# SAE 2.2 Visualisation de Graphe

---

Aulas Quentin
Dumarchat Gurvan

# Partie 1 - Développement Objet

## Fig 1.1 - Diagramme UML paquet graphs initial

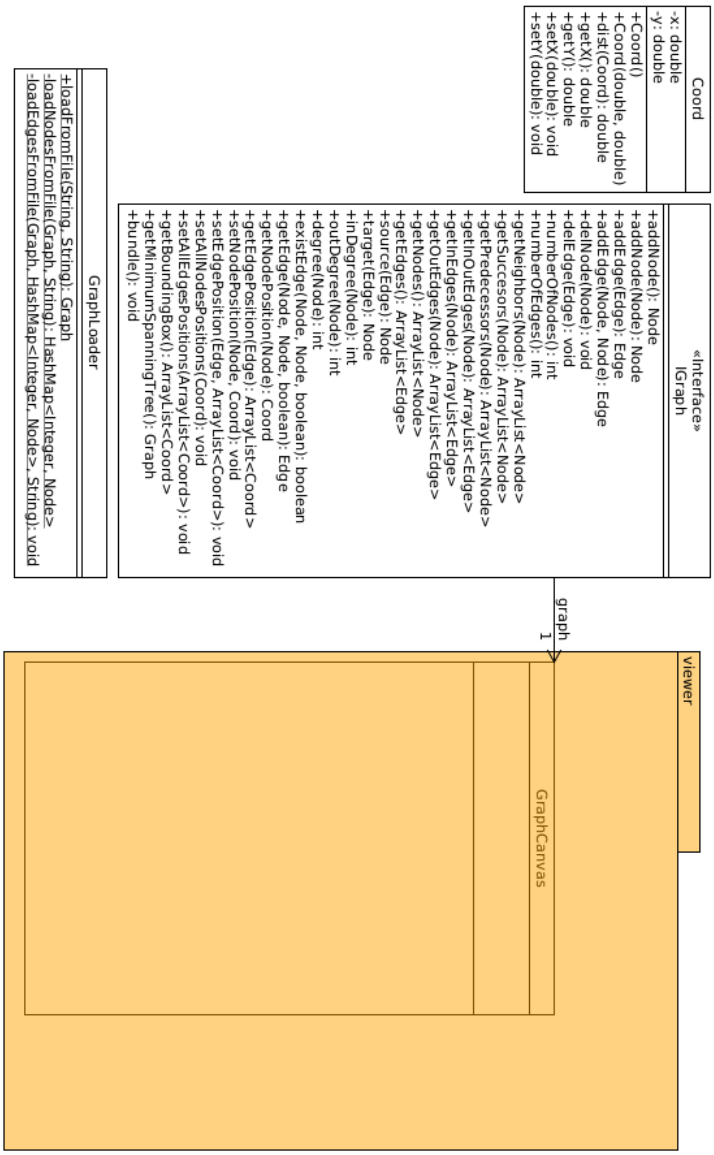


Fig 1.2 - Diagramme UML paquet viewer initial

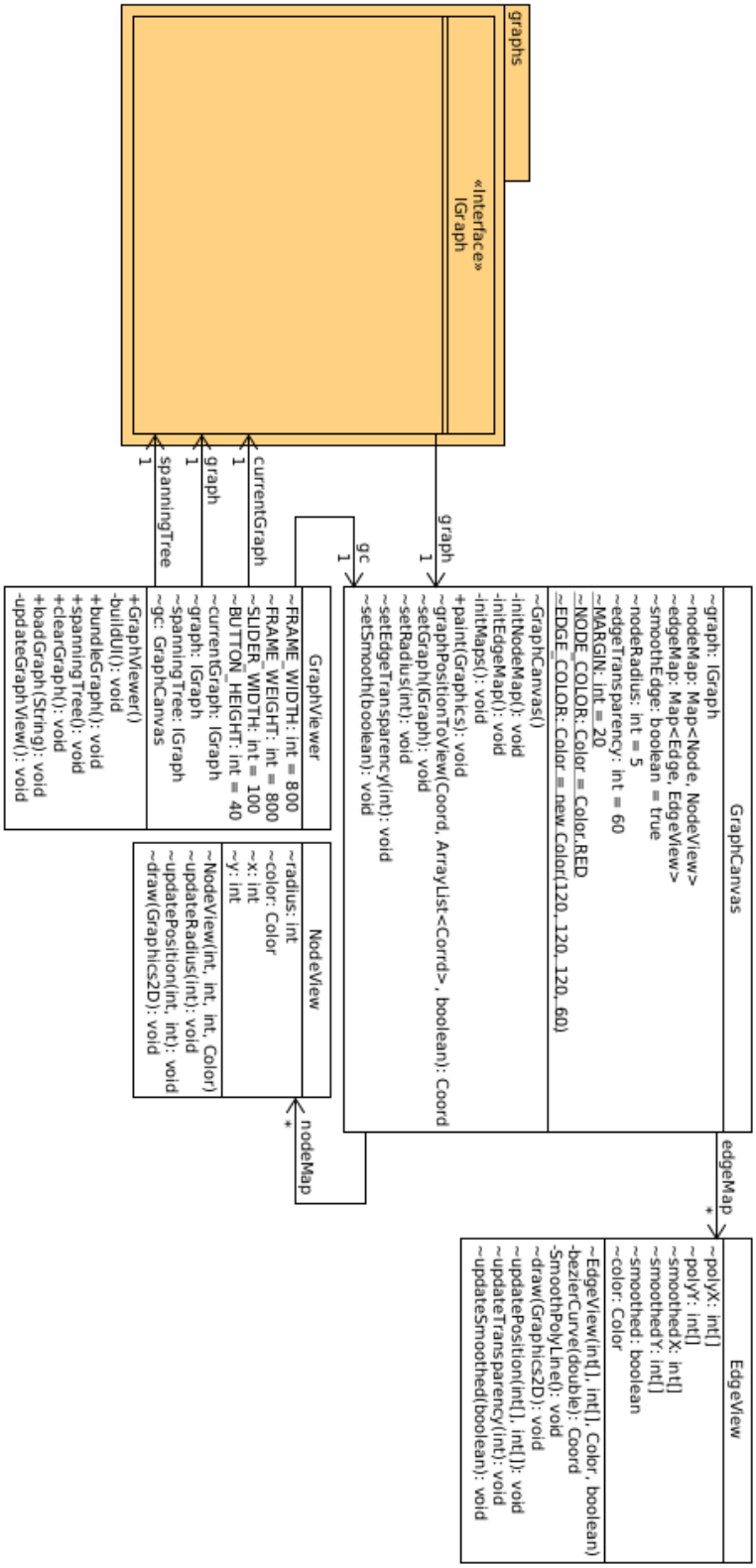


Fig 1.3 - Diagramme UML paquet graph final

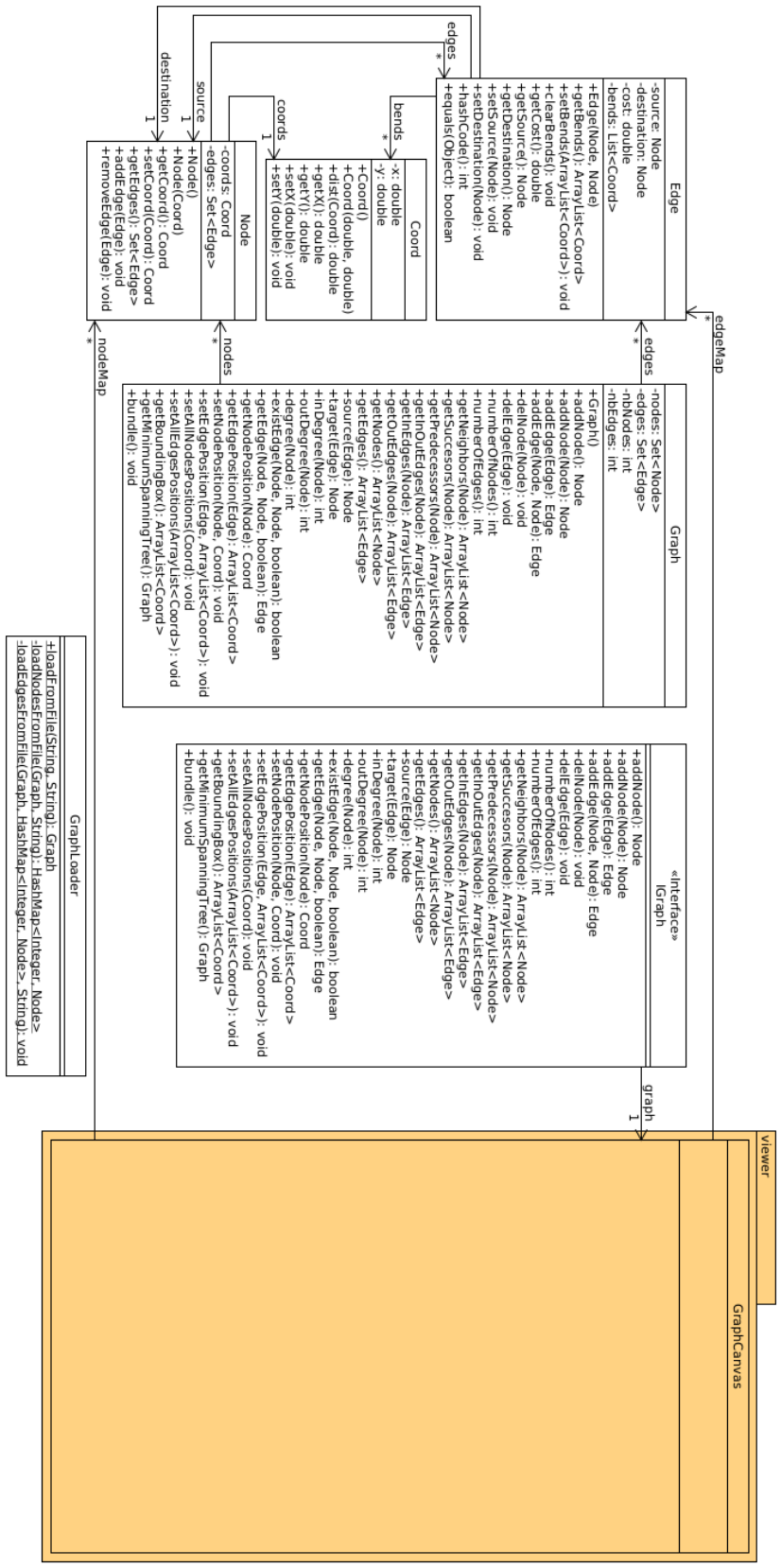
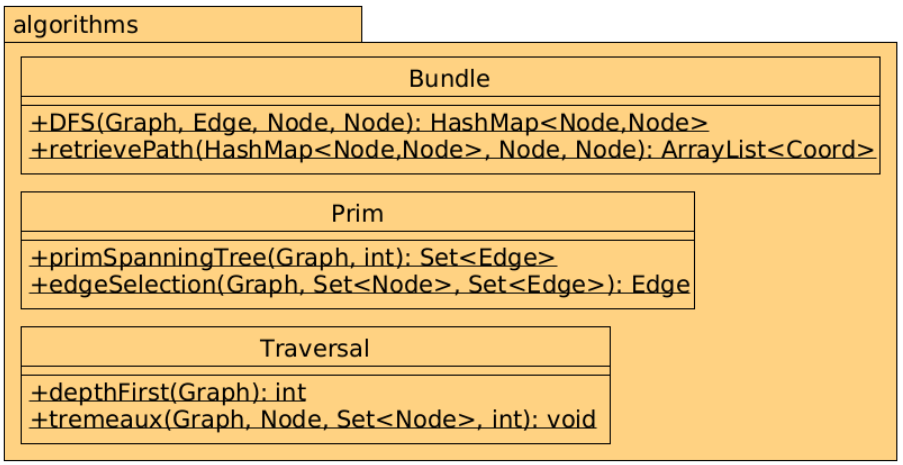


Fig 1.4 - Diagramme UML paquet algorithms final




## Approche

L'implémentation de la logique de graphe est regroupée dans un paquet `graphs`.

Pour réaliser l'implémentation de la classe Graph nous avons fait le choix d'utiliser une approche "haut niveau", c'est-à-dire en utilisant des collections plutôt qu'une matrice d'adjacence. Pour arriver à ces fins, plusieurs classes devaient être implémentées pour la cohérence des données.
La première classe est la classe `Node`. Cette classe représente un sommet d'un graphe. Cette classe possède plusieurs attributs tels que des coordonnées issus de la classe `Coord`, un ensemble d'arêtes représentant les arêtes connectées à ce sommet. Nous avons fait le choix d'implémenter deux constructeurs :

- Un constructeur dit "par défaut" sans argument, initialisant le sommet aux coordonnées (0;0) dans la fenêtre de visualisation.
- Un constructeur avec passage de Coordonnées en paramètres pour pouvoir définir sa position.

La deuxième classe est la classe `Edge` représentant une arête dans le graphe entre deux sommets. Cette classe est constituée d'attributs représentant le sommet de source et le sommet de destination ainsi qu'une liste de Coordonnées pour tracer le chemin dans la phase 3 du projet.
Enfin, pour la classe `Graph` nous avons choisi de créer un ensemble de sommets et un ensemble d'arêtes pour garder cette approche "haut niveau", nous avons également intégré des variables pour compter les arêtes et les sommets. Le reste des méthodes est l'implémentation des primitives de haut niveau du graphe.

Pour la partie implémentation d'algorithmes, nous avons décidé de créer un paquet `algorithms` pour séparer le code d'exploitation du code de définition de graphe.

Trois classes sont présentes dans ce paquet :

- La classe `Prim`, contenant les méthodes d'implémentation de l'algorithme de Prim, pour trouver un arbre couvrant de coût minimal.
- La classe `Bundle`, contenant la logique pour la récupération de liste de coordonnées pour créer les courbes des chemins absents après la construction de l'arbre de coût minimal.
- La classe `Traversal`, contenant l'algorithme de parcours en profondeur pour compter le nombre de composantes connexes dans un graphe

# Partie 2 - Test des algorithmes de graphe

## Algorithme de Prim

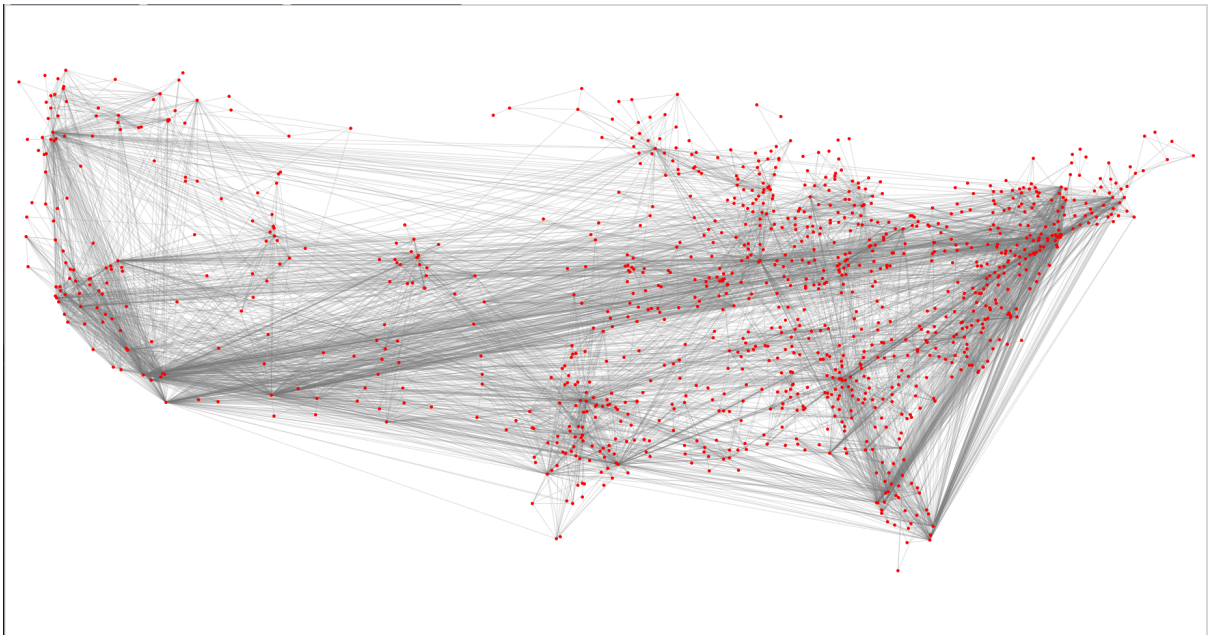
L'algorithme de Prim est un algorithme glouton pour déterminer l'arbre couvrant de coût minimal d'un graphe, c'est-à-dire un arbre passant par tous les sommets du graphe avec des arêtes de coût minimal. Cet algorithme permet par exemple de préparer un graphe pour un algorithme de parcours en éliminant toutes les arêtes les moins intéressantes.

Cet algorithme prend plusieurs paramètres :

- `Graph` g, représentant le graphe à traiter
- Un entier `n` représentant le nombre de sommets

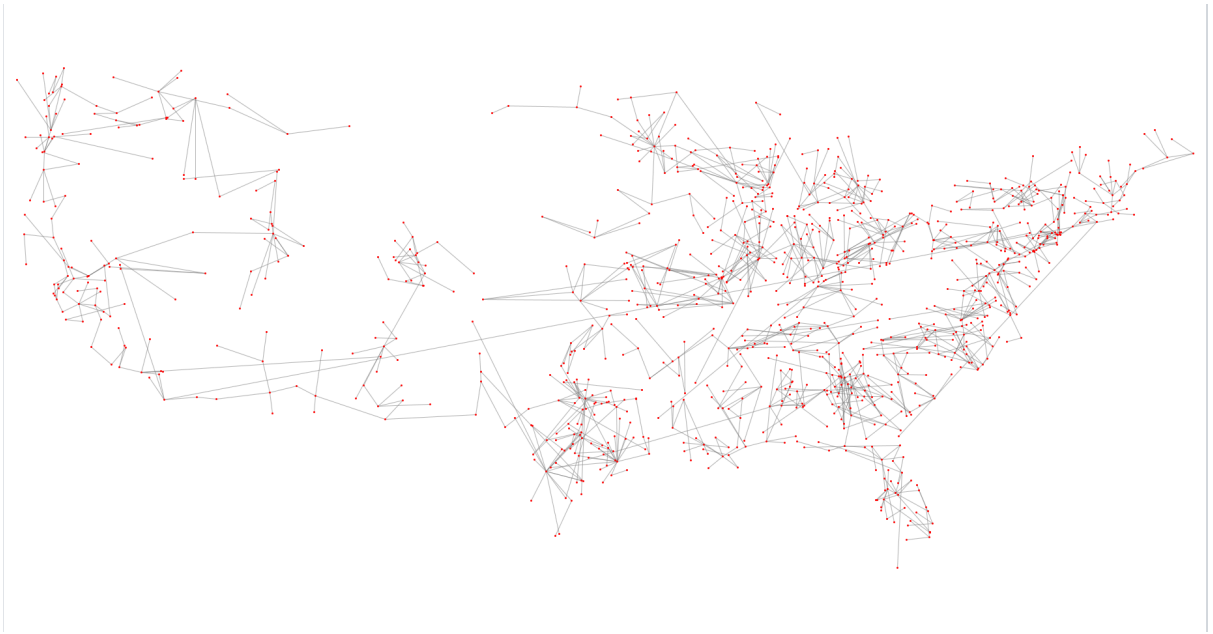
Le résultat attendu est un arbre contenant `n-1` arêtes pour `n` sommets.

Fig 2.1 - Graphe initial pour la démonstration de cet algorithme.



Dans ce cas, nous devrions obtenir un graphe avec une baisse considérable du nombre d'arêtes et donc de bruit visuel. Comme précisé précédemment, l'arbre devra passer par tous les sommets du graphe.

Fig 2.2 - Résultat obtenu après l'application de l'algorithme



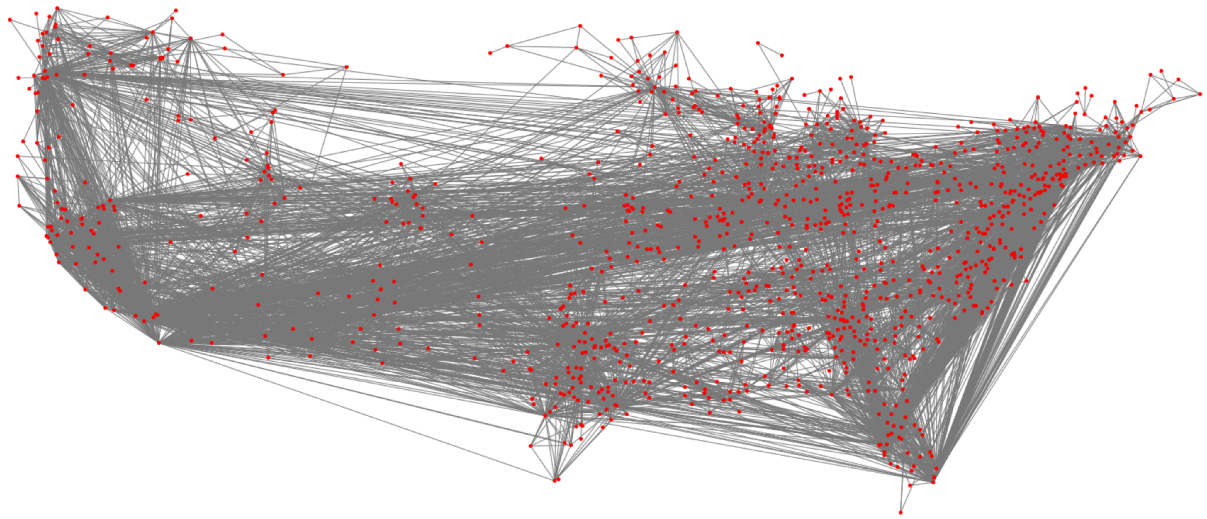
Voici l'arbre couvrant obtenu, il est conforme aux exigences définies précédemment. Le bruit visuel a considérablement diminué et seules les arêtes de coût minimal sont présentes.

La classe Prim dispose d'une classe de test `PrimTest` composée de fonctions de test pour ses 2 méthodes `primSpanningTree` et edgeSelection qui simulent la création d'un très petit arbres de 4 nodes et 5 edges et s'assurent que les résultats des méthodes sont bien ceux attendus. Par exemple, `testPrimSpanningTree` vérifie que le graph donné en paramètre n'est pas vide et que l'arbre de coût minimal et correcte alors que `testEdgeSelection` vérifie que le bon edge est choisi en fonction des paramètres passés. Il n'est pas nécessaire de tester si le graph est vide pour `edgeSelection` car celui-ci ne serait pas appelé par `primSpanningTree`.

## Edge Bundling

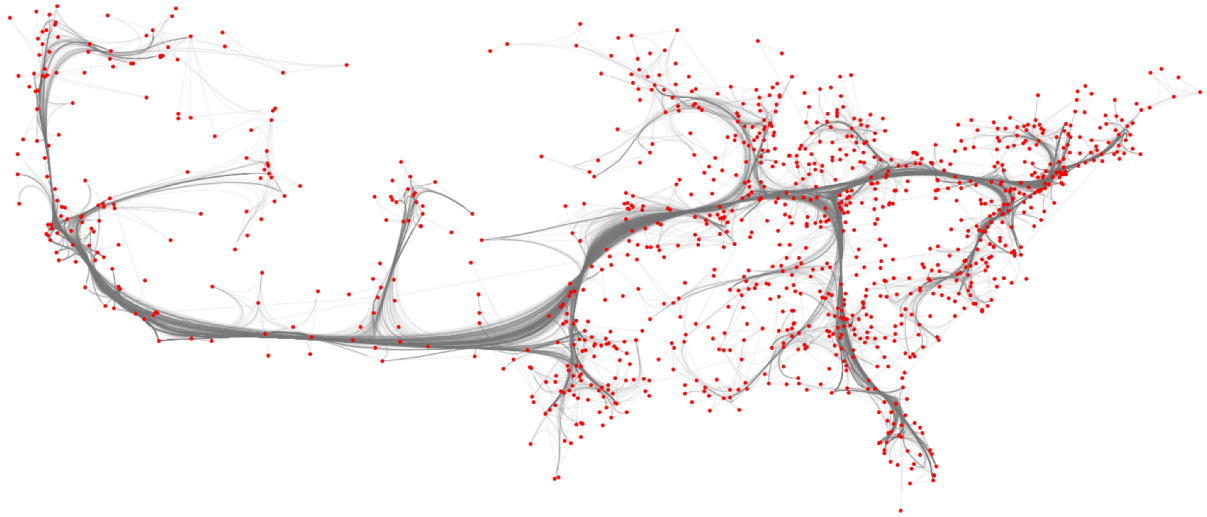
L'edge bundling est une technique de visualisation de graphes permettant de visualiser des tendances graphiques et créer des courbes pour visualiser ces tendances. La démarche consiste à utiliser un algorithme de construction d'arbre couvrant minimal afin de réduire les possibilités de chemin et trouver le plus rapide.
Tous les chemins du graphe initial seront représentés, pour les chemins supprimés à la suite de l'élagage par arbre couvrant, leurs chemins passeront donc par l'arbre couvrant et la liste de coordonnées des points du chemin sera déterminée pour construire la courbe associée à cette arête (en omettant les coordonnées de la source et de la destination).

Fig 2.3 - Graphe initial



Comme dans l'algorithme précédent, nous avons un graphe avec beaucoup de bruits visuels. On peut distinguer quelques tendances à vue d'œil mais une clarification est nécessaire.

Fig 2.4 - Graphe attendu



Dans cette nouvelle représentation, nous pouvons voir les tendances de façon très explicite, l'algorithme de Prim a été appliqué et les arêtes coupées sont maintenant représentées par des courbes convergentes vers l'arbre couvrant de coût minimal.