OUTIL D'OPTIMISATION DES FRÉQUENCES AFFECTÉES A DES ANTENNES DE TÉLÉPHONIE MOBILE

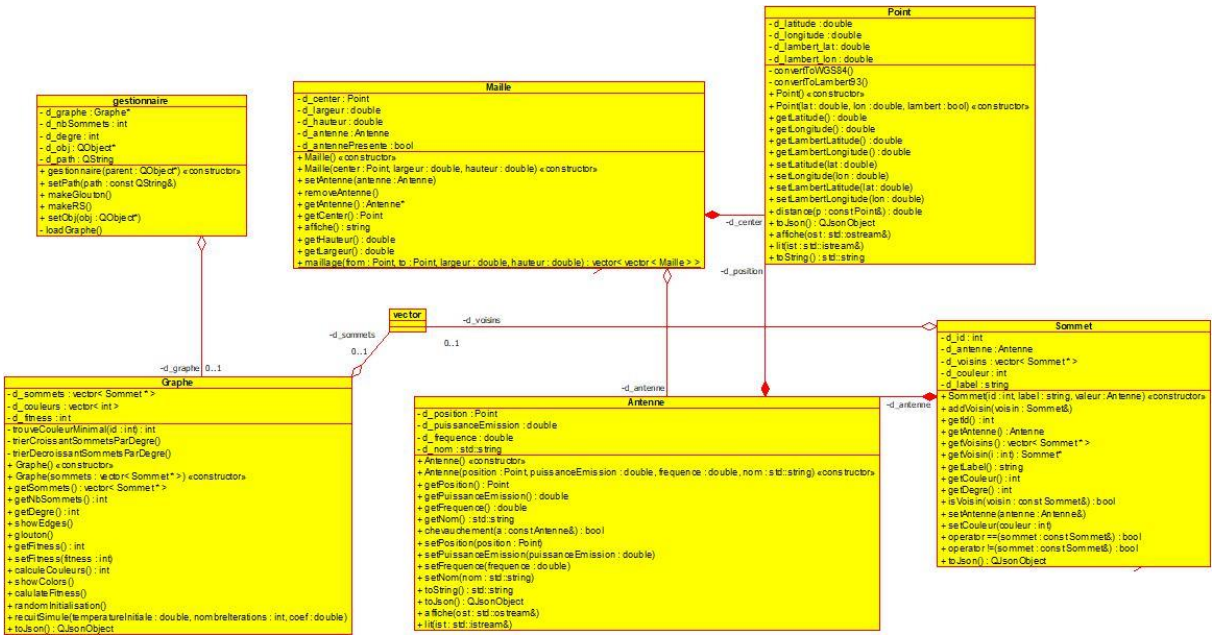
MANUEL UTILISATEUR

CHEF DE PROJET :
BOURAI MASSINSSA
DÉVELOPPEUR :
GEILLER VALENTIN
BEGOUD IMAD-EDDIN
MOUDACHE IDIR

ANNÉE UNIVERSITAIRE 2022/2023
MASTER INFORMATIQUE & MOBILITÉ - MIAGE

Table des matières

I. Architecture du code



La classe Point :

La classe "Point" représente un point géographique, qui peut être défini soit par sa **latitude** et sa **longitude**, soit par des coordonnées dans le système de projection de Lambert 93. Elle dispose de deux constructeurs :

- Un constructeur par défaut, qui initialise les coordonnées WGS84 (latitude et longitude) et Lambert à 0.
- Un constructeur qui prend en entrée une latitude et une longitude, ainsi qu'un booléen indiquant si ces coordonnées sont en WGS84 ou en Lambert. Selon la valeur de ce booléen, les coordonnées seront converties dans l'un ou l'autre des deux systèmes de projections (à l'aide des fonctions **convertToLamber93()** et **convertToWGS84()**).

La classe comprend également une méthode **distance()** permettant de calculer la distance entre deux points.

La classe Antenne :

La classe "Antenne" représente une antenne sur notre carte. Elle possède deux constructeurs

- Un constructeur par défaut.
- Un constructeur qui prend en arguments : un objet "**Point**" définissant sa position, deux doubles représentant la **fréquence** et la **puissance d'émission** de l'antenne, et une chaîne de caractères pour le **nom** de l'antenne.

La classe Antenne comprend également une méthode **chevauchement()** permettant de comparer la position et la puissance d'émission de deux antennes pour déterminer si leur périmètre d'émission se chevauchent.

Notez que chaque classe possède également plusieurs fonctions membres de getters et setters pour chacune de ses variables membres, ainsi que des surcharges pour les opérateurs de lecture et d'écriture '<<' et '>>', et des fonctions appelées **affiche()** et **lit()** pour afficher l'objet dans un flux de sortie, mais aussi une fonction toJson() permettant de convertir l'objet de la classe en objet JSON et une méthode toString() qui convertit l'objet de la classe en chaîne de caractères.

La classe Sommet :

La classe "Sommet" représente un sommet de notre graphe. Elle possède les attributs suivants :

- **d_id** : un entier unique qui identifie le sommet.
- **d_label** : une chaine de caractères qui étiquette le sommet.
- **d_antenne** : un objet de type « **Antenne** » qui représente l'antenne associé au sommet.
- **d_couleur** : un entier qui représente la couleur du sommet.
- **d_voisins** : un tableau de pointeurs sur des objets de type « **Sommet** » qui représente les sommets voisins de notre sommet courant.

La classe Sommet possède également plusieurs méthodes :

- Constructeur **Sommet()** : initialise les attributs **d_id**, **d_label**, **d_antenne**.
- **addVoisin()** : ajoute un sommet dans le tableau **d_voisins**.
- **getVoisin()** : retourne un pointeur sur un sommet voisin à l'indice donné à condition que l'indice soit valide.
- **getDegre()** : retourne le degré du sommet, c'est-à-dire son nombre de voisins.

Comme pour les classes Point et Antenne, la classe Sommet possède plusieurs Getter() et Setter() et une méthode **toJson()**, elle re définit également les opérateurs == et != qui permettent de comparer deux sommets selon leur identifiant, et aussi l'opérateur de flux de sortie '<<' qui permet d'afficher un sommet.

La classe Maille :

La classe "Maille" représente une région géographique carrée définie par un centre, une hauteur et une largeur. Elle possède les attributs suivants :

- **d_center** : un objet de type "**Point**" représentant le centre géographique de la maille.
- **d_largeur** : un double représentant la largeur de la maille.
- **d_hauteur** : un double représentant la hauteur de la maille.
- **d_antenne** : un objet de type "**Antenne**" définissant une antenne présente géographiquement sur la maille courante.
- **d_antennePresente** : un booléen indiquant si une antenne est présente sur la maille ou non.

La classe "Maille" possède les méthodes suivantes :

- Deux constructeurs, un par défaut et l'autre qui initialise seulement **d_center**, **d_largeur** et **d_hauteur**.
- Des fonctions membres de getters et setters pour certains de ses attributs.
- **removeAntenne()** : permet de retirer l'antenne associée à la maille.
- **affiche()** : retourne une chaîne de caractères indiquant si une antenne est présente dans la maille ou non.

- une methode static **maillage()** : crée un maillage géographique en créant une grille de mailles carrées à partir de deux points géographiques et de deux dimensions servant à définir la largeur et la hauteur de la maille.

La classe Graphe :

La classe "Graphe" représente un graphe, c'est-à-dire une structure de données constituée de sommets reliés par des arêtes. Elle possède les attributs suivants :

- **d_sommets** : un tableau de pointeurs de sommet qui représente les sommets du graphe.
- **d_fitness** : un entier qui indique le nombre de couleurs différentes utilisées pour la coloration du graphe, cela donne un score.
- **d_couleurs** : un tableau d'entiers utilisé pour stocker les couleurs disponibles pour colorer les sommets du graphe.

La classe "Graphe" possède les méthodes suivantes :

- Deux constructeurs, un par défaut qui initialise **d_fitness** à -1, et un autre qui prend en paramètre un tableau de pointeurs de sommet qui servira à initialiser **d_sommets**. Ce deuxième initialisera aussi **d_fitness** à -1, et le tableau **d_couleurs** sera quant à lui initialisé de 1 à la taille de ce dernier.
- **trouveCouleurMinimal()** : prend en paramètre un entier id et retourne la couleur minimale qui n'est utilisée par aucun des voisins du sommet identifié par id.
- **trierDecroissantSommetParDegre()** : trie les sommets du graphe de manière décroissante selon leur degré.
- **trierCroissantSommetParDegre()** : trie les sommets du graphe de manière croissante selon leur degré.
- **getSommets()** : retourne le tableau de sommets du graphe.
- **getNbSommets()** : retourne le nombre de sommets du graphe.
- **geDegre()** : retourne le degré maximal des sommets du graphe.
- **showEdges()** : parcourt tous les sommets du graphe et affiche leurs labels ainsi que ceux de leurs voisins.
- **showColors()** : parcourt tous les sommets du graphe et affiche leurs labels ainsi que leurs couleurs.
- **calculeCouleurs()** : parcourt tous les sommets du graphe et retourne le nombre de couleurs différentes utilisées.
- **getFitness()** : retourne la valeur de **d_fitness**. (si elle est égale à -1, elle est d'abord calculée avant d'être renvoyée).
- **setFitness()** : met à jour la valeur de **d_fitness**.
- **randomInitialisation()** : initialise aléatoirement la couleur de chaque sommet du graphe.
- **calculateFitness()** : calcule la valeur de **d_fitness** pour le graphe courant.
- **toJson()** : retourne une chaîne de caractères au format JSON représentant le graphe courant.
- **glouton()** : utilise l'algorithme glouton pour colorer les sommets du graphe de manière que deux sommets adjacents ne soient pas de la même couleur. Et cela en :
    1. Triant les sommets du graphe par ordre décroissant de degré.
    2. Pour chaque sommet du graphe, utilisez la fonction trouveCouleurMinimale() de sorte à lui affecter une couleur différente de celle de ces voisins.
    3. Répétez 2 jusqu'à ce que tous les sommets aient été coloriés.

- **recuitSimule()** : cette méthode est basée sur l'algorithme "recuit simulé" qui s'inspire du comportement du métal qui est chauffé à haute température et refroidi lentement pour atteindre sa forme finale.
    1. Elle commence par générer une solution initiale au hasard, et utilise ensuite une température initiale qui est progressivement refroidie à chaque itération.
    2. À chaque itération, l'algorithme génère une nouvelle solution en modifiant au hasard la couleur d'un sommet.
    3. Il calcule alors la différence de fitness entre la nouvelle solution et la solution actuelle. Si la nouvelle solution est meilleure, elle sera retenue, sinon elle est acceptée avec une probabilité qui dépend de la température et de la différence de fitness. Plus la température est élevée, plus il est probable d'accepter une solution moins bonne.
    4. Répété 2 et 3 jusqu'à ce que le nombre d'itérations précisé ait été atteint.
    5. Enfin, la fonction retourne la meilleure solution trouvée jusqu'ici.

La classe Gestionnaire :

La classe "Gestionnaire" fait le lien entre le back end de notre application écrit en C++ et le front end écrit en QML. Elle possède les attributs suivants :

- **d_graphe** : un pointeur sur un objet de type « **Graphe** ».
- **d_nbSommet** : un entier qui contient le nombre de sommets du graphe.
- **d_degre** : un entier qui contient le degré du graphe.
- **d_path** : une chaine de caractères (QString) qui contient le chemin à partir duquel le graphe a été chargé.
- **d_obj** : un objet de type « QObject » qui sert à invoquer les méthodes en QML.

La classe "Gestionnaire" possède les méthodes suivantes :

- Le constructeur qui prend en argument un objet QObject "parent", et qui initialise d_graphe, ainsi que d_nbSommets et d_degre en utilisant les méthodes getNbSommets() et getDegre() de l'objet d_graphe.
- **setPath()** : prend en argument une chaîne de caractères (QString) ''path''. Si d_path est égale à path, la méthode ne fait rien, sinon elle affecte path à d_path en enlevant les 8 premiers caractères de ce dernier et appelle la méthode **loadGraphe()**.
- **setObj()** : prend en argument un objet QObject 'obj', et l'affecte à l'attribut d_obj s'il est différent de celui-ci.
- **makeGlouton()** : utilise l'algorithme glouton pour colorer les sommets du graphe et met à jour la valeur de **d_fitness**.
- **makeRS()** : utilise l'algorithme de recuit simulé pour colorer les sommets du graphe et met à jour la valeur de **d_fitness**.
- **loadGraphe()** : charge le graphe défini par le chemin **d_path** et met à jour **d_nbSommets** et **d_degre**. Puis invoque la méthode « **loadGrapheQML()** » à travers l'objet d_obj

II. Prérequis

Avant de pouvoir utiliser l'application, il est nécessaire de prétraiter les données des antennes et de les mettre dans un format compatible. Pour ce faire, vous devez créer un fichier CSV contenant les informations suivantes pour chaque antenne : le nom de l'antenne, sa position en coordonnées X et Y dans la projection Lambert 93, ainsi que sa puissance d'émission en mètres.

Exemple :

Nom	x	y	rayon
Antenne1	1021203	6750516	875
Antenne2	1025505	6744661	1077
Antenne3	1024830	6744347	543
Antenne4	1022326	6749854	478
Antenne5	1023360	6746484	611

Il est obligatoire d'inclure une ligne d'entête avec les titres suivants : Nom, x, y, rayon. Il n'y a pas de limite au nombre d'antennes que vous pouvez inclure dans le fichier (c'est-à-dire, au nombre de lignes de données).

III. Outils de développement

Pour réaliser ce projet, nous avons utilisé une suite de logiciels et de technologies qui nous ont permis d'atteindre nos objectifs. Grâce à ces outils, nous avons pu développer une solution efficace et fonctionnelle.

1. Les logiciels de développement

   - **Visual Studio** : C'est un IDE très puissant qui permet de concevoir des applications de bureau, des applications mobiles ainsi que des sites web avec les langages tel que l'ASP.NET, le Visual Basic, le C#, le C et le C++.

   - **Qt Creator** : C'est un IDE qui permet d'écrire du code C++ avec le Framework Qt. C'est entre autres un éditeur qui propose un débogueur et des outils de gestion de versions intégré ainsi qu'une myriade de fonctionnalités qui permet de développer des interfaces modernes.

2. Les technologies utilisées

- **C++** : un langage de programmation compilé permettant la programmation sous de multiples paradigmes, dont la programmation procédurale, la programmation orientée objet et la programmation générique. Ses bonnes performances, et sa compatibilité avec le C en font un des langages de programmation les plus utilisés dans les applications où la performance est critique. (“C++ — Wikipédia”)

- **QML** : un langage de balisage d'interface utilisateur. C'est un langage déclaratif pour la conception d'applications centrées sur l'interface utilisateur. Il est associé à Qt Quick, le kit de création d'UI développé à l'origine par Nokia dans le Framework Qt. (“QML — Wikipédia”)

- **Open Street Map** : Un projet collaboratif de cartographie en ligne qui vise à constituer une base de données géographiques libre du monde, en utilisant le système GPS et d'autres données libres. (“OpenStreetMap — Wikipédia”)

3. Les logiciels de gestion de projets/groupes et de code.

- **GanttProject** : un logiciel de gestion de projet qui permet de créer le planning des projets. C’est celui utilisé pour la séparation des tâches de ce projet.

- **Git & GitHub** : Git est un logiciel de gestion de versions décentralisé. GitHub est un logiciel de contrôle de version. Il sert tout d’abord à effectuer des dépôts distants d’un projet local. Il permet de gérer l’évolution du code du projet.

- **Discord** : un logiciel propriétaire gratuit qui permet d’échanger des messages instantanés. Essentiellement utilisé pour gérer le partage d’information et faciliter les échanges entre les collaborateurs du projet.

IV. Procédure de lancement de l’application

Pour lancer l'application, il suffit de lancer l'exécutable "ProjetReseau.exe" qui se trouve dans le dossier "Exécutable". (Si l'application se lance, vous pouvez ignorer la suite de cette instruction qui explique comment créer un exécutable.)

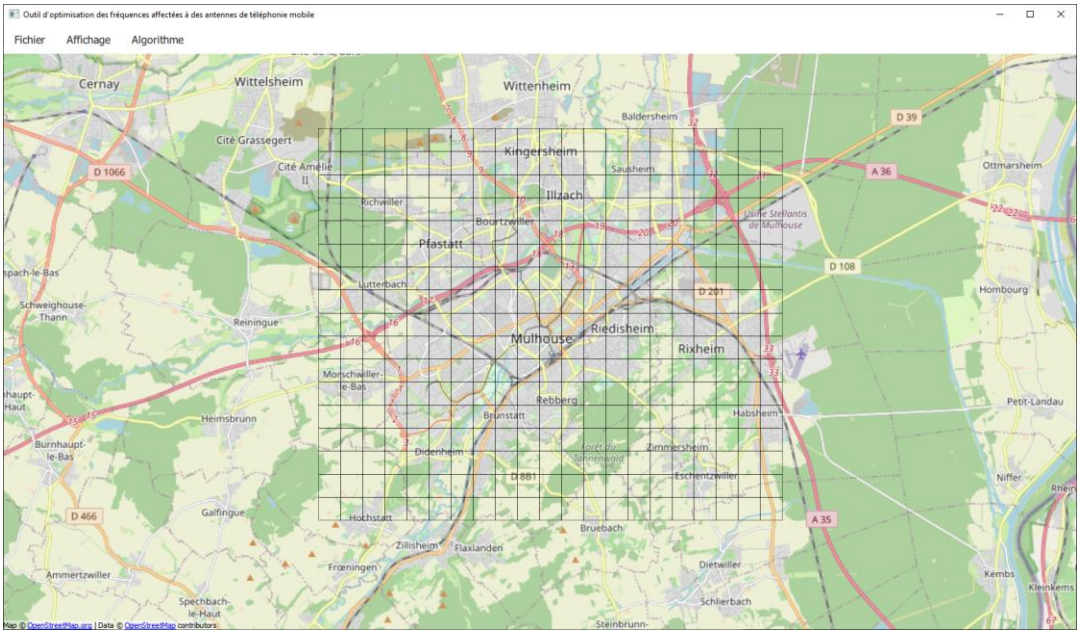
Si l'application ne démarre pas, il faut installer une version de Qt avec une version de Mingw (5.12.10) et compiler les sources. Pour ce faire, il faut supprimer le fichier .pro.user qui se trouve dans le dossier du code source et lancer le fichier .pro. Une fenêtre de configuration va alors s'ouvrir et il suffit de sélectionner Mingw dans la liste des compilateurs disponibles.

Une fois que l'application est ouverte, vous avez deux options pour lancer le programme :

1. Cliquez sur le triangle vert en bas à gauche de l'écran pour lancer le programme directement.
2. Créez un exécutable en suivant les étapes suivantes :
    a. Sélectionnez le mode "release" et compilez l'application.
    b. Accédez au dossier de build et récupérez le fichier .exe généré.
    c. Créez un nouveau dossier et placez l'exécutable à l'intérieur.
    d. Ouvrez votre terminal avec Qt (Qt 5.12.10 (MinGW 7.3.0 64-bit)) et exécutez la commande appropriée.

windeployqt –release –qmldir chemin/vers/les/soures ProjetReseau.exe ou si vous avez linux linuxdeployqt (https://github.com/probonopd/linuxdeployqt) ou si vous avez un mac macdeployqt

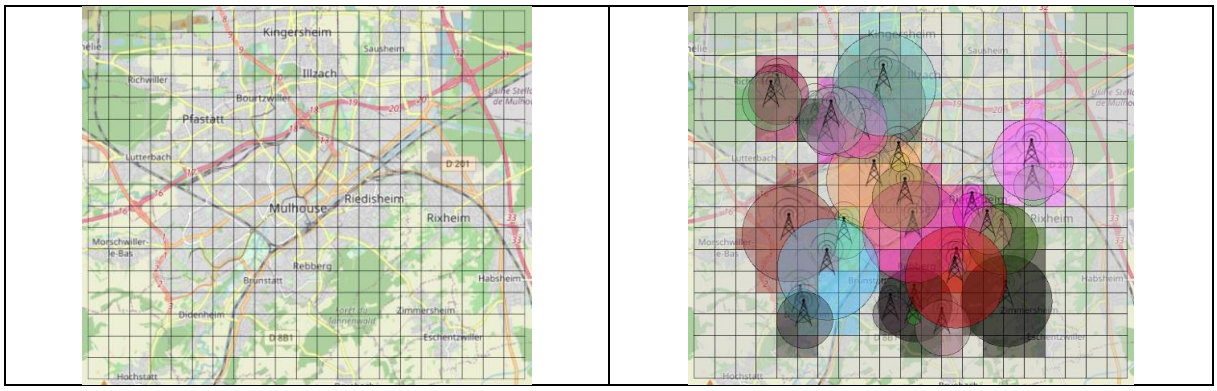
Une fois l'application lancé, vous arrivez sur cette interface :



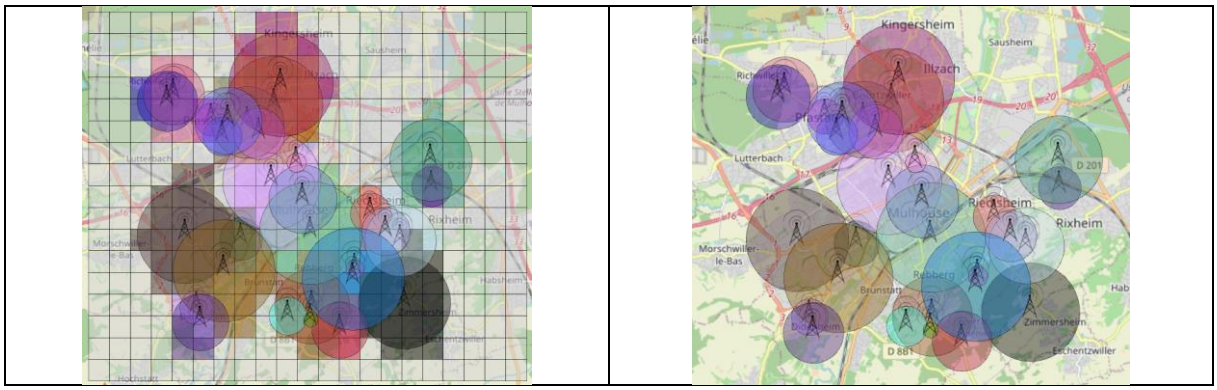
V. Les fonctionnalités de l'application

Dans l'application :

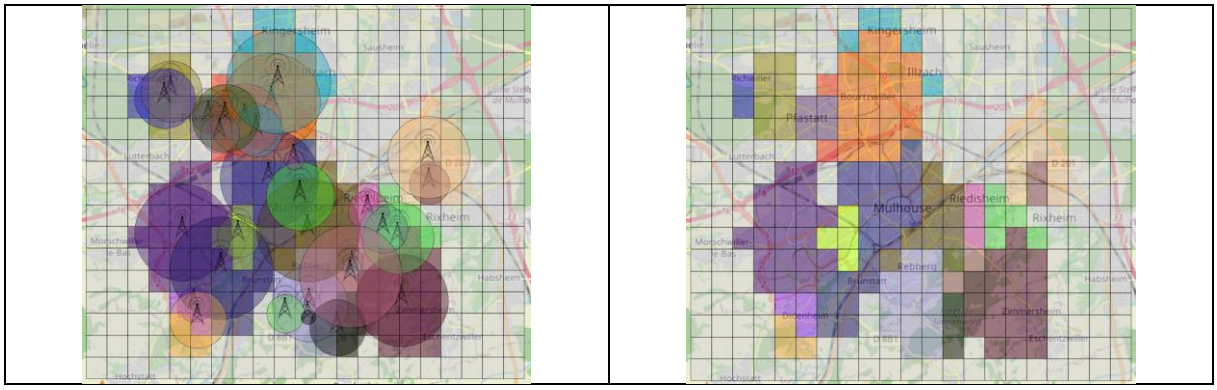
- Nous pouvons importer un fichier au format CSV (avec la bonne structure, voir II. Prérequis). L'application charge alors les antennes et affiche leur portée à leur emplacement sur la carte.



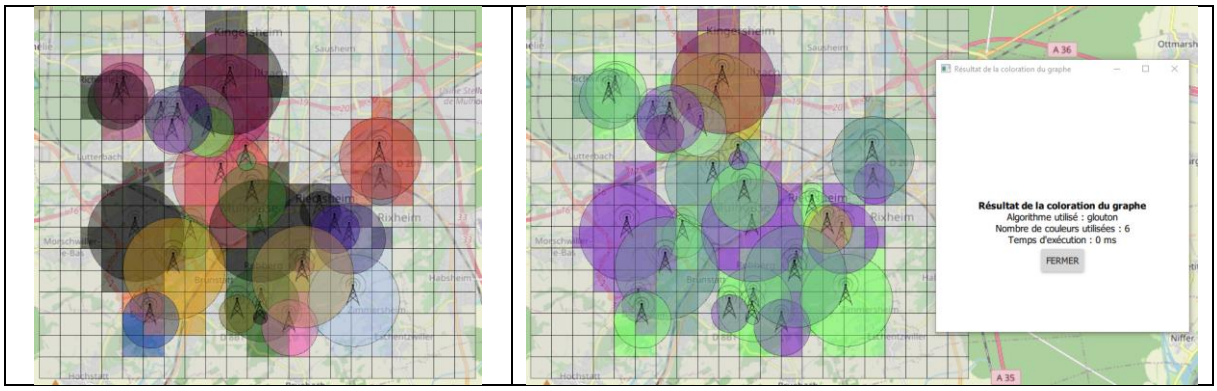
- Nous pouvons masquer les mailles pour avoir un meilleur visuel sur les antennes (le contraire est aussi possible, c'est-à-dire afficher les mailles).



- Nous pouvons masquer les antennes pour vérifier si elles couvrent toutes les zones de notre environnement et voir quelle antenne couvre quelle portion de l'environnement grâce aux couleurs (le contraire est aussi possible).



- Nous pouvons exécuter des algorithmes de coloration de graphes et afficher le résultat.



- Nous pouvons aussi naviguer sur la carte en utilisant les fonctions de zoom et de déplacement (en maintenant le clic gauche enfoncé).