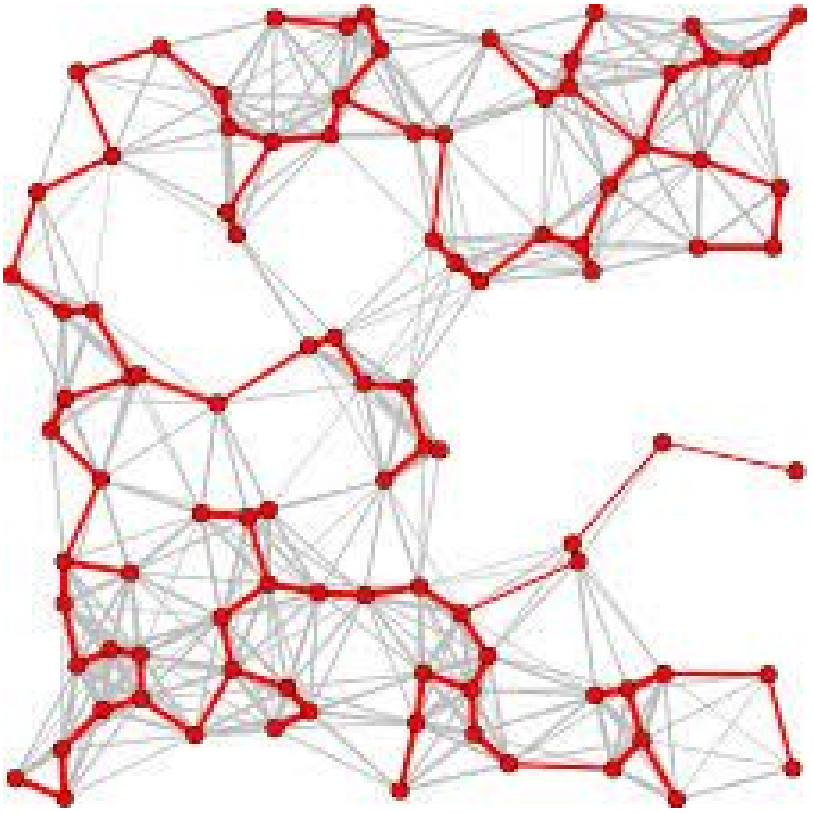
GASSION Axel
GOUBIN Fabien

INFO-3A

# Projet Spanning Tree



# I. Introduction

Les minimum spanning tree sont des arbres couvrants d'un graphe donné, pour lequel la somme des poids des arcs est minimale.
Ils sont particulièrement utiles pour les réseaux (notamment téléphoniques).
Globalement, le but est d'être rapide tout en respectant la contrainte de traiter tout le monde.

# II. Implémentation

Comme ce projet se centre plus sur les arcs que les sommets d'un graphe, contrairement au projet de coloration, nous avons décidé de changer drastiquement d'implémentation pour nos graphes.
Pour cela nous avons décidé d'utiliser deux librairies java :
Tout d'abord, nous avons utilisé la classe Pair de javafx. Cette classe est implémentable à la main, mais nous allons simplement l'importer.
Cette classe nous permet de créer une Pair<T,T>. Ici, nos types génériques seront des entiers correspondant aux sommets, ce qui nous permet facilement de dire qu'une paire représentera un arc.
Ensuite, nous avons utilisé les HashMap<T,T>. Elles vont nous permettre d'associer une clé en premier élément à une valeur. Ainsi, on associera une paire qui sera donc unique, à un poids.
Notre graphe est donc entièrement implémenté.
Le nombre de sommets est obtenable dans les fichiers .mst (2ème ligne), et on peut dans le pire des cas le récupérer en trouvant le maximum dans les paires.
Les arcs sont tous dans la HashMap, et sont tous associés à leur poids.

L'avantage de cette implémentation est la complexité. En effet, le temps d'accès en moyenne pour une HashMap est en O(1), et dans le pire des cas en O(n), ce qui nous permet de garder un temps d'exécution raisonnable.

***Nous avons effectué des tests sans librairies, et normalement, elles sont bien implémentées et référencées, mais si un problème survient, il est possible que l'IDE ne contienne pas javafx, utilisé pour la classe Pair.***

# III. Organisation de notre archive

Notre archive est divisée en plusieurs parties.

## A. Le dossier graphes

Ce dossier contient tous les fichiers de test fournis pour le projet, en .mst.

## B. Le dossier src

Ce dossier contient tout le code pour exécuter ce projet.

## C. Le dossier css

Le dossier css contient un fichier permettant un affichage plus confortable pour nos graphes.

## D. Le dossier res

Le dossier res contient deux fichiers textes de résultats remplis automatiquement par le programme.
Ces fichiers texte sont utilisés pour remplir le excel nommé analyse, où vous pourrez observer nos résultats, analyses, et quelques graphiques.
Ce document présente en première ligne tous les fichiers, et on leur associera le temps mis par un algorithme, le poids obtenu, ou encore un rapport entre deux algorithmes.
Les graphiques sont disponibles sur la partie en haut à droite de ce tableau.

## E. Le dossier lib

Le dossier lib contient les librairies extérieures pour l'affichage.

# IV. Programme

## A. Les différentes classes

### 1. La classe Main

Elle est la classe qui se lancera à l'initialisation.
On peut y saisir un graphe à étudier, et la valeur de d souhaitée pour notre d-MST.
On y trouve aussi une fonction qui a servi à lancer le calcul sur tous les fichiers afin d'enregistrer les résultats.

### 2. La classe STree

Elle est le coeur du programme, on y fait tout.
On peut y trouver les méthodes de lecture et d'écriture de fichiers.
On y trouve aussi les méthodes pour ordonner les HashMap représentant nos graphes par ordre croissant ou décroissant de poids des arcs, et une méthode pour obtenir la somme des poids des arcs.
Il y a les méthodes utiles aux principaux algorithmes qui vérifient la connexité, la présence d'un cycle, ou encore trouvent le meilleur arc en termes de poids.
Et enfin, il y a nos 4 algorithmes qui utilisent le tout afin de nous donner nos résultats.

## B. Affichage

Pour l'affichage, nous avons choisi d'utiliser la libraire GraphStream - GraphStream - A Dynamic Graph Library.
Elle permet l'implémentation et l'affichage d'objets de types graphes, mais nous ne l'utiliserons seulement pour l'affichage car l'implémentation est une partie de notre travail !
Ainsi, pour chaque algorithme utilisé, nous créerons un affichage qui initialisera un graphe avec des sommets, des arcs, puis l'afficher de manière claire.
Des fenêtres différentes s'ouvriront pour chaque algorithme pour observer les différents graphes
**Dans l'ordre (gauche à droite), Kruskal 1, Kruskal 2, Prim, d-MST.**
**Attention, en fermer une termine le programme et ferme donc les autres fenêtres !**
Pour se déplacer, il suffit d'utiliser les touches directionnelles du clavier.
Pour zoomer et dézoomer, il faut utiliser les touches ↧ et ↥.

## C. Utilisation de notre code

Pour utiliser notre code, vous pouvez lancer depuis un IDE (Eclipse par exemple) le main tel qu'il est présent dans l'archive.
On vous demandera le nom d'un fichier à tester (par exemple *crd300*), et le d personnalisé que vous souhaitez observer pour l'algorithme de d-MST.
De base, 4 fenêtres d'affichage seront ouvertes : une pour Kurksal1, une pour Kruskal2, une pour Prim, et une pour d-MST. Les résultats s'afficheront aussi dans la console (poids total, et temps d'exécution).
**A noter : il est possible qu'il y ait des problèmes de synchronisation et que certaines valeurs s'affichent avant la phrase les introduisant dans la console.**

Une 2ème grande partie de l'exécution est commentée. vous pouvez la commenter pour avoir une affichage de d-MST pour les valeurs de d suivantes : 5, 7, 9, 11.

Les deux fonctions *write()* et *writeMST()* sont commentées. Elles ont servi à écrire dans les fichiers de résultats après le lancement du programme sur tous les fichiers de dossier graphes.
Cela a été fait grâce à la partie commentée dans le main, qui appelle l'exécution pour chacun d'entre eux.
**A noter : Il n'est pas recommandé d'altérer le main, car cela pourrait créer énormément d'affichages d'un seul coup.**
Cependant, vous pouvez commenter les appels aux fonctions *init()* pour enlever un affichage souhaité.
Le reste des zones commentées sont des affichages de test.

# V. Algorithmes de MST

## A. Kruskal 1

### 1. Description

L'algorithme de Kruskal1 est un algorithme glouton et constructif. En effet, on part d'un graphe vide que l'on va alimenter petit à petit.
On y crée une "forêt" d'arbres qui va au fur et à mesure se réduire jusqu'à devenir un seul arbre.

### 2. Fonctionnement

*On trie le graphe initial G dans l'ordre croissant des poids des arcs*
*On crée un nouveau graphe vide T*
*Pour chaque arc appartenant* à *G* :
    *On vérifie que T n'a pas atteint son nombre d'arcs maximal* $(n-1)$
        *Si c'est le cas, renvoyer T*
    *On ajoute l'arc* à *T*
    *Si T est un cycle*
        *Alors on enlève l'arc*
*Fin Pour*

## B. Kruskal 2

### 1. Description

L'algorithme de Kruskal2 est un algorithme destructif. En effet, on part du graphe, et on retire des arcs jusqu'à obtenir un arbre.
Son principe est donc complètement l'inverse de Kruskal1.

### 2. Fonctionnement

*On trie le graphe initial G dans l'ordre décroissant des poids des arcs*
*On fait une copie T de G*
*Pour chaque arc appartenant* à *G* :
    *On vérifie que T n'a pas atteint son nombre d'arcs minimal* $(n-1)$
        *Si c'est le cas, renvoyer T*
    *On enlève l'arc* à *T*
    *Si T n'est plus connexe*
        *Alors on remet l'arc*
*Fin Pour*

# C. Prim

## 1. Description

L'algorithme de Prim est un algorithme glouton et constructif. Tout comme Kruskal1, on part d'un graphe vide que l'on va alimenter petit à petit.
Mais contrairement à Kruskal1, on y crée un seul arbre qui s'étend petit à petit

## 2. Fonctionnement

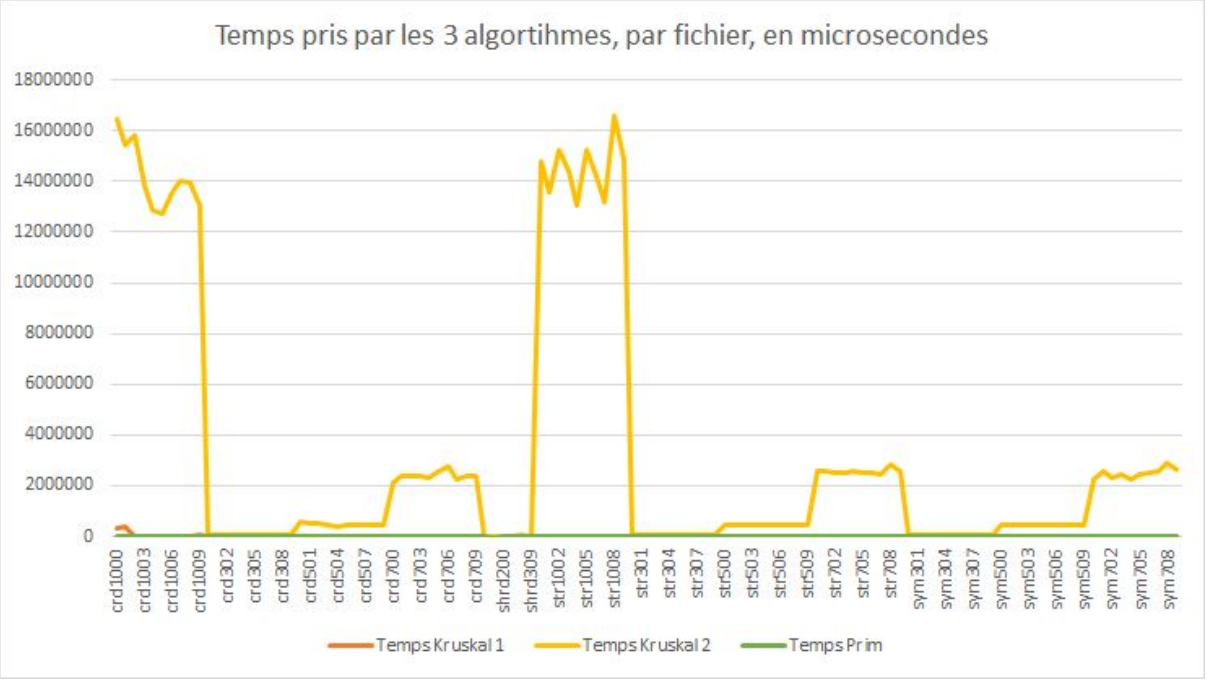
*On trie le graphe initial G dans l'ordre croissant des poids des arcs*
*On crée un nouveau graphe vide T*
*On crée une liste vide qui contiendra les sommets visités par l'arbre*
*On prend un sommet de départ aléatoire que l'on ajoute à la liste*

*Tant que l'on n'a pas traité tous les sommets*
    *On prend l'arc de poids le plus faible dont un sommet est dans la liste, et l'autre non*
    *On l'ajoute à T*
*Fin Tant que*

*Pour trouver l'arc de poids le plus faible, on parcourt à chaque fois G*

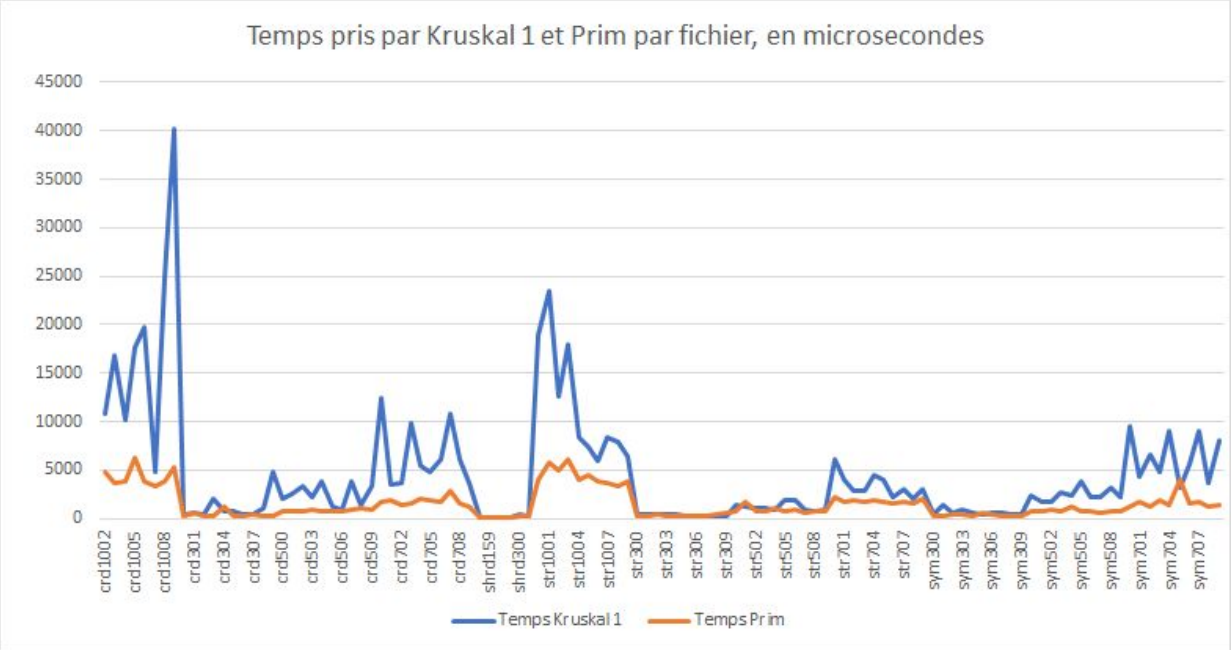
# D. Résultats

Les poids des arbres obtenus par l'exécution des algorithmes sont tous les mêmes (cf colonnes R et S du fichier excel).
Cependant, ces trois algorithmes ont des temps d'exécution bien différents entre eux, comme nous pouvons le voir sur ce graphique :




On y voit très clairement que l'algorithme Kruksal2 est largement plus long que ses deux autres compagnons. Normal ? Et bien oui !
Cet algorithme possède une complexité plus élevée dans notre code. De plus, il effectue des opérations plus longues, et en plus grand nombre.
On a donc un temps d'exécution qui grandit de manière très forte lorsque le nombre de noeuds (et donc d'arcs, on a $\frac{n(n-1)}{2}$ arcs possibles dans les fichiers) présents de base grandit. Aussi, l'implémentation d'un algorithme de parcours pour vérifier la connexité qui est utilisé à chaque itération fait fortement augmenter ce temps.
Enfin, il ne faut pas oublier qu'en tant que algorithme destructif, Kruskal2 doit retirer plus d'arcs que Kruksal1 doit ajouter. Là où dans le meilleur des cas, Kruskal1 trouve directement ses $n-1$ arcs et les insère dans le nouveau graphe, Kruskal2 doit retirer et donc faire des tests pour au moins $\frac{n(n-1)}{2} - (n-1)$ arcs car il doit en rester $n-1$ .

On aimerait maintenant comparer Kruksal1 et Prim puisque l'on ne peut pas vraiment le faire sur le graphique précédent :




**A noter : Nous avons retiré de cette étude les fichiers crd1000.mst et crd1001.mst car ils présentent des valeurs élevées pour Kruskal1 rendant le graphique moins lisible.**
On remarque ici que Prim est bien plus efficace que Kruskal1, dans la mesure où il ne dépasse que très rarement les 5 millisecondes. En moyenne, Kruskal1 prend 10ms, et Prim 1.7ms.
Ces deux algorithmes restent bien meilleurs que Kruskal2 pour les raisons énoncées plus tôt.
La différence entre eux est issue du test de cycle dans Kruskal1.
En effet, là où l'on doit vérifier qu'il n'y a pas de cycle pour chaque itération dans Kruskal1 $(O(n^2))$, on se contente de rechercher la meilleure paire dans Prim $(O(n))$. Cela crée cette petite différence qui est parfois quasi-nulle si les meilleures paires dans Prim se retrouvent à chaque fois loin dans notre HashMap.

# VI. Algorithme de d-MST

## A. Recherche & Pistes

Ici, le but est de former un arbre recouvrant de poids minimum avec une contrainte supplémentaire : le degré maximal de chaque sommet est fixé à d.
L'algorithme d-MST est plus complexe puisque c'est un problème NP-Complet.
On ne trouvera donc pas *à priori* de solution en temps polynomial pour un poids parfait, mais il faut donc essayer de minimiser ce temps, en maximisant la qualité de l'algorithme.

Nos recherches nous ont mené à penser à utiliser un des trois algorithmes implémentés précédemment.
Ainsi, nous avons utilisé Kruskal1.
**L'algorithme obtenu est plutôt performant en termes de complexité, mais le poids renvoyé n'est pas le meilleur possible.**

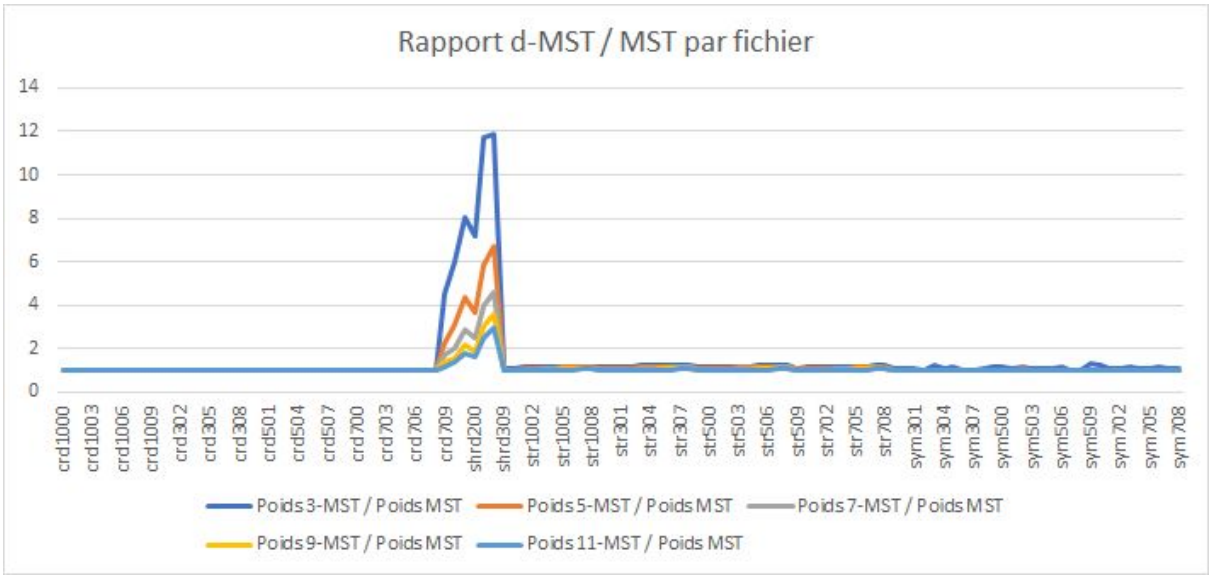
## B. Fonctionnement

*On trie le graphe initial G dans l'ordre croissant des poids des arcs*
*On crée un nouveau graphe vide T*
*On crée une HashMap M associant un noeud à son degré*
*On prend un sommet de départ aléatoire que l'on ajoute à la liste*

*Pour chaque arc appartenant à G :*
    *On vérifie que T n'a pas atteint son nombre d'arcs maximal* $(n-1)$
        *Si c'est le cas, renvoyer T*
    *On ajoute l'arc à T*
    *Si T est un cycle ou si l'un des deux sommets est déjà de degré maximal d dans M*
        *Alors on enlève l'arc*
*Fin Pour*

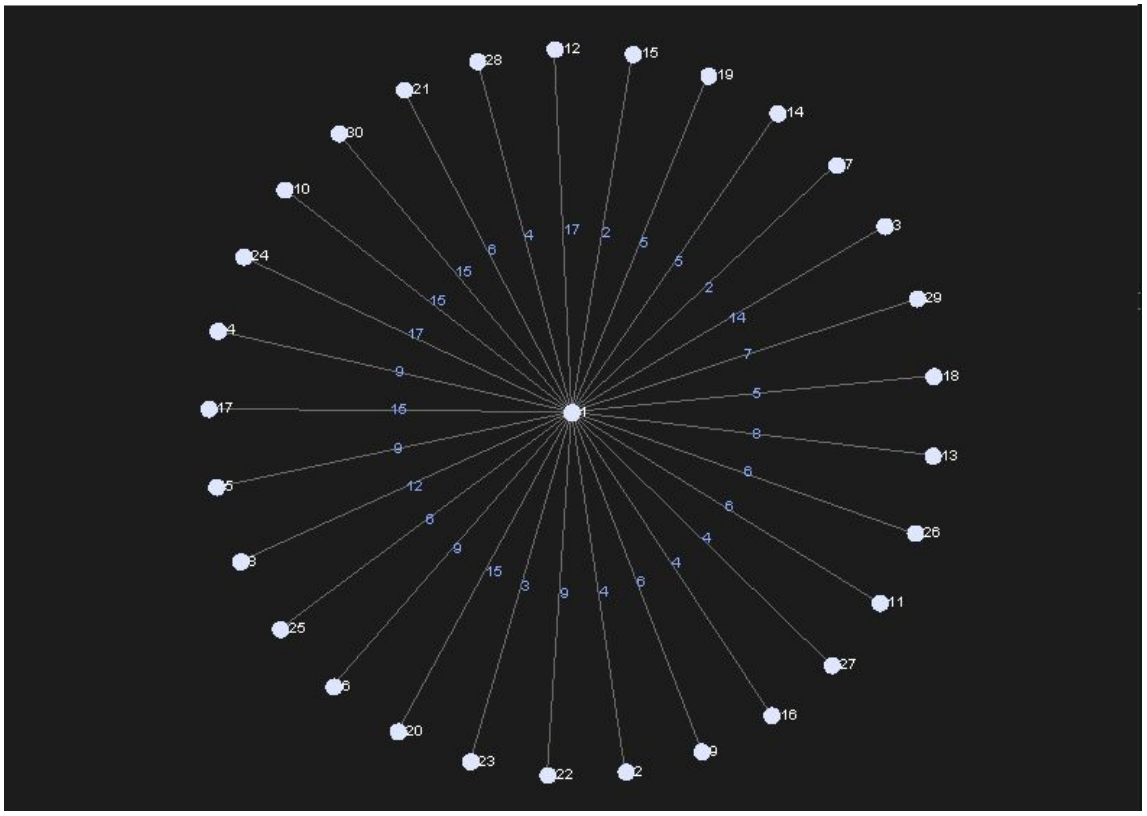
## C. Résultats & Analyse

Nous avons décidé de sauvegarder plusieurs d-MST pour les valeurs de d suivantes : 3 , 5 , 7 , 9 , 11.
Les résultats obtenus sont les suivants :

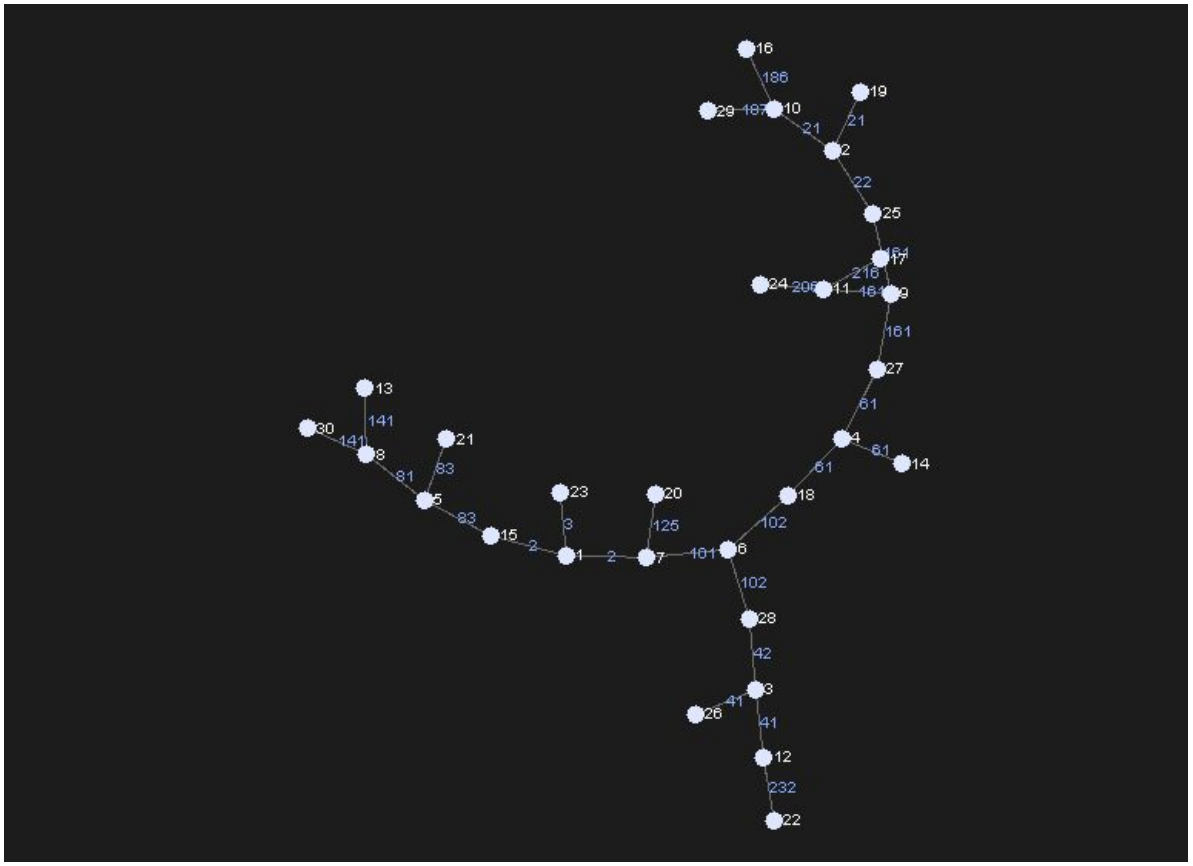



On obtient une courbe peu lisible. Les rapports obtenus pour les fichiers shrd sont élevés et diminuent quand d augmente.

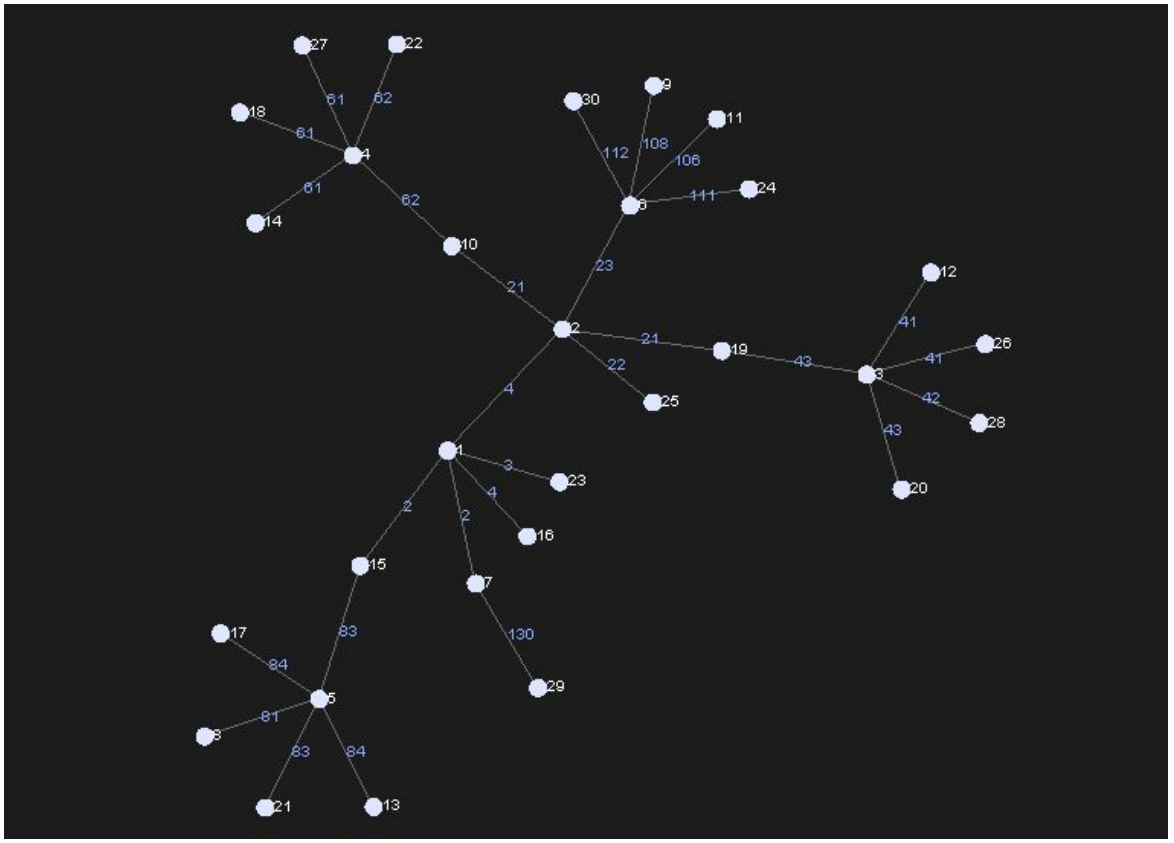
Regardons les résultats pour comprendre :
Pour Prim, on obtient :



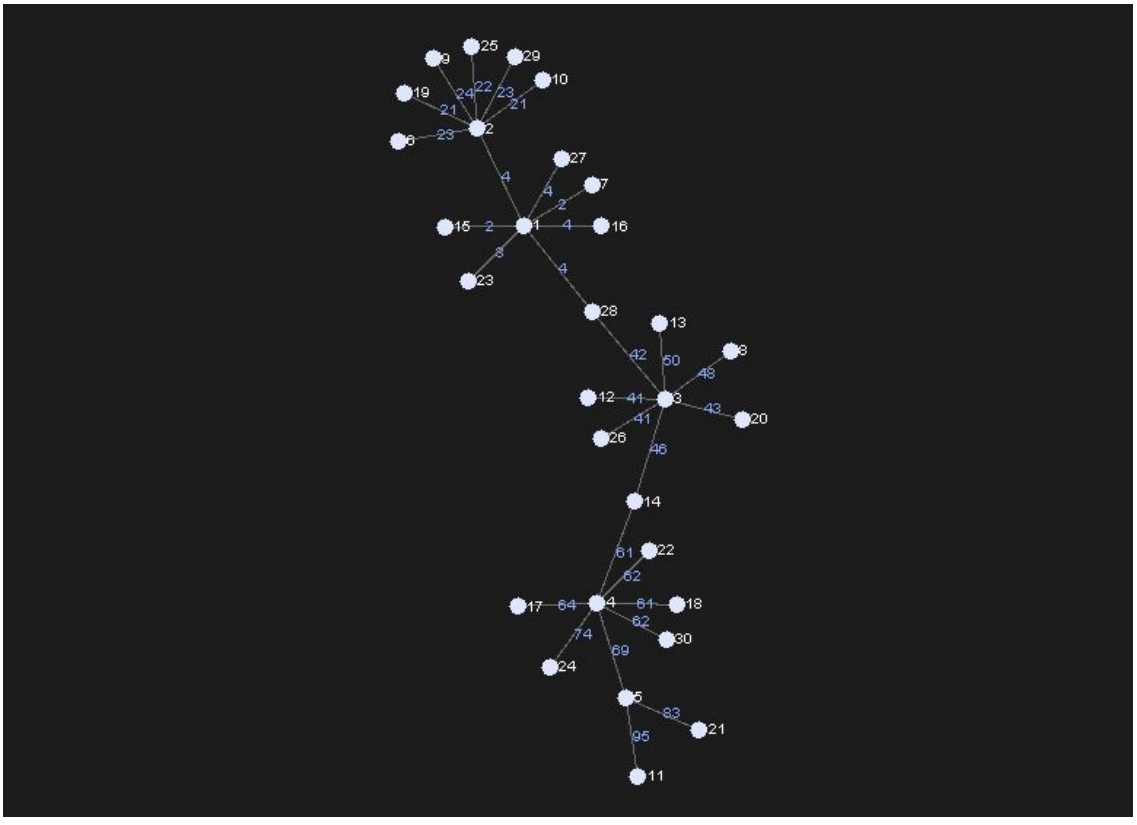

Pour un 3-MST :



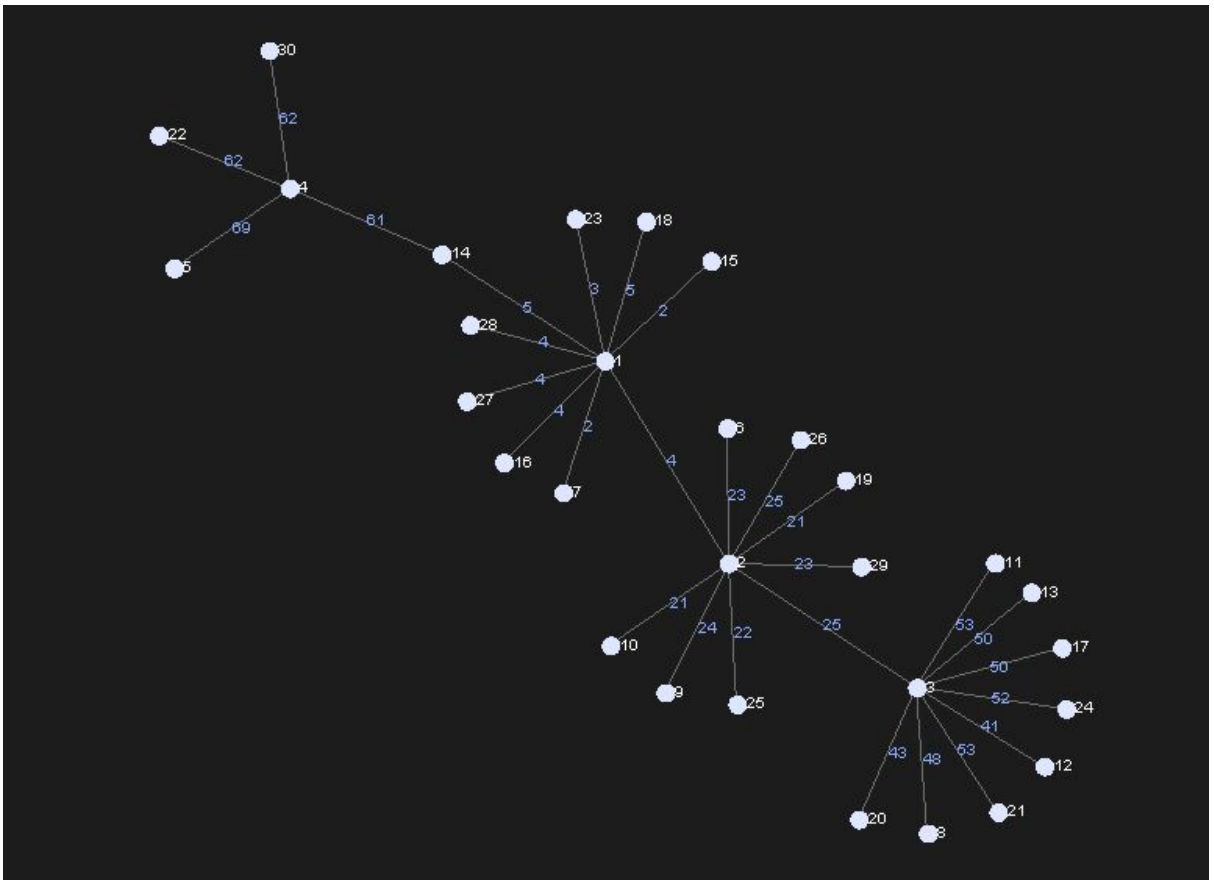
Pour un 5-MST :



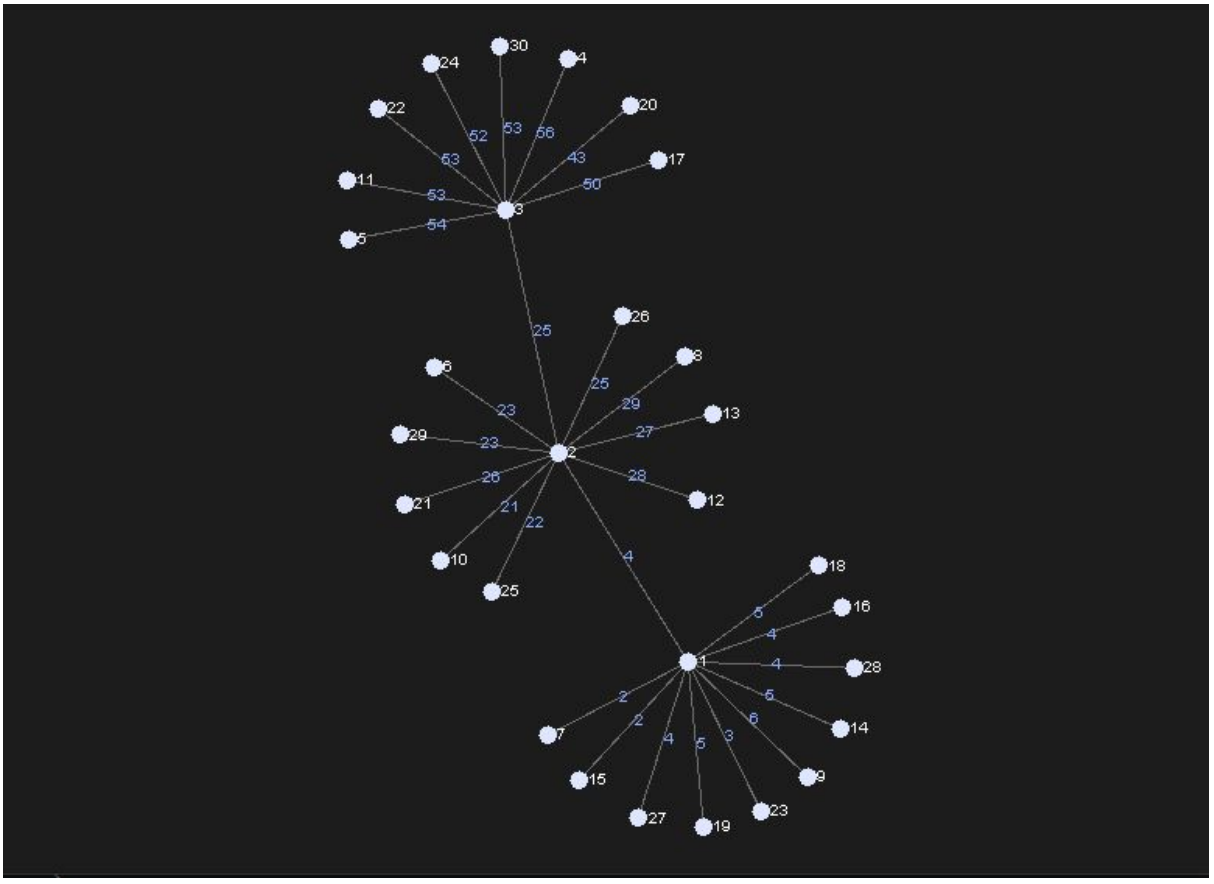

Pour un 7-MST :



Pour un 9-MST :

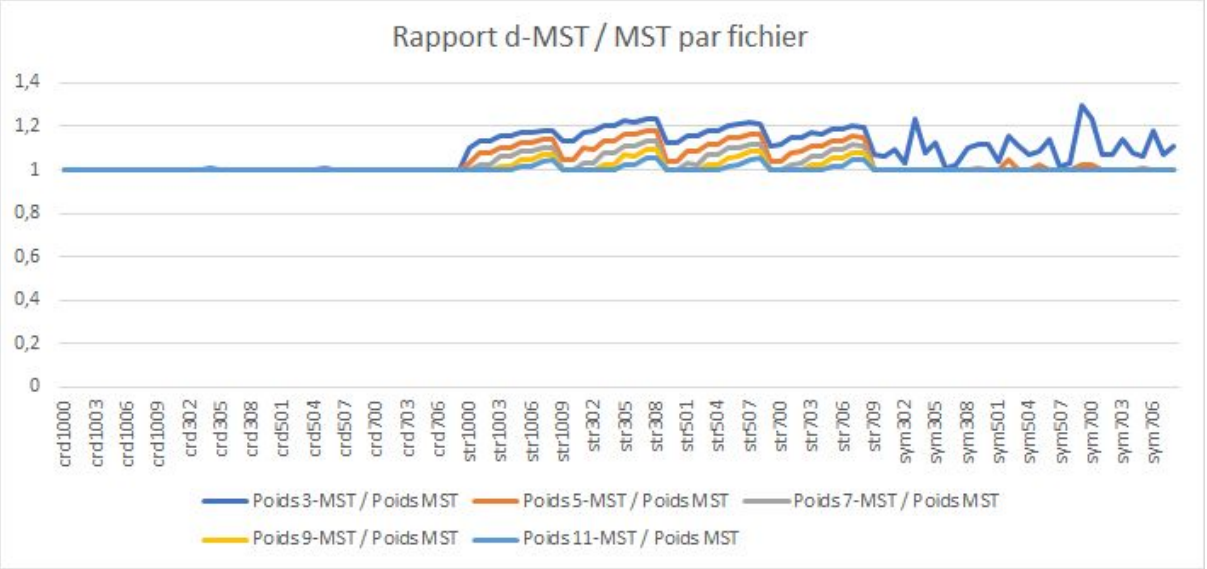



Et enfin, pour un 11-MST :



On peut en déduire que ces graphes sont très utiles pour tester notre algorithme d-MST. Plus on fait grandir la valeur de d, plus le graphe tend vers une forme proche de celle donnée par Prim, qui est idéale sans contrainte pour le degré des sommets. Il n'est donc pas étonnant d'obtenir un rapport élevé lorsque d est bas, car les poids des arcs sélectionnés par l'algorithme seront forcément plus grands que ceux sélectionnés par Prim.

Maintenant, retirons ces fichiers de notre graphique, et observons le résultat :




On peut remarquer plus facilement l'impact de d sur le poids total de nos graphes.
Les fichiers str en sont un parfait exemple puisque l'on peut bien voir la diminution du rapport des poids avec l'augmentation de d.
Les fichiers crd semblent être déjà très proches d'un état parfait, même avec 3 en contrainte, et les fichiers sym ont l'air d'être proches du résultat donné par Prim à partir de d=5.
Au final, la valeur de d a un impact qui peut être très important ou nul en fonction du graphe étudié !

## D. Autres idées

Globalement, notre algorithme fonctionne plutôt correctement.
Nous aurions pu créer une version avec Prim, car il est plus rapide que Kruskal1.
Cependant, il n'est pas adapté à tous les graphes, mais plutôt à ceux donnés pour ce projet.
En effet, un graphe initial G ne présentant pas tous les arcs possibles, mais seulement une partie, peut être traité plus facilement !
Dans le cas d'un tel graphe, on pourrait déterminer des conditions initiales pour ajouter certains arcs au nouvel arbre, comme par exemple :
Inclure les sommets de degré 2 qui sont des points de coupure (ils sont obligés d'être dans l'arbre).

Ajouter petit à petit les sommets de degré 1. Ces sommets ne possèdent qu'un arc et doivent donc être présents dans l'arbre.

Mais ce n'est pas tout ! Nos recherches nous ont amené à des d-MST où la contrainte sur le degré était propre à chaque sommet. L'algorithme devient alors un peu plus lourd, puisqu'il faut enregistrer chaque noeud et y associer son degré maximal, et après tester cette contrainte. Algorithmiquement parlant, ça va, mais le programme devient un peu plus long..
Nous avons donc aussi réfléchi à ce problème et effectué des recherches pour déterminer une possible condition pour supprimer certains arcs candidats pour être dans le nouvel arbre :
il suffirait d'ignorer les arcs entre deux noeuds qui ont une contrainte de poids égale à 1. Ces deux noeuds ne peuvent effectivement pas être reliés entre eux dans l'arbre, car ils formeraient une composante connexe à ce dernier à cause de cette contrainte.

Tout ça pour créer un algorithme plus flexible et capable de traiter plus de cas !

A noter que ces idées ne sont pas implémentées.

# VII. Conclusion

Ce projet nous a beaucoup intéressé et nous avons tous deux pris du temps pour nous y intéresser.
Cela a été très enrichissant d'effectuer des recherches pour le problème de d-MST, notamment avec les variantes que nous avons découvert.