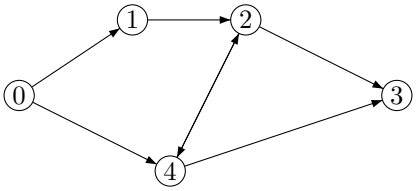
Exploration de graphes

1 Graphes et exploration

On appelle *graphe dirigé* un dessin de ce genre :



où les ronds sont appelés sommets et les flèches arcs ou arêtes. Étant donnés deux sommets s et v, on dit que v est un voisin de s si le graphe possède un arc allant de s à v (attention au sens !). Un chemin dans un tel graphe est une séquence continue d'arcs orientés dans le bon sens. Sur notre exemple, les voisins de 2 sont 3 et 4. On a une infinité de chemins allant de 1 à 3 : $1 \to 2 \to 3$, $1 \to 2 \to 4 \to 3$, $1 \to 2 \to 4 \to 2 \to 3$ .... En revanche, aucun chemin ne va de 2 à 0. Etant donné un sommet source s, on appelle sommet *accessible* tout sommet a tel qu'il existe un chemin de a à s. À ce chemin peut être associée une longueur, par exemple égale au nombre d'arcs de la séquence. La distance de s à a est la longueur minimale d'un chemin de s à a.
Pour représenter concrètement un graphe, on commencera par associer un entier positif à chaque sommet (de 0 à $n-1$ pour un graphe à n sommets), et on a ensuite deux variantes :

- La matrice d'adjacence : une matrice booléenne m (carrée et d'ordre n) telle que m.(i).(j) vaut true si et seulement si on a un arc du sommet i au sommet j.
- Les listes d'adjacence : un tableau v de listes d'entiers tel que v.(s) contient la liste des voisins du sommet s.

L'exemple admet donc les deux représentations :

$$\begin{pmatrix} f & V & f & f & V \\ f & f & V & f & f \\ f & f & f & V & V \\ f & f & f & f & f \\ f & f & V & V & f \end{pmatrix} \qquad \begin{bmatrix} 1;4 \\ 2 \\ 3;4 \\ \\ 2;3 \end{bmatrix}$$

On utilisera aujourd'hui la seconde représentation, par listes d'adjacence. Le type des graphes sera donc :

```
type graphe = int list array
```

▶ **Question 1** *Écrivez deux fonctions* taille *et* n_arcs *comptant respectivement le nombre de sommets et le nombre d'arcs d'un graphe.*

```
val taille : graphe -> int
val n_arcs : graphe -> int
```

▶ **Question 2** *Écrivez l'exemple du dessin comme un objet de type* graphe *en Caml*

```
val exemple : graphe
```

1.1 Principes de l'exploration

Lors de l'exploration d'un graphe, on peut diviser les sommets en trois catégories :

1. Les sommets déjà complètement explorés.
2. Les sommets pas encore découverts.
3. Les sommets déjà découverts mais pas encore traités.

L'algorithme consiste alors à prendre un sommet s de la troisième catégorie, découvrir ses voisins inconnus, puis déclarer qu'il n'y a plus rien à explorer depuis s. Le point crucial est la manière dont on organise les sommets nouvellement découverts. Il s'agit de la question : quand on prend un sommet de la troisième catégorie, comment le choisit-on ? C'est ceci qui va déterminer l'ordre dans lequel l'exploration va être menée.

On verra aujourd'hui deux organisations possibles pour ces sommets à traiter : la pile et la file.

- La pile fait réfèrence à une pile d'assiettes : à chaque fois qu'on range une nouvelle assiette on la place au sommet, et quand on veut en retirer une on la prend également au sommet. On parle de structure *Last In First Out*, ou LIFO, car la première assiette à sortir est toujours la dernière qui est entrée.
- La file fait référence à une file d'attente : les nouveaux arrivants se placent en queue, et les gens attendant en tête sont les premiers traités. Ici les objets sont traités dans l'ordre de leur arrivée, d'où l'appellation *First In First Out*, ou FIFO.

▶ **Question 3** *Regardez sur l'exemple les explorations générées par ces deux structures. Connaissez-vous les noms de ces deux modes d'exploration ?*

2 Accessibles d'un graphe

On veut, par une exploration en profondeur, connaître tous les points accessibles à partir d'une source donnée.
On utilisera la structure de pile contenue dans Caml, qu'on manipule avec les fonctions suivantes :

- Stack.create () crée une pile vide.
- Stack.is_empty p teste si la pile p est vide.
- Stack.push s p ajoute le sommet s a la pile p.
- Stack.pop p retire l'élement de tête de la pile p et le renvoie.

2.1 Préliminaires

Dans les questions suivantes on va utiliser les structures suivantes :
- Un graphe.
- Un tableau de booléens représentants les états accessibles.
- Une pile LIFO de sommets à explorer.

▶ **Question 4** *Ecrivez une fonction traite voisin qui prend un sommet et l'ajoute aux accessibles et aux sommets à traiter si nécessaire.*

```
val traite_voisin : bool array -> int Stack.t -> int -> unit
```

▶ **Question 5** *Ecrivez une fonction traite sommet qui prend un sommet et explore tous ses voisins.*

```
val traite_sommet : graphe -> bool array -> int Stack.t -> int -> unit
```

2.2 Accessibles

▶ **Question 6** *Ecrivez une fonction accessibles qui calcule les sommets accessibles d'un graphe a partir d'une source donnée.*

```
val accessibles : graphe -> int -> bool array
```

3 Distances

L'accessibilité ne nous suffit plus. On veut maintenant une information plus précise de distance par rapport à la source. Utiliser le parcours en profondeur est une mauvaise idée ici car ce parcours cherche à donner d'abord des chemins les plus longs possibles. Ainsi on peut découvrir tardivement un chemin plus court vers un sommet qu'on croyait dejà enterré depuis longtemps, ce qui demande de reprendre toute l'exploration depuis ce point.

▶ **Question 7** *Donnez un exemple.*

On procède alors par un parcours en largeur : on explorera d'abord tous les sommets à distance 1, puis ceux à distance 2, 3, et ainsi de suite jusqu'à epuisement des accessibles. On utilise pour cela la structure de file de Caml. On la manipule avec les mêmes fonctions que les piles, il faut simplement remplacer tous les Stack. par Queue.. Dans la prochaine question on va cette fois travailler avec les structures suivantes :
- Un graphe.
- Un tableau d'entiers optionnels representant les distances.
- Une file FIFO de sommets à explorer.

▶ **Question 8** *Modifiez les fonctions* traite_voisin *et* traite_sommet *pour qu'elles tiennent compte des distances.*

```
val traite_voisinD : int option array -> int Queue.t -> int -> int -> unit
val traite_sommetD : graphe -> int option array -> int Queue.t
  -> int -> int -> unit
```

▶ **Question 9** *Concluez en donnant une fonction calculant les distances dans un graphe à partir d'une source donnée.*

```
val distances : graphe -> int -> int option array
```

4 Graphes pondérés

On va maintenant explorer des mondes moins gentils, où les arêtes n'auront pas toutes la même longueur. On utilise toujours les listes d'adjacence, mais elles contiendront cette fois des couples d'entiers : chaque voisin de la liste sera accompagné d'une distance (un entier positif). Le nouveau type est alors :

```
type graphP = (int * int) list array
```

Dans ce cadre, le parcours en profondeur est toujours aussi mauvais, mais même le parcours en largeur n'est plus efficace.

▶ **Question 10** *Donnez un exemple.*

▶ **Question 11** *Pouvez-vous imaginer une structure pour l'ensemble* a_faire *qui assure que l'algorithme de calcul des distances n'aura jamais à explorer deux fois un sommet ?*

Vous allez implémenter cette structure, pour cela on a besoin de définir un type t et les quatres fonctions de base.

▶ **Question 12** *Écrivez les quatres fonctions*

```
create : unit -> t
is_empty : t -> bool
push : int -> int -> t -> unit
pop : t -> int * int
```

▶ **Question 13** *Modifiez les fonctions* traite_voisin *et* traite_sommet.

```
val traite_voisinP : int option array -> t -> int -> int -> unit
val traite_sommetP : graphP -> int option array -> t
  -> int -> int -> unit
```

▶ **Question 14** *Concluez en donnant une fonction calculant les distances dans un graphe à partir d'une source donnée.*

```
val distances : graphP -> int -> int option array
```

Exploration de graphes

Un corrigé

▶ Question 1

```
let taille (g:graphe) = Array.length g

let n_arcs (g:graphe) =
  Array.fold_left (fun t s -> List.length s + t) 0 g
```

▶ Question 2

```
let (exemple:graphe) =
  [| [1;4] ; [2] ; [3;4] ; [] ; [2;3] |]
```

▶ Question 4

```
let traite_voisin acc afaire s =
  if acc.(s) then ()
  else (acc.(s) <- true ; Stack.push s afaire)
```

▶ Question 5

```
let traite_sommet (g:graphe) acc afaire s =
  List.iter (fun v -> traite_voisin acc afaire v) g.(s)
```

▶ Question 6

```
let accessibles g s =
  let acc = Array.make (taille g) false in
  let afaire = Stack.create () in
    Stack.push s afaire;
    acc.(s) <- true;
    while not (Stack.is_empty afaire)
    do
      let a = Stack.pop afaire in
        traite_sommet g acc afaire a
    done;
    acc
```

▶ Question 8

```
let traite_voisinD distances afaire s d =
  match distances.(s) with
    | Some (x) -> ()
    | None ->
      distances.(s) <- Some d ;
      Queue.push s afaire

let traite_sommetD (g:graphe) distances afaire s d =
  List.iter (fun v -> traite_voisinD distances afaire v d) g.(s)
```

▶ Question 9

```
let distances g s =
  let distances = Array.make (taille g) None in
  let afaire = Queue.create () in
    Queue.push s afaire;
    distances.(s) <- Some 0;
    while not (Queue.is_empty afaire)
    do
      let a = Queue.pop afaire in
        traite_sommetD g distances afaire a
          (let Some x = distances.(a) in x + 1)
    done;
    distances
```

▶ Question 12

```
type t = (int * int) list ref

let create () = ref []

let is_empty a = match !a with
  | [] -> true | _ -> false

let push a p (f:t) =
  let rec rma = function
    | [] -> []
    | (b,q) :: s ->
if b = a then s
else (b,q) :: rma s
  in
  let rec aux = function
    | [] -> [a,p]
    | (b,q) :: s ->
if p < q then (a,p) :: (b,q) :: s
else (b,q) :: aux s
  in f := aux (rma !f)

let pop (f:t) = match !f with
  | (a,p) :: s -> f := s ; a,p
```

▶ Question 13

```
let traite_voisinP distances afaire s d =
  match distances.(s) with
    | Some (x) ->
      if x < d then ()
      else
        ( distances.(s) <- Some d;
          push s d afaire)
    | None ->
      distances.(s) <- Some d ;
      push s d afaire

let traite_sommetP (g:graphP) distances afaire s d =
  List.iter (fun (v,p) ->
    traite_voisinP distances afaire v (d+p)) g.(s)
```

▶ Question 14

```
let distances g s =
  let distances = Array.make (Array.length g) None in
  let afaire = create () in
    push s 0 afaire;
    distances.(s) <- Some 0;
    while not (is_empty afaire)
    do
      let a,p = pop afaire in
        traite_sommetP g distances afaire a p
    done;
    distances
```