1 Graphes fournis en exemple

Voici les graphes que vous trouverez dans le fichier `ex_graphe.ml`, sur la page web du cours, donnés sous forme de listes d'adjacence : tableau `v` à n éléments, `v.(i)` est une liste contenant les sommets j tels qu'il existe un arc (i, j) dans le graphe (dans le cas orienté) ou une arête $\{i, j\}$ (cas non orienté). Ce sont les graphes `g1`, `g2` et `g3` du fichier.

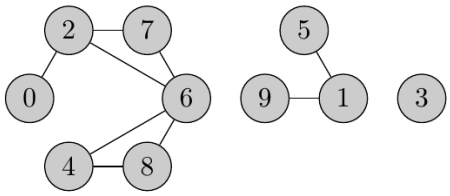

FIGURE 1: Graphe non orienté

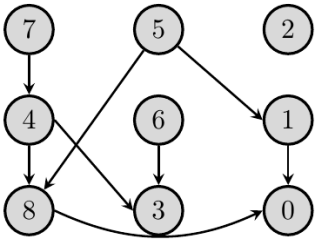

FIGURE 2: Un graphe orienté sans circuit

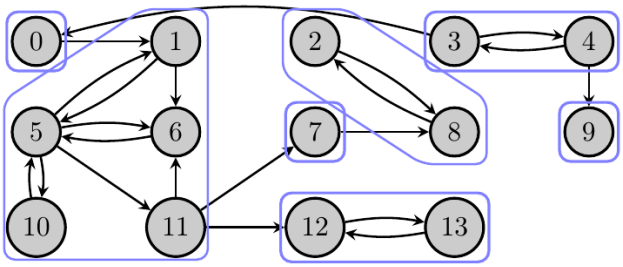

FIGURE 3: Un graphe orienté et ses composantes fortement connexes

2 Parcours en largeur

Rappel : la bibliothèque `Queue` fournit une structure de file ; les opérations sont les suivantes :

```
Queue.create : unit -> 'a Queue.t
Queue.add : 'a -> 'a Queue.t -> unit
Queue.pop : 'a Queue.t -> 'a
Queue.is_empty : 'a Queue.t -> bool
```

Remarque : `Queue` est le nom du module ; `Queue.t` est le nom du type défini dans le module.

Exercice 1 (Parcours en largeur) : Implémenter un parcours en largeur comme une fonction `parcours_largeur : int list array -> int -> int array * int array` prenant en entrée un graphe donné par listes d'adjacence, et un sommet source s, et calculant le tableau des distances depuis s, ainsi que le tableau des prédécesseurs. On conviendra qu'un sommet non accessible est à distance -1, et que le prédécesseur de s est lui-même.

```
# parcours_largeur g1 0 ;;
- : int array * int array =
([|0; -1; 1; -1; 3; -1; 2; 2; 3; -1|], [|0; -1; 0; -1; 6; -1; 2; 2; 6; -1|])
# parcours_largeur g2 7 ;;
- : int array * int array =
([|3; -1; -1; 2; 1; -1; -1; 0; 2|], [|8; -1; -1; 4; 7; -1; -1; 7; 4|])
# parcours_largeur g3 0 ;;
- : int array * int array =
([|0; 1; 6; -1; -1; 2; 2; 4; 5; -1; 3; 3; 4; 5|],
 [|0; 0; 8; -1; -1; 1; 1; 11; 7; -1; 5; 5; 11; 12|])
```

Exercice 2 (Reconstruction d'un chemin à l'aide du tableau des prédécesseurs) : Déduire de la fonction précédente une fonction `pcc : int list array -> int -> int -> int list` prenant en entrée un graphe donné par une représentation creuse, un sommet source s, et un sommet cible x, et renvoyant sous forme de liste un plus court chemin de s vers x s'il existe.

```
# pcc g1 0 4 ;;
- : int list = [0; 2; 6; 4]
# pcc g2 0 1 ;;
Exception: Failure "x non accessible !"
# pcc g3 0 13 ;;
- : int list = [0; 1; 5; 11; 12; 13]
```

3 Parcours en profondeur "complet" et variantes

On travaille ici uniquement avec des graphes sous la forme `graphe_creux`. On utilisera le type :

```
type couleur = Blanc | Gris | Noir ;;
```

La convention dans un parcours en profondeur est d'attribuer les couleurs suivantes aux sommets :

- `Blanc` pour un sommet non encore découvert ;
- `Gris` pour un sommet dont le parcours en profondeur est en cours ;
- `Noir` pour un sommet dont le parcours en profondeur est terminé.

Exercice 3 : Écrire une fonction `creer_liste : int -> int list` prenant en argument un entier n, et renvoyant la liste des entiers de 0 à $n-1$, avec une complexité en $\mathcal{O}(n)$.

Exercice 4 (Parcours en profondeur générique) : Implémenter une fonction `parcours_prof_gen : int list array -> int list -> bool * int list list * int list` prenant en entrée un graphe et une liste de ses sommets, et effectue un parcours en profondeur complet du graphe, en lançant chaque parcours élémentaire dans l'ordre fourni par la liste des sommets. Votre fonction devra renvoyer un triplet (b, l, p) tel que :

- b est un booléen, indiquant si à un moment du parcours, un arc d'un sommet `Gris` vers un autre sommet `Gris` a été découvert ;
- l est une liste de listes, contenant une énumération des sommets pour chaque parcours élémentaire effectué (dans l'ordre de fin de parcours) ;
- p est la concaténation des listes précédentes : les sommets sont ordonnés par date de fin de parcours décroissante.

```
# parcours_prof_gen g1 (creer_liste 10) ;;
- : bool * int list list * int list =
(true, [[3]; [1; 9; 5]; [0; 2; 6; 7; 4; 8]], [3; 1; 9; 5; 0; 2; 6; 7; 4; 8])
# parcours_prof_gen g2 (creer_liste 9) ;;
- : bool * int list list * int list =
(false, [[7]; [6]; [5]; [4; 8]; [3]; [2]; [1]; [0]], [7; 6; 5; 4; 8; 3; 2; 1; 0])
# parcours_prof_gen g3 (creer_liste 14) ;;
- : bool * int list list * int list =
(true, [[3; 4; 9]; [0; 1; 5; 11; 12; 13; 7; 8; 2; 10; 6]],
 [3; 4; 9; 0; 1; 5; 11; 12; 13; 7; 8; 2; 10; 6])
```

Le but de la suite du sujet est d'exploiter le parcours en profondeur pour soutirer des informations sur le graphe.

Exercice 5 (Calcul d'un tri topologique) : Utiliser l'algorithme de parcours en profondeur pour implémenter une fonction `tri_topologique : int list array -> int list` calculant un tri topologique d'un graphe orienté sans circuit. On renverra une erreur si on détecte l'existence d'un circuit.

```
# tri_topologique g2 ;;
- : int list = [7; 6; 5; 4; 8; 3; 2; 1; 0]
# tri_topologique g3 ;;
Exception: Failure "il y a un cycle".
```

Exercice 6 (Calcul des composantes connexes) : Utiliser l'algorithme de parcours en profondeur pour implémenter une fonction `composantes_connexes : int list array -> int list list` renvoyant les composantes connexes d'un graphe non orienté. On renverra la liste des composantes (elles-mêmes sous forme de listes).

```
# composantes_connexes g1 ;;
- : int list list = [[3]; [1; 9; 5]; [0; 2; 6; 7; 4; 8]]
```

Exercice 7 (Graphe transposé) : Écrire une fonction `graphe_transpose : int list array -> int list array` renvoyant le graphe transposé du graphe passé en entrée.

```
# graphe_transpose g1 ;;
- : int list array = [|[2]; [9; 5]; [7; 6; 0]; []; [8; 6]; [1]; [8; 7; 4; 2];
[6; 2]; [6; 4]; [1]|]
# graphe_transpose g2 ;;
- : int list array = [|[8; 1]; [5]; []; [6; 4]; [7]; []; []; []; [5; 4]|]
# graphe_transpose g3 ;;
- : int list array =
[|[3]; [5; 0]; [8]; [4]; [3]; [10; 6; 1]; [11; 5; 1]; [11]; [7; 2]; [4]; [5];
[5]; [13; 11]; [12]|]
```

Exercice 8 (Calcul des composantes fortement connexes) : Implémenter l'algorithme de Kosaraju par une fonction `kosaraju : int list array -> int list list` calculant les composantes fortement connexes d'un graphe orienté.

```
# kosaraju g1 ;;
- : int list list = [[0; 2; 7; 6; 8; 4]; [1; 5; 9]; [3]]
# kosaraju g2 ;;
- : int list list = [[0]; [1]; [2]; [3]; [8]; [4]; [5]; [6]; [7]]
# kosaraju g3 ;;
- : int list list = [[8; 2]; [7]; [12; 13]; [1; 5; 6; 11; 10]; [0]; [9]; [3; 4]]
```

4 Application à 2-SAT

On a vu en cours une application de l'algorithme précédent au problème 2-SAT : satisfiabilité de formules logiques, conjonction de clauses de 2 littéraux (à variables distinctes). On choisit une implémentation simple de telles formules :

```
type litteral = X of int | Xb of int ;;
type clause2 = litteral * litteral ;;
type formule2sat = clause2 list ;;
```

Dans cette représentation, un littéral est soit de la forme `X p` (ce qui symbolise la variable x_p), soit sous la forme `Xb p` (qui symbolise $\neg x_p = \bar{x_p}$). Une 2-clause (disjonction de deux littéraux) est représentée par un couple de littéraux, et une instance de 2-SAT (conjonction de 2-clauses) par une liste de tels couples.

Exercice 9 (Graphe associé à une instance de 2-SAT) : Écrire une fonction `construit_graphe : int -> formule2sat -> int list array`, prenant un entier n et une instance de 2-SAT dont les variables ont des numéros dans $[\![0, n-1]\!]$, et renvoyant le graphe orienté à $2n$ sommets associé.

On rappelle que celui-ci est obtenu en remplaçant chaque clause $\ell \vee \ell'$ par les deux implications $\neg\ell \to \ell'$ et $\neg\ell' \to \ell$. À chaque implication est associée un arc du graphe. On pourra convenir que les sommets numérotés de 0 à $n-1$ sont associés aux variables, et ceux de n à $2n-1$ à leurs négations. Vous êtes libres d'écrire toute fonction nécessaire pour vous faciliter la vie. On supposera que :

- toutes les clauses contiennent des littéraux à variables distinctes ;
- il n'y a pas deux clauses égales.

```
# construit_graphe 3 fsat3 ;;
- : int list array = [|[1]; [5]; [0; 4]; [5; 2]; [2; 3]; [0; 1]|]
# construit_graphe 3 fnsat3 ;;
- : int list array = [|[5; 2]; [2; 0]; [3]; [4; 1]; [0]; [4; 3]|]
```

Exercice 10 (Résolution de 2-SAT) : On rappelle que la formule 2-SAT est satisfiable si et seulement si dans le graphe associé, on ne trouve pas une variable et sa négation dans la même composante fortement connexe. En déduire une fonction `est_satisfiable : int -> formule2sat -> bool` prenant en entrée la même chose que précédemment, et renvoyant un booléen indiquant si la formule est satisfiable.

```
# est_satisfiable 3 fsat3 ;;
- : bool = true
# est_satisfiable 3 fnsat3 ;;
- : bool = false
# est_satisfiable 5 fsat5 ;;
- : bool = true
# est_satisfiable 5 fnsat5 ;;
- : bool = false
```

Exercice 11 (Extraction d'un certificat) : Reprendre la question précédente pour produire un **certificat** dans le cas où la formule est satisfiable, c'est-à-dire une valuation ν satisfiant la formule. On rappelle que celle-ci s'obtient en remontant les composantes connexes à l'envers dans un tri topologique du graphe des composantes fortement connexes, en associant aux littéraux apparaissant dans une même composante la valeur "vraie" sils n'ont pas déjà de valeur de vérité.

Remarques :

- on renverra la valuation sous la forme d'un tableau `v` de taille n, avec `v.(i)` valant $\nu(x_i) \in \{0, 1\}$;
- l'algorithme de Kosaraju traitait le graphe transposé, et ses composantes fortement connexes sont renvoyées dans l'ordre topologique **du graphe des composantes fortement connexes du graphe transposé à l'envers**. C'est dans cet ordre-là qu'il faut les traiter !

```
# certificat 3 fsat3 ;;
- : int array = [|1; 1; 0|]
# certificat 5 fsat5 ;;
- : int array = [|1; 1; 0; 1; 1|]
# certificat 3 fnsat3 ;;
Exception: Failure "non satisfiable".
```