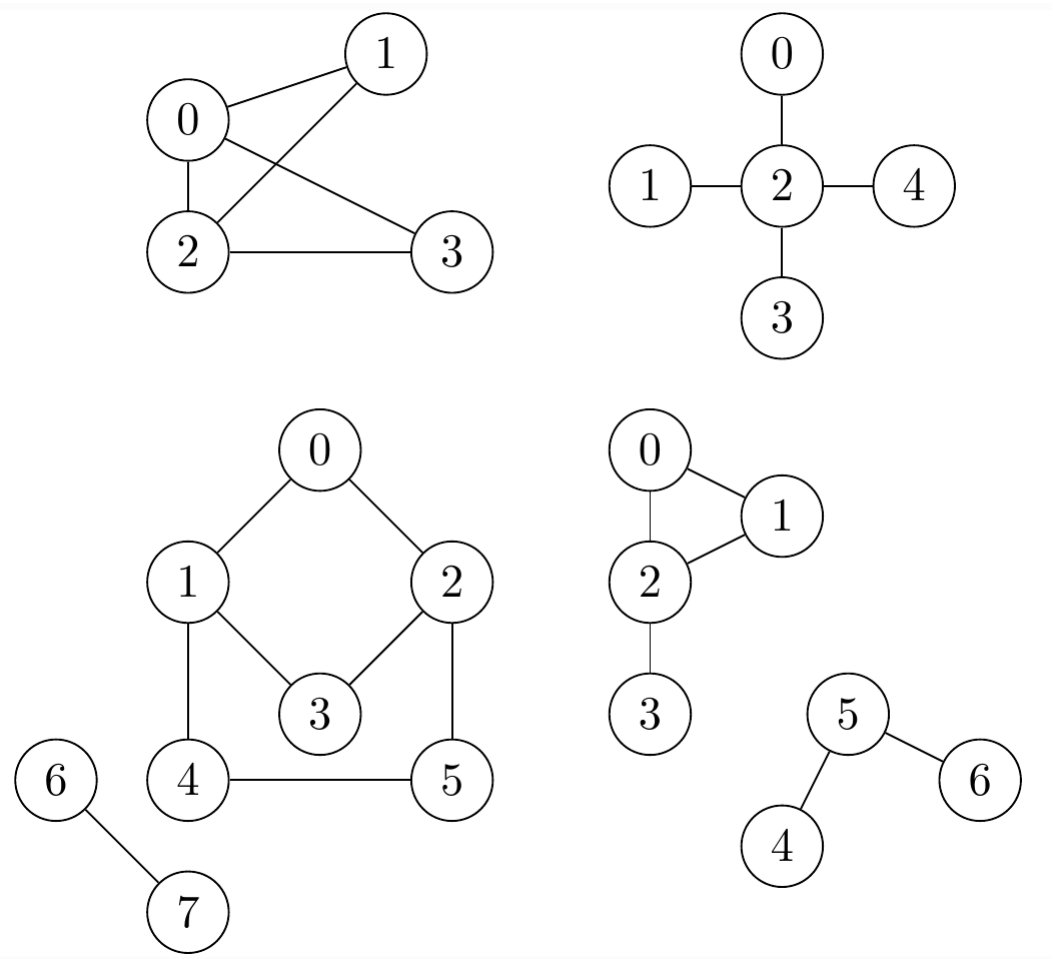
TP n°24 – Parcours de graphes et applications

Dans ce TP, on implémente les différents parcours de graphes et quelques-unes de leurs applications : recherche de composantes connexes, tri topologique d'un graphe et détection de cycles dans un graphe.

1 Parcours itératifs en Ocaml

Dans un premier temps, on va implémenter les versions itératives des parcours en largeur et en profondeur en OCaml. Les graphes seront représentés par leurs listes d'adjacence.

On pourra tester les parcours sur les exemples fournis suivants :



1.1 Piles et files

L'avantage du Ocaml ici est que la bibliothèque standard nous donne une implémentation de pile et de file.

On utilisera donc les modules `Stack` et `Queue`. On rappelle les fonctions utiles ci-dessous :

Stack.create : unit -> 'a Stack.t crée une pile vide Stack.is_empty : 'a Stack.t -> bool teste si la pile est vide Stack.push : 'a -> 'a Stack.t -> unit ajoute une valeur à la pile Stack.pop : 'a Stack.t -> 'a retire une valeur de la pile Stack.top : 'a Stack.t -> 'a regarde le premier élément de la pile (peek n'existe pas)

Queue.create : unit -> 'a Queue.t crée une pile vide Queue.is_empty : 'a Queue.t -> bool teste si la file est vide Queue.push : 'a -> 'a Queue.t -> unit ajoute une valeur à la file Queue.pop : 'a Queue.t -> 'a retire une valeur de la file Queue.peek : 'a Queue.t -> 'a regarde le premier élément de la file

L'exception Empty peut être levée si la pile/file est vide et qu'on essaie d'y trouver un élément.

1.2 Parcours en largeur

- **Q1.** Implémenter le parcours en largeur à partir d'un sommet, de manière itérative en OCaml. Pour débugguer, on pourra afficher les sommets dans l'ordre de leur visite. La signature sera `parcourslargeur1 : int list array -> int -> unit`.
- **Q2.** Implémenter maintenant une version complète du parcours. Il est possible que vous deviez modifier la fonction de la Q1.

`parcourslargeurcomplet1 : int list array -> unit`

- **Q3.** Comment obtenir en sortie la liste de tous les sommets dans l'ordre de leur visite ? Copier et modifier votre fonction précédente. On aura le droit d'utiliser une référence de liste.
  `parcourslargeur2 : int list array -> int -> int list`.
- **Q4.** Copier et modifier également le parcours en largeur complet pour qu'il renvoie la liste des listes des sommets parcourus à chaque parcours lancé (à chaque appel de `parcourslargeur2`).
  `parcourslargeurcomplet2 : int list array -> int list list`.

1.3 Composantes connexes d'un graphe non-orienté

On veut modifier `parcourslargeurcomplet1` pour obtenir les composantes connexes, sous la forme d'un tableau `composantes` où `composantes.(s)` est l'identifiant (un entier unique pour chaque composante) de la composante connexe à laquelle appartient `s`.

On va donc définir un tableau `composantes` initialisé à -1 et une référence `id` que l'on va incrémenter à chaque découverte d'une nouvelle composante. Lorsque l'on visite un sommet, il suffit de lui attribuer l'identifiant de la composante en cours : `composantes.(s) <- !id`.

- **Q5.** Dans le code que vous avez déjà écrit, à quel endroit faut-il incrémenter la référence `id` pour que le parcours permette de calculer les composantes connexes ?
- **Q6.** Écrire une fonction `composantes_connexes : int list array -> int array` qui renvoie un tableau `composantes` qui associe à chaque sommet l'identifiant de sa composante connexe, comme proposé ci-dessus.
- **Q7.** Écrire une fonction `est_connexe : int list array -> bool` qui vérifie si un graphe est connexe.
  On cherchera une fonction plus efficace que l'appel à la fonction précédente suivie d'un test.

Pour finir on voudrait obtenir les sous-graphes (donc les listes d'adjacences) correspondant aux composantes connexes.

Cette partie est un peu plus difficile, je conseille de passer et de revenir si vous finissez le reste.

- **Q8.** Écrire une fonction qui étant donné un graphe G et une liste l de sommets de G, renvoie la représentation sous forme de liste d'adjacence du sous-graphe induit par l.
  Les sommets de l devront être renumérotés, vous pouvez choisir de le faire dans l'ordre de l ($[4;6;1;0]$ devient $[0;1;2;3]$) ou en conservant l'ordre originel et en diminuant les numéros ($[4;6;1;0]$ devient $[2;3;1;0]$).
  La signature sera `sousgraphe : int list array -> int list -> int list array`.
- **Q9.** Écrire une fonction `liste_composantes_connexes : int list array -> (int list array) list` qui renvoie la liste des sous-graphes correspondant aux composantes connexes.

1.4 Parcours en profondeur

- **Q10.** Implémenter le parcours en profondeur itératif d'un graphe à partir d'un sommet, pour débugger on peut faire afficher les sommets dans l'ordre de parcours. La signature sera `parcoursprofondeur : int list array -> int -> unit`.
- **Q11.** Implémenter le parcours en profondeur complet d'un graphe. La signature sera `parcoursprofondeurcomplet : int list array -> unit`.
- **Q12.** Modifier les fonctions précédentes pour calculer le tableau π des prédécesseurs. ($\pi[i]$ doit contenir le sommet depuis lequel on a découvert le sommet i)

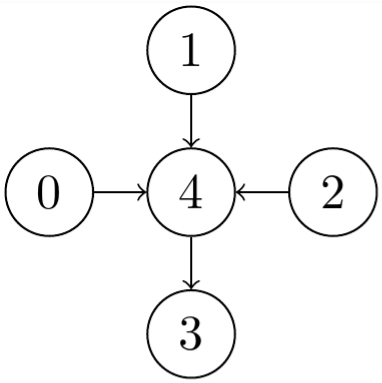
1.5 Détection de cycles

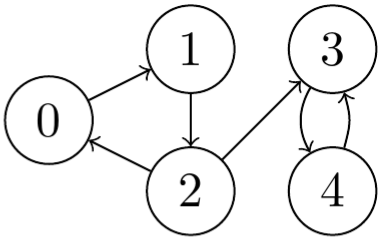
- **Q13.** Copier et modifier les fonctions précédentes pour pouvoir détecter un cycle dans un graphe **non-orienté**.

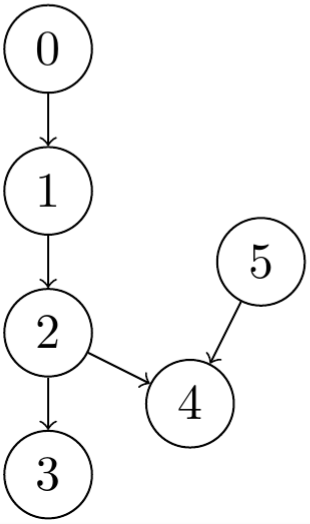
2 Parcours profondeur récursif en C

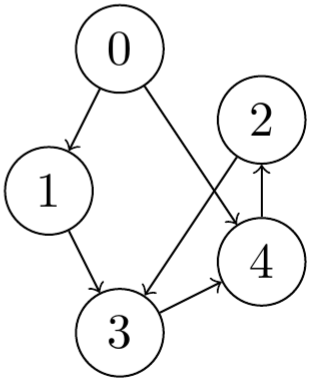
Dans cette partie, on veut implémenter le parcours profondeur récursif en C, avec ses options. Ici on considèrera les graphes représentés par leur matrice d'adjacence.

On pourra tester sur les exemples suivants :









2.1 Parcours, dates de début et dates de fin

- **Q14.** Écrire la fonction `void parcoursprofondeur(int** adj, int n, int s)` qui réalise le parcours profondeur récursif à partir du sommet s. Pour le moment votre parcours doit simplement afficher les sommets dans l'ordre de leur visite.

  On pourra modéliser les couleurs blanc, gris et noir par les entiers 0, 1 et 2.
- **Q15.** Écrire la version complete du parcours `void parcoursprofondeurcomplet(int** adj)`. Il faut modifier la signature de la fonction de la question précédente pour que ça marche.
- **Q16.** Rajouter la gestion des dates de début et de fin des sommets à vos fonctions. Vous pouvez modifier les signatures des fonctions déjà écrites.

2.2 Graphes orientés acycliques et tri topologique

- **Q17.** Copier et modifier les fonctions précédentes pour obtenir un algorithme qui détecte si un graphe **orienté** est acyclique.