INF 431

F. Morain

Graphes II: parcours

14 mars 2007

Où en est-on ?

Amphi 1: introduction.

Amphi 2: génie logiciel avec Java.

Amphi 3: analyse lexicale.

Amphi 4: analyse syntaxique.

Amphi 5: graphes I.

Amphi 6: graphes II (parcours).

Amphi 7: graphes III (optimisation combinatoire).

Amphi 8: graphes IV (topologie).

Plan

I. Motivations

Comment détecter la connexité d'un graphe non orienté, énumérer ses composantes connexes ?

Classer les sommets d'un graphe en fonction de leur distance (nombre minimal de voisins intermédiaires) à un sommet donné ?

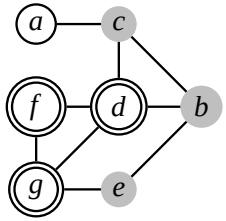
Généralisations aux graphes orientés : recherche de circuits, tri topologique.

Tous les graphes considérés sont simples.

On commence avec des graphes non orientés.

II. Définitions et premiers algorithmes

L'idée: à partir d'un sommet s, on explore une partie des sommets du graphe, les voisins de s. De proche en proche, nous sommes conduits à choisir les sommets suivants dans la bordure du graphe.



Déf. La bordure $\mathscr{B}(T)$ de $T \subset \mathscr{S}$ est l'ensemble des sommets de $\mathscr{S} - T$ adjacents à (un des sommets de) T.

Parcours

Déf. Un parcours de $\mathscr{G} = (\mathscr{S}, \mathscr{A})$ connexe à partir du sommet s est une liste de sommets L telle que:

- le premier sommet de L est s;
- chaque sommet de $\mathscr{S}$ apparait une fois et une seule dans L;
- tout sommet de la liste (sauf le premier) est adjacent dans $\mathscr{G}$ à au moins un sommet placé avant lui dans la liste.

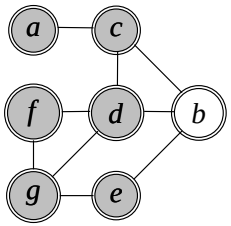
Rem. Il n'y a pas de parcours canonique.

Déf. Le support $\sigma(L)$ de L est l'ensemble des sommets contenus dans L. Par abus de notation, on notera souvent $\mathscr{B}(L) = \mathscr{B}(\sigma(L))$.

L'algorithme générique de parcours

```
parcoursGenerique(G, s)
1. L <- (s);
   B <- V(s); // les voisins de s
2. tantque B ≠ ∅ faire
    2.1 choisir u dans B; // exploration
    2.2 L <- L # u;
    2.3 B <- B - {u};
    2.4 B <- B ∪ (V(u) \ L); // !Z!
```

Ex. $L = (b,d,e,g,f,c,a)$:



L	B
(b)	$\{c,d,e\}$
(b,d)	$\{c,e,f,g\}$
(b,d,e)	$\{c,f,g\}$
(b,d,e,g)	$\{c,f\}$
(b,d,e,g,f)	$\{c\}$
(b,d,e,g,f,c)	$\{a\}$
(b,d,e,g,f,c,a)	$\emptyset$

Propriétés fondamentales

Prop. Avant 2.1, $L \cap B = \emptyset$.

Dém. Sinon, on aurait rajouté un voisin v de $u \in L$ qui n'était pas déjà dans B. □

Thm. Si les opérations ensemblistes coûtent $O(1)$, alors la complexité du parcours est $O(|\mathscr{S}| + |\mathscr{A}|) = O(n+m)$.

Dém. Chaque sommet de $\mathscr{G}$ est choisi une fois et une seule dans un parcours.
Le coût total est

$$\leqslant \sum_{u \in \mathscr{S}} \left(O(1) + \sum_{v \text{ voisin de } u} O(1) \right) = O(n) + O\left(\sum_{u \in \mathscr{S}} n_u \right) = O(n+m). \square$$

Application à la connexité

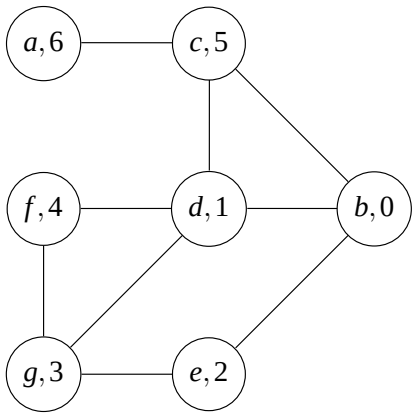
```
// affiche toutes les composantes connexes
composantesConnexes(G = (S, A))
T <- copie(S);
tantque T ≠ ∅
  choisir s dans T;
  T <- T - {s};
  L <- uneComposante(G, s, T);
  afficher "Composante : ", L;

// retourne la composante connexe contenant s
uneComposante(G, s, T)
1. L <- (s); B <- V(s);
2. tantque B ≠ ∅ faire
    2.1 choisir u dans B;
    2.2 L <- L # u;
    2.3 B <- B - {u};
    2.4 B <- B ∪ (V(u) \ L); // !Z!
    2.5 T <- T - {u};
3. retourner L;
```


Numérotation

Tout parcours induit une (re)numérotation des sommets de $\mathscr{G}$, par ordre d'apparition dans le parcours.

```
parcoursNumérotation(G, s)
1. num <- 0;
2. 2.1 L <- (s); B <- V(s);
   2.2 numéro[s] <- num++;
3. tantque B ≠ ∅ faire
   3.1 choisir u dans B;
   3.2 L <- L # u;
   3.3 B <- B - {u};
   3.4 B <- B ∪ (V(u) \ L); // !Z!
   3.5 numéro[u] <- num++;
```


$L = (b,d,e,g,f,c,a)$:

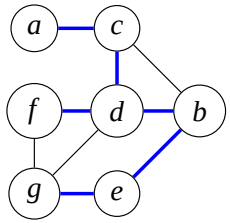


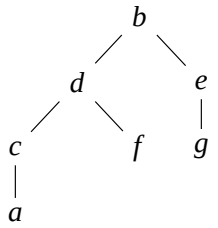
Arbre couvrant

Prop. Soit $L = (x_0, \ldots, x_{n-1})$ un parcours de $\mathscr{G}$. Pour tout k, $1 \leqslant k < n$, soit x'_k un sommet de $(x_0, \ldots, x_{k-1})$ adjacent à x_k. Le sous-graphe induit par les arêtes (x_k, x'_k), $1 \leqslant k < n$, est un arbre, appelé arbre couvrant de G (relatif à L).

Déf. Les arêtes (x_k, x'_k) sont appelées arêtes de liaison.

Ex. $L = (b,d,e,g,f,c,a)$:





Démonstration. On note G_p le graphe induit par $k \in \{0,\dots,p\}$ et on raisonne par récurrence sur p.

$p = 1$: seule arête (x_0, x_1), donc $x'_1 = x_0$.

$p > 1$: G_{p-1} connexe $\Rightarrow$ G_p connexe car on a rajouté une arête avec un sommet dans G_{p-1}; G_p a $p-1$ arêtes et p sommets, n'a pas de cycle, donc est un arbre. □

Implantation réaliste !Z!

Problème: quelle structure choisir pour $\mathscr{B}$ pour choisir et réaliser l'insertion/suppression et test d'appartenance en temps minimal ?

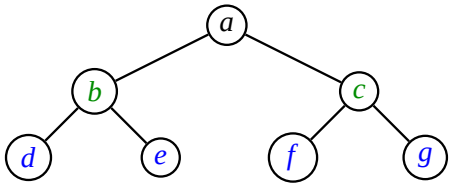
Rappels:

- Un ensemble représenté par table de hachage peut réaliser l'insertion/suppression et l'appartenance en temps $O(1)$. Mais comment choisir ?
- Une liste de sommets à visiter permet facilement l'insertion/suppression et le choix (e.g. le premier élément). Mais comment tester l'appartenance?
- Une solution: dupliquer en utilisant à la fois liste et table de hachage (mais on perd en temps et mémoire même si en $O(1)$).
- Autre solution: utiliser une liste et gérer un état pour un sommet qui indique s'il est déjà dans la liste ou pas.

III. Parcours en largeur d'abord

Idée: (*Breadth-first search*) les voisins de nos voisins sont nos voisins. On procède ainsi par "cercles" concentriques.

Exemple avec un arbre:



Règle de choix: le prochain sommet visité est le plus ancien ajouté à la bordure.

$\Rightarrow$ on utilise une file d'attente pour gérer la bordure.

Le pseudocode

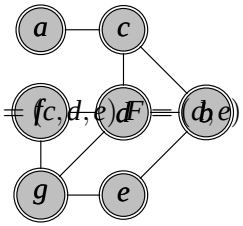
```
bfs(G, s)
0. pourtout sommet t faire
     t.etat <- inexploré;
1. F <- (s); s.etat <- encours;
2. tantque F ≠ ∅
     t <- tête(F);
     pour u voisin de t faire
       si u.etat == inexploré alors
         u.etat <- encours;
         F <- F # u;
     t.etat <- exploré;
```

Prop. La complexité de bfs est $O(|\mathscr{S}| + |\mathscr{A}|)$.

Dém. Le théorème générique s'applique, puisque les opérations de file et de gestion des états coûtent $O(1)$. □

Exemple

$F = \emptyset$ $F = (b)$ $F = (c)$ $F = (c,d)$ $F = (c,d,e)$ $F = (d,e)$ $F = (d,e)$ $F = (d,e,a)$ $F = (d,e,a$



La classe **Sommet** (suite)

On utilise les marques pour gérer les états:

```
public class Sommet{
  String nom;
  private int marque;

  public Sommet(String nn, int mm){
    nom = nn;
    marque = mm;
  }

  public int valeurMarque(){ return marque; }
  public void modifierMarque(int m){ marque = m; }
```


Le code Java

```
final static int inexplore=0,explore=1,encours=2;
void bfs(Sommet s){
  LinkedList<Sommet> F = new LinkedList<Sommet>();

  for(Sommet t : sommets())
    t.modifierMarque(inexplore);
  s.modifierMarque(encours);
  F.addLast(s);
  while(! F.isEmpty()){
    Sommet t = F.removeFirst();
    for(Arc a : voisins(t)){
      Sommet u = a.destination();
      if(u.valeurMarque() == inexplore){
        u.modifierMarque(encours);
        F.addLast(u);
      }
    }
    t.modifierMarque(explore);
  }
}
```


Recherche de composantes connexes

```
composantesConnexes(G)
0. pourtout sommet t faire t.etat <- inexploré;
   nc <- 0;
1. pour s dans S
     si s.etat == inexploré alors
       nc <- nc+1; // numéro de composante
       afficher("Composante " + nc);
       bfs(G, s);
// composante connexe contenant s
bfs(G, s)
1. F <- (s);
2. tantque F ≠ ∅ faire
     t <- tête(F);
     t.etat <- exploré;
     afficher(t);
     pour u voisin de t faire
       si u.etat == inexploré alors
         u.etat <- encours;
         F <- F # u;
```

Ex. écrire le code en Java.

Numérotation

On utilise une table de hachage pour numéroter les sommets:

```
void bfsNum(Sommet s){
  LinkedList<Sommet> F = new LinkedList<Sommet>();
  Hashtable<Sommet,Integer> numero =
    new Hashtable<Sommet,Integer>();
  int num = 0;
  for(Sommet t : sommets()){
    t.modifierMarque(inexplore);
    numero.put(t, 0);
  }
  s.modifierMarque(encours);
  F.addLast(s);
  numero.put(s, num++);
```



```
  while(! F.isEmpty()){
    Sommet t = F.removeFirst();
    for(Arc a : voisins(t)){
      Sommet u = a.destination();
      if(u.valeurMarque() == inexplore){
        u.modifierMarque(encours);
        F.addLast(u);
        numero.put(u, num++);
        System.out.print("num["+u+"]=");
        System.out.println(numero.get(u));
      }
    }
    t.modifierMarque(explore);
  }
}
```


IV. Parcours en profondeur d'abord

BFS: le prochain sommet exploré est le plus ancien sommet de type **`encours`**.

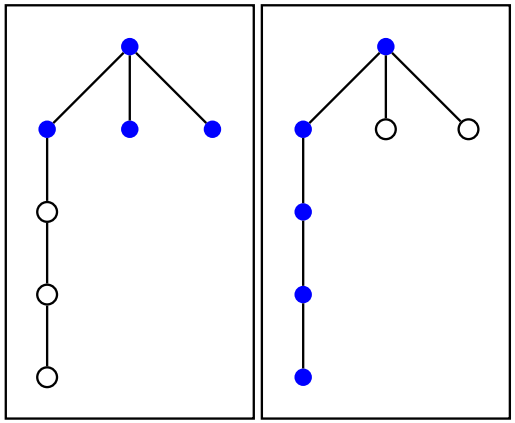
DFS: (*Depth First Search*) le prochain sommet visité est le plus récent sommet de type **`encours`**.

$\Rightarrow$ revient à remplacer la file par une pile $\Rightarrow$ même complexité.

```
dfs(G, s)
0. pourtout sommet t faire t.etat <- inexploré;
1. F <- (s); s.etat <- encours;
2. tantque F ≠ ∅
     t <- tête(F);
     pour u voisin de t faire
       si u.etat == inexploré alors
         u.etat <- encours;
         F <- u # F; // la ligne qui change
     t.etat <- exploré;
```


La différence géographique

Largeur $=$ au plus près; profondeur $=$ au plus loin.



Exercice. À quoi correspond une dfs sur un arbre?

Le code Java

```
void dfs(Sommet s){
  LinkedList<Sommet> F = new LinkedList<Sommet>();

  for(Sommet t : sommets())
    t.modifierMarque(inexplore);
  s.modifierMarque(encours);
  F.addFirst(s); // changement
  while(! F.isEmpty()){
    Sommet t = F.removeFirst();
    for(Arc a : voisins(t)){
      Sommet u = a.destination();
      if(u.valeurMarque() == inexplore){
        u.modifierMarque(encours);
        F.addFirst(u); // changement
      }
    }
    t.modifierMarque(explore);
  }
}
```

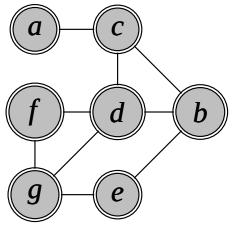

DFS récursive

Très souvent: itératif + pile = récursif.

```
void dfsRec(Sommet s){
  if(s.valeurMarque() == inexplore){
    s.modifierMarque(encours);
    for(Arc a : voisins(s))
      dfsRec(a.destination());
    s.modifierMarque(explore);
  }
}
```

Rem. bon exercice sur récursif + effets de bord.

Exemple



Construction de l'arbre couvrant

Déf. Une arborescence est un arbre dont on a distingué un sommet, la racine.

Déf. Nous appelerons arborescence de Trémaux un arbre couvrant obtenu lors d'une dfs.

On se donne un type d'arbre n-aire:

```
class Arbre{
  Sommet racine;
  LinkedList<Arbre> fils;

  Arbre(Sommet r){
    racine = r;
    fils = new LinkedList<Arbre>();
  }
  void ajouterFils(Arbre A){
    fils.addLast(A);
  }
}
```

```
// on retourne un arbre couvrant de racine s
Arbre dfsRec(Sommet s){
  s.modifierMarque(encours);
  Arbre A = new Arbre(s);
  for(Arc a : voisins(s)){
    Sommet t = a.d;
    if(t.valeurMarque() == inexplore)
      A.ajouterFils(dfsRec(t));
  }
  s.modifierMarque(explore);
  return A;
}
void dfsCouvrant(){
  Sommet ALPHA = new Sommet("ALPHA", 0);
  Arbre F = new Arbre(ALPHA);

  for(Sommet s : sommets())
    s.modifierMarque(inexplore);
  for(Sommet s : sommets())
    if(s.valeurMarque() == inexplore)
      F.ajouterFils(dfsRec(s));
}
```


V. Le cas orienté

Déf. La bordure $\mathscr{B}(T)$ d'une partie $T \subset \mathscr{S}$ est le sous-ensemble des sommets de $\mathscr{S} - T$ qui sont les extrémités d'un arc dont l'origine est dans T.

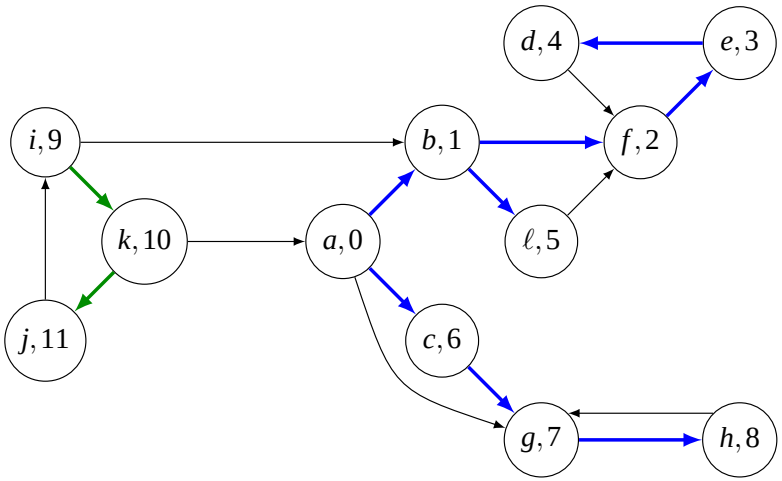
Déf. Un parcours de $\mathscr{G}$ est une liste L de sommets de G telle que:

- chaque sommet de $\mathscr{S}$ apparaît une fois et une seule dans L;
- chaque sommet de L (sauf le premier) appartient à la bordure du sous-ensemble des sommets placés avant lui dans la liste, si toutefois celle-ci est non vide.

On définit de même les arborescences de Trémaux.

Exemple de dfs dans le cas orienté

$L = (a, b, f, e, d, \ell, c, g, h, i, k, j)$



Le rang

$\mathrm{rang}(x) =$ numéro d'ordre du sommet x dans le parcours L.

```
dfsRang(G)
1. rang <- tableau de taille n;
2. pourtout sommet s faire s.etat <- inexploré;
3. rg <- 0;
4. tantqu'il reste un sommet s non exploré faire
5.   rg <- dfsRec(rang, rg, s);

dfsRec(rang, rg, s)
10. s.etat <- encours;
    rang[s] <- rg++;
    pour t voisin de s faire
      si t.etat == inexploré alors
        rg <- dfsRec(rang, rg, t);
    s.etat <- exploré;
12. retourner rg.
```

Arborescence préfixe

Prop. Les rangs des nœuds d'une arborescence de Trémaux forment un intervalle des entiers $[n,m]$. L'arborescence est préfixe, c'est-à-dire que pour tout sommet s, les rangs des descendants de s est un sous-intervalle de $[n,m]$ dont le plus petit élément est $\text{rang}(s)$.

Les différents types d'arc

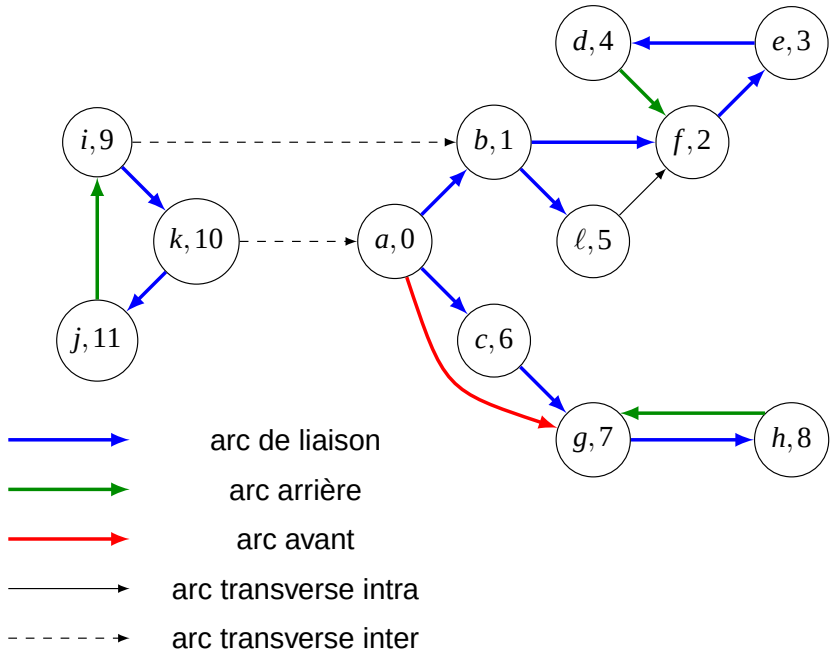
Soit $\mathscr{F}$ la forêt couvrante induite de L.

Les arcs de $\mathscr{F}$ sont les arcs de liaison. On peut distinguer trois autres types d'arcs dans $\mathscr{G}$:

Déf. Un arc (s,t) est:

- avant si s est un ascendant de t dans $\mathscr{F}$;
- arrière si t est un ascendant de s dans $\mathscr{F}$;
- transverse inter-arbre si s et t appartiennent à deux arborescences différentes, ou transverse intra-arbre si s et t ont un ancêtre commun z dans $\mathscr{F}$ distinct de s et t. Si on n'a pas besoin de distinguer ces deux cas, on parlera d'arc transverse.

$L = (a,b,f,e,d,\ell,c,g,h,i,k,j)$



Classification

Prop. **Relativement à L (ou $\mathscr{F}$)**:

(i) Si t est un descendant (resp. ascendant) strict de s dans l'arborescence, alors $\text{rang}(s) < \text{rang}(t)$ (resp. $\text{rang}(s) > \text{rang}(t)$).

(ii) Si (s,t) est un arc arrière, alors $\text{rang}(t) < \text{rang}(s)$.

(iii) Si (s,t) est un arc avant, alors $\text{rang}(s) < \text{rang}(t)$.

(iv) Le sommet t appartient à l'arborescence de racine s si et seulement si $\text{rang}(s) \leqslant \text{rang}(t) \leqslant \text{rang}(s) + |\mathscr{D}(s)|$ où $\mathscr{D}(s)$ est l'ensemble des descendants de s.

(v) Si (s,t) est un arc transverse, alors $\text{rang}(s) > \text{rang}(t)$. Autrement dit, s est visité après t.

Démonstration

Soient $A_1, \ldots, A_n$ les arborescences correspondant à L. Par définition des arboresences, il n'existe pas d'arc (u,v) avec $u \in A_i$ et $v \in A_j$ pour $i < j$.

(i-iv) On ne fait là que traduire les propriétés élémentaires de la fonction rang.

(v) **Si** (s,t) **est inter-arbre:** $s \in A_i$ et $t \in A_j$ avec $i \neq j \Rightarrow i > j \Rightarrow \text{rang}(s) > \text{rang}(t)$.

Si (s,t) **est intra-arbre:** s et t ont un ancêtre commun $z \notin \{s,t\}$. Si $\text{rang}(s) < \text{rang}(t)$, quand s est visité, t ne l'est pas encore et donc s sera un ascendant de t, ce qui est absurde, l'arc (s,t) serait soit de liaison, soit avant. □

Identification des arcs

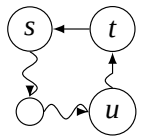
```
dfsRec(rang, rg, racine, s)
// s est non exploré,
// racine est la racine de l'arborescence courante
10. s.etat <- encours;
    rang[s] <- rg++;
    pour t voisin de s faire
      si t.etat == inexploré alors
        rg <- dfsRec(rang, rg, racine, t);
      sinon si t.etat == encours alors
        écrire "(s, t) arrière";
      sinon // t est déjà exploré
        si rang[t] < rang[racine] alors
          écrire "(s, t) transverse inter-arbre";
        sinon si rang[s] < rang[t] alors
          écrire "(s, t) avant";
        sinon // rang[s] > rang[t]
          écrire "(s, t) transverse intra-arbre";
    s.etat <- exploré;
13. retourner rg.
```


Graphes sans circuits

Prop. Le graphe $\mathscr{G}$ est sans circuit $\Longleftrightarrow$il n'existe pas d'arc arrière.

Démonstration. S'il existe un arc arrière, il referme un circuit, donc la condition est nécessaire.

$\mathscr{G}$ **contient un circuit** C**:** après avoir effectué la dfs correspondant à L, soit s le sommet de C de rang minimum.



Soit t le prédécesseur de s sur le circuit; t doit être un descendant de s dans une arborescence d'exploration $\Rightarrow (t,s)$ n'est pas transverse. $\text{rang}(t) > \text{rang}(s) \Rightarrow (s,t)$ n'est pas de liaison $\Rightarrow (t,s)$ est un arc arrière. □
Ex. Écrire un algorithme qui détecte la présence de circuit.

Résumé du cours

- Parcours: outil fondamental dans l'algorithmique des graphes; sortie de labyrinthe, etc.
- Cas orientés ou non orientés.
- Classification des arcs (**à suivre...**).

Prochains rendez-vous: PC cet après-midi; amphi 07 mercredi prochain 14/03.