TI301 – Algorithmique et Structures de données 2

Projet Informatique et Mathématiques

Ce projet est rédigé en commun avec le département de mathématiques

Etude de Graphes de Markov – PARTIE 2

Nous allons maintenant nous intéresser aux propriétés des graphes de Markov.

Pour obtenir l'ensemble de ces propriétés, deux étapes sont fondamentales

Étape 1 – Regroupement de sommets en classes

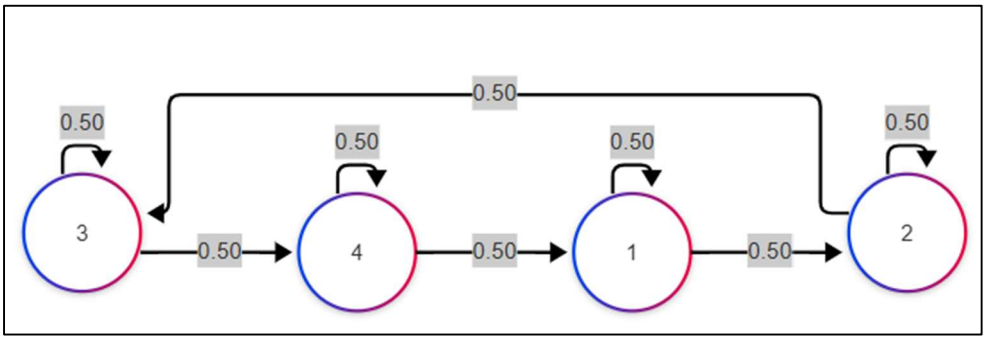
Dans ces graphes, nous allons rassembler certains sommets entre eux, selon le critère suivant :

> Des sommets doivent être regroupés **dans une classe** s'ils communiquent tous entre eux**, c'est à dire qu'à partir d'un de ces sommets, on peut accéder à tous les sommets de cette classe (lui-même y compris).**

Ces regroupements de sommets sont nommés **composantes fortement connexes** ou **classes.**

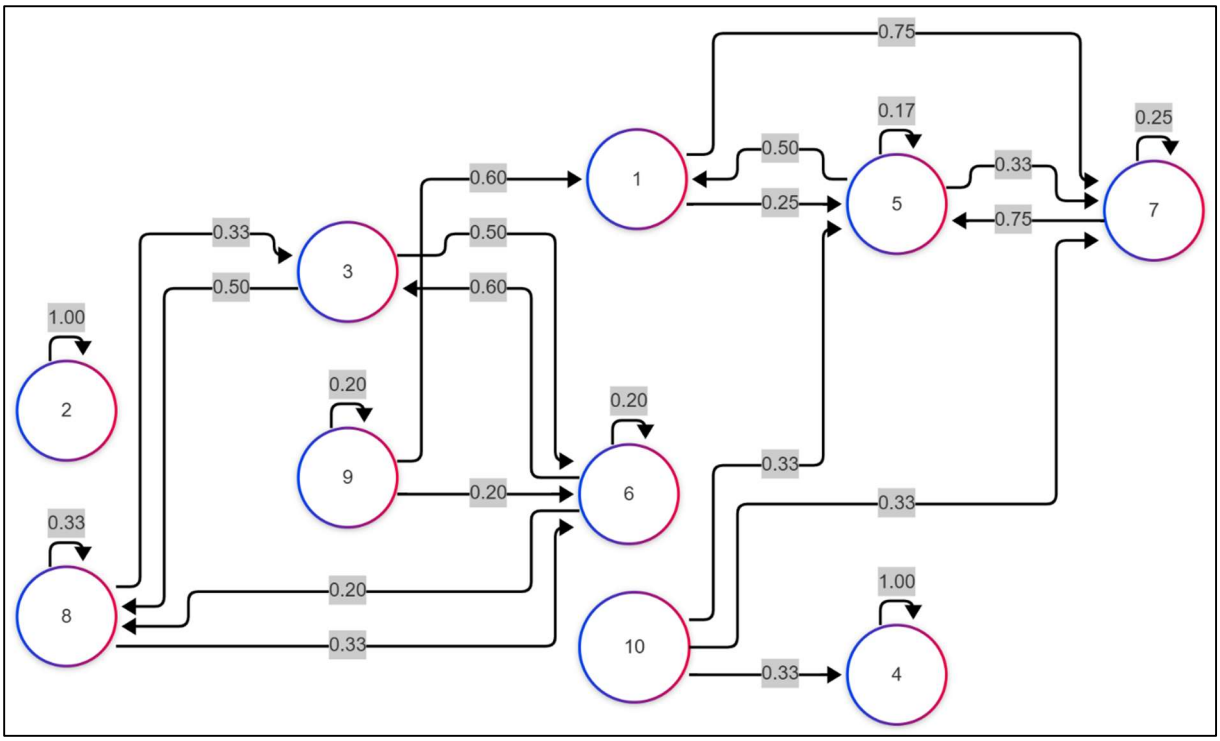
Quelques exemples

1) Graphe avec 4 sommets, sa matrice est $M = \begin{pmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0 & 0.5 \end{pmatrix}$



Ici, tous les sommets communiquent entre eux : partant d'un sommet, on peut accéder à tous les autres, et y revenir. On doit regrouper les 4 sommes dans une **classe** que l'on nommera {1,2,3,4}

2) Graphe avec 10 sommets, utilisé pour valider l'étape 3 de la partie 1



- Le sommet **2** ne communique qu'avec lui-même : il forme une **classe** à lui seul
- Le sommet **4** ne communique qu'avec lui-même : il forme une **classe** à lui seul
- Le sommet **10** forme une **classe** à lui seul également : on peut partir de **10** vers d'autres sommets, mais on ne peut pas y revenir.
- Le sommet **9** est dans le même cas : on peut en partir mais pas y revenir
- Les sommets **1**, **5** et **7** forment une classe : partant de **1**, **5** ou **7**, on peut toujours revenir à **1**,**5** ou **7**
- Les sommets **3**, **6** et **8** forment une classe : partant de **3**, **6** ou **8**, on peut toujours revenir à **3**, **6** ou **8**.

Pour ce graphe, l'ensemble des classes est donc : $\{1,5,7\}, \{2\}, \{3,6,8\}, \{4\}, \{9\}$ et $\{10\}$.

Propriétés des classes

- ✓ Tout sommet du graphe appartient à une et une seule classe ;
- ✓ Tout sommet du graphe doit appartenir à une classe ;
- ✓ Toute classe comporte au moins 1 sommet ;
- ✓ Un graphe comprend au moins 1 classe ;
- ✓ L'ensemble des classes d'un graphe s'appelle une **partition**.

Déterminer ces classes

Pour déterminer ces classes, de nombreux algorithmes existent, chacun avec sa difficulté de conception (est-il facile à concevoir, à comprendre ?) et sa complexité (est-il efficace ?). Les algorithmes les plus simples à comprendre et concevoir ont en général une mauvaise complexité, en $O(\mathrm{N}^3)$ au minimum (N étant le nombre de sommets).

Nous allons utiliser un algorithme très efficace pour réaliser la partition du graphe en classes : **l'algorithme de Tarjan** (voir https://fr.wikipedia.org/wiki/Algorithme_de_Tarjan). C'est la version de cet algorithme que vous devrez implémenter.

Cet algorithme étant assez complexe (notamment, sa présentation sur la page wikipedia nécessite une fonction dans une fonction, ce qui n'est pas réalisable facilement avec ce que nous savons faire pour le moment), il sera décomposé en étapes.

Etape 1 pour l'algorithme de Tarjan – structures de données

1.1) Cet algorithme utilise des informations associées à chaque sommet :

- ✓ Un **identifiant (`entier`)** : c'est le numéro du sommet dans le graphe, tel que vu en Partie 1
- ✓ Un **numéro (`entier`)** : pour effectuer une numérotation temporaire pour déterminer les classes – ce numéro n'est pas le numéro du sommet dans le graphe !
- ✓ Un **numéro accessible** (`entier`): pour regrouper les sommets qui communiquent ensemble
- ✓ Un **indicateur booléen** (un `entier` valant 0 ou 1) indiquant si le sommet est dans la pile de traitement ou pas (l'algorithme utilise une pile).

➢ **Il vous faut donc créer une structure de données correspondante.** Pour continuer l'exemple, supposons que ce type s'appelle `t_tarjan_vertex` (ce n'est qu'un exemple, vous pouvez choisir le nom que vous souhaitez).

1.2) Cet algorithme a besoin de connaître l'état de tous les sommets quand il s'exécute.

➢ **Il vous faut donc créer un tableau** stockant tous les `t_tarjan_vertex` pour un graphe.

1.3) Cet algorithme utilise des classes, pour y stocker les sommets au fur et à mesure. Une classe stocke des 'sommets' `t_tarjan_vertex`, et a un nom : « C1 », « C2 »....

➢ **Il vous faut donc créer une structure de données correspondante,** qui stocke un nom (le nom de la classe) et 'un certain nombre de' sommets `t_tarjan_vertex`. Pour continuer l'exemple, supposons que ce type s'appelle `t_classe`.

A vous de choisir comment faire ce stockage, sachant qu'on ne sait pas à l'avance combien une classe contiendra de sommets.

1.4) Cet algorithme fournit la partition du graphe en classes. Une partition est un ensemble de classes.

- **Il vous faut donc créer une structure de données correspondante,** qui stocke 'un certain nombre de classes `t_classe`. (à vous de choisir comment faire ce stockage)

  A vous de choisir comment faire ce stockage, sachant qu'on ne sait pas à l'avance combien une partition contiendra de classes.

Etape 2 pour l'algorithme de Tarjan – éléments utilitaires

Il est très fortement conseillé d'implémenter les éléments suivants :

2.1) Une fonction à qui on fournit un graphe (une liste d'adjacence) et qui retourne un tableau de `t_tarjan_vertex`, (un pour chaque sommet présent dans la liste d'adjacence), initialisé avec :

- ✓ **identifiant (`entier`)** : numéro du sommet dans le graphe (liste d'adjacence)
- ✓ **numéro (`entier`)** : -1
- ✓ **numéro accessible** (`entier`): -1
- ✓ Un **indicateur booléen** (un `entier` valant 0 ou 1) : 0

2.2) Une pile dans laquelle on stockera des `int` ou des `t_tarjan_vertex`

Etape 3 pour l'algorithme de Tarjan – découpage en fonctions

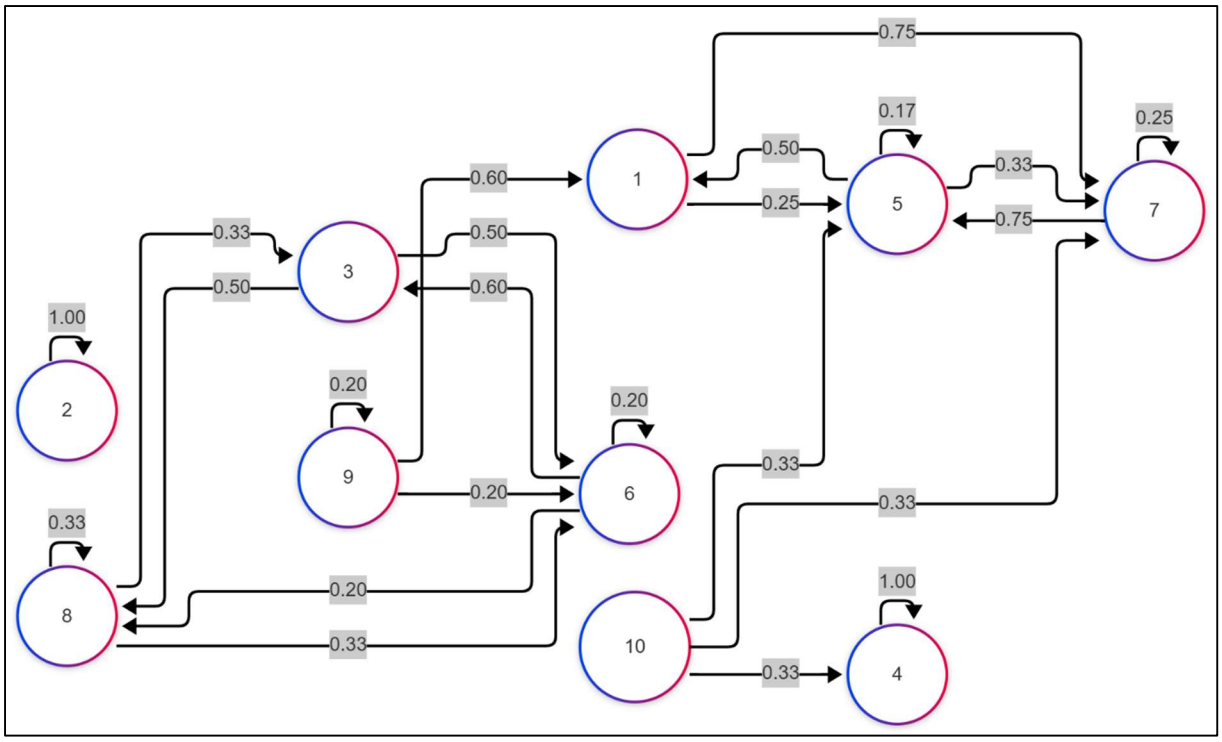
3.1) Écrire une fonction qui réalise la sous-fonction nommée '**parcours**' de la page wikipedia. Attention, cette fonction aura besoin de 5 ou 6 paramètres.

3.2) Écrire une fonction '**tarjan**' qui initialise une **partition** vide, les `t_tarjan_vertex`, une **pile** vide, et initie les parcours.

Validation de l'étape 1

Voici le type de sortie attendue pour le résultat de l'algorithme de Tarjan.

Avec le graphe :



La sortie du programme devrait être : (seules les valeurs sont importantes, pas les formats d'affichage)

```
Composante C1: {1,7,5}
Composante C2: {2}
Composante C3: {3,8,6}
Composante C4: {4}
Composante C5: {9}
Composante C6: {10}
```

L'ordre dans lequel les composantes sont affichées n'a pas d'importance

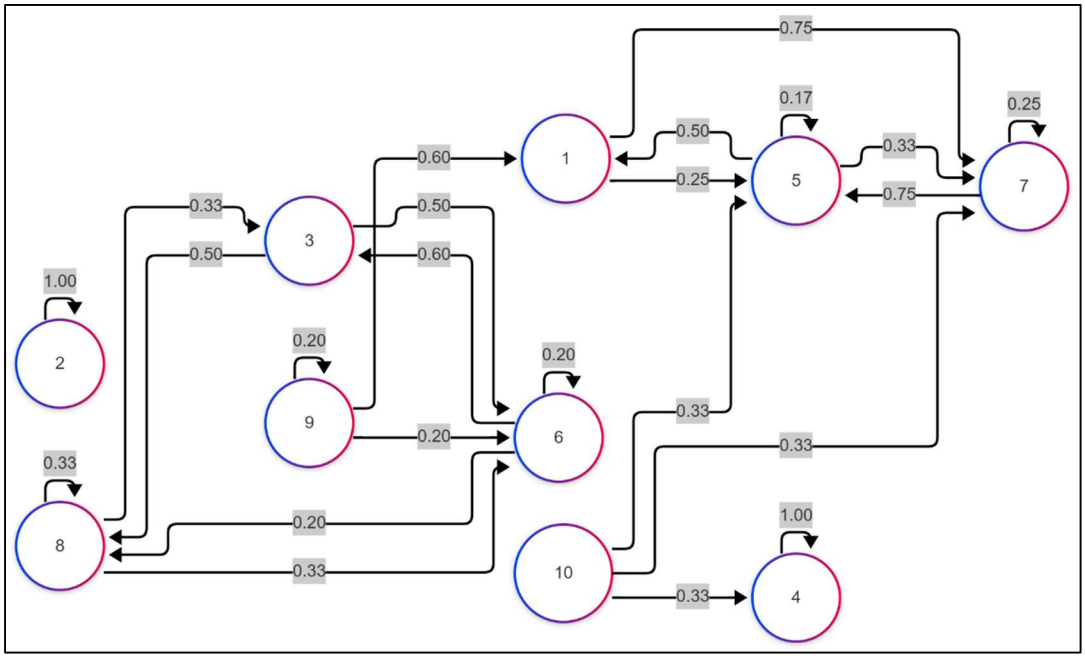
Étape 2 – Diagramme de Hasse

On cherche maintenant à obtenir le diagramme de Hasse associé aux **classes** obtenues à l'étape 1.

Ce diagramme indique les communications entre classes. En effet, certains sommets d'une classe peuvent mener à des sommets d'autres classes (mais il n'y a pas de 'retour' possible).

Ce diagramme de Hasse nous permettra d'obtenir toutes les propriétés des sommets, des classes, et du graphe de Markov.

Illustration, toujours avec le même graphe pris en exemple.



Les classes sont :

```
Composante C1: {1,7,5}
Composante C2: {2}
Composante C3: {3,8,6}
Composante C4: {4}
Composante C5: {9}
Composante C6: {10}
```

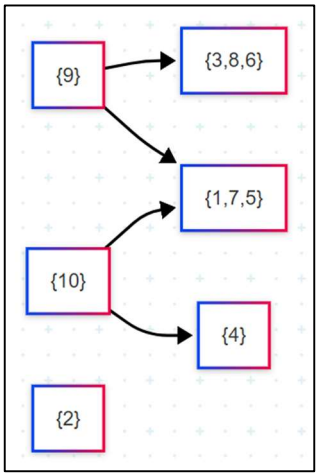
Du sommet **10**, on peut aller vers sommet **4** : on a donc un lien de **C6** vers **C4**, que l'on représentera par **C6->C4**

Du sommet **10**, on peut aller vers le sommet **7** : on a donc un lien de **C6** vers **C1** : **C6->C1**

Du sommet **10**, on peut aller vers le sommet **5** : on a donc un lien de **C6** vers **C1** (déjà identifié)

Du sommet **1**, on peut aller vers les sommets **5** et **7**, qui sont dans la même classe **C1** : pas de lien entre classes

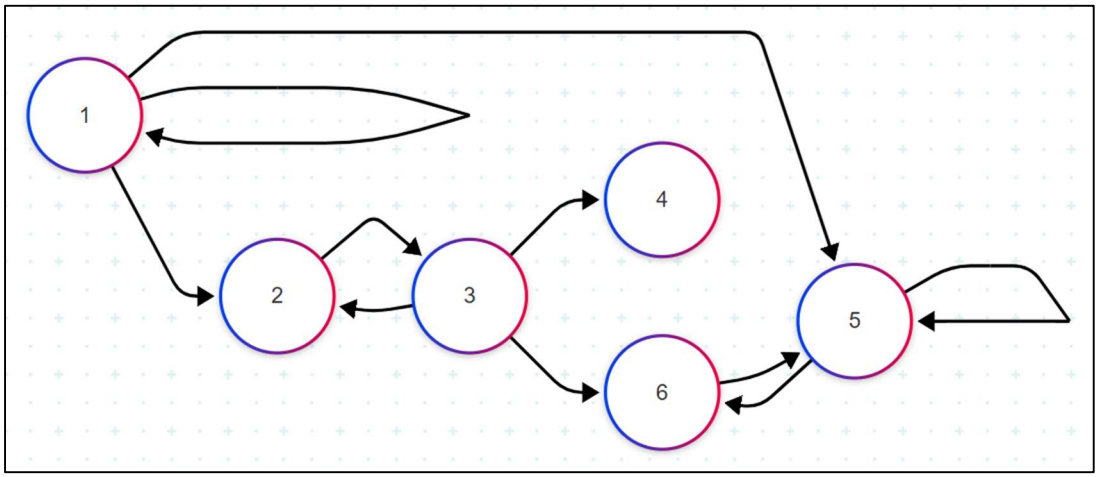
En suivant le même principe pour tous les sommets, on obtient finalement (visuel final avec Mermaid)



[EN OPTION] Dernière définition

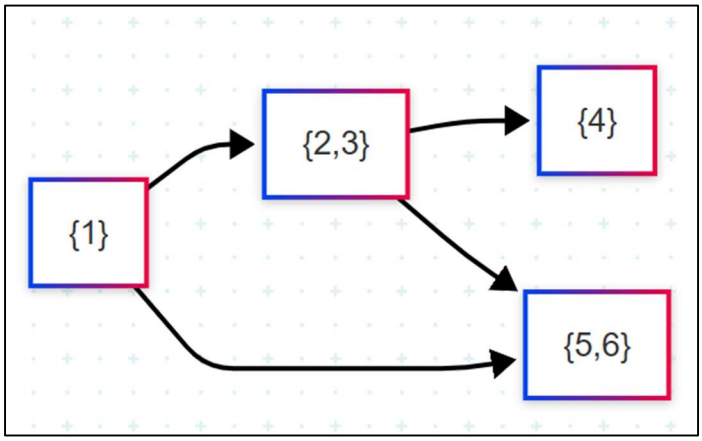
Dans ce diagramme, on ne devrait pas trouver de redondance.

Par exemple, pour le graphe suivant :



Les classes sont {1}, {2,3}, {4}, {5,6}

Les liens entre classes produisent le diagramme suivant :



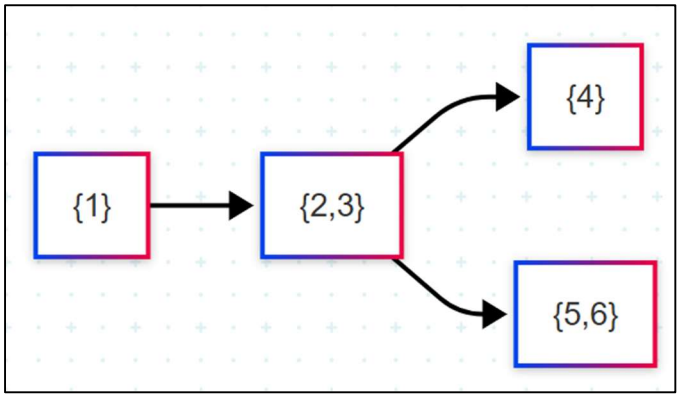
Il y a une redondance, car

le lien de **{1}** vers **{5,6}**

est redondant avec

le lien de **{1}** vers **{2,3}** + le lien de **{2,3}** vers **{5,6}**.

Le diagramme de Hasse est un diagramme sans redondance. Pour le même graphe, on obtient :



Comment recenser les liens

Pour obtenir un algorithme efficace, nous allons utiliser procéder ainsi :

- ✓ Créez une structure 'lien' qui stockera 2 entiers : un entier pour la classe de « départ », un entier pour la classe d' « arrivée ».
- ✓ Créez une structure pour stocker plusieurs 'liens'.
- ✓ Créer un tableau qui indique, pour chaque sommet du graphe, la classe à laquelle il appartient.

Attention, pour le moment, chaque classe 'connaît' les sommets qui la composent, mais chaque sommet ne 'connaît' pas la classe à laquelle il appartient.

Voici l'algorithme à implémenter

```
Pour chaque sommet i du graphe
     Ci = classe à laquelle appartient i // avec le tableau
     Pour tous les sommets j dans la liste d'adjacence du sommet i
           Cj = classe à laquelle appartient j // avec le tableau
           Si Ci est différent de Cj // arête entre classes
               Si le lien (Ci,Cj) n'existe pas
                   Ajouter le lien (Ci,Cj) dans la structure qui
stocke les liens
```

[EN OPTION] Comment retirer les redondances

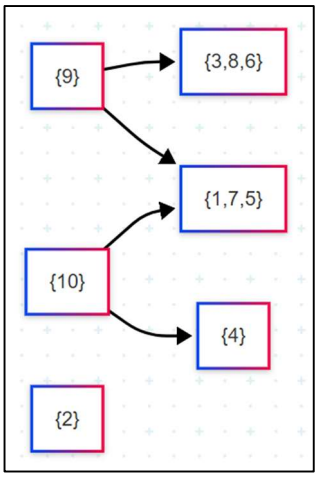
La fonction qui réalise cette opération est déjà programmée : elle se trouve dans le module **`hasse.h`/`hasse.c`**.

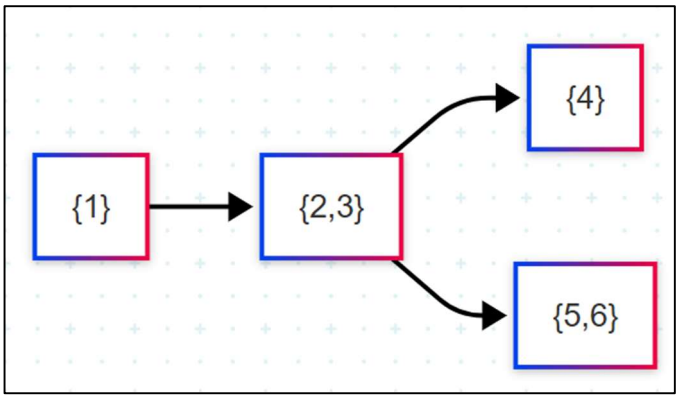
son prototype est : **`void removeTransitiveLinks(t_link_array *p_link_array);`**

Il vous faudra probablement l'adapter à vos propres structures de données.

Validation de l'étape 2

Produire un diagramme de Hasse (avec ou sans redondances) au format Mermaid, par exemple :





Vous aurez besoin, pour construire ces diagrammes, de la partition du graphe (ensemble des classes) et de tous les liens entre classes que vous aurez trouvés.

Étape 3 : caractéristiques du graphe

Une fois les classes et les liens obtenus, ces caractéristiques s'obtiennent de manière quasi immédiate :

- ✓ Une classe est dite '**transitoire**' si on peut 'sortir' de cette classe : il existe une 'flèche sortante' de cette classe. Tous les états de cette classe sont dits '**transitoires'.**
- ✓ Une classe est dite '**persistante**' si on ne peut pas 'sortir' de cette classe : il n'y a pas de flèche 'sortante' de cette classe. Tous les états de cette classe sont dits '**persistants**'
- ✓ Un état est dit '**absorbant**' s'il est dans une classe persistante, et qu'il est le seul état de cette classe.
- ✓ Un graphe de Markov est dit '**irréductible**' s'il n'est composé que d'une seule classe.

Sur le schéma de droite :

- ✓ La classe {1} est transitoire – l'état 1 est transitoire ;
- ✓ La classe {2,3} est transitoire – les états 2 et 3 sont transitoires ;
- ✓ La classe {4} est persistante – l'état 4 est persistant – l'état 4 est absorbant ;
- ✓ La classe {5,6} est persistante – les états 5 et 6 sont persistants ;
- ✓ Le graphe de Markov n'est pas irréductible.

Validation de l'étape 3

Votre programme affiche toutes les caractéristiques du graphe :

- ✓ Si les classes sont transitoires ou persistantes ;
- ✓ S'il y a des états absorbants ;
- ✓ Si le graphe est irréductible ou non.