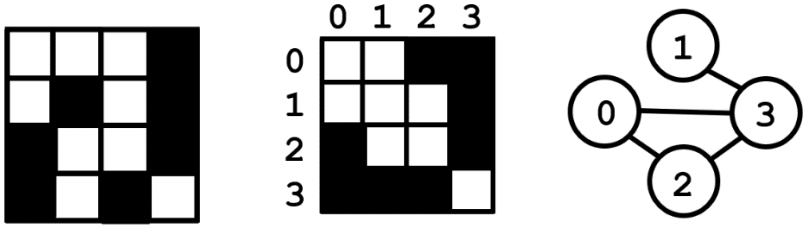
Propagatio

En combien d'étapes une information se propage-t-elle dans un réseau ?



a b c

Figure 1: (a) image de type **pbm** lue par le programme sur l'entrée standard,
(b) image symétrisée avec diagonale nulle pour en faire la matrice d'adjacence d'un graphe non-orienté,
(c) graphe non-orienté correspondant à (a) et (b) ; un outil en ligne est fourni pour obtenir ce type de dessin.

1. Introduction

Ce projet explore la propagation d'une information dans un système composé de plusieurs éléments reliés entre eux par un moyen de communication tel qu'un réseau informatique ou un réseau social.

1.1 Représentation d'un graphe non-orienté :

Pour cela nous modélisons ce système par un ***graphe*** dans lequel un élément du système est un ***nœud*** du graphe ; si deux éléments du système sont reliés par un moyen de communication direct alors leurs nœuds sont reliés par une ***arête*** ; on dit qu'ils sont voisins.

L'ensemble des relations de voisinage des nœuds d'un graphe est mémorisé dans la *matrice d'adjacence*, notée **matAdj**. Si le graphe contient **n** nœuds, que nous numérotons de ***zéro à n-1,*** alors la matrice d'adjacence **matAdj** est une matrice carrée **n x n**. Un élément de cette matrice ne peut prendre que deux valeurs : 0 ou 1. Le nœud **i** est relié au nœud **j** par une arête si l'élément **matAdj (i,j)** vaut **1**. La figure 1b montre un exemple de matrice d'adjacence avec un pixel noir pour chaque valeur 1 ; on peut voir le graphe correspondant sur la figure 1c.

Pour ce projet nous supposons que le graphe est ***non-orienté***, c'est-à-dire que le lien entre deux nœuds n'a pas de direction. Cela se traduit par une matrice **matAdj** symétrique pour laquelle **matAdj (i,j) = matAdj (j,i)**. Enfin les éléments **matAdj (i,i)** de la diagonale valent **0** car on considère pour ce projet qu'un nœud n'a pas d'arête qui reboucle sur lui-même.

Pour le graphe de la Fig1 on peut voir qu'une information partant du nœud de numéro zéro se propage en maximum 2 étapes à l'ensemble du graphe.

1.2 Tâches principales (détails en section 3):

Le problème contient plusieurs tâches à effectuer dans cet ordre :

1) Lire sur l'entrée standard pour construire le graphe non-orienté
   - Vérifier que l'image fournie en entrée est carrée et ne contient que des 1 et des 0
   - Symétriser l'image pour obtenir une *matrice d'adjacence* d'un graphe non-orienté
2) Vérifier que le graphe est ***connexe***
3) Propager une information à partir du premier nœud (numéro zéro).
   - Afficher les listes triées de nœuds recevant l'information simultanément
4) Calculer et afficher le degré de séparation moyen du graphe

2. Méthode du travail

2.1 Mise en oeuvre des grands principes

Le projet représente une quantité de travail bien supérieure aux exercices proposés dans les séries. La mise en oeuvre des principes ***d'abstraction*** et de ***ré-utilisation*** est un élément central dans la stratégie de résolution.

En effet, certaines tâches correspondent à un sous-problème (*abstraction*) dont la solution sous forme d'une fonction peut être utilisée pour résoudre une tâche plus complexe. Il est possible mais pas obligatoire qu'une fonction de bas niveau puisse être *ré-utilisée* à plusieurs endroits dans le code.

2.2 Vérification précoce par les tests (*scaffolding*)

Il est important de réfléchir au *but* de chaque fonction pour déterminer sa « mission » : sur quelles données travaille-t-elle ? Quel(s) résultat(s) fournit-elle ? Au-delà de cette réflexion, il faut ensuite *vérifier* que chaque fonction réalise bien son but avec un solide éventail de tests pour lesquels on connait les résultats attendus. Grâce à cette méthode un sous-problème est résolu une fois pour toute dès le niveau le plus élémentaire.

L'approche de vérification par des tests implique d'*écrire du code supplémentaire dédié à ces tests* AVANT de commencer à traiter les niveaux supérieurs du projet. Vous serez ainsi amené à écrire un certain nombre de petits programmes dédiés à ces tests. Cette méthode de travail est appelée *scaffolding* (échafaudage) et cherche à rendre votre code robuste à la grande variété des cas possibles (à défaut de disposer d'une méthode qui garantirait que votre code est toujours correct).

En effet nous disposons d'outils tels que l'éditeur et le compilateur pour détecter certaines erreurs mais ils ne permettent pas de toutes les trouver. Nous savons déjà qu'il est recommandé de *recompiler très fréquemment* afin réduire au minimum le temps de recherche des ***erreurs syntaxiques*** (fautes d'orthographe/grammaire du langage C++). Le raisonnement est qu'une erreur se trouve dans la portion de code écrite depuis la dernière compilation avec succès, ce qui permet de réduire à très peu de lignes de codes l'espace de recherche de cette erreur.

La méthode de l'échafaudage (scaffolding) est destinée à trouver les ***erreurs sémantiques*** que le compilateur ne trouve pas car le programme respecte la syntaxe du langage ; elles vont produire un comportement incorrect du programme. Le point essentiel dans cette méthode est qu'il faut tester *chaque fonction* ***individuellement*** pour *vérifier* qu'elle produit le résultat attendu AVANT de l'utiliser ailleurs. Là aussi cette approche vous rend plus efficace dans le développement de code car l'erreur sémantique se trouve normalement dans la dernière fonction en cours de test car les autres fonctions qu'elle appelle ont déjà été validées.

2.3 Bottom-up ou top-down ?

La méthode présentée dans la section précédente suggère de vérifier d'abord les fonctions de bas-niveaux avant celles qui les utilisent. C'est l'approche *bottom-up*. C'est en général ce que nous recommandons pour un projet.

A l'inverse, il existe aussi une approche ***top-down*** pour la mise au point des fonctions ; il peut être légitime de vouloir tester une fonction **f()** qui appelle une fonction **g()** avant que **g()** soit

écrite en détail. Cette approche *top-down* est possible si le résultat de **g()** est facile à définir, par exemple si elle renvoie true ou false, car la seule chose utilisée par **f()** est ce résultat.

On peut ainsi vérifier **f()** à l'aide d'une *forme minimale de la fonction* **g()** que l'on appelle un ***stub***. Cette forme minimale respecte le prototype prévu pour **g()** en terme de paramètres et de type de la valeur de retour ; par contre sa définition peut se restreindre à un corps vide si aucune valeur n'est retournée ou très peu de chose, par exemple une instruction return de *true* ou *false* si **g()** est supposée renvoyer un booléen.

2.4 Redirection des entrées-sorties pour automatiser les tests d'un programme

On utilisera **exclusivement** l'entrée standard (*clavier*) pour fournir les données et la sortie standard (*terminal*) pour afficher les résultats. Il ne faut pas écrire de code de lecture-écriture de fichier dans ce projet.

A la place, le mécanisme de *redirection* des entrées-sorties a été vu en TP (semaine6) et permet de travailler avec des fichiers de test *sans avoir à changer une seule ligne de code*. En effet les données d'un test peuvent être éditées avec l'éditeur de programme et mémorisées dans un fichier de test. Ensuite il suffit de préciser le nom du fichier de test sur la ligne de commande qui lance l'exécution du programme et celui-ci obtient les données du fichier comme si elles venaient du clavier. C'est bien pratique pour le niveau élémentaire du projet pour lequel il faut donner un grand nombre de valeurs 0 et 1 sans se tromper…

Cette méthode de test est recommandée surtout pour relancer de manière très efficace un ensemble de tests et vérifier que votre programme est toujours correct. C'est aussi en redirigeant des fichiers de test sur l'entrée standard que nous testerons votre programme ; nous redirigerons le résultat dans un fichier que nous comparerons avec le résultat obtenu avec le programme de démo. Votre programme devra veiller à ne rien afficher de plus que les résultats demandés dans les formats précisés en section 3.

3. Spécifications détaillées

3.1 Buts des tâches

Cette section approfondit les éléments fournis en section 1.2 en cherchant à identifier des structures de données et des fonctions pouvant être ré-utilisées. La structure générale du programme est séquentielle avec une suite de 4 tâches (Fig 2) ; le programme peut se terminer prématurément en cas de détection d'erreur (un message standardisé est alors affiché).

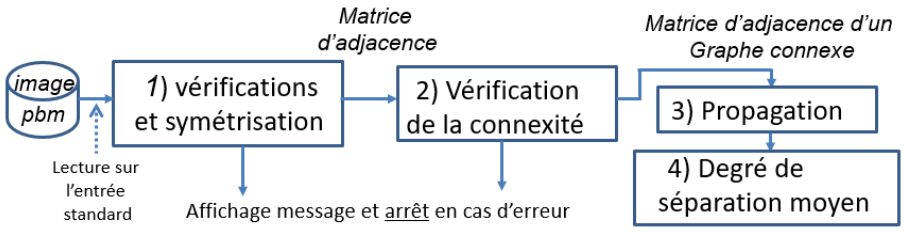


Figure 2 : Suite des tâches à réaliser par le programme

La section 1.1 a déjà défini la *matrice d'adjacence* **matAdj** de taille **n x n**. Les tâches 2 à 4 peuvent avoir besoin de mémoriser une ou plusieurs informations en plus de la matrice pour progresser correctement dans le traitement du graphe.

La description suivante est en phase avec le cours[1] : si on désire mémoriser une donnée pour chacun des n nœuds du graphe il suffit de créer un tableau de n éléments du type de la donnée ; par exemple un tableau de n booléens, que l'on appellera ici **visited**, est initialisé à FAUX puis chaque booléen est mis à VRAI pendant la tâche 2 quand le nœud correspondant a été traité.

3.1.1 Buts et ébauche de méthode : Tâche1

La **tâche 1** lit les données entrées au clavier pour initialiser **matAdj** (détails techniques en section 3.2.1). Cette étape de lecture de données au clavier ne présente pas de difficulté algorithmique mais requiert l'écriture d'une ou de plusieurs fonctions conformément au principe d'abstraction.

3.1.2 Buts et ébauche de méthode : Tâche2

La **tâche 2** consiste à vérifier que le graphe correspondant à **matAdj** est ***connexe***, c'est-à-dire qu'il est d'un seul tenant (Fig1c) ou encore qu'il n'y a pas de nœud isolé du reste du graphe. La figure 3a montre la matrice d'adjacence d'un graphe *non-connexe* car le nœud 1 est isolé et donc inatteignable à partir d'un autre nœud (Fig 3b).

Ebauche de l'algorithme de vérification récursive de la connexité :

Pour vérifier la propriété de connexité, il suffit de parcourir le graphe en commençant à partir de n'importe quel nœud, par exemple celui d'indice **0**. Ce nœud de départ est *marqué comme étant traité* en faisant passer le booléen d'indice **0** de FAUX à VRAI dans le tableau **visited** déjà mentionné plus haut. Ensuite l'algorithme va propager récursivement ce marquage aux nœuds voisins du premier nœud.

L'algorithme étant récursif, ce qui vient d'être décrit pour le nœud 0 doit pouvoir être fait pour n'importe quel nœud dont l'indice **i** est reçu en paramètre de l'algorithme. Cet indice **i** permet de mettre à jour le booléen correspondant dans le tableau **visited** pour indiquer que le noeud est traité.

La clef du parcours du graphe est d'utiliser la matrice d'adjacence **matAdj** pour savoir qui sont les voisins d'un nœud **i** en examinant la ligne **i** de **matAdj** ; si un élément **(i,j)** est VRAI alors le nœud **j** est un voisin du nœud **i**.

Le critère d'arrêt est de rencontrer un nœud déjà traité (i.e. son booléen dans le tableau **visited** est VRAI). Ainsi l'algorithme ne va *pas refaire le traitement de ce noeud une seconde fois.* De proche en proche, les nœuds du graphe sont visités jusqu'à ce qu'il n'en reste plus à traiter.

Si après l'exécution de l'algorithme il reste au moins un nœud non-visité, alors le graphe n'est pas connexe : le programme doit s'arrêter et afficher le message d'erreur standard (section 4.4.1). Grâce à cette vérification nous avons la garantie que la tâche 3 va travailler avec un graphe connexe.

[1] Les étudiants maitrisant déjà le langage C++ peuvent mettre en œuvre des structures là où ils le jugent nécessaire.

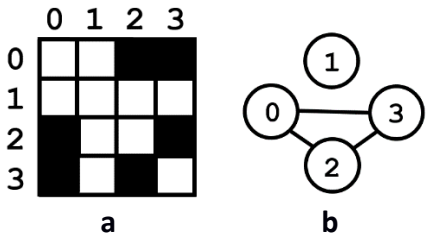


Figure 3 : exemple de matrice d'adjacence (a) produisant un graphe non-connexe (b) car il n'est pas d'un seul tenant.

3.1.3 Buts et ébauche de méthode : Tâche3

La **tâche 3**, consiste à simuler la ***propagation*** d'une information à partir du nœud 0 en *affichant sur une même ligne* tous les indices triés des nœuds qui *reçoivent l'information en même temps* parce qu'ils sont *à la même distance* du nœud de départ. La distance utilisé pour le graphe est appelée le ***degré de séparation*** : deux nœuds voisins ont un degré de séparation de ***une unité***. Pour toute paire de noeuds cette distance sera donc toujours un nombre entier.

Pour le graphe de la Fig1c la première étape de la propagation identifie tous les nœuds ayant un seul degré de séparation avec le nœud de départ 0 ; on obtient ainsi les noeuds 2 et 3. Ensuite pour l'étape suivante on identifie les nœuds ayant un degré de séparation supplémentaire ; on obtient le nœud 1. La tâche 3 s'arrête là. Le résultat de la tâche 3 pour la Fig1 donne :

0
2 3
1

Ebauche de l'algorithme récursif :

Pour cette tâche aussi il faut mettre à jour un booléen de visite de chaque nœud ; le tableau **visited** sera donc à nouveau utile. Cependant cela n'est pas suffisant ; il faut aussi mémoriser l'*ensemble des nœuds qui ont le même degré de séparation*. En effet, à chaque étape on doit travailler avec deux ensembles :

> **depart** : Les nœuds de départ à partir desquels on va avancer d'un degré de séparation
> **arrivee** : Les nœuds (pas encore traités) distants de un seul degré de séparation des nœuds de départ

Si on reprend l'exemple de la figure 1, au début l'ensemble **depart** ne contient que le nœud 0. Son traitement permet de construire l'ensemble **arrivee** contenant les nœuds distants d'un seul degré de séparation, c'est-à-dire 2 et 3.

Ensuite, l'étape suivante récursive consiste à utiliser l'ensemble **arrivee** comme ensemble **depart** et à construire un nouvel ensemble **arrivee**. Pour cet exemple, le nœud 2 ne contribue pas à ajouter de nouveaux nœuds car le nœud 3 a déjà été visité. Le nœud 3 ajoute le nœud 1 qui sera l'unique nœud de **arrivee** de la seconde étape.

Lors du dernier appel récursif avec cet ensemble comme départ on détecte alors que l'unique voisin de 1 a déjà été traité ce qui termine la récursion.

Comme pour la tâche 2 le critère d'arrêt est la détection qu'un nœud a déjà été traité.

3.1.4 Buts et ébauche de méthode : Tâche4

La **tâche 4** demande simplement d'afficher le ***degré de séparation moyen du graphe***. Cela implique deux niveaux de calculs :

- Niveau nœud individuel : chaque nœud, pris comme nœud de départ, permet de calculer les (**n-1**) degrés de séparation entre ce nœud et les n-1 autres nœuds ; cela permet de calculer leur moyenne appelée le ***degré de séparation moyen du nœud***.

- Niveau graphe : il s'agit simplement de la moyenne des degrés de séparation moyens des **n** nœuds.

Cette information nous renseigne sur la cohésion du graphe et la rapidité de transmission d'une information[2] ; par exemple, pour un graphe totalement connecté pour lequel chaque nœud est relié à tous les autres nœuds sa valeur vaut 1.

3.2 Eléments techniques et ACTIONS proposées

Les sous-sections « ACTIONS » sont seulement des suggestions pour organiser votre travail.

3.2.1 Tâche1 : Lecture sur l'entrée standard pour construire le graphe non-orienté

3.2.1.1 utilisation du format d'image **pbm** en entrée :

Comme discuté en section 2.4 sur la méthode de *redirection*, il est préférable de mémoriser les données à fournir en lecture dans un fichier texte pour éviter de se tromper en les entrant manuellement à chaque exécution.

Nous voulons utiliser le format d'image **pbm**[3] (*portable bitmap file format*) comme format des données en entrée de notre projet. Ce format d'image permet de visualiser une matrice d'adjacence avec le logiciel imageMagick (les commandes sont détaillées en complément de la série5) ; une valeur à 1 est affichée avec un pixel noir tandis que les valeurs à zéro restent en blanc dans l'image (comme pour notre illustration de la Fig1b).

L'entrée du programme est donc une image en format **pbm** (sans ligne de commentaire) qui doit être entrée au clavier ligne par ligne. Votre programme doit mémoriser les informations utiles et construire la matrice d'adjacence du graphe lors de cette tâche de lecture. Il effectue aussi quelques vérifications pouvant produire un message d'erreur et l'arrêt du programme.

```
P1
3 4
0 1 1
1 0 0
1 0 0
0 1 0
```

Figure 4 : contenu d'un fichier image **pbm** pour une image de 3 colonnes et de 4 lignes

Le choix de lire un fichier en format **pbm** en entrée du programme nous impose les contraintes

[2] ou d'une maladie si les nœuds sont des personnes et les arêtes expriment une proximité physique sans mesure barrière
[3] source : http://fr.wikipedia.org/wiki/Portable_pixmap

suivantes liées à ce format, comme illustré dans la figure 4 ci-dessus:

- La première ligne du format contient les deux caractères alphanumériques « **P1** » pour indiquer qu'il s'agit d'un format **pbm**.
- La ligne suivante donne deux nombres entiers strictement positifs : le nombre de ***colonnes*** suivi par le nombre de ***lignes*** (respectivement 3 et 4 dans la figure 4).
- Les lignes suivantes donnent les valeurs 0 (affiché en blanc) et 1 (affiché en noir) séparées par un espace. Chaque ligne de l'image est décrite sur une ligne de texte.

3.2.1.2 Vérifications à effectuer sur les données de l'image **pbm** fournie en entrée :
Le programme doit effectuer les vérifications suivantes sur les données en entrée :

- lire les deux caractères indiquant le code du fichier et vérifier que l'on reçoit bien les deux caractères alphanumériques **P** puis **1**.
- Vérifier que le nombre de lignes est égal au nombre de colonnes car la matrice d'adjacence doit être carrée.
- Vérifier que les valeurs de l'image sont seulement des 0 et des 1.

Dès la ***première*** erreur détectée sur l'entrée, le programme doit afficher le **message d'erreur défini dans le fichier error.txt** qui est disponible avec les fichiers de test sur le site de l'autograder (section 4.4.1). Recopiez les quelques lignes de **error.txt** au début de votre fichier source, juste avant la liste des prototypes de fonctions. Voici un exemple de la syntaxe à adopter pour afficher un message ; ici supposons qu'on ait trouvé que le nombre de lignes n'est pas égal au nombre de colonnes. L'instruction d'affichage doit être :

```
cout << MATRIX_SQUARE << endl;
```

Ensuite on peut quitter immédiatement le programme avec l'appel : `exit(0) ;`

3.2.1.3 Transformation à effectuer pour construire la matrice d'adjacence:
Cette étape consiste à appliquer deux règles très simples pour obtenir une matrice d'adjacence symétrique représentant un graphe non-orienté :

- Les éléments de la diagonale sont mis à zéro car pour ce projet un nœud n'a pas d'arête qui reboucle sur lui-même
- Si un élément (i, j) vaut 1 alors l'élément (j, i) est aussi mis à 1

3.2.1.4 Représentation **interne** de l'image **pbm** et de la matrice d'adjacence :
Pour représenter l'image **pbm** et la matrice d'adjacence nous avons besoin de la notion de *tableau à deux indices* (semaine 7) et plus particulièrement de la notion de **vector** car la taille **n** n'est pas connue au moment de l'écriture du programme. Voici comment on peut déclarer la matrice d'adjacence **matAdj**.

```
vector<vector<bool>> matAdj(n,vector<bool>(n)); // init à false
```

Dans la suite de la donnée nous respectons la convention des tableaux pour accéder à un élément de **matAdj**[4] : l'indice de gauche est celui indiquant la ligne et celui de droite indique la colonne: `matAdj[i][j]`.

[4] Pour les étudiants plus avancés par rapport au cours : dans le cas particulier d'un **vector** de **bool** le langage C++ optimise l'espace mémoire en stockant 8 valeurs **bool** par octet de la mémoire. De ce fait on ne peut *pas* utiliser de pointeur sur un élément de **matAdj**.

3.2.1.5 ACTIONS 1 : tests de l'initialisation

Concevez une première version du programme qui gère seulement **l'initialisation** Dans ce premier exercice d'abstraction des actions à mettre au point, rappelez-vous que la fonction principale main() joue seulement le role de « table des matières » et est constituée essentiellement d'appels de fonctions.

Concernant les déclarations de variables, seules les variables stratégiques, telles que **n** ou **matAdj,** ont le droit d'exister au niveau de main(). Les variables utilisées temporairement par une tâche doivent être définies localement dans les fonctions qui gèrent cette tâche. Une fonction peut modifier un paramètre passé par référence ou renvoyer une valeur lue.

Testez cette première version pour détecter les erreurs de format (section 3.2.2) en tirant parti de la redirection des entrées-sorties (série 6) pour couvrir une grande variété de cas. Créez vos fichiers tests avec geany ou avec l'outil Propagation – Interface mis à disposition sur moodle.

Optionnel : cette version de votre programme pourrait afficher dans le terminal l'état de vos données après cette phase d'initialisation. Si vous faite cet affichage en respectant le format pbm vous pouvez rediriger la sortie vers un fichier pbm qui est visualisable avec l'outil Propagation – Interface. Vous pouvez alors vérifier que c'est bien le même graphe que celui obtenu avec le fichier test pbm utilisé en entrée.

3.2.2 Tâche 2 : Vérification que le graphe est connexe

Cette tâche utilise la matrice d'adjacence **matAdj** construite par la tâche 1. Une seule structure de donnée supplémentaire est nécessaire pour le bon fonctionnement de la tâche 2 : le tableau des booléens **visited**. Il peut être mis en œuvre très simplement à l'aide d'un **vector** de n booléens initialisé avec FAUX.

```
vector<bool> visited(n);
```

Ce vector **visited** est une variable *interne* à la tâche et doit donc être défini localement dans la fonction responsable de la tâche 3.

3.2.2.1 ACTIONS 2 : test de la tâche 2

Cette tâche pourrait être testée indépendamment de la tâche 1 en passant une matrice d'adjacence prédéfinie en paramètre à la fonction responsable de la tâche 3.

Sachant que la matrice d'adjacence donne les indices des voisins dans l'ordre croissant, il est possible de simuler sur le papier l'ordre des nœuds qui vont être traités par l'algorithme. Par exemple pour la Fig1, nous avons le nœud de départ qui est 0 puis on traite d'abord 2 ; celui-ci traite alors son unique voisin le nœud 3 qui traite son unique voisin non déjà traité 1. Pour le vérifier pendant la phase de mise au point il est recommandé de faire afficher un bref message à chaque appel récursif, par exemple l'indice du nœud traité. Il est important de disposer d'une trace qui permet d'identifier les erreurs qui ne vont pas manquer d'apparaître. Il faudra bien sûr enlever ce code une fois que le programme fonctionne correctement.

Faites ces vérifications sur plusieurs petits graphes avec des configurations très différentes

dont au moins quelques-unes non connexes.

3.2.3 Tâche 3 : Propagation d'information à partir du premier nœud

Chacun des 2 tableaux de nœuds utilisés à chaque étape de la propagation, **depart** et **arrivee,** peut être modélisé par un **vector** contenant les indices entiers des noeuds.

Concernant l'opération de tri sur les nœuds d'**arrivee** avant affichage, on peut écrire soi-même ce tri ou utiliser la fonction disponible en C++ (`#include <algorithm>`).

3.2.3.1 ACTIONS 3 : test de la tâche 3

Lancez-vous dans la tâche 3 seulement après avoir validé la tâche 2 qui est plus simple mais qui est néanmoins récursive.

N'hésitez pas à ajouter plus d'affichages temporaires (contenus de **depart**, de **arrivee** avant et après le tri) pour la phase de mise au point. De la même manière que pour la tâche précédente, prédisez ce qui doit être affiché et comparez.

3.2.4 Tâche 4 : Calcul du degré de séparation moyen du graphe

Une ré-utilisation partielle (sans tri, ni affichage) de ce qui est fait pour la tâche 3 est pertinent ici puisque cette tâche permet de construire le degré de séparation moyen du nœud 0.

Le calcul du degré de séparation au niveau du graphe n'est que l'affaire d'une boucle de calcul de moyenne.

3.2.4.1 ACTIONS 2 : test de la tâche 4

Comme déjà indiqué, un graphe totalement connecté produit une valeur de 1. Faites d'autres tests dont le cas limite d'un graphe où tous les nœuds sont comme des perles sur un collier.

4. Mise en œuvre en langage C++ (*sera compilé avec* `-std=c++11`)

Au sem1 nous demandons que le code soit écrit dans un seul fichier source.

4.1 Clarté et structuration du code avec des fonctions

La clarté de votre code sera prise en compte pour le rendu. Un examen manuel de votre code source sera effectué par une personne chargée d'évaluer le respect des conventions de programmation utilisées dans ce cours. Par souci d'efficacité seuls les codes indiqués dans nos conventions vous seront communiqués avec les numéros des lignes concernées dans votre fichier source.

4.2 Variables locales ou globales ?

Toutes les ***variables*** ou ***tableaux*** utilisés pour ce projet seront déclarés ***localement*** et transmis en paramètres aux fonctions **seulement** si c'est nécessaire. Aucune exception ne sera admise.

4.2.1 Qu'en est-il des ***constantes*** ?

Voici les règles que nous nous donnons, en conformité avec nos conventions de programmation :

- Si une constante n'est utilisée qu'à l'intérieur d'*une seule fonction*, alors la déclaration d'une variable avec **constexpr** peut être faite dans cette fonction ou globalement.

- Si la constante est utilisée dans *plus d'une fonction*, alors la déclaration d'une variable avec **constexpr** *doit* être faite de manière globale, en début de fichier comme décrit dans les conventions de programmation.
- L'alternative de la déclaration de symboles avec #define est autorisée pour des constantes utilisées dans plusieurs fonctions. Elles sont aussi déclarées en début de fichier comme décrit dans les conventions de programmation.

4.3 Précision sur les nombres à virgule : ajouter en début de programme #include <iomanip>
Tous les nombres à virgule nécessaires pour ce projet doivent utiliser le type **`double`**. Pour l'affichage demandé pour la **tâche 4**, on utilisera ces instructions C++ :

```
cout << setprecision(6)   // 6 chiffres à droite de la virgule
cout << fixed
```

4.4 Fichiers test :
La donnée identifie plusieurs phases de tests (sections ACTION) pour valider votre programme. Nous vous fournissons 13 fichiers de tests téléchargeables depuis le site de l'autograder (section suivante).

A vous de compléter ces fichiers avec vos propres fichiers plus élaborés ou juste différents. Il suffit d'ouvrir geany pour écrire les données et de sauvegarder avec l'extension **.txt** .

Cette année un autre outil est fourni pour créer des fichiers de test pour un graphe. Il s'agit de l'application **Propagation – Interface** de visualisation et d'édition visible sur moodle.

4.4.1 Vérification anticipée de l'exécution de votre projet avec *l'autograder*

Cette année un outil de type *autograder* a été mis au point pour vous permettre d'évaluer, avant la date du rendu, si votre projet s'exécute comme demandé sur l'ensemble des fichiers de test publics et non-publics qui seront utilisés pour la notation du projet. Le lien et son mode d'emploi sont visibles sur moodle.

Le site web de l'autograder fournit des liens utiles :
- ensemble des fichiers de test publics
- messages d'erreurs standardisés
- site officiel du téléversement final sur moodle.

Attention : l'autograder efface systématiquement tous les fichiers archive qui lui sont soumis. De ce fait il ne peut pas constituer une preuve que votre projet a été effectué. Votre projet devra impérativement être téléversé sur moodle comme expliqué dans la section suivante.

5. Rendus :
Il faudra téléverser (upload) un fichier archive à l'aide du lien qui sera mis à disposition sur **moodle** (Topic 12). Ce fichier archive doit seulement contenir :

1) **Votre unique fichier source** dont le nom est donné par votre numéro SCIPER et l'extension est .cc. Par exemple pour Monsieur X de numéro SCIPER 123456, le nom de fichier source est **123456.cc**.Vous êtes responsables de vérifier que l'upload s'est bien passé en le

téléchargeant (download) dans votre compte et en examinant que tout est bien présent. Vous pouvez toujours faire un nouvel upload jusqu'à la date limite.

- il est autorisé d'avoir au maximum 87 caractères par ligne et 40 lignes par fonction (conventions de programmation).

2) **Un rapport** en format **pdf** d'au maximum **2 pages** écrit avec un traitement de texte.

Vous devez avoir téléversé le fichier archive pour le **dimanche 6 décembre à 23h59**.

5.1 Rapport

Le Rapport ne contient PAS de page de titre, ni de table des matières
Le Rapport contient :

a) Résultat de la phase **d'analyse** (max une demi-page, police de taille 11) :
Décrire l'organisation générale du programme en faisant ressortir, avec concision (Fig 5), la mise en oeuvre des principes *d'abstraction* et de *ré-utilisation* dans votre projet. Ne donnez pas la liste exhaustive de vos fonctions ; concentrez-vous sur la décomposition des tâches en fonctions ; préciser en particulier si vous avez tiré parti de la ré-utilisation d'une fonction à plusieurs endroits du code.

b) **Pseudocode** de votre **algorithme** effectuant la **tâche 3** décrite dans la section **3** (PAS de code source). Le pseudocode doit être au plus sur une demi-page (soit sur la p 1 ou la p 2 mais pas entre les deux).

c) Estimation de l'**ordre de complexité** de la **tâche 4** pour **N** nœuds et un nombre total de **L** arêtes à l'échelle du graphe entier. Pour vous aider à répondre à cette question, nous posons que l'ordre de complexité de l'algorithme récursif qui détermine les (n-1) degrés de séparation de un seul nœud de départ est en $O(N^2)$ avec la matrice d'adjacence.

d) Fournir le temps d'exécution « **user** » obtenu avec la commande **time** (série 6) sur les 4 fichiers publics de test suivants : t07, t08, t09 et t10 ; quel type de croissance du temps calcul observez-vous ? Est-ce cohérent avec votre réponse à la question précédente ? Si non, à quoi attribuez-vous cette différence ?

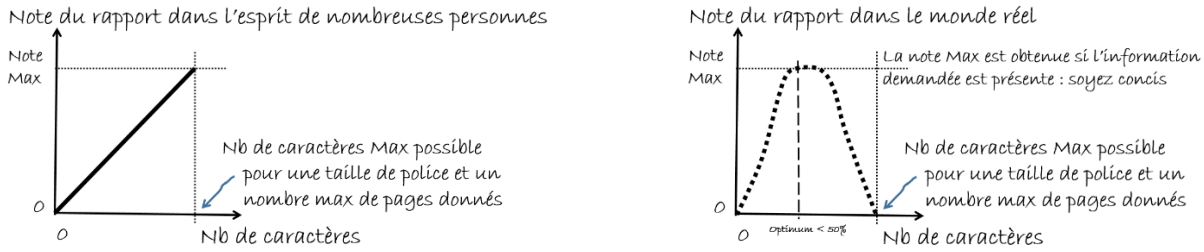


Figure 5 : un mauvais rapport dilue l'information utile jusqu'à atteindre le nombre maximum de caractères possibles sur la page (fig gauche) : en fait ce type de rapport sera pénalisé parce qu'il est peu lisible (fig droite) ; un bon rapport est celui qui fournit les informations demandées avec concision avec une mise en page aérée et lisible

6. Barème indicatif (12pts):

Ce barème est indicatif et provisoire. Il sera éventuellement modifié par la suite.

(2pt) rapport : soyez concis
(4pt) Lisibilité, structuration du code et conventions de programmation du cours
(2pt) votre programme fonctionne correctement avec les fichiers publics
(4pt) votre programme fonctionne correctement avec les fichiers non-publics

Dernier rappel : le projet d'automne est INDIVIDUEL. Le détecteur de plagiat sera utilisé selon les recommandations du SAC.