Algorithmie Avancée

TP/TD Arbre couvrant

A. Document Algo_TDParE_Birmele.pdf

Vous traiterez les exercices 2.1 à 2.3

B. Prim sur une matrice d'adjacences

Vous implémenterez en C un algorithme de Prim en vous inspirant éventuellement du canevas proposé ci-après.

```
#define INFINI 1000.0 // un poids réel supérieur à la plus grande longueur totale
        // Fonction qui retourne l'arbre couvrant de poids minimum d'un graphe valué et non orienté
        // depuis un sommet de référence aléatoire
        //      Paramètres :
        //      adjacencePoids : matrice d'adjacence pondérée du graphe
        //      ordre : nombre de sommets
t_arete * prim (float **adjacencePoids, int ordre) {
        // Variables locales
        t_arete *arbre ; // arbre d'incidence nœud-arc de poids minimum à retourner
        int indiceA = 0 ; // indice de l'arbre initialisé à 0
        int *marques ; // tableau dynamique indiquant si les sommets sont marqués ou non
        int s, x, y, ymin ; // numéros de sommets intermédiaires
        float min ; // distance minimale du prochain sommet à marquer

        // Allouer l'arbre de « ordre-1 » arêtes et le tableau marque de « ordre » entiers
        …
        // Initialiser le marquage des sommets à 0
        …
        // Choisir un sommet s aléatoirement compris entre 0 et ordre-1
        s = rand()%ordre ;

        // Marquer le sommet aléatoire s
        marques[s] = 1 ;
        //tant que les arêtes de l'arbre ne sont pas toutes traitées
        while (indiceA<ordre-1) {

        // Initialiser la longueur minimale à l'INFINI
        min = INFINI ;

        // Pour tous les sommets x marqués
        // Chercher le sommet de longueur minimale « ymin » adjacent à x
        // et non marqué
        for (x=0 ; x<ordre ; x++)
                        if (marques[x])
                        for (y=0 ; y<ordre ; y++)
                                        if (adjacencePoids[x][y] &&  !marques[y] &&
                                            adjacencePoids[x][y]<min) {
                                        min = adjacencePoids[x][y] ; // poids min
                                        ymin = y ; // sommet y de poids min
                                        }

        // marquer le sommet « ymin » de longueur minimale
        marques[ymin] = 1 ;
        // Insérer l'arête (x, ymin) de longueur min à la position « indiceA » de l'arbre
        …
        // Passer à l'arête suivante de l'arbre
        indiceA++ ;
        }
        return arbre ; // retourner l'arbre de poids minimum
}
```

Activité supplémentaire : Kruskal sur tableau d'arêtes

Vous implémenterez en C un algorithme de Kruskal en vous inspirant éventuellement du canevas proposé ci-après.

```
// Fonction qui retourne l'arbre couvrant de poids minimum d'un graphe valué et non orienté
// depuis un sommet de référence
//        Paramètres :
//        graphe : tableau d'arêtes du graphe
//        ordre : nombre de sommets
//        s : numéro de sommet de référence
//        n : nombre d'arêtes du graphe

t_arete * kruskal (t_arete * graphe, int ordre, int s, int n) {
        // Variables locales
        t_arete *arbre ; // tableau d'arêtes de poids minimum à retourner
        int *connexe ; // tableau dynamique des numéros de sommets connexes de l'arbre
        int indiceA = 0, indiceG = 0 ; // indices de l'arbre et du graphe initialisés à 0
        int x, s1, s2 ; // numéros de sommets intermédiaires
        t_arete u ; // arête reliant 2 sommets x1 et x2

        // Allouer l'arbre de « ordre - 1 » arêtes
        …

        // Allouer le tableau connexe de « ordre » sommets
        …

        // Initialiser les connexités indicées sur les numéros de sommets
        for (x=0 ; x<ordre ; x++) connexe[x] = x ;

        // Trier le graphe par ordre croissant des poids de ses « n » arêtes
        …

        // tant que les arêtes de l'arbre et du graphe ne sont pas toutes traitées
        while (indiceA<ordre-1 && indiceG<n) {
                u = graphe[indiceG] ; // retourner l'arête u numéro indiceG du graphe
                s1 = connexe[u.x] ; s2 = connexe[u.y] ; // les sommets s1, s2 de l'arête u

                // Tester si les sommets s1 et s2 de l'arête u forment un cycle dans l'arbre
                if (s1==s2) // cycle si s1 et s2 connexes
                    indiceG++ ; // passer à l'arête suivante du graphe

                else { // pas de cycle
                    // insérer l'arête u à la position « indiceA » de l'arbre
                    arbre[indiceA] = u ;

                    indiceA++ ; indiceG++ ; // passer à l'arête suivante de l'arbre et du graphe

                    // Indiquer que les sommets s1 et s2 sont connexes
                    for (x=0 ; x<ordre ; x++)
                          if (connexe[x]==s1)     connexe[x] = s2 ;
                }
        }

        // Le graphe est non connexe si le nombre d'arêtes de l'arbre < nombre de sommets-1
        if (indiceA<ordre-1) { printf("Le graphe n'est pas connexe\n") ; }

        return arbre ; // retourner l'arbre de poids minimum
}
```