

DS1-Informatique-MPI (durée 4h)

1 Questions de cours

1. Ecrire une fonction Ocaml `arb_pp : int list array -> int array` qui prend en entrée un graphe `g` sous forme de listes d'adjacence et qui renvoie un tableau `parent` tel que pour tout sommet `x` du graphe, `parent.(x)` est son père dans une arborescence de parcours en profondeur de `g`. Si un sommet n'a pas de père dans cette arborescence, on mettra la valeur `-1`.
2. Prouver que le complémentaire d'un langage reconnaissable est un langage reconnaissable.

2 Automates : exercice 1

Dans cet exercice, on étudie différents langages sur l'alphabet $A = \{0,1\}$ à deux lettres. On note A^* l'ensemble des mots construits sur l'alphabet A . Le mot vide est noté ε . En OCaml, un mot sur A est implémenté par le type

```
type mot = bool list;;
```

à savoir par une liste de booléens où la lettre 0 est représentée par le booléen `false` et la lettre 1 par le booléen `true`.

1. On note L_1 le langage des mots de A qui représentent l'écriture binaire d'un entier naturel, où le bit de poids faible se situe en fin de mot. Pour assurer l'unicité de la représentation, l'écriture binaire d'un entier ne commence jamais par 0. C'est pourquoi l'entier nul est représenté par le mot vide.
 - (a) Donner l'écriture binaire de l'entier 41.
 - (b) Donner l'entier représenté par le mot 10101010.
 - (c) Dessiner un automate déterministe complet $\mathcal{A}_1$ reconnaissant le langage L_1 .
 - (d) Écrire en OCaml une fonction `langage_1` de type `mot -> bool` qui prend en argument un mot m et qui renvoie `true` si et seulement si m appartient au langage L_1 .
2. On note L_2 le langage dénoté par l'expression rationnelle

$$(0 + 1)^* \cdot 0.$$

- (a) Dessiner un automate déterministe $\mathcal{A}_2$ reconnaissant le langage L_2 .
 - (b) Écrire en OCaml une fonction `langage_2` de type `mot -> bool` qui prend en argument un mot m et qui renvoie `true` si et seulement si m appartient au langage L_2 .
3. On note L_3 le langage reconnu par l'automate déterministe $\mathcal{A}_3$ défini par :
 - l'ensemble d'états $Q = \{0,1,2\}$;
 - l'état initial $i = 0$;
 - l'ensemble d'états finals $F = \{0\}$;

- la fonction de transition

$$\delta : Q \times A \longrightarrow Q$$

définie par

$$\forall q \in Q, \forall a \in A, \quad \delta(q, a) = (2q + a) \bmod 3.$$

On rappelle que $(n \bmod 3)$ désigne le reste de la division euclidienne de l'entier n par 3.

- Dessiner l'automate $\mathcal{A}_3$.
- Écrire en OCaml une fonction `langage_3` de type `mot -> bool` qui prend en argument un mot m et qui renvoie `true` si et seulement si m appartient au langage L_3 .
- Pour tout $n \in \mathbb{N}^*$, démontrer la propriété $\mathcal{P}(n)$ suivante :

$$\forall \omega_1, \dots, \omega_n \in A, \quad \widehat{\delta}(0, \omega_1 \cdots \omega_n) = \left(\sum_{k=1}^n \omega_k 2^{n-k} \right) \bmod 3,$$

où la fonction

$$\widehat{\delta} : Q \times A^* \longrightarrow Q$$

est définie par

$$\forall q \in Q, \forall \omega \in A^*, \forall a \in A, \quad \widehat{\delta}(q, \varepsilon) = q$$

et

$$\widehat{\delta}(q, \omega \cdot a) = \delta(\widehat{\delta}(q, \omega), a).$$

- On note

$$L_4 = L_1 \cap L_2 \cap L_3.$$

- Décrire simplement l'ensemble des mots du langage L_4 .
- Existe-t-il un automate reconnaissant le langage L_4 ?

3 Automates : exercice 2

L'objectif de cette partie est de proposer quelques éléments autour d'un automate, dit augmenté, construit autour d'un automate fini non déterministe et d'un sous-ensemble d'états.

Définition 1 (Automate fini non déterministe)

Un *automate fini non déterministe* est un quintuplet $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ avec :

- Q un ensemble fini non vide d'états, de cardinal $|Q|$;
- Σ un alphabet ;
- $I \subset Q$ l'ensemble des états initiaux ;
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ une fonction de transition : si $q \in Q$ et $a \in \Sigma$, $\delta(q, a)$ désigne l'ensemble des états q' de Q tels qu'il existe une transition étiquetée par a de q vers q' ;
- $F \subset Q$ l'ensemble des états finaux.

Définition 2 (Automate augmenté)

Soient $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ un automate fini non déterministe et $O \subset Q$ un sous-ensemble d'états de Q . On appelle *automate augmenté* par O la paire $(\mathcal{A}, O)$, que l'on notera par la suite $\mathcal{A}_O$.

La notion de calcul est la même entre $\mathcal{A}$ et $\mathcal{A}_O$: un calcul dans $\mathcal{A}$ est une séquence d'états $q_1 \cdots q_n$ de Q telle qu'il existe toujours au moins une transition entre deux états successifs de cette séquence. Le mot construit par concaténation des étiquettes de chacune des transitions est alors reconnu par l'automate. L'ensemble O introduit la notion de calcul réussi au seuil s .

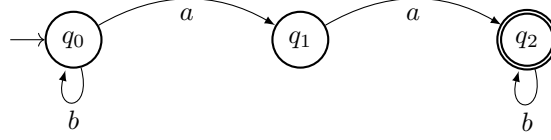


FIG. 1 – Automate exemple

Définition 3 (Calcul réussi au seuil s)

Soit $\mathcal{A}_O$ un automate augmenté. Un calcul passant par les états $q_i \in Q, i \in \llbracket 0, n \rrbracket$ est dit réussi au seuil $s \in \mathbb{N}$ si :

- i) $q_0 \in I$;
- ii) $q_n \in F$;
- iii) pour tout $i \geq 0$ tel que $i + s \leq n, \{q_i \cdots q_{i+s}\} \cap O \neq \emptyset$.

Définition 4 (Langage reconnu par un automate augmenté au seuil s)

Soit $\mathcal{A}_O$ un automate augmenté et $s \in \mathbb{N}$. Le langage reconnu par $\mathcal{A}_O$ au seuil s , noté $L_s(\mathcal{A}_O)$, est l'ensemble des calculs réussis par $\mathcal{A}_O$ au seuil s .

On considère, pour les questions 1 à 3, l'automate de la figure 1, où $Q = \{q_0, q_1, q_2\}, \Sigma = \{a, b\}, I = \{q_0\}$ et $F = \{q_2\}$.

1. Donner sans justification $L_2(\mathcal{A}_{|q_0|})$.
2. Donner, en justifiant votre réponse, $L_2(\mathcal{A}_{|q_1|})$.
3. Soit $s \in \mathbb{N}$. Donner, en justifiant votre réponse, le cardinal de $L_s(\mathcal{A}_0)$ en fonction de s , où $\emptyset$ est l'ensemble vide.
4. Donner, sans justification, un majorant des calculs réussis au seuil s ne passant par aucun état de O .
5. Soit maintenant un calcul réussi au seuil s qui passe au moins par un état de O . On note $\tilde{q} = q_0 \cdots q_l$ la suite des états correspondants et $i_0 \cdots i_k \in \llbracket 0, l \rrbracket$ la suite des indices dans $\tilde{q}$ correspondant aux états de O . Donner, sans justification, en fonction de s :
 - i) une majoration de i_0 ;
 - ii) un encadrement de i_k ;
 - iii) une majoration de $i_{j+1} - i_j$ pour $j \in \llbracket 0, k-1 \rrbracket$.
6. Soit $\mathcal{A}_O$ un automate augmenté, avec $|Q| = p$, où Q est l'ensemble des états de l'automate. Pour $s \in \mathbb{N}$, construire, en justifiant votre construction, un automate fini à $(s+1)p$ états reconnaissant $L_s(\mathcal{A}_O)$.

4 Composantes connexes et biconnexes

4.1 Site Internet et bases de données

On s'intéresse dans cette partie à un site Internet d'échange de supports de cours entre enseignants de NSI. Chaque personne désirant proposer ou récupérer du contenu doit commencer par se créer un compte sur ce site et peut ensuite accéder à du contenu ou en proposer.

Ce site repose sur une base de données contenant en particulier trois tables.

- La table comptes possède un enregistrement par utilisateur ou utilisatrice, et ses attributs sont :
- id, un identifiant numérique, unique pour chaque compte ;
- nom, le nom de la personne possédant le compte ;

- et d'autres informations, concernant le mot de passe, l'adresse mail, des préférences sur le site, etc., que nous ne détaillons pas ici.
- La table ressources possède un enregistrement par document téléversé sur le site. Ses attributs sont :
 - id, un identifiant numérique, unique pour chaque ressource ;
 - owner, l'identifiant de la personne ayant créé la ressource ;
 - titre, une chaîne de caractères décrivant la ressource ;
 - type, chaîne de caractères pouvant être cours, ds, tp ou td.
- La table chargement mémorise chaque fois qu'un utilisateur télécharge une ressource sur le site. Ses attributs sont :
 - date, date du téléchargement, par exemple '2021-02-28' pour le 28 février 2021 (on peut utiliser des opérations de comparaison classiques avec ce format) ;
 - id_u, identifiant de l'utilisateur qui télécharge la ressource ;
 - id_r, identifiant de la ressource téléchargée.

Voici un extrait de chacune de ces tables :

comptes			ressources			
id	nom	...	id	owner	titre	type
1	Ada Lovelace	...	4	1	Machine à décalage	cours
4	Alan Turing	...	13	4	Intelligence artificielle	td
...	...	...	...	...	...	...

chargement		
date	id_u	id_r
'1931-06-29'	4	4
'2020-05-30'	27	458
...	...	...

1. Écrire une requête SQL permettant de connaître le nombre total de ressources de type cours présentes sur le site.
2. Que fait la requête suivante :?


```
SELECT ressource.titre, comptes.nom
FROM chargement
JOIN ressources ON ressources.id = chargement.id_r
JOIN comptes ON comptes.id = chargement.id_u
ORDER BY chargement.date DESC
LIMIT 1
```
3. Écrire une requête SQL qui permet de déterminer la liste des couples (x,y) , signifiant que la personne possédant l'identifiant x a téléchargé des documents téléversés par la personne possédant l'identifiant y .
On définit le graphe non-orienté $G = (V,E)$ où V est l'ensemble des identifiants de comptes sur le site et $E \subset V \times V$ l'ensemble des paires d'identifiants telles que le premier compte a déjà téléchargé des documents téléversés par l'autre et réciproquement. Ainsi, si $(x,y) \in E$, alors on doit avoir $(y,x) \in E$.
4. Écrire une requête SQL qui renvoie la table des couples (x,y) de E .

4.2 Composantes connexes

L'objectif de cette partie est de déterminer les composantes connexes du graphe G défini à la partie précédente. Dans toute la suite, on notera $|X|$ le cardinal d'un ensemble X . On supposera que l'ensemble

V est constitué de sommets numérotés par des entiers consécutifs commençant à 0, c'est-à-dire que $V = \{0, 1, \dots, |V| - 1\}$. On dira que deux sommets x, y de V sont voisins lorsque $(x, y) \in E$.

La requête de la question 4 permet de récupérer le résultat sous forme d'une liste de couples. On souhaite avoir plutôt une représentation par listes d'adjacence, à savoir une liste de $|V|$ sous-listes, la i^e sous-liste contenant les voisins du sommet i .

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

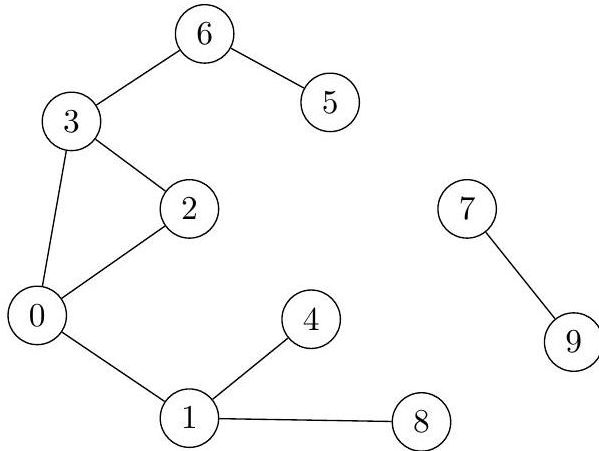


FIGURE 3 - Un graphe G_{ex}

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;
};

```

- Ecrire une fonction `void nb_sommets(struct liste* l, bool vu[], int* nbvertices)` qui prend en entrée une liste chaînée d'arêtes (de couples), un tableau de booléens de taille importante (vous supposerez que la taille est nécessairement plus grande que le nombre de sommets du graphe) dont toutes les cases sont initialisées à `false` et qui met à jour l'adresse `nbvertices` pour qu'elle contienne le nombre de sommets qui composent le graphe représenté par la liste `l` après l'exécution de la fonction. A l'issue de l'exécution, le tableau `vu` aura `true` dans sa case `i` ssi le graphe contient le sommet `i`.

Pour travailler avec une représentation des graphes par listes 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;
    g->nedges++;
}
```

7. Ecrire une fonction `graph* convert(struct liste* l)` qui prend en entrée une liste des arêtes et renvoie le graphe sous forme de listes d'adjacence.

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){
    if (!g->discovered[s]){
        g->discovered[s]=true;
        edgenode* liste = g->edges[s];
        while(liste !=NULL){
            int v = liste->y;
            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. Que représente la valeur renvoyée par la fonction `parcours`? Appliquer à la main cette fonction sur la liste d'adjacence du graphe G_{ex} de la figure 3, et donner la valeur de retour de cette fonction. Quel est le nom de ce parcours?
9. Montrer que la complexité de la fonction `parcours` est en $O(|V| + |E|)$. Dans toute la suite, on dira qu'un algorithme ayant cette complexité est linéaire.
10. Rappeler la définition de la connexité d'un graphe.
11. Écrire une fonction `bool connexe(graph* g)` qui renvoie `true` si le graphe `g` est connexe et `false` sinon.

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. Que représente la valeur renvoyée par la fonction `parcours`? Quel est son type? Appliquer à la main cette fonction sur la liste d'adjacence du graphe G_{ex} de la figure 3, et représenter la valeur de retour de cette fonction.
13. Écrire une fonction `liste_parcours : .....->(int list) list` qui prend en entrée le résultat de `parcours g` (et donc les points de suspension contiennent la réponse à la question précédente) et qui renvoie la liste des composantes connexes de `g`: chaque composante connexe sera une liste d'entiers et on renverra la liste des listes obtenues. Par exemple, sur le graphe de la figure 3, on obtiendra `[[0;1;2;3;4;5;6;8];[7;9]]` (peu importe l'ordre des sommets dans chaque liste) .

4.3 Graphes biconnexes

On suppose dans cette partie que G est un graphe connexe. Si $\forall i \in [0; k], x_i \in V$, on appelle chaîne une suite finie $(x_0, x_1, \dots, x_k)$ telle que pour tout i , on ait $(x_i, x_{i+1}) \in E$. Cette chaîne est un cycle lorsque $x_0 = x_k$, et c'est de plus un cycle élémentaire si tous les sommets $x_0, \dots, x_{k-1}$ sont distincts deux à deux. On dit que $G(V, E)$ est biconnexe lorsque :

- $|V| = 1$
- $|V| = 2, V = \{a, b\}$ et $(a, b) \in E$
- ou $|V| \geq 3$ et pour toute paire $(x, y) \in V^2$, il existe un cycle élémentaire contenant x et y .

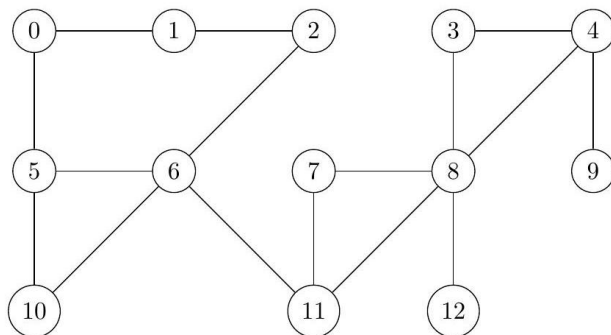
14. Montrer qu'un graphe biconnexe est également connexe.

15. Donner un exemple de graphe connexe mais pas biconnexe.

On dit qu'un sommet x de G est un point d'articulation lorsque le graphe G privé du sommet x (et des arêtes issues de x) n'est plus connexe. Notre objectif dans cette partie est de montrer la propriété suivante si $G = (V, E)$ possède au moins 3 sommets :

$G = (V, E)$ est biconnexe si et seulement si G ne possède pas de point d'articulation.

16. Sur le graphe G'_{ex} de la figure 4, donner les points d'articulations.



FigureE 4 - Le graphe connexe G'_{ex}

Dans toute la suite, on considère un graphe G ayant au moins 3 sommets.

17. Soit $G = (V, E)$ possédant un point d'articulation. Montrer que G n'est pas biconnexe.
18. Inversement, supposons $G = (V, E)$ un graphe sans point d'articulation, et tel que $|V| \geq 3$. Considérons deux sommets x et y .
- Justifier qu'il existe une chaîne $(x_0 = x, x_1, \dots, x_k = y)$ dans le graphe.
 - Montrer qu'il existe un cycle élémentaire contenant x_0 et x_1 .
 - Pour $i \geq 1$, on suppose qu'il existe un cycle élémentaire C contenant x_0 et x_i . Montrer qu'il existe alors un cycle élémentaire contenant x_0 et x_{i+1} . On pourra distinguer deux cas selon que C contient ou non x_{i+1} .
 - En déduire que G est biconnexe.
19. Expliquer comment on peut déterminer si un sommet particulier est un point d'articulation à l'aide d'un parcours en profondeur.
20. En déduire un algorithme qui prend en entrée un graphe connexe décrit par ses listes d'adjacences, et détermine si ce graphe est biconnexe en utilisant la propriété précédente. On ne demande pas de programmer cet algorithme. Quelle serait sa complexité en fonction des caractéristiques $|E|$ et $|V|$ du graphe?

4.4 Algorithme efficace pour déterminer les points d'articulation

Dans cette partie, on détaille comment déterminer tous les points d'articulation d'un graphe $G(V, E)$ avec une complexité linéaire.

On modifie le programme OCAML `parcours` pour lui faire remplir et renvoyer un tableau `prefixe` correspondant aux ordres d'appel de la fonction `explorer` (dates de découverte). De plus, on suppose désormais que le graphe passé en entrée est connexe, et on ne renverra qu'un seul arbre, issu de l'exploration à partir du sommet 0, et le tableau `prefixe`.

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

```

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

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

```

21. Donner les valeurs du tableau `prefixe` renvoyé par le programme ci-dessus si on l'applique sur le graphe G'_{ex} de la figure 4. On supposera que les voisins sont rangés par ordre croissant de leur numéro dans les listes d'adjacence.
22. Soit G un graphe connexe dans lequel on réalise le parcours avec la fonction ci-dessus, et soit (i,j) une arête de G telle que `prefixe[i] < prefixe[j]`. Montrer que j est un descendant de i dans l'arbre. Pour chaque sommet i , on note $\mathcal{V}(i)$ le voisinage de i , c'est-à-dire l'ensemble constitué de i et de ses voisins. Par extension, pour tout $A \subset V$, on notera $\mathcal{V}(A) = \cup_{i \in A} \mathcal{V}(i)$. En supposant réalisé un parcours dans l'arbre, on notera de plus $\mathcal{D}(i)$ l'ensemble des descendants de i dans l'arbre. Enfin, on définit `ord[i]` par :

$$\text{ord}[i] = \min_{j \in \mathcal{V}(\mathcal{D}(i))} \text{prefixe}[j]$$

23. Sur le graphe G'_{ex} de la figure 4, donner pour chaque sommet les valeurs de `ord[i]`, en se basant sur les valeurs obtenues à la question 21. On admettra qu'un sommet i du graphe qui n'est pas la racine est un point d'articulation si et seulement si un de ses fils j vérifie `ord[j]=prefixe[i]`. On supposera de plus qu'on dispose d'une fonction `calcule_ord : graphe -> int array` qui renvoie le tableau des `ord[i]` du graphe passé en entrée, avec une complexité linéaire.
24. Ecrire une fonction `points_articulation : graphe -> int list` qui renvoie la liste des points d'articulation d'un graphe passé en entrée. On fera attention à traiter la racine de l'arbre comme un cas particulier. On définit une composante biconnexe d'un graphe G comme un sous-ensemble de sommets maximal (au sens de l'inclusion) qui est biconnexe.
25. Sur le graphe G'_{ex} de la figure 4, donner la liste des composantes biconnexes.
26. Décrire un algorithme qui renvoie les composantes biconnexes d'un graphe avec une complexité linéaire. On ne demande pas de programmer cet algorithme.