

Dans ce rapport de TP, nous vous présentons comment nous avons procédé pour programmer notre algorithme de PageRank. Nous présentons aussi quelques résultats et analyse de l'application de notre algorithme sur des graphes. Nous avons utilisé le langage JAVA pour mettre en place ce programme.

1. Un format simple pour les graphes et les matrices creuses:

On définit le type **matrice** dans la classe *SparseMatrix*. Il s'agit d'une matrice creuse décrite de la façon suivante:

- Un entier **m** qui correspond à au nombre de valeurs non nulles dans la matrice, ou le nombre de nœuds dans le graphe.
- Un entier **n** qui représente la taille de la matrice d'adjacence du graphe.
- Un tableau de réels **C** de taille **m** et qui contient toutes les cases non nulles de la matrice.
- Un tableau d'entiers **L** de taille **n+1**, **L[i]** est l'indice du début de la i-ème ligne de la matrice de **C**.
- Un tableau d'entier **I** de taille **m** contenant les indices des colonnes associées aux éléments non nul de la ligne i (pour **I[i]**).

La matrice creuse nous permettra d'économiser de la mémoire car on ne stocke ici que les valeurs positives de la matrice. De manière générale, la matrice d'adjacence d'un graphe représentant un réseau d'interaction contient un grand nombre d'éléments nuls.

Si on représentait la matrice d'un graphe de 1000 noeuds, la taille de la matrice sera de 1 000 x 1 000, par exemple, et 2000 arêtes, avec un tableau de deux dimensions, on aurait gaspillé 1 Million (1000000) de cases. Alors que si on représente notre matrice avec **LCI**, on aurait juste utilisé 5001 cases (1001 pour **L**, 2000 pour **C** et 2000 pour **I**).

$$100 - (5001 * 100 / 10000000) \approx 99,5$$

Donc sur cette exemple on a économisé $\approx 99,5\%$ avec LCI par rapport à la méthode du tableau à deux dimensions.

Pour la suite, nous considérons des graphes orienté fortement connexe et apériodique.

La multiplication de la matrice par un vecteur est calculé par la fonction *MultiplyWithVector(float[] v)* :

```
for (int i = 0; i < n; i++) {
    for (int j = L[i]; j <= L[i + 1] - 1; j++) {
        product[i] += C[j] * v[I[j]];
    }
}
```

Pour chaque itération i, nous calculons le produit en (L[i+1] – L[i]) opérations.

Alors, nous avons un total de L[n] – L[0] = L[n] = m opérations.

Donc ce calcul se fait en temps linéaire. Sa complexité est de **O(m)**.

Nous observons expérimentalement, c'est aussi en **O(m)**.

La fonction *MultiplyTransposeWithVector*(*float*[] *v*) calcule le produit la transposée de la matrice par un vecteur *v* en $O(m)$, sans calculer la explicitement la transposé de la matrice :

```
for (int i = 0; i < n; i++) {
    for (int j = L[i]; j < L[i + 1]; j++) {
        product[I[j]] += C[j] * v[i];
    }
}
```

Pour la complexité, il s'agit de la même explication que pour la fonction *MultiplyWithVector*(*v*).

Nous avons tester nos fonctions à l'aide de testes unitaires dans la classe *GraphTest* sur un graphe dont nous avons préalablement calculer *L*, *C* et *I* mais aussi la multiplication par un vecteur que nous avons choisi et la multiplication de la transposé par ce même vecteur. Nous avons comparer les résultats obtenus avec les résultats attendus.

On définit le type **graphe** dans la classe **Graph** de la même façon que pour le type matrice car *Graph* hérite de *SparseMatrix*. Il représentera plus une interface vers la matrice creuse définit plus haut.

D'autre part, nous avons définit une classe **GraphReader** qui nous permettra de construire notre graphe à partir d'un fichier texte donnée dans un format spécifique afin que nous puissions simplifier le travail du parser.

Format de fichier

Les graphes sont décrits par des fichiers placés dans le répertoire data/ tels que :

- les lignes vides ou commençant par le caractère # sont ignorées ;
- les autres lignes doivent être de la forme « i j », symbolisant un arc du sommet i vers le sommet j.

En supposant que nous connaissons déjà *n* et *m* (resp. nombre de nœuds et nombres de liens), la construction du graphe se fait en temps linéaire puisqu'elle calcule les tableaux *I* et *L* à la volet et ensuite parcourt *L* pour remplir le tableaux *C* avec la valeur de $1/d+(p)$, *d*+ étant le degré sortant d'un sommet *p*. La complexité de la construction du graphe est donc en $O(n + m)$.

La méthode *getGraph()* de la classe *GraphReader* nous permettra donc de récupérer le graphe construit afin de la manipuler pour le calcul du PageRank.

2. Pagerank:

Nous avons définit une classe Page Rank pour faire le calcul. Nous avons définit d'une part quelques variables qui nous permettrons d'effectuer nos calculs :

```
private boolean verbose = false; // permet d'effectuer le PageRank en mode verbeux
private float[] pageRankVector; // vecteur PageRank
private float[] stochVector; // vecteur stochastique
private SparseMatrix matrix; // matrice creuse associée au graphe
private float epsilon = 0.001F; // epsilon par défaut
```

D'autre part, nous définissons les fonctions qui nous permettront de calculer les différents PageRank comme décrit dans le sujet :

(Exercice 4)

1 – *computePageRankStd* qui calcul le PageRank en considérant un vecteur stochastique équipropable et s'arrête quand la différence de la norme de $P(n)$ et $P(n-1)$ est assez petite \sim epsilon qui par défaut est de 0,01.

```
public void computePageRankStd()
```

2 – *computePageRankZero* qui prend en argument un numéro de nœud de départ et un nombre de pas. Elle initialise le vecteur stochastique V à 0 et $V[start]$ à 1.

```
public void computePageRankZero(int start, int nbPas)
```

3 – *computePageRank* qui prend en argument le nombre de pas qu'on souhaite effectué pour le calcul du PageRank :

```
public void computePageRank(int nbPas)
```

(Exercice 5)

4 – *computePageRankWithZap* qui prend en argument le facteur Zap compris entre 0.1 et 0.2 et le nombre de pas qu'on souhaite effectué qui permet de s'assurer que l'algorithme se termine bien.

```
public void computePageRankWithZap(float zap, int nbPas)
```

3. Résultats et interprétation :

Pour cette partie, nous avons tester notre algorithme Page Rank sur deux types de données :

- Des graphes joués que nous avons fabriqué en cours et dont nous connaissons les propriétés.
- Un corpus de graphe orienté que nous avons récupéré à partir de SNAP¹ et plus précisément sur les graphes du protocole pair-à-pair Gnutella.

3.1 Exemples :

Nom	Type	Noeuds	Liens	Description
exemple1	Directed	5	8	Graphe quelconque vu en cours
exemple1bis	Directed	3	3	Graphe cyclique
exemple2	Directed	3	3	Graphe pas fortement connexe
exemple3	Directed	3	4	Graphe à puits avec cycle
exemple4	Directed	4	5	Graphe avec un circuit égal à 2
exemple5	Directed	5	8	Graphe avec 3 cycles
exemple6	Directed	3	6	Graphe symétrique
exemple7	Directed	5	8	Graphe étoile

Pour ces graphes là, nous avons effectué les calculs du PageRank en utilisant les 4 méthodes décrits dans le paragraphe 2 afin de vérifier certaines propriétés sur la convergence et la vitesse dans laquelle le calcul du PageRank converge.

Pour le fichier *exemple1*, nous observons que avec la fonction du *PageRankStd*, le PageRank est plus important sur le sommet numéro 2 (~ 0.66) étant donné qu'il y a deux autres sommets qui pointent sur lui. Donc le PageRank déverse dans le nœud 2, même s'il ne s'annule pas sur les autres sommets. Après 100 itérations, nous obtenons le suivant :

```
P(100) = ( 0.3333333 2.4520837E-8 0.6666666 2.8801182E-8 1.1276233E-8 )
```

¹ Stanford Large Network Dataset Collection – <http://snap.stanford.edu/data/index.html>

Avec le PageRankZero, nous calculons le PageRank à partir de tous les nœuds du graphe. Dans ce cas, le paramètre Z est initialisé de sorte à ce que la probabilité d'être sur le sommet de départ soit 1 et le reste initialisé à 0.

Remarque : Les graphes que nous utilisons ici sont relativement petits par rapport aux graphes que nous utilisons par la deuxième partie (cf. 3.2), mais les résultats nous permettent quand même de faire quelques observations.

- > Test PageRankZero on data/exemple1.data a partir de 0
=> P converge apres 28 pas.
- > Test PageRankZero on data/exemple1.data a partir de 1
=> P converge apres 308 pas.
- > Test PageRankZero on data/exemple1.data a partir de 2
=> P converge apres 27 pas.
- > Test PageRankZero on data/exemple1.data a partir de 3
=> P converge apres 307 pas.
- > Test PageRankZero on data/exemple1.data a partir de 4
=> P converge apres 309 pas.

Nous observons dans ce cas que la convergence est beaucoup plus rapide pour le PageRankZero en partant de 0 et 2. Si nous revenons sur le résultat obtenu par le PageRankStd, nous remarquons que les mêmes sommets qui accumulaient le PageRank présentent aussi une certaine vitesse vis-à-vis le calcul du PageRank-zero par rapport aux autres sommets du graphe.

Par contre, avec la fonction PageRank, le calcul du PageRank nous fait remarquer que même si nous choisissons un nombre de pas élevé (nous avons choisi 1000 pas pour ce test), le PageRank fini quand même par converger au bout de 305 pas.

- > Test PageRank on data/exemple1.data
=> P converge apres 305 pas.

Enfin, nous avons effectué le calcul du PageRank en considérant un facteur Zap

- > Test PageRank with Zap Factor on data/exemple1.data
=> P converge apres 54 pas.

Le PageRank converge beaucoup plus vite avec le facteur Zap !

Nous avons effectué le calcul du PageRank sur tout les autres exemples.

Le graphe *exemple1bis* est un graphe cyclique avec 3 sommets. Nous remarquons qu'il converge dès le premier pas. Par contre avec la fonction PageRankZero, le PageRank diverge. Le graphe *exemple5* permet de vérifier le même comportement. Par contre pour *exemple4*, c'est le cycle qui va attirer tout le PageRank.

Résultat du PageRank avec 1000 pas sur l'*exemple1bis*

```
P(966) = ( 0.0 0.0 1.0 )
P(967) = ( 1.0 0.0 0.0 )
P(968) = ( 0.0 1.0 0.0 )
P(969) = ( 0.0 0.0 1.0 )
```

Le graphe *exemple2* est un graphe qui n'est pas fortement connexe, nous observons qu'il converge après 273 pas avec la méthode PageRankStd, mais après 2 pas en partant de 1 avec PageRankZero.

Le graphe *exemple3* contient un puits vers le sommets 2, le PageRank va se perdre dans le puits. Nous remarquons ce comportement avec toutes les méthodes que nous avons utilisé. Concernant le graphe symétrique (*exemple6*), nous obtenons le suivant :

```
> Test PageRankStd on data/exemple6.data
P(0) = ( 0.33333334 0.33333334 0.33333334 )
P(1) = ( 0.33333334 0.33333334 0.33333334 )
=> P converge apres 1 pas.
P(1) = ( 0.33333334 0.33333334 0.33333334 )
> Test PageRankZero on data/exemple6.dataa partir de 0
=> P converge apres 28 pas.
> Test PageRankZero on data/exemple6.dataa partir de 1
=> P converge apres 28 pas.
> Test PageRankZero on data/exemple6.dataa partir de 2
=> P converge apres 28 pas.
> Test PageRank with Zap Factor on data/exemple6.data
=> P converge apres 3 pas.
> Test PageRank on data/exemple6.data
=> P converge apres 1 pas.
```

Il semble que le PageRank fini toujours par converger à la même vitesse pour le PageRank-zero mais il est beaucoup plus rapide avec les autres méthodes utilisées.

Enfin pour le graphe étoile (*star.data*),

```
> Test PageRankStd on data/star.data
P(0) = ( 0.2 0.2 0.2 0.2 0.2 )
P(1) = ( 0.8 0.05 0.05 0.05 0.05 )
```

Avec le PageRank standard, nous observons que le PageRank diverge.

```
> Test PageRankZero on data/star.dataa partir de 3
P(0) = ( 0.0 0.0 0.0 1.0 0.0 )
P(1) = ( 1.0 0.0 0.0 0.0 0.0 )
P(2) = ( 0.0 0.25 0.25 0.25 0.25 )
P(3) = ( 1.0 0.0 0.0 0.0 0.0 )
```

Idem que pour le PageRank standard. Par contre avec le facteur Zap, il converge avec une meilleur distribution de probabilité.

```
P(50) = ( 0.48170674 0.12957329 0.12957329 0.12957329 0.12957329 )
P(51) = ( 0.4832809 0.12917975 0.12917975 0.12917975 0.12917975 )
P(52) = ( 0.4818799 0.12953 0.12953 0.12953 0.12953 )
```

3.2 Informations sur l'ensemble de données étudier:

Il s'agit d'une séquence de snapshots du réseau de partage de fichiers pair-à-pair Gnutella à partir de Août 2002. Les noeuds représentent les hôtes dans la topologie du réseau Gnutella et les liens représentent les connexions entre les hôtes Gnutella.

Gnutella² est un protocole informatique décentralisé de recherche et de transfert de fichiers pair-à-pair (aussi appelés P2P). Il a été imaginé en 2000 par Tom Pepper et Justin Frankel alors programmeurs pour la société Nullsoft, qui a également édité WinAmp.

2 Selon Wikipedia

Pour imaginer comment Gnutella a initialement travaillé, imaginez un graphe d'utilisateurs (appelés nœuds), chacun d'entre eux possède le logiciel client Gnutella. Au démarrage initial, le logiciel client doit amorcer et trouver au moins un autre nœud. Diverses méthodes ont été utilisées pour cela, y compris une liste d'adresses pré-existant de nœuds éventuellement de travail fournis avec le logiciel, à l'aide de caches Web mises à jour de nœuds connus (appelés Gnutella Web caches), UDP Host Caches et même IRC. Une fois connecté, le client demande une liste d'adresses de travail. Le client tente de se connecter aux nœuds trouvés, ainsi que les nœuds qu'il reçoit d'autres clients, jusqu'à ce qu'il atteigne un certain quota. Il se connecte à seulement à ces nœuds, et met en cache localement des adresses qu'il n'a pas encore essayé, et ignore les adresses qu'il a essayé qui étaient invalides.

Nom	Type	Noeuds	Liens	Description
p2p-Gnutella04	Directed	10,877	39,994	Gnutella peer to peer network from August 4 2002
p2p-Gnutella05	Directed	8,846	31,839	Gnutella peer to peer network from August 5 2002
p2p-Gnutella06	Directed	8,717	31,525	Gnutella peer to peer network from August 6 2002
p2p-Gnutella08	Directed	6,301	20,777	Gnutella peer to peer network from August 8 2002
p2p-Gnutella09	Directed	8,114	26,013	Gnutella peer to peer network from August 9 2002
p2p-Gnutella24	Directed	26,518	65,369	Gnutella peer to peer network from August 24 2002
p2p-Gnutella25	Directed	22,687	54,705	Gnutella peer to peer network from August 25 2002
p2p-Gnutella30	Directed	36,682	88,328	Gnutella peer to peer network from August 30 2002
p2p-Gnutella31	Directed	62,586	147,892	Gnutella peer to peer network from August 31 2002

Après une bref présentation de Gnutella, nous présentons les testes de PageRank sur ces réseaux.

Sur p2p-Gnutella04, on observe que même après 1000000, le Page Rank converge très lentement, donc notre programme n'est pas conclusif sur la convergence en utilisant le Page Rank Standard.

```
> Testing PageRank with pace on p2p-Gnutella04.data :
P(90) = ( 2.78177751E10 1.20288133E10 2.56148603E10 2.69004513E10 1.20677081E11
2.57388667E10 4.3914519E10...

P(100) = ( 2.78177751E10 1.20288133E10 2.56148603E10 2.69004513E10 1.20677081E11
2.57388667E10 4.3914519E10...
```

Les informations associées au graphe ci-dessous était incorrect. Nous avons du le modifier légèrement afin de pouvoir faire nos calculs. Par contre, avec PageRankZero, nous observons bien que le PageRank converge vers 130 pas en moyenne, mais parfois il converge après 3 ou 4 pas.

Avec le facteur Zap, la convergence se fait après 53 pas. Ce qui confirme bien que le facteur Zap permet une convergence plus rapidement que dans le cas standard.

En annexe 1 vous trouverez les autres résultats de testes que nous avons effectué et qui reflète le même comportement.

L'annexe 2 représente les résultats d'une série de tests effectués sur le graphe des discussions Wikipedia à l'aide de PageRank avec le facteur Zap. Le graphe contient 2394385 nœuds et 5021410 liens. Le graphe est énorme par rapport au corpus que nous avons analysé auparavant mais le calcul reste quand même assez rapide. La construction de la matrice creuse associée au graphe ce fait en 4,057 sec. Le calcul du PageRank s'effectue en moyenne ~ 12 sec pour ce graphe quelque soit le facteur Zap à l'exception du dernier teste avec d = 0.19 qui s'effectue en 6,993 sec. (Cf. Annexe 3)

Annexe 1: Quelques résultats sur Gnutella.

- > Testing PageRank with pace on p2p-Gnutella05.data :
=> P converge apres 157 pas.
- > Test PageRank with Zap Factor on data/p2p-Gnutella05.data
=> P converge apres 69 pas.

- > Testing PageRank with pace on p2p-Gnutella09.data :
=> P converge apres 99 pas.
- > Test PageRank with Zap Factor on data/p2p-Gnutella09.data
=> P converge apres 42 pas.

- > Testing PageRank with pace on p2p-Gnutella24.data :
=> P converge apres 56 pas.
- > Test PageRank with Zap Factor on data/p2p-Gnutella24.data
=> P converge apres 30 pas.

- > Testing PageRank with pace on p2p-Gnutella25.data :
=> P converge apres 70 pas.
- > Test PageRank with Zap Factor on data/p2p-Gnutella25.data
=> P converge apres 33 pas.

- > Testing PageRank with pace on p2p-Gnutella30.data :
=> P converge apres 64 pas.
- > Test PageRank with Zap Factor on data/p2p-Gnutella30.data
=> P converge apres 32 pas.

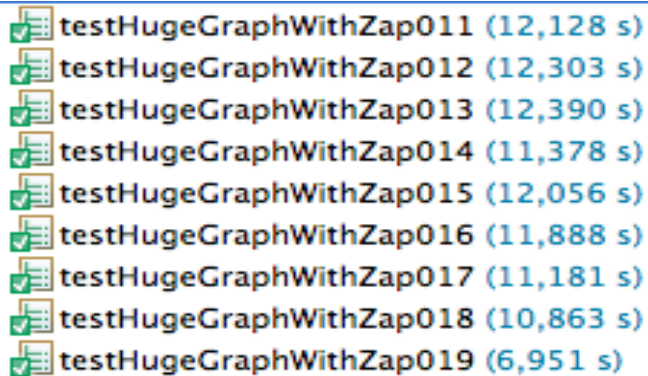
- > Testing PageRank with pace on p2p-Gnutella31.data :
=> P converge apres 51 pas.
- > Test PageRank with Zap Factor on data/p2p-Gnutella31.data
=> P converge apres 30 pas.










Avec le facteur Zap, le calcul est deux fois plus rapide !

Annexe 2 : Quelques résultats avec PageRank avec le facteur Zap d sur wiki-Talk

- > Test on data/wiki-Talk.data with Zap factor $d = 0.11$
=> P converge apres 65 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.12$
=> P converge apres 67 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.13$
=> P converge apres 67 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.14$
=> P converge apres 60 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.15$
=> P converge apres 64 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.16$
=> P converge apres 62 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.17$
=> P converge apres 58 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.18$
=> P converge apres 56 pas.
- > Test on data/wiki-Talk.data with Zap factor $d = 0.19$
=> P converge apres 54 pas.

Annexe 3 : Capture d'écran sur le benchmark du PageRank avec Zap d sur wiki-Talk



	testHugeGraphWithZap011 (12,128 s)
	testHugeGraphWithZap012 (12,303 s)
	testHugeGraphWithZap013 (12,390 s)
	testHugeGraphWithZap014 (11,378 s)
	testHugeGraphWithZap015 (12,056 s)
	testHugeGraphWithZap016 (11,888 s)
	testHugeGraphWithZap017 (11,181 s)
	testHugeGraphWithZap018 (10,863 s)
	testHugeGraphWithZap019 (6,951 s)