TP 3 : Graphes non pondérés

1 Graphes fournis en exemple

Voici les graphes que vous trouverez dans le fichier joint, 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 joint.

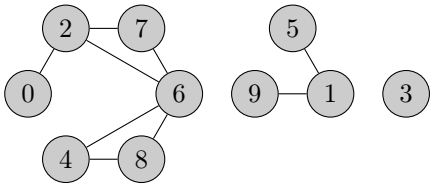

FIGURE 1: Graphe non orienté

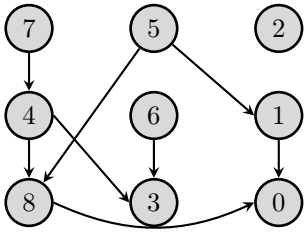

FIGURE 2: Un graphe orienté sans circuit

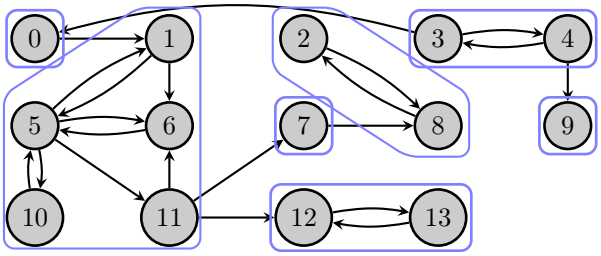

FIGURE 3: Un graphe orienté et ses composantes fortement connexes

2 Parcours en largeur

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 g s`, prenant en entrée un graphe donné par listes d'adjacence, et un sommet source `s`, et calculant le tableau des distances depuis `s`, qu'on renverra, ainsi que le tableau des prédécesseurs. On convient qu'un sommet non accessible est à distance -1, et que le prédecesseur de `s` est `s` lui-même.

```
# parcours_largeur g1 0 ;;
- : int array * int array =
([|0; -1; 1; -1; 3; -1; 2; 2; 3; -1|], [|-1; -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; -1; 4|])
# parcours_largeur g3 0 ;;
- : int array * int array =
([|0; 1; 6; -1; -1; 2; 2; 4; 5; -1; 3; 3; 4; 5|], [|-1; 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 prédecesseurs*. Déduire de la fonction précédente une fonction `pcc g s x` prenant en entrée un graphe donné par représentation creuse, un sommet source `s`, et un sommet `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 ;;
Uncaught 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 ;

— noire pour un sommet dont le parcours en profondeur est terminé.

La fonction `parcours_prof_gen g liste_som`, **gentiment fournie**, effectue un parcours en profondeur complet du graphe, les sommets dans la boucle principale (remplacée par un parcours récursif ici) étant considérés par ordre d'apparition dans `liste_som`. La fonction renvoie un triplet `(b, q, p)` où :

— `b` est un booléen, indiquant si à un moment dans le parcours, un arc d'un sommet gris vers un autre a été découvert ;

— `q` 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.

Exercice 3. Écrire une fonction `creer_liste n` renvoyant la liste des entiers de 0 à $n-1$, avec une complexité $O(n)$.

Voici les résultats de `parcours_prof_gen` sur les graphes pris en exemple :

```
# 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 est d'exploiter le parcours en profondeur pour soutirer des informations sur le graphe.

Exercice 4. *Calcul d'un tri topologique*. Utiliser l'algorithme de parcours en profondeur pour calculer 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 5. *Calcul des composantes connexes (facile !)*. Utiliser l'algorithme de parcours en profondeur pour obtenir 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 6. *Graphe transposé.* Écrire une fonction `graphe_transpose g` renvoyant le graphe transposé du graphe passé en paramètre. On créera un nouveau graphe.

```
# 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 7. *Calcul des composantes fortement connexes.* L'algorithme de Kosaraju pour le calcul des composantes fortement connexes d'un graphe orienté est rappelé. L'implémenter.

Algorithme 1 : Algorithme de Kosaraju pour le calcul de composantes fortement connexes

Entrée : Un graphe G orienté
Sortie : Ses composantes fortement connexes
Effectuer un parcours en profondeur de G, en stockant les sommets dans l'ordre de leur date de fin de parcours d_f dans une liste L;
Effectuer un parcours en profondeur de ${}^tG$, mais dans la boucle principale du parcours, prendre les sommets par date d_f décroissante. Chaque parcours élémentaire fournit une composante fortement connexe.

```
# 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 deux 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 2 littéraux) est représentée comme un couple de littéraux, et une instance de 2-SAT (conjonction de 2-clauses) comme une liste de tels couples. Attention, cette section est moins guidée !

Exercice 8. *Graphe associé à une instance de 2-SAT.* Écrire une fonction `construit_graphe n f` prenant en entrée un entier n, 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 $\neg\ell \Rightarrow \ell'$ et $\neg\ell' \Rightarrow \ell$. 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 9. *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 n f` prenant en entrée la même chose que précédemment, et renvoyant un booléen indiquant si `f` 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 10. *Extraction d'un certificat*. Reprendre la question précédente pour produire un *certificat* dans le cas où `f` est satisfiable, c'est-à-dire une distribution de vérité d 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 » s'ils n'ont pas déja de valeur de vérité. Remarques :

— on renverra la distribution de vérité sous la forme d'un tableau `d` de taille n, avec `d.(i)` valant $d(x_i) \in \{0, 1\}$;

— 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".
```