# Projet PIM

HATHOUTE Hamza
DUONG Tom

Département Sciences du Numérique - Première année
2020-2021

# Contents



# List of Figures

# 1 Introduction

L'objectif de ce projet était de manipuler les notions que nous avions vues en cours de programmation impérative en implémentant l'algorithme PageRank de Google, de deux manières différentes. Il fallait donc commencer par décrire la structure de l'algorithme général que nous allions implémenter, et ce grâce à la méthode des raffinages. Une fois la structure établie, nous avons commencé par implémenter la méthode naive, puis l'avons testée pour être sûrs que cette dernière produisait les bons résultats. Nous avons refait de même avec la méthode creuse ensuite.
Nous présenterons un rappel du sujet, puis l'architechture des implémentations, avec les differents algorithmes et choix principaux. Nous expliquerons ensuite la structure du programme, et presenterons les durées d'éxécution selon les implantations et les fichiers tests. Enfin, nous conclurons par notre ressenti sur le projet dans sa globalité.

# 2 Rappel du sujet

Le PageRank porte le nom du cofondateur de Google, Larry Page, et est utilisé pour classer les sites Web dans les résultats de recherche de Google. Il compte le nombre et la qualité des liens vers une page qui détermine une estimation de l'importance de la page. L'hypothèse sous-jacente est que les pages importantes sont plus susceptibles de recevoir un volume plus élevé de liens provenant d'autres pages.

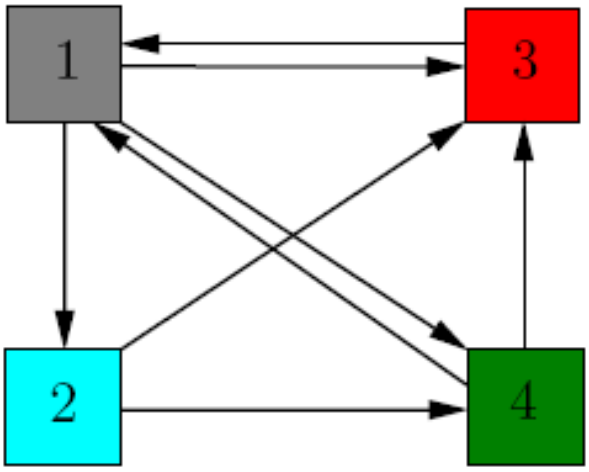


Figure 1: Graphe orienté des liens entre des pages (Source: http://pi.math.cornell.edu/)

Si on note $a_i$ les valeurs du poids sortant de chaque noeud, pour la figure 1. On aura alors:

$$a = (a_1, a_2, a_3, a_4) = (3, 2, 1, 2)$$

On peut alors construire une matrice $H = (h_{ij})_{1 \leq i,j \leq N}$ qui va nous aider a construire la matrice de Google $G$, telle que.

$$H_{ij} = \begin{cases} 1/a_i & \text{si il existe une liaison de } i \text{ vers } j \\ 0 & \text{sinon} \end{cases}$$

On definit ensuite la matrice S tels que

$$S_{ij} = \begin{cases} 1/N & \text{si } a_i = 0 \\ H_{ij} & \text{sinon} \end{cases}$$

Le calcul des poids du PageRank se fait par la matrice $G = (G_{ij})_{1 \leq i,j \leq N}$, qui est définie comme.

$$G_{ij} = \alpha.S_{ij} + \frac{1-\alpha}{N} \qquad \text{où } \alpha \in ]0,1]$$

La matrice G calculée, on peut alors calculer les poids des differents pages $\pi$. On effectue $I$ itération (avec $I$ un entier $> 0$ quelconque) tels que:

$$\pi_{k+1}^T = \pi_k^T.G \qquad \text{où } k \text{ entier} \in [0,I\text{-}1]$$

Avec comme valeur initiale

$$\pi_0 = \frac{1}{N}.e$$

où e représente un vecteur de taille N dont tous les éléments valent 1.

Il suffit alors de trier le vecteur $\pi = \pi_I$ pour avoir le PageRank.

# 3 Implémentation

## 3.1 Architecture

Nous avons choisi de creer un executable, prenant en arguments *optionnels* **la méthode de calcul** (Naive ou Creuse), **le nombre d'itération maximal**, et **la valeur de** $\alpha$. Il prend également un argument *obligatoire*, **le nom du fichier** contenant les liaisons entre les différentes pages.

Tout d'abord, il faut extraire les arguments de la ligne de commande (Nom du fichier, $\alpha$, ...). Une fois le nom du fichier lu, il faut extraire les informations qu'il contient (nombre de pages et les différents liens entre les pages). Puis, en se basant sur le choix de l'utilisateur, on calcule les poids en utilisant la méthode naïve ou creuse. On trie ensuite les poids pour obtenir les rangs et on écrit les rangs et les poids triés dans les fichiers .ord et .p respectivement. Enfin, on libère la mémoire occupée par les variables du programme.

### 3.1.1 Calcul des poids: Methode Naive

Pour cette méthode, la matrice Google a été représentée avec un tableau à deux dimensions (donc de taille fixe). Le type T_Google a été défini comme un enregistrement contenant : Un Pointeur vers la Matrice, sa taille, le coefficient $\alpha$, et le nombre d'itérations maximum.

Tout d'abord, nous avons initialisé l'objet T_Google avec sa taille, le nombre maximum d'itération et $\alpha$. Nous avons ensuite créé la matrice avec les liens lus à partir du fichier *.net*. Il reste ensuite à calculer les rangs des pages et a vider les ressources utilisées par T_Google.

### 3.1.2 Calcul des poids: Methode Creuse

Ici, la matrice Google a été représentée comme une matrice creuse, implementée comme une table de hachage des indices. Le type T_Google a ensuite été défini comme un enregistrement contenant : La matrice creuse, sa taille, le coefficient $\alpha$, le nombre d'itérations maximum et un terme *Addition* précalculé. L'algorithme de la matrice creuse suit ensuite la même démarche que celle de la méthode naive.

## 3.2 Choix réalisés

Dans les deux versions de l'implémentation, nous avons choisi de:

- Mettre tous les types dont nous aurons besoin dans un seul fichier *types.ads*, et cela pour manipuler ces types dans les differents modules.
- Ne pas créer de module Matrice contentant les différentes opérations sur les matrices et vecteurs, mais faire les calculs *à la main* pour optimiser le calcul.
- Definir deux types de listes, l'une est un vecteur (de taille fixe, utilisé pour stocker les rangs calculés), l'autre est une liste chainée (utilisée pour stocker les liens, lus depuis le fichier *.net*.
- Organiser le projet de sorte que le fichier principal soit facilement compréhensible, et que les opérations plus complexes soient dans un module à part. Par exemple, la lecture et l'écriture des fichiers sont realisées dans un module *module_io*.

## 3.3 Algorithmes

Nous avons utilisés un algorithme pour ce projet:

- L'algorithme de tri: Quicksort ou Tri Rapide, pour sa complexité plus avantageuse surtout pour des grands fichiers de liens.

## 3.4 Mise au point du programme et des modules

### 3.4.1 Modules

Les modules utilisés sont :

- **types**: Definit les types utilisés dans les autres modules.

- **module_io**: Il gère la lecture et l'écriture des fichiers.
- **google_naive**: Il s'agit de l'algorithme de calcul par la méthode naive.
- **google_creuse**: Il s'agit de l'algorithme de calcul par la méthode creuse.
- **th**: Structure de données associative sous forme de table de hachage.
- **lca**: Structure de données associative sous forme d'une liste chaînée.
- **lc**: Structure de données sous forme d'une liste chaînée.
- **vecteur**: Liste de taille fixe choisie par l'utilisateur.

#### 3.4.2 Programme

Le programme principal est le *pagerank.adb*. Tout d'abord, il lit les arguments de la ligne de commande. Il lit ensuite les liens du fichier *.net* à l'aide du module **module_io**. En fonction du choix de l'utilisateur donné dans les arguments de la ligne de commande, le programme calcule les poids avec la méthode creuse ou naive (par défaut, la méthode naive sera utilisée). On trie alors les valeurs de poids pour obtenir les rangs, et ce à l'aide du sous programme de tri, contenu dans le module **vecteur**. On écrit ces rangs et poids dans les fichiers *.ord* et *.p* respectifs, toujours à l'aide du **module_io**. Enfin, on vide les ressources utilisées par le programme principal (liens et rangs).

# 4 Analyse d'algorithme

## 4.1 Complexité temporelle

Le premier algorithme, **Google Naive**, a une complexité temporelle moins élevée que la methode Creuse. Ceci vient du fait que l'accès à une valeur dans un tableau à deux dimensions est $\mathcal{O}(1)$, ce qui n'est pas le cas pour une table de hachage qui doit parcourir tout les éléments qui ont la même valeur de hachage.

## 4.2 Complexité spatiale

L'algorithme **Google Creuse** en revanche possède une complexité spatiale bien inférieure à celle de l'algorithme précédent, pour un nombre de pages élevé. Par exemple, si $L$ est le nombre de liens, et $P$ le nombre de pages, alors la matrice creuse a comme compléxité $\mathcal{O}(L)$, alors que celle Naive c'est $\mathcal{O}(P^2)$. En général on a $L << P^2$ pour $P$ assez grand. Par exemple, pour le fichier test *brainlinks.net*: $P = 10^4$ et $L \approx 10^6$, donc $P^2 << L$

## 4.3 Temps d'exécutions

### 4.3.1 Test Exemple_Sujet

| Precision | 3 | 6 | 10 |
|---|---|---|---|
| Naive | 4ms | 4ms | 4ms |
| Creuse | 4ms | 5ms | 5ms |

### 4.3.2 Test Worm

| Precision | 3 | 6 | 10 |
|---|---|---|---|
| Naive | 230ms | 240ms | 240ms |
| Creuse | 1.3s | 1.35s | 1.4s |

### 4.3.3 Test Brainlinks

| Precision | 3 | 6 | 10 |
|---|---|---|---|
| Naive | 3min | - | - |
| Creuse | $\approx 4h$ | - | - |

## 4.4 Conclusion

Pour un nombre de pages petit ($\leq 10^3$), le tableau à deux dimensions ne prend que quelques kB/MB et donc il est préférable d'utiliser la méthode Naive, alors que lorsque nous avons besoin de traiter des plusieurs dizaines de milliers de pages, seule la méthode creuse fonctionne.

# 5 Difficultés rencontrées et solutions adoptées

Pour la manipulation d'une variable de type défini par nous dans plusieurs modules, nous avons essayé d'initialiser le module generique dans tout les modules où nous utilisons ce type, ce qui donne un erreur de compilation.

La seule solution que nous avons trouvé était de definir tout ces types dans un seul module que nous avons nommé **types**, et d'importer ce module dans les differents modules où nous allons utiliser ces type.

# 6 Conclusion

Lors de la réalisation de ce projet, nous avons du faire preuve de régularité dans notre travail, surtout lorsqu'il s'agissait de travailler sur nos heures de temps libre. De plus, avec les conditions particulières de cette année, le travail a exclusivement été réalisé à distance, ainsi, la communication et la coordination étaient primordiales. Malgré les contraintes, nous avons su progresser à deux en utilisant les différents outils (svn, partage d'écran...).