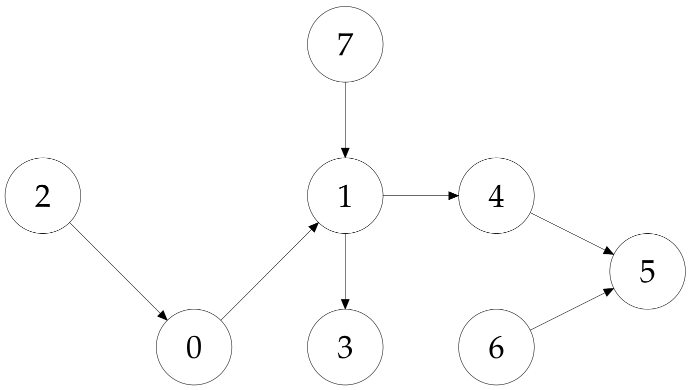
Tri topologique et composantes fortement connexes

Dans ce sujet nous apprendrons à ordonner les sommets d'un graphe orienté de manière utile pour résoudre certains problèmes. On présentera en particulier l'algorithme de Kosaraju qui permet de calculer les composantes fortement connexes d'un graphe.

1 Tri topologique d'un graphe orienté acyclique

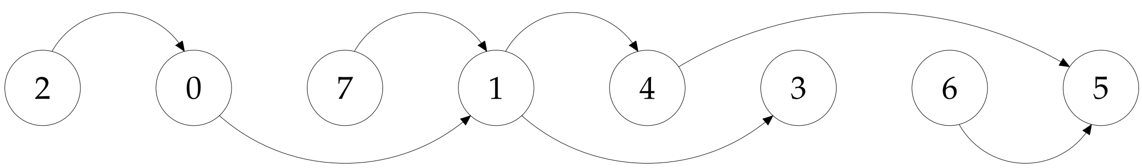
Vous étudiez un livre de mathématiques comportant 8 chapitres numérotés de 0 à 7. Malheureusement les chapitres n'ont pas été numérotés dans un ordre logique. Pour comprendre le chapitre 0 il faut avoir lu le chapitre 2. Pour comprendre le chapitre 1, il faut avoir lu le chapitre 0 et le chapitre 7. Pour comprendre le chapitre 3 et le chapitre 4 il faut avoir lu le chapitre 1. Enfin pour comprendre le chapitre 5 il faut avoir lu les chapitres 4 et 6. Le but est de trouver un ordre de lecture correct des chapitres.

Un tel problème est souvent représenté à l'aide d'un **graphe de dépendance** :



Le graphe est un graphe orienté acyclique car s'il contenait un cycle le problème n'aurait pas de solution (problème de dépendances cycliques). Les graphes orientés acycliques sont souvent appelés **DAG** en informatique pour *directed acyclic graph*.

Un tri topologique d'un DAG consiste à ordonner totalement les sommets de $G = (S, A)$ de telle sorte que pour toute arc $xy \in A$, $x \leq y$ pour l'ordre choisi. Un exemple de tri topologique dans l'exemple précédent est 2, 0, 7, 1, 4, 3, 6, 5 mais ce n'est pas l'unique solution. Ce tri peut également être représenté graphiquement :



On constate alors que les arcs sont tous orientés vers l'avant.

En OCAML, on représentera les graphes par listes d'adjacence avec le type suivant

```
type graphe = {taille : int ; succ : int list array};;
```

On notera n le nombre de sommets et on les numerotera de 0 à $n-1$.

Pour obtenir un tri topologique d'un graphe on peut utiliser l'algorithme suivant :

1. Définir une pile vide p.
2. Pour chaque sommet k de 0 à $n-1$, déclencher un parcours en profondeur depuis le sommet k pour explorer les sommets qui n'ont jamais été visités (dans ce parcours ou les précédents).
3. Dans le parcours en profondeur, ajouter chaque sommet rencontré dans la pile p. On procèdera à cet ajout seulement après que le parcours de tous les successeurs est terminé. Cela correspond donc à un parcours en profondeur postfixe.
4. La pile contient alors les sommets dans un ordre topologique.

Question 1

Compléter le programme suivant calculant un ordre topologique. La pile sera codée à l'aide d'une liste.

```
let tri_topo g =
  let n = g.taille in
  let visite = Array.make n false in
  let pile = ref [] in
  (* parcours en profondeur depuis s. On ajoutera
     chaque sommet dans la pile une fois tous ses successeurs
     visités (ordre postfixe)
   *)
  let rec parcours_profondeur s =
     (* COMPLETER ICI *)
  in
  for k = 0 to n - 1 do
    parcours_profondeur k
  done;
  !pile
;;
```

On vérifiera à l'aide du graphe donné en exemple que l'on obtient bien en sortie un tri topologique.

2 Transposée d'un graphe

Soit $G = (S, A)$ un graphe orienté. On appelle **transposée** du graphe G le graphe $\overline{G} = (S, \overline{A})$ tel que pour tous sommets x, y, on a $xy \in A$ si et seulement si $yx \in \overline{A}$. La transposition interviendra dans l'algorithme de Kosaraju.

> **Question 2**
>
> Écrire une fonction `transpose : graphe -> graphe` calculant la transposée d'un graphe.

3 Algorithme de Kosaraju

L'algorithme de tri topologique peut aussi être utilisé sur un graphe orienté quelqueconque. Dans ce cas, on obtient un ordre des sommets parfois appelé ordre **pré-topologique**. L'ordre pré-topologique vérifie la propriété suivante : pour tous sommets x, y, s'il existe un chemin de y à x et $x \leq y$ dans l'ordre pré-topoligique alors il existe aussi un chemin de x à y.

L'algorithme de Kosaraju permet de calculer les composantes fortement connexes (CFC) d'un graphe orienté. Il procède par parcours en profondeur successifs depuis chaque sommet pris dans l'ordre (pré-)topologique :

1. Calculer un tri topologique de G. On obtient le résultat sous forme d'une pile (liste).
2. Calculer le graphe transposé $\overline{G}$.
3. Pour chaque sommet x de la pile, si le sommet x n'a jamais été visité alors déclencher un parcours en profondeur depuis x dans le graphe transposé $\overline{G}$. Tous les sommets rencontrés seront notés comme appartenant à la même composante connexe.
4. L'algorithme termine lorsqu'il n'y a plus de sommets dans la pile.

L'algorithme repose sur l'idée suivante :

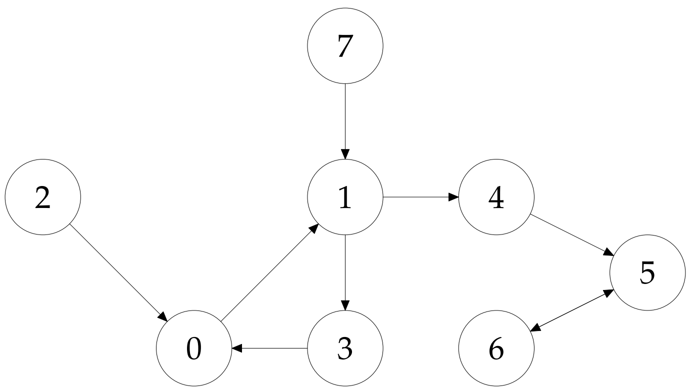
- Par définition de la CFC, le parcours en profondeur depuis x dans $\overline{G}$ va bien atteindre au moins tous les sommets de la CFC contenant x.
- Réciproquement si un sommet y est atteint lors de ce parcours alors on sait qu'il existe un chemin de y à x dans G. De plus, $x \leq y$ dans l'ordre (pré-)topologique, donc il existe un chemin menant de x à y dans G. Ainsi y est bien dans la même CFC que x.

> **Question 3**
>
> Compléter le programme suivant calculant les composantes fortement connexes d'un graphe orienté. La réponse est donnée sous frome d'un tableau `composante` où `composante.(i)`

contient le numéro de la CFC à laquelle le sommet i appartient. Lorsque `composante.(i) = -1` cela signifie que le sommet n'a pas encore été visité.

```
let kosaraju g =
  let n = g.taille in
  let gbar = transpose g in
  let ordre = tri_topo g in
  let composante = Array.make n (-1) in
  let composante_actuelle = ref 0 in
  (* realise le parcours en profondeur
   * depuis s dans gbar des sommets
   * non visites et les marque avec composante_actuelle *)
  let rec parcours_profondeur s =
    (* A COMPLETER *)
  in
  (* prend une liste de sommets à traiter. Pour chaque sommet
     si le sommet n'a pas été visité déclencher le parcours
     en profondeur depuis ce sommet, puis incrémenter le
     numéro de composante_actuelle
   *)
  let rec traite_sommets l =
    (* A COMPLETER *)
  in
  traite_sommets ordre;
  composante
;;
```

On pourra vérifier le bon fonctionnement de l'algorithme sur le graphe suivant :



Question 4

Quelle est la complexité temporelle de tous ces algorithmes ?