TD10 : Parcours de graphes

Exercice 1 – Parcours en profondeur, marquage et détection de circuits/cycles

Dans cet exercice, on va considérer quatre graphes :

— $G_1 = (S_1, A_1)$: graphe orienté avec $S_1 = (0, 1, 2, 3, 4, 5)$ et $A_1 = \{(0, 1), (0, 4), (1, 2), (1, 3), (1, 5)\}$.
— $G_2 = (S_1, A_2)$: graphe non-orienté avec $A_2 = \{\{0, 1\}, \{0, 4\}, \{1, 2\}, \{1, 3\}, \{1, 5\}\}$.
— $G_3 = (S_1, A_3)$: graphe orienté avec $A_3 = A_1 \cup \{(3, 4), (4, 5), (5, 3), (2, 3)\}$.
— $G_4 = (S_1, A_4)$: graphe non-orienté avec $A_4 = A_2 \cup \{\{3, 4\}, \{4, 5\}, \{5, 3\}, \{2, 3\}\}$.

Q 1.1 Dessiner ces quatre graphes, puis donner leurs caractéristiques.

Q 1.2 On considère la structure de données `GrapheSimple` suivante, dotée de trois fonctions de manipulation.

```c
typedef struct cellule {
  int v;                     /* indice d'un sommet de tabS */
  struct cellule * suiv ; /* pointeur sur le lien suivant */
} Cellule ;

typedef struct {
  int nbsom ;        /* Nombre de sommets */
  Cellule** tabS ;   /* Tableau de listes chainees de sommets */
} GrapheSimple ;

void cree_graphe(GrapheSimple *G, int n){
  int i;
  G->nbsom=n;
  G->tabS=(Cellule**) malloc(n*sizeof(Cellule*));
  for (i=0;i<n;i++)
    G->tabS[i]=NULL;
}

void ajoute_lien(GrapheSimple *G, int i, int j){// ajoute le lien orient'e (i,j)
  Cellule *nouv=(Cellule*) malloc(sizeof(Cellule));
  nouv->v=j;
  nouv->suiv=G->tabS[i];
  G->tabS[i]=nouv;
}

void aff_graphe(GrapheSimple *G){
  printf("\n\nGraphe:\n");
  int i;
  for (i=0;i<G->nbsom;i++){
    printf("%d : ",i);
    Cellule* cour=G->tabS[i];
    while (cour!=NULL){
      printf("%d ",cour->v);
      cour=cour->suiv;
    }
    printf("\n");
  }
}
```

Supposons que les graphes G_1, G_2, G_3 et G_4 ont été créés :

— à partir de graphes vides (retournés par la fonction `cree_graphe`)

— en y ajoutant les liens dans l'ordre de leur définition (avec la fonction `ajouter_lien`).

Donnez l'affichage obtenue par la fonction `aff_graphe` appliquées à ces graphes.

Partie 1 : *Parcours en profondeur et marquage.*

On considère la fonction d'affichage suivante, permettant d'afficher un sous-parcours de graphe à partir d'un sommet initial de départ.

```c
void aff_parcours_1(GrapheSimple* G, int r){
   printf("%d ",r);
   Cellule* cour = G->tabS[r];
   while (cour!=NULL){
      int v = cour->v;
      aff_parcours_1(G,v);
      cour=cour->suiv;
   }
}
```

Q 1.3 Donner l'affichage de cette fonction pour $r = 0$ dans le graphe G_1. Même questions pour G_2. Que constatez-vous ?

Q 1.4 Proposer une façon de corriger cet affichage en modifiant `aff_parcours_1` en `aff_parcours_2`, de manière à ne pas afficher les doubles liens symbolisant une arête. Donner le nouvel affichage.

Q 1.5 Donnez l'affichage obtenu par la fonction `aff_parcours_2` appliquée au graphe non-orienté G_4. Que constatez-vous ? Même question pour le graphe orienté G_3.

Q 1.6 En utilisant un tableau de marquage `visit` indiquant 1 si un sommet est visité et 0 sinon, proposer une nouvelle fonction `aff_parcours_3` permettant de corriger le problème. Le tableau est supposé être alloué et initialisé avec des valeurs 0 avant l'appel.

Partie 2 : *Détection de circuit et de cycle*

On se concentre ici sur les graphes orientés en prenant pour exemple le graphe G_3. On veut transformer la fonction `aff_parcours_3` de l'exercice précédent pour détecter s'il existe un circuit dans le graphe. Un circuit est détecté au cours d'un parcours lorsqu'on rencontre un arc (k, l) alors qu'on a déjà visité l et que l est un ascendant de k. Cette propriété est en fait suffisante pour la détection de circuit. Pour mettre en œuvre cette propriété, on va utiliser plusieurs statuts pour un sommet au sein du tableau de marquage `visit` :

— 0 : non encore visité.

— 1 : rencontré mais ayant encore des descendants non visités (on appelle ce statut "ouvert").

— 2 : visité et dont tous les descendants ont été visités (on appelle ce statut "fermé").

Q 1.7 Donner une fonction renvoyant une valeur booléenne indiquant s'il existe un circuit dans la partie d'un graphe accessible à partir d'un sommet r.

Q 1.8 Même question, mais pour la détection d'un circuit dans tout le graphe.

On désire à présent récupérer en retour un circuit s'il en existe un. Pour cela, nous allons conserver l'arborescence du sous-parcours du graphe correspondant au parcours en profondeur effectué. Une première idée est d'utiliser des listes chaînées d'arcs : mais cela demande de faire des mises à jour fréquentes. On préfère utiliser un simple tableau indicé sur les sommets : on utilise un tableau *pred* contenant pour chaque sommet le numéro de son père dans l'arborescence et -1 s'il n'en a pas.

Q 1.9 Adapter la fonction précédente pour qu'elle retourne à la fois *pred* et l'arc (k, l) d'un éventuel circuit.

Q 1.10 Donnez une fonction affichant un circuit encodé par un tableau *pred* et un arc (k, l). Si cet affichage est à l'envers, comment l'obtenir à l'endroit ?

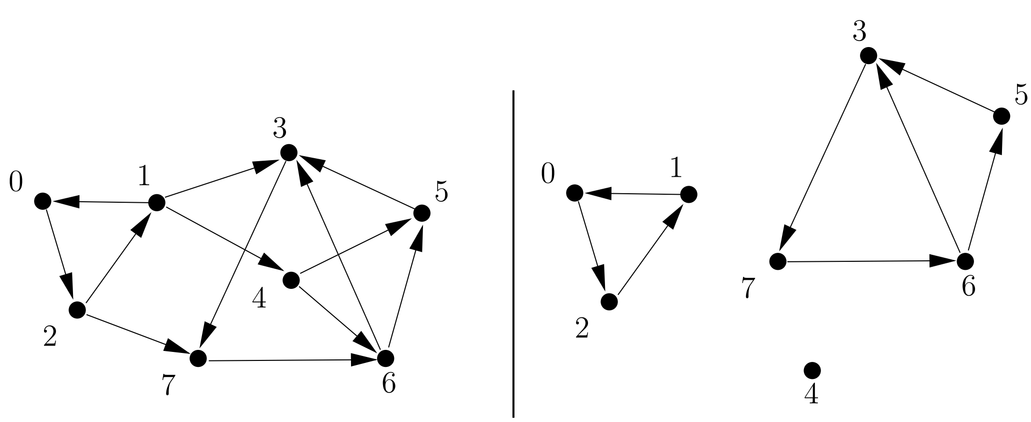
Q 1.11 On s'intéresse à présent aux graphes non-orientés. Appliquez vos fonctions au graphe G_2 et au graphe G_4. Que constatez-vous ? Comment corriger ?

Exercice 2 – Recherche de composantes fortement connexes

On désire étudier les liens communautaires sur internet. Pour cela, on s'intéresse au graphe orienté que crée un ensemble donné de blogs : chaque sommet représente un blog, et il existe un arc (i, j) si le blog représenté par le sommet i contient un lien hypertexte qui pointe vers le blog représenté par le sommet j.

Le fait que le blog i pointe vers j ne signifie pas que les deux blogs fassent partie de la même communauté. En effet, le site i peut citer par hasard un élément du blog j (par exemple un film critiqué par un journal ne cite que rarement la critique sur son site). En revanche, si j pointe à son tour vers i, on peut dire qu'ils font partie de la même communauté. On peut généraliser cette remarque de la façon suivante : on dit que deux sommets i et j sont *fortement reliés* s'il existe un chemin de i vers j ET un chemin de j vers i. On remarque alors que l'ensemble des sommets d'un graphe se partitionne en sous-ensembles non-vides de sommets $(C_1, \ldots, C_q)$ tels que, au sein de chaque ensemble C_i, $i = 1, ...q$, tous les couples de sommets sont fortement reliés. On appelle ces ensembles C_i les *composantes fortement connexes* (CFC) du graphe. En fait, ces composantes fortement connexes correspondent à des communautés de blogueurs.

La figure ci-dessous propose à gauche un graphe orienté de 8 sommets et à droite les 3 CFC du graphe. On peut remarquer qu'une des 3 CFC se limite à un seul sommet. On nomme habituellement chaque CFC du nom du sommet de plus petit indice. Ainsi, dans cet exemple, les CFC sont nommée CFC_0, CFC_4 et CFC_3.



Les graphes orientés considérés ici sont implémentés par la structure de données suivante, où les listes `L_succ` et `L_prec` sont des listes simplement chaînées d'éléments `Arc`.

```
typedef struct arc {
  int v ;     // numero sommet queue u
  struct arc *suiv; // pointeur sur le sommet suivant
} Arc ;

typedef struct sommet {
  int u; // nom Sommet
  Arc *L_succ;  // liste des sommets successeurs de ce sommet
  Arc *L_prec;  // liste des sommets predecesseurs de ce sommet
} Sommet ;

typedef struct {
  int nbsom;        // Nombre de sommets
  Sommet* t_som;  // Tableau des sommets
} Graphe;
```

Q 2.1 Donner le code d'une fonction `void creeGraphe(Graphe *G, int n);` qui initialise un élément `Graphe` comme un graphe sans arcs à `n` sommets. Donner le code d'une fonction `void ajoutArc(Graphe *G, int i, int j);` qui ajoute un arc dans la structure.

Q 2.2 Soit un sommet r du graphe. En vous appuyant sur votre cours, expliquer comment déterminer tous les sommets j tel qu'il existe un chemin de r à j. Appliquer la méthode sur l'exemple de la figure pour retrouver tous les sommets j tel qu'il existe un chemin de 1 à j.
On dispose d'une bibliothèque de gestion de file d'entiers. Cette bibliothèque contient le type `File` et les fonctions :
- `void initFile(File* f);` qui initialise une file vide.
- `int estFileVide (File f);` qui renvoie vrai si et seulement si la file est vide.
- `void enfile (File * f, int donnee);` qui ajoute l'élément `donnee` en fin de file.
- `int defile (File * f);` qui retourne la valeur de l'élément en tête de file puis qui supprime cet élément de la file.

Q 2.3 Donner la fonction `void liste_descendants(Graphe *G, int r, int *marquage);` qui retourne un tableau `marquage` (préalablement alloué de taille `G.nbsom`) tel que `marquage[j]` indique si le sommet numéroté `j` est au bout d'un chemin débutant au sommet `r`. Puis, sans donner le code correspondant, expliquer les différences avec une fonction `liste_ascendants` qui retourne un tableau `marquage` tel que `marquage[j]` indique si le sommet numéroté `j` est au départ d'un chemin aboutissant au sommet `r`.

Q 2.4 Donner la fonction `void composantes_fortement_connexes(Graphe *G, int * CFC);` qui retourne un tableau `CFC` (préalablement alloué de taille `G.nbsom` et remplit de l'entier -1) tel que la case `CFC[i]` contienne le nom de la composante fortement connexe contenant le sommet `j` (c'est-à-dire le nom du sommet de plus petit indice de la composante).