Représentation des graphes

option informatique

Vers une structure de données

Expression des besoins

- Il existe de nombreuses manières d'implementer les graphes dans un langage de programmation.
- Toutes présentent des éléments communs en définissant un **type** associé à un graphe et des **fonctions de manipulation**.
- Cet exposé présente différentes implémentations en Ocaml et les compare.
- Les **fonctions de parcours** de graphes font l'objet d'une présentation spécifique ultérieure.

Signatures de graphe non étiqueté

Quelle que soit l'implémentation, un **type** `graph` et les **fonctions de manipulation** dont les signatures sont données ci-dessous sont définis.

```
type 'a graph = ...

vertex_exist :      'a graph -> 'a -> bool
edge_exist :        'a graph -> 'a -> 'a -> bool

vertex_neighbors :  'a graph -> 'a -> 'a list

edge_add :          'a graph -> 'a -> 'a -> unit
edge_remove :       'a graph -> 'a -> 'a -> unit
vertex_add :        'a graph -> 'a -> unit
vertex_remove :     'a graph -> 'a -> unit
```


- arrtes@gmail.com

Implémentations

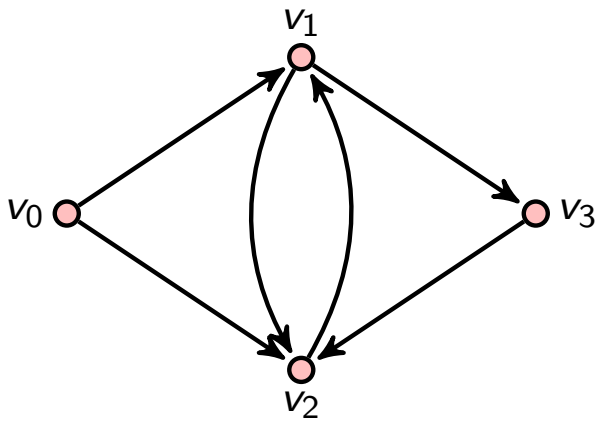
Les implémentations présentées dans cet exposé sont les suivantes.

- **Enregistrements d'adjacence**
- **Listes d'adjacence**
- **Tableaux d'adjacence**
- **Dictionnaires d'adjacence**
- **Matrices d'adjacence**

L'exposé s'attache à définir certaines fonctions de manipulation de ces différentes approches et compare leur efficacité.

Graphe exemple

Le graphe orienté suivant est adopté pour illustrer les codes.



$$V = \{v_0, v_1, v_2, v_3\}$$

$$E = \{(v_0, v_1), (v_0, v_2), (v_1, v_2), (v_1, v_3), (v_2, v_1), (v_3, v_2)\}$$

Enregistrement d'adjacence

Type

Un première solution définit un type `graph` comme un **enregistrement de listes** : une liste pour stocker les sommets, une liste pour stocker les arc.

Le type `graph` suivant permet une telle construction.

```
type 'a graph = {vertices: 'a list; edges: ('a * 'a) list;}
```

Le graphe exemple est alors directement défini par :

```
let gr = {vertices = [0; 1; 2; 3];
          edges =    [(0,1); (0,2); (1,2); (1,3); (2,1); (3,2)]}
```

Fonctions prédicats

Les fonctions suivantes testent la présence d'un sommet et d'un arc dans un graphe.

```
let vertex_exist gr v = List.mem v gr.vertices

let edge_exist gr v1 v2 = List.mem (v1, v2) gr.edges
```

La fonction **List.mem** permet le parcours des listes.

Voisins d'un sommet

La fonction suivante construit la **liste des sommets voisins d'un sommet**.

```
let vertex_neighbors gr v =
  let rec filter = function
    | [] -> []
    | (a,b)::q when a = v -> b::(filter q)
    | _::q -> filter q
  in filter gr.edges
```

Suppression d'un arc

La fonction suivante **supprime un arc**.

```
let edge_remove gr (v1,v2) =
  let rec filter = function
    | [] -> []
    | (a,b)::q when a = v1 && b = v2 -> filter acc q
    | h::q -> h::(filter q)
  in {vertices = gr.vertices; edges = (filter gr.edges)}
```

Cette solution parcourt la liste des arcs pour supprimer l'arc demandé, s'il est présent. Si l'arc n'est pas présent, la fonction parcourt malgré tout la liste.

Suppression d'un arc

La fonction `List.fold_left` permet d'obtenir (presque) le même résultat. Sa signature est rappelée ci-dessous.

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Elle reçoit une fonction, un objet de type `'a`, une liste et renvoie un objet du type `'a`. Cet objet peut être une liste ce qui permet de construire la fonction de **suppression d'un arc**.

```
let edge_remove gr (v1,v2) =
  let e = List.fold_left (fun lst (a,b) -> if a = v1 && b = v2 then lst
  ↪  else (a,b)::lst) [] gr.edges
  in {vertices = gr.vertices; edges = List.rev e}
```

L'appel à la fonction `List.rev`, non indispensable, permet de renvoyer des listes dont les éléments sont dans le même ordre que celui des listes initiales. Noter également que la fonction ne modifie pas le graphe sur place mais construit un nouveau graphe.

Suppression d'un sommet

Une fonction de **suppression d'un sommet** peut être construite en utilisant encore la fonction `List.fold_left`. Les deux listes des sommets et des arcs dovient être mises à jour.

```
let vertex_remove gr v =
  let new_edges = List.fold_left (fun lst (a,b) -> if a = v || b = v
  ↪  then lst else (a,b)::lst) [] gr.edges
  and new_vertices = List.fold_left (fun lst a -> if a = v then lst
  ↪  else a::lst) [] gr.vertices
  in {vertices = List.rev new_vertices; edges = List.rev new_edges}
```

Ajout d'un sommet

La fonction suivante **ajoute un sommet** s'il n'est pas déjà présent dans le graphe.

```
let vertex_add gr v =
  if not (vertex_exist gr v) then {vertices = v::(gr.vertices); edges
  ↪  = gr.edges}
  else gr
```

Si deux sommets sont déjà présents dans un graphe, l'**ajout d'un arc** entre ces sommets peut être réalisé par la fonction suivante.

```
let edge_add gr v1 v2 =
  if (vertex_exist gr v1) && (vertex_exist gr v2) && not (edge_exist
  ↪  gr v1 v2)
  then {vertices = gr.vertices; edges = (v1,v2)::gr.edges}
  else gr
```

Complexités

- `vertex_exist` : parcours de la liste des sommets en $O(|V|)$.
- `edge_exist` : parcours de la liste des arc en $O(|E|)$.
- `vertex_neighbors` : parcours de la liste des arc en $O(|E|)$.
- `edge_remove` : parcours de la liste des arcs en $O(|E|)$.
- `vertex_remove` : parcours successifs de la liste des arcs et celle des sommets en $O(|E| + |V|)$.
- `vertex_add` et `edge_add` : coûts $O(|V|)$ et $O(|V| + |E|)$ liés aux appels préliminaires aux fonctions prédicats.

Solution naïve

- Ces premières fonctions suffisent à montrer que cette première solution se révèlera vite peu efficace.
- Un graphe d'ordre n peut comporter jusqu'à $\binom{n}{2}$ arcs. Les coûts en $O(|E|)$ deviennent alors vite prohibitifs : $O(n^2)$.
- D'autres implémentations doivent permettre de **réduire significativement ces coûts des traitements**.

Liste d'adjacence

Type

Un graphe peut être défini comme une liste d'enregistrements.
Cette implémentation constitue une **liste d'adjacence**.

```
type 'a edge = {id: 'a; neighbors: 'a list}
type 'a graph = 'a edge list
```

Le graphe exemple est ainsi défini par :

```
let gr = [{id = 0; neighbors = [1;2]};
          {id = 1; neighbors = [2;3]};
          {id = 2; neighbors = [1]};
          {id = 3; neighbors = [2]}]
```

Fonctions de manipulation

La fonction suivante teste la **présence d'un sommet**.

```
let rec vertex_exist gr v = match gr with
  | [] -> false
  | h::q -> (h.id = v) || (vertex_exist q v)
```

La fonction suivante renvoie la **liste des voisins** d'un sommet.

```
let rec vertex_neighbors gr v = match gr with
  | [] -> []
  | h::q when h.id = v -> h.neighbors
  | _::q -> vertex_neighbors q v
```

Puis la fonction suivante teste la **présence d'un arc**.

```
let edge_exist gr v1 v2 =
  let v1_neighbors = vertex_neighbors gr v1 in
  List.mem v2 v1_neighbors;;
```

Complexités

D'autres fonctions de manipulation sont étudiées en td.

- `vertex_exist` : $O(|V|)$.
- `vertex_neighbors` : $O(|V|)$.
- `edge_exist` : $O(|V| + \delta)$ où δ est le dégré sortant d'un sommet.

Ces fonctions montrent une amélioration des complexités par rapport à la précédente implémentation en passant de $O(|E|)$ à $O(|V|)$.

Un inconvénient de cette solution est le parcours des listes avec un coût linéaire en leurs tailles. Toutefois, un avantage est la propriété de structure dynamique des listes.

Tableau d'adjacence

Type

L'accès aux éléments d'un tableau est de coût constant, propriété qui peut être exploitée pour accéder aux listes des voisins dans un graphe.

Le type suivant définit un graphe par un **tableau de listes**.

```
type 'a graph = 'a list array
```

Le graphe exemple est alors donné par :

```
let gr = [|[1; 2]; [2; 3]; [1]; [2]|]
```

Fonctions de manipulation

La construction des premières fonctions de manipulation est immédiate.

```
let vertex_exist gr v = v < Array.length gr
let vertex_neighbors gr v = gr.(v)
let edge_exist gr v1 v2 = List.mem v2 gr.(v1)
```

- Pour les **fonctions** `vertex_exist` et `vertex_neighbors`, les coûts d'accès constants aux éléments d'un tableau mènent à une complexité en $O(1)$.
- La complexité de la **fonction** `edge_exist` est directement liée à celle de l'appel à la fonction `List.mem`, à savoir $O(\delta)$ où δ désigne le degré sortant du premier sommet argument de la fonction.

Fonctions de manipulation

Construisons à présent les deux fonctions permettant l'**ajout** et la **suppression d'un arc** [1].

```
let edge_remove gr v1 v2 =
  gr.(v1) <- List.fold_left (fun lst v3 -> if v3 = v2 then lst else
  ↪  v3::lst) [] gr.(v1)

let edge_add gr v1 v2 =
  if not (edge_exist gr v1 v2) then gr.(v1) <- v2::gr.(v1)
```

- Ces deux fonctions sont de complexité $O(\delta)$ où δ est le degré sortant du premier sommet argument.
- Les **tableaux** étant des structures de données **mutables**, ces fonctions modifient les tableaux sur place.

1. On admet que cela est effectivement possible pour éviter de surcharger les codes.

Fonctions de manipulation

Ajouter un sommet ajoute une liste vide en fin de tableau.

```
let vertex_add gr =
  let n = Array.length gr in
  let new_gr = Array.make (n+1) [] in
  for i = 0 to n-1 do new_gr.(i) <- gr.(i) done;
  new_gr
```

Noter ici la nécessité de renvoyer un nouveau tableau.

Fonctions de manipulation

Supprimer un somme nécessite deux opérations :

- supprimer la liste associée au sommet à supprimer ;
- modifier les listes qui contiennent ce sommet de sorte que les étiquettes qui lui sont supérieures soient diminuées de 1, les autres restant inchangées.

```
let vertex_remove gr v =
  let rec lst_update lst u = match lst with
    | [] -> []
    | h::q when h=u -> lst_update q u
    | h::q when h>u -> (h-1)::(lst_update q u)
    | h::q when h<u -> h::(lst_update q u)
  in
  let n = Array.length gr in
  let new_gr = Array.make (n-1) [] in
  for i = 0 to v-1 do new_gr.(i) <- lst_update gr.(i) v done;
  for i = v+1 to n-1 do new_gr.(i-1) <- lst_update gr.(i) v done;
  new_gr
```

Complexités

- La **complexité temporelle** de `vertex_add` est en $O(|V|)$ puisqu'un tableau de taille proche du tableau initial est créé.
- Celle de `vertex_remove` est de en $O(|V|^2)$ en raison de $|V|$ appels à la fonction `lst_update` qui, au pire, peut traiter des listes de tailles proches de $|V|$.
- À ce coût temporel, il convient d'ajouter un **coût spatial** lié à la création d'un nouveau tableau : $O(|V|)$.
- Une telle solution ne présente donc d'intérêt que si le graphe est figé : ni ajout, ni suppression de sommets.
- Cette observation nuance la critique à l'égard des listes d'adjacence (listes de listes) dont le caractère dynamique peut être un avantage dès que le graphe évolue beaucoup. Mais une autre solution peut être envisagée.

Dictionnaire d'adjacence

Utilisation d'un dictionnaire

- L'implémentation par tableau de listes permet d'améliorer l'efficacité de certaines fonctions de manipulation par rapport à l'implémentation par liste de listes.
- Mais le caractère statique du tableau rend peu efficace l'ajout ou la suppression de sommets.
- Un **dictionnaire**, structure de données dynamique offrant un accès à ses données à coût constant, peut résoudre cette difficulté.
- Une implémentation de graphe par **dictionnaire de listes** est proposée en td.

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

Matrice d'adjacence

Définition

Une **matrice d'adjacence** permet de stoker un graphe pourvu que les sommets soient numérotés de 0 à $n-1$ en posant $n = |V|$.

Si M est la matrice d'adjacence d'un graphe orienté, pour tout couple d'entiers (i, j) pris dans $[\![0, n-1]\!]^2$:

$$M_{i,j} = \begin{cases} 0 & \text{s'il n'existe } \textbf{pas d'arc} \text{ entre les sommets } i \text{ et } j\,; \\ 1 & \text{s'il existe } \textbf{un arc} \text{ entre les sommets } i \text{ et } j. \end{cases}$$

Pour un graphe simple, la diagonale ne comporte que des zéros.

Type

Le type graph est un alias du type `int array array`.

```
type graph = int array array
```

La fonction suivante crée un graphe sans arcs.

```
let create_empty_graphe n = Array.make_matrix n n 0;;
```

Le graphe exemple est défini comme suit.

```
let (gr: graph) = create_empty_graphe 4;;
gr.(0).(1) <- 1;;
gr.(0).(2) <- 1;;
gr.(1).(2) <- 1;;
gr.(1).(3) <- 1;;
gr.(2).(1) <- 1;;
gr.(3).(2) <- 1;;
```

Fonctions de manipulation

La construction des premières fonctions de manipulation est là encore immédiate.

```
let vertex_exist gr v = v < Array.length gr
let edge_exist gr v1 v2 = gr.(v1).(v2) = 1
let edge_remove gr v1 v2 = gr.(v1).(v2) <- 0
let edge_add gr v1 v2 = gr.(v1).(v2) <- 1
```

La construction de la **liste des voisins** d'un sommet requiert le parcours d'une ligne de la matrice.

```
let rec vertex_neighbors gr v =
  let n = Array.length gr in
  let lst_neighbors = ref [] in
  for v' = 0 to n-1 do
    if edge_exist gr v v' then lst_neighbors := v'::!lst_neighbors
  done;
  !lst_neighbors
```

La construction d'autres fonctions de manipulation est laissée au soin du lecteur.

Complexités

- Les **complexités temporelles** de quatre premières fonctions sont $O(1)$.
- Construire la liste des voisins a un **coût linéaire** vis-à-vis du nombre de sommets : $O(|V|)$.
- Un inconvénient de cette implémentation est son **coût spatial** en $O(|V|^2)$, argument d'autant plus valable que le graphe est faiblement connecté, la matrice contenant beaucoup de zéros. Pour des graphes fortement connectés, $|E|$ est proche de $|V|^2$ et le coût en mémoire est tout à fait comparable à celui des autres solutions envisagées.
- Les chapitres suivants mettront en évidence d'autres avantages de cette représentation étroitement liés au calcul matriciel.