TP 4 : Algorithmes sur les Graphes (Partie 1)

7 novembre 2024

1 Rappels

1.1 Implémentation des Graphes

On associe les sommets d'un graphe à des entiers. On rappelle que les graphes (orientés) peuvent être représentés par des matrices d'adjacences ou par des listes d'adjacences (donnée pour chaque sommet de la liste de ses successeurs).

On définit en OCaml le type suivant pour représenter les graphes orientés par liste d'adjacence, ainsi qu'une fonction pour connaître l'ordre du graphe :

```
type graphe_oriente = int list array ;;

let ordre (g : graphe_oriente) : int = Array.length g ;;
```

La liste d'adjacence est un tableau dont la taille est l'ordre du graphe. Pour un graphe `g` de type `graphe_oriente`, pour un entier `i`, `g.(i)` correspond à la liste des successeurs de `i` dans le graphe.

1.2 Parcours en Largeur

On rappelle la procédure pour les parcours en largeur et en profondeur à partir d'un sommet :

- On stocke dans une structure (tableau par exemple) l'information de visite des différents sommets du graphe.
- On stocke dans une structure S les éléments à traiter ; on ajoute initialement la source souhaitée.
- A chaque élément retiré de la structure S, si celui-ci n'a pas déjà été visité, on ajoute ses successeurs dans la structure, et on le marque comme visité.

Cette procédure effectue le parcours de la composante connexe d'un sommet s dans le cas d'un graphe non-orienté, et de l'ensemble des sommets accessibles à partir de s dans un graphe orienté.

Dans le cas d'un graphe orienté, appliquer un parcours de graphe à partir de tout sommet non-visité précédemment permet de compter le nombre de composantes connexes.

Selon la structure S choisie, le type de parcours effectué ne sera pas le même : l'utilisation d'une file donne un parcours en largeur tandis que celui d'une pile donne un parcours en profondeur.

En OCaml, on peut utiliser le module `Queue` pour manipuler des files. Le module contient notamment les fonctions suivantes :

- `Queue.create ()` crée une file vide ;
- `Queue.is_empty q` vérifie qu'une file est vide ;
- `Queue.enqueue q e` rajoute l'élément `e` à la file `q` ;
- `Queue.dequeue q` enlève un élément dans la file.

1) Ecrire une fonction en OCaml `parcours_largeur (g : graphe_oriente) (source : int) : int * int array` qui prend en entrée un graphe orienté sous la forme d'une liste (tableau) d'adjacence, une source dans le graphe, et renvoie le nombre n de sommets accessibles depuis la source, ainsi qu'un tableau dont les n premières cases sont ces sommets dans leur ordre de parcours.

2) Ecrire une fonction en OCaml `parcours_largeur_distance (g : graphe_oriente) (source : int) : int array` qui prend en entrée un graphe orienté sous la forme d'une liste (tableau) d'adjacence, une source dans le graphe, et renvoie un tableau `dist` tel que `dist.(i)` est la distance du sommet `i` à la source choisie, calculée à partir d'un parcours en profondeur.

Remarque. Pour une fonction `f` de type `'a -> unit`, pour une liste `l` de type `'a list`, on peut utiliser `List.iter f l` appliquer `f` successivement à tous les éléments de la liste `l`.

1.3 Parcours en Profondeur (itératif)

Il est également possible d'effectuer la procédure avec une pile à la place d'une file pour obtenir un parcours en profondeur.

De la même façon, en OCaml, le module `Stack` permet d'utiliser des piles, avec notamment les fonctions suivantes :

- `Stack.create ()` crée une file vide ;
- `Stack.is_empty s` vérifie qu'une file est vide ;
- `Stack.push s e` rajoute l'élément `e` au sommet de la pile `s` ;
- `Stack.pop s` enlève un élément dans la pile.

3) Adapter la fonction de la question 1) pour écrire une fonction `parcours_profondeur (g : graphe_oriente) (source : int) : int * int array` qui effectue le parcours en profondeur d'un graphe à partir d'une source donnée.

1.4 Parcours en Profondeur (récursif)

Le parcours en profondeur d'un graphe peut également être défini récursivement : à partir d'un sommet source, à partir d'une structure indiquant les sommets déjà visités (initialement vide), on indique que la source a été visitée, on parcours la liste des successeurs du sommet source, et pour chacun de ceux-ci, on applique récursivement l'algorithme de parcours de graphe dessus si le successeur n'a pas déjà été visité.

4) Ecrire une fonction `parcours_profondeur_rec (g : graphe_oriente) (source : int) : bool array` qui renvoie un tableau de booléens indiquant pour chaque sommet s'il est accessible depuis la source.

1.5 Tri Topologique

5) En s'inspirant de la fonction précédente, écrire une fonction `tri_topo (g : graphe_oriente) : int list` qui effectue successivement des parcours en profondeur à partir des sommets du graphe non-visités et renvoie la liste des sommets visités dans l'ordre postfixe (qui correspond à un tri topologique comme on l'a vu en cours).

2 Calcul de Composantes Connexes

2.1 Composantes Connexes dans un Graphe Non-Orienté

On suppose dans cette question qu'on représente un graphe non-orienté avec un élément de type `graphe_oriente` tel que toutes arête entre deux sommets existe dans les deux sens.

A l'aide d'un parcours de graphe, crire une fonction `composantes_connexes (g : graphe_oriente) : int * int array` qui prend en entrée un élément de type `graphe_oriente` représentant un graphe non-orienté et qui renvoit le nombre de composantes connexes du graphe ainsi qu'un tableau indiquant le numéro de la composante connexe de chaque sommet.

2.2 Composantes Fortement Connexes dans un Graphe Orienté

Ecrire une fonction `kosaraju (g : graphe_oriente) : int * int array` qui prend en entrée un graphe non-orienté et renvoie le nombre de composantes fortement connexe du graphe ainsi que le numéro de la composante fortement connexe de chaque sommet.