UE LU3IN003 - Algorithmique.
Licence d'informatique.

Partiel du 4 décembre 2020. Durée : 1h45

Documents non autorisés. Seule une feuille A4 portant sur les cours est autorisée.
Téléphones portables éteints et rangés dans vos sacs.

Exercice 1 (5 points)

Soit $G = (S, A)$ un graphe non orienté *connexe*. L'objet de cet exercice est de concevoir un algorithme qui détermine si G est *biparti*. Un graphe est biparti si l'ensemble S peut être partitionné en deux sous-ensembles S_1 et S_2 (avec $S_1 \cup S_2 = S$ et $S_1 \cap S_2 = \emptyset$) tel qu'il n'existe pas d'arêtes entre sommets d'un même sous-ensemble (par exemple, si $(x, y) \in S_1^2$ alors il n'existe pas d'arête entre x et y).

Soit $L = (s_1, \ldots, s_n)$ un parcours de G à partir d'un sommet quelconque s_1 de S et $F(L)$ la forêt couvrante associée à L. On colore (en rouge ou bleu) tous les sommets de S selon la règle de coloration suivante :

- le sommet s_1 est bleu ;
- pour $i = 2, \ldots, n$, le sommet s_i est bleu (resp. rouge) si son père dans $F(L)$ est rouge (resp. bleu).

Le but de cet exercice est de montrer que G est biparti si et seulement si il n'existe pas d'arête de A entre deux sommets de même couleur.

Question 1 (0.5/5) — Montrer que s'il n'existe pas d'arête entre deux sommets de même couleur dans $A \setminus F(L)$, alors le graphe G est biparti.

On note S_r le sous-ensemble des sommets rouges et S_b le sous-ensemble des sommets bleus. Par construction, il n'existe pas d'arête entre sommets de même couleur dans $F(L)$. Si de plus il n'existe pas d'arête entre sommet de même couleur dans $A \setminus F(L)$, on n'a donc aucune arête de A entre sommets de même couleur. En prenant $S_1 = S_r$ et $S_2 = S_b$ on vérifie que le graphe est biparti.

Question 2 (1.5/5) — Soit G un graphe biparti. Montrer que :

a) s'il existe une chaîne de longueur paire (en nombre d'arêtes) dans G entre deux sommets x et y alors ces deux sommets x et y appartiennent nécessairement au même sous-ensemble (S_1 ou S_2) dans la partition des sommets ;

b) s'il existe une chaîne de longueur impaire dans G entre deux sommets x et y alors l'un appartient à S_1 et l'autre à S_2.

Soit $\mu = (x_1 = x, \ldots, x_p = y)$ une chaîne entre x et y. Sans perte de généralité, supposons $x_1 \in S_1$. On a nécessairement $x_2 \in S_2$ (car il n'existe pas d'arête entre deux sommets de S_1), $x_3 \in S_1$ (car il n'existe pas d'arête entre deux sommets de S_2), et ainsi de suite...

Preuve du a). Si la chaîne μ est de longueur paire, on a $x_p \in S_1$. Donc x_1 et x_p appartiennent au même sous-ensemble. *Preuve du b).* Si la chaîne μ est de longueur impaire, on a $x_p \in S_2$. Donc x_1 et x_p appartiennent l'un à S_1 et l'autre à S_2.

Question 3 (1.5/5) — Soit L un parcours de G. Après coloration, on considère une arête $\{x, y\}$ entre deux sommets de même couleur. En considérant la chaîne qui relie x et y dans $F(L)$, montrer que le graphe n'est pas biparti.

Par l'absurde. Supposons que G est biparti. Les sommets le long de la chaîne reliant x et y alternent de couleur. Comme x et y sont de même couleur, on en déduit que la chaîne est de longueur paire. Donc x et y appartiennent au même sous-ensemble d'après le a) de question 3. Or il existe une arête entre x et y. Contradiction.

Question 4 (1.5/5) — Compléter la procédure ci-dessous pour qu'elle permette de détecter si un graphe est biparti en $O(n + m)$. La variable globale *est_biparti* est initialisée à vrai. La valeur de cette variable à l'issue de la procédure doit être vrai si le graphe est biparti, et faux sinon. Le tableau C indexé sur les sommets indique la couleur des sommets. Il est initialisé de la façon suivante : pour tout x, $C[x] =$ incolore. Etant donné un sommet s_1 de départ, le premier appel de la procédure est $Parcours\_Rec(G, s_1, bleu)$.

Parcours_Rec(G : Graphe, s : Sommet, c : Couleur)

```
1  variable locale x : Sommet;
2  C[s] ← c;
3  pour tous les voisins x de s faire
4  |  si C[x] = C[s] alors
5  |  |  est_biparti ← faux;
6  |  fin
7  |  si C[x] = incolore alors
8  |  |  si C[s] = bleu alors
9  |  |  |  Parcours_Rec(G, x, rouge)
10 |  |  fin
11 |  |  sinon
12 |  |  |  Parcours_Rec(G, x, bleu)
13 |  |  fin
14 |  fin
15 fin
```

Exercice 2 (7 points)

Considérons un graphe qui représente un réseau social : les sommets sont des personnes et les arêtes entre deux sommets représentent les liens d'amitié (ou de connaissance) entre les personnes

correspondant à ces sommets. Suivant l'adage "les amis des mes amis sont mes amis", il est probable que si la personne a connaît deux personnes b et c alors b et c se connaissent aussi. Ce lien est en tout cas plus probable que le fait que deux personnes au hasard dans le graphe se connaîssent. Les graphes des réseaux sociaux comportent ainsi un grand nombre de triangles, un triangle étant un ensemble de 3 personnes se connaissant mutuellement. Le but de cet exercice est de concevoir un algorithme qui compte le nombre de triangles dans un graphe, et ce de manière efficace.

Plus formellement : étant donné un graphe $G = (S, A)$ non-orienté, on dira que le triplet de sommets $\{i, j, k\}$ forme un triangle de G si et seulement si les arêtes $\{i, j\}$, $\{j, k\}$ et $\{k, i\}$ appartiennent à A. On notera par la suite $n = |S|$, $m = |A|$ et d_{max} le degré maximum d'un sommet de G.

Question 1 (1/7) — Quelle est la complexité temporelle d'un algorithme vérifiant que le triplet $\{i, j, k\}$ est un triangle de G

a) si G est représenté par une matrice d'adjacence ?

b) si G est représenté par un tableau de listes d'adjacences ?

Vous donnerez la complexité la plus précise possible, et vous justifiez vos réponses en une phrase.

a) Au format matrice d'adjacence, il suffit de vérifier que les arêtes $\{i, j\}$, $\{j, k\}$ et $\{k, i\}$ existent, ce qui se fait en temps constant $\Rightarrow \Theta(1)$.

b) Au format listes d'adjacence, il faut vérifier que j et k sont voisins de i et que j est voisin de k, ce qui se fait en $O(d(i) + d(k))$, ce qui est en $O(d_{\max})$ (et aussi en $O(n)$ mais c'est moins précis).

Question 2 (1.5/7) — On suppose que le graphe G est représenté par une matrice d'adjacence M. Déduisez de la question précédente un algorithme en $\Theta(n^3)$ qui compte le nombre de triangles de G.

Comptage_Matrice(M)

```
1  tr ← 0
2  pour i de 1 à n faire
3  |  pour j de i+1 à n faire
4  |  |  pour k de j+1 à n faire
5  |  |  |  si M(i,j)=1 et M(j,k)=1 et M(k,j)=1 alors
6  |  |  |  |  tr ← tr + 1
7  |  |  |  fin
8  |  |  fin
9  |  fin
10 fin
11 retourner tr
```

Dans de nombreux graphes représentant des réseaux sociaux, on observe en pratique que le degré maximum d_{max} est très petit devant n. Par la suite, nous supposerons que $d_{max} \leq \sqrt{n}$.

Question 3 (0.5/7) — Quelle est la complexité de l'algorithme que vous avez proposé à la question précédente dans ce cas ?

L'algorithme est toujours en $\Theta(n^3)$.

Afin d'améliorer cette complexité, nous allons maintenant considérer que le graphe G est représenté par un tableau de listes d'adjacences. On considère l'algorithme suivant :

Comptage_par_arête(G)

```
1  tr ← 0
2  it ← 1
3  Tab ← tableau de n entiers, initialisé à 0
4  pour toute arête {i, j} de A faire
5  |  pour tout sommet x voisin de i faire
6  |  |  Tab[x] ← it
7  |  fin
8  |  pour tout sommet y voisin de j faire
9  |  |  si Tab[y] = it alors
10 |  |  |  tr ← tr + 1
11 |  |  fin
12 |  fin
13 |  it ← it + 1
14 fin
15 retourner tr/3
```

Question 4 (1.5/7) — Prouver que cet algorithme est valide (c'est-à-dire qu'il retourne bien le nombre de triangles dans le graphe). Vous pourrez pour cela indiquer de quelle valeur est incrémentée la variable tr lors de la k^{ieme} itération de la la première boucle "pour" (ligne 4).

Considérons que les arêtes sont examinées dans l'ordre $a_1, \ldots, a_m$. Lors de la k-ième itération, tr est incrémentée du nombre de triangles ayant pour côté a_k. En effet, supposons que $a_k = \{i, j\}$. A la fin de la ligne 7, une case x du tableau Tab contient la valeur k si et seulement si x est voisin de i. Pour chaque voisin y de j, on vérifie ainsi si le triangle $\{i, j, y\}$ existe (c'est le cas si y est voisin de i, i.e. si $Tab[y] = k$) – et si c'est le cas tr est incrémentée.
Ainsi, à la fin de l'algorithme, chaque triangle est compté 3 fois (une fois lors de l'examen de chacun de ses côtés). La valeur renvoyée par tr est donc égale au nombre de triangles du graphe.

Question 5 (2.5/7) — On se propose dans cette question d'analyser la complexité de l'algorithme.

a) Expliquez en une phrase comment parcourir l'ensemble des arêtes du graphe en $O(n + m)$ (et justifiez cette complexité).
b) En déduire la complexité de cet algorithme en fonction de n, m et d_{max}.
c) Si l'on suppose que $d_{max} \leq \sqrt{n}$, quelle est la complexité de l'algorithme en fonction de n ?

a) On parcourt le tableau de listes d'adjacences en $O(n + m)$. On parcourt le tableau en $O(n)$ et on parcourt les listes d'adjacences de tous les sommets i, pour i allant de 1 à n. Cela se fait en $\Sigma_{i=1}^{n} d(i) = m$.
b) La complexité de l'ensemble des instructions de la première boucle for (lignes 5 à 13) est $O(d(i) + d(j))$, ce qui est en en $O(d_{max})$ car $d(i) \leq d_{max}$ et $d(j) \leq d_{max}$. Cet algorithme est donc en $O(n + md_{max})$.
c) On remarque que $m \leq nd_{max}$ et donc que cet algorithme est en $O(n\ d_{max}^2)$.
Si l'on considère que $d_{max} \leq \sqrt{n}$, alors cet algorithme est en $O(n^2)$.

Exercice 3 (8 points)

Dans cet exercice, on considère un ensemble d'immeubles dont on veut dessiner la ligne de toits. Pour simplifier, on suppose que tous les immeubles correspondent à des rectangles reposant sur un même axe (le sol de la ville, qui est parfaitement plat). Un immeuble est donc caractérisé par sa hauteur $h > 0$, ainsi que par ses deux coordonnées sur cet axe : une coordonnée gauche $g \in \mathbb{N}$ et une coordonnée droite $d \in \mathbb{N}$. On suppose qu'il n'y a pas d'immeuble dont le bord droit se situe au delà de la coordonnée M sur l'axe, autrement dit $M \geq d > g$. Dans la suite de l'exercice, un immeuble est spécifié par un triplet (g, h, d), représentant le rectangle $(g, 0), (g, h), (d, h), (d, 0)$.

Par exemple, pour l'ensemble de 6 immeubles donnés par {(3,13,9), (1,11,5), (19,18,22), (3,6,7), (16,3,25), (12,7,16)} -voir figure A-, la ligne de toits sera donnée par la figure B (traits pleins). Une représentation naturelle de la ligne de toits consiste en une liste de *points clés*, ordonnée par abscisses croissantes, mentionnant les coordonnées où la hauteur change. Par exemple, la ligne de toits sera ici décrite par la liste [(1,11), (3,13), (9,0), (12,7), (16,3), (19,18), (22,3), (25,0)]. Plus généralement, une liste de points $[(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)]$ triée par abscisses croissantes correspond à la ligne tracée en reliant dans l'ordre les points suivants : $(x_1, 0)$, (x_1, y_1), (x_2, y_1), (x_2, y_2), (x_3, y_2), …, (x_i, y_i), (x_{i+1}, y_i), (x_{i+1}, y_{i+1}), …, (x_p, y_{p-1}), (x_p, y_p). Dans une telle liste, deux points consécutifs ont des ordonnées différentes. De plus, $y_1 > 0$ et $y_p = 0$.

L'objet de cet exercice est d'étudier des algorithmes permettant de déterminer la ligne de toits à partir de l'ensemble d'immeubles.

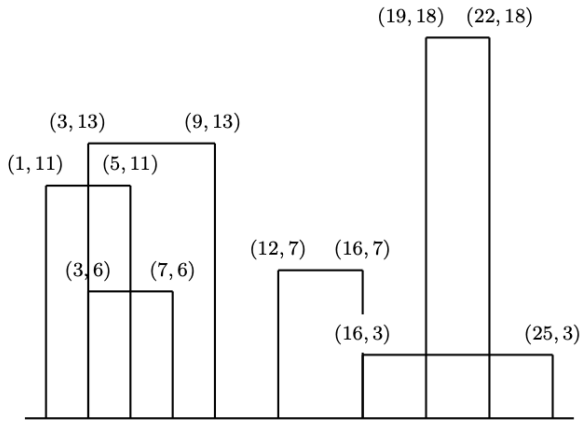


Figure A

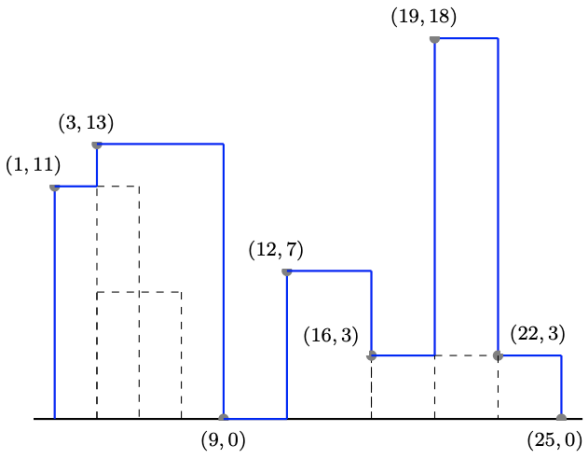


Figure B

Un premier algorithme, appelé Calculer-ligne-1 et dont le pseudo-code est donné plus bas, consiste à parcourir toutes les abscisses de la gauche vers la droite, dans l'ordre $0, 1, \ldots, M$, afin de détecter les points clés où la hauteur de la ligne de toits change.

Question 1 (0.5/8) — A la fin de chaque itération de la boucle *pour* des lignes 3–14, que représente la valeur de y ?

La valeur de y est la hauteur de la ligne de toits au niveau de l'abscisse x.

Calculer-ligne-1(E)

Entrées : un ensemble $E = \{(g_1, h_1, d_1), \ldots, (g_n, h_n, d_n)\}$ d'immeubles
Sortie : la liste décrivant la ligne de toits de l'ensemble E d'immeuble

```
1  ℓ ← [ ]          // ligne de toits initialement vide
2  h ← 0            // hauteur courante de la ligne de toits
3  pour x = 0, ..., M faire          // examen des différentes abscisses x en ordre croissant
4  |  y ← 0
5  |  pour i = 1, ..., n faire
6  |  |  si g_i ≤ x < d_i alors          // l'immeuble i surplombe l'abscisse x
7  |  |  |  si h_i > y alors y ← h_i
8  |  |  fin
9  |  fin
10 |  si y ≠ h alors
11 |  |  Ajouter (x, y) en dernière position de ℓ          // instruction en O(1)
12 |  |  h ← y
13 |  fin
14 fin
15 retourner ℓ
```

Question 2 (1/8) — Déterminer la complexité de l'algorithme Calculer-ligne-1.

La complexité de l'algorithme Calculer-ligne-1 est $O(nM)$.

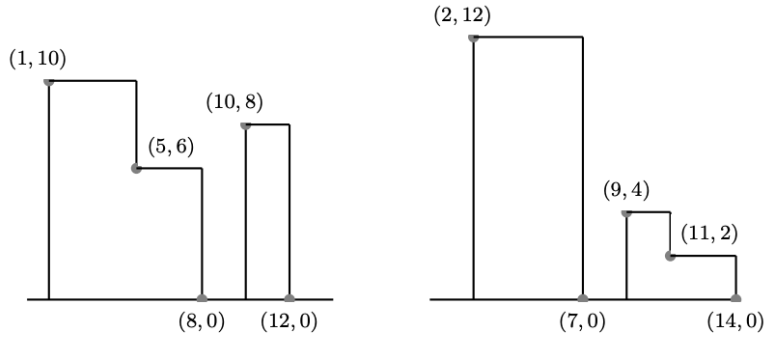
On s'intéresse à présent à un algorithme de type diviser pour régner, appelé Calculer-ligne-2, dont le pseudo-code est donné ci-après, afin de déterminer la ligne de toits d'un ensemble E d'immeubles. Dans la suite de l'exercice, on suppose qu'on accède en $O(1)$ au i-ème élément $\ell[i] = (x_i, y_i)$ d'une ligne de toits $\ell = [(x_1, y_1), \ldots, (x_p, y_p)]$. On accède également en $O(1)$ à $\ell[i].x = x_i$ et $\ell[i].y = y_i$. Enfin, on note $|\ell|$ le nombre d'éléments de ℓ ($|\ell| = p$ pour la ligne de toits précédente).

Calculer-ligne-2(E)

Entrées : un ensemble $E = \{(g_1, h_1, d_1), \ldots, (g_n, h_n, d_n)\}$ d'immeubles
Sortie : la liste représentant la ligne de toits de l'ensemble E d'immeubles

```
1  si n = 0 alors retourner [ ]
2  si n = 1 alors retourner (g_1, h_1), (d_1, 0)
3  sinon
4  |  k ← ⌊n/2⌋
5  |  ℓ_1 ← Calculer-ligne-2({(g_1, h_1, d_1), ..., (g_k, h_k, d_k)})
6  |  ℓ_2 ← Calculer-ligne-2({(g_{k+1}, h_{k+1}, d_{k+1}), ..., (g_n, h_n, d_n)})
7  |  ℓ ← Fusionner-lignes(ℓ_1, ℓ_2)
8  fin
9  retourner ℓ
```

Lors de la mise en œuvre de cet algorithme, on sera donc amené à fusionner deux lignes de toits en une unique ligne de toits. Par exemple, soit la première ligne donnée par [(1,10), (5,6), (8,0), (10,8), (12,0)] et la seconde par [(2,12), (7,0), (9,4), (11,2), (14,0)], comme dans la figure ci-dessous :



Ligne 1 Ligne 2

Question 3 (1/8) — Quelle est la ligne de toits obtenue par fusion dans cet exemple ?

La fusion des lignes des toits est $[(1,10),(2,12),(7,6),(8,0),(9,4),(10,8),(12,2),(14,0)]$.

Question 4 (0.5/8) — Cette question permet de faire une observation utile pour la suivante. Soient ℓ_1 et ℓ_2 deux lignes de toits, x une abscisse, h_1 la hauteur de ℓ_1 en x et h_2 la hauteur de ℓ_2 en x. Soit ℓ la ligne de toits obtenue par fusion de ℓ_1 et ℓ_2. Quelle est la hauteur h de la ligne de toits ℓ au niveau de l'abscisse x ? On ne demande pas de justification.

La hauteur de ℓ au niveau de l'abscisse x est $h = \max\{h_1, h_2\}$.

Question 5 (2/8) — Compléter les lignes 15, 17 et 18 de Fusionner-lignes afin que l'algorithme Fusionner-lignes soit valide.

Voir les expressions soulignées dans Fusionner-lignes(ℓ_1, ℓ_2).

Question 6 (1/8) — Quelle est la complexité de l'algorithme Fusionner-lignes en fonction de $|\ell_1|$ et $|\ell_2|$? Justifier brièvement votre réponse.

Il y a exactement une itération pour chaque valeur de i comprise entre 1 et $|\ell_1|$, et une itération pour chaque valeur de j comprise entre 1 et $|\ell_2|$. Comme toutes les opérations se font en $O(1)$, l'algorithme est en $O(|\ell_1|+|\ell_2|)$.

Question 7 (2/8) — Soit $T(n)$ le nombre d'opérations effectuées par l'algorithme Calculer-ligne-2 pour un ensemble E de n immeubles. Donner la formule de récurrence vérifiée par T. En déduire la complexité de l'algorithme.

Comme $|\ell_1|+|\ell_2| \leq n$, l'opération de fusion est en $O(n)$. L'équation de récurrence est donc $T(n) = 2T(n/2) + O(n)$. D'après le théorème maître, l'algorithme est en $O(n \log n)$.

Fusionner-lignes(ℓ_1, ℓ_2)

Entrées : deux lignes de toits ℓ_1 et ℓ_2
Sortie : la ligne de toits correspondant à la fusion de ℓ_1 et ℓ_2

```
1  h_1 ← 0 ; h_2 ← 0        // h_i est la hauteur courante de ℓ_i (i ∈ {1,2})
2  ℓ ← [ ] ; h ← 0          // ℓ est la ligne fusionnée en cours de construction, h sa hauteur courante
3  i ← 1 ; j ← 1 ;
4  tant que i ≤ |ℓ_1| et j ≤ |ℓ_2| faire
5  |  x_1 ← ℓ_1[i].x
6  |  x_2 ← ℓ_2[j].x
7  |  si x_1 ≤ x_2 alors
8  |  |  h_1 ← ℓ_1[i].y
9  |  |  i ← i + 1
10 |  fin
11 |  si x_2 ≤ x_1 alors
12 |  |  h_2 ← ℓ_2[j].y
13 |  |  j ← j + 1
14 |  fin
15 |  si max{h_1, h_2} ≠ h alors
16 |  |  h ← max{h_1, h_2}
17 |  |  Ajouter (min{x_1, x_2}, h) en dernière position de ℓ        // instruction en O(1)
18 |  fin
19 fin
20 tant que i ≤ |ℓ_1| faire
21 |  Ajouter ℓ_1[i] en dernière position de ℓ        // instruction en O(1)
22 |  i ← i + 1
23 fin
24 tant que j ≤ |ℓ_2| faire
25 |  Ajouter ℓ_2[j] en dernière position de ℓ        // instruction en O(1)
26 |  j ← j + 1
27 fin
28 retourner ℓ
```