POLYTECH – ELEC3 2022–2023

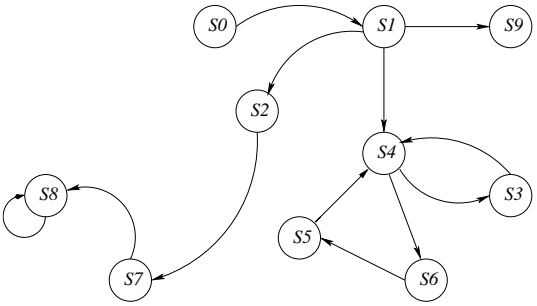
Contrôle de Info C

Durée : 1h30 Aucun document autorisé

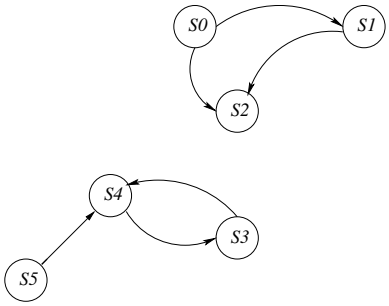
Nom : Prénom :

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 :



(a) *Graphe 1*



(b) *Graphe 2*

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.

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[getIndice(s1)][getIndice(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						

► 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);
```

▶ 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.

▶ 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. On ajoute à `graphe.h` la fonction :

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

Programmez cette fonction de `graphe.c`.

► 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`.

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 :

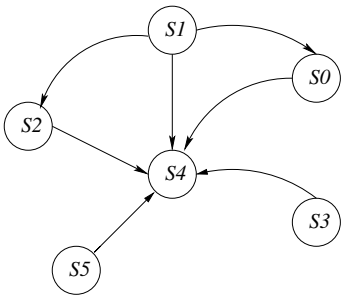


FIGURE 1 – Un puits

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

► 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.

► 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 ->  }
```

▶ 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`.