TP 15 : Graphes

5 juin 2024

Graphes en C

Matrices et Listes d'Adjacences

On considère pour les questions suivantes que l'on représente les n sommets d'un graphe d'ordre n par les entiers entre 0 et $n-1$.

1) Définir un type `mat_graphe` avec un champ `taille` entier et un champ `mat` correspondant à un tableau de booléens en deux dimensions, pour représenter un graphe à l'aide de sa matrice d'adjacence.

2) Définir un type `liste_graphe` avec un champ `taille` entier et un champ `l` correspondant à un tableau de listes chaînées, pour représenter un graphe à l'aide de sa liste d'adjacence.

3) Ecrire une fonction `creer_mat_graphe` qui prend en entrée une taille n et crée et renvoie la matrice d'adjacence d'un graphe à n sommets et 0 arêtes. Créer la fonction `supprimer_mat_graphe` qui prend en entrée un graphe sous forme de matrice d'adjacence et libère son espace en mémoire dans le graphe.

4) Ecrire une fonction `ajout_arete_go` qui prend en entrée un graphe sous la forme d'une matrice d'adjacence, deux sommets i et j et modifie le graphe en ajoutant l'arête correpsondante, dans le cas d'un graphe orienté. Ecrire une fonction `ajout_arete_gno` qui fait la même chose pour un graphe non-orienté.

5) Ecrire une fonction `nb_aretes_go_mat` qui prend en entrée un graphe orienté sous la forme d'une matrice d'adjacence et renvoie son nombre d'arêtes.

6) Ecrire une fonction `nb_aretes_go_list` qui fait la même chose pour un graphe donné sous la forme d'une liste d'adjacence.

7) Ecrire une fonction `mat_to_list` qui prend en entrée un graphe sous la forme d'une matrice d'adjacence et le renvoie sous la forme d'une liste d'adjacence, ainsi qu'une fonction `list_to_mat` pour faire l'opération dans l'autre sens.

8) Ecrire une fonction `est_gno` qui prend en entrée un graphe sous la forme d'une matrice d'adjacence, et renvoie Vrai s'il s'agit de la matrice d'un graphe non-orienté et Faux sinon.

Parcours de Graphes

On définit les types fonctions suivantes, contenues dans le fichier `structures.c` : un type `pile` sous la forme d'une liste chaînée, avec des fonctions `ajout_pile`, `creer_pile`, `depiler`

et `isempty_pile` pour effectuer les opérations élémentaires sur les piles.

1) A l'aide de ces fonctions, écrire une fonction `parcours_profondeur` qui prend en entrée un pointeur vers un graphe représenté sous la forme d'une liste d'adjacence, un sommet initial, et effectue le parcours du graphe à partir du sommet initial en affichant la liste des sommets parcourus dans l'ordre de parcours.

2) Définir une fonction `parcours_profondeur_bis` qui effectue le parcours en profondeur sans utiliser de pile, en utilisant une fonction auxiliaire récursive du parcours en profondeur d'une partie de graphe à partir d'un sommet donné.

3) Définir un type `file` avec les opérations nécessaires pour implémenter la structure de file en C, et écrire une fonction `parcours_largeur` en adaptant la fonction précédente.

4) Utiliser la fonction `parcours_profondeur` pour calculer l'ensemble des composantes connexes du graphe.

5) Adapter la fonction `parcours_largeur` pour écrire une fonction `distance` qui prend en entrée un graphe sous la forme d'une liste d'adjacence et deux sommets, et renvoie la distance entre les deux sommets.

Graphes en OCaml

On va maintenant définir les implémentations des graphes en OCaml et les algorithmes associés.

Parcours de graphes

On définit le type `graphe` de façon analogue au type `liste_graphe` défini en C précédemment, c'est-à-dire sous la forme d'un tableau contenant les listes d'adjacences :

```
type graph = int list array ;;
```

1) Définir une fonction `parcours_profondeur_rec` qui prend en entrée un graphe et un sommet initial s (entier), et affiche les sommets dans la composante connexe de s dans l'ordre du parcours en profondeur du graphe, en utilisant une implémentation analogue à celle de `parcours_profondeur_bis` dans la section précédente.

2) Définir une fonction `parcours_largeur` qui prend en entrée un graphe et un sommet s (entier), et affiche les sommets dans l'ordre du parcours en largeur.

Algorithme de Dijkstra

On utilise ici une implémentation des graphes orientés pondérés en utilisant une représentation sous forme de liste d'adjacence, analogue à la précédente. Ici, les listes d'adjacences de chaque sommet s sont donnés sous la forme d'une liste de couples d'entier et de flottant : le sommet s' adjacent à s, et le poids de l'arête (s, s').

```
type graphe_pondere = (int * float) list array ;;
```

On stockera les couples sommets/distances à l'origine dans une file de priorité implémentée naïvement, sous la forme d'une liste de couples de type (entier, flottant) :

```
type file_prio = (int * float) list ;;
```

1) Implémenter les fonctions suivantes sur les files de priorités :

- `ajout_file_prio`, qui prend en entrée une file de priorité et un élément et ajoute l'élément à la bonne place dans la file de priorité ;
- `premier_elt_file_prio` qui renvoie l'élément prioritaire dans la file de priorité ;
- `enlever_file_prio` qui renvoie la file de priorité privée de son premier élément.

2) On utilise le type `option`, qui a un constructeur sans argument `None` et un constructeur avec argument `Some`, pour représenter les distances dans le graphe : si le chemin minimal entre l'origine s et un sommet s' est de distance 5, on la représentera par `Some(5.)`. S'il n'existe pas de chemin, on représentera la distance par `None`.

Ecrire une fonction `dijkstra` qui prend en entrée un graphe pondéré orienté sous forme de liste d'adjacence, un sommet initial, et renvoie un couple de tableau, avec dans le premier tableau les distances de chaque élément au sommet initial dans le graphe, et dans le deuxième tableau leur prédécesseur dans le plus court chemin. (On représente un prédécesseur avec le constructeur `Some`, et l'absence de prédécesseur avec le constructeur `None`)