

DS1-Informatique-MPI (durée 4h)

1 Composantes connexes et biconnexes

1.1 Site Internet et bases de données

1. `SELECT count(*) FROM ressources WHERE type = "cours"`
2. Cette requête renvoie le nom de la ressource la plus récemment chargée avec l'utilisateur qui l'a alors chargée.
3. `SELECT c1.id,c2.id FROM compte as c1 JOIN
chargement on id_u= c1.id JOIN
ressources on id_r = ressources.id JOIN
compte as c2 on owner = c2.id`
4. `SELECT c1.id,c2.id FROM compte as c1 JOIN
chargement on id_u= c1.id JOIN
ressources on id_r = ressources.id JOIN
compte as c2 on owner = c2.id
INTERSECT
SELECT c2.id,c1.id FROM compte as c1 JOIN
chargement on id_u= c1.id JOIN
ressources on id_r = ressources.id JOIN
compte as c2 on owner = c2.id`

1.2 Composantes connexes

5. Donner la représentation du graphe de la figure 3 sous forme de liste des arêtes ainsi que sous forme de listes d'adjacence.

On obtient : [7,9; 6,5;3,6;3,2;3,0;2,0;0,1;1,4;1,8] pour la liste des arêtes et [| [1;2;3] ; [0;4;8] ; [0;3] ; [0;2;] pour les listes d'adjacence.

Nous allons tout d'abord travailler en C.

Nous utiliserons alors le type suivant pour travailler sur des listes chaînées de couples (arêtes) :

```
struct liste {  
    int i;  
    int j;  
    struct liste* next;  
};
```

```

6. void nb_sommets(struct liste* l, int vu[],int* nbvertices){
    if (l==NULL){
        *nbvertices=0;
    }
    else {
        nb_sommets(l->next, vu,nbvertices);
        int res=0;
        if(!vu[l->i]) {res++; vu[l->i]=true;}
        if (!vu[l->j]) {res++; vu[l->j]=true;}
        *nbvertices+=res;
    }
}

```

Pour travailler avec une représentation des graphes par liste d'adjacence, on utilisera le type suivant :

```

#define MAXV 100

struct edgenode{
    int y;
    struct edgenode* next;
};
typedef struct edgenode edgenode;

struct graph {
    edgenode* edges[MAXV];
    int nb_vertices;
    int nedges;
    bool discovered[MAXV];
};
typedef struct graph graph;

```

On disposera des fonctions suivantes :

```

void initialize_graph(graph * g){
    for (int i=0;i<MAXV;i++){
        g->edges[i]=NULL;
    }
    g->nedges=0;
}

void insert_edge(graph* g, int x ,int y){

    edgenode* newnode = (edgenode*)malloc(sizeof(edgenode));
    newnode->y = y;
    newnode->next = g->edges[x];
    g->edges[x]=newnode;

    edgenode* newnodebis = (edgenode*)malloc(sizeof(edgenode));
    newnodebis->y = x;
    newnodebis->next = g->edges[y];
    g->edges[y]=newnodebis;
}

```

```

    }

7. void parcours(struct liste* l, graph* g){
    if (l!=NULL){
        insert_edge(g,l->i,l->j);
        g->nedges++;
        parcours(l->next,g);
    }
}

graph* convert(struct liste* l){
    graph* res = malloc(sizeof(graph));
    initialize_graph(g);
    int vu[MAXV];
    for (int i=0;i<MAXV;i++){
        vu[i]=false;
    }
    nb_sommets(l,vu, &res->nb_vertices);
    g->nedges=0;
    parcours(l,res);
    return res;
}

```

On se donne le programme suivant :

```

void initialize_search(graph* g){
    for (int i=0;i<MAXV;i++){
        g->discovered[i]=false;
    }
}

void explorer(int s, graph* g){
    g->discovered[s]=true;
    edgenode* liste = g->edges[s];
    while(liste !=NULL){
        int v = liste->y;
        if (g->discovered[v]){
            liste=liste->next;
        }
        else{
            explorer(v,g);
            liste=liste->next;
        }
    }
}

int parcours(graph* g){
    initialize_search(g);
    int num=0;
    for (int i=0;i<g->nb_vertices;i++){
        if (!(g->discovered[i])){
            explorer(i,g);

```

```

        num++;
    }
}
return num;
}

```

8. C'est le nombre de composantes connexes. Elle renvoie ici 2. Il s'agit d'un parcours en profondeur.
9. Grace au tableau `discovered`, chaque sommet lancera la procédure `explorer` exactement une fois. Lors de cette procédure, on consulte toute la liste d'adjacence ce qui a un coût $\deg(x)$ pour un sommet x . Ainsi la complexité est en $O(|V|)$ pour les opérations d'initialisations plus $O(|E|)$ pour la consultation de tous les voisins de chaque sommet. En effet, on sait que $\sum_{x \in V} \deg(x) = 2|E|$. Finalement on a bien $O(|V| + |E|)$.
10. Un graphe est connexe ssi il existe un chemin entre tout couple de sommets.
11.

```
bool connexe(graph* g){
    return (parcours(g) == 1);
}
```

Nous allons maintenant travailler en OCAML sur le même sujet.

Considérons le code suivant :

```

type arbre = Root of int*arbre list
type graphe = int list array

let rec explorer g vu s =
    vu.(s) <- true;
    let fils = ref [] in
    List.iter (fun i->if (not vu.(i)) then fils := (explorer g vu i)::(fils)) (g.(s));
    Root(s, !fils);;

let parcours g =
    let n = Array.length g in
    let vu = Array.make n false in
    let res = ref [] in
    for i = 0 to n-1 do
        if (not vu.(i)) then
            res := (explorer g vu i)::(!res);
    done;
    !res;;

```

12. C'est la forêt du parcours en profondeur. Elle est renvoyée sous la forme d'une liste d'arbres. Sur l'exemple, on obtient :
13.

```
let rec concat_map f l = match l with
| [] -> []
| a::suite -> f(a)@(concat_map f suite);;

let rec arbre_to_list a = match a with

```

```

|Root(x,l)-> x::(concat_map arbre_to_list l);;

let rec liste_parcours l =
  match l with
  | [] -> []
  | a::suite-> (arbre_to_list a)::(liste_parcours suite);;

```

1.3 Graphes biconnexes

14. Si $|V| = 1$, le graphe est connexe. Si $|V| = 2$, la définition impose que les deux sommets soient adjacents : il est donc connexe. Si $|V| \geq 3$, pour tous sommets distincts x, y , il existe un cycle élémentaire contenant x et y . Une portion de ce cycle est une chaîne reliant x à y . Ainsi tous les sommets sont reliés deux à deux : G est connexe.
15. Le chemin $a - b - c$, de sommets $\{a, b, c\}$ et d'arêtes $\{\{a, b\}, \{b, c\}\}$, est connexe. Il n'est pas biconnexe : aucun cycle élémentaire ne contient a et c (il n'y a d'ailleurs aucun cycle).
16. Les points d'articulation sont

$$\boxed{\{4, 6, 8, 11\}}.$$

17. Soit a un point d'articulation. Comme $G - a$ n'est pas connexe, il existe deux sommets $x, y \neq a$ appartenant à deux composantes connexes distinctes de $G - a$. Si x et y étaient dans un cycle élémentaire alors ils resteraient reliés dans $G - a$: G n'est pas biconnexe.
18. On suppose G connexe, $|V| \geq 3$, et sans point d'articulation. Fixons deux sommets distincts x, y .

(a) Puisque G est connexe, il existe une chaîne reliant x à y . En supprimant d'éventuelles boucles dans cette chaîne, on peut la choisir simple : $x = x_0, x_1, \dots, x_k = y$.

(b) L'arête $\{x_0, x_1\}$ appartient à un cycle élémentaire. En effet, comme $G - x_0$ est connexe et possède au moins deux sommets, x_0 ne peut avoir pour unique voisin x_1 : sinon, x_0 serait une feuille et $G - x_1$ serait non connexe (le sommet x_0 serait isolé). Il existe donc un autre voisin $z \neq x_1$ de x_0 . Puisque $G - x_0$ est connexe, on dispose d'un chemin simple de x_1 à z ne passant pas par x_0 . Les arêtes x_0x_1 et zx_0 complètent ce chemin en un cycle élémentaire contenant x_0 et x_1 .

(c) Supposons qu'un cycle élémentaire C contienne x_0 et x_i . Si $x_{i+1} \in C$, ce même cycle convient. Sinon, comme $G - x_i$ est connexe, il existe un chemin de x_{i+1} vers x_0 évitant x_i . Considérons son premier sommet z appartenant à C ; en choisissant un chemin simple et en l'arrêtant à cette première rencontre, sa portion de x_{i+1} à z ne rencontre C qu'en z . On forme alors un cycle avec l'arête x_ix_{i+1} , ce chemin de x_{i+1} à z , puis l'un des deux arcs de C reliant z à x_i , choisi pour contenir x_0 . Ce cycle est élémentaire et contient x_0 et x_{i+1} . (Si $z = x_0$, les deux arcs conviennent.)

(d) Par récurrence sur i , il existe un cycle élémentaire contenant $x_0 = x$ et chaque x_i , notamment $x_k = y$. Les sommets distincts x, y étant arbitraires, G est biconnexe.

19. Pour déterminer si a est un point d'articulation, on supprime provisoirement a et toutes ses arêtes incidentes, puis on effectue un parcours en profondeur à partir d'un sommet quelconque $s \neq a$. Si au moins un autre sommet de $G - a$ n'est pas découvert, alors $G - a$ n'est pas connexe et a est un point d'articulation. Sinon, ce n'en est pas un. Un tel test prend $O(|V| + |E|)$ opérations avec des listes d'adjacence, sans nécessairement construire physiquement le graphe privé de a : il suffit d'ignorer a pendant le parcours.

20. Pour chaque sommet $a \in V$, exécuter le test de la question 19. Si l'un de ces tests détecte un point d'articulation, répondre « non biconnexe » ; sinon répondre « biconnexe ». La correction découle de l'équivalence établie aux questions 17 et 18. Chaque test coûte $O(|V| + |E|)$ et il y a $|V|$ sommets. La complexité totale est donc

$$O(|V|(|V| + |E|)).$$

21. En visitant les voisins dans l'ordre croissant à partir de 0, le parcours en profondeur découvre les sommets dans l'ordre

$$0, 1, 2, 6, 5, 10, 11, 7, 8, 3, 4, 9, 12.$$

Les dates sont attribuées à partir de 0. Le tableau **prefixe**, indexé par numéro de sommet de 0 à 12, vaut donc

$$[0; 1; 2; 9; 10; 4; 3; 7; 8; 11; 5; 6; 12].$$

22. Soit $\{i, j\}$ une arête avec $\text{prefixe}[i] < \text{prefixe}[j]$. Quand le DFS découvre i , le sommet j n'est pas encore découvert. Avant que l'appel récursif explorant i se termine, toutes les arêtes incidentes à i sont examinées, en particulier $\{i, j\}$. Si j n'a pas encore été découvert au moment de cet examen, l'appel **explorer**(j) en fait un descendant direct de i . Sinon, j a nécessairement été découvert entre-temps pendant un appel récursif issu de i : il est donc aussi un descendant de i . Ainsi $j \in D(i)$ (avec la convention habituelle où les descendants comprennent les sommets situés strictement sous i).
23. On utilise $\text{ord}[i] = \min\{\text{prefixe}[j] : j \in V(D(i))\}$. Dans cette formule, $D(i)$ doit inclure i lui-même (descendance au sens large), sinon la formule serait indéfinie pour une feuille ; c'est la convention implicite nécessaire dans le sujet. En prenant les voisins et les descendants dans l'arbre DFS de la question 21, on obtient :

$$\text{ord} = [0; 0; 0; 8; 8; 0; 0; 6; 6; 10; 3; 6; 8].$$

Sommet i	0	1	2	3	4	5	6	7	8	9	10	11	12
$\text{prefixe}[i]$	0	1	2	9	10	4	3	7	8	11	5	6	12
$\text{ord}[i]$	0	0	0	8	8	0	0	6	6	10	3	6	8

Par exemple, $\text{ord}[10] = 3$ grâce à l'arête $10 - 6$, $\text{ord}[7] = 6$ grâce à l'arête $8 - 11$, et $\text{ord}[4] = 8$ grâce à l'arête $4 - 8$. Pour les feuilles 9 et 12, les plus petites dates dans leur voisinage fermé sont respectivement 10 et 8.

Remarque sur l'énoncé. La caractérisation des points d'articulation donnée après la question 23 doit normalement s'écrire, pour un sommet non racine i : il existe un fils j tel que $\text{ord}[j] \geq \text{prefixe}[i]$ (et non uniquement l'égalité). Dans cet exemple, le sommet 4 est bien un point d'articulation car son fils 9 vérifie $\text{ord}[9] = 10 = \text{prefixe}[4]$, et 8 en est un car son fils 12 vérifie $\text{ord}[12] = 8 = \text{prefixe}[8]$.

24. (*on suppose que le graphe est connexe donc on se contente de la partie recursive*)

```

let detecte_articulations g =
  let n = Array.length g in
  let vu = Array.make n false in
  let dec = Array.make n (-1) in
  let pere = Array.make n (-1) in
  let anc = Array.make n (-1) in
  let time = ref 0 in
  let art = Array.make n false in
  let rec visiter g s =
    if (not vu.(s)) then
      begin

```

```

vu.(s)<-true;
anc.(s)<-(!time); dec.(s)<-(!time);
time:=(!time)+1;
let rec parcours l = match l with
| []->()
| a::suite when vu.(a)-> if (a<>pere.(s)) then anc.(s)<- min anc.(s) dec.(a); parcours s
| a::suite-> pere.(a)<- s; visiter g a; anc.(s)<- min anc.(s) anc.(a);
               if (pere.(s)<>(-1) && anc.(a)>=dec.(s)) then art.(s)<-true;
               parcours suite;

      in parcours g.(s);
    end
in visiter g 0;
let deg0 = ref 0 in
for i = 0 to n-1 do
  if (pere.(i)=0) then
    incr deg0;
done;
if (!deg0 >1) then art.(0)<-true;
art;;

```

25. Une composante biconnexe est un ensemble maximal de sommets dont le sous-graphe induit est biconnexe. Les arêtes qui n'appartiennent à aucun cycle forment chacune une composante biconnexe à deux sommets, conformément à la définition du sujet.

Sur le graphe G'_{ex} de la figure 4, on obtient exactement six composantes :

$ \begin{aligned} B_1 &= \{0, 1, 2, 5, 6, 10\}, \\ B_2 &= \{6, 11\}, \\ B_3 &= \{7, 8, 11\}, \\ B_4 &= \{3, 4, 8\}, \\ B_5 &= \{4, 9\}, \\ B_6 &= \{8, 12\}. \end{aligned} $
--

La première composante contient notamment le cycle $0 - 1 - 2 - 6 - 10 - 5 - 0$; les troisième et quatrième sont des triangles. Les arêtes $(6, 11)$, $(4, 9)$ et $(8, 12)$ sont des ponts et forment chacune une composante à deux sommets. Les points d'articulation sont 4, 6, 8 et 11. Ils peuvent appartenir à plusieurs composantes biconnexes, contrairement aux composantes connexes qui sont disjointes.