# Projet Maths Info

Yann BERTHELOT & Louis LEENART

## Création

La création d'un object graphe simple se fait avec la commande `new GraphSimple(taille)`. L'ajout de données dans ce graphe se fait par liste d'adjacence en utilisant la méthode `setAdjacencyList(sommet, liste)`. On note que `sommet` est un entier entre `1` et `taille` du graphe.

Dans le fichier `Main.java`, on utilise la lecture sur la console pour ajouter les données.
Il est possible d'initialiser le graphe de la manière suivante :

```
Scanner scanner = new Scanner(System.in);
        // Lecture de la taille du graphe
        int size = scanner.nextInt();
        // Initialisation du graphe
        GraphSimple graphe = new GraphSimple(size);
        // Lecture des données d'entrée
        // Pour chaque ligne, on lit le flot de données, et on l'applique
au graphe.
        for (int i = 0; i < size; i++) {
            int sommet = scanner.nextInt();
            int[] temp = new int[size];
            int count = 0;
            while (scanner.hasNextInt()) {
                int nextInt = scanner.nextInt();
                if (nextInt == 0) {
                    break;
                } else {
                    temp[count] = nextInt;
                    count++;
                }
            }
            int[] newArray = new int[count];
            System.arraycopy(temp, 0, newArray, 0, count);
            graphe.setAdjacencyList(sommet, newArray);
        }
        // Le flot de données est terminé, on ferme alors le scanner
(entrée de données)
        scanner.close();
        // Le graphe est alors completement initialisé
```

## Utilisation

L'objet GraphSimple comporte plusieurs modules permettant de faire des opérations sur le graphe.

### Conversion

Étant donné que le graphe est créé via sa matrice d'adjacence, il est possible d'en déduire la matrix d'adjacence via la méthode `toMatrix()`.
Il est aussi possible de faire chemin inverse via `fromMatrix()` pour obtenir le tableau de listes d'adjacences via la matrice d'adjacence mais cela a peu d'interet ici.

## Parcours en Largeur

Le parcours en largeur se lance via la méthode `parcoursLargeur(sommet?)`. Le sommet indiqué en entrée correspond au sommet de départ du parcours. On note que l'on peut aussi executer cette méthode sans argument pour réaliser le parcours en largeur sur un sommet aléatoire.
Cette méthode fait alors appel à `initParcoursLargeur()` qui initialise (ou réinitialise si ce n'est pas le premier parcours réalisé) les données, puis le parcours est lancé via `parcoursLargeurAux(sommet)`.
Pour récupérer le résultat de ce parcours, les données sont contenues dans les trois listes suivante : `colors`, `distances` et `parents`. Ces données sont accessibles via les getters suivants : `getColor(sommet)`, `getDistance(sommet)` et `getParent(sommet)`.

## Parcours Complet

Le parcours complet se lance via la méthode `parcoursComplet()`. Cette méthode permet de faire un parcours en largeur sur tous les sommet du graphe (on note que l'on execute le parcours en largeur sur tous les sommets qui n'ont jamais été visité jusqu'à ce que tous les sommets le soient).

## Test de connexité

Le test de connexité se lance via la méthode `testConnexity()` et retourne un booléen. Cette méthode réalise un parcours en largeur à partir d'un sommet aléatoire du graphe, et si on remarque que tous les sommets n'ont pas été parcourus, alors on en déduit que le graphe n'est pas connexe.

## Nombre de composantes connexes

Le nombre de composantes connexes est calculé via la méthode `countComposantesConnexe()` qui retourne un entier (le nombre de composantes connexes). Pour compter le nombres de composantes connexes, on réalise un parcours en largeur sur chaque sommet non-visité jusqu'a ce tous les sommets le deviennent. On stocke les donnés accessible via `getComposanteConnexe(sommet)`.

## Accès a certaines données

Il est possible d'acceder a de nombreuses données pour chaque graphe :

- `getAdjacencyList(sommet) -> int[]` retourne la liste d'adjacence du sommet
- `order() -> int` retourne l'ordre du graphe
- `degree(sommet) -> int` retourne le degrée du sommet
- `isVertex(sommet) -> boolean` retourne si un sommet est un vertex ou non
- `isEdge(sommet_a, sommet_b) -> boolean` retourne si les deux sommets forment une edge
- `getColor(sommet) -> Enum_Color` retourne la couleur du sommet (uniquement apres parcours en largeur du graphe)
- `getDistance(sommet) -> int` retourne la distance du sommet au sommet de départ du dernier parcours en largeur. Initialisé à `-1`

- `getParent(sommet) -> int` retourne le sommet parent du sommet d'entrée dans le cadre du dernier parcours en largeur réalisé
- `getConnexe() -> boolean` retourne si le graphe est connexe ou non, initialisé apres le lancement de la méthode `testConnexity()`
- `getComposanteConnexe(sommet) -> int` retourne la composante connexe a laquelle appartient le sommet. Disponible apres le lancement de la méthode `countComposanteConnexe()`

# Exercice

## Question A

Voir fichier `Enum_Color.java`.

## Question B

Voir fichier `GraphSimple.java` section `ATTRIBUTS`.

## Question C

Voir `GraphSimple.java`, section `METHODES`, méthode `initParcoursLargeur()`.

## Question D

Voir `GraphSimple.java`, section `METHODES`, méthode `parcoursLargeur()`. Note : On peut lancer l'algo de parcours en largeur soit sans argument (sommet aléatoire) soit avec un entier en argument (qui est le sommet de départ du parcours).

## Question E

On test l'algorithme de parcours sur le graphe Peterson. Il faut décommenter la section `EXERCICE 1 QUESTION E` dans le fichier `Main.java` et puis lancer les commandes suivantes :

```
javac graph/*.java
java graph.Main < data/graph-002.alists
```

## Question F

Il faut commenter la section `EXERCICE 1 QUESTION E` et décommenter la section `EXERCICE 1 QUESTION F`. Il faut ensuite lancer les commandes suivantes :

```
javac graph/*.java
java graph.Main < data/graph-003.alists
```

Suite au parcours en largeur, on remarque que le graphe n'est pas connexe. Il est composé d'au moins 2 composantes connexes. Le résultat du parcours (sur un sommet aléatoire) est le suivant :

```
1  : [Color : Green | Distance : -1 | Parent : -1 ]
2  : [Color : Red   | Distance :  2 | Parent :  3 ]
3  : [Color : Red   | Distance :  1 | Parent :  4 ]
4  : [Color : Red   | Distance :  0 | Parent :  4 ]
5  : [Color : Green | Distance : -1 | Parent : -1 ]
6  : [Color : Green | Distance : -1 | Parent : -1 ]
7  : [Color : Red   | Distance :  1 | Parent :  4 ]
8  : [Color : Red   | Distance :  1 | Parent :  4 ]
9  : [Color : Green | Distance : -1 | Parent : -1 ]
10 : [Color : Green | Distance : -1 | Parent : -1 ]
11 : [Color : Green | Distance : -1 | Parent : -1 ]
12 : [Color : Red   | Distance :  1 | Parent :  4 ]
13 : [Color : Green | Distance : -1 | Parent : -1 ]
```

### Question G

Voir `GraphSimple.java`, section `METHODES`, méthode `parcoursComplet()`

## Exercice 3

Voir `GraphSimple.java`, section `METHODES`, méthodes `testConnexity()` et `countComposantesConnexe()`. Pour executer l'algorithme, il faut commenter la section `EXERCICE 1 QUESTION E` et décommenter la section `EXERCICE 3` puis executer les commandes suivantes :

```
javac graph/*.java
java graph.Main < data/graph-002.alists
```

Note : il est évidemment possible d´utiliser n'importe quel autre fichier en entrée via sa liste de tableaux d'adjacences (.alist).

### Note

Une description plus en profondeur de chaque algorithme est faite dans le fichier `GraphSimple.java` pour chaque méthode importante.