Rapport de TP - Big Data
Page Rank

Alexis Evaristo
Hector Baril

30 janvier 2026

Polytech Lille, Informatique, Statistique et Intelligence Artificielle

Table des matières

1 Introduction

L'objectif de ce projet est d'implémenter et d'analyser l'algorithme PageRank en utilisant la méthode de la puissance. Nous appliquons cette méthode sur un jeu de données réel issu du projet *Wikispeedia*, fourni par le SNAP (Stanford Network Analysis Project).

Le PageRank, initialement conçu pour classer les pages web, mesure l'importance relative d'un nœud au sein d'un graphe. Dans ce rapport, nous détaillons d'abord le traitement des données et la construction de la matrice de transition. Ensuite, nous analysons les résultats de l'algorithme standard en étudiant sa convergence et la distribution des scores (Scan & Add). Enfin, nous explorons une variation de l'algorithme via le PageRank Personnalisé (PPR) pour mettre en évidence des thématiques spécifiques.

2 Les Données

2.1 Description et Structure

Les données utilisées proviennent de l'archive `wikispeedia_paths-and-graph.tar.gz`. Pour cette étude, nous nous sommes concentrés principalement sur le fichier `paths_finished.tsv`. Ce fichier contient les chemins de navigation complets effectués par des utilisateurs humains tentant de relier deux articles Wikipédia donnés.

Chaque ligne du fichier représente une session de navigation réussie. Les données brutes nécessitent un prétraitement important, notamment pour gérer les retours en arrière (symbolisés par le caractère `<`).

2.2 Nettoyage et Gestion

La construction de la matrice d'adjacence L a été réalisée en Python. Le défi principal réside dans l'interprétation des chemins de navigation.

Nous avons implémenté une logique de **pile (stack)** pour simuler le parcours utilisateur :

— Lorsqu'une page est visitée, elle est ajoutée à la pile.

— Lorsque le symbole `<` est rencontré, la dernière page est retirée de la pile (retour arrière).

— Un lien est créé (valeur 1 dans la matrice) entre la page précédente (sommet de la pile avant déplacement) et la page actuelle.

Ce processus permet d'obtenir une matrice représentant fidèlement les clics effectifs des utilisateurs.

2.3 Outils Utilisés

Le projet a été réalisé en langage **Python**. Les principales bibliothèques utilisées sont :

— `Numpy` : Pour les opérations matricielles et l'algèbre linéaire.

— `Matplotlib` : Pour la génération des graphiques (courbes de convergence, Scan & Add).

— `Csv` : Pour la lecture optimisée des fichiers de données.

3 PageRank Standard

3.1 Choix des structures de données

Nous avons opté pour l'utilisation de tableaux `Numpy` pour représenter la matrice d'adjacence et les vecteurs de probabilité. Un mappage bidirectionnel (Dictionnaire `Nom` $\rightarrow$ `Index` et Liste `Index` $\rightarrow$ `Nom`) a été mis en place pour faire le lien entre les indices matriciels et les titres des articles.

L'algorithme implémenté suit la formule itérative de la méthode de la puissance :

$$q_{k+1} = \beta P q_k + (1-\beta)\mathbf{v}\mathbf{e}^t q_k$$

Où P est la transposée de la matrice d'adjacence normalisée par ligne, β le facteur d'amortissement ou *dumping factor*, N le nombre total de pages, $\mathbf{v} = (1/N, 1/N, ..., 1/N)^t$, et $\mathbf{e} = (1, 1, ..., 1)$.

Les colonnes vides (remplies de zéros) de P, seront remplies par le facteur $1/N$.

Notons, qu'il faut normaliser q_k à chaque fin d'itération (on utilisera $\|q_k\|_1$, la norme 1).

Pour le PageRank simple, après simplification, on obtient la formule suivante :

$$q_{k+1} = \beta P q_k + \frac{(1-\beta)}{N} \sum_{i=1}^{N} q_i \ . \ (1, 1, ..., 1)^t$$

3.2 Expérimentations et Résultats

3.2.1 Analyse de la Convergence

Nous avons étudié l'impact des paramètres β (damping factor) et ϵ (précision) sur la vitesse de convergence.

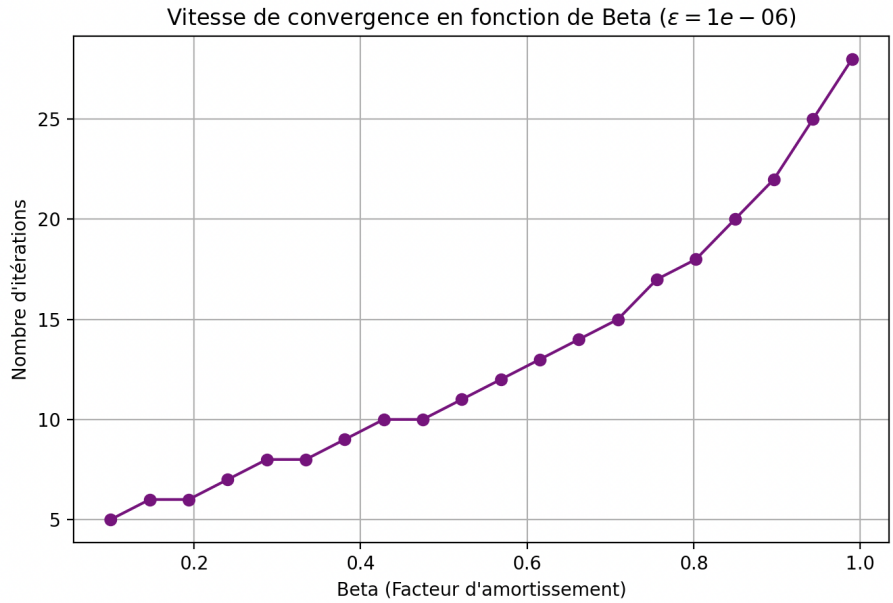

FIGURE 1 – Vitesse de convergence du PageRank simple en fonction de β

Interprétation : *On observe que plus β est proche de 1, plus la convergence est lente, car le poids de la matrice structurelle est plus fort par rapport au terme de téléportation.*

3.2.2 Précision (ϵ)

La figure suivante illustre l'impact de la précision demandée sur le nombre d'itérations.

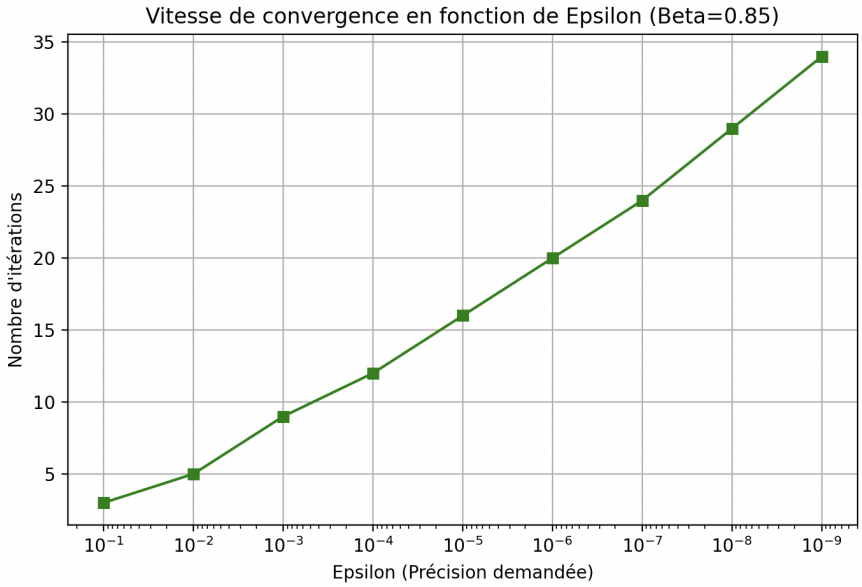

FIGURE 2 – Nombre d'itérations nécessaires pour converger selon la précision ϵ.

3.2.3 Analyse Scan & Add

La méthode "Scan & Add" permet de visualiser la concentration de l'information dans le vecteur PageRank.

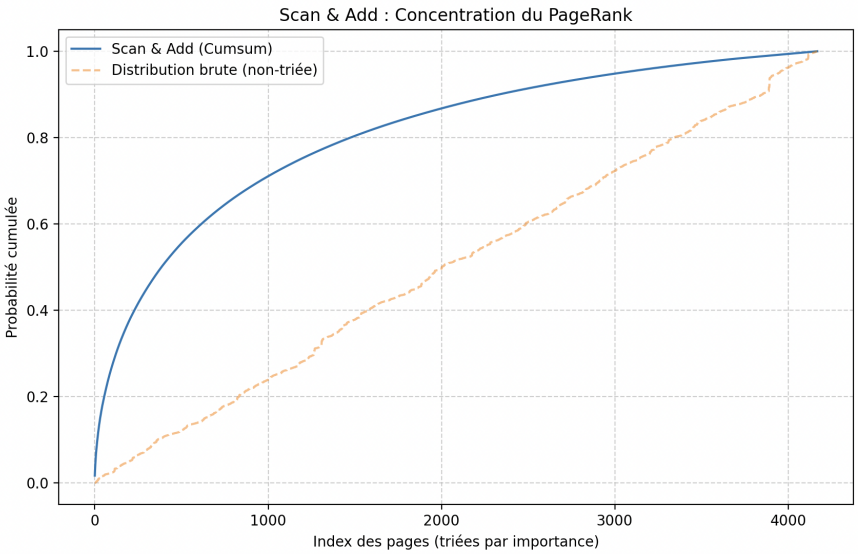

FIGURE 3 – Courbe Scan & Add (Somme cumulée des scores triés).

Analyse : La courbe montre une forte concentration de l'importance sur un nombre réduit de pages.

3.2.4 Temps de calcul et nombre d'itération en fonction de β

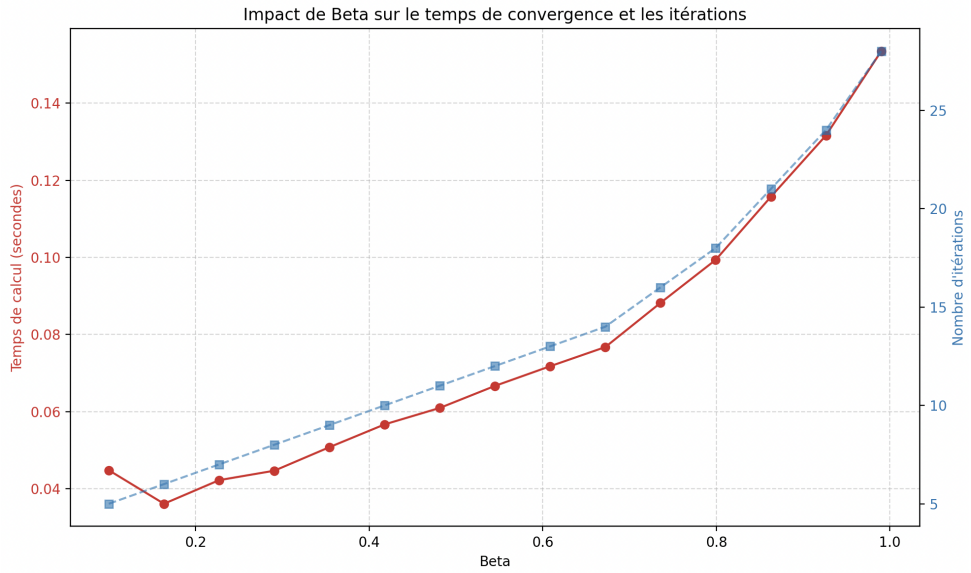

FIGURE 4 – Évolution de la convergence en fonction de β (ϵ = 1e-6)

3.3 Analyse du Top 20

Voici les 20 pages les plus importantes identifiées par l'algorithme avec $\beta = 0.85$.

Rang	Page	Score
1	United_States	0.016869
2	Europe	0.009784
3	United_Kingdom	0.009187
4	England	0.007301
5	World_War_II	0.006960
6	France	0.006916
7	Germany	0.005464
8	English_language	0.005241
9	Africa	0.004917
10	India	0.004394
11	Latin	0.004352
12	Japan	0.004153
13	China	0.004069
14	United_Nations	0.003965
15	North_America	0.003824
16	Italy	0.003818
17	London	0.003752
18	Russia	0.003717
19	Earth	0.003513
20	Australia	0.003368

TABLE 1 – Top 20 des pages selon le PageRank standard ($\beta = 0.85$).

On remarque une prédominance des pays et des concepts géographiques majeurs, ce qui s'explique par la nature du jeu Wikispeedia où ces pages servent de "hubs" de navigation.

4 PageRank Personnalisé (PPR)

4.1 Méthodologie

Le PageRank Personnalisé modifie le terme de téléportation. Au lieu de se redistribuer uniformément sur toutes les pages en cas de téléportation, le surfeur aléatoire revient préférentiellement vers un sous-ensemble de pages spécifiques (le vecteur de personnalisation v).

Nous avons testé cette approche avec un facteur de pondération $s = 3$ et en ciblant des pages spécifiques.

4.2 Expérimentation sur des thèmes

Nous avons choisi de personnaliser le vecteur sur les nœuds suivants : *[Russia, Communism, Socialism.*

Rang	Page	Score
1	Communism	0.054928
2	Russia	0.054851
3	Socialism	0.053294
4	United_States	0.013942
5	Europe	0.009572
6	World_War_II	0.009286
7	United_Kingdom	0.008054
8	Soviet_Union	0.006977
9	France	0.006590
10	China	0.005055
11	Germany	0.004870
12	World_War_I	0.004797
13	United_Nations	0.004650
14	England	0.004576
15	People_Republic_of_China	0.004546
16	Italy	0.004298
17	English_language	0.004244
18	Marxism	0.004217
19	Vladimir_Lenin	0.004118
20	Capitalism	0.004075

TABLE 2 – Top 20 des pages selon le PageRank standard ($\beta = 0.85$).

Page	Rang (Standard)	Rang (PPR)
Communism	89	1
Russia	18	2
Socialism	223	3
Soviet_Union	38	8
Marxism	365	18
Vladimir_Lenin	798	19

TABLE 3 – Comparaison des rangs entre PageRank Standard et Personnalisé.

Synthèse et analyse : *Pour commencer, les trois nœuds sélectionnés sont en top de classement. Il est possible de voir que des pages de sujet connexes ont grandement avancée dans*

le classement, par exemple "Soviet_ Union", "Marxism", "Vladimir_ Lenin". Également, il est possible de remarquer que les pays sont toujours aussi présents dans le top ranking.

5 Conclusion

Ce travail pratique nous a permis de mettre en œuvre l'algorithme PageRank sur des données réelles. Nous avons pu constater la robustesse de la méthode de la puissance et analyser l'impact des hyperparamètres β et ϵ sur la convergence.

L'analyse des résultats sur Wikispeedia confirme que la structure de navigation humaine s'appuie fortement sur des concepts généraux (pays, guerres mondiales) pour transiter d'un sujet à l'autre. Le PageRank Personnalisé a démontré sa capacité à biaiser ces résultats pour favoriser des thématiques spécifiques, ouvrant la voie à des moteurs de recherche contextuels.

A Annexes : Codes Python

L'intégralité du code a été développé en Python 3. Voici les fonctions principales utilisées pour la construction de la matrice et le calcul du PageRank.

```
1
2  def get_mappings_and_pages(filepath):
3      """
4      Lit le fichier pour identifier toutes les pages uniques.
5      Retourne :
6          - page_to_idx : dictionnaire {nom_page: index}
7          - pages_list : liste ou pages_list[i] donne le nom de la page a l'index
       i
8      """
9      unique_pages = set()
10
11     with open(filepath, "r", encoding="utf-8") as fh:
12         reader = csv.reader(fh, delimiter="\t")
13         for ligne in reader:
14             # On verifie que la ligne a bien le bon format
15             if len(ligne) > 3:
16                 consult = ligne[3].split(";")
17                 for page in consult:
18                     if page != "<":
19                         unique_pages.add(page)
20
21     # On trie pour avoir un ordre deterministe
22     pages_list = sorted(list(unique_pages))
23
24     # Comprehension de dictionnaire pour creer le mapping inverse
25     page_to_idx = {page: i for i, page in enumerate(pages_list)}
26
27     return page_to_idx, pages_list
```

Listing 1 – Code pour associer les pages à des indexs

```
1
2  def build_adjacency_matrix(filepath, page_to_idx, N):
3      """
4      Construit la matrice d'adjacence L en utilisant la methode de la PILE.
5      Respecte strictement la logique originale.
6      """
7      L = np.zeros((N, N), dtype=np.float64)
8
9      with open(filepath, "r", encoding="utf-8") as fh:
10         reader = csv.reader(fh, delimiter="\t")
11
12         for ligne in reader:
13             if len(ligne) <= 3:
14                 continue
15
16             consult = ligne[3].split(";")
17             stack = []  # Initialisation de la pile pour ce chemin
18
19             for token in consult:
20                 if token == "<":
21                     # Retour arriere : on depile
22                     if stack:
23                         stack.pop()
24                 else:
25                     # C'est une page visitee
26                     page_actuelle = token
27
```

```
                    # Si la pile n'est pas vide, le sommet est la page precedente (source)
                    if stack:
                        page_precedente = stack[-1]

                        i_source = page_to_idx[page_precedente]
                        j_target = page_to_idx[page_actuelle]

                        L[i_source][j_target] = 1

                    # On empile la page actuelle (elle devient potentielle source)
                    stack.append(page_actuelle)
    return L
```

Listing 2 – Code pour créer la matrice d'adjacence

```

def normalize_matrix(L, N):
    """
    Gere les 'dangling nodes' (lignes de zeros) et normalise par ligne.
    """
    # 1. Gestion des culs-de-sac (Dangling nodes)
    # Si une page ne pointe vers rien, on suppose qu'elle pointe vers tout le monde (surf aleatoire)
    for i in range(N):
        if np.sum(L[i]) == 0:
            L[i, :] = 1.0 / N

    # 2. Normalisation stochastique (la somme de chaque ligne doit faire 1)
    # On divise chaque ligne par sa somme
    row_sums = L.sum(axis=1)
    # astuce numpy : on divise la matrice par le vecteur colonne des sommes
    L = L / row_sums[:, np.newaxis]

    return L
```

Listing 3 – Code pour obtenir la matrice de transition

```

def compute_pagerank(beta, epsilon, P, v, s, N, max_iter=1000):
    """
    Calcule le vecteur PageRank q.
    P : Transposee de la matrice d'adjacence normalisee (L.T)
    v : Vecteur de personnalisation
    s : Somme de v (ou facteur de normalisation de v)
    """

    # Demarrage du chronometre
    start_time = time.perf_counter()

    # Initialisation uniforme
    q = np.ones(N) / N
    q = q / nla.norm(q, 1)

    iterations = 0
    diff = epsilon + 1.0  # Pour entrer dans la boucle

    while diff > epsilon and iterations < max_iter:
        q_old = q.copy()
        sum_q = np.sum(q)

        # Formule originale :
        # q = beta * P * q + terme de teleportation
```

```
        q = beta * (P @ q)
        q = q + ((1 - beta) / s * sum_q * v)

        # Renormalisation (securite numerique)
        q = q / nla.norm(q, 1)

        # Calcul de la difference pour la convergence
        diff = nla.norm(q - q_old, 1)
        iterations += 1

    # Arret du chronometre
    end_time = time.perf_counter()
    execution_time = end_time - start_time

    return q, iterations, execution_time
```

Listing 4 – Code du PageRank

```

if __name__ == "__main__":
    FILE_PATH = "./paths_finished.tsv"

    print("1. Chargement et Mapping...")
    # On recupere le dictionnaire (nom->idx) et la liste (idx->nom)
    page_to_idx, idx_to_page = get_mappings_and_pages(FILE_PATH)

    N = len(pages_list := idx_to_page)  # Syntaxe walrus (python 3.8+)
    print(f"   Nombre de pages uniques : {N}")

    print("2. Construction de la matrice L...")
    L = build_adjacency_matrix(FILE_PATH, page_to_idx, N)

    print("3. Normalisation...")
    L = normalize_matrix(L, N)

    # Preparation pour PageRank
    # On utilise la transposee pour l'equation P @ q
    P = L.T

    # Configuration du vecteur de personnalisation (v)
    v = np.ones(N)  # = np.zeros(N) dans le PPR
    indices_cles = [3237, 3422, 919]  # Indices specifiques

    # Verification que les indices existent (pour eviter crash si fichier
    different)
    valid_indices = [k for k in indices_cles if k < N]
    if valid_indices:
        v[valid_indices] = 1
    else:
        # Fallback si indices hors limites, on met tout a 1
        v = np.ones(N)

    s = np.sum(v)  # Somme pour la normalisation dans la formule
    if s == 0:
        s = 1  # Securite division par zero

    # --- Calcul Principal ---
    print("4. Calcul du PageRank...")
    beta_val = 0.85
    epsilon_val = 0.000001

    final_q, iterations, duration = compute_pagerank(beta_val, epsilon_val, P, v
    , s, N)

```

```
    print(f"   Convergence en {iterations} iterations et {duration:.4f} secondes.")

    # --- Affichage des resultats (Top 20) ---
    print("\n--- TOP 10 PAGES ---")
    # On associe score et index, on trie, puis on recupere le nom
    top_indices = np.argsort(final_q)[:20]
    for idx in top_indices:
        print(f"Score: {final_q[idx]:.6f} | Page: {idx_to_page[idx]} (Index: {idx})")
```

Listing 5 – Code du PageRank