Projet – UE AAGA
Master 2 – Sorbonne Université

Dissertation

Calcul de jumeaux temporels.

DUTRA
Enzo
Avril 2021

TABLE DES MATIERES

INTRODUCTION

Dans cette dissertation, nous allons analyser la méthode de calcul de jumeaux temporels proposé sur le papier *« Computing Temporal Twins in Time Logarithmic in History Length »* de Binh-Minh Bui-Xuan, Hugo Hourcade, et Cédric Miachon.

Il y est en effet proposé une méthode permettant de calculer tous les jumeaux temporels d'un graphe temporel en temps logarithmique par rapport à la durée totale de l'histoire représentée.

Un jumeau temporel est, pour un instant t une paire de sommets u et v ayant les mêmes voisins en dehors de u et v.

La détection de tels jumeaux sur des graphes temporels peut permettre de détecter des similarités de comportement entre plusieurs agents et ainsi distinguer des groupes, par exemple, ce qui peut avoir plusieurs applications, notamment en optimisation pour des serveurs de données ou dans l'analyse des interactions pour des réseaux sociaux ou autres recherches sur le comportement.
Dans un graphe statique, retirer les jumeaux peut également permettre de réduire la complexité d'analyse.

Nous proposerons ensuite une alternative que nous analyserons et comparerons avec l'algorithme proposé par le papier.

DEFINITION DU PROBLEME

Nous allons ici préciser la définition des termes que nous allons utiliser dans la suite du papier.

Vrais jumeaux : u et v sont dit « vrais jumeaux » s'ils possèdent les mêmes voisins et qu'ils ne sont pas voisins entre eux.

Faux jumeaux : u et v sont dit « faux jumeaux » s'ils possèdent les mêmes voisins en dehors de {u, v} et qu'ils sont voisins entre eux.

Jumeaux éternels : u et v sont dit « jumeaux éternels » s'ils sont vrais jumeaux ou faux jumeaux à tout moment t dans le graphe temporel.

Δ-jumeaux : u et v sont dit « Δ-jumeaux » s'ils sont vrais jumeaux ou faux jumeaux pendant au moins Δ moments t consécutifs dans le graphe temporel.

L = (T, V, E) : Graphe temporel entier, où T est l'ensemble des moments, V l'ensemble des sommets et E l'ensemble des arrêtes.

Le papier que nous étudions ici apporte des solutions pour deux problèmes :

- Trouver tous les jumeaux éternels d'un graphe temporel le plus rapidement possible
  - ⇨ Ce sujet n'étant pas central, ni étudié en détail dans le papier, nous nous concentrerons sur le deuxième problème ;
- Trouver tous les delta-jumeaux d'un graphe pour un delta donné le plus rapidement possible.

Ici le but recherché est donc la vitesse d'exécution mais la complexité en espace reste tout de même considérée, ce qui a notamment mené à proposer une variante de l'algorithme pour certains cas.

Une particularité qu'il est important de noter dans cet article est qu'on s'intéresse d'avantage au temps de calcul relativement à la durée totale de l'histoire qu'au nombre de points du graphe ou tout autre paramètre.

Pour mesurer les performances des algorithmes étudiés par la suite, nous utiliserons les notations suivantes :

n : nombre de sommets
m : total des arêtes enregistrées
τ : longueur totale de l'histoire
λ : nombre de moments t enregistrés
N : taille de l'output

ALGORITHME PROPOSÉ

Le papier propose de revisiter l'implémentation de raffinement de partition basée sur une matrice d'adjacences en utilisant un arbre rouge noir lors de la génération du résultat. Cet algorithme a été appelé MEI

L'utilisation d'une matrice pouvant mener à une surconsommation de la RAM, une variante sans cette matrice est également proposée. Cette variante a été appelée MLEI.

Dans les deux algorithmes, on commence en supposant tous les couples {a, b} (avec $a, b \in V$) jumeaux. On va ensuite chercher un « splitter », c'est-à-dire un sommet étant voisin avec a mais pas avec b.

Algorithme 1 :

Entrée : Un graphe temporel et un entier delta
Sortie : Une liste de tous les delta-twins du graphe temporel

```
Initialisation de Tw(u, v) qui est un arbre contenant pour un couple
{u, v} les intervalles de temps où les deux membres de ce couple
peuvent être jumeaux.
Initialisation de R pour toutes les entrées de Tw.

Pour toute paire enregistrée (t, {u, v}) de E :
  Pour tout sommet w :
    Si (t, {u, w}) n'appartient pas à E alors
      Retirer les instants t de Tw(v, w)
      Si le retrait épuise tous les instants de Tw(v, w) :
        Retirer l'entrée (v, w) de R
      Fin si
    Fin si
  Fin pour tout
Fin pour tout

Pour tout (u, v) restants dans R :
  Scanner tous les intervalles d'au moins delta instants consécutifs
  dans Tw(u, v) et ajouter les Delta-jumeaux associés à la sortie.
Fin pour tout

Retourner la sortie
```

La gestion de l'arbre a un certain cout. En effet la suppression d'un t pour un couple donné dans celui-ci est en O(d) avec d la profondeur de l'arbre.
Une optimisation est cependant apportée dans le cas où les certains intervalles restants soient trop petits pour produire des Delta-jumeaux.

On garde l'arbre équilibré pour limiter le nb d'opérations mais cela a également un cout dont la complexité théorique dans le pire des cas est de O(d) par moment t supprimé.

La complexité théorique dans le pire des cas de MLEI est la suivante :

$$\boxed{O(m^2 \times n\log(\tau) + N)}$$

Avec une matrice, elle est de $O(m^2 \times n\log(\tau) + N)$ mais la complexité en espace explose.

DISCUSSION ET PROPOSITION D'AMÉLIORATION

Dans cette partie, nous allons tenter de trouver une solution alternative de calcul de delta-jumeaux temporels.

On peut identifier 3 points ralentissant la recherche du résultat dans l'algorithme MLEI :

1. Récupérer des informations sur les voisins de chaque points O(n^2)
2. Gérer un arbre rouge noir qui peut prendre beaucoup d'espace mémoire et demander beaucoup d'opérations pour son entretien
3. Déterminer si une paire de points est delta-jumeaux ou non

Dans la proposition que nous allons étudier ici, nous allons tenter d'économiser un maximum d'opérations induites par les trois points ci-dessus.

1. Nous allons utiliser une liste d'adjacence au lieu d'une matrice, comme proposé dans l'algorithme MEI, ce qui va nous permettre d'accéder à la liste des voisins de chaque points en O(1) sans pour autant que cela prenne trop de mémoire.
2. Nous allons utiliser une simple liste contenant tous les résultats des calculs permettant de dire si une paire de points sont jumeaux ou non. L'ajout d'une information se fait donc en O(1), la lecture du tableau entier en O(λ) car le tableau est de taille λ et aucun calcul de rééquilibrage n'est nécessaire. A savoir que dans les cas étudiés par l'article, λ n'est pas nécessairement proportionnel à τ.
3. Nous allons utiliser la taille de la liste des voisins pour éliminer d'office un grand nombre de paires de points qui ne peuvent pas être jumeaux en O(1).

Nous allons désormais parler « d'état » d'une paire pour faire référence au fait qu'elle soit composée de deux points jumeaux ou non.

Pour éviter d'avoir à calculer l'état d'une paire de points qui ne peuvent pas être jumeaux à un instant t mais dont un des points est présent dans une autre paire à cet instant, nous allons créer une liste de « points d'intérêt » pour chaque sommet du graphe lors du calcul de l'état des paire données en entrée.

Pseudo-code de la proposition

Algorithme « perso » :

```
Entrées :
Liste d'adjacences pour chaque moment t de Linkstream L : (T, V, E)
Liste de tous les sommets de Linkstream L : (T, V, E)
Un entier delta
MAX_T = t maximum dans Linkstream L : (T, V, E)
MIN_T = t minimum dans Linkstream L : (T, V, E)

Sortie :
Liste de tous les delta-jumeaux

G_index = Liste de couples et leur etat (jumeaux ou non)
HashMap<Point a : int, HashMap<Point b : int, TreeMap<Moment t : int,
Etat : boolean>>>

P_interets = [] // liste des points d'intérêt
HashMap<Point : int, TreeSet<Moment t : int>>

Points_vus = []
Pour chaque moment t de du graphe temporel :
  Pour chaque sommet a du graphe répertorié au moment t
    Pour chaque sommet b du graphe répertorié au moment t
      Si a ou b n'ont pas été vus :
        points_a_etudier = []
        points_a_etudier recois a, b ainsi que tous les voisins de a et b

        P_interets.ajouter(tous les points de points_a_etudier)

        Pour tout x dans points_a_etudier :
          Pour tout y dans points_a_etudier :
            Si x < y :
              Boolean etat = non jumeaux

              Si x et y sont jumeaux :
                Etat = jumeaux
              Fin si

              G_index[x][y][t][etat]
            Fin si
          Fin pour tout
        Fin pour tout

        Points_vus.ajouter(tous les points de  points_a_etudier)
      Fin si
    Fin pour tout
  Fin pour tout
  Points_vus.vider()
Fin pour tout
```

```
Resultat = []

Pour tout u de la liste de tous les sommets de Linkstream L : (T, V, E)
  Pour tout v de la liste de tous les sommets de Linkstream L : (T, V, E)
    Si u < v :
      Si P_interets contient u OU P_interets contient v :
        Tous_p_interets = P_interets[u] union P_interets[v]

        Dernier_t = MIN_T – 1
        Pour tout Tous_p_interets trié :
          Si (
            // Si pour tout t un des points a des voisins mais pas
            l'autre, ils ne peuvent pas être jumeaux
            (G_index[u] n’existe pas OU G_index[v] n’existe pas)
            // Si à ce t un des points a des voisins mais pas l'autre,
            ils ne peuvent pas être jumeaux
            OU (G_index[u][v] n’existe pas)
            // Si aucune reférence du couple au moment t mais que 1 des
            pts est dans un autre couple, ils ne peuvent pas être
            jumeaux
            OU (G_index[u][v][t] n’existe pas)
            // Si le couple n'est pas composé de deux sommets jumeaux
            OU (G_index[u][v][t] == non jumeaux)
          ) :
            Si (t - Dernier_t) >= delta) :
              Resultat.ajouter(DeltaJumeau(Dernier_t + 1, t – 1, u, v))
            Fin si
            Dernier_t = t
          Fin si

          // Si le couple est bien composé de deux sommets jumeaux
            // on ne faire rien
        Fin pour tout
        Si (MAX_T – Dernier_t) >= delta
          Resultat.ajouter(DeltaJumeau(Dernier_t  + 1, MAX_T, u, v))
        Fin si
      Sinon
        Resultat.ajouter(DeltaJumeau(MIN_T, MAX_T, u, v))
      Fin si
    Fin si
  Fin pour tout
Fin pour tout

Retourner Resultat
```

La création des listes d'adjacence se fait en même temps que la lecture du fichier avec une quantité d'écriture constante pour chaque ligne (qui contient une arête à un moment t). Ce traitement a donc une complexité en O(m)
La taille de la liste est de l'ordre de $2m$ (car chaque sommet contient l'information de toutes les arêtes dont il est membre).
Cette structure étant sous la forme de HashMaps imbriqués, la complexité d'accès à n'importe quel élément est en O(1).

On peut voir que l'algorithme commence par, pour chaque t enregistré (sachant que $\lambda \leq m$), analyser chaque couple de sommets étant voisins (O(m)). Pour chaque couple, on va chercher un spliter dans une des deux listes de voisins. L'opération « contains » étant en O(1) avec un hashset, seul la taille de la plus longue liste des voisins compte dans la complexité de la comparaison. De plus étant donné que chaque point effectivement enregistré au moment t n'est observé qu'au maximum x fois pour x son nombre de voisins, la complexité maximale est de $O(m^2)$.
Pour réduire au maximum la comparaison de listes, on va d'abord comparer leur taille, ce qui va nous permettre de remplacer ce $O(m^2)$ théorique par quelque chose de plus proche de $O(m)$ en pratique. Il est en effet très peu probable que presque toutes les listes de voisins aient la même taille à part dans un graphe extrêmement dense.

On va ensuite tenter la construction de tous les Delta-Jumeaux possible, avec une complexité maximale en $O(n^2)$, et s'arrêter pour chacun de ces potentiels jumeaux aux points d'intérêt dont la quantité maximale est de m.

La complexité théorique dans le pire des cas de cet algorithme est donc de $O(m^2 + n^2 \times m)$
Pour comparer cette complexité avec celle de l'algorithme décrit dans le papier, on va utiliser la notation N au lieu de $n^2 \times m$. La complexité maximale théorique est donc la suivante :

$$\boxed{O(m^2 + N)}$$

En utilisant la comparaison de taille de listes, on va arriver dans la plupart des cas de graphes peu denses à une complexité proche de :

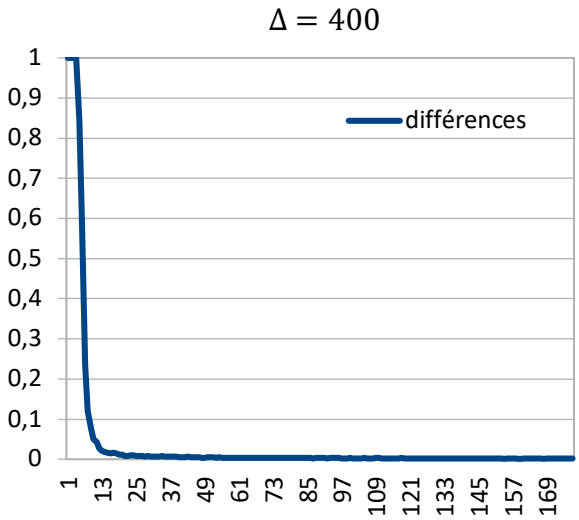
$$O(m + N)$$

Validité de l'algorithme

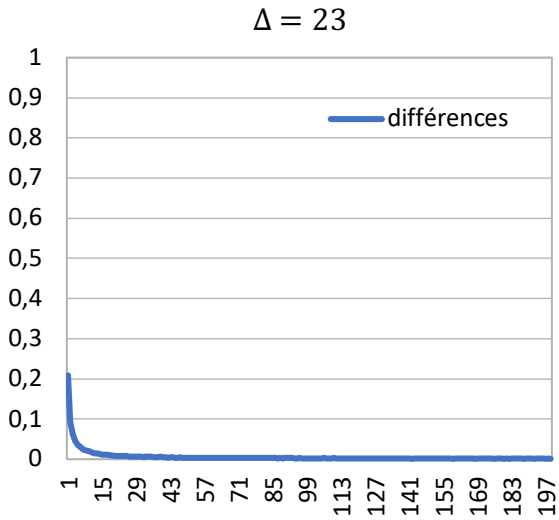
L'amélioration proposée ne renvoie pas exactement les mêmes résultats que l'algorithme de référence (l'implémentation Java de MLEI). On a en effet une différence de quantité de DeltaJumeaux en sortie de l'ordre de 0.1 à 1%.

Pour pouvoir mesurer cette différence plus précisément, nous avons calculé un « taux de différence » qui est la proportion de delta-edges différents entre deux résultats. Un taux à 1 signifie que les deux résultats n'ont aucun delta-edge en commun, tandis qu'un taux à 0 signifie que les deux résultats sont strictement égaux

En calculant ce taux pour toutes les instances de time-progression, on obtient les résultats suivants :

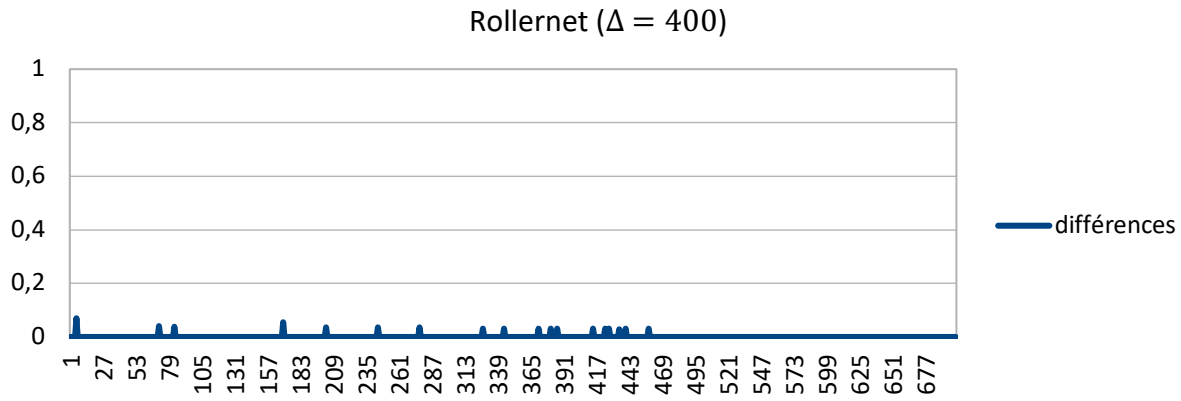


$\Delta = 400$

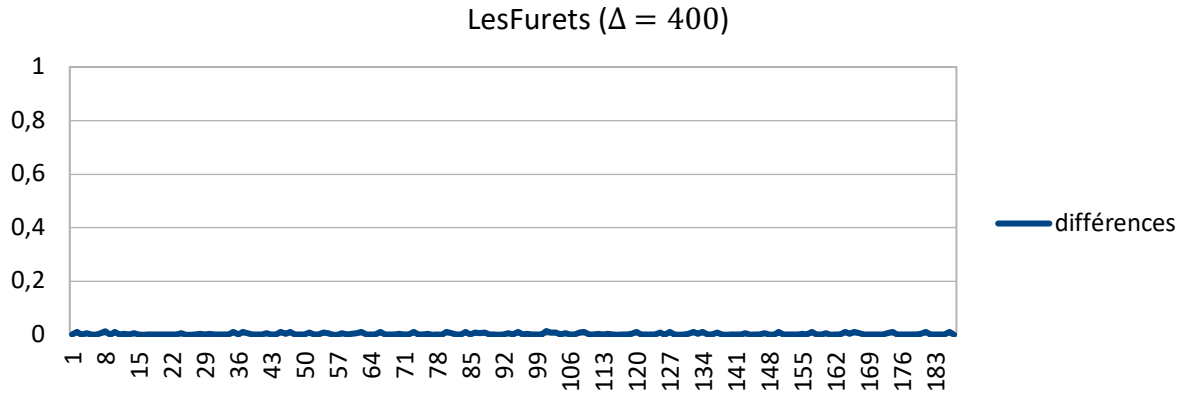


$\Delta = 23$

On voit ici une grosse différence dans les premières instances (ou le nombre de delta-jumeaux retournés est de l'ordre de la dizaine à la centaine) mais qui décroit très rapidement à mesure que la taille de la sortie augmente.



Ici la différence est nulle (< 0.000) dans 97% des tests avec quelques pics allant jusqu'à 0.07.



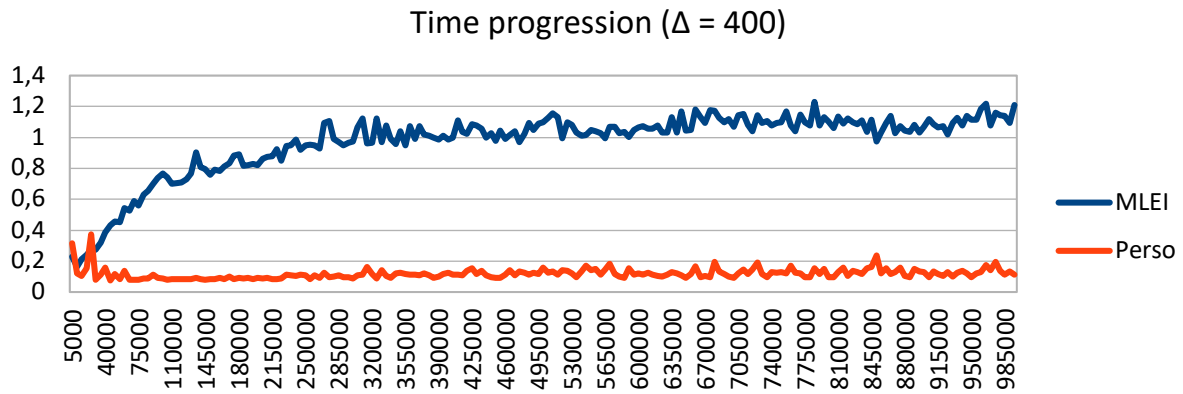
Enfin, pour les tests sur les données de LesFurets, la différence fluctue beaucoup avec un maximum à 0.014 sur 291 tests.

Au vu des éléments ci-dessus, on peut raisonnablement dire que l'algorithme 1 est, dans le pire des cas, une approximation relativement proche de la réalité.

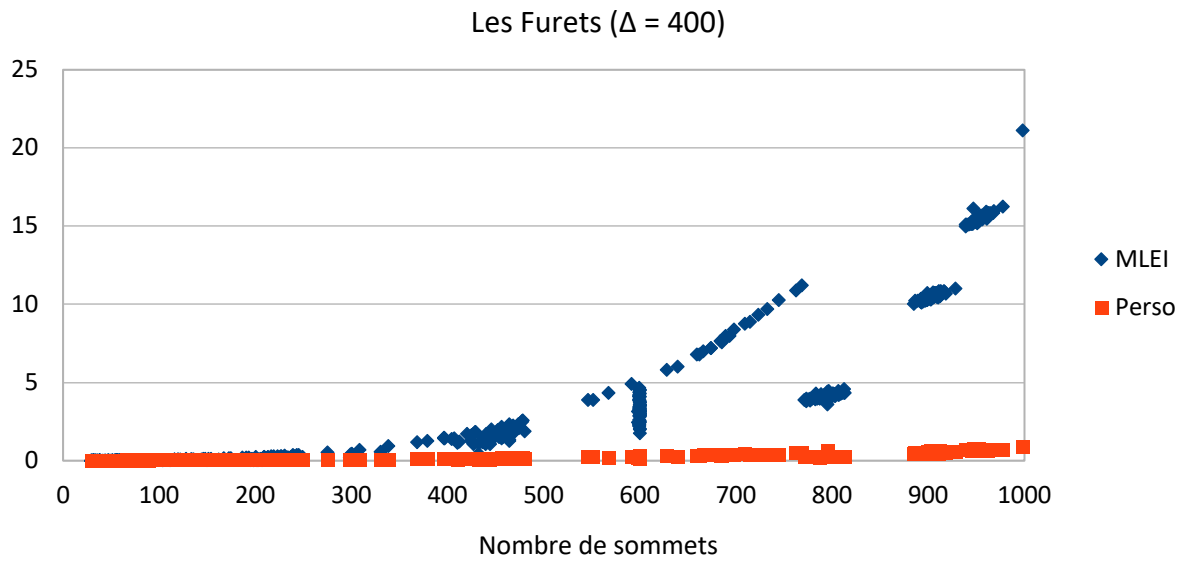
S'il s'avérait qu'une telle approximation serait inexploitable en tant que tel dans le cas pratique, elle pourrait cependant rester intéressante pour d'éventuelles optimisations de MLEI tels que l'utilisation de structure de données non dynamiques car on aurait déjà une idée de l'espace mémoire à réserver.

Comparatif de performances

Nous allons maintenant comparer les performances des deux algorithmes.



Sur la base de données time-progression, on voit une différence de temps de calcul assez conséquente.
On remarque que l'évolution des deux courbes est opposée dans les premières instances de test. Les pics en temps de calculs de l'algorithme perso dans les premières instances qui sont très denses confirme peut-être la faiblesse de la comparaison sur les tailles des listes lorsque presque tous les sommets sont voisins entre eux.

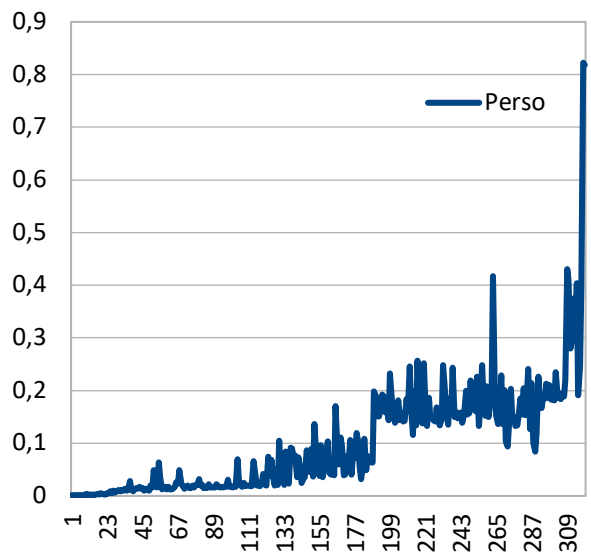


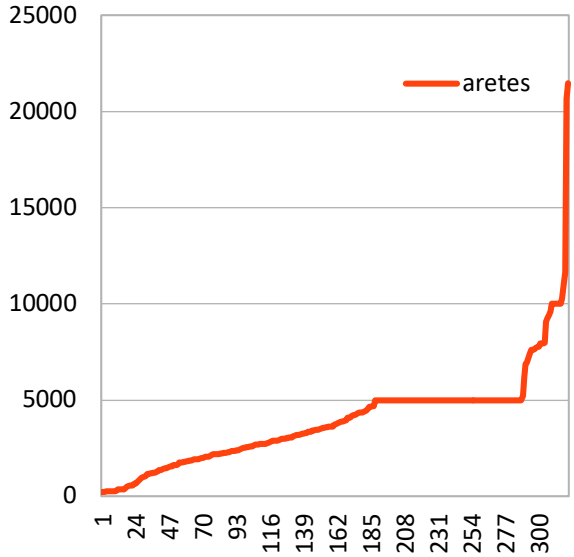
On voit ici aussi que les paramètres influençant le temps de calcul de l'algorithme MLEI n'influencent pas le temps de calcul de l'algorithme perso de la même manière.

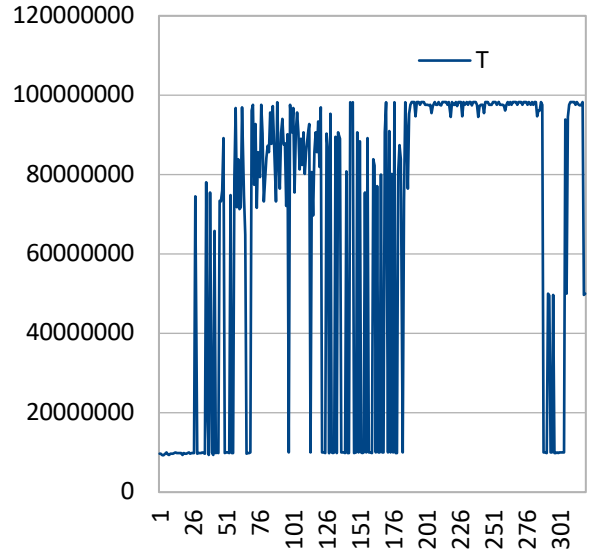
Analyse de la complexité en pratique

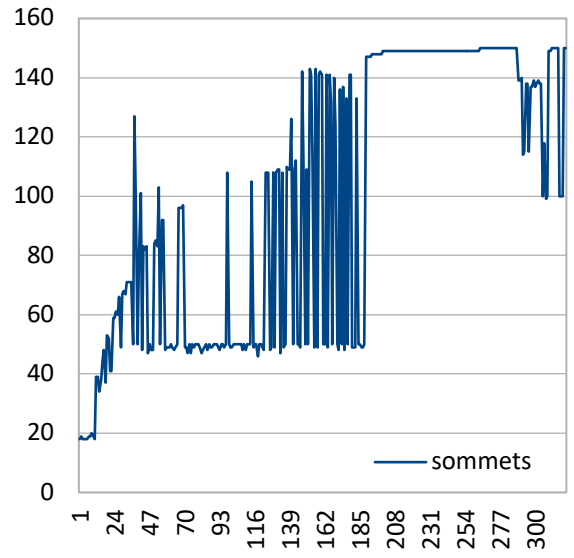
Nous compléter l'analyse de la complexité théorique dans le pire des cas faite précédemment en essayant d'observer quel paramètre du dataset a l'influence la plus notable sur le temps de calcul de notre proposition.

Après quelques essais sur différents datasets, nous avons pu, de manière relativement approximative, sur enron qui contient beaucoup de combinaisons de paramètres différents, isoler les facteurs influant le plus sur le temps de calcul de notre algorithme en pratique.









Ici, on a trié les instances de test par nombre d'arêtes puis par nombre de sommets en cas d'égalité ce qui explique la propreté du graphique montrant la quantité d'arête pour chaque instance de test.
On peut remarquer que l'évolution du temps de calcul de l'algorithme perso suit approximativement la courbe de la quantité d'arêtes tout en étant, dans une moindre mesure, également influencé par le nombre de sommets.
Comme entrevu dans la section précédente, τ ne semble pas avoir d'influence notable sur le temps de calcul.

Ce résultat met en évidence l'impact plus ou moins limité du facteur m^2 dans la complexité maximale théorique de l'algorithme perso qui est de $O(m^2 + N)$. Comme vu précédemment un certain nombre d'optimisations supplémentaires permettent de déduire certaines situations et ainsi éviter beaucoup de calculs inutiles.

Il semblerait donc que le facteur le plus important lors des différentes séries de tests qui ont pu être effectuées soit le nombre d'arêtes du fichier fournis, ce qui correspond dans notre cas à la taille de l'entrée.

CONCLUSION

Après avoir décrit l'algorithme proposé par le papier étudié, nous avons pu identifier quelques points requérant beaucoup de temps de calcul et proposer une alternative qui semble, à première vue, intéressante.

Il serait probablement judicieux d'étudier l'alternative proposée plus en détail avant de se prononcer définitivement sur la supériorité ou non de l'un sur l'autre. Il est en effet tout à fait possible que l'alternative proposée performe particulièrement mal dans certaines conditions précises qu'il serait intéressant de rechercher.