TP n°29 - Graphes - Implémentations concrètes

Tous les fichiers de code devront être déposés sur Moodle pour vendredi prochain.

Dans ce TP, on cherche à implémenter une structure de données de graphe pondéré, orienté ou non, **avec un nombre de sommets fixé au départ**. Cette structure de données sera nommée `wgraph` pour *weighted graph*.

Nous coderons tout d'abord une version très générale, que nous adapterons ensuite pour gérer les cas plus restreints des graphes non orientés et/ou non pondérés.... qui peut le plus peut le moins !

Le but de ce TP est de coder les 3 implémentations concrètes de graphe vues en cours.
L'interface abstraite sera, bien sûr, à peu de choses près ; la même pour les 3 implémentations concrètes.
La voici pour les deux premières implémentations en C :

```c
typedef struct wgraph_t wgraph;

// constructeur
wgraph *wgraph_init_no_edge(int nv, bool directed);

// destructeur
void    wgraph_free(wgraph **addr_g);

// accesseurs
list   *wgraph_succ(wgraph *g, int u); // liste des successeurs d'un sommet
void    wgraph_print(wgraph *g);
int     wgraph_number_of_vertices(wgraph *g);
int     wgraph_number_of_edges(wgraph *g);

// transformateurs
bool    wgraph_add_edge(wgraph *g, int u, int v, double weight);
bool    wgraph_remove_edge(wgraph *g, int u, int v);
```

Cette interface abstraite est mise à votre disposition sous la forme d'un fichier d'en-tête `mygraph.h`.

A noter que, si l'arête existe déjà, la fonction `add_edge` ne fait que remplacer l'ancien poids par la nouvelle valeur fournie et renvoie `false` car il n'y a pas de création d'arête à proprement parler. Cette fonction renvoie true seulement si une vraie nouvelle arête a été créée.
La fonction `remove_edge` renvoie `false` si l'arête que l'on cherche à supprimer n'existe en fait pas, et `true` seulement si l'arête a été trouvée et supprimée.
On met à disposition sur Moodle une bibliothèque de liste en langage C sous la forme de deux fichiers `mylist.h` et `mylist.c` que vous pourrez utiliser dans votre code.

On essaiera de coder de la manière la plus modulaire possible, notamment :

- Vos deux implémentations concrètes en C, l'une avec matrice et l'autre avec tableau de listes, seront codées dans deux fichiers C, `NOM_mygraph_mat.c` et `NOM_mygraph_lst.c`
- Vous créerez un fichier `NOM_test.c` qui contiendra plusieurs fonctions de test permettant de tester pas à pas vos implémentations (et donc le `main`).
- Vous créerez un script Shell de compilation `NOM_compile.sh` qui vous permettra facilement de tester une implémentation concrète ou une autre, avec le MEME fichier `NOM_test.c`.

Exercice 1 (Implémentation concrète avec matrice d'adjacence en C).

Toutes les fonctions de cette partie seront codées dans un fichier `NOM_mygraph_mat.c`.

1. Implémenter les primitives détaillées dans l'interface abstraite en utilisant une implémentation avec matrice d'adjacence. On testera très régulièrement et exhaustivement chaque fonction (notamment les cas limites) grâce au fichier `test.c` et on respectera scrupuleusement l'interface abstraite fournie sur Moodle.
2. Quelles sont les complexités des différentes primitives de votre implémentation ?
3. Quelle est la complexité spatiale de votre implémentation ?

Exercice 2 (Implémentation concrète avec un tableau de listes en C).

Toutes les fonctions de cette partie seront codées dans un fichier `NOM_mygraph_lst.c`.

1. Implémenter les primitives détaillées dans l'interface abstraite en utilisant une implémentation avec un tableau de listes.
   On testera très régulièrement et exhaustivement chaque fonction (notamment les cas limites) et on respectera scrupuleusement l'interface abstraite fournie sur Moodle.
2. Quelles sont les complexités des différentes primitives de votre implémentation ?
3. Quelle est la complexité spatiale de votre implémentation ?

Exercice 3 (Implémentation par table de hachage en OCaml - Création d'un module).

On pourra pour cet exercice utiliser les fonctions des modules `Hashtbl` et `List`.
Avec la matrice d'adjacence, tester la présence d'une arête se fait en temps constant, mais obtenir les voisins d'un sommet est coûteux. Avec les listes adjacence, c'est le contraire.
Il est cependant possible de réconcilier les deux en représentant l'adjacence d'un sommet non pas comme une liste mais comme une table de hachage. Ainsi, toutes les opérations (tester, ajouter, ou supprimer une arête, obtenir la liste des voisins) auront une complexité optimale.

1. Définir en premier lieu un type OCaml polymorphe `'w t` permettant d'implémenter concrètement une structure de données de graphe pondéré comme expliqué ci-dessus.
2. Implémenter toutes les fonctions décrites dans la signature (interface abstraite) :

```
module Wgraph :
  sig
    type 'w t

    (* constructeur *)
    val init_no_edge: int -> bool -> 'w t

    (* accesseurs *)
    val succ: 'w t -> int -> (int*'w) list
    val print: float t -> unit
    val number_of_vertices : 'w t -> int
    val number_of_edges : 'w t -> int

    (* transformateurs *)
    val add_edge:  'w t -> int -> int -> 'w -> bool
    val remove_edge:  'w t -> int -> int -> bool

  end
;;
```

   On respectera scrupuleusement l'interface abstraite ci-dessus et on fera des tests régulièrement
3. Nous allons maintenant mettre en œuvre les principes de la modularité et créer un module `Wgraph` qui pourra nous resservir !
   Vous avez à disposition sur Moodle l'interface abstraite du module `Wgraph` sous la forme d'un fichier `Wgraph.mli`. Observez la syntaxe `sig...end;;` qui encadre l'interface abstraite du module, appelée **signature** (sig) en OCaml.

   a. Pour créer votre implémentation concrète du module, complétez le fichier `Wgraph.ml` renommé `NOM_Wgraph.ml` en intégrant les fonctions implémentées dans l'implémentation concrète du module. Essayez de retenir cette syntaxe `struct...end;;` qui encadre l'implémentation concrète du module.

   b. Ouvrir le script Shell `check_ocaml_implem.sh` et lire les commentaires. Lancer ce script pour vérifier l'adéquation de votre implémentation concrète avec cette interface abstraite.

   c. Déplacer et réorganisez proprement vos tests dans un fichier `NOM_test.ml` que vous exécuterez en mode interactif avec Tuareg. Pour importer le module `Wgraph` on utilisera la directive : `#use toto.ml`(cf TP Huffman).

   d. Quelles sont les complexités des différentes primitives de votre implémentation ?

   e. Quelle est la complexité spatiale de votre implémentation ?

Exercice 4 (Améliorations, à faire à la maison après avoir bien compris les 3 exercices précédents).

1. Ajouter dans vos 3 implémentations un accesseur nommé `has_edge` qui indique si une arête (u, v) est présente ou non dans le graphe.
2. Sauvegardez l'implémentation par liste d'adjacence et par tables d'adjacence OCaml puis améliorer ces deux implémentations pour permettre l'introduction de nouveaux sommets dans le graphe en cours de construction. Pour cela, vous créerez deux nouvelles primitives de transformation :

   `add_vertex :` rajoute un sommet d'étiquette n_v, où n_v était le nombre de sommets avant cet ajout, et retourne l'étiquette de ce nouveau sommet, ou bien -1 si l'opération a échoué. En OCaml, on pourra utiliser un type `'a option`.

   `remove_vertex :` enlève un sommet existant d'étiquette donnée du graphe, et toutes les arêtes connectées à ce sommet ; renvoie `false` si l'opération s'est mal passée (par exemple si le sommet à supprimer n'existait pas) et `true` sinon.

   Vous devrez bien sûr adapter la structure de données :

   D'abord en utilisant des tableaux de taille fixe avec une capacité maximale

   Puis, dans un second temps, en utilisant une table de hachage à la place du tableau, pour améliorer la complexité spatiale, gagner en flexibilité, sans perdre en complexité temporelle (puisque une table de hachage est proche en terme de performance, de l'association par adressage direct que donnait l'implémentation par tableau)

   Vous devrez également modifier des fonctions existantes pour être certains de ne pas créer d'arêtes entre des sommets ... qui n'existent pas encore ou qui n'existent plus par exemple.

   Et vous testerez bien sûr dans votre fichier `test.c`, en créant de nouvelles fonctions de test appelées ensuite dans le `main`.