LSINF1250 – Mathématiques pour l'informatique
Projet PageRank

Céline Deknop
4106-1400

Sébastien Strebelle
8143-1300

12 mai 2016

Table des matières

---

1 Introduction

Dans le cadre du cours de mathématiques pour l'informatique, LSINF1250, nous avons étudié les graphes et leurs utilisations dans l'informatique. Notre projet était de créer une implémentation de l'algorithme *PageRank*. Celui-ci permet de classer des pages internet selon leur importance. Elles sont représentées par un graphe dirigé dont les nœuds sont les différentes pages et les arcs sont les liens entre elles. Ce rapport explique nos choix d'implémentations, l'algorithme utilisé ainsi que la méthode mathématique sous-jacente.

2 Explication de la procédure en Java

La procédure Java applique presque à la lettre l'algorithme vu au cours. Tout d'abord, on lit le fichier en entrée et on rempli un tableau de *double* avec la matrice d'adjacence qu'il représente. Cette matrice, ainsi que le vecteur de personalisation et le paramètre de téléportation α sont passés à la méthode *pageRank*. Celle-ci commence par normaliser la matrice d'adjacence qu'elle a reçue, puis elle crée les différentes matrices nécéssaires aux itérations (le vecteur de personnalisation transposé et le vecteur $\mathbf{x}$ en $t = 0$).

Ensuite, la fonction *rec* calcule les valeurs de x via la power method (voir ci-dessous pour l'algorithme), et à chaque itération, elle vérifie si les valeurs convergent via la fonction *converge*. De plus, si le nombre d'itération atteint 5000 sans converger, on considère que les valeurs ocillent et on arrête l'itération pour ne pas surcharger la machine. Néanmoins, durant les premières récursions, les résultats vont peu changer. Nous n'arrêtons alors la récursion que si le nombre d'itération est au moins de 10.

Quand la recursion est terminée, on récupère le vecteur de sortie. Ce vecteur contient les scores *PageRank* des nœuds. On affiche ensuite à l'écran le résultat.

3 Algorithme utilisé pour les scores *PageRank*

L'algorithme que nous avons utilisé est celui de la *power method.* Nous utilisons la formule (1) :

$$\mathbf{x}_{t+1}^{\intercal} = \alpha \cdot \mathbf{x}_t^{\intercal} \cdot \mathbf{P} + (1 - \alpha) \cdot \mathbf{v}^{\intercal} \tag{1}$$

Dans la formule (1), la matrice $\mathbf{P}$ est la matrice d'adjacence normalisée, le facteur α est le paramètre de téléportation et le vecteur $\mathbf{v}$ est le vecteur de personnalisation. Le vecteur $\mathbf{x}_t$ représente quand à lui les scores *PageRank* au temps t (après t itérations de la formule).

À l'initialisation de l'algorithme, on choisit arbitrairement une valeur pour le vecteur $\mathbf{x}$. Après t itérations, ce vecteur va converger vers la valeur attendue, le score *PageRank.* L'avantage de cette formule utilisée est qu'elle n'utilise que des multiplications sur des matrices *sparses*, c'est à dire dont une majorité d'élément sont probablement nuls. De cette façon, les multiplications peuvent se faire de manières plus efficaces et l'algorithme est plus rapide.

Si l'on veut utiliser l'algorithme *PageRank* sans téléportation, il suffit de donner la valeur 1 à α. Si l'on veut l'utiliser sans personnalisation, il suffit de donner aux éléments du vecteur $\mathbf{v}$ la valeur 1.

4 Librairie JAMA

La librairie JAMA nous a semblé être le meilleur choix pour manipuler les matrices, malgré le fait qu'elle nécéssite un téléchargement en plus de la commande *import.* En effet, elle permet de passer d'un tableau de *double* vers une *Matrix* (type généré par cette librairie) et elle comprend toutes les méthode dont nous avions besoin (transposée d'une matrice, somme de deux matrices, multiplications de matrices et multiplication d'une matrice par un entier). Elle contient également une méthode pour résoudre un système d'équation linéaire, que nous n'avons bien sûr pas utilisée.

5 Références

Tout d'abord, afin de comprendre l'algorithme PageRank en lui-même, ainsi que d'en apprendre un peu plus sur celui-ci, nous avons utilisé le site web webmaster-hub.com : l'algorithme du pagerank expliqué.

La formule que nous avons appliquée est, elle, tirée directement des slides du cours : Chapitre 10, slide 135.

Enfin, afin de comprendre la librairie JAMA dont nous vous parlons ci-dessus, nous nous sommes renseignés sur sa documentation, à l'adresse math.nist.gov : Jama doc.

6 Conclusion

En conclusion, nous pensons que nous proposons une implémentation correcte et efficace. En effet, elle converge en un nombre raisonnable d'itération et le temps d'exécution global est assez rapide.

Grâce à ce projet, nous avons à présent une bien meilleure compréhension de *PageRank*, ainsi qu'une meilleure idée des applications pratiques que peut avoir la théorie des graphe au sein de l'informatique.

A Scores *PageRank* des graphes exemples

L'exemple *Bott* converge en 11 itérations et donne les résultats suivants :

1. $0,04831$
2. $0,08704$
3. $0,10471$
4. $0,12946$
5. $0,07280$
6. $0,03583$
7. $0,03204$
8. $0,10542$
9. $0,17210$
10. $0,07216$
11. $0,13921$

Ce qui donne le classement suivant : 9 - 11 - 4 - 8 - 3 - 2 - 5 - 10 - 1 - 6 - 7. La figure 1 donne la répartition des scores *PageRank*.

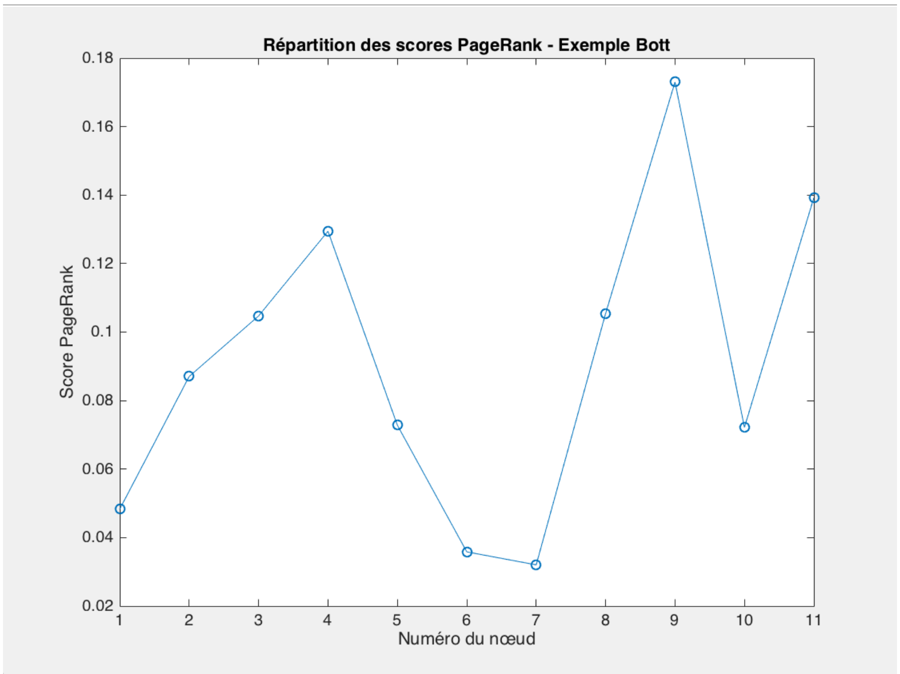

FIGURE 1 – Répartition des scores *PageRank* de l'exemple Bott

Nous allons à présent vous prouver l'efficacité de notre vecteur de personnalisation, sur le même graphe que le précédent exemple.

Avec un vecteur de personnalisation "neutre" : 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 et alpha valant 0.85, nous obtenons :

1. 0.07839
2. 0.08928
3. 0.10552
4. 0.10145
5. 0.08566
6. 0.07301
7. 0.07074
8. 0.09784
9. 0.11457
10. 0.08466
11. 0.09888

Cela nous donne le classement suivant : 9 - 3 - 4 - 11 - 8 - 2 - 5 - 10 - 1 - 6 - 7

Avec ce vecteur de personnalisation : 0.05, 0.05, 0.05, 0.05, 0.5, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05 (toutes les valeurs valent 0.5 sauf la cinquième) et aplha valant 0.85, nous obtenons :

1. 0.04691
2. 0.08264
3. 0.09685
4. 0.11107
5. 0.13410
6. 0.04423
7. 0.03258
8. 0.09637
9. 0.15502
10. 0.07501
11. 0.12521

Le classement est cette fois-ci : 9 - 5 - 11 - 4 - 3 - 8 - 2 - 10 - 1 - 6 - 7

On peut voir que le noeud 5, qui était septième au classement avec le vecteur de personnalisation neutre, est à présent second, ce qui prouve bien que le poids donné au valeurs du vecteur influe sur le classement PageRank (vous pourrez voir une graphe représentant les deux répartitions en figure 2).

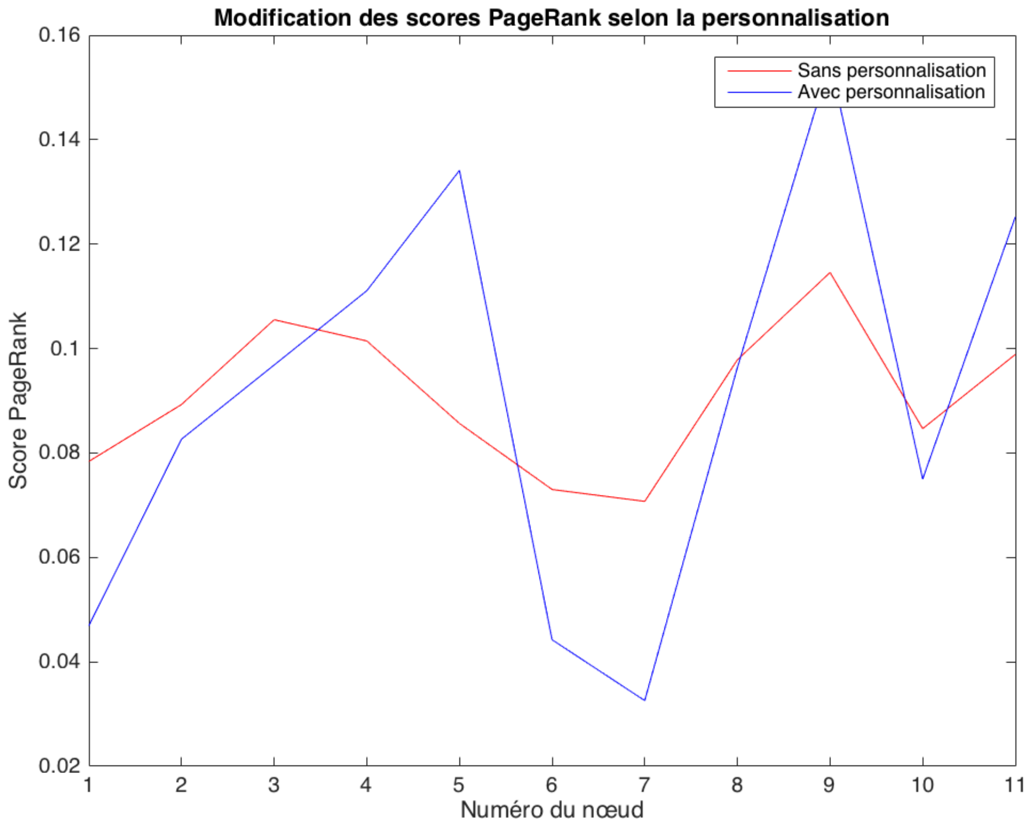

Figure 2 – Vecteur de personnalisation

Le deuxième exemple, *Coleman*, converge en 271 itérations et donne les résultats suivant :

1. $1,50673 \cdot 10^{-5}$
2. $1,50673 \cdot 10^{-5}$
3. $1,50673 \cdot 10^{-5}$
4. $3,02744 \cdot 10^{-5}$
5. $2.26805 \cdot 10^{-5}$
6. $5.38491 \cdot 10^{-5}$
7. $11.96620 \cdot 10^{-5}$
8. $11.96620 \cdot 10^{-5}$
9. $2.26454 \cdot 10^{-5}$
10. $104.82624 \cdot 10^{-5}$
11. $5.73537 \cdot 10^{-5}$
12. $5.84049 \cdot 10^{-5}$
13. $4.90877 \cdot 10^{-5}$
14. $7.82823 \cdot 10^{-5}$
15. $2.56766 \cdot 10^{-5}$
16. $9.75931 \cdot 10^{-5}$
17. $20.79630 \cdot 10^{-5}$
18. $8.07615 \cdot 10^{-5}$
19. $7.55899 \cdot 10^{-5}$
20. $19.27723 \cdot 10^{-5}$
21. $37.50654 \cdot 10^{-5}$
22. $38.70707 \cdot 10^{-5}$
23. $1.50673 \cdot 10^{-5}$
24. $1.50673 \cdot 10^{-5}$
25. $1.50673 \cdot 10^{-5}$
26. $2.40682 \cdot 10^{-5}$
27. $3.02744 \cdot 10^{-5}$
28. $3.17424 \cdot 10^{-5}$
29. $3.17424 \cdot 10^{-5}$
30. $177.71063 \cdot 10^{-5}$
31. $308.14142 \cdot 10^{-5}$
32. $3.57926 \cdot 10^{-5}$
33. $4.07585 \cdot 10^{-5}$
34. $4.54767 \cdot 10^{-5}$
35. $198.77159 \cdot 10^{-5}$
36. $6.63699 \cdot 10^{-5}$
37. $213.32340 \cdot 10^{-5}$
38. $18.31171 \cdot 10^{-5}$
39. $8.28871 \cdot 10^{-5}$
40. $213.70392 \cdot 10^{-5}$
41. $16.00141 \cdot 10^{-5}$
42. $14.72031 \cdot 10^{-5}$
43. $18.16927 \cdot 10^{-5}$
44. $48.05317 \cdot 10^{-5}$
45. $19.90590 \cdot 10^{-5}$
46. $5.46149 \cdot 10^{-5}$
47. $185.09427 \cdot 10^{-5}$
48. $303.14053 \cdot 10^{-5}$
49. $23.91056 \cdot 10^{-5}$
50. $276.42911 \cdot 10^{-5}$
51. $27.24860 \cdot 10^{-5}$
52. $326.75415 \cdot 10^{-5}$
53. $261.14691 \cdot 10^{-5}$
54. $35.65520 \cdot 10^{-5}$
55. $34.29064 \cdot 10^{-5}$
56. $1.50673 \cdot 10^{-5}$
57. $5.51139 \cdot 10^{-5}$
58. $755.84584 \cdot 10^{-5}$
59. $4.54767 \cdot 10^{-5}$

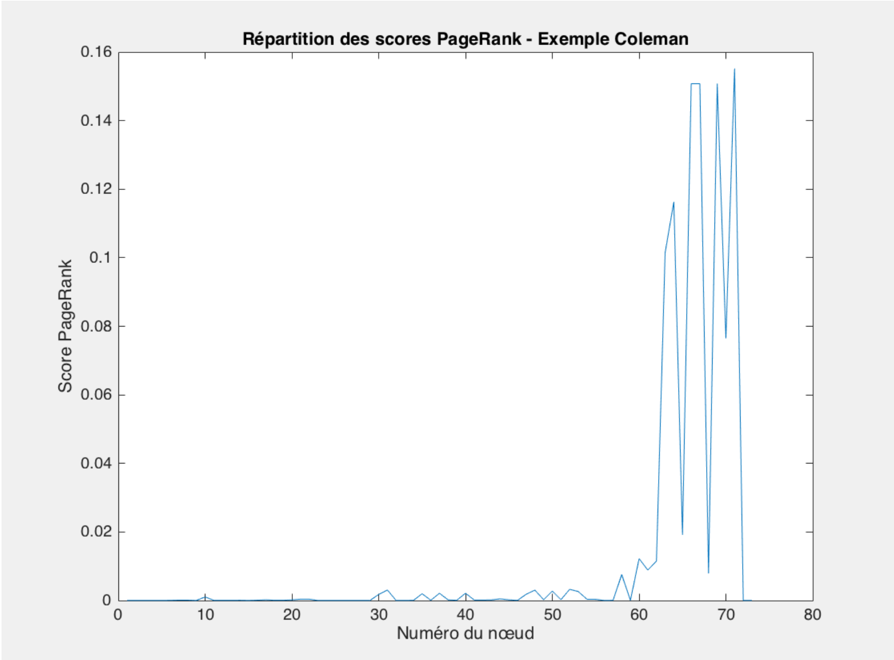

FIGURE 3 – Répartition des scores *PageRank* de l'exemple Coleman

60. $1222.70523 \cdot 10^{-5}$
61. $888.48113 \cdot 10^{-5}$
62. $1151.35563 \cdot 10^{-5}$
63. $10148.33608 \cdot 10^{-5}$
64. $11627.50856 \cdot 10^{-5}$
65. $1914.58235 \cdot 10^{-5}$
66. $15076.64624 \cdot 10^{-5}$
67. $15075.87701 \cdot 10^{-5}$
68. $792.70152 \cdot 10^{-5}$
69. $15080.85344 \cdot 10^{-5}$
70. $7648.49514 \cdot 10^{-5}$
71. $15517.87108 \cdot 10^{-5}$
72. $1.50673 \cdot 10^{-5}$
73. $1.50673 \cdot 10^{-5}$

Le classement est le suivant : 71 - 69 - 66 - 67 - 64 - 63 - 70 - 65 - 60 - 62 - 61 - 68 - 58 - 52 - 31 - 48 - 50 - 53 - 40 - 37 - 35 - 47 - 30 - 10 - 44 - 22 - 21 - 54 - 55 - 51 - 49 - 17 - 45 - 20 - 38 - 43 - 41 - 42 - 7 - 8 - 16 - 39 - 18 - 14 - 19 - 36 - 12 - 11 - 57 - 46 - 6 - 13 - 34 - 59 - 33 - 32 - 28 - 29 - 4 - 27 - 15 - 26 - 5 - 9 - 1 - 2 - 3 - 23 - 24 - 25 - 56 - 72 - 73. La figure 3 donne la répartition des scores *PageRank*.

Il est également important de remarquer que la somme de toutes les valeurs des vecteurs PageRank que nous venons de vous présenter vaut bien 1 (à 0.0001 près, étant donné notre algorithme de convergence).

B Code java

La méthode principale de notre code est la méthode *main*. Nous y récupérons les différents arguments du programme. Chacun de ceux-ci est un fichier contenant la matrice d'adjacance d'un graphe. Notre méthode va lancer la fonction *openFile* pour chaque fichier.

Dans cette méthode, le programme transforme les différentes lignes du fichier en une matrice d'adjacance et va lancer la méthode *pageRank* dessus. Cette méthode (au moyen des fonctions auxilières *rec, converge* et *normalize*) va récupérer les scores *PageRank*. Ensuite, cette méthode va afficher ces scores via la fonction *print* et ensuite le classement via la fonction *classement* avec sa fonction auxilière *maxIndice*.

```java
public static void main(String[] args) {
    System.out.println("Bienvenue␣dans␣ce␣programme␣de␣calcul␣PageRank!");
    switch (args.length) {
        case 0:
            System.out.println("Il␣n'y␣a␣pas␣de␣fichier␣à␣lire.");
            break;
        case 1:
            System.out.println("Il␣y␣a␣un␣fichier␣à␣lire.");
            break;
        default:
            System.out.println("Il␣y␣a␣" + args.length + "␣fichiers␣à␣lire.");
    }
    for (int i = 0; i < args.length; i++) {
        try {
            System.out.println("Fichier␣n" + (i+1));
            openFile(args[i]); //Tente d'ouvrir le fichier
        } catch (IOException e) {
            System.err.println("Error␣in␣file␣:␣" + args[i]);
        }
    }
}
```

TABLE 1 – Méthode main

```java
public static double[] pageRank(double[][] adj, double alpha, double[] pers) {
    normalize(adj); //Normalisation
    Matrix p = new Matrix(adj); //Création des matrices à envoyer à la fonction récusive
    Matrix v = (new Matrix(pers, 1)).transpose(); //Vecteur de personalisation
    Matrix x = new Matrix(p.getRowDimension(), 1, 1); //Vecteur de résultat en t=0
    Matrix result = rec(x, alpha, p, v, false, 0); //Calcul du résultat final
    return result.getRowPackedCopy(); //Renvoie un vecteur colonne
}
```

TABLE 2 – Méthode pageRank

```java
/**
* Fonction qui lit un fichier contenant une matrice formatée et lance le calcul de pageRank sur celle-ci
*/
public static void openFile(String filename) throws IOException {
    BufferedReader bf = new BufferedReader(new FileReader(filename)); // Initialisations des variables pour la lecture
    String line = bf.readLine();
    String[] tmp = line.split(","); //On sépare la chaine sur ',' pour obtenir toutes les valeurs
    int l = tmp.length;
    double[][] mat = new double[l][l];
    for (int x = 0; x < l; x++) { //Remplis le tableau
        tmp = line.split(",");
        for (int y = 0; y < l; y++) {
            try {
                mat[x][y] = Double.parseDouble(tmp[y]);
            } catch (NumberFormatException e) {
                System.err.println("Nombre à l'index (" + x + "," + y + ") mal encodé. Remplaçé par un 0.");
                mat[x][y] = 0;
            }
        }
        line = bf.readLine();
    }
    bf.close(); //fermer le buffer
    double[] q = new double[l];
    for (int x = 0; x < l; x++) { //On crée le vecteur de personnalisation, par défaut rempli de 1
        q[x] = 1;
    }
    double[] ranked = pageRank(mat, 1, q); //Lancement du calcul de pageRank
    print(ranked); //Affichage du résultat
    classement(ranked); //Affichage du classement
}
```

TABLE 3 – Méthode openFile

```java
/**
* Fonction récursive pour calculer le pagerank
* x(t+1)^T = apha * x(t)^T * P + (1-aplha)*pers^T
*/
public static Matrix rec(Matrix x, double alpha, Matrix p, Matrix v, boolean flag, int count){
    if ((flag && count > 10) || count > 5000) { //Fin de la récursion
        System.out.println("Il y a eu " + count + " récursions");
        return x;
    } else {
        Matrix xT = x.transpose(); //Calcul mathématique de l'algorithme
        Matrix vT = v.transpose();
        double minAplh = (1-alpha);
        Matrix temp = p.times(alpha);
        Matrix left = xT.times(temp);
        Matrix right = vT.times(minAplh);
        Matrix nXt = left.plus(right);
        double[] vecXt = nXt.getRowPackedCopy(); //Normalisation du résultat
        vecXt = normalize(vecXt);
        nXt = (new Matrix(vecXt, 1));
        flag=converge(x, nXt.transpose()); //Vérification de la convergence
        return rec(nXt.transpose(), alpha, p, v, flag, count+1); //Récursion
    }
}
```

TABLE 4 – Méthode rec

```java
/**
* Fonction qui vérifie si les deux matrices convergent, return true si elles sont égale, false sinon
* @ pre : a et b sont deux matrices ligne valides et de même tailles
* @ post : renvoie true si les valeurs de a et b convergent, false sinon
*/
public static boolean converge(Matrix a, Matrix b){
    double[] xA = a.getRowPackedCopy(); //Récupérer les deux matrices
    double[] xB = b.getRowPackedCopy();
    for(int i=0; i<xA.length; i++){
        if(xB[i]-xA[i]>0.0001) //Si on trouve deux valeurs avec plus de 0.0001 d'écart, les valeurs ne convergent pas, renvoyer false
            return false;
    }
    return true; //Si on est sorti de la boucle, les valeurs convergent
}
```

TABLE 5 – Méthode converge

```java
/**
* @ pre : a est un vecteur valide
* @ post : renvoie la version normalisée de a, c'est-à-dire que la somme
*           des éléments du vecteur renvoyée est égale à 1
*/
public static double[] normalize(double[] a) {
    double count = 0;
    for (int i = 0; i < a.length; i++)
        count += a[i];
    for (int i = 0; i < a.length; i++) {
        if (count != 0)
            a[i] = a[i] / count;
        else
            a[i] = 1.0 / a.length;
    }
    return a;
}
```

TABLE 6 – Méthode normalize (pour un vecteur)

```java
/**
* @ pre : a est une matrice d'adjacence valide (carrée)
* @ post : renvoie la version normalisée de a, c'est-à-dire avec chaque ligne
*           divisiée par le degré du noeud qu'elle représente
*           Si la ligne valait 0, chaque valeur est remplacée par 1/N où N vaut
    le nombre de noeud du graphe (téléportation possible)
*/
public static double[][] normalize(double[][] a) {
    double count;
    double[] vector = new double[a.length];
    for (int i = 0; i < a.length; i++) {
        count = 0;
        for (int j = 0; j < a[0].length; j++) {
            count += a[i][j]; //Pour chaque colonne, compter le degré du noeud
                qu'elle représente
        }
        vector[i] = count; //Stocker dans un vecteur
    }
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a[0].length; j++) {
            if (vector[i] != 0) {
                a[i][j] = a[i][j] / vector[i]; //Normaliser en divisant
            } else {
                a[i][j] = 1.0 / a.length;
            }
        }
    }
    return a;
}
```

TABLE 7 – Méthode normalize (pour une matrice)

```java
/**
* Affiche la repréentation d'un vecteur (en colonne)
*/
public static void print(double[] a) {
    for (int i = 0; i < a.length; i++) {
        System.out.println("\tNœud n" + (i+1) + " : " + a[i]);
    }
}
```

TABLE 8 – Méthode print

```java
/**
* Classe les valeurs du vecteur par ordre croissant
* @pre : un tableau de double
* @post : Affiche sur la sortie standard les indices du vecteur par ordre
   croissant
*/
public static void classement(double[] a) {
    System.out.print("\tClassement :");
    for (int i = 0; i < a.length; i++) {
        int imax = maxIndice(a); //Chercher l'indice de la valeur maximum
        System.out.print(" " + (imax+1) + " "); //Afficher le résultat
        a[imax] = Double.MIN_VALUE; //Stocker la valeur minimale dans la case
            pour ne plus qu'elle soit choisie
    }
    System.out.println();
}
```

TABLE 9 – Méthode classement

```java
/**
* Retourne l'indice de la valeur maximum contenue dans un tableau de double
*/
public static int maxIndice(double[] a) {
    double max = Double.MIN_VALUE; //Initialisation
    int imax = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] > max) { //Si la valeur de cette case est meilleure que celle
            qu'on a actuellement
            imax = i; //Stocker l'indice
            max = a[i]; //Stocker la valeur
        }
    }
    return imax;
}
```

TABLE 10 – Méthode maxIndice