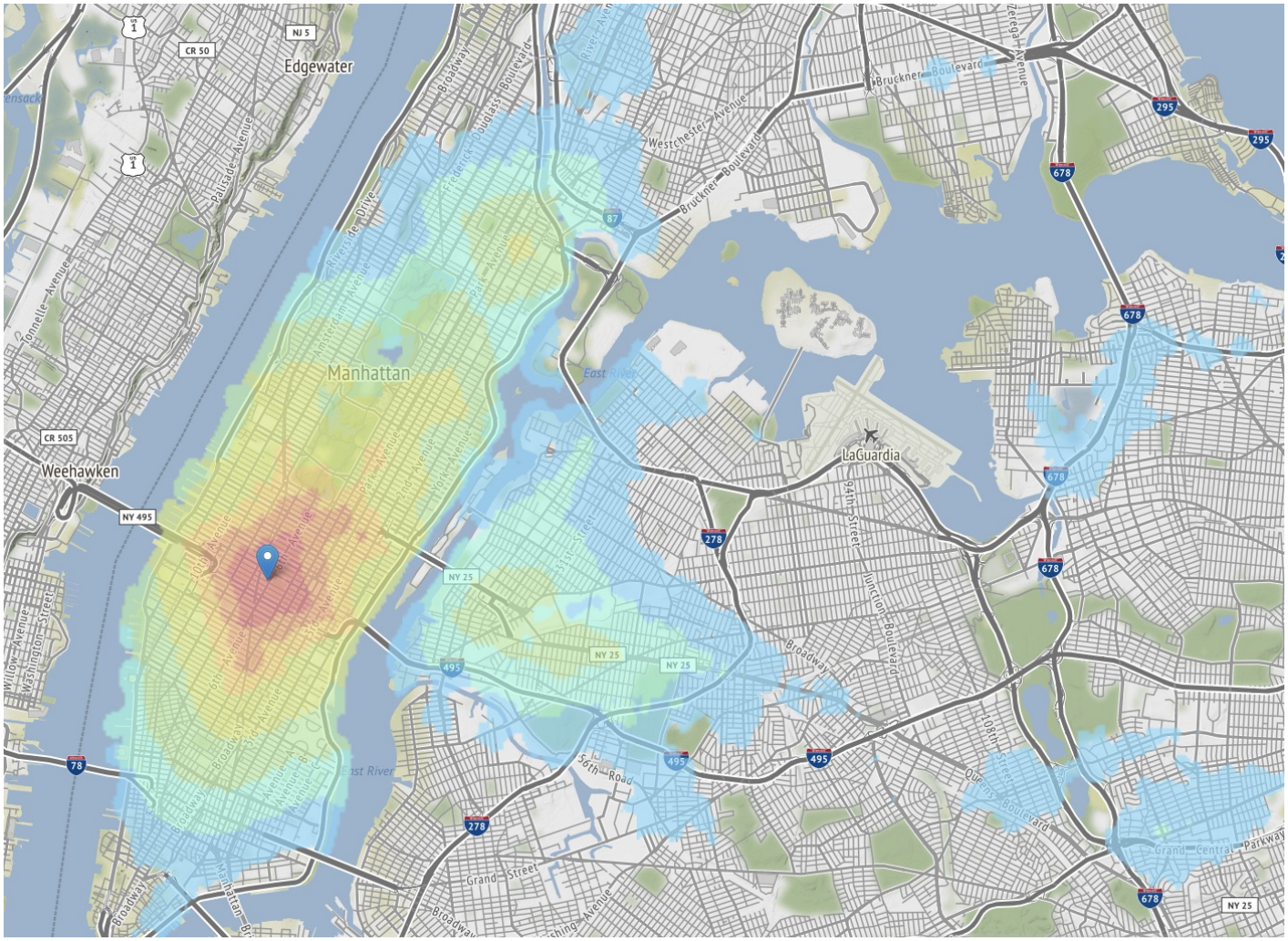
Calcul d'isochrones

Objectif

Lors de l'implantation de nouveaux services (transport, aménagement, magasin), on peut être amené à réaliser des études d'accessibilité. Ces études visent à établir les parties d'un graphe accessibles à partir d'un point donné tout en se fixant une limite dans le parcours de ce graphe. Par exemple : évaluer dans un graphe routier les zones accessibles en 10 minutes, 20 minutes à partir d'un nœud donné. La zone accessible est aussi appelé isochrone. L'exemple ci-dessous montrent les isochrones à partir de Time Square en transport en commun (par pas de 10 minutes).



Votre mission est d'implémenter un algorithme capable de calculer un isochrone sur un graphe orienté pondéré.

Modélisation orientée objet d'un graphe

Dans un premier temps, vous devez définir les classes qui permettront de représenter un graphe orienté pondéré. On veut pouvoir manipuler des graphes qui sont codés soit sous forme de matrices d'adjacence soit sous forme de listes d'adjacence. Par contre, il faut qu'un algorithme puisse travailler sur l'une ou l'autre version de manière transparente.

Indication : dans ce projet, les sommets peuvent être représentés par des entiers de 0 à $n-1$ avec n le nombre de sommets du graphe.

On veut pouvoir disposer de deux constructeurs. Le premier est un constructeur sans paramètre qui crée un graphe vide. Le second constructeur dispose d'un paramètre de type chaîne de caractères qui est le chemin vers un fichier texte contenant une description du graphe.

On veut pouvoir effectuer *a minima* les opérations suivantes sur le graphe :

- redimensionner le nombre de sommets (ajout / retrait, dans le cas du retrait, on retire les sommets de plus grand indice);

- récupérer le nombre de sommets ;
- récupérer le nombre d'arcs ;
- ajouter un arc d'un sommet i vers un sommet j avec un poids p ;
- retirer un arc d'un sommet i vers un sommet j ;
- tester si un arc (i, j) existe dans le graphe ;
- récupérer le poids d'un arc (i, j) ;
- récupérer les successeurs d'un sommet ;
- récupérer les prédécesseurs d'un sommet ;
- récupérer les voisins d'un sommet ;
- sauvegardez le graphe dans un fichier texte dont on fournit le chemin.

Production : un diagramme de classes UML.

Format des fichiers textes

On va être amené à lire et écrire des graphes dans un fichier texte. Le format du fichier est le suivant :

```
c grid graph x=512 lX=0 lY=8192 y=16 lY=0 hY=8192 seed=1913762381
p sp 8192 31712
a 3069 6629 2551
a 3069 4921 3436
a 6629 3069 2928
a 6629 6331 3640
...
```

- La première ligne est un commentaire donnant des informations sur l'instance (vous pouvez mettre ce que vous voulez après le caractère 'c').
- La seconde ligne contient les informations (en plus de 'p' et 'sp') : le nombre de sommets, le nombre d'arcs.
- Ensuite, il y a une ligne par arc commençant par 'a' avec l'origine de l'arc, la destination de l'arc et le poids de l'arc. Tous les poids sont des entiers.

Différents graphes vous sont fournis ici. Attention, certains de ces graphes sont de très grandes tailles.

Remarque : il peut y avoir plusieurs arcs reliant deux sommets. Dans ce cas, vous ne garderez que le premier arc lu.

Indication : pour tester vos algorithmes, vous êtes encouragés à créer vos propres fichiers contenant des graphes de petites tailles. Vous pouvez par exemple utiliser les graphes vus en cours de Graphes.

Lecture / écriture d'un fichier texte

Module `fs` de Node

Pour lire un fichier, le plus simple est d'utiliser le module `fs` de Node. Ce module est installé par défaut. Les fonctions dont vous aurez besoin sont importées par l'instruction :

```
import { readFileSync, createWriteStream } from "fs";
```

Lecture d'un fichier

L'instruction suivante lit le contenu d'un fichier et stocke chaque ligne comme une chaîne de caractères dans un tableau de chaînes de caractères :

```
let lignes = readFileSync("~/Instances/Random4-n/Random4-n.21.0.gr", "utf8").split("\n");
```

Le premier argument de la fonction `readFileSync` est une chaîne de caractères contenant le chemin jusqu'au fichier que l'on souhaite lire. Le second argument est l'encodage du fichier.

La fonction `readFileSync` retourne le contenu du fichier sous la forme d'une chaîne de caractères. La méthode `split` des chaînes de caractères permet de découper une chaîne de caractères en plusieurs chaînes en se basant sur un caractère comme délimiteur, ici le retour à la ligne.

Indication : si vous voulez découper une chaîne par rapport aux espaces par exemple, vous pouvez le faire avec la méthode `split`. Après les instructions :

```
let chaîne = "a 1 3";
let info = chaîne.split(" ");
```

la variable **info** sera un tableau de chaînes de caractères égal à ['a' , '1' , '3'].

Ecriture dans un fichier
Pour écrire dans un fichier, on commence par ouvrir un flux vers un fichier avec la fonction **createWriteStream** dont l'argument est une chaîne de caractères contenant le chemin vers le fichier :

```
let output = createWriteStream("IOFiles/output.txt") ;
```

Attention : cela efface le contenu du fichier s'il existe déjà.
Ensuite, la méthode **write** des flux d'écriture permet d'écrire dedans :

```
output.write("c random graph n=2097152 m=8388608 lo=0 hi=2097152 seed=1292225576\n") ;
output.write("p sp 2097152 8388608\n") ;
output.write("a 1182391 1259944 1442304\n") ;
output.write("a 1259944 1647105 225203\n") ;
```

Lorsque l'écriture dans le fichier est terminée, il faut fermer le flux :

```
output.end() ;
```

Implémentation en TypeScript
Dans cette partie, il faut implémenter les classes que vous avez définies ci-dessus. Il faut qu'il y ait une concordance entre votre diagramme de classes et votre code. Certaines actions ne sont pas possibles. Il vous faudra décider par exemple de ce que vous voulez faire si l'utilisateur ajoute un arc entre deux sommets et que l'un des sommets n'existe pas.

Production : le code des classes + un court rapport expliquant comment vous avez définies les actions comme l'accès à des sommets qui n'existent pas.

Test de votre code

Vous implémenterez l'algorithme de Dijkstra vu en cours. Votre fonction devra prendre en paramètre un graphe et un sommet initial et retourner un objet de type **Resultat.** On devra pouvoir retrouver *a minima* via cet objet la longueur du plus court chemin vers un sommet et le prédécesseur d'un sommet sur le plus court chemin.
Vous devrez mettre en place des tests unitaires permettant de vérifier que votre algorithme fonctionne basés sur les exercices fait en TD.
Production : le diagramme UML de la classe **Resultat** + le code de la fonction Dijkstra + fichier de tests unitaires + les fichiers / fonctions nécessaires au test.

Consignes

- Le travail se fait en binôme.
- Le travail doit être rendu via un dépôt git privé (vous n'y mettrez pas les fichiers de graphes)
- Votre dépôt ne doit pas être créé à la fin avec un unique upload des fichiers.
- Vous veillerez à mettre des textes liés aux *commit* pertinents.
- Le dépôt git devra faire apparaître différentes étapes du travail (utilisation de tag) et une vraie collaboration.
- Vous ajouterez à votre dépôt vos enseignants de développement objet.
- La lisibilité du code (formatage des fichiers, découpages, choix des identifiants…) sera prise en compte.
- Votre code doit lever que des exceptions que **vous avez prévu** avec un message d'erreur informatif pour l'utilisateur.

Important : Ce travail va ensuite être continué durant la SAE. Vous veillerez donc à concevoir votre modèle et votre code pour qu'ils soient facilement adaptable. Vous devriez être capable d'implémenter les algorithmes vus en cours de graphes avec vos structures de données.