TP 27

Représentation des graphes

Dans ce TP, on se propose d'étudier différentes implémentations possibles pour représenter les graphes en OCaml.

Dans les premières parties, on suppose que les sommets sont fixés une fois pour toutes et numérotés de 0 à $n-1$. On utilise soit une matrice d'adjacence, soit des listes d'adjacence. On s'intéresse ensuite à des graphes à nombre de sommets variable et propose l'utilisation de dictionnaires d'adjacence à l'aide de tables de hachage.

1 Interface abstraite

Comme on veut permettre la construction d'algorithmes génériques indépendants de la structure de données choisie, il est nécessaire de définir l'interface abstraite que doit vérifier toute implémentation de graphe. Comme nous le verrons, différentes implémentations peuvent réaliser les opérations avec des complexités différentes, ce qui pourra par la suite guider le choix d'une implémentation ou d'une autre en fonction des applications.
On considère donc tout d'abord que les sommets sont représentés par les entiers de 0 à $n-1$, c'est-à-dire que $S = [\![0, n[\![$. Le code suivant propose une signature minimale pour ce type de graphe qui se trouve également dans le fichier `graph.mli` (inutile de le recopier).

```
(* Abstract interface for a graph *)

type vertex = int
type edge = vertex * vertex
type graph

(* Create a graph with a given number of vertices *)
val create_graph : int -> graph
(* Return the number of vertices *)
val nb_vertices : graph -> int

(* Check whether an edge is in the graph *)
val has_edge : graph -> edge -> bool
(* Add edge to the graph *)
val add_edge : graph -> edge -> unit
(* Remove edge from the graph *)
val remove_edge : graph -> edge -> unit
```

```
(* Return neighbours of a vertex in a graph*)
val neighbours : graph -> vertex -> vertex list
(* Return all edges from a graph *)
val all_edges : graph -> edge list
```

Remarque. La signature convient aussi bien aux graphes orientés et non-orientés. Dans le cas orienté, les voisins (*neighbours*) sont les successeurs d'un sommet (*vertex*). On commencera à chaque fois par les graphes orientés.

1. S'agit-il d'une structure de données persistante ou impérative ?

II Matrice d'adjacence

Dans cette partie, un graphe est donné par sa matrice d'adjacence :

```
type graph = bool array array
```

2. Implémenter la structure de données de *graphes orientés* en utilisant une matrice d'adjacence, c'est-à-dire écrire en OCaml toutes les fonctions de la signature précédente.
3. Quelle est la complexité des différentes opérations ?
4. Quelle est la complexité en espace de cette représentation ?
5. Implémenter la fonction `outdegree` ainsi que la fonc- tion `indegre` permettant le calcul des degrés sortants d^+ et entrants d^-, respectivement.

```
outdegree : graph -> vertex -> int
```

```
indegree : graph -> vertex -> int
```

6. Modifier votre implémentation de graphe orienté pour obtenir une structure de données pour les *graphes non-orientés*.
   N'oubliez pas de sauvegarder la version pour les graphes orientés. Le plus simple est de copier votre code dans un nouveau fichier pour le modifier.
7. Écrire une fonction prenant un graphe orienté et le transformant en sa version non-orientée. La fonction aura pour type `graph -> unit`.

III Listes d'adjacence

Dans cette partie, un graphe est donné par son tableau de listes d'adjacence.

```
type graph = vertex list array
```

8. Reprendre les questions 2 à 7, en utilisant cette fois-ci des listes d'adjacence.

IV D'une représentation à l'autre

9. Écrire une fonction prenant en entrée un graphe représenté sous forme d'une matrice d'adjacence et renvoyant une représentation du même graphe sous forme de tableau de listes d'adjacence. On fera en sorte que les listes d'adjacence soient triées par ordre croissant.

```
lists_of_matrix : bool array array -> vertex list array
```

10. Écrire une fonction réalisant la transformation inverse.

```
matrix_of_lists : vertex list array -> bool array array
```

V Quelques manipulations sur les graphes

A. Miroir d'un graphe orienté

Si $G = (V, E)$ est un graphe orienté, son graphe miroir $G^{\leftarrow}$ est obtenu en gardant le même ensemble de sommets et en inversant le sens de tous les arcs :

$$G^{\leftarrow} = \Big(V,\ \{(v,u),\ (u,v) \in E\}\Big).$$

11. Écrire une fonction `mirror` réalisant cette tâche.

```
mirror : graph -> unit
```

B. Validité d'un coloriage

Soit $G = (V, E)$ un graphe non orienté et soit $f : V \to \mathbb{N}$ une fonction. La fonction f est un *coloriage* de G lorsque :

$$\text{pour tout } uv \in E,\ f(u) \neq f(v).$$

12. Écrire une fonction `coloriage` qui détermine si une fonction (donnée par un tableau) est un coloriage du graphe G, donné par listes d'adjacence.

```
is_coloring_valid : graph -> int array -> bool
```

VI Dictionnaires d'adjacence

Les deux représentations précédentes supposent que les sommets sont fixés une fois pour toutes. Dans de nombreuses applications (penser à un réseau d'ordinateurs où un nouvel ordinateur peut se connecter/déconnecter à tout moment), on souhaite pouvoir ajouter ou supprimer des sommets dynamiquement. On peut imaginer marquer les sommets supprimés pour les exclure du graphe, mais cela aura un impact négatif sur la complexité.
On cherche donc à implémenter la signature suivante (fichier `dyngraph.mli`). Remarquer que le type `vertex` a disparu : ce peut être désormais n'importe quel type `'a`.

```
type 'a edge = 'a * 'a
type 'a graph

(* Create an empty graph *)
val create_graph : unit ->'a graph

(* Check whether a vertex is in the graph *)
val has_vertex : 'a graph -> 'a -> bool
(* Add vertex to the graph *)
val add_vertex : 'a graph -> 'a -> unit
(* Remove vertex from the graph *)
val remove_vertex : 'a graph -> 'a -> unit

(* Check whether an edge is in the graph *)
val has_edge : 'a graph -> 'a edge -> bool
(* Add edge to the graph *)
val add_edge : 'a graph -> 'a edge -> unit
```

```
(* Remove edge from the graph *)
val remove_edge : 'a graph -> 'a edge -> unit

(* Return neighours of a vertex in a graph*)
val neighbours : 'a graph -> 'a -> 'a list
(* Return all vertices from a graph *)
val all_vertices : 'a graph -> 'a list
(* Return all edges from a graph *)
val all_edges : 'a graph -> 'a edge list
```

On se propose pour cela d'utiliser ici les structures de dictionnaires et d'ensembles. Un graphe peut être vu comme un dictionnaire qui associe à chaque sommet l'ensemble de ses successeurs.
Autrement dit, on conserve l'idée des listes d'adjacence, mais on remplace le tableau de listes par un dictionnaire et les listes par des ensembles. On appelle cette structure un dictionnaire d'adjacence.
Concernant l'efficacité, on peut utiliser une table de hachage à la fois pour le dictionnaire et pour les ensembles. Nous utiliserons la bibliothèque `Hashtbl` de OCaml.
Une table de hachage en OCaml est de type `('a, 'b) hashtbl.t` où `'a` est le type des clés et `'b` est le type des valeurs associées aux clés.
Un ensemble peut être vu comme un dictionnaire associant une clé à rien (ici `()` de type `unit`).
On peut donc définir :

```
type 'a set = ('a, unit) Hashtbl.t
type 'a graph = ('a, 'a set) Hashtbl.t
```

Créer un graphe revient à construire un dictionnaire vide :

```
let create () = Hashtbl.create 42
```

13. Implémenter la structure de données de graphe orienté à sommets dynamiques, en utilisant des dictionnaires d'adjacence.
14. En supposant que les opérations élémentaires sur les tables de hachage sont de coût constant (ce qui est raisonnable) quelle est la complexité temporelle des opérations ?

VII Petit bonus

The *d-dimensional hypercube* is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

15. A *Hamiltonian cycle* in a graph G is a cycle of edges in G that enters each vertex of G exactly once. Prove that for all $d \geqslant 2$, the d-dimensional hypercube has a Hamiltonian cycle.
16. Write a function that lists the vertices of a Hamiltonian cycle for the 2^d hypercube.