Devoir 2 : Recherche des composantes fortement connexes d'un graphe.

Nous allons coder un algorithme de recherche des composantes fortement connexes d'un graphe. Au passage, nous résolverons aussi le problème du tri topologique d'un graphe acyclique. Comme application, nous donnerons un algorithme décidant si une formule propositionnelle de la forme 2-SAT possède une assignation qui la satisfait.

1 Rendu

Le projet se déroule sur les séances de TP 4 à 6 en monôme ou en binôme. À leurs issues et à la date prescrite (selon votre campus), vous devrez rendre sur Ametice, sous forme d'un unique fichier archive d'extension `gz` ou `tar.gz` **exclusivement**, contenant l'ensemble de vos sources et un rapport décrivant comment utiliser votre projet, vos résultats, et le cas échéant une description des fonctionnalités non-implémentées. Si votre projet contient du code source que vous n'avez pas écrit vous-même, vous indiquerez précisément sa nature et sa provenance. Tout projet contenant du code non-correctement attribué à ces auteurs s'expose à recevoir une note nulle.

La date de rendu dépend de votre campus, demandez à votre chargé de TP.

2 Les composantes fortement connexes

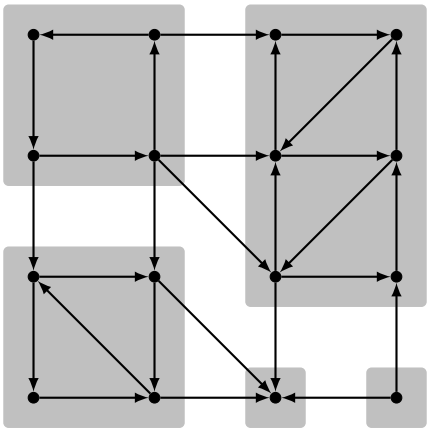
2.1 Introduction

Soit G un graphe orienté. Rappelons qu'un sommet u est accessible depuis un sommet v s'il existe un chemin de source v et de destination u, ce que nous appelons un vu-chemin.

Il est possible que u soit accessible depuis v sans que v ne soit accessible depuis u. La relation *être accessible depuis* n'est donc pas symétrique. On peut par contre vérifier facilement que c'est une relation réflexive et transitive. Nous pouvons alors définir la relation *être mutuellement accessible* par : u et v sont *mutuellement accessibles* si u est accessible depuis v et v est accesssible depuis u.

Nous appelons *composante fortement connexe* de G tout ensemble maximal de sommets C telle que pour toute paire $(u, v) \in C$, u est accessible depuis v et v est accessible depuis u. Il s'agit d'une classe d'équivalence d'*être mutuellement accessible*. Nous avons donc que pour tout sommet $w \notin C$, ou bien w n'est pas accessible depuis les sommets de C, ou bien aucun sommet de C n'est accessible depuis w (possiblement les deux).

L'exemple suivant montre les cinq composantes fortement connexes d'un graphe :



2.2 Algorithme

Nous utilisons l'algorithme de Kosaraju.

— Nous procédons à un premier parcours itéré en profondeur de G, ceci nous donne un ordre de sortie sur les sommets. Il s'agit du même algorithme que celui permettant de trouver un tri topologique, mais que nous utilisons sur un graphe qui n'est pas nécessairement acyclique.

— Dans cet ordre en sens décroissant (donc en partant du dernier sommet trouvé par la première étape), nous exécutons un deuxième parcours itéré en profondeur, mais dans $\overleftarrow{G}$. Il s'agit du graphe obtenu en inversant la direction de chaque arc dans G. Chaque parcours élémentaire produit une liste de sommets qui est une composante fortement connexe de G.

La Figure 1 montre un exemple d'exécution de l'algorithme.

2.3 Tâches

Votre travail devra aborder les points suivants :

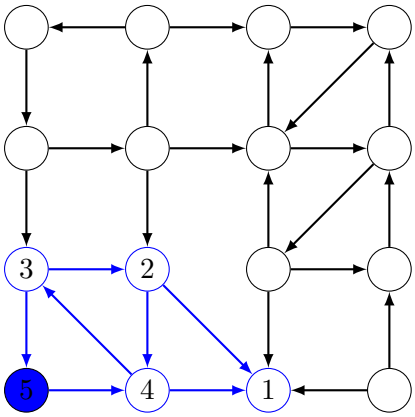
— Représentation des graphes par listes d'incidence. Vous utiliserez les ArrayList et les LinkedList de Java pour les tableaux et les listes respectivement. Les sommets seront indicés de 0 à $n-1$.

— Exécution d'un parcours en profondeur itéré, en fonction d'une liste donnant un ordre sur les sommets. Vous prendrez donc comme première source le premier sommet de la liste, comme deuxième source le premier sommet non-parcouru de la liste, etc. L'algorithme doit renvoyer une liste de listes de sommets. Chaque liste interne représente le résultat d'un appel au parcours sur une source, et contient la liste des sommets découverts par ordre de sortie. Ainsi lors du second parcours cela vous donnera la liste des composantes fortement connexes.

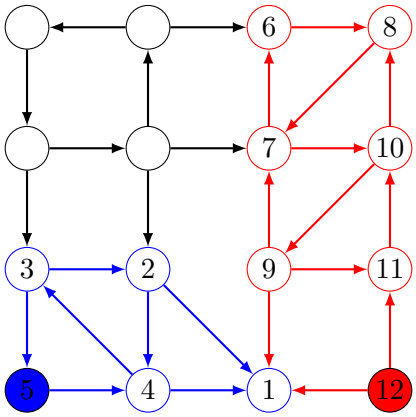
— Pour le deuxième parcours, il vous faut travailler sur le graphe $\overleftarrow{G}$. Une possibilité est de construire deux graphes dès le début, l'autre de faire une méthode renversant le graphe, selon votre choix.

— Lecture d'une formule 2-SAT et construction du graphe associé.

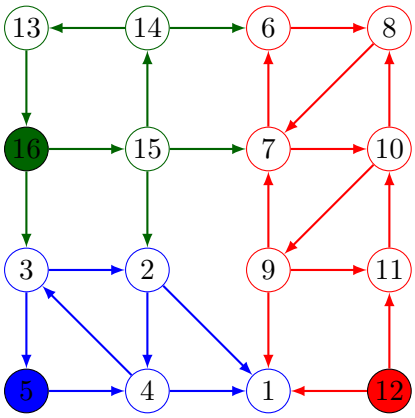
— Facultatif : construire une assignation satisfaisant la formule.



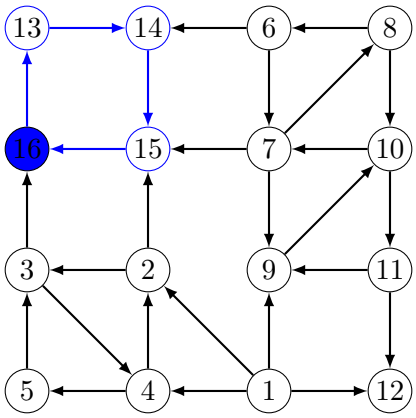
Parcours en profondeur du sommet bleu



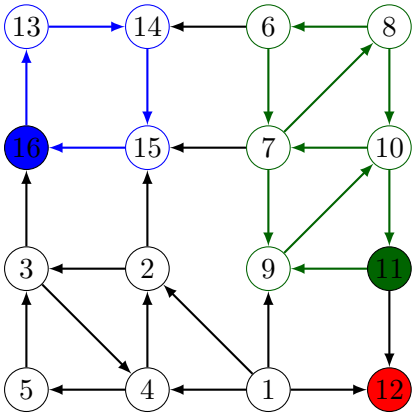
On repart du sommet rouge



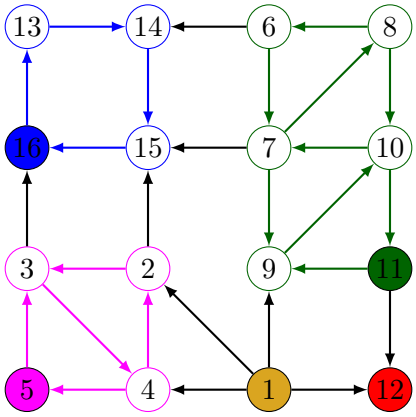
On repart du vert, ceci termine la phase 1



Première composante fortement connexe



Deuxième (rouge) et troisième (verte)



Quatrième (fuschia) et Cinquième (jaune)

FIGURE 1 – Exemple d'exécution de l'algorithme

3 Application

Étant donnée une formule de logique propositionnelle sur un ensemble de variables booléennes, par exemple $\phi_{x,y,z} = x \wedge (y \vee \overline{z})$, se pose la question de savoir s'il existe un choix de valeurs booléennes telles que la formule s'évalue en vrai. Rappelons que $\wedge$ est le symbole de la conjonction (et), $\vee$ celui de la disjonction (ou) et $\overline{z}$ est la négation de z. Ainsi $\phi_{x,y,z}$ est vraie pour le choix suivant : $x =$ vrai, $y =$ faux, $z =$ faux.

En règle générale, trouver un choix satisfaisant une formule est un problème difficile à résoudre. On ne sait pas s'il existe un algorithme de complexité polynomiale pour ce problème, et la plupart des experts pensent que non. Nous allons donc être moins ambitieux, et résoudre le problème dans un cas particulier 2-SAT.

Une formule de 2-SAT est une disjonction de clauses, chaque clause étant une conjonction de deux littéraux. C'est donc une formule de la forme :

$$(x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

Nous lirons ces formules dans des fichiers, chaque ligne comprendra une clause codée comme deux entiers, un entier positif pour une variable, et un entier négatif pour la négation d'une variable. La formule précédente sera donc décrite par :

```
1 -2
1 3
1 -3
```

Pour résoudre 2-SAT avec des variables $x_1, \ldots, x_n$, on construit un graphe sur les sommets $\{x_1, x_2, \ldots, x_n\} \cup \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$, et pour toute clause $(a \vee b)$, on ajoute deux arcs $\overline{a} \to b$ et $\overline{b} \to a$.

Alors la formule possède un assignement de valeurs aux variables qui la satisfait si et seulement si il n'existe pas de variable x_i tel que x_i et $\overline{x_i}$ sont dans la même composante fortement connexe.

4 Implémentation

À nouveau l'usage de Java est imposé. Vous pouvez (devez ?) utiliser les classes ArrayList et LinkedList pour coder les graphes. Ces classes possèdent en plus des itérateurs prédéfinis, ce qui vous permet de faire des boucles for pour parcourir ces structures très facilement.

Le graphe est lu depuis l'entrée standard, en utilisant les méthodes hasNextInt et nextInt de la classe Scanner. Adaptez ce code selon vos besoins :

```
Scanner scan = new Scanner(System.in);
while (scan.hasNextInt()) {
  source = scan.nextInt();
  destination = scan.nextInt();
  graph.addEdge(source,destination);
}
```