Compte Rendu TME2

Wenzhuo ZHAO, Zhaojie LU, Zhen HOU, Chengyu YANG

Février 2021

1 Page Rank: Structure de données

Dans le pseudo code de l'algorithme Page Rank vu dans le cours 2, le calcul est réalisé par des produit de matrice de transition **T** par le vector de Page Rank **P**. Cependant, pour un graphe orienté assez grand contenant plus de 10^6 de noeuds et plus de 10^7 liens, il est impossible de stocker une matrice dans la mémoire d'un ordinateur ordinaire. Il nous faut donc trouver une solution de faire la multiplication sur **P** sans utiliser la matrice **T** au cours de la méthode *PowerIteration*.

1.1 Optimisation de mémoire

Pour tout lien orienté (u, v) dans le graphe **G** avec le nombre de liens n, nous avons la matrice $T(n, n)$ où $T_{vu} = 1/d_{out}(u)$ ($d_{out}(u)$ est degré sortant de noeud u) et le vector initial de Page Rank $P = 1/n * n$. Pour remplacer la multiplication, prenons un vector temporaire $page_rank_temp = 0 * n$. Voici le pseudo code de notre implémentation qui est effective par rapport au pseudo code dans la slide 19 de cours 2.

Algorithm 1: Page Rank

Data: List *edges*
Data: List *pages*
Result: List *pageRank*
`/* nombre de noeuds et nombre de liens */`
n = length(*pages*);
e = length(*edges*);
`/* précision de différence entre 2 itérations */`
eps = 100000 ;
`/* Vector Page Rank: rempli par` $1/n$ `répété` n `fois */`
pageRank = vector($1/n$, n);
`/* Vector Page Rank temporaire: rempli par 0 répété` n `fois */`
pageRankTemp = vector(0, n);
while *eps > 0.00001* **do**
  for *edge in edges* **do**
    source = getSource(*edge*);
    destination = getDestination(*edge*);
    pageRankTemp[*destination*] += *pageRankTemp*[*source*] / *degreeOut*[*source*] ;
  pageRankTemp = (1 - *alpha*) * *pageRankTemp* + *alpha* / n ;
  `/* normalize: p += (1 -` $||P||_1$`) / n */`
  normalize(*pageRankTemp*) ;
  `/* Avant d'affectation: Calcul de différence entre le page rank ancien et celui nouveau itéré */`
  eps = difference(*pageTank*, *pageRankTemp*);
  pageRank = *pageRankTemp*;
  `/* réinitialiser le pageRankTemp */`
  pageRankTemp = vector(0, n);
return pageRank

1.2 Tests

Nous testons notre programme implémenté en Python avec le jeu de données du graphe des liens de WikiPedia [4]. Prenons les 5 pages ayant la plus haute valeur de Page Rank et les 5 pages ayant la plus basse valeur de Page Rank.

```
Top 5 highest page rank nodes:
ID: 3434750 Name: United States Page Rank value: 0.003631807898328141
ID: 31717 Name: United Kingdom Page Rank value: 0.001589414349814768
ID: 11867 Name: Germany Page Rank value: 0.0013618033701903356
ID: 36165 Name: 2007 Page Rank value: 0.0013591697085100147
ID: 36164 Name: 2006 Page Rank value: 0.0013551384688190596
Top 5 lowest page rank nodes:
ID: 632 Name: Aberdeen (disambiguation) Page Rank value: 7.305691005495847e-08
ID: 679 Name: Animal (disambiguation) Page Rank value: 7.305691005495847e-08
ID: 951 Name: Antigua and Barbuda Page Rank value: 7.305691005495847e-08
ID: 964 Name: AWK (disambiguation) Page Rank value: 7.305691005495847e-08
ID: 1110 Name: Demographics of American Samoa Page Rank value: 7.305691005495847e-08
```

Les résultats sont raisonnables. Les pages qui ont un maximum degré entrant comme les pages de pays sont introduits dans des autres pages peuvent avoir une valeur de Page Rank assez haute. Les pages qui sont moins cherchées ont une valeur basse.

2 Corrélations

Nous illustrons les corrélations entre la valeur Page Rank, la veleur alpha et le degré sortant/entrant de chaque noeud.

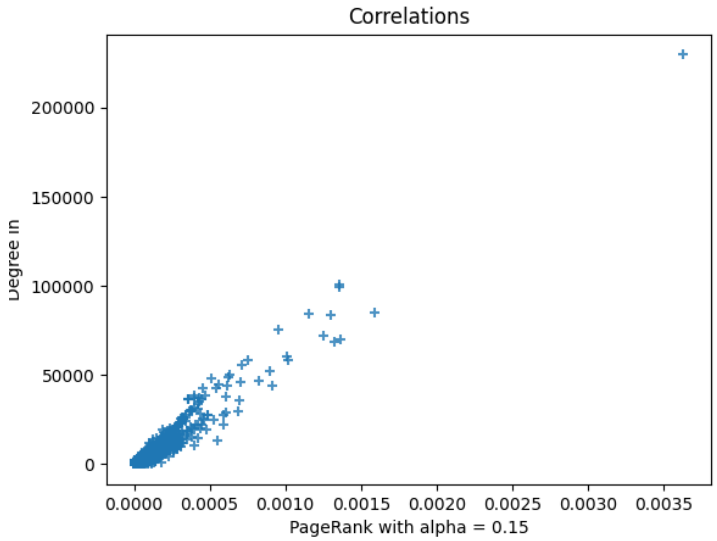

Figure 1: Corrélation entre les Page Rank valeurs avec alpha = 0.15 et degré entrant de chaque noeud en échelle linéaire

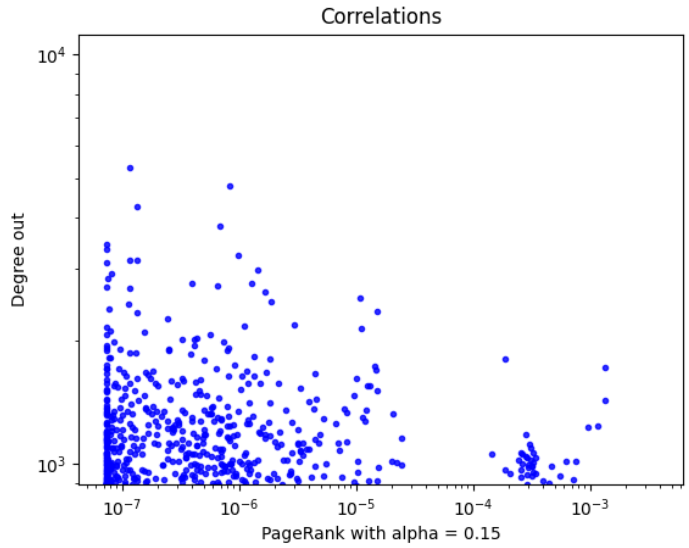

Figure 2: Corrélation entre les Page Rank valeurs avec alpha = 0.15 et degré sortant de chaque noeud en échelle logarithmique

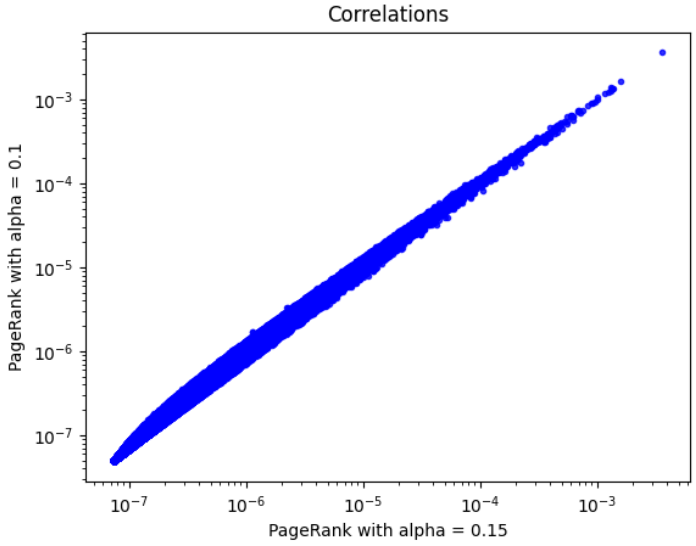

Figure 3: Corrélation entre les Page Rank valeurs avec alpha = 0.15 et les Page Rank valeurs avec alpha = 0.1 en échelle logarithmique

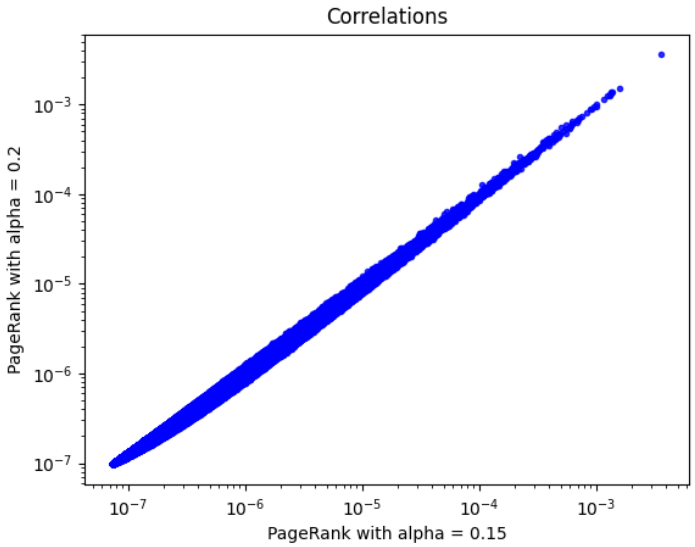

Figure 4: Corrélation entre les Page Rank valeurs avec alpha = 0.15 et les Page Rank valeurs avec alpha = 0.2 en échelle logarithmique

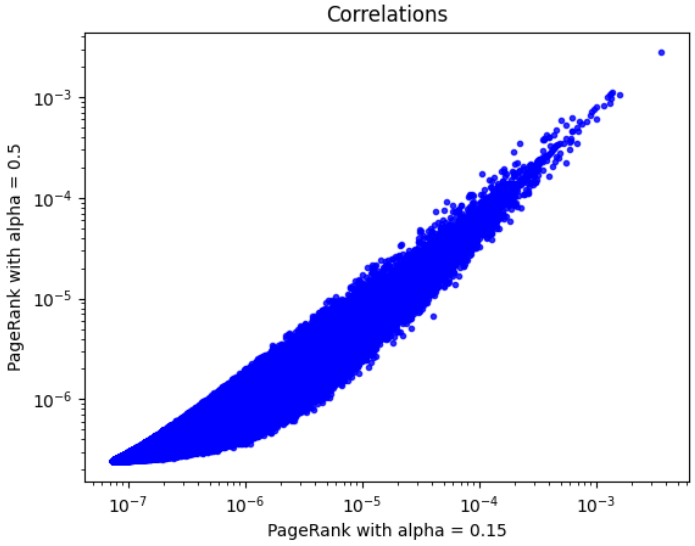

Figure 5: Corrélation entre les Page Rank valeurs avec alpha = 0.15 et les Page Rank valeurs avec alpha = 0.5 en échelle logarithmique

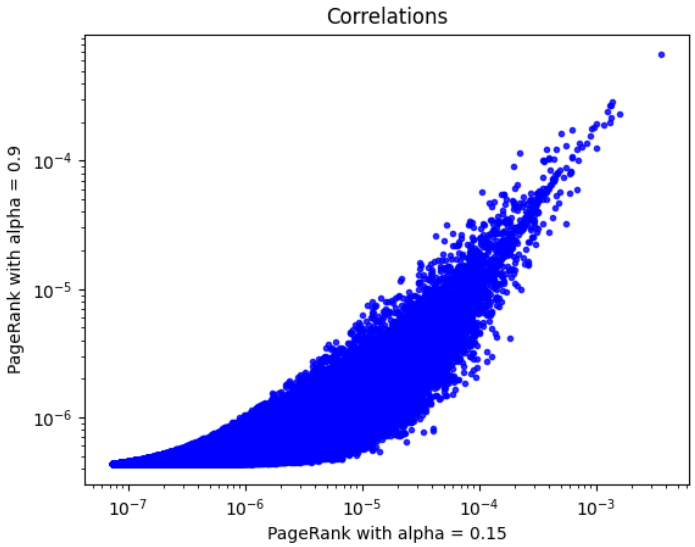

Figure 6: Corrélation entre les Page Rank valeurs avec alpha = 0.15 et les Page Rank valeurs avec alpha = 0.9 en échelle logarithmique

Nous pouvons voir sur le figure 11 que il y a une corrélation positive entre la valeur de Page Rank et le degré entrant. Pour les comparaisons de valeurs Page Rank sur de différentes valeur α, le Page Rank avec α entre 0.1 et 0.2 a une partition de valeurs plus continue dans son intervalle.

3 Personalized Page Rank

Le Personalized Page Rank se contente à mettre plus de probabilité de visiter les pages dont les utilisateurs s'intéressent plus. Prenons un vector initial $page_rank_rooted$ et un ensemble de pages $interested_pages$ avec le nombre d'éléments n qui s'intéressent aux utilisateurs, ce vector satisfait le principe:

$$PageRankRooted[page] = \begin{cases} 1/n, & \text{if } page \text{ in } interestedPages \\ 0, & \text{else} \end{cases} \quad (1)$$

Ce principe est équivalent à $||PageRankRooted||_1 = 1$.

Nous calculons le Personalized Page Rank valeurs en mettant plus de probabilité sur les pages dans ce vector.

Algorithm 2: Personalized Page Rank

```
Data: List edges
Data: List pages
Data: List interestedPages
Result: List pageRank
/* nombre de noeuds et nombre de liens                                    */
n = length(pages);
e = length(edges);
/* précision de différence entre 2 itérations                             */
eps = 100000 ;
/* Rooted Page Rank                                                       */
pageRankRooted = vector(0, n);
for page in interestedPages do
    pageRankRooted[page] += 1 / length(interestedPages)
/* Vector Page Rank:  rempli par 1/n répété n fois                        */
pageRank = vector(1/n, n);
/* Vector Page Rank temporaire:  rempli par 0 répété n fois               */
pageRankTemp = vector(0, n);
while eps > 0.00001 do
    for edge in edges do
        source = getSource(edge);
        destination = getDestination(edge);
        pageRankTemp[destination] += pageRankTemp[source] / degreeOut[source] ;
    pageRankTemp = (1 - alpha) * pageRankTemp + alpha * pageRankRooted ;
    /* normalize2:  p += pageRankRooted * (1 - ||P||_1)                   */
    normalize2(pageRankTemp, pageRankRooted) ;
    /* Avant d'affectation:  Calcul de différence entre le page rank ancien
       et celui nouveau itéré                                             */
    eps = difference(pageTank, pageRankTemp);
    pageRank = pageRankTemp;
    /* réinitialiser le pageRankTemp                                      */
    pageRankTemp = vector(0, n);
return pageRank
```

Pour donner un restart vector *interestedPgaes*, il est simple de trouver tous les pages qui sont dans la catégorie s'intéressant aux utilisateurs, par exemple Chess ou Boxing. Ces pages constituent le vector *interestedPgaes* et nous aurons un résultat où les pages dans la catégorie de Chess ou Boxing ont une plus grande de probabilité d'être visités.
Voici deux extraits de résultats après l'exécution de cette procédure sur les catégories Chess ou Boxing.

```
Calculating Personalized Page Rank value of categories Chess with alpha = 0.15

Top 5 highest page rank nodes:
ID: 5134 Name: Chess Page Rank value: 0.09969391714555559
ID: 47653 Name: Game clock Page Rank value: 0.05123532707520636
ID: 559997 Name: List of chess topics Page Rank value: 0.05026073709130644
ID: 72171 Name: Time control Page Rank value: 0.003215089599670034
ID: 2073557 Name: Bobby Fischer Page Rank value: 0.0031825794007963163
```

```
Calculating Personalized Page Rank value of categories Boxing with alpha = 0.15

Top 5 highest page rank nodes:
ID: 4243 Name: Boxing Page Rank value: 0.011791092419571624
ID: 12206656 Name: Miscellaneous of Boxing Page Rank value: 0.0036626454816693836
ID: 3434750 Name: United States Page Rank value: 0.003594474380712173
ID: 4284031 Name: Amateur boxing Page Rank value: 0.003026819456250737
ID: 12206918 Name: List of Quadruple Champions of Boxing Page Rank value: 0.002856869
```

4 Push method

La méthode *push*[1] de Page Rank est une méthode pour calculer un approximative vector de Page Rank avec une basse complexité en temps. Nous mettons à jour le vector Page Rank $p = vector(0, n)$ et le vector résiduel r qui est

$$r[page] = \begin{cases} 1/n, & \text{if } page \text{ in } interestedPages \\ 0, & \text{else} \end{cases} \tag{2}$$

La méthode push concentre sur la liasion entre noeuds donc il nous faut préparer une liste adjacente du graphe. Nous maintenons une pile *list* contenant les noeuds avec $r(u)/d(u) \geq \epsilon$. Sur chaque étape, l'opération push est exécutée sur la tête de pile dépilé jusqu'à $r(u)/d(u) < \epsilon$ pour ce noeud. Si durant l'opération push, un noeud x est trouvé que $r(u)/d(u) \geq \epsilon$ alors ce noeud x est empilé dans le pile. Répétons cette procédure jusqu'au pile est vide. En lisant le code sur GitHub[3], nous avons implémenté cet algorithme en Python et voici le pseudo code.

Algorithm 3: Push Page Rank

```
Data: List edges
Data: List pages
Data: List interestedPages
Data: Dictionary adjArray
Result: List pageRank
/* nombre de noeuds et nombre de liens                                  */
n = length(pages);
e = length(edges);
/* précision de différence entre 2 itérations                           */
eps = 0.00001 ;
/* Vectors p et r                                                       */
p, r = vector(0, n);
for page in interestedPages do
    r[page] = 1 / interestedPages.length()
/* la pile contenant les noeuds considérés à push                       */
queue = [];
for source in interestedPages do
    if eps * d[source] < 1 then
        queue.put(source)
while queue.length() > 0 do
    u = queue.pop();
    p[u] += alpha * r[u] ;
    tmp = (1 - alpha) * r[u] / 2;
    r[u] = tmp;
    /* tous les noeuds v connectés au noeud u                           */
    for v in adjArray[u] do
        r[v] += tmp / d[u];
        if r[v] /d[v] < eps then
            queue.put(v)
return p
```

4.1 Performance

Exécutons l'implémentation simple de Personnalized Page Rank et celle de cette méthode push sur un même ensemble de données, une même valeur α et un même vector de pages d'intérêt en fonction du temps.

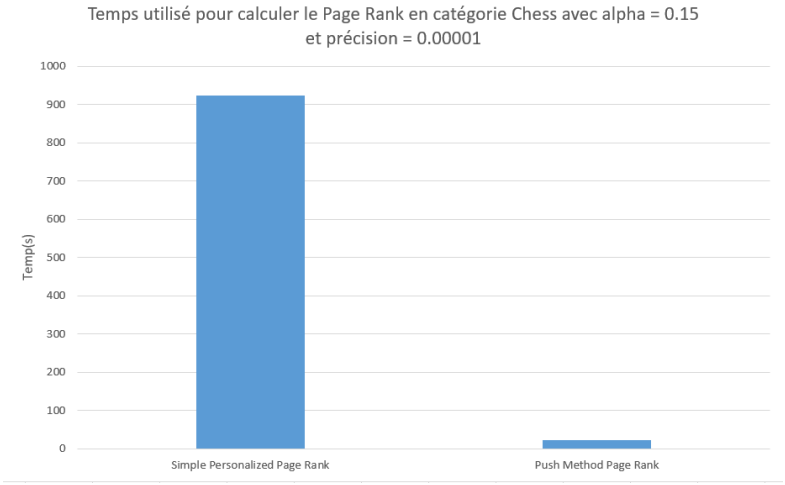

Figure 7: Performance de Personalized Page Rank simple et de Push méthode

Bien évidemment, ce nouvel algorithme est assez compétent par rapport à l'algorithme simple. Il est aussi effectif en fonction de rang de Page Rank valeurs de pages.

Appendix A Source Code

Le source code de ce TME est disponible sur notre répertoire de GitHub [2].

References

[1] *Local Graph Partitioning using PageRank Vectors.*

[2] *Page Rank Implémentation en Python.* `https://github.com/valeeraZ/Sorbonne_CPA_Graph/tree/master/TME2`.

[3] *Page Rank Push Implémentation en C.* `https://github.com/maxdan94/push`.

[4] *Wikipedia: Network of pages, Page categories, Category hierarchy.* `http://cfinder.org/wiki/?n=Main.Data`.