**République Française**
**Ministère de l'Enseignement Supérieur et de la Recherche Scientifique**

**Université d'Orléans**
**UFR Sciences**
**Département d'Informatique**

# *Mémoire Intermédiaire*

# Librairie de manipulation de Grands Graphes

**Présenté par :**
Marco Da Cunha Fernandes - Clément Desmazeaud - Éléonore Gédéon -
Nicolas Opériol-Gerbal - Sullivan Perrin

**Encadrement :**
Nicolas Dugué - Anthony Perez

Année universitaire 2014-2015

# Table des matières

# 1 Résumé du projet

Dans le cadre de recherches sur les réseaux, tout particulièrement les réseaux sociaux, nous sommes rapidement amenés à traiter des quantités importantes de données. Les données provenant des réseaux peuvent être représentées sous forme de graphes, nous pouvons donc faire appel à certaines informations que la théorie des graphes nous permet d'obtenir. Par exemple, la taille d'un graphe en nombre de noeuds ou de liens. Ou encore utiliser des algorithmes permettant de parcourir les différents noeuds d'un graphe.

Pour aller plus loin, la théorie des réseaux complexes permet de comprendre comment de tels réseaux se forment, grandissent, quelles caractéristiques ils partagent. Ceci permet par exemple de comprendre comment l'information ou une infection se propage dans un réseau.

Afin d'obtenir de telles informations, il est nécessaire d'avoir à sa disposition un ensemble d'outils informatiques permettant d'y répondre de manière efficace, et ce dans un temps acceptable. Notre mission est donc d'analyser les solutions existantes de traitements de grands graphes, et de mettre au point une librairie répondant à cette problématique.

Le but est donc d'organiser les programmes existants en une librairie maintenable, évolutive et facilement utilisable, sans oublier d'enrichir la librairie d'algorithmes classiques de la théorie des graphes, tout en portant un soin particulier à la performance.

# 2 Introduction

Lors de la dernière décennie nous avons vu émerger de nombreux services de réseaux sociaux, c'est à dire des environnements collaboratifs centrés sur l'individu. En effet, il existe une multitude de réseaux sociaux, certains destinés à regrouper des amis de la vie réelle, comme Facebook. D'autres aident à trouver des relations professionnels (emplois, liens commerciaux), amicales, amoureuses, ou permettent la découverte et le partage de contenus.

L'ensemble de ces plateformes a vu son nombre d'utilisateurs exploser, comme c'est le cas pour Twitter (permettant le partage, la découverte, et le débat d'informations) qui est passé de 200 millions de comptes en Avril 2011 à 500 millions en Octobre 2012.


FIGURE 1 – Diffusion du mème #tcot sur les réseaux sociaux

L'analyse de ces données permet par exemple d'améliorer les résultats des moteurs de recherche, ou encore d'étudier les comportements sociaux. D'autre part, il est intéressant de connaître les comportements des utilisateurs les plus influents, dans le but de comprendre la raison de leur popularité, comprendre ce que leur apporte cette popularité et quels sont leurs buts (en politique par exemple).

Mais les réseaux ne sont pas limités aux réseaux sociaux, et peuvent faire leur apparition dans de multiples domaines : les transports, l'organisation hiérarchique d'une entreprise, les espaces géographiques, les connexions entres les neurones d'un cerveau, ou encore les interactions entre les protéines d'un composé chimique ou biologique. Tant de domaines qui cherchent pourtant des réponses communes.

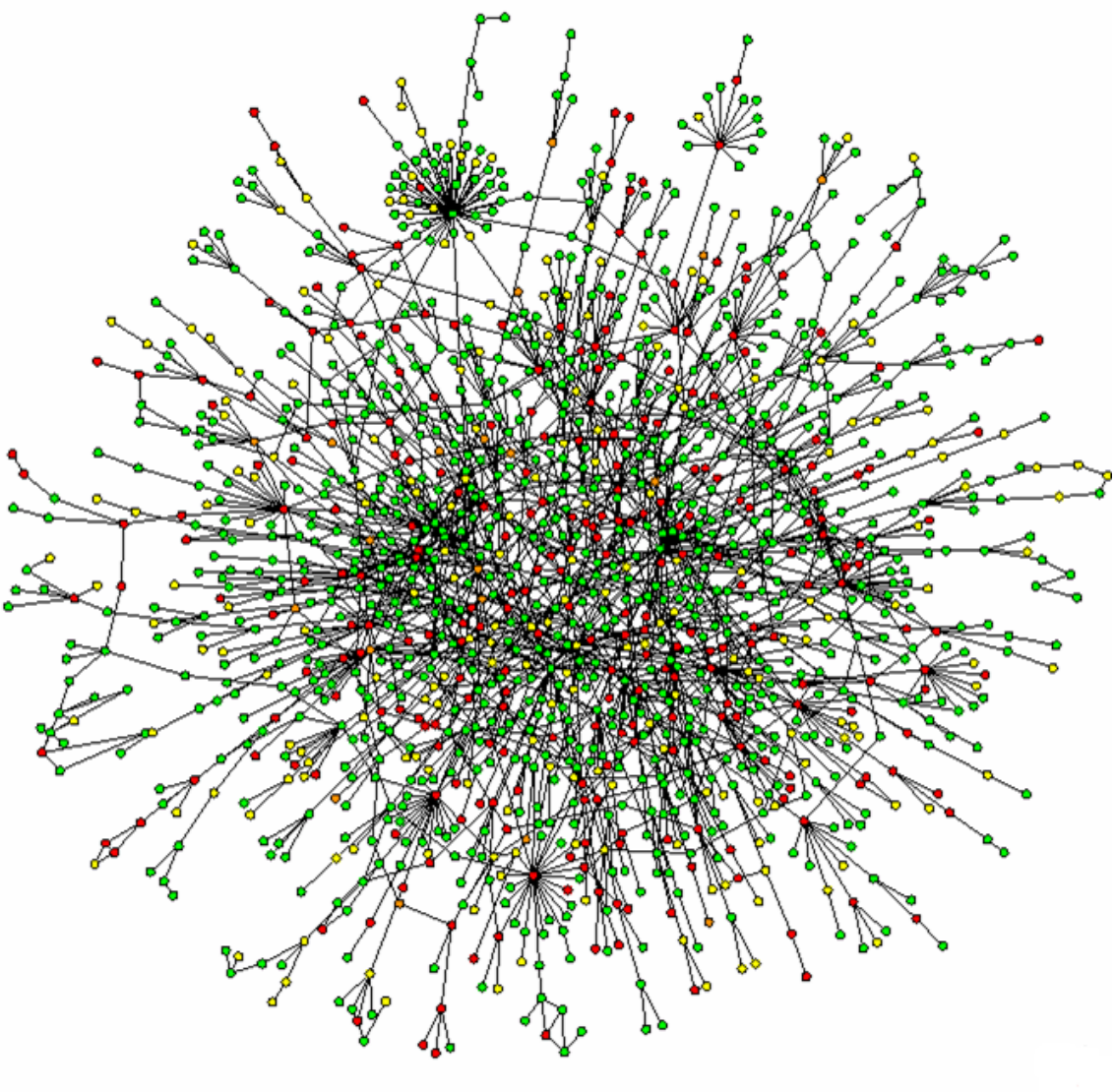

FIGURE 2 – Interactions de protéines dans le Saccharomyces cerevisiae, une levure

Dans un premier temps, il est nécessaire de définir quelques concepts liés à la représentation de réseaux sous forme de graphes. Un graphe est un ensemble de points, dont certaines paires sont directement reliées par un (ou plusieurs) lien(s). Ces liens peuvent être orientés, c'est-à-dire qu'un lien entre deux points u et v relie soit u vers v, soit v vers u : dans ce cas, le graphe est dit orienté. Sinon, les liens sont symétriques, et le graphe est dit non orienté.

Afin de représenter au mieux les données, nous pouvons placer des poids sur les noeuds ou sur les arêtes, par exemple : dans un graphe représentant des villes, on pourrait mettre une valeur en kilomètres sur les liens séparants 2 villes ; ou encore affecter le nombre d'habitants d'une ville à chaque noeud correspondant. Le degré d'un noeud correspond au nombre de ses voisins. On définit comme voisins d'un noeud N l'ensemble des noeuds ayant un lien direct avec N.

Grâce à cela, il est déjà possible de répondre à certaines questions, comme par exemple trouver le chemin le plus court entre deux noeuds. Ou encore, de répondre à des problèmes comme le voyageur de commerce : on se place dans un graphe à n noeuds où l'on connaît les distances séparant chaque noeuds, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque noeuds et revienne au noeud de départ. L'ensemble des algorithmes de la théorie des graphes nous sert donc comme base pour nos algorithmes d'études de réseaux.

Dans un second temps, la théorie des réseaux complexes définit plusieurs mesures comme le nombre de sauts (utilisé dans les réseaux internet pour compter le nombre d'intermédiaire pour aller d'un point à un autre), ou encore la distribution des degrés, c'est à dire la probabilité qu'un noeud, choisi de manière aléatoire, comporte un certain degré. Le degré de séparation est également une notion importante qui définit le nombre de noeuds minimum pour aller d'un noeud à un autre.

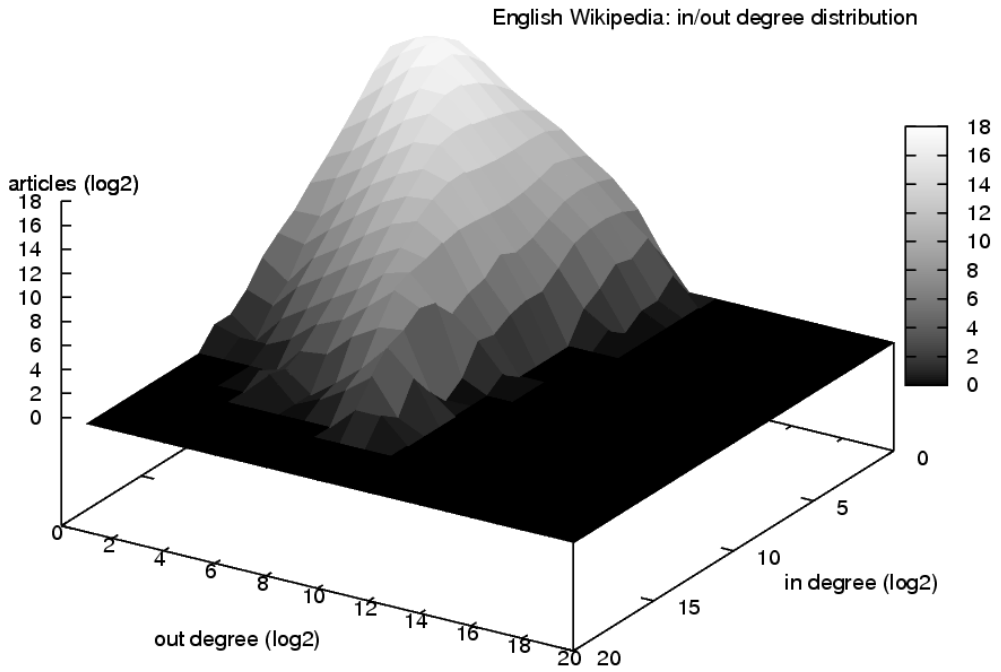


FIGURE 3 – Exemple de la représentation de la distribution des degrés d'un graphe

Nous avons vu précédemment qu'avec la théorie des réseaux complexes, on recherche à comprendre comment les réseaux se forment, grandissent, et quelles caractéristiques ils partagent. Nous allons donc décrire plusieurs caractéristiques que les réseaux peuvent avoir en commun.

Un réseau est appelé small-world, en référence au phénomène du même nom qui indique que dans un graphe représentant un réseau social, il y a au plus $k$ (avec $k$ petit, souvent inférieur à $10$) degrés de séparation entre deux noeuds. L'idée est donc que, dans un réseau small-world, le degré de séparation entre deux noeuds est très petit (mathématiquement, il augmente de manière logarithmique en fonction de la taille du réseau).

Il existe également les réseaux appelés scale-free, dont la distribution des degrés suit une loi de puissance. Une loi de puissance est une relation entre deux quantités x et y qui peut s'écrire de la façon suivante : $y = ax^k$ De part la loi de puissance et son principe d'invariance d'échelle, les réseaux scale-free suivent cette propriété de loi de puissance peu importe l'échelle du graphe (nombre de noeuds/liens).

Nous pouvons être amenés à vouloir partitionner un ensemble d'utilisateurs, afin d'observer les groupes d'individus ayant une liaison forte (ces groupes sont appelés communautés [1, 2]). Grâce à ce que nous avons vu, nous pouvons par exemple essayer de savoir quels noeuds du réseau sont importants, influents, centraux, ou connaitre quel contenu recommander à quelqu'un dont on connaît les goûts de ses amis, ou de sa communauté.

# 3 Analyse de l'existant

Actuellement, une partie des algorithmes que nous allons reprendre a déjà été implémentée, notre travail va donc consister à adapter ces algorithmes pour notre structure de données, et à implémenter nous-mêmes les algorithmes restants. Les algorithmes que nous allons devoir traiter sont les suivants :

**Jaccard :** Il s'agit d'étudier le jaccard [3] de deux ensembles de sommets A et B, c'est à dire le cardinal de l'intersection de A et B, divisé par le cardinal de l'union de A et B. A est l'ensemble des voisins sortants et B l'ensemble des voisins entrants.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Cela permet de voir les similitudes en deux ensembles. Par exemple, on peut comparer le voisinage de deux utilisateurs et déterminer leur ressemblance, ou encore comparer les communautés résultantes d'algorithmes différents afin de voir leur correspondance. Il existe aussi OverlapIndex qui permet de mesurer le chevauchement de deux ensembles et est défini comme ceci :

$$O(A,B) = \frac{|A \cap B|}{min(|A|,|B|)}$$

**K-core [4] :** On cherche à obtenir un sous graphe maximal induit $G_{max}$, tel que chacun des degrés des nœuds de $G_{max}$ soit supérieur ou égal à K dans $G_{max}$. C'est la définition d'un sous graphe dense par l'assouplissement de la définition d'une clique (tous les sommets d'une clique sont deux-à-deux adjacents) car les cliques sont peu présentes dans les réseaux. On constate en général que dans des processus de diffusion d'informations, celle-ci se fait au sein d'un même core, du k le plus élevé vers le plus petit.

**Louvain :** Pour faire simple, on étudie avec cet algorithme la comparaison entre le fait que deux éléments soient connectés, et leur probabilité de l'être. C'est grâce à cela qu'on peut détecter des communautés. On utilise donc la modularité pour que chaque nœud appartienne à une seule communauté. La modularité est définie par :

$$\Delta = \sum_{ij} \left[ A_{ij} - \frac{d_i d_j}{2^m} \right] \delta\left(c_i, c_j\right)$$

où $A_{ij}$ est la matrice d'adjacence, $\frac{d_i d_j}{2^m}$ est la probabilité de i et j d'être connectés et $\delta(c_i, c_j)$ le fait ou non que i et j soit dans la même communauté.

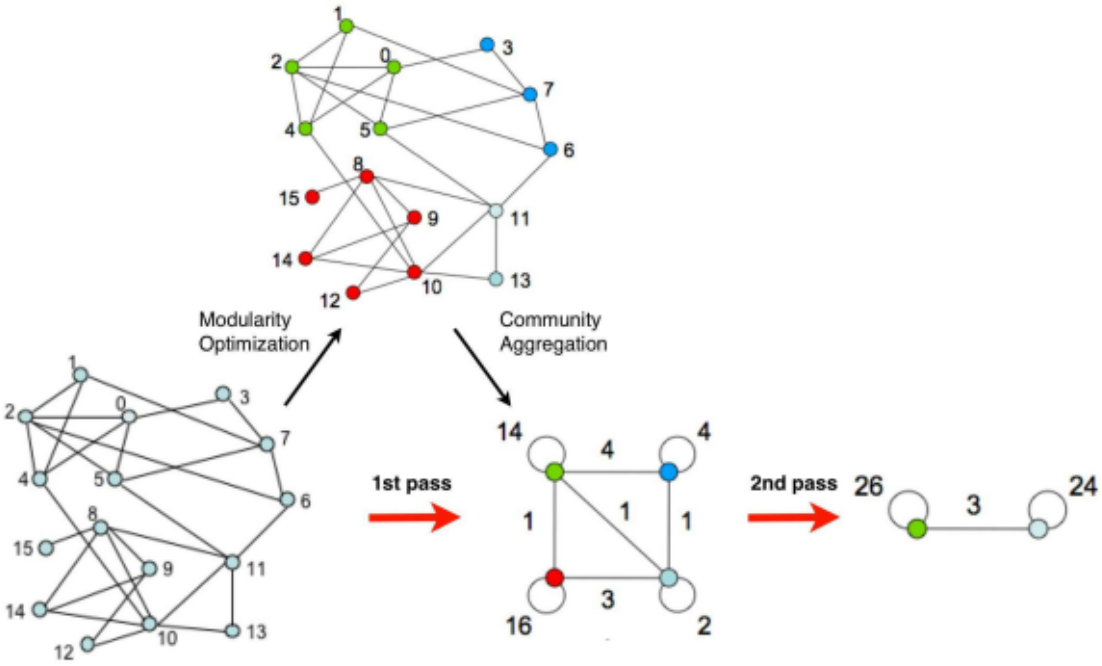


FIGURE 4 – Un exemple des différentes étapes de l'algorithme de Louvain

**Guimera et Amaral :** On se penchera ici sur le rôle que jouent les nœuds, c'est à dire s'ils sont hubs (s'ils sont incontournables dans une communauté), ce qui est définit par le z-score de leur degré interne - le nombre de noeuds de la même communauté auquel un noeud est connecté. Empiriquement, un hub est dit hub si ce score est supérieur à 2.5. On étudie également la connectivité d'un noeud aux autres communautés à travers la participation externe. Un noeud est dit connecteur s'il est connecté à beaucoup d'autres communautés. En revanche, il est dit périphérique s'il dispose de peu de connexions vers d'autres communautés que la sienne. Le z-score est défini comme suit :

$$Z_i = \frac{l_i - \overline{L}}{\sigma_L}$$

où $L = \{l_i, ..., l_n\}$ avec $l$ les liaisons du nœud actuel, $L$ la moyenne des liaisons des nœuds de la communauté et $\sigma$ l'écart type. Le z-score du degré interne est la normalisation entre les différents graphes. Il permet d'obtenir le nombre de liaisons dans la communauté proportionnellement aux liaisons des autres nœuds de la communauté.

$$Z_i = \frac{d_{int}^i - \overline{d_{int}^{c_i}}}{\sigma_{d_{int}}^{c_i}}$$

La participation externe permet de connaître la connectivité avec l'extérieur (vers 0 quand peu connecté avec l'extérieur, vers 1 quand très connecté).

$$P_i = \sum_i \left( \frac{d_c}{d_i} \right)^2$$

$d_c$ est le degré du noeud i dans sa communauté.[5]

**Dugué et Perez :** Ce dernier point est une adaptation des mesures de Guimera et Amaral. On traitera de la connectivité externe, en étudiant la diversité, l'intensité et l'hétérogénéité. La diversité correspond au z-score du nombre de communautés externes auquel le noeud est connecté. L'intensité correspond au z-score du nombre de liens vers l'extérieur. Enfin, l'hétérogénéité est l'écart type de la moyenne des liens vers les communautés extérieures, par communauté extérieure, et pour chaque nœud. [6][7]

D'autre part, les algorithmes déjà codés sont répartis de manière éparse sur différents fichiers, et il n'y a aucune cohésion entre eux ; notre travail va donc être de rassembler toutes les fonctionnalités de manière organisée, d'en faire une librairie.

# 4 Besoins non fonctionnels

Ce qui est attendu de nous pour ces travaux, c'est de débuter la constitution d'une librairie, comportant tous les algorithmes abordés précédemment, qui simplifierait leur utilisation de manière uniformisée et par conséquent simplifierait l'étude des grands graphes.

## 4.1 Choix du langage

**Description :** Notre bibliothèque devra être développée en C++.
**Priorité :** 10/10
**Justification :** Les solutions existantes sont déjà en C++. De plus, les programmes qui utilisent les solutions existantes le sont également. Enfin, cela permet un usage simple dans des programmes en C++. Par ailleurs, C++ est un language performant et OpenMP permet de paralléliser simplement du code C++ . Ainsi, par souci de continuité et de performance, C++ est choisi

## 4.2 Environnement cible

### 4.2.1 La machine Speed

**Description :** L'environnement cible de notre bibliothèque est la machine Speed qui comporte 64 coeurs et 64 Go de RAM. En outre l'utilisation de la mémoire doit être limité.
**Priorité :** 10/10
**Justification :** Avec 64Go et des réseaux qui peuvent en occuper la moitié, il est nécessaire d'avoir des algorithmes n'abusant pas de la mémoire vive.

### 4.2.2 Algorithmes non-récursif

**Description :** On tentera d'éviter au maximum l'utilisation d'algorithme récursif.
**Priorité :** 9/10
**Justification :** Les graphes peuvent comporter un très grand nombre de noeuds et de sommets (plusieurs millions de noeuds), il est donc important de ne pas utiliser d'algorithme récursif sur le nombre de noeuds par exemple. En effet, la pile d'appel de fonctions pourrait ne pas pouvoir contenir l'ensemble de nos appels récursif et ferait donc planter l'exécution de notre programme avec une erreur de type *StackOverflow*.

## 4.3 Performance

### 4.3.1 Utilisation de la mémoire vive

**Description :** L'environnement cible comporte 64 Go de RAM, son utilisation est donc limité. Néanmoins, il est nécessaire de savoir l'utiliser intelligemment.

**Priorité :** 10/10

**Justification :** En effet, bien qu'étant une ressource précieuse, la RAM peut nous aider à améliorer les performances en temps de nos algorithmes. En outre, si stocker certaines informations en mémoire (sans en abuser) lors de l'utilisation d'un algorithme permet d'augmenter sensiblement sa rapidité d'exécution, alors il est important d'étudier si le gain en temps vaut le sacrifice en mémoire.

### 4.3.2 Temps de calcul

**Description :** Les algorithmes ne doivent pas nécessiter une exécution de plus de quelques heures sur des réseaux tels que Twitter, sans compter la parallélisation.

**Priorité :** 10/10

**Justification :** Étant donné la quantité d'informations à traiter et le nombre de personnes utilisant la machine Speed, il est nécessaire de restreindre son temps d'utilisation.

## 4.4 Mise en place d'une bibliothèque dynamique

**Description :** Si possible, nous pourrions mettre en place une librairie dynamique sous forme de .DLL (Windows) et .so (Linux). [8][9][10] Dans le cas contraire, la bibliothèque sera utilisées comme les solutions existantes, c'est à dire comme un ensemble de fichier .h et .cpp (fichier template et code source de C++).

**Priorité :** 1/10

**Justification :** Une bibliothèque dynamique permet de regrouper l'ensemble d'une bibliothèque de fonctions sous un seul fichier (.dll ou .so). Permettant une distribution facilitée de la bibliothèque à une tierce personne. De plus, cela permet d'avoir une seule instance de la bibliothèque qui s'exécute et répond aux demandes en temps réel.

# 5 Besoins fonctionnels

## 5.1 Code

### 5.1.1 Pattern

**Description :** Utilisation de design pattern.
**Priorité :** 10/10
**Justification :** La librairie sur laquelle nous travaillons doit pouvoir être enrichie par la suite, et ce facilement, sans avoir à se plonger dans le code. Le principe du design pattern est de penser la conception d'un programme, l'organisation des classes le composant, de manière à pouvoir intégrer très aisément de nouveaux composants, à travers une structure du code modulaire.

### 5.1.2 Parallélisation

**Description :** Afin d'améliorer le temps d'exécution de nos algorithmes, nous utiliserons OpenMP [11] sur les algorithmes les plus gourmands en temps.
**Priorité :** 9/10
**Justification :** En effet, l'environnement cible dispose de 64 coeurs, il est donc très intéressant de paralléliser nos programmes et de bénéficier d'un temps de calcul beaucoup plus réduit. OpenMP [11] permet une parallélisation simple de programmes grâce à l'utilisation de directives (lignes de code qui seront remplacées à la compilation pour transformer certains blocs en blocs parallèles, avec des paramètres définis).

## 5.2 Algorithmes

### 5.2.1 BFS

**Description :** Breadth First Search, ou algorithme de parcours en largeur. [12]
**Priorité :** 10/10
**Justification :** Cet algorithme permet le parcours d'un graphe de manière itérative, à l'aide d'une structure de type file. Nos algorithmes suivants requièrent pour la plupart à un moment ou à un autre de parcourir le graphe à étudier, cet algorithme est donc fondamental.

### 5.2.2 Jaccard

**Description :** Permet de connaître le jaccard [3] de deux ensembles.
**Priorité :** 9/10
**Justification :** Comme on l'a vu précédemment, le jaccard permet la comparaison de deux ensembles, cet algorithme est donc un élément essentiel puisqu'utilisé par plusieurs autres algorithmes.

### 5.2.3 K-core

**Description :** Obtention d'un sous graphe de type K-core
**Priorité :** 9/10
**Justification :** Il permet d'étudier la diffusion d'informations et les noyaux centraux qui les diffusent, et est lui aussi utilisé par d'autres algorithmes.

### 5.2.4 Louvain

**Description :** Comparaison entre le fait que deux objets soit connectés et leur probabilité de l'être.
**Priorité :** 8/10
**Justification :** Cet algorithme permet de détecter des communautés, son appartenance à la librairie est capitale, même si sa priorité n'est pas absolue du fait qu'il ne soit pas utilisé dans la plupart des autres algorithmes.

### 5.2.5 Guimera et Amaral

**Description :** Le rôle des nœuds.
**Priorité :** 7/10
**Justification :** Pour étudier un graphe, il est important de connaître l'importance de chaque nœud au sein de sa communauté comme au sein des autres communautés. Il est donc fort utile de disposer de cet algorithme dans notre librairie, mais, pour autant, peu d'algorithmes risquent d'avoir besoin de celui-ci.

### 5.2.6 Dugué et Perez

**Description :** Adaptation des mesures de Guimera et Amaral.
**Priorité :** 6/10
**Justification :** Cet algorithme servira pour traiter la connectivité externe, à travers la diversité, l'intensité et l'hétérogénéité. De même que Guimera et Amaral, cet algorithme sera beaucoup utilisé pour étudier les graphes, mais ne sera pas intégré à d'autres algorithmes.

## 5.3 Tests

**Description :** Réalisation de tests.
**Priorité :** 8/10
**Justification :** Lorsque l'on écrit un programme, il est nécéssaire de vérifier qu'il fonctionne ; pour cela on réalise des tests. Ces tests permettent de s'assurer que le code ne bug pas, qu'il réalise ce pourquoi il a été écrit, et qu"il ne fait rien d'autre que cela. Il est donc vital pour notre librairie de tester notre code.

## 5.4 Documentation

**Description :** Ajout de documentation au code.
**Priorité :** 10/10
**Justification :** De manière à rendre un code compréhensible par toutes les personnes ayant à le relire, que ce soit pour ceux qui l'écrivent au départ ou pour ceux qui le reprendront, il est important de bien le documenter : de la sorte sa compréhension sera facilitée.

## 5.5 Prototype papier

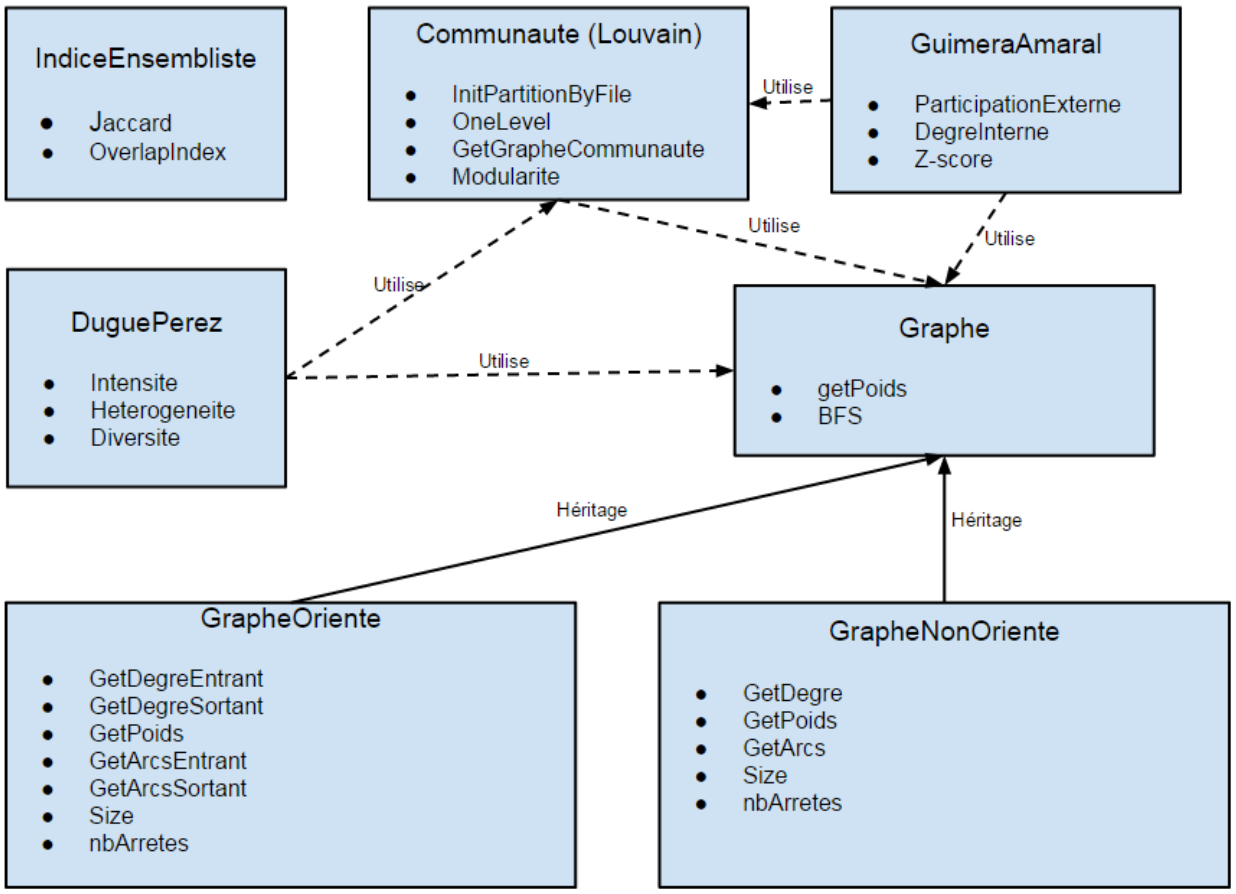


FIGURE 5 – Prototype papier de la bibliothèque

La librairie que nous allons constituer devra rassembler l'ensemble des algorithmes que nous serons à même de retranscrire depuis la solution existante, c'est à dire l'ensemble des algorithmes vus précédemment. Mais elle devra également comporter des algorithmes de base de la théorie des graphes, tels que le parcours en largeur ou en profondeur.

L'objectif est d'avoir une librairie suffisamment flexible et modulaire, afin que de nombreux algorithmes puissent être ajoutés et cela même après la fin de ce TER.

C'est pourquoi il est essentiel de mettre en place un modèle permettant de représenter un graphe, qu'il soit orienté ou non, de pouvoir convertir un graphe orienté en non-orienté. Ou encore, de pouvoir utiliser un graphe de la même manière qu'il soit orienté ou non, ce qui peut être utile dans des algorithmes qui ne se préoccupent pas de l'orientation. En outre, il est important de pouvoir spécifier si une fonction prend en paramètre uniquement un graphe orienté, non-orienté ou les deux.

Afin de faciliter l'utilisation de la librairie, nous avons effectué des recherches sur la mise en place d'une librairie dynamique. Ceci permet de pouvoir facilement partager le code présent dans la librairie pour de multiples exécutions, et ce sans avoir accès à l'implémentation de la librairie, facilitant ainsi la distribution de cette dernière auprès de tierces personnes. Pour autant, les difficultés liées aux environnements de développement multiples (Linux avec l'extension .so, et Windows avec l'extension .dll), nous amèneraient à passer du temps sur une fonctionnalité non-essentielle, temps qui pourrait être mieux investi.

C'est pour cette raison que la librairie sera constituée de multiples fichiers .cpp et .h, comme c'est le cas de la solution existante. En effet, cela permet une utilisation simple et rapide de l'ensemble de la librairie pour l'écriture d'un nouveau projet de manipulation de graphe, et ce, peu importe l'environnement de développement ou d'exécution. C'est également un moyen permettant plus de libertés quant à l'enrichissement de cette librairie.

L'utilisation de cette librairie s'effectuera par l'inclusion des bons fichier header (#include "Graphe.h"), et l'utilisation des fonctions et classes sera aussi aisé que l'utilisation de classes et fonctions d'un projet en cours. Bien évidement, aucune interface graphique n'est demandée, puisque le but est de fournir un ensemble d'outils (classes et fonctions) permettant la mise en oeuvre simple, rapide et uniforme de nouveaux projets/solutions pour le traitement des graphes importants sur super-calculateurs.

# 6 Prototypes et résultats de tests

Nous avons tout d'abord commencé par coder la structure de données permettant de gérer les grands graphes. Notre structure de données est constituée d'une classe Graphe permettant de gérer les graphes de manière globale, et nous avons ensuite deux classes GrapheOriente et GrapheNonOriente qui permettent, comme leur nom l'indique, de gérer la différence entre les graphes orientés et les graphes non orientés.

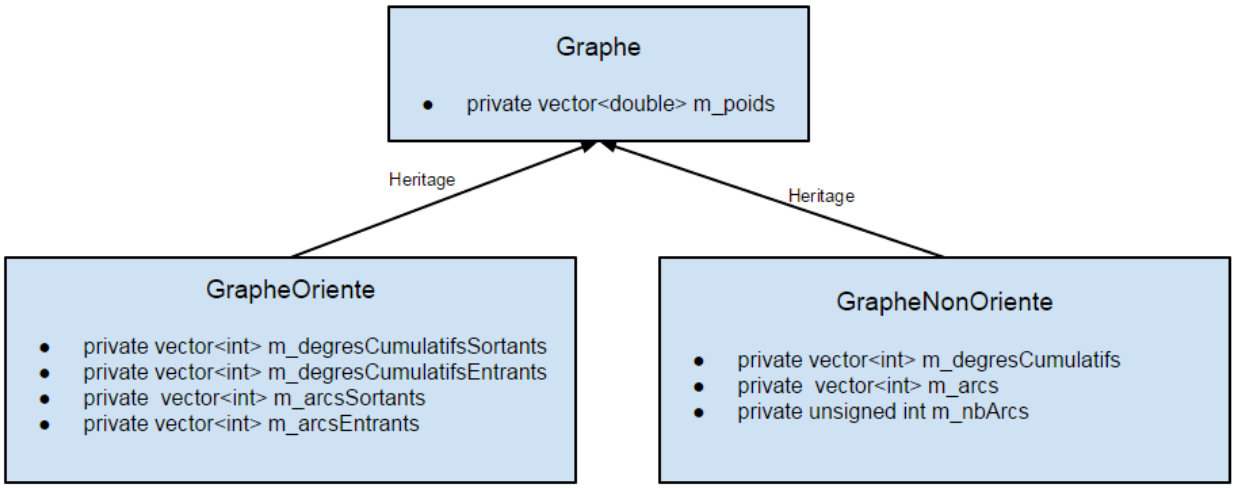


FIGURE 6 – Graphes orientés, graphes non orientés

Les deux classes de graphes sont composées de plusieurs tableaux qui permettent de stocker les arcs de chaque sommet ainsi que les degrés de ces même sommets. La principale différence entre les deux classes est que, pour un graphe orienté, les tableaux sont dédoublés afin d'avoir accès aux arcs et degrés sortants ou entrants. Le but de cette structure de données est d'avoir un accès à un élément i d'un tableau en temps constant.

Nous avons donc choisi de coder les tableaux grâce à des vectors. Nous avons opté pour cette structure car elle permet un accès en temps O(1) à un élément n du tableau, et permet également d'avoir des tableaux dynamiques, et donc de rajouter facilement de nouveaux nœuds dans le graphe. Pour le moment, le stockage des poids est encore assez incertain, nous avons essayé de le stocker dans un vector, mais les résultats ne sont pas forcément convaincants.

Une fois la structure de données créée, nous avons écrit les fonctions de base, permettant de récupérer les données du graphe, tels que les voisins d'un sommet, ou son degré, de façon constante. Ce dernier point est très important, comme nous devons travailler sur de très grands graphes. La première fonction que nous avons codé est

celle permettant de récupérer les degrés d'un sommet. Pour ce faire, nous avons le tableau des degrés cumulatifs. Ce tableau stocke par exemple à la case N le degré du sommet N ainsi que tous les degrés des sommets précédents, stockés dans le tableau. Donc le degré de la case N correspondra à la valeur en N, moins la valeur en N-1. Cela donne un accès en temps O(1), tout en conservant les valeurs précédentes, et donc en ayant le nombre total d'arcs stockés à la dernière case du tableau.

La deuxième fonction dont nous avions besoin est celle permettant de récupérer les arcs correspondant à un sommet N. Pour cela nous utilisons un deuxième tableau, qui est celui correspondant aux arcs. Afin de récupérer la liste des sommets voisins de N, nous avons besoin de connaître deux éléments. Le premier est le degré du sommet récupéré grâce à la méthode vue précédemment. Le deuxième est bien sûr la première case à regarder dans le tableau des arcs, que nous pouvons récupérer grâce à la case N-1 du tableau des degrés cumulatifs. En effet, cette case nous indique directement la case à regarder dans le tableau. Si elle est à 1, on se place à la case 1 du tableau des arcs. Il nous suffit ensuite de regarder le nombre de cases correspondant au degré du sommet N. Cela nous permet ainsi de récupérer les voisins en temps constant.

Nous pouvons voir dans ce schéma comment marche la liaison entre les deux tableaux, grâce à un exemple.

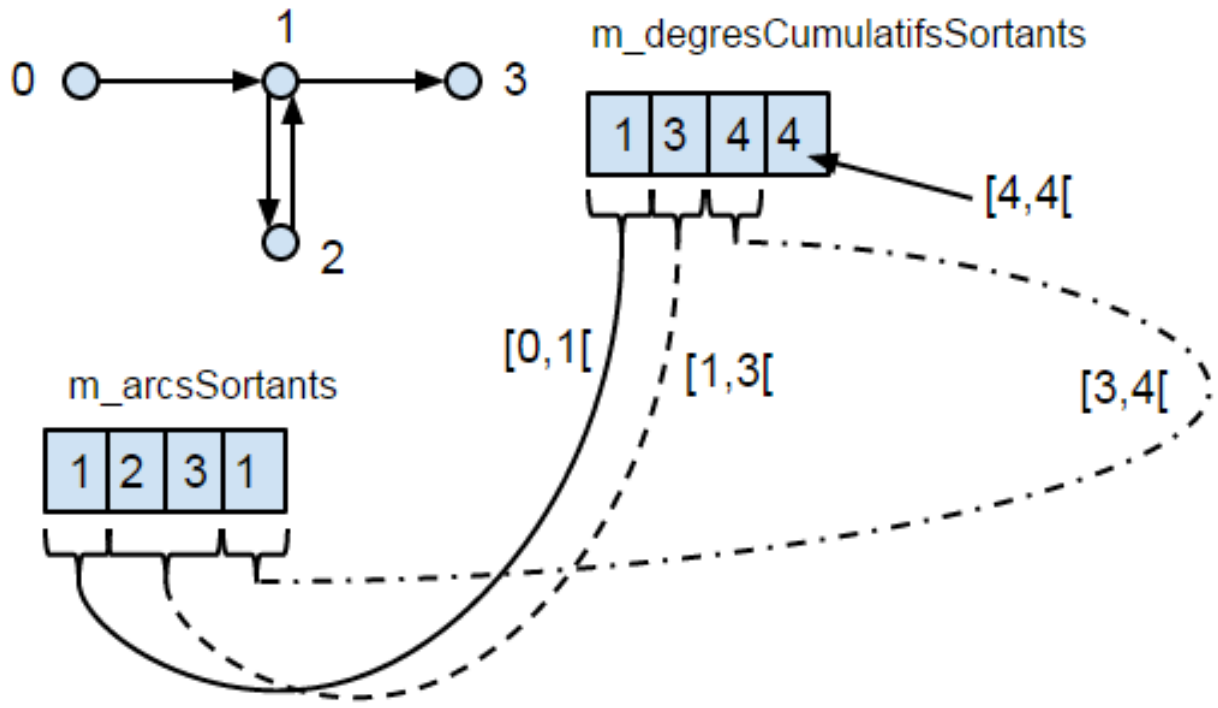


FIGURE 7 – Liaison entre nos deux tableaux

L'une des difficultés pour tester notre prototype est bien entendu la taille des graphes.

Pour une question de mémoire vive ainsi que de temps d'exécution, nous n'avons pas testé notre prototype sur les graphes pour lesquels notre code sera dédiés. Malgré cela, le fait de travailler sur des graphes beaucoup plus abordables ne nous handicape en aucun sens. Au contraire, cela nous permet un debuggage plus rapide. Nous pouvons bien entendu tester nos temps d'exécution sur de petits graphes, et ainsi voir si certains algorithmes sont effectivement plus rapides que d'autres.

À partir de ce point, nous avons codé notre premier algorithme à proprement parler, le BFS, ou algorithme de parcours en largeur. Cet algorithme est très important car il sera utilisé dans l'algorithme de Louvain, qui est l'un des algorithmes les plus importants du projet. L'algorithme du BFS permet de parcourir le graphe en temps O (n+m), et donc d'avoir un parcours de graphe efficace.

Comme dit précédemment, nous avons testé notre code sur de petits graphes ; voici un exemple d'exécution dans la sortie console ci-dessous.


FIGURE 8 – Sortie console

# 7 Planning et affectation des taches

Pour mener à bien ce travail, nous prenons contact avec Nicolas Dugué une fois par semaine, pour faire le point sur nos avancées et définir ou redéfinir des objectifs. Nous utilisons un dépôt svn pour gérer les fichiers, et l'outil Trello pour nous répartir les tâches.

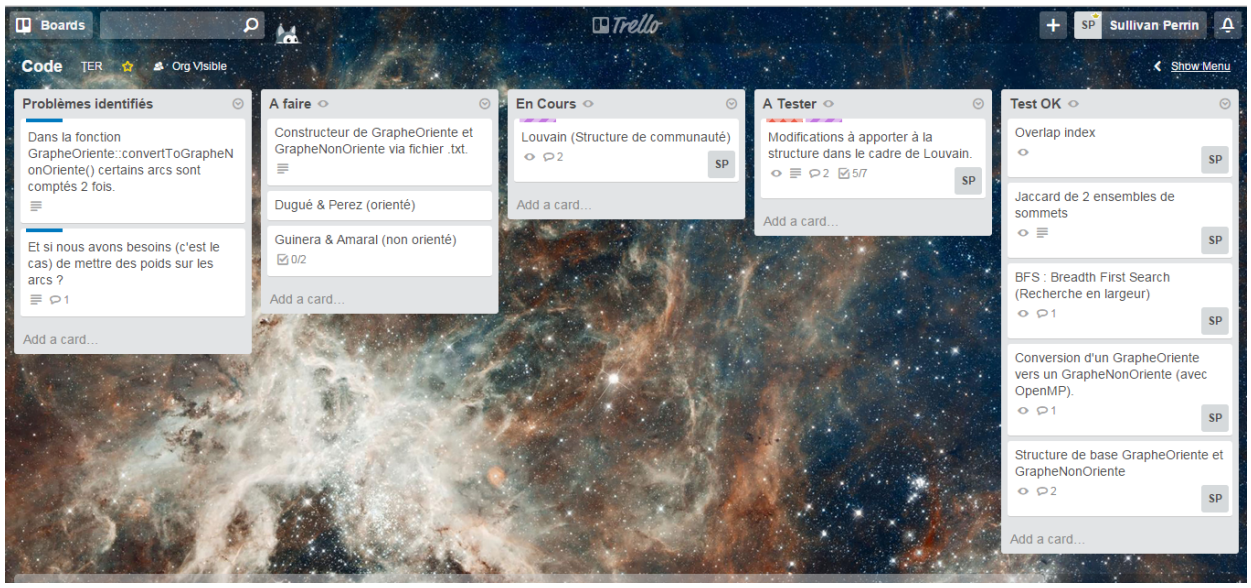


FIGURE 9 – Notre page d'organisation sur Trello

Nous avons cinq algorithmes principaux à implémenter, mais les cinq n'ont pas la même complexité ; nous essayons donc de chacun travailler sur l'un des algorithmes, tout en faisant régulièrement le point pour pouvoir venir en aide à l'un ou l'autre si une difficulté est rencontrée. Si cette difficulté nous dépasse, nous n'hésiterons pas à faire appel à Nicolas Dugué et/ou à Anthony Perez.

Notre objectif est d'avancer progressivement, de construire notre librairie brique par brique, dans l'optique de la mener le plus loin possible, le mieux possible.

# 8 Bibliographie

## Références


[1] Renaud Lambiotte et Etienne Lefebvre Vincent D. Blondel, Jean-Loup Guillaume. Fast unfolding of communities in large networks. 2008. http://arxiv.org/pdf/0803.0476.pdf.

[2] Ivan Keller et Emmanuel Viennet. Caractérisation de la structure communautaire d'un grand réseau social. http://lipn.fr/marami12/actes/keller.pdf.

[3] Wikipédia. Indice et distance de jaccard, 2015. https://fr.wikipedia.org/wiki/Indice_et_distance_de_Jaccard.

[4] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. K-core decomposition of Internet graphs : hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media*, 3 :371, April 2008.

[5] Roger Guimerà et Luis A Nunes Amaral. Cartography of complex networks : modules and universal roles. 2005. http://www.nature.com/nature/journal/v433/n7028/abs/nature03288.html.

[6] Vincent Labatut et Anthony Perez Nicolas Dugué. A community role approach to assess social capitalists visibility in the twitter network. 2014. Social Network Analysis and Mining.

[7] Nicolas Dugué et Anthony Perez. Social capitalists on twitter : detection, evolution and behavioral analysis. 2014. http://link.springer.com/article/10.1007%2Fs13278-014-0178-4.

[8] Seb13. Créer des bibliothèques avec code : :blocks, 2013. http://openclassrooms.com/courses/creer-des-bibliotheques-avec-code-blocks.

[9] Alex Blekhman. Howto : Export c++ classes from a dll, 2012. http://www.codeproject.com/Articles/28969/HowTo-Export-C-classes-from-a-DLL.

[10] Hiko-Seijuro. Exportation de classes c++ dans une bibliothèque dynamique sous linux, 2007. http://hiko-seijuro.developpez.com/articles/bibliotheque-dynamique.

[11] Vincent Reverdy. Open mp, 2011. https://software.intel.com/fr-fr/articles/debuter-avec-openmp.

[12] Wikipédia. Algorithme de parcours en largeur, 2014. http://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur.