

Projet 1 AAGA 2025

Breton Noé

n°21516014

Boudrouss Réda

n°28712638

Durbin Deniz Ali

n°21111116



Sorbonne Université
France

Table des matières

1. Introduction	2
1.1. Contexte	2
1.2. Objectifs	2
1.3. Choix techniques	2
2. Implémentation	3
2.1. PageRank	3
2.1.1. Principe	3
2.1.2. Implémentation	3
2.2. Personalized PageRank	4
2.2.1. Principe	4
2.2.2. Implémentation	4
2.3. Méthode PUSH	5
2.3.1. Principe	5
2.3.2. Implémentation	5
3. Résultats et analyse	6
3.1. Méthodologie expérimentale	6
3.2. Performance des algorithmes	6
3.2.1. PageRank	6
3.2.2. Personalized PageRank	7
3.2.3. PUSH	8
3.2.4. Comparaison PUSH vs PPR	8
3.3. Détection de communautés	10
4. Conclusion	10
4.1. Synthèse des résultats	10
4.2. Limites et observations	10
4.3. Pistes d'amélioration	11

1. Introduction

Tout le code pour ce projet est disponible sur github https://github.com/rboudrouss/aaga_projet

1.1. Contexte

L'algorithme PageRank, conçu à la fin des années 1990 par les fondateurs de Google, permet de mesurer l'importance relative des pages web en fonction de la structure de leurs liens hypertextes.

On suppose qu'un utilisateur parcourt le web en cliquant aléatoirement sur les liens disponibles. Un "score" ou "rank" est alors attribué à chaque page par l'algorithme. D'un point de vue probabiliste, On peut dire que ce score représente la probabilité de visiter cette page après de nombreuses navigations.

Au début du calcul, cette probabilité est répartie uniformément entre toutes les pages du graphe. L'algorithme ajuste ensuite, par itérations successives, les valeurs de chaque rank jusqu'à atteindre une distribution stable. Par conséquent une page est considérée comme importante si elle est référencée par d'autres pages importantes.

Par ailleurs, PageRank dépasse aujourd'hui son usage initial et constitue une mesure de centralité largement utilisée en analyse de graphes, que ce soit dans des domaines comme les réseaux sociaux (pour identifier les utilisateurs les plus influents), la bio-informatique (pour étudier les interactions entre protéines ou gènes) ou encore l'analyse de citations scientifiques (pour mesurer l'impact d'un article ou d'un auteur).

Pour certaines de ces applications, il n'est pas suffisant de connaître l'importance globale de chaque nœud dans le graphe. Il peut être plus pertinent et moins coûteux de se concentrer sur l'influence locale d'un nœud particulier ou d'un petit sous-ensemble de nœuds. La variante PPR (Personalized PageRank) répond à ce besoin.

Ici, la téléportation ne se fait pas uniformément vers tous les sommets du graphe, mais est concentrée sur un sommet source spécifique. Cette personnalisation permet de mesurer la proximité ou l'influence des autres nœuds par rapport à cette source, donnant un aperçu des communautés locales et de la structure du graphe autour d'un point d'intérêt particulier. Cependant, son calcul se révèle coûteux en temps et en mémoire sur de grand graphe ou sur de trop multiple sources.

La méthode PUSH propose une approche approximative mais efficace. Elle permet de concentrer le calcul sur les parties du graphe les plus pertinentes autour de la source, réduisant ainsi considérablement le temps de calcul.

1.2. Objectifs

Les objectifs de notre projet sont les suivants :

- Implémenter et étudier la convergence du PageRank classique et du PPR, en fonction de la précision souhaitée.
- Détecter et analyser les communautés locales autour d'un nœud source à l'aide du PPR.
- Mettre en œuvre une approximation du PPR via la méthode PUSH et comparer les résultats obtenus au PPR, en termes de précision que de temps de calcul.

1.3. Choix techniques

Pour l'implémentation de ces algorithmes, nous avons choisi d'utiliser TypeScript avec Deno comme runtime. Ce choix a été en partie par l'expérience et les préférences des membres du groupe, mais il a également des avantages techniques. Le typage statique rend le code plus sûr et plus robuste et sa syntaxe proche du langage naturel facilite la lecture et la compréhension.

Nous avons également utilisé Python pour la visualisation des résultats, en particulier avec les bibliothèques Matplotlib et NumPy pour tracer des graphiques.

Une cli a été développée pour faciliter l'interaction avec le code et lancer les différents tests. Pour l'exécuter, il suffit de lancer la commande suivante :

```
./cli [command] [options]
```

Pour plus d'information, veuillez vous référer au README.md

2. Implémentation

2.1. PageRank

2.1.1. Principe

L'algorithme PageRank mesure l'importance relative des nœuds dans un graphe orienté selon le principe du random surfer : un utilisateur navigue aléatoirement en suivant les liens, et le PageRank d'un nœud correspond à la probabilité stationnaire que cet utilisateur s'y trouve.

Le facteur d'amortissement $d \in [0, 1]$ modélise la probabilité d'un utilisateur à suivre les liens (avec probabilité d) plutôt que de se téléporter aléatoirement sur une autre page, parfois sans lien (avec probabilité $1 - d$). La formule pour un nœud v est :

$$PR(v) = \frac{1-d}{N} + d \sum_{u \in \text{In}(v)} \frac{PR(u)}{\deg^+(u)}$$

où N est le nombre de nœuds, $\text{In}(v)$ l'ensemble des nœuds pointant vers v , et $\deg^+(u)$ le degré sortant de u .

Les nœuds pendants (sans liens sortants) nécessitent un traitement particulier : leur PageRank est redistribué uniformément pour éviter qu'ils n'absorbent toute la probabilité. La formule devient :

$$PR(v) = \frac{1-d}{N} + \frac{d \cdot S}{N} + d \sum_{u \in \text{In}(v)} \frac{PR(u)}{\deg^+(u)}$$

où $S = \sum_{u \in \text{Dangling}} PR(u)$ est la somme des PageRanks des nœuds pendants.

2.1.2. Implémentation

Notre implémentation (`src/pagerank.ts`) utilise la méthode de power iteration. Le prétraitement (fonction `preprocessGraph`) construit les listes d'arêtes entrantes, calcule les degrés sortants et identifie les nœuds pendants en complexité $O(N + M)$, où N est le nombre de nœuds et M le nombre d'arêtes du graphe.

L'algorithme principal initialise tous les nœuds à $PR^{(0)}(v) = \frac{1}{N}$ puis itère jusqu'à convergence :

code de l'iteration :

```
const danglingSum = danglingNodes.reduce((acc, node) => acc + ranks[node], 0);
const baseRank = (1 - damping) / N + (damping * danglingSum) / N;

const newRanks = Array.from({ length: N }, (_, node) => {
  let rank = baseRank;
  incomingEdges[node].forEach((fromNode) => {
    rank += (damping * ranks[fromNode]) / outDegrees[fromNode];
  });
  return rank;
});
```

La convergence est vérifiée par la distance L1 entre deux itérations : $\sum_v |PR^{(t+1)}(v) - PR^{(t)}(v)| < \epsilon$.

```
// Check for convergence
diff = newRanks.reduce(
  (acc, rank, index) => acc + Math.abs(rank - ranks[index]),
  0
);

ranks = newRanks;

if (diff < tolerance) {
  console.log(`Converged after ${iter + 1} iterations`);
  break;
}
```

```

    }
  }

```

La complexité par itération est $O(N + M)$: calcul de la somme des nœuds pendants en pire des cas $O(N)$, parcours des arêtes entrantes en $O(M)$, et calcul de la distance L1 en $O(N)$. La complexité totale est donc $O(k \cdot (N + M))$ où k est le nombre d'itérations, qui dépend logarithmiquement de la tolérance ϵ .

2.2. Personalized PageRank

2.2.1. Principe

Le Personalized PageRank (PPR) est une variante du PageRank classique qui permet de calculer l'importance des nœuds relativement à un ensemble de nœuds sources appelés seeds. Contrairement au PageRank où la téléportation se fait uniformément vers tous les nœuds, le PPR restreint la téléportation uniquement vers les seeds.

Cette modification permet d'obtenir un PageRank localisé autour des seeds : les nœuds proches des seeds obtiennent des scores élevés, tandis que les nœuds éloignés ont des scores faibles. Le PPR est particulièrement utile pour la détection de communautés locales, la recommandation personnalisée, ou l'analyse de proximité dans un graphe.

La formule du PPR pour un nœud v est :

$$PPR(v) = \begin{cases} \frac{1-d}{|S|} + \frac{d \cdot D}{|S|} + d \sum_{u \in \text{In}(v)} \frac{PPR(u)}{\deg^+(u)} & \text{si } v \in S \\ d \sum_{u \in \text{In}(v)} \frac{PPR(u)}{\deg^+(u)} & \text{sinon} \end{cases}$$

où S est l'ensemble des seeds, $|S|$ leur nombre, et $D = \sum_{u \in \text{Dangling}} PPR(u)$ la somme des scores des nœuds pendants. Seuls les seeds reçoivent la probabilité de téléportation et la redistribution des nœuds pendants.

2.2.2. Implémentation

Notre implémentation (`src/PPR.ts`) suit la même structure que PageRank avec des modifications pour gérer les seeds. Le prétraitement est identique (complexité $O(N + M)$) avec en plus la conversion et validation des seeds (lignes 71-80).

L'initialisation diffère du PageRank : seuls les seeds reçoivent un score initial non nul (lignes 95-98) :

```

let ranks = new Array(N).fill(0);
seedIndices.forEach((idx) => {
  ranks[idx] = seedWeight;
});

```

où `seedWeight = 1 / seedIndices.length` assure que la somme initiale vaut 1.

L'algorithme principal calcule à chaque itération (lignes 111-126) :

```

const newRanks = Array.from({ length: N }, (_, node) => {
  const isSeed = seedSet.has(node);
  const teleportation = isSeed ? teleportationWeight : 0;
  const danglingContribution = isSeed
    ? damping * danglingSum * seedWeight
    : 0;
  let rank = teleportation + danglingContribution;

  incomingEdges[node].forEach((fromNode) => {
    rank += (damping * ranks[fromNode]) / outDegrees[fromNode];
  });

  return rank;
});

```

La différence clé avec PageRank est que seuls les seeds (vérifiés par `seedSet.has(node)`) reçoivent la téléportation et la contribution des nœuds pendants. Les autres nœuds n’obtiennent de score que par propagation depuis leurs voisins entrants.

Le critère de convergence reste la distance L1 : $\sum_v |PPR^{(t+1)}(v) - PPR^{(t)}(v)| < \epsilon$.

La complexité par itération est identique à PageRank : $O(N + M)$. Le test d’appartenance aux seeds (`seedSet.has(node)`) est en $O(1)$ grâce à l’utilisation d’un `Set`. La complexité totale reste donc $O(k \cdot (N + M))$ où k est le nombre d’itérations. En pratique, PPR converge souvent plus rapidement que PageRank car la distribution de probabilité est plus localisée autour des seeds.

2.3. Méthode PUSH

2.3.1. Principe

La méthode PUSH est un algorithme d’approximation locale du Personalized PageRank. Contrairement au PPR qui itère sur tous les nœuds du graphe jusqu’à convergence, PUSH ne traite que les nœuds ayant un résidu significatif, ce qui le rend particulièrement efficace pour les grands graphes.

L’algorithme maintient deux vecteurs : le vecteur de rang **rank** (approximation du PPR) et le vecteur de résidu **residual** (probabilité non encore distribuée). À chaque opération de push sur un nœud v , une partie du résidu est ajoutée au rang, et le reste est propagé aux voisins sortants. Le processus continue tant qu’il existe des nœuds avec un résidu supérieur au seuil ϵ .

Le paramètre ϵ contrôle le compromis précision/performance : une valeur faible (ex : 10^{-6}) donne une approximation très précise mais nécessite plus d’opérations, tandis qu’une valeur élevée (ex : 10^{-3}) est plus rapide mais moins précise. En pratique, $\epsilon = 10^{-4}$ offre un bon équilibre.

L’opération de push sur un nœud v avec résidu r_v consiste à : 1. Ajouter $(1 - d) \cdot r_v$ au rang de v . 2. Distribuer $\frac{d \cdot r_v}{\deg^+(v)}$ à chaque voisin sortant 3. Mettre le résidu de v à zéro

Pour les nœuds pendants, le résidu est redistribué uniquement vers les seeds, comme dans le PPR.

2.3.2. Implémentation

Notre implémentation (`src/PUSH.ts`) utilise une file pour gérer les nœuds à traiter. Le prétraitement construit les listes d’arêtes sortantes (contrairement à PageRank et PPR qui utilisent les arêtes entrantes), en complexité $O(N + M)$.

L’initialisation (lignes 84-95) place tout le résidu sur les seeds avec `residual[idx] = seedWeight`, tandis que le vecteur **rank** est initialisé à zéro. Les seeds sont ajoutés à une file de traitement.

La boucle principale (lignes 101-150) extrait un nœud de la file et, si son résidu dépasse ϵ , effectue l’opération de push :

```
rank[node] += (1 - damping) * res;

const pushValue = (damping * res) / degree;
neighbors.forEach((neighbor) => {
  residual[neighbor] += pushValue;
  if (!inQueue.has(neighbor) && residual[neighbor] >= epsilon) {
    queue.push(neighbor);
  }
});
```

Pour les nœuds pendants (lignes 135-146), le résidu est redistribué vers les seeds uniquement, proportionnellement à `seedWeight`.

L’algorithme s’arrête lorsque tous les résidus sont inférieurs à ϵ . Un nœud n’est ajouté à la file que si son résidu dépasse ϵ , évitant ainsi de traiter les nœuds peu pertinents.

La complexité dépend du nombre d’opérations de push effectuées. Dans le pire cas (tous les nœuds traités), elle est $O(M/\epsilon)$. En pratique, pour un PPR localisé, seule une fraction du graphe est explorée, donnant une complexité sous-linéaire en N et M . Le nombre d’opérations de push est typiquement $O(1/\epsilon)$ pour des graphes avec structure de communauté, rendant PUSH beaucoup plus rapide que PPR sur les grands graphes.

3. Résultats et analyse

Nous rappelons les objectifs fixés dans l'introduction : étudier la convergence du PageRank et du PPR en fonction de la précision, détecter les communautés locales via PPR, et comparer la méthode PUSH au PPR en termes de précision et de temps de calcul.

3.1. Méthodologie expérimentale

Les expérimentations ont été automatisées via le script `study.sh` qui orchestre l'ensemble des analyses. Ce script génère d'abord un graphe aléatoire d'Erdős–Rényi de 50 nœuds avec une probabilité de connexion de 0.08 (commande `./cli generate --nodes 50 -p 0.08`), puis exécute systématiquement :

1. L'étude de convergence du PageRank pour 8 niveaux de tolérance (10^{-2} à 10^{-9})
2. L'étude de convergence du PPR avec un seed unique (nœud 0) et multiples seeds (nœuds 0, 10, 20)
3. La détection de communautés avec 6 seuils différents (0.001 à 0.1) et 4 seeds différents
4. La comparaison PUSH vs PPR pour 8 valeurs d' ϵ (10^{-2} à 10^{-6})
5. L'étude de scalabilité sur 6 tailles de graphes (10 à 500 nœuds)

Tous les résultats sont sauvegardés au format JSON dans le répertoire `results/`. Le script `visualize.py` génère ensuite automatiquement les graphiques d'analyse en utilisant Matplotlib (si plusieurs exécutions ont été effectuées, les moyennes sont calculées). Cette approche garantit la reproductibilité complète des expérimentations : l'exécution de `./study.sh` suivie de `python visualize.py results/` régénère l'intégralité des données et visualisations.

Par défaut, le script `study.sh` exécute l'ensemble des analyses 10 fois pour chaque expérience, mais pour le rapport, nous avons exécuté l'analyse de scalabilité 100 fois afin d'améliorer la fiabilité des résultats.

3.2. Performance des algorithmes

3.2.1. PageRank

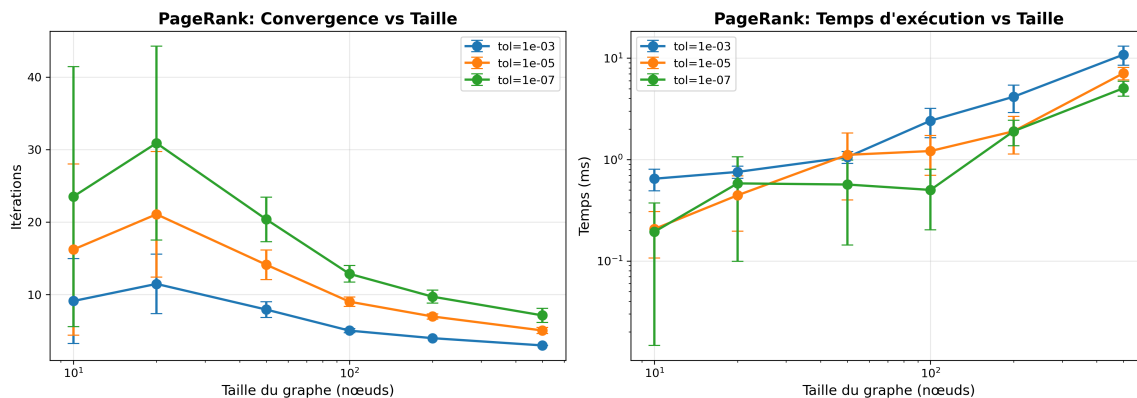


FIGURE 1 – Scalabilité PageRank

La figure 1 présente deux graphiques : le nombre d'itérations (gauche) en échelle semi-log et le temps d'exécution (droite) en échelle log-log en fonction de la taille du graphe.

En ce qui concerne le nombre d'itérations (graphique de gauche), on observe une forte disparité pour les petits graphes. Cependant, à mesure que la taille du graphe augmente, le nombre d'itérations tend à se stabiliser. Ce comportement s'explique principalement par le mode de génération aléatoire des graphes : sur de petits graphes, il est plus probable d'obtenir des structures particulières qui rendent la convergence plus ou moins rapide. En revanche, pour des graphes de taille moyenne à grande, ces effets aléatoires s'atténuent et le nombre d'itérations devient plus régulier.

En moyenne, hormis les deux premiers points, on constate une légère diminution du nombre d'itérations avec la taille du graphe. Cela peut s'expliquer par le fait que, dans un graphe plus grand,

le score associé à chaque nœud est plus faible, ce qui fait que la différence entre deux itérations successives devient plus rapidement inférieure au seuil de tolérance fixé.

Concernant le temps d'exécution (graphique de droite), celui-ci présente une croissance quasi linéaire en échelle log-log, mais avec une pente inférieure à 1 (comprise entre 0,5 et 0,75). Cette pente plus faible traduit une croissance sous-linéaire du temps d'exécution, cohérente avec la diminution du nombre d'itérations observée précédemment : plus le graphe est grand, moins il faut d'itérations pour atteindre la convergence, ce qui réduit le coût global.

3.2.2. Personalized PageRank

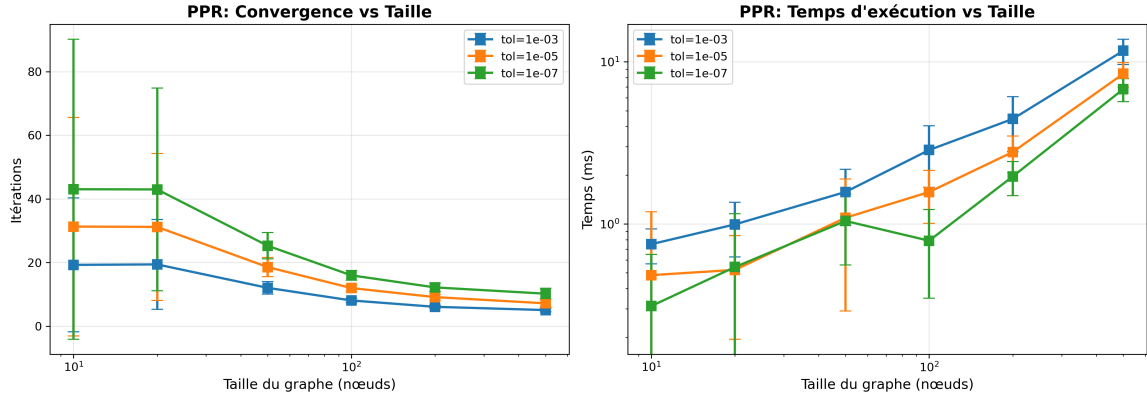


FIGURE 2 – Scalabilité PPR

Tout comme pour PageRank, la figure 2 présente deux graphiques pour l'itération et le temps d'exécution en fonction de la taille du graphe.

En termes de nombre d'itérations (graphique de gauche), tout comme pour PageRank, on observe une forte disparité pour les petits graphes. Cependant, les moyennes sont relativement stables avec une légère diminution. Cela se traduit par une pente très proche de 1 pour le temps d'exécution (graphique de droite), ce qui confirme que le temps d'exécution de PPR est $O(k \cdot (N + M))$ comme attendu.

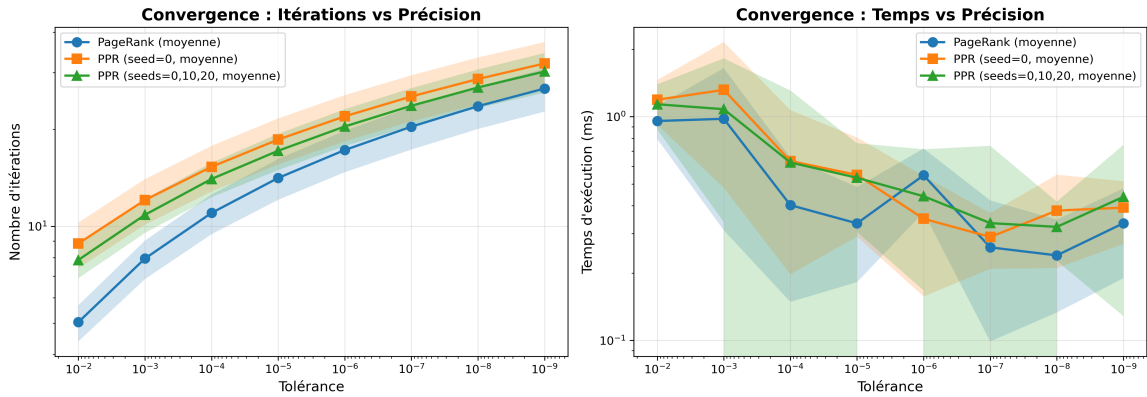


FIGURE 3 – Convergence PR vs PPR

La figure 3 compare les algorithmes PageRank et PPR en termes de convergence sur le graphe de 50 nœuds en fonction de la tolérance (échelle log-log).

Pour le nombre d'itérations, nous remarquons que les courbes des 3 algorithmes ont la même forme (augmentent légèrement), mais que PPR nécessite légèrement plus d'itérations pour converger. En effet, les seeds commencent et reçoivent des probabilités plus élevées (notamment la téléportation), ce qui crée des gradients de probabilité plus prononcés qui nécessitent plus d'itérations pour se stabiliser uniformément sur tout le graphe.

Pour ce qui est du temps d'exécution, nous remarquons qu'il a tendance à diminuer avec la tolérance, ce qui est contre-intuitif. Les exécutions à elles seules semblent toujours montrer une croissance du

temps d'exécution, mais nous ne savons pas pourquoi les moyennes ont cette tendance. Peut-être des optimisations du runtime JavaScript ?

3.2.3. PUSH

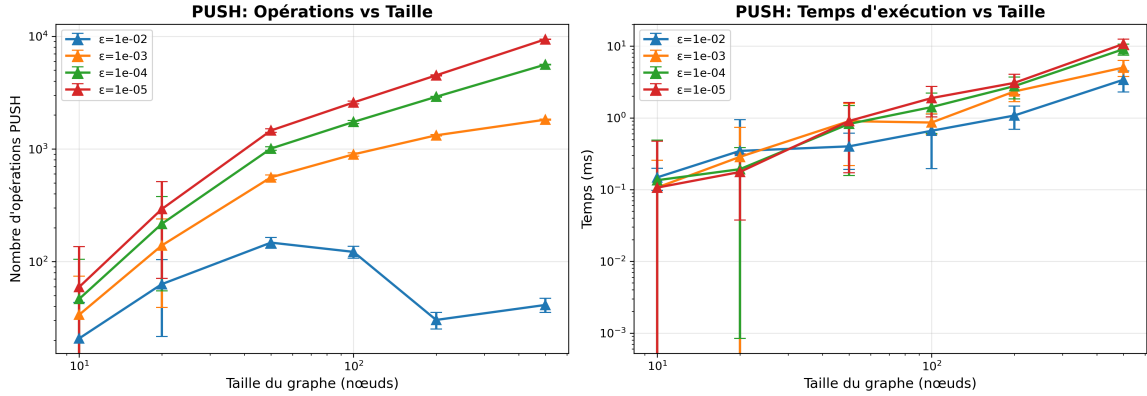


FIGURE 4 – Scalabilité PUSH

La figure 4 compare les performances de PUSH en fonction de la taille du graphe.

Nous remarquons dans le graphique de droite que le temps d'exécution de PUSH subit une croissance sous-linéaire, ce qui est assez étonnant puisque la complexité théorique de PUSH est $O(1/\epsilon)$. Cette tendance peut être expliquée par le fait que plus le graphe est grand, plus il y a de nœuds à traiter, et donc l'exécution de PUSH devient moins locale. La complexité théorique reste valable en ordre de grandeur, mais les coûts constants cachés dans la notation O deviennent plus importants. Cela est confirmé par le graphique de gauche qui montre le nombre d'opérations de PUSH. On observe une croissance sous-linéaire du nombre d'opérations avec la taille du graphe due à un plus grand nombre de nœuds à traiter.

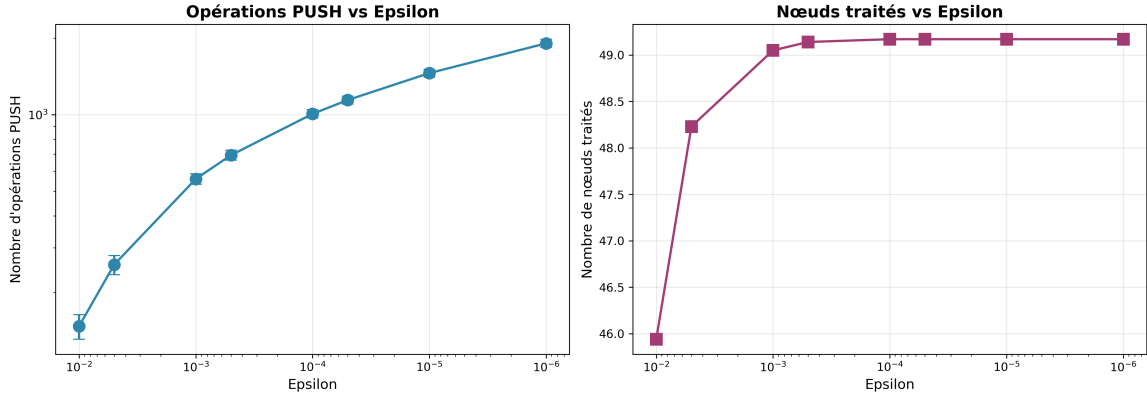


FIGURE 5 – Opérations PUSH

La figure 5 montre la relation entre ϵ et le nombre d'opérations/nœuds traités sur le graphe de 50 nœuds.

Dans le graphique de gauche, nous pouvons observer que le nombre d'opérations de PUSH croît avec la diminution de ϵ . Sauf qu'en théorie, nous sommes censés avoir une droite linéaire, mais nous observons une courbe légèrement concave. La courbe croît plus lentement que la théorie $O(1/\epsilon)$. Cela pourrait probablement être expliqué par la saturation de la propagation locale. En effet, quand ϵ est très petit, le résidu se propage à presque tout le graphe. Cela est confirmé par le graphique de droite qui montre le nombre de nœuds traités. On observe une croissance qui atteint assez vite un plateau de 50 nœuds traités.

3.2.4. Comparaison PUSH vs PPR

La figure 6 présente 3 graphiques comparant PUSH et PPR sur le graphe de 50 nœuds.

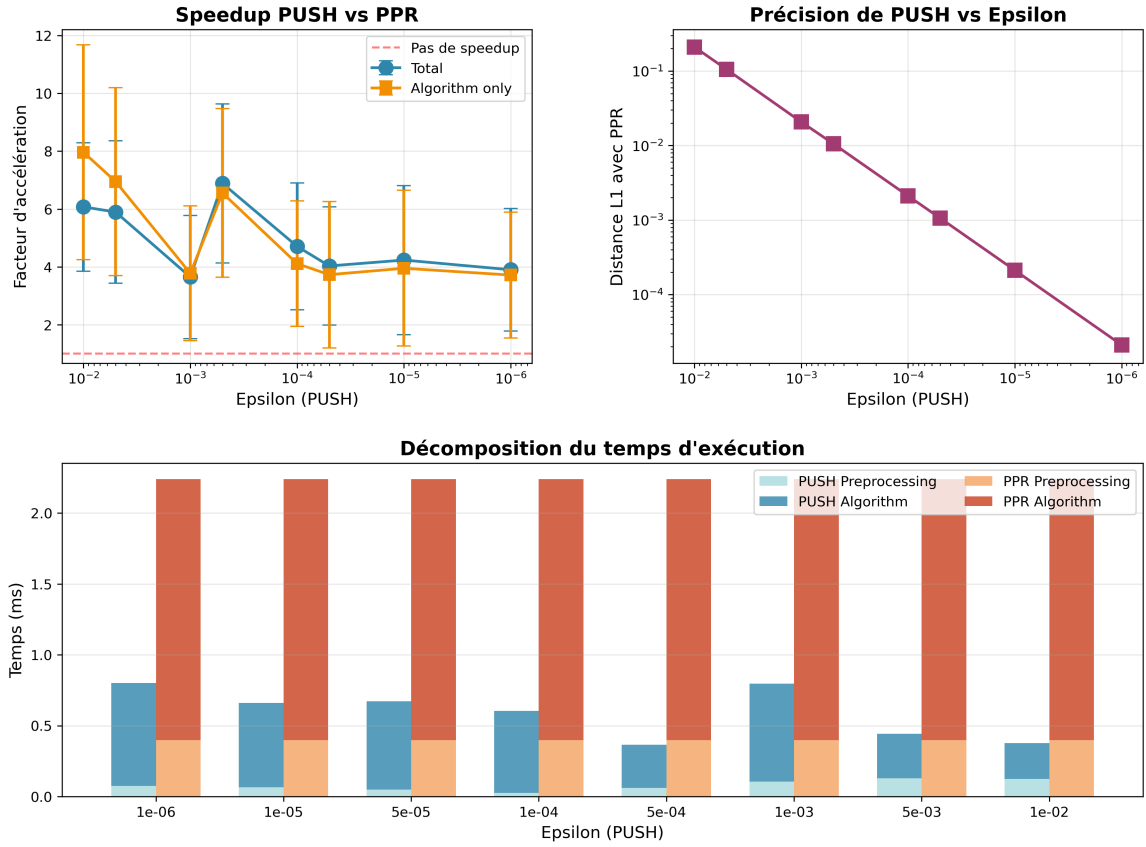


FIGURE 6 – Comparaison PUSH vs PPR

D'après le graphique de gauche, nous pouvons affirmer dans un premier temps que PUSH est plus rapide que PPR pour tous les ϵ testés. À noter que pendant nos tests, il est arrivé dans certains cas, notamment pour des graphes de grande taille avec un ϵ petit, que PUSH soit plus lent que PPR.

À partir du graphique de droite, nous pouvons observer que la distance L1 entre PUSH et PPR diminue avec la diminution de ϵ de façon linéaire. Ce comportement est conforme à la théorie : le seuil ϵ borne directement le résidu maximal autorisé à chaque nœud, entraînant une erreur globale proportionnelle à ϵ .

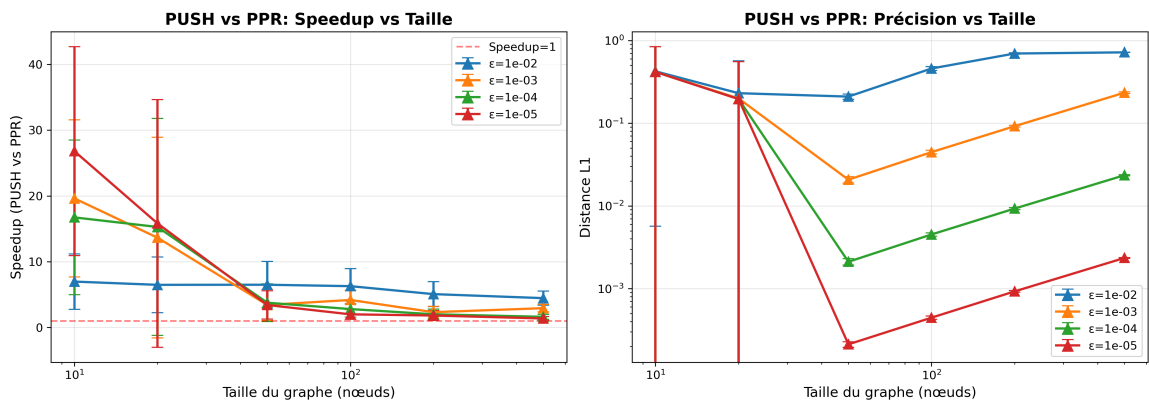


FIGURE 7 – Scalabilité PUSH vs PPR

La figure 7 montre l'évolution du speedup et de la précision en fonction de la taille du graphe.

Dans le graphique de gauche (hormis les deux premiers points qui sont assez sensibles à la structure du graphe), nous remarquons une linéarité du speedup sur un graphique log-log avec la taille du graphe. Cela signifie que le speedup est proportionnel à la croissance de PPR et donc à la taille du graphe, ce qui est contre-intuitif. Peut-être que les choix de ϵ sont trop grands pour de petits graphes

(de taille 50) ?

3.3. Détection de communautés

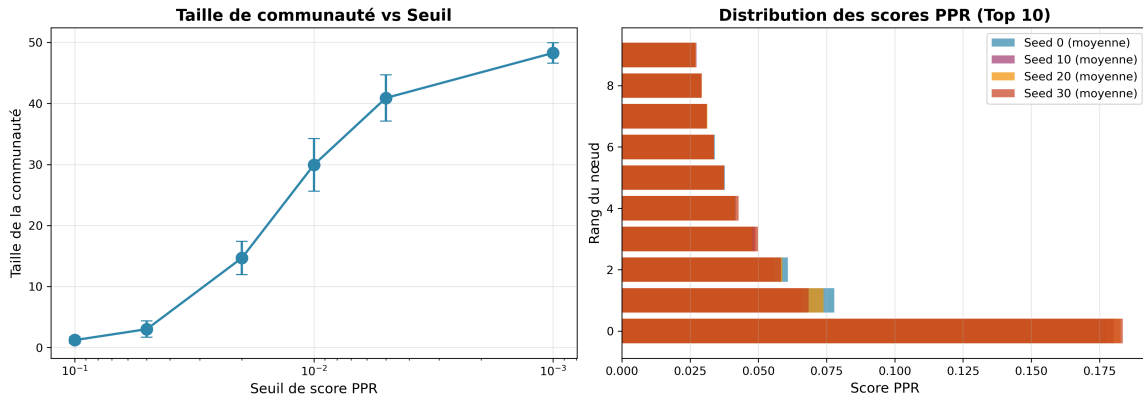


FIGURE 8 – Analyse des communautés

La figure 8 présente deux graphiques analysant la détection de communautés via PPR sur le graphe de 50 nœuds : la taille des communautés en fonction du seuil (gauche) et la distribution des scores au sein d’une communauté (droite).

Le graphique de gauche montre une augmentation de la taille de la communauté détectée à mesure que le seuil diminue. Ce comportement est attendu : un seuil plus faible permet d’inclure davantage de nœuds ayant des scores PPR plus faibles, élargissant ainsi la communauté autour du seed.

Le graphique de droite illustre la distribution des scores PPR au sein d’une communauté détectée, en fonction du rang des nœuds (triés par score décroissant). On observe une décroissance exponentielle très marquée : les premiers nœuds (rang faible) possèdent des scores significativement plus élevés que les suivants. Cette distribution confirme que PPR identifie efficacement un noyau central de nœuds fortement connectés au seed (scores élevés), entouré de nœuds périphériques (scores faibles). La forme exponentielle de cette courbe est caractéristique de la structure des communautés locales : quelques nœuds centraux dominent la probabilité de visite, tandis que la majorité des nœuds de la communauté ont des scores beaucoup plus faibles.

4. Conclusion

Ce projet nous a permis d’implémenter et d’analyser trois algorithmes fondamentaux pour le calcul de centralité dans les graphes : PageRank, Personalized PageRank (PPR) et la méthode d’approximation PUSH. Les résultats expérimentaux confirment globalement les prédictions théoriques tout en révélant certains comportements intéressants.

4.1. Synthèse des résultats

Nos expérimentations ont démontré que PageRank et PPR présentent des comportements de convergence similaires, avec une complexité par itération en $O(N + M)$ comme attendu. PPR nécessite généralement plus d’itérations que PageRank en raison des gradients de probabilité plus prononcés créés par la concentration de la téléportation sur les seeds.

La méthode PUSH s’est révélée systématiquement plus rapide que PPR dans la majorité des cas testés. La distance L1 entre PUSH et PPR décroît linéairement avec ϵ , conformément à la théorie. L’analyse de détection de communautés via PPR a montré des résultats cohérents avec une distribution des scores suivant une décroissance exponentielle caractéristique.

4.2. Limites et observations

Certains résultats ont révélé des comportements inattendus. Nous avons observé une tendance contre-intuitive du temps d’exécution moyen à diminuer avec la tolérance pour PageRank et PPR, dont

l'origine reste incertaine. Par ailleurs, le speedup de PUSH par rapport à PPR stagne malgré l'augmentation de la taille du graphe, ce qui est contre-intuitif. Une hypothèse est que nos graphes de taille 50 sont peut-être trop petits pour observer le comportement local attendu.

4.3. Pistes d'amélioration

- Nos expérimentations se sont concentrées sur des graphes de 50 nœuds. Cette taille relativement modeste a pu influencer certains résultats, notamment les comportements observés avec PUSH. Des graphes plus grands permettraient d'observer plus clairement les comportements asymptotiques théoriques.
- Il serait pertinent de tester les algorithmes sur différents types de graphes : graphes scale-free, graphes avec structure de communautés prononcée, graphes réguliers, ou encore des graphes réels issus de réseaux sociaux ou de citations scientifiques. Ces différentes topologies pourraient révéler des comportements spécifiques et permettre de mieux comprendre dans quels contextes chaque algorithme excelle.
- Le choix de ϵ pour PUSH mériterait une étude plus systématique en fonction de la taille et de la structure du graphe. Il serait intéressant d'établir des heuristiques pour sélectionner automatiquement un ϵ optimal offrant le meilleur compromis précision/performance selon les caractéristiques du graphe.