

Examen de Info C

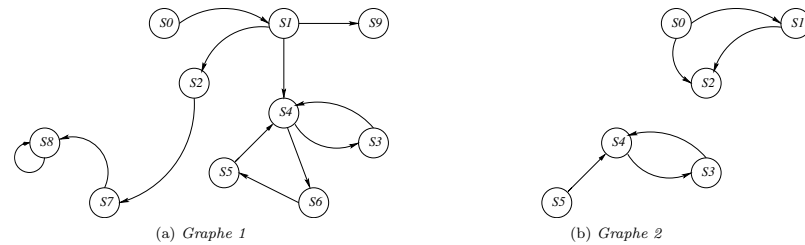
Durée : 1h30

Aucun document autorisé

Mobiles interdits

Notez que les affirmations (antécédents, conséquents, rôles, et invariants) dans vos codes C entreront pour partie dans la note finale.

- 1. Un *graphe orienté* est un ensemble de *sommets* reliés par des *arcs*, comme dans les exemples suivants :



Un graphe est dit faiblement connexe s'il existe un chemin *non orienté*, reliant toute paire de sommets (x, y) . Il est unilatéralement connexe s'il existe un chemin *orienté* reliant toute paire de sommets (x, y) . Il est fortement connexe si un tel chemin *orienté* existe de x vers y et de y vers x .

- 2. Les graphes donnés ci-dessus sont-ils faiblement, unilatéralement, fortement connexes ou non connexes ? Expliquez.

Question sur 1 pt.

Le graphe 1 est faiblement connexe, tous les sommets sont reliés. Toutefois, il n'est pas unilatéralement connexe puisque, par exemple, il n'y a pas de chemin entre les sommets $S6$ et $S7$, et donc pas fortement connexe.

Le graphe 2 n'est pas connexe puisque, par exemple, il n'y a pas de chemin entre les sommets $S2$ et $S5$.

On veut implémenter la notion de graphe d'ordre n (i.e. n est le nombre de sommets du graphe). Une façon de représenter un graphe est d'utiliser une *matrice d'adjacence* de taille $n \times n$. C'est une matrice de booléens. Deux sommets $s1$ et $s2$ sont dits adjacents s'il existe un arc qui les relie, allant de $s1$ à $s2$, et donc `matAdj[getIndex(s1)][getIndex(s2)]==vrai`.

- 3. Ci-dessous, complétez la matrice d'adjacence du *Graphe 2* de la page précédente :

	S0	S1	S2	S3	S4	S5
S0						
S1						
S2						
S3						
S4						
S5						

Question sur 1 pt.

	S0	S1	S2	S3	S4	S5
S0	faux	vrai	vrai	faux	faux	faux
S1	faux	faux	vrai	faux	faux	faux
S2	faux	faux	faux	faux	faux	faux
S3	faux	faux	faux	faux	vrai	faux
S4	faux	faux	faux	vrai	faux	faux
S5	faux	faux	faux	faux	vrai	faux

- 4. On possède le fichier `graphe.h` suivant :

```
#include "sommet.h"
typedef enum {false, true} bool;
```

```
typedef struct {
    int ordre;
    bool **matAdj;
} graphe;
```

```
// Rôle : crée un graphe d'ordre n vide
extern graphe créerGraphe(const int n);
```

```
// Rôle : renvoie l'ordre du graphe g
extern int ordre(const graphe g);

// Rôle : ajoute dans le graphe g l'arc s1 vers s2
extern void ajouterArc(graphe *g, const Sommet s1, const Sommet s2);

// Rôle : enlève dans le graphe g l'arc s1 vers s2
extern void enleverArc(graphe *g, const Sommet s1, const Sommet s2);

// Rôle : teste si dans le graphe g l'arc s1 vers s2 existe
extern bool arcExiste(const graphe g, const Sommet s1, const Sommet s2);
```

Dans le fichier `graphe.c`, programmez 4 les fonctions précédentes. **La matrice doit être allouée dynamiquement**. Vous pourrez utiliser directement les fonctions de conversion suivantes :

```
// Rôle : renvoie le sommet d'indice i dans la matrice d'adjacence
extern Sommet getSommet(const int i);
// Rôle : renvoie l'indice du sommet s dans la matrice d'adjacence
extern int getIndice(const Sommet s);
```

Question sur 4 pts.

```
/*
 * Antécédent : n>0
 * Rôle : crée un graphe d'ordre n vide (i.e. sans arcs)
 */
graphe créerGraphe(const int n) {
    assert(n>0);
    graphe g = {n, calloc(sizeof(bool*), n) };
    for (int i=0; i<n; i++)
        g.matAdj[i] = calloc(sizeof(bool), n);
    //
    return g;
}

/* Rôle : renvoie l'ordre du graphe g */
int ordre(const graphe g) {
    return g.ordre;
}

/* Rôle : renvoie l'ordre du graphe courant (i.e. son nb de sommets) */
int ordre(const graphe g) { return g.ordre; }

/* Rôle : ajoute dans le graphe g l'arc s1 vers s2 */
void ajouterArc(graphe *g, const Sommet s1, const Sommet s2) {
    g->matAdj[getIndice(s1)][getIndice(s2)] = true;
}

/* Rôle : enlève dans le graphe g l'arc s1 vers s2 */
void enleverArc(graphe *g, const Sommet s1, const Sommet s2) {
    g->matAdj[getIndice(s1)][getIndice(s2)] = false;
}

/* Rôle : teste si dans le graphe g l'arc s1 vers s2 existe */
bool arcExiste(const graphe g, const Sommet de, const Sommet vers) {
    return g.matAdj[getIndice(s1)][getIndice(s2)];
}
```

-
- 5. Écrivez la fonction `main` qui déclare la variable `g` de type `graphe` dont les sommets sont des entiers (`int`) et qui construit le *Graphe 1* de la première page de ce DS.

Question sur 1 pt.

```
int main(void) {
    graphe g = créerGraphe(10);

    ajouterArc(&g,0,1);
    ajouterArc(&g,1,2); ajouterArc(&g,1,4); ajouterArc(&g,1,9);
    ajouterArc(&g,2,7);
    ajouterArc(&g,3,4);
    ajouterArc(&g,4,3); ajouterArc(&g,4,6);
    ajouterArc(&g,5,4);
    ajouterArc(&g,6,5);
    ajouterArc(&g,7,8);
    ajouterArc(&g,8,8);
    return EXIT_SUCCESS;
}
```

-
- 6. Un arc u est *incident* à un sommet s vers l'extérieur si s est l'extrémité initiale de u . Le *demi-degré extérieur* est le nombre d'arcs incidents vers l'extérieur à un sommet s . En d'autres termes, c'est le nombre d'arcs qui partent du sommet s . On ajoute à `graphe.h` la fonction :

```
extern int demiDegréExt(const graphe g, const Sommet s);
```

Programmez cette fonction de `graphe.c`.

Question sur 2 pts.

```
/* Rôle : renvoie le nombre d'arcs qui partent du sommet s */
int demiDegréExt(const graphe g, const Sommet s) {
    const int indice_s = getIndice(s);
    int n=0;
    for (int i=0; i<g.ordre; i++)
        if (g.matAdj[indice_s][i]) n++;
    return n;
}
```

ou bien, mais moins efficace :

```
int demiDegréExt(const graphe g, const Sommet s) {
    int n=0;
    for (int i=0; i<ordre(g); i++)
        if (arcExiste(g, s, getSommet(i))) n++;
    return n;
}
```

-
- 7. De même, un arc u est *incident* à un sommet s vers l'intérieur si s est l'extrémité finale de u .

Le *demi-degré intérieur* est le nombre d'arcs incidents vers l'intérieur à un sommet s . En d'autres termes, c'est le nombre d'arcs qui arrivent sur le sommet s . On ajoute à `graphe.h` la fonction :

```
extern int demiDegréInt(const graphe g, const Sommet s);
```

Programmez cette fonction de `graphe.c`.

Question sur 2 pts.

```
/* Rôle : renvoie le nombre d'arcs qui arrivent sur s */
int demiDegréInt(const graphe g, const Sommet s) {
    const int indice_s = getIndice(s);
    int n=0;
    for (int i=0; i<g.ordre; i++)
        if (g.matAdj[i][indice_s]) n++;
    return n;
}
```

ou bien, mais moins efficace :

```
int demiDegréInt(const graphe g, const Sommet s) {
    int n=0;
    for (int i=0; i<ordre(g); i++)
        if (arcExiste(g, getSommet(i)), s) n++;
    return n;
}
```

On appelle *puits* d'une composante connexe, un sommet s tel que pour tout sommet $x \neq s$, il existe un arc (x, s) mais pas l'arc (s, x) . Par exemple, dans le graphe suivant, le sommet 4 est un puits :

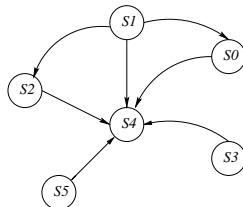


FIGURE 1 – Un puits

- 8. Montrez qu'une composante connexe ne peut avoir au maximum qu'un seul puits.

Question sur 1 pt.

Supposons qu'il existe 2 puits $p1$ et $p2$. S'il tel est le cas, puisque le graphe est connexe, il existe alors un arc entre $p1$ et $p2$ et entre $p2$ et $p1$, ce qui est en contradiction avec la définition d'un puits.

- 9. On ajoute à `graphe.h` la fonction :

```
extern Sommet chercherPuits(const graphe g);
```

Programmez cette fonction dans le fichier `graphe.c`. La fonction `chercherPuits` renverra la constante `SOMMET_NULL` s'il n'y a pas de puits dans le graphe.

Question sur 3 pts.

```
/* Rôle : renvoie le sommet, puits du graphe courant,
// s'il existe, sinon renvoie SOMMET_NULL
Sommet chercherPuits(const graphe g) {
    const int n = ordre(g);
    for (int i=0; i<n; i++)
        if (demiDegréExt(g, getSommet(i))==0 &&
            demiDegréInt(g, getSommet(i))==n-1)
            // s est le puits du graphe
            return getSommet(i);
    // le graphe ne possède pas de puits
    return SOMMET_NULL;
}
```

- 10. On ajoute à `graphe.h` la fonction :

```
extern void écrireGraphe(const graphe g, const char *fname);
```

Programmez cette fonction de `graphe.c` qui écrit le graphe g , passé en paramètre, dans le fichier de texte de nom `fname`. Vous pourrez utiliser directement la procédure suivante :

```
extern void écrireSommet(FILE *fd, const Sommet s);
```

Par exemple, pour le *Graphe 1*, on écrira dans le fichier :

```
{ S0 -> (S1) }
{ S1 -> (S2) (S4) (S9) }
{ S2 -> (S7) }
{ S3 -> (S4) }
{ S4 -> (S3) (S6) }
{ S5 -> (S4) }
{ S6 -> (S5) }
{ S7 -> (S8) }
{ S8 -> (S8) }
{ S9 -> }
```

Question sur 4 pts.

```
/* Rôle : écrit le graphe g sans le fichier de nom fname */
void écrireGraphe(const graphe g, const char *fname) {
    FILE *fd;
```

```

if ((fd=fopen(fname, "w")) == NULL) {
    perror(fname);
    exit(errno);
}
// le fichier est ouvert en écriture
for (int i=0; i<ordre(g); i++) {
    fprintf(fd, "{ ");
    écrireSommet(fd, getSommet(i));
    fprintf(fd, " ->");
    for (int j=0; j<g.ordre; j++)
        if (g.matAdj[i][j]) {
            fprintf(fd, " (");
            écrireSommet(fd, getSommet(j));
            fprintf(fd, ")");
        }
    fprintf(fd, " }\n");
}
fclose(fd);
}

```

```

bool cheminExiste(const graphe g, const Sommet s1, const Sommet s2) {
    bool *marques = calloc(sizeof(bool), ordre(g));
    return pProf(g, s1, s2, marques);
}

```

- 11. On veut savoir s'il existe un chemin orienté entre 2 sommets. Par exemple, dans le *Graphe 1* de la 1ère page, c'est le cas du sommet *S0* au sommet *S9*, ou du sommet *S6* au sommet *S3*, mais pas du sommet *S7* au sommet *S1*. On ajoute à `graphe.h` la fonction :

```
bool cheminExiste(const graphe g, const Sommet s1, const Sommet s2);
```

Programmez cette fonction de `graphe.c` qui teste s'il existe un chemin orienté du sommet `s1` vers le `s2` dans le graphe `g`.

Question sur 3 pts.

Une façon de procéder est de faire un parcours en profondeur du graphe. Dès que le sommet final est trouvé, on obtient un chemin. Notez qu'il faut marquer les nœuds visités pour éviter les cycles.

```

/*
 * Rôle : parcourt en profondeur le graphe g du sommet s1 au sommet s2
 * le parcours s'achève quand s1==s2
 */
static bool pProf(const graphe g, const Sommet s1, const Sommet s2, bool *marques)
{
    if (s1 == s2) return true;
    else {
        bool trouvé = false;
        marques[getIndice(s1)]=true;
        // parcourir les voisins de s1, et si s2 en fait partie => on a un chemin
        for (int i=0; i<ordre(g); i++)
            if (g.matAdj[getIndice(s1)][i] && !marques[i])
                // getSommet(i) est un voisin de s1 non visité
                trouvé = trouvé || pProf(g, getSommet(i), s2, marques);
        //
        return trouvé;
    }
}

/* Rôle : teste s'il existe un chemin orienté entre les sommets s1 et s2 */

```