TP3 : GRAPHES ET ALGORITHMES

1) Procédures à faire

2) Procédures supplémentaires

Maccarinelli Chloé L2-informatique

1) PROCEDURES A FAIRE :

Graphes non orienté :

Sommet * chercherSommet :

```
Sommet * chercherSommet(ListeSommet *gr, int j){
   while(gr != NULL) { // parcourir la liste
        if (j == gr->debut->num) {
                return gr->debut;
        }
          gr = gr->suivant;
 }
     return NULL;
}
```

Cette procédure parcours la liste des sommets du graphe.

AfficherlisteSommet :

```
void affichierListeSommet(ListeSommet *file) {
  //à faire
  printf("File : ");
  while(file != NULL) {
        //int test = file->debut->num;
        //printf("# %d", test);
        afficherSommet(file->debut);
        file = file->suivant;
 }
        printf("\n");

}
```

Cette procédure affiche la liste des sommets du graphe.

```
Liste des sommets du graphe:

liste sommet : 8 7 6 5 4 3 2 1


Process returned 0 (0x0)   execution time : 0.179 s
Press any key to continue.
```

Parcours largeur :

```
void parcoursLargeur(Sommet *s){
     Maillon *file = NULL;
     if (s == NULL) return;
     file = enfiler(file, s);
     //file = lastFile(file, s);
     while (file !=NULL)
         { //affichierFile(file);
          s = file->val;
          file = defiler(file);
          s->marque = 1;
           printf(" %d ", s->num);
           Adjacents *ladj = s->adj;
           while (ladj != NULL) {
                 Sommet *t;
                 //Sommet *t= (ladj->arete->som1->num==s->num) ? ladj->arete->som2 : ladj->arete->som1;
                 if (ladj->arete->som1->num==s->num) {
                       t= ladj->arete->som2;
                 }else{
                     t= ladj->arete->som1;
                 }
                 if (t->marque != 1) file =enfiler(file, t);
                 t->marque = 1;
                 ladj = ladj->suivant;
                 }
         }
```

Cette procédure parcourt le graphe en largeur (de gauche à droite).

```
parcours en largeur g :

 1  2  3  4  5  6  7  8
Process returned 0 (0x0)   execution time : 0.158 s
Press any key to continue.
```

Parcours profondeur récursif :

```
void parcoursProfondeurRecursif(Sommet *s) { // post visit
     Adjacents *ladj = s->adj;
     if (s->marque == 1) return;
     printf("%d ", s->num);
     s->marque = 1;
     while (ladj != NULL) {
             Sommet *t= (ladj->arete->som1->num==s->num) ? ladj->arete->som2 : ladj->arete->som1;
             parcoursProfondeurRecursif(t);
             ladj = ladj->suivant;
         }
}
```

Cette procédure parcourt le graphe en profondeur.

```
parcours en profondeur g (recursif) :

1 2 4 5 6 7 8 3
Process returned 0 (0x0)   execution time : 0.167 s
Press any key to continue.
```

Parcours profondeur itératif:

```
void parcoursProfondeurIteratif(Sommet *s) { //  visit
    Maillon *pile = NULL;
    if (s == NULL) return;
    pile = empiler(pile, s);

    while (pile !=NULL)
        {
         s = pile->val;
         /// ici problème avec la pile
         //pile = depiler(pile);
         s->marque = 1;
          printf(" %d ", s->num);
          Adjacents *ladj = s->adj;
          while (ladj != NULL) {;
                Sommet *t;
                if(ladj->arete->som1->num==s->num){
                    t = ladj->arete->som2;

                }else{
                   t=ladj->arete->som1;
                }

                if (t->marque != 1) pile =empiler(pile, t);
                t->marque = 1;
                ladj = ladj->suivant;
                }

        }
}
```

Cette procédure parcourt également le graphe en profondeur.

Empiler, Enfiler :

```
Maillon * enfiler(Maillon *file, Sommet *s) {
    // à faire
    Maillon *p = (Maillon *)malloc(sizeof(Maillon));
    p->val = s;
    p->suiv = NULL;
    if (file == NULL) file = p;
    else {
          Maillon *t = file;
          while (t->suiv != NULL) t = t->suiv;
          t->suiv = p;
         }
      return file;
}
```

Cette procédure permet de mettre les sommets du graphe dans une file. On l'utilise pour le parcours de graphe en largeur.

```
Maillon *empiler(Maillon *pile, Sommet *s) {
     // à faire

    Maillon *p = (Maillon *)malloc(sizeof(Maillon));
  p->val = s;
  p->suiv = pile;
 // if (pile == NULL) *pile = *p;
 pile=p;


}
```

Cette procédure permet de mette les sommets du graphe dans une pile. Utile pour le parcours de graphe en profondeur.

Dépiler, Défiler :

```
Maillon * defiler(Maillon *file) {// à faire
     Maillon *p = (Maillon *)malloc(sizeof(Maillon));
     p = file;
     file = file->suiv;
     free(p);
     return file;
}
```

Cette procédure permet d'enlever les éléments de la file qui contient les sommets du graphe.

```
Sommet * depiler(Maillon *pile) {
     // à faire

      if (pile != NULL)
   {

    Maillon *p=pile;
    pile=pile->suiv;
    return p->val;
   }
     return NULL;


}
```

Cette procédure permet d'enlever les éléments de la pile le premier élément entré sera le dernier sorti :FIFO.

Graphes orientés :

On a les mêmes procédures et on obtient :

```
File : 8 7 6 5 4 3 2 1

parcours en profondeur g2 (recursif) :
1 2 4 5 6 7 8 3
parcours en largeur g2 :
 1

Process returned 0 (0x0)   execution time : 0.168 s
Press any key to continue.
```

Pour les prochains algorithmes le problème rencontré est lors de l'exécution à l'affichage dans le terminal : rien ne s'affiche.

Est-Connexe :

```
int est_Connexe(ListeSommet *g)
{
    Adjacents *p;
    Sommet *q;
    ListeSommet *r=g;
    Maillon *pile = NULL;
    pile = empiler(pile, (*g).debut);
    (*(*g).debut).marque= 1;

    while(pile != NULL)
    {
        q=depiler(pile);
        p = (*q).adj;
        while(p != NULL)
        {
            if((*(*(*p).arete).som1).num == (*q).num)
            {
                if((*(*(*p).arete).som2).marque != 1)
                {
                    pile = empiler(pile,(*(*p).arete).som2);
                    (*(*(*p).arete).som2).marque = 1;

                }

            }else
            {
                if((*(*(*p).arete).som1).marque != 1)
                {
                    pile = empiler(pile,(*(*p).arete).som1);
                    (*(*(*p).arete).som1).marque = 1;

                }
            }
            p = (*p).suivant;
        }
    }
    while(r != NULL)
    {
        if((*(*r).debut).marque==0)
        {
            return(0);
        }
        r=(*r).suivant;
    }
```

Est-fortement-connexe :

```
int est_fortementConnexe(ListeSommet *g)
{
    CFC *h;
    h = Composante_fortementConnexe(g);
    afficher_CFC(h);
    if((*h).suivant == NULL)
    {
        return(1);
    }
    return(0);
}
```

2) PROCEDURES ET FONCTIONS SUPPLEMENTAIRES :

Sommet * lastFile :

```
Sommet *lastFile(Maillon *file)
{
    return (*file).val;
}
```

Cette fonction renvoi le dernier élément de la file.

BONUS :

Voici les entêtes des procédures utilisées pour Composantes Fortement Connexes :

```
CFC *Composante_fortementConnexe(ListeSommet *g)
{

ListeSommet* grapheInverse(ListeSommet *g) ///rajoutée
{

void afficher_CFC(CFC* g) ///rajoutée
{

Maillon *parcoursRecursifCFC(Sommet *s, Maillon *pile) ///rajoutée
{
```