Efficient Top-k Closeness Centrality Search

Écrit par :

Benaissa Meriem - 21418006
Sadi Aksil - 21427293
Sadi Massin - 21416091

Encadrant :

Mehdi Naima
Lionel Tabourier

Promotion 2025 - 2026

Table des matières

1 Introduction

La centralité de proximité (*closeness centrality*) évalue l'importance d'un nœud selon sa distance moyenne aux autres, utile pour analyser la structure de réseaux tels que ceux du transport ou des communications. Ce projet met en œuvre trois approches : une méthode classique basée sur le BFS, une version optimisée inspirée de [1] pour identifier efficacement les k nœuds les plus centraux, et une version temporelle issue de [3] intégrant la dimension temporelle des arêtes. L'objectif est de détecter rapidement les nœuds les plus centraux sur des graphes réels extraits via **OSMnx** et de comparer les performances des approches en termes de précision et de temps d'exécution.

Le code source complet, les scripts expérimentaux et les visualisations sont disponibles sur le dépôt GitHub suivant : `github.com/MassinS/Projet_AAGA`

2 Méthode classique

2.1 Principe général

La méthode classique calcule la centralité de proximité d'un nœud v comme l'inverse de la somme des distances le séparant des autres nœuds atteignables. Pour un graphe quelconque $G = (V, E)$, la centralité normalisée s'écrit :

$$C_C(v) = \frac{(|R_v| - 1)^2}{(|V| - 1) \times \sum_{u \in R_v \setminus \{v\}} dist(v, u)}$$

où R_v désigne l'ensemble des sommets atteignables depuis v. Cette formulation est valable pour tout type de graphe (orienté, non orienté, pondéré ou non). Dans le cas particulier d'un graphe non orienté et non pondéré, si le graphe est connexe, on a $|R_v| = |V|$, et la formule se réduit alors à la forme classique :

$$C_C(v) = \frac{n - 1}{\sum_{u \neq v} dist(v, u)}$$

ce qui correspond à la centralité de proximité usuelle.

2.2 Principe de calcul

Le calcul repose sur un **parcours en largeur (BFS)** depuis chaque nœud du graphe :

1. Pour chaque nœud $v \in V$, on initialise toutes les distances à l'infini, sauf $dist(v, v) = 0$.
2. On parcourt ensuite le graphe à l'aide d'un BFS pour calculer $dist(v, u)$ pour tous les nœuds atteignables u.
3. On cumule ces distances pour obtenir $\sum_u dist(v, u)$.
4. On applique la formule précédente pour calculer $C_C(v)$.

Cette procédure est répétée pour chaque nœud du graphe afin d'obtenir la centralité complète. On peut ensuite trier les valeurs de $C_C(v)$ pour déterminer les nœuds les plus centraux (Top-k).

2.3 Complexité

Chaque parcours en largeur (BFS) a un coût $O(|V| + |E|)$, où $|V|$ est le nombre de sommets et $|E|$ le nombre d'arêtes du graphe. Comme ce parcours est exécuté pour chaque nœud $v \in V$, la complexité totale est :

$$O(|V| \times (|V| + |E|))$$

Cette approche fournit des valeurs exactes mais devient rapidement coûteuse pour les graphes de grande taille.

3 Méthode optimisée

3.1 Principe général

L'algorithme **Efficient Top-k Closeness Centrality** [1] a été conçu pour rendre le calcul de la centralité de proximité beaucoup plus efficace sur de grands graphes. L'idée principale est d'éviter de relancer un parcours complet (*BFS*) depuis chaque nœud, ce qui coûte $O(|V| \times (|V| + |E|))$.

Au lieu de cela, l'algorithme exploite deux mécanismes complémentaires :

— une **planification adaptative des parcours** (*Scheduling*) qui choisit dans quel ordre les nœuds doivent être explorés ;

— une **réutilisation partielle des calculs** via des parcours différentiels (**Δ-PFS**), qui permettent de propager les résultats entre sommets voisins.

Ainsi, seuls les k nœuds présentant les valeurs de centralité les plus élevées sont calculés en détail. Une structure **Top-k** (implémentée par un tas min) conserve dynamiquement les meilleurs résultats. Des techniques d'**élagage** (*pruning*) permettent d'interrompre tôt les calculs dont la contribution potentielle est inférieure au seuil minimal actuel θ_A du top-k.

3.2 Étapes principales de l'algorithme

L'approche est composée de trois grandes phases :

— **Phase PREP :** Pour chaque nœud v, cette phase estime deux quantités : le nombre de sommets atteignables $V_{\hat{v}}$ et la somme des distances $S_{\hat{v}}$, à l'aide de **Sketches probabilistes** dérivés du **Flajolet–Martin Sketch (FM-Sketch)** [4], base de la méthode *HyperLogLog*. Chaque Sketch associe à un nœud une signature binaire issue du hachage de son identifiant ; le nombre de zéros consécutifs dans cette valeur indique la

taille approximative de son ensemble atteignable. Pour réduire la variance, chaque Sketch contient **64 registres indépendants** dont les valeurs sont fusionnées par une opération maximale registre par registre lors de la propagation dans le graphe. Ainsi, chaque nœud reçoit progressivement un résumé de la portée de ses voisins, ce qui permet d'obtenir une estimation rapide et robuste de la visibilité de chaque sommet **sans parcourir explicitement tout le graphe**.

— **Phase SCHEDULE :** À partir des estimations issues de la phase PREP, cette étape détermine un **ordre d'exploration optimal** des sommets. Les nœuds dont la centralité estimée est élevée sont sélectionnés comme **sources** et explorés entièrement via un parcours complet (**PFS**). Les autres sont reliés à un parent déjà traité et analysés à l'aide d'une version optimisée (**Δ-PFS**) qui réutilise partiellement les distances déjà calculées. L'idée est d'éviter de répéter des parcours similaires : si deux sommets sont proches dans le graphe, leurs distances étant fortement corrélées, il est plus efficace de déduire les résultats du second à partir du premier. Cette étape construit ainsi une **structure hiérarchique** S reliant chaque nœud à son parent direct, limitant la redondance des calculs et guidant la propagation des informations dans la phase suivante.

— **Phase PROCESS :** Les nœuds sources sont explorés individuellement : pour chacun, la centralité est calculée puis le **Top-k** est mis à jour. Les résultats sont ensuite propagés récursivement vers les sommets dépendants définis dans la structure S. Ces derniers sont traités par un parcours différentiel (**Δ-PFS**) qui réutilise les distances déjà calculées pour leur parent, limitant ainsi les parcours redondants. Un mécanisme de **rollback** restaure l'état initial après chaque propagation pour isoler les calculs entre branches. Enfin, l'**élagage adaptatif (pruning)** interrompt les explorations dont la centralité potentielle est inférieure au seuil minimal θ_A, afin de concentrer le calcul sur les nœuds les plus prometteurs.

3.3 Pseudo-code simplifié

Algorithme 1 : Top-k Closeness Centrality — version optimisée

Entrée : Graphe $G = (V, E)$, entier k
Sortie : Top-k sommets les plus centraux
$(V, S) \leftarrow$ `prep`(G) `// Estimations via sketches`
$S \leftarrow$ `schedule`(G, V, S) `// Planification adaptative`
$A \leftarrow \emptyset$, dead $\leftarrow \emptyset$
pour chaque $v \in \mathit{Start}(S)$ **faire**
  $(L, s, \delta_p) \leftarrow$ `PFS`(G, v) `// Parcours complet depuis la source`
  `process`$(G, v, L, s, A, S, k, |V|, \delta_p, \text{dead})$
fin
retourner A `// Top-`k `nœuds les plus centraux`

Sous-fonctions principales :

— `PFS(G,v)` : effectue un parcours complet (BFS) depuis v et calcule sa centralité exacte.
— `optimized_PFS(G,v,p)` : calcule la centralité de v à partir des distances déjà obtenues pour son parent p.
— `update_topk(A,p,c_p,k)` : met à jour le top-k en remplaçant le plus petit élément si c_p est supérieur.
— `prune(v,L,s,`θ_A`,S,`δ_v`,G)` : supprime les sous-branches dont la centralité maximale estimée $< \theta_A$ (élagage adaptatif).

3.4 Complexité

La complexité de chaque composant de l'algorithme est résumée ci-dessous :

— **PFS** : $O(|E_v| + |V_v| \log |V_v|)$, ou $O(|E_v| + |V_v|)$ pour un graphe non pondéré.
— **Δ-PFS** : $O(|E_{vp}| + |V_{vp}| \log |V_{vp}|)$, avec $|V_{vp}| \ll |V_v|$ en pratique.
— **Pruning** : $O(|E| \log |V|)$ pour la construction des bornes et un coût marginal à l'exécution.
— **Schedule** : $O(|E| \log |V|)$ pour la génération de l'ordre optimal d'exploration.
— **Prep** : $O(\psi \frac{\delta}{\mu} |E|)$, avec $\psi = 64$ registres par *FM-sketch*, δ le diamètre du graphe et μ le poids minimal d'une arête.

La complexité globale s'obtient en combinant les contributions de chaque phase. Les étapes **PREP** et **SCHEDULE** ont un coût linéaire en $|E|$ et ne sont exécutées qu'une fois, tandis que la phase **PROCESS** domine le coût total. Cette dernière exécute au plus k parcours complets (**PFS**), les autres sommets étant traités par des parcours différentiels (**Δ-PFS**) beaucoup plus rapides. Ainsi, le temps total moyen est proportionnel à k parcours :

$$O(k \cdot (|V| + |E|)).$$

Dans le pire cas, si aucun partage ni élagage n'est possible, chaque nœud doit être exploré entièrement, et la complexité rejoint celle de la méthode classique :

$$O(|V| \cdot (|V| + |E|)).$$

4 Méthode temporelle

4.1 Principe général

L'algorithme de **Top-k Temporal Closeness** [3] étend la centralité classique aux graphes où les arêtes possèdent un temps d'apparition t et une durée λ. Il mesure à quelle vitesse un sommet peut atteindre les autres dans un intervalle temporel donné $I = [\alpha, \beta]$. Deux étapes principales sont nécessaires :

— le calcul des **chemins temporels les plus rapides** (Algorithme 1) via une approche à labels (*label-setting*) ;

— l'évaluation de la **centralité temporelle** pour chaque sommet (Algorithme 2) afin d'extraire le Top-k.

4.2 Représentation du graphe temporel

Un graphe temporel $G = (V, E_T)$ est composé d'arêtes temporelles $e = (u, v, t, \lambda)$ actives à l'instant t pour une durée λ. Une arête n'est empruntable que si elle respecte la causalité : le voyageur ne peut partir qu'après son apparition ($a \leq t$).

$$E_T = \{(u, v, t, \lambda) \mid u, v \in V,\ t \in \mathbb{R}^+,\ \lambda > 0\}$$

4.3 Algorithme 1 — Chemins temporels les plus rapides

Algorithme 2 : Incremental Fastest Paths — `incremental_fastest_paths(G, source, I)`

Entrée : Graphe temporel $G = (V, E_T)$, source s, intervalle $I = [\alpha, \beta]$
Sortie : Flux progressif $(v, d[v], d_{next})$ pour chaque $v \in V$ atteint
pour chaque $v \in V$ **faire**
  $d[v] \leftarrow \infty$, $\Pi[v] \leftarrow []$
fin
$d[s] \leftarrow 0$; $Q \leftarrow [(0, s, 0)]$; $F \leftarrow \emptyset$
tant que $Q \neq \emptyset$ *et* $|F| < |V|$ **faire**
  extraire le label (v, s, a) de durée minimale
  si $v \in F$ **alors**
    continuer
  fin
  $d[v] \leftarrow a - s$; $F \leftarrow F \cup \{v\}$
  $d_{next} \leftarrow$ durée minimale restante dans Q (ou $d[v]$ si vide)
  émettre $(v, d[v], d_{next})$
  pour chaque $(v \to u, t, \lambda) \in E_T$ *actifs dans* I *et* $a \leq t$ **faire**
    $s' \leftarrow s$ si $s \neq 0$ sinon t ; $a' \leftarrow t + \lambda$
    créer label (u, s', a') ; **si** non dominé, l'ajouter à $\Pi[u]$
    supprimer labels dominés dans $\Pi[u]$; insérer $(a' - s', u)$ dans Q
  fin
fin

Version incrémentale de l'algorithme 1 du papier d'Oettershagen et Mutzel (2020), adaptée pour le calcul de la centralité temporelle. L'algorithme explore les arêtes actives dans le temps et conserve uniquement les labels non dominés, garantissant la durée minimale d'arrivée pour chaque sommet. La version originale complète est également implémentée dans le code.

4.4 Algorithme 2 — Top-k Temporal Closeness

Algorithme 3 : Top-k Temporal Closeness avec Pruning (version compacte)

Entrée : Graphe temporel G, entier k, intervalle I
Sortie : Top-k nœuds les plus centraux `topk`
$topk \leftarrow [] \,;\, B_k \leftarrow 0 \,;\, \delta \leftarrow 0 \,;\, \lambda_{\min} \leftarrow \min(\text{durées des arêtes})$
pour chaque *u dans G.V triés par degré sortant décroissant* **faire**
  $S_F \leftarrow 0 \,;\, F \leftarrow \emptyset \,;\, T \leftarrow \emptyset \,;\, N \leftarrow |G.V|$
  pour chaque (v, d, d_{next}) *dans* `incremental_fastest_paths`(G, u, I) **faire**
    si $d = 0$ **alors**
      continue
    fin
    $; F \leftarrow F \cup \{v\} \,;\, S_F \leftarrow S_F + 1/d$
    $T \leftarrow \{w \notin F \mid (v, w) \in E_T\}$
    $c_{\hat{u}} \leftarrow S_F + \frac{|T|}{d_{next}} + \frac{N-|F|-|T|}{d_{next}+\delta+\lambda_{\min}}$
    si $c_{\hat{u}} < B_k$ **alors**
      break
    fin
  fin
  insérer (S_F, u) dans $topk$; **si** $|topk| > k$ **alors**
    `heappop`(topk)
  fin
  $; B_k \leftarrow \min(topk)$
fin
retourner *topk trié décroissant*

Chaque sommet sert de source pour mesurer sa capacité à atteindre rapidement les autres. Une borne supérieure $c_{\hat{u}}$ permet un *pruning* anticipé : si un sommet ne peut dépasser le seuil actuel B_k, il est ignoré.

4.5 Complexité

L'algorithme `Incremental_fastest_paths` est analogue à celui de Dijkstra, avec une complexité $O(|E_T| \log |V|)$ par sommet source, où $|V|$ désigne le nombre de sommets du graphe, et $|E_T|$ le nombre d'arêtes temporelles (c'est-à-dire les transitions actives dans le temps).

La recherche complète du Top-k a donc une complexité moyenne :

$$O(k \cdot |E_T| \log |V|),$$

où k représente le nombre de sommets effectivement explorés jusqu'au bout. Cette approche est nettement plus efficace que la version classique, qui nécessiterait d'explorer le graphe depuis chaque sommet pour les graphes dynamiques.

5 Expérimentations et Résultats

5.1 Protocole expérimental

Les expérimentations ont été menées sur dix graphes routiers réels extraits via la bibliothèque **OSMnx**, représentant les villes de Paris, Lyon, Marseille, Toulouse, Bordeaux, Nice, Nantes, Dijon, Reims et Annecy. Chaque graphe a été testé en version **non orientée** (circulation bidirectionnelle) et **orientée** (prise en compte des sens uniques).

Pour chaque ville, nous avons exécuté :

— l'algorithme **classique** basé sur des parcours complets BFS ;

— l'algorithme **efficient Top-k** (Olsen et al., 2014) ;

— l'algorithme **Top-k temporel** (Oettershagen & Mutzel, 2020) sur uniquement les graphes orientés.

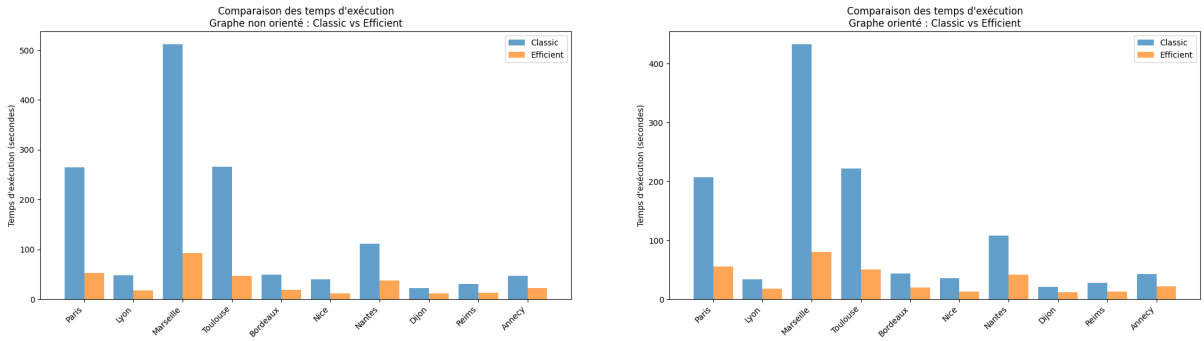
Les indicateurs suivants ont été calculés automatiquement dans le script Python :

1. le **temps d'exécution** de chaque méthode (*Classic*, *Efficient* et *Temporal*) ;
2. le **gain (%)** :
$$\text{Gain} = \frac{T_{classic} - T_{efficient}}{T_{classic} \times 100}$$
représentant la réduction relative du temps de calcul ;
3. le **speed-up (×)** :
$$\text{Speed-up} = \frac{T_{classic}}{T_{efficient}}$$
mesurant le facteur d'accélération absolu ;
4. le **taux d'overlap (%)** :
$$\text{Overlap} = \frac{|Top5_{classic} \cap Top5_{efficient}|}{5 \times 100}$$
indiquant la similarité entre les ensembles des cinq nœuds les plus centraux obtenus par les deux méthodes.

Les résultats ont été enregistrés dans `resume_no_oriented.csv` et `resume_oriented.csv`, puis visualisés à l'aide de graphiques comparatifs générés automatiquement.

5.2 Comparaison des temps d'exécution

Les figures suivantes montrent la comparaison des temps d'exécution pour les graphes non orientés et orientés. La méthode optimisée réduit le temps de calcul de manière significative — jusqu'à **70–85% de gain** pour certaines villes (ex. Marseille, Toulouse). Sur l'ensemble des tests, la moyenne du **speed-up** se situe entre **3× et 5×** selon la taille et la densité du graphe.



(a) Non orienté (b) Orienté

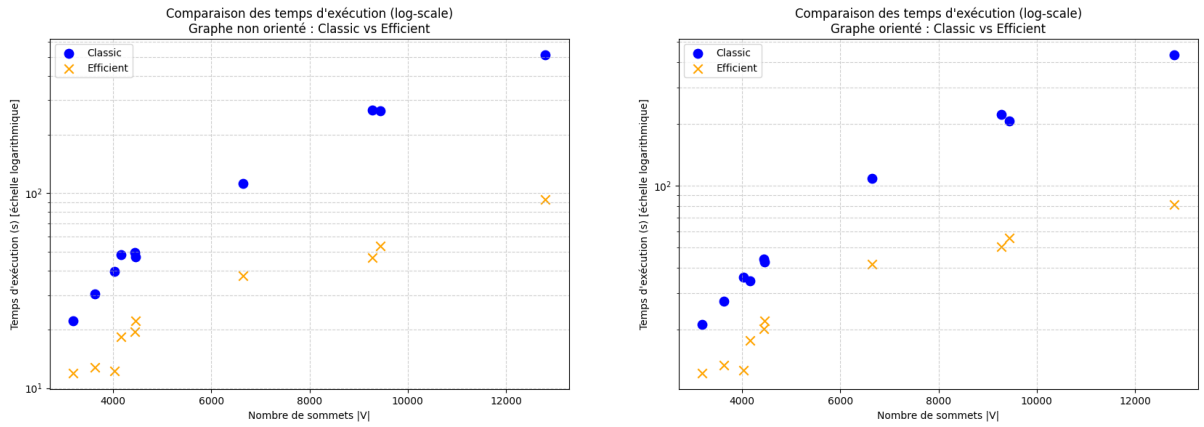
FIGURE 1.1 – Comparaison des temps d'exécution — Classic vs Efficient.

5.3 Analyse de la complexité empirique

La relation entre la taille du graphe et le temps d'exécution (Figures 1.2a et 1.2b) montre que :

— le temps d'exécution de la version classique augmente rapidement avec la taille du graphe ;

— la version optimisée présente une croissance quasi linéaire, grâce à la planification adaptative et à la réutilisation des résultats de parcours.

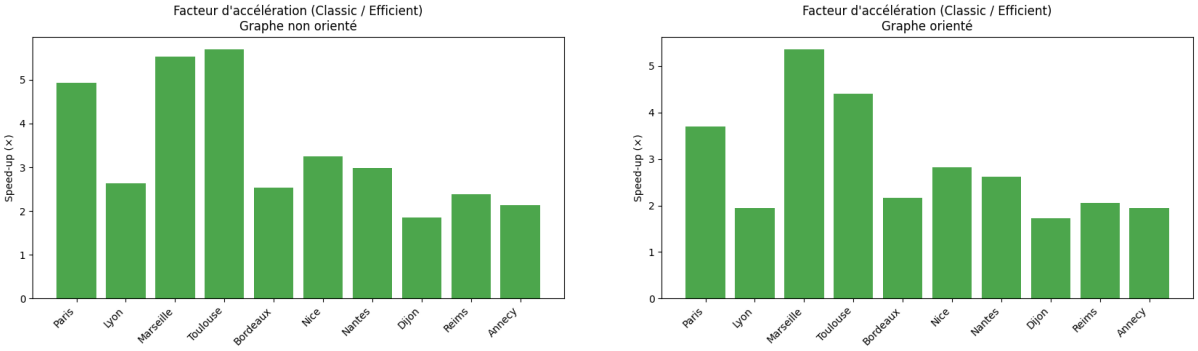


(a) Non orienté (b) Orienté

FIGURE 1.2 – Temps d'exécution selon la taille du graphe (échelle logarithmique).

5.4 Facteur d'accélération et recouvrement des Top-5

Les Figures 1.3a et 1.3b présentent le **speed-up** mesuré pour chaque ville. Les plus forts gains sont observés sur les graphes les plus denses (Marseille, Toulouse, Paris), atteignant jusqu'à **5.6×**.



(a) Non orienté (b) Orienté

FIGURE 1.3 – Facteur d'accélération (Classic / Efficient).

5.5 Évaluation sur graphes sociaux

Pour compléter l'analyse, les deux méthodes ont été appliquées sur un graphe de type *Wiki-Vote*, contenant plusieurs milliers de sommets et arêtes. La Figure 1.4 montre un **gain moyen de 72%** et un **speed-up de 3.5×**, confirmant la robustesse de la méthode sur des topologies non spatiales.

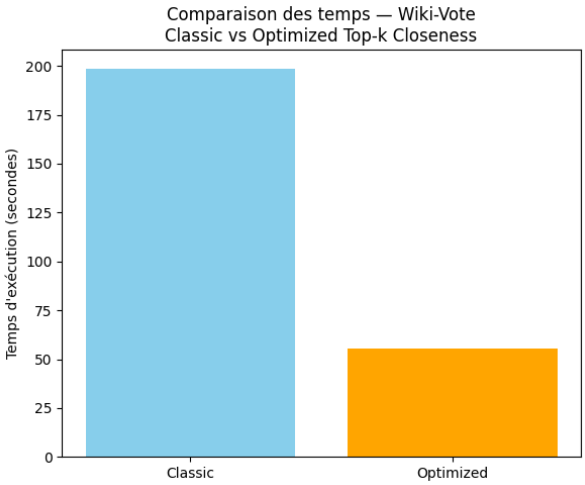


FIGURE 1.4 – Comparaison sur le graphe social Wiki-Vote.

5.6 Visualisation des Top-5 Closeness

Les visualisations (non incluses ici pour respecter la limite de pages elles se trouvent dans le dossier visualisation) montrent que les nœuds les plus centraux identifiés par la méthode optimisée coïncident presque toujours avec ceux de la méthode classique : carrefours, ronds-points et jonctions principales. Dans les graphes orientés, ces sommets

se déplacent légèrement vers les zones à fort degré sortant, correspondant aux axes majeurs de diffusion du trafic.

6 Perspectives

Ce travail pourrait être prolongé selon deux axes principaux.

Tout d'abord, il serait intéressant de **comparer les trois approches** (classique, optimisée et temporelle) dans un même cadre expérimental. Faute de temps, seule la comparaison entre les deux premières a été réalisée. Une piste future serait de transformer les graphes temporels en versions statiques équivalentes afin d'évaluer leurs performances respectives en termes de temps de calcul, de précision et de cohérence du Top-k.

Ensuite, la méthode temporelle pourrait être **optimisée et généralisée** : par exemple en parallélisant les calculs de plus courts chemins temporels ou en l'appliquant à des réseaux dynamiques de plus grande échelle (transport multimodal, réseaux sociaux). Cela permettrait d'adapter la centralité de proximité à des contextes encore plus réalistes et évolutifs.

7 Conclusion

Ce projet a permis de concevoir et d'évaluer plusieurs approches du calcul de la centralité de proximité, depuis la méthode classique fondée sur des parcours BFS jusqu'à la version optimisée proposée par Olsen et al. (2014), puis son extension temporelle inspirée de Oettershagen et Mutzel (2020). Les expérimentations menées sur des graphes routiers réels montrent que la méthode optimisée réduit considérablement le temps de calcul — jusqu'à cinq fois plus rapide — tout en conservant une cohérence supérieure à 40 % avec la version exacte. Cette efficacité, combinée à une précision stable, démontre la pertinence de l'approche pour l'analyse de réseaux de grande échelle, qu'ils soient urbains ou sociaux.

Bibliographie

[1] Paul W. Olsen, Alan G. Labouseur et Jeong-Hyon Hwang. *Efficient Top-k Closeness Centrality Search*. In IEEE 30th International Conference on Data Engineering (ICDE), pages 196–207, 2014.

[2] Sorbonne Université. *Cours : Centralités (Closeness, Betweenness, ...)*. Disponible en ligne : https://moodle-sciences-25.sorbonne-universite.fr/pluginfile.php/250523/mod_resource/content/1/cours_centralites.pdf.

[3] Lutz Oettershagen et Petra Mutzel. *Efficient Top-k Temporal Closeness Calculation in Temporal Networks*. In IEEE International Conference on Data Mining (ICDM), pages 402–411, 2020.

[4] P. Flajolet and G. N. Martin. *Probabilistic Counting Algorithms for Data Base Applications*. Journal of Computer and System Sciences, vol. 31, no. 2, pp. 182–209, 1985.

[5] Meriem Benaissa, Massin Sadi, Aksil Sadi, et al. *Efficient Top-k Closeness Centrality Search*. Code source disponibles sur : https://github.com/MassinS/Projet_AAGA.