et qui ne se déverse pas dans un puits. C'est en fait un flot qui respecte les contraintes de capacité, de symétrie et de conservation sur tous les noeuds (pas d'exceptions pour la source ni le puits).

($\star\star$) 8. On prend un flot maximum f dont le coût n'est *pas* minimal, et un autre flot maximum f^* dont le coût, lui, est minimal. Montrez que leur différence $f' = f^* - f$ est une circulation de G_f, le graphe résiduel du flot f, de coût total négatif.

($\star\star$) 9. Montrer qu'il existe un cycle le long duquel la circulation du flot est strictement positive dans f' (c'est-à-dire une séquence de sommets $u_1, \ldots, u_k, u_1$ avec $f'(u_i, u_{i+1}) > 0$).

($\star\star\star$) 10. Montrer qu'il existe un cycle le long duquel la circulation du flot est strictement positive dans f', et tel que la somme des coûts le long des arêtes est (strictement) négative.

On en revient au problème de calculer un flot maximum de coût minimum. L'idée générale est la suivante : on trouve un cycle de coût négatif dans le graphe résiduel. On fait circuler le plus de flot possible le long de ce cycle, ce qui fait baisser le coût du flot sans changer sa valeur. On recommence tant que c'est possible. On a la garantie que lorsqu'on ne peut plus, c'est qu'on a un flot de coût minimal.

($\star$) 11. Combien de cycles négatifs peut-on "annuler" au maximum avant d'atteindre la solution optimale ?

($\star$) 12. Exprimez la complexité de l'algorithme pour le problème du flot maximal de coût minimal. Spécialisez le résultat pour notre problème de couverture monotone.

Exercice 2 : Raffinage de partition et application aux sommets jumeaux. Une partition est une structure de donnée qui décrit un ensemble de classes (disjointes), et une classe est elle-même une structure de donnée qui décrit un ensemble d'éléments. On peut supposer sans problème que les éléments qu'il s'agit de partitionner sont des entiers de l'intervalle $[0; n[$. On notera Ω l'univers qui contient les éléments de la partition (c'est l'union des classes). On cherche à effectuer les trois opérations suivantes :

A. Initialiser la partition $\mathcal{P}$ avec une seule classe égale à Ω

B. Raffiner la partition $\mathcal{P}$. Étant donné un ensemble $X \subset \Omega$, on remplace chaque classe $\mathcal{C}$ de $\mathcal{P}$ par $\mathcal{C} \cap X$ et $\mathcal{C} - X$. On ne stocke pas les classes vides.

C. Renvoyer $\mathcal{P}$, sous une forme acceptable

On veut une structure de donnée pour représenter une partition permettant d'effectuer l'opération de raffinage en temps $\mathcal{O}(|X|)$, indépendamment du nombre d'éléments dans la partition. Les autres opérations doivent s'effectuer en temps linaire en la taille de leur entrée (resp. leur sortie).

($\star\star\star$) 1. Décrivez cette structure de données, et expliquer comment on peut effectuer le raffinage en temps $\mathcal{O}(|X|)$. N'oubliez pas la suppression des classes vides. Il peut être utile de commencer par déterminer, pour chacune des classes $\mathcal{C}$ qui intersectent X, les tailles de $\mathcal{C} \cap X$ et de $C - X$, avant de faire le raffinage lui-même...

($\star$) 2. Montrez comment on peut retirer un élément de sa classe en temps $\mathcal{O}(1)$.

On considère désormais un graphe non-orienté sans boucles $G = (V, E)$ à n sommets, qu'on peut supposer sans perte de généralité numérotés de 0 à $n-1$. On note $N(v)$ le voisinage d'un sommet v, c'est-à-dire l'ensemble des sommets qui lui sont adjacents. Deux sommets u et v sont de faux (resp. vrais) jumeaux si $N(u) = N(v)$ (resp. $\{u\} \cup N(u) = \{v\} \cup N(v)$). On cherche un algorithme qui identifie les sommets jumeaux d'un graphe. Comme la relation "être jumeaux" est une relation d'équivalence, on cherche en fait à partitionner l'ensemble des sommets en classes de jumeaux. L'algorithme pour les faux jumeaux est le suivant :

```
1: function Twins(V,E)
2:     P ← {V}
3:     for all v ∈ V do Refine(P, N(v))
4:     return P
5: end function
```

($\star$) 3. Montrez que si deux sommets ne sont pas jumeaux, alors une étape de raffinage va les séparer.

($\star\star$) 4. Montrez qu'à tout moment la partition $\mathcal{P}$ est compatible avec le résultat attendu : deux (faux) jumeaux sont toujours dans la même classe. Déduisez-en la correction de l'algorithme.

($\star$) 5. Quelle est la complexité de l'algorithme ?

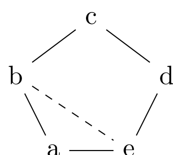
($\star\star$) 6. Adaptez l'algorithme (et les preuves) au cas des vrais jumeaux.

Exercice 3 : Clique maximum sur les graphes d'intervalle et les graphes chordaux. Cet exercice vise à fournir un algorithme *linéaire* pour trouver une clique maximum sur une certaine famille de graphes. On rappelle qu'une clique est un sous-ensemble des sommets qui sont tous reliés entre eux, et qu'une clique maximum possède le plus grand nombre de sommet. Il n'existe pas (à ce jour...) d'algorithme polynomial pour résoudre ce problème dans des graphes arbitraires. D'ailleurs, c'est un exemple archi-classique de problème NP-complet.

Un graphe d'intervalle est un graphe (non-orienté) $G = (V, E)$ satisfaisant la condition suivante : il est possible d'étiqueter chaque sommet par un intervalle (non-vide) de $\mathbb{R}$, de telle sorte que deux sommets sont adjacents si et seulement si les intervalles qui les étiquettent ont une intersection non-vide. Les graphes d'intervalle ont plein de bonne propriétés, et ces propriétés sont particulièrement faciles à exploiter lorsque les étiquette sont présentes.

($\star$) 1. Proposez un algorithme simple qui prend le graphe en entrée *avec les étiquettes d'intervalles*, et qui trouve une clique maximum en temps $\mathcal{O}(n \log n)$.

On s'intéresse maintenant au cas où les étiquettes ne sont pas disponibles. Il devient alors indispensable d'exploiter les informations fournies par les arêtes. Ceci est néanmoins possible de façon très efficace, mais avec plus de sophistication. La suite de l'exercice vise à concevoir et prouver un algorithme *linéaire* qui résout le problème sur les graphes chordaux. Un graphe chordal est tel que n'importe quel cycle de longueur supérieure ou égale à 4 est traversé par une corde, c'est-à-dire une arête entre deux sommets non-consécutifs du cycle. Les graphes chordaux sont aussi appelés graphes triangulés.



Un cycle de longueur 5, avec une corde $\{b, e\}$.
Il n'est pas chordal, à cause du cycle (b, c, d, e).

($\star$) 2. Démontrez qu'un graphe d'intervalle est chordal.

Un perfect elimination ordering est une bijection σ qui envoie V sur $\{1, \ldots, |V|\}$ vérifiant une propriété qu'on va définir (en fait, c'est une numérotation des sommets). Le Voisinage à droite de $v \in V$ est l'ensemble

$$RN(v) = \Big\{u \in N(v) \mid \sigma(u) > \sigma(v)\Big\}$$

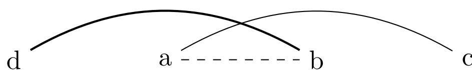
La propriété qui caractérise σ est la suivante : pour tout sommet $v \in V$, l'ensemble de sommets $RN(v)$ forme une clique (et donc comme ils sont tous adjacents à v, $\{v\} \cup RN(v)$ est une clique aussi).

($\star\star$) 3. Supposons qu'un perfect elimination ordering soit fourni. Donnez une procédure de complexité $\mathcal{O}(V + E)$ qui renvoie une clique maximum du graphe.

Le problème se ramène donc à trouver un perfect elimination ordering. Il en existe toujours pour les graphes chordaux, mais pas forcément pour des graphes arbitraires. Pour en calculer un, on va utiliser un parcours de graphe. En effet, n'importe quel parcours de graphe (en largeur, en profondeur, ...) marque les sommets comme “visités” dans un certain ordre. Ceci permet de numéroter les sommets dans l'ordre où ils sont marqués, et induit une relation d'ordre total sur les sommets : $u > v$ si u est marqué avant v.

Les différents parcours de graphes produisent des ordres aux propriétés différentes. Par exemple, l'algorithme de tri topologique des graphes orientés acycliques utilise de façon cruciale les propriétés de l'ordre induit par le parcours en profondeur. Nous allons, de notre côté, nous intéresser aux parcours en largeur. L'ordre produit par un parcours en largeur vérifie la propriété (L) suivante :

Si $a > b > c$ et si $\{a, c\} \in E, \{a, b\} \notin E$, alors il existe un sommet d tel que $d > a$, et $\{b, d\} \in E$.



En d'autres termes, d est la raison pour laquelle b est marqué avant c (les arêtes pointillées sont des anti-arêtes, l'indication de l'absence d'une arête).

($\star$) 4. Rappelez le principe du parcours en largeur et démontrez la propriété (L).

On va utiliser un parcours de graphe spécial, adapté à nos besoin, le LexBFS (parcours en largeur lexicographique), car il fournit directement un perfect elimination ordering sur les graphes chordaux. Cette procédure affecte des étiquettes aux sommets, qui sont des ensembles d'entiers pris dans l'ordre décroissant. On compare deux de ces étiquettes selon l'ordre lexicographique :

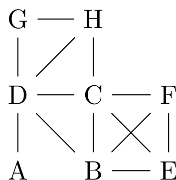
$$\{p_1 > p_2 \cdots > p_k\} \prec \{q_1 > q_2 \cdots > q_\ell\} \Longleftrightarrow \begin{cases} p_1 = q_1, \dots, p_{j-1} = p_{j-1}, p_j < q_j \\ p_1 = q_1, \dots, p_k = q_k, k < \ell \end{cases}$$

Donc, par exemple, $\{7, 5, 2, 1\} \prec \{7, 6\}$ et $\{4, 3\} \prec \{4, 3, 1\}$. $\emptyset$ est plus petit que n'importe quoi.

1: **function** LexBFS(G, s)
2: **for each** $v \in V$ **do** $label(v) \leftarrow \emptyset$
3: $label(s) \leftarrow \{n\}$
4: **for** i **from** n **to** 1 **do**
5: choisir un sommet non-marqué v dont le label est le plus grand selon $\prec$
6: Marquer v
7: $\sigma(v) \leftarrow i$
8: **for each** $u \in N(v)$, u non-marqué **do** $label(u) \leftarrow label(u) \cup \{i\}$
9: **end for**
10: **return** σ (c'est l'ordre des sommets qui nous intéresse)
11: **end function**

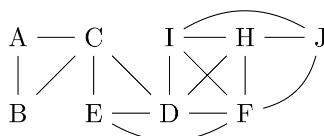
Voici un exemple d'exécution de LexBFS, en partant du sommet “A” :

Sommet	numéro affecté	label
A	8	8
B	6	7
C	5	7,6
D	7	8
E	2	6,5
F	1	6,5,2
G	3	7,4
H	4	7,5



On notera qu'à la troisième étape, le choix de B n'est pas forcé (on aurait pu choisir G,H ou C). Par contre, à la 4ème étape, le choix de C est forcé (car il a le label $\{7, 6\}$, qui est plus grand que le label $\{7\}$ porté par les autres).

($\star$) 5. Calculer un ordre LexBFS sur le graphe suivant, en partant du sommet de votre choix.



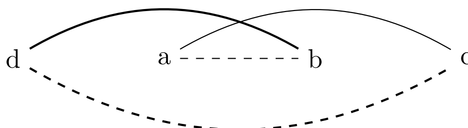
La page Wikipedia (anglaise) sur LexBFS affirme :

> The algorithm is called lexicographic breadth-first search because the lexicographic order it produces is an ordering that could also have been produced by a breadth-first search, and because if the ordering is used to index the rows and columns of an adjacency matrix of a graph then the algorithm sorts the rows and columns into Lexicographical order.

Sans s'étendre sur le caractère légèrement imprécis de cette phrase, on va supposer qu'elle signifie que si on rangeait les sommets du plus grand au plus petit et qu'on écrivait la matrice d'adjacence (la ligne/colonne du plus grand sommet en haut/à gauche), alors on devrait trouver les lignes dans l'ordre lexicographique décroissant.

($\star$) 6. Montrez que ceci impliquerait que la suite formée par les labels des sommets sélectionnés (dans l'ordre de sélection) est décroissante, et... que c'est faux.

($\star\star$) 7. La propriété (LexB) caractéristique des parcours LexBFS est : (L) est vraie, et en plus $\{c, d\} \notin E$.



a) Démontrer que dans LexBFS, si deux sommets reçoivent des labels différents à un moment donné, alors le classement relatif de leur label ne changera pas jusqu'à la terminaison de l'algorithme.

b) On appelle ℓ_a, ℓ_b et ℓ_c les labels de a, b et c au moment où a est sélectionné. Démontrez que ℓ_b est strictement plus grand que ℓ_c.

c) Démontrez que le moment où le label de b devient plus grand que celui de c est le moment où d est sélectionné.

d) Concluez la démonstration de la propriété (LexB)

Le pseudo-code de LexBFS laisse un certain nombre de détails à l'imagination du lecteur, et sa complexité n'est pas claire.

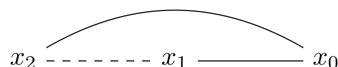
($\star$) 8. Proposez une manière simple d'implémenter LexBFS, et précisez la complexité du résultat. N'hésitez pas à utiliser des structures de donnée standard si c'est nécessaire. Il suffira d'en décrire la fonctionnalité et les performances.

Pour implémenter LexBFS en temps linéaire, le seul véritable problème est de pouvoir choisir le prochain sommet à traiter en temps constant. Pour cela, on utilise une technique de raffinage de partition. Il s'agit de maintenir à chaque étape la liste des sommets non-marqués sous la forme d'une partition ordonnée dans laquelle les sommets qui ont le même label sont dans la même classe, et où les classes sont triées par ordre de label. Pour mettre à jour les labels, on effectue une opération de raffinage (attention à maintenir l'ordre de la partition).

($\star\star\star$) 9. Décrivez une manière d'implémenter LexBFS qui s'exécute en temps $\mathcal{O}(V + E)$. Prouvez sa correction et sa complexité

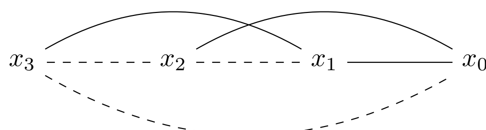
Il reste, pour compléter l'algorithme pour la clique maximum sur les graphes chordaux, à démontrer que LexBFS fournit effectivement un perfect elimination ordering sur ces graphes. Pour faire cette preuve, un peu technique, on raisonne par l'absurde, et on suppose que LexBFS ne renvoie pas un perfect elimination ordering sur un graphe chordal.

($\star$) 10. Démontrez qu'il existe alors 3 sommets $x_2 > x_1 > x_0$ dans la configuration suivante :



Dans toute la suite, on va suppose que x_0 est le plus petit sommet (c.a.d. sélectionné en dernier) tel que cette configuration existe, et on va également supposer que x_2 est le plus grand sommet qui réalise cette configuration avec notre choix de x_0.

($\star\star$) 11. Démontrez qu'il existe un sommet x_3 qui vérifie $x_3 > x_2$ et qui est dans la configuration suivante :



N'oubliez pas de montrer que $\{x_3, x_2\} \notin E$. Argumentez qu'on peut supposer x_3 maximal (selon l'ordre LexBFS) sans perte de généralité.

(⋆⋆⋆⋆) 12. On suppose qu'on nous une séquence de n sommets du graphe $x_n, \ldots, x_1$ qui satisfont les propriétés suivantes, pour tout $i, j > 0$:

i) $\{x_0, x_i\} \in E \iff i \leq 2$,

ii) $\{x_i, x_j\} \in E \iff |i - j| = 2$,

iii) $x_n > x_{n-1} > \cdots > x_1 > x_0$ (selon l'ordre LexBFS)

iv) x_j est le plus grand sommet de tout le graphe tel que

$$\{x_{j-2}, x_j\} \in E \text{ mais } \{x_{j-3}, x_j\} \notin E$$

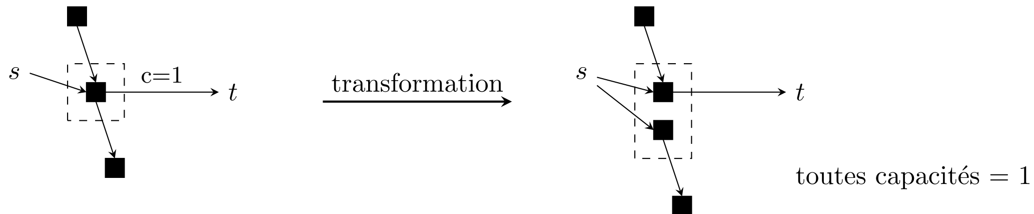
Démontrez qu'il y a dans le graphe un $(n+1)$-ème sommet qui satisfait ces propriétés.

(⋆) 13. Concluez.

1 Solutions

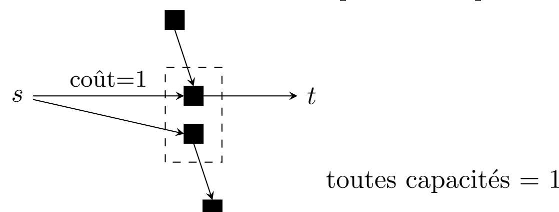
Solution de l'exercice 1

1. Notons N le nombre de cases noires. On fabrique une source s, et on relie la source à chaque case noire. On relie ensuite chaque case noire à un puits t par une arête de capacité 1. Il s'ensuit que la valeur maximale du flot est N. Si ce n'est pas précisé, la capacité des arêtes est infinie. N'importe quel flot maximal "emprunte" forcément chacune des N cases noires.
   Dans le fond, on n'est pas obligé de représenter les cases blanches. On peut se contenter de relier chaque case noire aux cases noires qui sont en dessous et/ou à droite.
2. Un "gadget" utilisable consiste à effectuer la substitution suivante :



   Une fois qu'on a réalisé la substitution, on réalise que la quantité maximale de flot qui peut circuler dans le réseau est inchangée (elle est toujours égale au nombre de cases noires).
3. Étant donné un tel flot, on peut le convertir en couverture monotone (et vice-versa). Le sens couverture → flot est facile. Il y a trois "types" de cases noires à considérer : celles qui ouvrent un chemin, celle qui sont au milieu, et celles qui ferment un chemin. Quand une case noire ouvre un chemin, alors on fait circuler une unité de flot le long de chacune des deux arêtes qui relie la source au gadget de la case noire. Si une case noire est au milieu d'un chemin, alors une unité de flot arrive "par au-dessus", et il suffit d'en envoyer une autre "par en-dessous", en en faisant venir une de la source. Si une case noire est une queue de chemin, alors il n'est pas la peine de faire venir du flot de la source, car on peut se contenter de celui qui arrive par au-dessus pour alimenter le puits. On pourrait vérifier que le flot ainsi défini vérifie toutes les contraintes (symétrie, conservation, etc.).
   Dans le sens flot → couverture, il suffit de faire la transformation inverse. Comme une unité de flot transite entre chaque gadget et le puit, il s'ensuit qu'au moins une unité de flot entre dans chaque gadget. Le flot peut entrer soit par au-dessus, soit par la source, soit les deux. Il y a donc plusieurs cas :
   1. une unité par au-dessus et rien par la source (queue de chemin)
   2. une seule unité par la source source, rien par au-dessus (sommet isolé)
   3. une unité par au-dessus, une seule unité par la source (milieu de chemin)
   4. une unité par au-dessus, deux unités par la source (impossible, débordement)
   5. deux unités par la source (tête de chemin)

   Mis à part les sommets isolés, chaque autre sommet peut être relié à un prédécesseur et/ou à un successeur de manière unique, vu que la capacité des arêtes est de un. C'est d'ailleurs plus simple à faire si les cases noires sont directement reliées entre elles.
4. Il faut jouer sur les coûts. Ceci est une des manières possible de procéder :



   On prétend que le coût du flot est le nombre de chemin de la couverture monotone correspondante. Comme le flot est maximal, une unité de flot doit passer de chaque case noire à t. Il en résulte que le coût est le nombre de case noire qui ne reçoivent pas de flot d'une autre case noire. Ces cases-ci sont donc des têtes de chemins (ou, à la rigueur, des cases noires isolées qui forment un chemin à elles toute seules). La solution du problème max-flow/min-cost fournit bien une solution optimale au problème de couverture. En effet, si ce n'était pas le cas, on pourrait prendre une meilleure solution au problème de couverture, la retransformer en solution du problème de flot, et on contredirait l'optimalité du résultat du max-flow/min-cost.

5. Il suffit de poser la capacité résiduelle comme dans le cas "normal" : $c_f(u,v) = c(u,v) - f(u,v)$. Ceci peut faire apparaître des arêtes dans G_f qui n'existent pas dans le graphe original. Par exemple, si dans G on a $(u,v) \in E$, et $c(u,v) > 0$, mais $(v,u) \notin E$. Dans ce cas, on a dans le résiduel $c_f(u,v) = c(u,v) - f(u,v)$ et $c_f(v,u) = f(u,v)$, ce qui peut être strictement positif. Dans ce cas, il faut poser $cost_f(v,u) = -cost_f(u,v) = -cost(u,v)$. L'idée est de maintenir l'intuition qu'on peut pousser le flot "dans le sens inverse" dans le graphe résiduel, et qu'il faut faire les économies correspondantes.
6. Une fois le flot maximal calculé, on sait qu'il n'existe plus de chemin entre s et t dans le graphe résiduel. Il peut cependant exister des cycles le long desquels on pourrait pousser (ou retirer) du flot. Ca ne change en rien la valeur totale du flot. Par contre, ça peut en modifier le coût. En effet, s'il existe un cycle tel que la somme des coûts sur ses arêtes est négative, alors faire passer le plus de flot possible dans ce cycle ne change pas la valeur du flot, mais fait baisser son coût !
En d'autres termes, la présence d'un cycle de coût négatif dans le graphe signifie que le flot n'est pas de coût minimal. La contraposé de cette affirmation est que si le flot est de coût minimal, alors il n'y a pas de cycle de coût minimal. L'implication inverse serait particulièrement intéressante, car elle donnerait la base d'un algorithme, mais comme elle est plus dure à prouver, c'est l'objet des questions qui suivent.
7. Pour trouver un cycle de coût négatif, on lance une recherche de plus courts chemins avec pour origine s, et en considérant les coûts comme poids des arêtes. Comme les poids peuvent être négatifs, on utilise l'algorithme de Bellman-Ford. Cet algorithme détecte la présence de cycles négatifs (quand il y en a, le plus court chemin n'est pas défini !). Je rappelle l'algorithme en question (qui est discuté dans [1], au passage).

```
1: procedure Bellman-Ford(V, E, s)
2:     for each v ∈ V do d(v) ← ∞,   π(v) ← ⊥
3:     d(s) ← 0
4:     for i = 1 to |V| − 1 do
5:         for each (u, v) ∈ E do
6:             if d(v) > d(u) + poids(u, v) then
7:                 d(v) ← d(u) + poids(u, v)
8:                 π(v) ← u
9:     // Maintenant on détecte les cycles négatifs
10:    for each (u, v) ∈ E do
11:        if d(v) > d(u) + poids(u, v) then Cycle négatif !
12: end procedure
```

Il n'est pas forcément évident de voir pourquoi l'algorithme peut dire "cycle négatif". En fait, ce qui est "facile" à voir, c'est qu'il n'est pas possible qu'il dise "cycle négatif" s'il n'y a pas de cycle négatif. En effet, après i itérations de la boucle extérieure, tous les plus courts chemins de longueur inférieure ou égale à i sont corrects s'il n'y a pas de cycles négatifs. Comme ils peuvent être de longueur au plus n, ils tous sont censés être corrects. Si l'un d'entre eux est incorrect, alors il y a eu un cycle négatif.
Dans l'autre sens, s'il y a un cycle négatif, alors même si on effectuant un nombre arbitrairement grand d'itérations de la boucle extérieure, on aurait toujours une arête "sous tension" (telle que $d(v) > d(u) + poids(u,v)$), mais je vous laisse le démontrer puisque c'est classique.
En fait, on peut modifier un peu l'algorithme pour qu'il fasse $2V$ itérations de la boucle principale. Dans ce cas, les "parents" π qui définissent normalement un arbre des plus courts chemins depuis s, contiennent alors un cycle. En effet, s'il y a un cycle de coût négatif $(u_1, \dots, u_k)$, alors ce cycle est de longueur inférieure ou égale à n, et l'un des sommets du cycle est accessible par un chemin de taille au plus n depuis le point de départ. Notons u_0 le sommet du cycle qui le premier reçoit un parent $\pi(u_0) \neq \perp$. Alors, au pire k itérations plus tard, il recevra un nouveau parent qui est son prédécesseur dans le cycle, car il est plus court d'aller de s à u_0 en faisant un tour du cycle que directement (vu que le cycle est de coût négatif).
Il s'agit donc de détecter le cycle dans π. Ceci est assez simple, et même très très très naïvement on peut le faire en temps $\mathcal{O}(V^2)$, qui est plus petit que $\mathcal{O}(VE)$ a priori.
8. On prend le graphe résiduel associé à f, qu'on appelle G_f. On prétend que f' est une circulation légitime dans le graphe résiduel G_f. On note $F = |f| = |f^*|$ la valeur du flot commune à f et f^*. D'abord, on observe que f' est bien symétrique :

$$f'(u,v) = f^*(u,v) - f(u,v) = f(v,u) - f^*(v,u) = -f'(v,u)$$

Puis, on vérifie que c'est bien une circulation. Pour cela, on peut écrire les équations de conserva-

tion des flots f et $f'*$:

$$\forall v \in V. \quad \sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u) = x_v$$
$$\forall v \in V. \quad \sum_{(u,v)\in E} f^*(u,v) - \sum_{(v,u)\in E} f^*(v,u) = x^*_v$$

où x_v vaut zéro sur les sommets normaux, vaut $-F$ pour la source, et F pour le puits. Il n'est pas difficile de voir que si on fait la différence, on trouve :

$$\forall v \in V. \quad \sum_{(u,v)\in E} f'(u,v) - \sum_{(v,u)\in E} f'(v,u) = 0$$

Il reste enfin à démontrer que cette circulation ne viole pas les contraintes de capacité du résiduel de f. On rappelle que dans le résiduel G_f, la capacité d'une arête $c_f(u,v)$ est exactement $c_f(u,v) = c(u,v) - f(u,v)$ (et cette grandeur est toujours positive). Il s'agit donc de vérifier qu'on a toujours $f'(u,v) \leq c_f(u,v)$. Or ceci s'écrit : $f^*(u,v) - f(u,v) \leq c(u,v) - f(u,v)$, et c'est donc immédiatement équivalent à $f^*(u,v) \leq c(u,v)$, ce qui est vrai vu que f^* est un flot acceptable de G.
Il reste à se convaincre que f' est de coût négatif. Si on écrit ce qu'est le coût de f', on va trouver $cost(f') = cost(f^*) - cost(f) < 0$, vu que f^* est de coût minimum mais pas f.

9. On part d'une arête (u,v) quelconque du graphe résiduel G_f, avec $f'(u,v) > 0$. Par conservation de la circulation, comme $f'(u,v)$ unités de flot entrent dans v, il faut bien qu'elles en sortent. Il existe donc une autre arête (v,w) dans G_f avec $f(v,w) > 0$. On peut continuer à trouver de nouvelles arêtes de la sorte, mais au bout d'un moment on va ré-atteindre le sommet de départ u, car cette procédure ne peut pas continuer indéfiniment. On a donc trouvé un cycle le long duquel circule un flot strictement positif.
10. On sait qu'il existe un cycle de circulation du flot dans la circulation f', qui est de coût négatif. Alors, ou bien la somme des coûts des arêtes de long de ce cycle est strictement négative, et c'est bon. Ou bien le "coût" du cycle est positif ou nul. Notons C le cycle, et notons $g = \min_{(u,v)\in C} f(u,v)$ la plus petite quantité de flot circulant le long d'une arête du cycle (notez que c'est une grandeur strictement positive).
Le plan global consiste à former une nouvelle circulation f'' en retirant de la circulation (si j'ose dire) g unités de flot le long de chacune des arêtes de C, puis à recommencer le raisonnement sur f''.
Il faut d'abord vérifier que f'' est bien une circulation valide du graphe résiduel G_f. On peut déjà affirmer que la contrainte de conservation ne sera pas violée, puisque le bilan de l'opération est nul pour chacun des sommets de C, et que les autres sommets ne sont pas affectés. Je vous laisse la symétrie en exercice. Reste la question des capacités. les seuls problèmes peuvent se poser sur les arêtes du cycle. Pour chaque arête de C, il faut vérifier que $f'(u,v) - g \leq c_f(u,v)$. Ceci est trivialement vrai. Mais il faut aussi s'assurer que tout se passe bien sur les arêtes du graphe résiduel qui sont en sens inverse de celles du cycle. Là, il faut s'assurer que $-f'(u,v)+g \leq c_f(v,u)$. Seulement, comme g est le min des $f(u,v)$ le long du cycle, on trouve que $-f'(u,v) + g \leq 0$, et tout va bien.
On argumente que le coût de la nouvelle circulation f'' est inférieur ou égal à celui de f' (qui était déjà négatif). En effet, on a retiré une quantité strictement positive de flot le long d'un cycle de coût total positif ou nul. Il en résulte que f'' est aussi une circulation de coût total négatif de G_f. Ensuite, le point-clef c'est qu'il y a une arête de moins dans f'' que dans f'. En effet, dans le cycle C, il y a au moins une arête le long de laquelle circulait g unité(s) de flot, par définition de g. Dans f'', plus aucun flot ne circule le long de cette arête particulière.
Il s'ensuit qu'on ne peut pas "retirer" infiniment des cycles positifs à la circulation de depart. Il y a forcément un moment où on va trouver un cycle négatif. CQFD.
11. Chaque fois qu'on "annule" un cycle strictement négatif en le saturant de flot, on fait baisser le coût global d'au moins un. Il en résulte que le nombre d'itérations de cette procédure est borné par le coût de la solution est borné par la différence entre le coût du flot maximal de départ et le coût du flot de coût minimal... On peut dire quelque chose d'un peu plus précis en notant C la capacité maximale d'une arête (si une arête a une capacité infinie, on peut la remplacer par la somme des capacités des autres arêtes), et D le maximum des valeurs absolues des coûts des arêtes. Très clairement, le flot maximal est majoré par C, tandis que le coût de n'importe quel flot est compris entre $-ECD$ et ECD. Il s'ensuit que le nombre de cycles qui peut être annulé est majoré par $2ECD$.

12. Si on utilise un algorithme de préflot pour trouver le flot maximal initial, alors la complexité totale va être en $\mathcal{O}\left(V^3 + VE^2CD\right)$, ce qui est exponentiel (puisque représenter une capacité C ne demande que $\log C$ bits). Cependant, dans notre cas précis, où les capacités et les coûts sont unitaires, on peut s'en tirer un peu mieux, en $\mathcal{O}\left(V^3 + VE^2\right)$. Si on exprime ça en terme du nombre de case noire, on trouve $\mathcal{O}\left(N^5\right)$, ce qui, en terme de taille de la matrice, fait.... $\mathcal{O}\left(n^{10}\right)$!

Solution de l'exercice 2

1. L'idée est de dire que la partition est une liste doublement chaînée de classes, et que chaque classe est une liste doublement chaînée d'éléments de Ω. Plus précisément, chaque élément contient
   – un pointeur vers l'élément suivant dans sa classe (ou un pointeur nul si c'est le dernier).
   – un pointeur vers l'élément précédent dans sa classe (ou un pointeur nul si c'est le premier).
   – un pointeur vers sa classe elle-même.
   Chaque classe contient
   – un pointeur vers la classe suivante dans la partition (ou un pointeur nul si c'est la dernière).
   – un pointeur vers la classe précédente dans la partition (ou un pointeur nul si c'est la première).
   – un pointeur vers son premier élément.
   – son nombre d'élément
   – un autre entier (qui est initialisé à zéro et qui sert à mesure la taille de $\mathcal{C} \cap X$).
   – un booléen qui signifie "déjà traité" et qui est initialisé à zéro.
   L'initialisation et le fait de renvoyer la partition sont (relativement) triviaux
   Pour effectuer le raffinage, on commence par déterminer les cardinalités des classes résultant du raffinage. Pour cela, on parcourt X. Pour chaque élément on trouve sa classe (grâce au pointeur ad hoc). On décrémente la taille de la classe, et on incrémente l'autre entier (celui qui est censé mesurer la taille de $\mathcal{C} \cap X$). On déplace l'élément au début de la liste doublement chaîne qui représente sa classe. Tout ceci peut se faire en temps $\mathcal{O}(1)$ par élément. On rappelle que l'insertion/la suppression dans une liste chaînée se font en modifiant deux pointeurs. Il faut aussi modifier le pointeur sur le premier élément de la classe en question.
   Après cette première phase, on sait que dans chaque classe les éléments de $\mathcal{C} \cap X$ sont au début de la liste, et on connaît leur nombre.
   On re-parcourt X une seconde fois. Pour chaque élément de X on trouve sa classe. Si le booléen "déjà traité" est **false**, alors on créée une nouvelle classe dans la partition pour recueillir les éléments de $\mathcal{C} \cap X$. On copie tous les éléments de $\mathcal{C} \cap X$ dans la nouvelle classe. C'est facile parce qu'on connaît leur nombre et que ce sont les premiers de la liste des éléments de $\mathcal{C}$. On les retire de la liste des éléments de l'ancienne classe. On marque la nouvelle classe "déjà traitée". On détermine si l'ancienne classe (qui représente alors $\mathcal{C} - X$) est vide. Si oui, on la retire de la liste.
   A ce stade, c'est presque fini. Les nouvelles classe sont apparues, les classes vides ont été supprimées. Il reste à effectuer une troisième passe sur X pour mettre à **false** tous les booléens "déjà" traité.
   Plus concrètement, au lieu d'avoir des pointeurs, on peut s'en tirer avec des tableaux d'entiers qui donnent pour chaque élément (et pour chaque classe), le successeur, le prédécesseur, etc.
2. Il suffit de raffiner la partition par $\{v\}$ si on veut supprimer l'élément v....
3. Si u et v ne sont pas de faux jumeaux, alors il existe (sans perte de généralité) un sommet w qui est un voisin de u et pas de v. Le raffinage par $N(w)$ va alors couper la classe $\mathcal{C}$ qui contenait u et v en deux : d'un côté $\mathcal{C} \cap N(w)$ contiendra u, et $\mathcal{C} - N(w)$ contiendra v.
4. La partition initiale $\mathcal{P} = \{V\}$ est compatible. On démontre ensuite que les opérations de raffinage la laissent compatible. Supposons que u et v soient des jumeaux dans la même classe, et qu'on raffine par $N(w)$.
   Si w est un voisin de u, alors c'est également un voisin de v par définition. Il s'ensuit que $\{u, v\} \in N(w)$, et si on note $\mathcal{C}$ la classe de u et v, alors après l'opération de raffinage, on aura $\{u, v\} \subseteq \mathcal{C} \cap N(w)$, et la propriété est vérifiée. Ensuite, si w n'est pas un voisin de u, alors ce n'est pas non plus un voisin de v. Il s'ensuit qu'après le raffinage, on aura $\{u, v\} \subseteq \mathcal{C} - N(w)$. CQFD.
5. La complexité de l'algorithme, est la complexité d'effectuer les $|V|$ opérations de raffinage. La somme des tailles des voisinages de tous les sommets est égale au double du nombre d'arête (chaque arête étant parcourue dans les deux sens). Il s'ensuit que la complexité est en $\mathcal{O}(V + E)$.
6. Dans le cas des vrais jumeaux, le seul changement à apporter à l'algorithme est qu'il faut raffiner par $\{v\} \cup N(v)$. La complexité ne change pas. Adapter les preuves n'est pas compliqué.

Solution de l'exercice 3

1. Le problème consiste à déterminer quel est le plus grand nombre d'intervalles qui ont une intersection non-vide. Si on place tous les intervalles sur la droite réelle, le problème consiste à déterminer quel est le plus grand nombre qui se superposent. Pour cela, on suppose que les intervalles sont $[s_i, t_i]$, et on construit une liste (ou un tableau, peu importe) contenant les paires $(s_i, 1)$ ainsi que $(t_i, -1)$ pour tout i. Ce tableau a donc $2n$ éléments. On le trie selon la première composante des paires avec un tri en $\mathcal{O}(n \log n)$, tel que le tri fusion. Ensuite, on initialise un compteur k à zéro, et on balaye le tableau trié. Pour chaque paire (x_i, δ_i), on pose $k \leftarrow k + \delta_i$. Le moment où k est maximal révèle la plus grande clique.
2. Par l'absurde. On suppose l'existence d'un cycle $(x_1, ..., x_k)$ de longueur $k \geq 4$ sans corde. On sait qu'il est possible d'étiqueter les sommets par des intervalles comme annoncé au début. Notons ces intervalles $[s_i; t_i]$. L'existence du cycle garantit que le i-ème intervalle et le $(i+1)$-ème ont une intersection non-vide, et l'absence de corde nous garantit que le i-ème et le $(i+2)$-ème sont disjoints. On en déduit donc que :

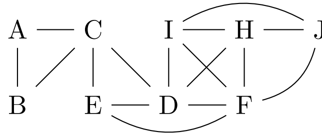
$$\begin{aligned} s_2 &\leq t_1 & t_1 &< s_3 \\ s_3 &\leq t_2 & t_2 &< s_4 \\ &\vdots & &\vdots \\ s_{k-1} &\leq t_{k-2} & t_{k-2} &< s_k \\ s_k &\leq t_{k-1} & t_{k-1} &< s_1 \\ s_1 &\leq t_k & t_k &< s_2 \end{aligned}$$

   Et il en découle que $s_2 < s_3 < \cdots < s_k < s_1 < s_2$. Contradiction.
3. Prenons une clique de taille maximale du graphe, et appelons la K. Notons v le sommet de K qui porte le plus petit numéro dans le perfect elimination ordering σ. Par définition, K est contenue dans $\{v\} \cup N(v)$, puisque v est adjacent à chacun des sommets de K. Maintenant, comme v est le sommet de K de numéro minimal, il s'ensuit que K est contenue dans $\{v\} \cup RN(v)$. Comme σ est un perfect elimination ordering, cet ensemble forme une clique. Puisque K est de taille maximale, il y a en fait égalité. Voilà donc la stratégie : mesurer la taille de chacun des $RN(v)$ pour trouver le plus gros.
   Reste à voir comment ceci peut se faire avec la complexité demandée. En fait, ce n'est pas difficile. Pour chacun des sommets, on compte le nombre de ses voisins qui a un numéro plus grand. Ceci se fait en parcourant chaque arête une fois. Au total, chaque arête est parcourue deux fois (une fois dans chaque sens). On peut donc facilement obtenir une complexité de $\mathcal{O}(V + E)$ avec un pseudo-code du style :

```
1: function MaxCliqueInterval(G, σ)
2:     max ← 0
3:     for each v ∈ V do
4:         k ← 0
5:         for each u ∈ V t.q. {u, v} ∈ E do
6:             if σ(u) > σ(v) then k ← k + 1
7:         if k > max then max ← k
8:     end for
9:     return max
10: end function
```

   Cette fonction ne retourne que la taille de la clique max, mais elle est facile à adapter pour renvoyer la clique max elle-même.
4. On ne rappelle pas le principe du parcours qui fait partie du cours. Supposons qu'on ait la situation décrite dans l'énoncé. Le sommet a est marqué avant b, qui est lui même marqué avant c, et $\{a, b\} \notin E$ tandis que $\{a, c\} \in E$.
   Lors du marquage de a, qui était alors le premier sommet de la liste des sommets à traiter, ses voisins non-marqués sont ajoutés à la fin de la liste des sommets à traiter. En particulier, c est alors inséré dans la liste. Mais b ne peut pas rentrer dans la liste en même temps que c, puisqu'il n'est pas adjacent à a. Puisque $a < b < c$, alors sort de la liste après a, mais avant c, ce qui signifie qu'il se trouvait déjà dans la liste au moment du marquage de a. La seule explication possible à la présence de b dans la liste au moment du marquage de a est l'existence d'un autre sommet d, marqué avant a, dont b est un voisin. Au moment du marquage de d, b rentre dans la liste. On peut noter que le sommet a se trouve déjà dans la liste à ce moment-là, mais ça ne change rien.
5. Voici le résultat, en partant du sommet A

Sommet	numéro affecté	label
A	8	8
B	6	8,7
C	7	8
D	5	7
E	4	7,5
F	3	5,4
H	2	5,3
I	1	5,3,2
J	0	3,2,1



6. Quelle que soit la manière dont on tourne cette phrase de Wikipedia, elle est fausse. D'abord, on observe qu'à la fin de l'algorithme, le label d'un sommet contient la liste des sommets plus grands que lui (sélectionnés avant) qui lui sont adjacents. Dans la matrice d'adjacence, le label du i-ème sommet (dans l'ordre LexBFS) décrit donc la partie de la i-ème ligne située à gauche de la diagonale (ou bien, ce façon équivalente, la partie de la i-ème colonne située au-dessus de la diagonale. Si LexBFS mettait les lignes dans l'ordre lexicographique décroissant, alors l'ensemble des labels devrait forcément lui aussi être dans l'ordre lexicographique décroissant. On voit bien dans l'exemple donné dans le sujet que ce n'est pas le cas. Par exemple, le label de B (7) est plus petit que le label de C (7,6), qui est pourtant sélectionné après.
7. On peut voir LexBFS un peu comme le BFS normal. Les sommets qui possèdent un label différent de $\emptyset$ sont ceux qui sont dans la liste des sommets à traiter. L'ordre des labels précise leur ordre dans la file.
   a) Pour comprendre les choses, il est nécessaire de commencer par démontrer que pendant le fonctionnement de LexBFS, si jamais deux sommets reçoivent des labels distincts, leur classement relatif ne changera plus au court de l'algorithme. Ceci est facile à démontrer par récurrence sur le nombre d'itération.
   b) Notons ℓ_a, ℓ_b et ℓ_c les labels de a, b et c au moment où a est sélectionné. Comme a est sélectionné en premier, c'est que ℓ_a est supérieur ou égal à ℓ_b et ℓ_c. La mise à jour des labels consécutive au marquage de a ne modifie pas le label de b, mais modifie le label de c, qui devient ℓ_c, i
   On affirme que ℓ_b est en fait strictement plus grand que ℓ_c. En effet, au début de l'itération qui suit la sélection de a, b et c ont forcément des labels *différents* (celui de c contient i et pas celui de b). L'un est donc strictement plus petit que l'autre, et leur ordre relatif ne peut pas changer. Comme b est sélectionné avant c, c'est que $\ell_b \succ \ell_c$.
   c) Considérons le plus long préfixe commun p de ℓ_b et ℓ_c. On peut alors écrire :
   $$\ell_b = p.\ell'_b$$
   $$\ell_c = p.\ell'_c$$
   Très clairement, le premier élément de ℓ'_b est strictement supérieur au premier élément de ℓ'_c (qui peut aussi éventuellement être vide). Notons j le premier élément de ℓ'_b. Puisque le label de b contient j, c'est qu'il y a eu un sommet marqué (avant a donc), portant le numéro $j > i$, qui est un voisin de b, et dont la sélection a entraîné l'ajout de j au label de b. Le sommet qui porte le numéro j est donc d. Ceci démontre que LexBFS satisfait la propriété (L).
   d) Il reste maintenant à démontrer que d et c ne sont pas voisin.
   Déjà, on affirme qu'au moment du marquage de d, les deux sommets b et c portent le label p (leur plus long préfixe commun). Ceci provient du fait que tout ce que contient p est strictement plus grand que j, donc a nécessairement été affecté avant (puisque ça a été affecté à un moment et que ça ne pourra pas être affecté après).
   On raisonne ensuite par l'absurde, en supposant que c et d sont voisins. Suite au marquage de d, les deux sommets b et c devraient se retrouver avec le label p, j, mais ceci contredirait le fait que p est leur plus long préfixe commun.
8. Si on fait les choses très naïvement, alors pour trouver le prochain sommet à marquer, on va calculer le max des labels des sommets restants, ce qui se fait avec au pire n comparaisons (selon $\prec$). Cela fera donc $\mathcal{O}(n^2)$ comparaisons selon $\prec$ au total pendant l'ensemble du déroulement de l'algorithme. Cependant, une comparaison selon $\prec$ peut nécessiter jusqu'à n comparaisons entières. On aboutirait donc à quelque chose en $\mathcal{O}(V^3 + E)$. Le terme en E provient de ce que la procédure parcours chaque arête au pire deux fois, et au fait que la mise à jour de labels a lieu au pire E fois. Elle peut se faire en temps constant si les labels sont représentés comme des listes doublement chaînées (ce qui permet l'insertion à la fin en temps constant, tout comme l'insertion au début).

Il est possible de faire un peu mieux de différentes manières. La plus simple consiste à utiliser une file de priorité, genre tas binaire ou tas de Fibonacci. Mettre à jour les labels est une opération Augmenter-Clef (il y en aura $\mathcal{O}(E)$), tandis que récupérer le prochain sommet à marquer est une opération Extraire-Max (il y en aura V). Dans les deux types de tas, Extraire-Max est en $\mathcal{O}(\log n)$ comparaisons. Dans les tas binaires, Augmenter-Clef est en $\mathcal{O}(\log n)$ comparaisons, et dans les tas de Fibonacci en $\mathcal{O}(1)$ comparaisons (amorti).
Avec des tas binaires, on arrive à une complexité de $\mathcal{O}((V+E))V \log V)$, et avec des tas de Fibonacci, on trouve quelque chose comme $\mathcal{O}\left(V^2 \log V + EV\right)$.

9. Comme annoncé, on garde au "début" de la partition les sommets qui doivent être traités d'abord (ceux dont les labels sont les plus grands). Il est alors possible d'accéder au prochain sommet à traiter en temps $\mathcal{O}(1)$. On peut également le supprimer de $\mathcal{P}$ en temps constant(on a vu ça dans l'exercice 2). L'idée est que si le sommet marqué est v, alors on raffine par $N(v)$. Par contre, lorsqu'on casse une classe $\mathcal{C}$, on prend soin de mettre $\mathcal{C} \cap N(v)$ *avant* $\mathcal{C} - N(v)$. En effet, si les sommets de $\mathcal{C}$ portent un label ℓ, et que le numéro de v est i, alors les sommets de $\mathcal{C} \cap N(v)$ porteront le label ℓ, i qui est plus grand que le label ℓ que vont continuer à porter ceux de $\mathcal{C} - N(v)$. Il est à noter qu'on a même pas besoin de maintenir explicitement les labels.
Voilà pour l'intuition. On passe au pseudo-code :

```
1: function Fast-LexBFS(G, s)
2:     P ← {V}
3:     Refine(P, {s})   [C ∩ X d'abord]
4:     for i from n to 1 do
5:         choisir un sommet v de la première classe de P
6:         Retirer v de P
7:         σ(i) ← v
8:         Refine(P, N(v))   [C ∩ X d'abord]
9:     end for
10:    return σ
11: end function
```

Démontrons maintenant que cette procédure est correcte. On va démontrer par récurrence sur le nombre d'itérations que :

i) tous les éléments de la même classe ont le même label.

ii) les classes sont toujours classées par ordre de labels décroissants.

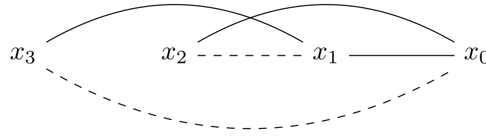
iii) la partition ne contient que des sommets non-traités.

Ceci suffit à conclure, car alors choisir un élément de la première classe revient à choisir un sommet de label maximal (même si les labels ne sont même pas calculés !). Cela montre que le pseudo-code ci-dessus "simule" correctement l'algorithme LexBFS.
Tout d'abord, au début de l'algorithme, le premier Refine aboutit à une partition $\mathcal{P} = \{s\}, V - \{s\}$. Comme le sommet s porte le label $\{s\}$ et que les autres ne portent aucun label, les hypothèses de récurrence sont vraies au début de la boucle **for**. Reste à montrer qu'elles sont encore vraies après une itération. Comme annoncé, on est convaincu que le sommet v choisi est un sommet de label maximal (vu les hypothèses de récurrence).
Comme v est retiré de la partition, l'hypothèse *iii* est trivialement satisfaite. La sélection de v doit ajouter i aux labels des voisins non-marqués de v. Prenons un voisin de v. Par HR, il partageait le même label que les autres sommets de sa classe au début de la boucle. Il faut donc que ça change. Mais c'est exactement ce que le Refine effectue : il sépare de leur classe d'origine tous les sommets voisins de v. Ceci garantit que l'hypothèse i est préservée. Reste à s'assurer que les classes sont encore bien ordonnées. Mais en mettant les voisins de v *avant* ceux qui portaient le même labels qu'eux au début de la boucle, on fait bien les choses : l'ajout d'un élément au label le fait grossir, donc les sommets voisins de v doivent passer avant ceux qui portaient le même labels mais qui ne sont pas adjacents à v. Par ailleurs, on a déjà démontré auparavant que si deux sommets ont des labels différents, alors leur ordre relatif ne change pas. Cela signifie que les voisins de v ne doivent pas passer avant des sommets qui portaient déjà un label plus grand au début de la boucle. Leur nouvelle position dans la partition est donc correcte.

10. On raisonne par l'absurde en supposant que LexBFS ne renvoie pas un perfect elimination ordering sur un graphe d'intervalle. Il y a donc au moins un sommet dont le voisinage à droite n'est pas une clique. On considère celui qui est sélectionné en dernier par LexBFS (donc qui porte le plus petit numéro), et on l'appelle x_0. Puisque son voisinage à droite n'est pas une clique, alors x_0 possède deux voisins dont les numéros sont plus grands et qui ne sont pas adjacents. Appelons-les x_1 et x_2. On va supposer sans perte de généralité que le numéro de x_2 est plus grand que celui

de x_1, et on peut également choisir que x_2 soit le plus grand sommet du graphe adjacent à x_0 et non-adjacent à un voisin à droite de x_0.

11. La propriété (LexB) nous garantit l'existence d'un sommet x_3, plus grand que x_2, tel que $\{x_3, x_1\} \in E$ et $\{x_3, x_0\} \notin E$.



Si $\{x_2, x_3\} \in E$, alors on voit bien qu'il existerait un cycle (x_0, x_1, x_3, x_2) sans corde. Par conséquent (comme le graphe est chordal), on déduit que $\{x_2, x_3\} \notin E$.

12. Passons au raisonnement par récurrence. L'existence d'un sommet plus grand que x_n, adjacent à x_{n-1} mais pas à x_{n-2} est garantie la propriété (LexB). On pose donc que x_{n+1} est le plus grand sommet qui satisfait ces deux conditions. Il reste à démontrer que x_{n+1} n'est pas adjacent à aucun autre x_i.
D'abord, on observe que si x_{n+1} est adjacent à un x_i, avec $i \neq n-3$, alors il se forme un cycle de longueur supérieure ou égale à quatre. En effet, si i et $n+1$ ont la même parité, alors le cycle est :

$$(x_{n+1}, x_{n-1}, \ldots, x_{i+2}, x_i).$$

Si $n+1$ et i n'ont pas la même parité, alors on commence par observer qu'il existe un chemin entre x_i et x_{i-1} (ou x_{i+1}, peu importe), qui passe par x_0 et x_1, ce qui nous ramène au cas précédent. Les hypothèses i et ii impliquent que ce cycle n'a pas de corde. Aussi, comme le graphe est chordal, on a une contradiction. On en déduit qu'il n'existe pas d'arête entre x_{n+1} et x_i si $i \neq n-3$.
Reste à traiter le cas où $i = n-3$. En effet, si $\{x_{n+1}, x_{n-3}\} \in E$, il se forme un cycle de longueur 3 seulement, et on ne peut pas invoquer la chordalité du graphe pour conclure. Seulement à ce moment-là, la propriété (LexB) appliquée avec $a = x_{n+1}, b = x_{n-2}$ et $c = x_{n-3}$ garantit l'existence d'un sommet, qu'on va appeler z, plus grand que x_{n+1}, adjacent à x_{n-2} et pas à x_{n-3}. Or, si $z > x_{n+1}$, a fortiori $z > x_n$, et donc ceci contredit la maximalité de x_n, qui est garantie par l'hypothèse *iv*. CQFD.

13. Ceci permet de conclure, car on pourrait fabriquer une séquence infinie de sommets satisfaisant les hypothèses (et donc des sommets distincts !), alors que le graphe est fini.

Références

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction à l'algorithmique*. Dunod, 2nd edition, 2001.