

Manipulation de données et algorithme de pathfinding multi-modal - Nerumir

Sommaire

- Introduction..... p. 1
- Collecte des données initiales..... p. 1
- Traitement et structuration des données brutes..... p. 3
- Stockage des données structurées..... p. 10
- Algorithme de `pathfinding` initial..... p. 15
- Algorithme de `pathfinding` pour les transports en commun..... p. 20
 - Création de l'algorithme..... p. 20
 - Optimisation de l'algorithme..... p. 25
 - Optimisation d'utilisation..... p. 25
 - Optimisation de conception..... p. 26
- Génération de notre `dataset` p. 27
- Nettoyage des données..... p. 29
- Visualisation des données..... p. 31
- Conclusion..... p. 35

Introduction

L'objet de ce projet sera d'analyser un très grand nombre de données autour des transports en commun. J'ai de travailler sur les arrêts de bus avec comme ambition :

- Récolter le plus d'informations possibles sur les arrêts de bus existants en France.
- Analyser, structurer et stocker toutes données dans une base de données `NoSQL` .
- Créer un algorithme de `pathfinding` afin de faire de la génération de données à partir des données stockées.
- Visualiser et interpréter les données collectées et les données générées.

Collecte des données initiales

Dans un premier temps, il faut collecter beaucoup de données. Mon but étant d'obtenir des informations sur les arrêts de bus en France, j'imagine que cela concerne un nombre assez conséquent d'arrêts. Pour la ressource, mon choix s'est porté sur le site du gouvernement :

<https://transport.data.gouv.fr> . Après une rapide consultation du site internet, j'en ai déduis que je voulais récupérer toutes les données `GeoJSON` des ressources présentent lors d'une recherche avec le mot clé "agrégat". Voici mon scraper :

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin
from urllib.parse import urlparse
import os
import zipfile

# Fonction qui permet de vérifier si un fichier est un fichier zip (pas besoin de l'extension)
def is_zip_file(file_path):
    try:
        with zipfile.ZipFile(file_path) as _:
            return True
    except zipfile.BadZipFile:
        return False

# Créer le dossier ou l'on va tout stocker
if not os.path.exists('data'):
```

```

os.makedirs('data')

# Récupération des URL de pages où se trouvent les liens de téléchargement
url = 'https://transport.data.gouv.fr/datasets?q=agr%C3%A9gat'
response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')

links = [title.select_one('a')['href'] for title in soup.find_all(class_='dataset__title')]
absolute_links = [urljoin(response.url, uri) for uri in links]

# On va sur chaque lien pour lister et récupérer les ressources afin de les télécharger et les décompresser dans des dossiers
for link in absolute_links:
    print("Collecting on : "+urlparse(link).path)
    # Créer le dossier
    nom_dossier = link.split('/')[-1]
    if not os.path.exists('data/'+nom_dossier):
        os.makedirs('data/'+nom_dossier)
    page_dir = 'data/'+nom_dossier

    # Récupération des ressources
    response = requests.get(link)
    soup = BeautifulSoup(response.content, 'html.parser')
    resources = soup.select('.ressources-list > .resource')

    # On boucle sur les ressources pour les télécharger et les décompresser
    for resource in resources:
        # Création du dossier
        nom_dossier = resource.select_one('h4').text
        if not os.path.exists(page_dir+'/'+nom_dossier):
            os.makedirs(page_dir+'/'+nom_dossier)

        # Récupération du lien de téléchargement
        download_link = resource.select_one('.download-button')['href']

        # Télécharger le fichier zip depuis l'URL
        response = requests.get(download_link)

        if response.status_code == 200:
            with open(page_dir+'/'+nom_dossier+'/'+download_link.split('/')[-1], 'wb') as file:
                file.write(response.content)
            print(f"Fichier {download_link.split('/')[-1]} téléchargé avec succès !")

            zip = page_dir+'/'+nom_dossier+'/'+download_link.split('/')[-1]
            # Décompression du fichier zip
            if(is_zip_file(zip)):
                with zipfile.ZipFile(zip, 'r') as zip_ref:
                    zip_ref.extractall(os.path.dirname(zip))
                print("Le fichier zip a été décompressé avec succès.")
            # Suppression du fichier zip
            os.remove(zip)
            print("Le fichier zip a été supprimé avec succès.")

```

📘 Fonction du script

Le script récupère le lien des pages de chaque ressource. Page sur laquelle sont présents les liens de téléchargement des fichiers **GeoJSON** zippés. Je récupère ensuite tous ces liens de téléchargement et je ramène les ressources sous la forme de fichier **ZIP** sur mon ordinateur. Je décompresse ensuite le contenu.

J'obtiens toute une architecture de plusieurs fichiers texte, cependant en les ouvrant, ils semblent être des fichiers `CSV`. Je vais donc changer leur extension :

```
#!/bin/bash

# Renommer les fichiers .txt en .csv de manière récursive
find data -type f -name "*.txt" | while read file; do
    mv "$file" "${file%.*}.csv"
done
```

En m'intéressant de plus près à ces fichiers `CSV`, je commence à bien comprendre leur structure et je sais qu'il y a 3 fichiers qui m'intéressent et dont je vais recouper les données, pour chacun des dossiers décompressés : `stops.csv`, `stop_times.csv` et `trips.csv`. Je crée alors un petit script `bash` pour compter le nombre d'arrêts mais aussi le nombre de passages à ces arrêts, histoire d'avoir une idée de la taille de mes données récoltées :

```
#!/bin/bash

# Compter le nombre total de lignes dans les fichiers CSV
total_lines=0
find data -type f -name "stops.csv" | while read file; do
    lines=$(wc -l < "$file")
    total_lines=$((total_lines + lines))
done
echo $total_lines

total_lines=0
find data -type f -name "stop_times.csv" | while read file; do
    lines=$(wc -l < "$file")
    total_lines=$((total_lines + lines))
done
echo $total_lines
```

Taille des données

Je me rends alors compte qu'il y a un peu plus de `62 millions` de passages sur un total de plus de `300 000` arrêts. Nous sommes face à une quantité de données assez conséquente. L'utilisation de `Spark` pour analyser, traiter et restructurer ces données est alors totalement justifiée et cohérente.

Traitement et structuration des données brutes

Comme nous l'avons vu précédemment, nous avons besoin de traiter une importante quantité de données. Pour simuler un environnement réel s'agissant d'une infrastructure de plusieurs machines, j'ai installé `Spark` sur deux containers afin de faire du calcul distribué. Voici le script d'installation que j'ai été amené à créer pour la mise en place de `Spark` sur une machine maître qui gère deux esclaves (une autre machine, et elle même) :

```
# Créer les images docker (portforwarding 8080 et 7077 pour l'interface web des workers et l'accès pour le pyspark)
docker run -it -d -p 8080:8080 -p 7077:7077 --name maître --hostname maître debian:latest
docker run -d --name esclave --hostname esclave debian:latest

# Sur la machine maître et les esclaves installer Spark+Hadoop :

apt update
# Télécharger Spark
apt install wget net-tools curl vim iproute2 iputils-ping ssh locate -y
cd /root
wget https://dlcdn.apache.org/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3-scala2.13.tgz
# Installer Java
```

```

apt install -y default-jre
apt install -y default-jdk
#Installer Spark
tar vxzf spark-3.5.1-bin-hadoop3-scala2.13.tgz
cp -r spark-3.5.1-bin-hadoop3-scala2.13 /usr/local

#Sur le maître :

#Ajouter les PATH nécessaires
export PATH="$PATH:/usr/local/spark-3.5.1-bin-hadoop3-scala2.13/bin"
export PATH="$PATH:/usr/local/spark-3.5.1-bin-hadoop3-scala2.13/sbin"
export SPARK_EXECUTOR_MEMORY=10g
#Vérifier que l'installation s'est bien déroulée
spark-shell
curl localhost:4040

#Sur toutes les machines, setup SSH :

#Autoriser l'accès root dans la config ssh
vim /etc/ssh/sshd_config # Ajouter "PermitRootLogin yes" et "PasswordAuthentication yes" et "PubkeyAuthentication yes"
service ssh restart
#Choisir un mot de passe pour le compte root afin de se connecter en SSH
passwd root
#Générer une clé RSA et autoriser les connexions sans password (Spark en a besoin)
ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
ssh-copy-id root@<IP_MACHINE_CIBLE>

#Sur toutes les machines :

locate spark-env.sh #Localiser les différents fichiers qu'on doit modifier
cp /usr/local/spark-3.5.1-bin-hadoop3-scala2.13/conf/spark-env.sh.template /usr/local/spark-3.5.1-bin-hadoop3-scala2.13/conf/spark-env.sh
vim /usr/local/spark-3.5.1-bin-hadoop3-scala2.13/conf/spark-env.sh #Ajouter export SPARK_MASTER_HOST="IP DU MAITRE"
cp /usr/local/spark-3.5.1-bin-hadoop3-scala2.13/conf/workers.template /usr/local/spark-3.5.1-bin-hadoop3-scala2.13/conf/workers
vim /usr/local/spark-3.5.1-bin-hadoop3-scala2.13/conf/workers #Ajouter les IP des workers donc l'esclave et le maître

#Sur la machine maître on démarre le cluster et on le teste avec un petit pyspark bidon :
/usr/local/spark-3.5.1-bin-hadoop3-scala2.13/sbin/start-all.sh # On se rend sur le port 8080 en web pour voir que l'on a accès à l'interface et que notre worker
est bien présent
apt install python3 python3-venv
cd /root
mkdir scripts
cd scripts
python3 -m venv venv
source venv/bin/activate
pip install pyspark
vim test.py #Y coller le script de test
python test.py
# Transverser tout mon environnement pyspark
docker cp . maître:/root/scripts/host

```

On pourrait installer Hadoop pour utiliser HDFS si on avait besoin de stocker les données sur le long terme. Cependant, nous allons utiliser Spark pour charger les données une seule fois afin de les restructurer de manière distribuée sur nos workers pour ensuite les transférer dans mongodb . Ainsi, nous n'avons nullement besoin de la technologie de stockage HDFS . Spark pre-build for Hadoop permet cependant une intégration facile avec Hadoop au cas où nous en aurions besoin.

Spark

3.5.1

Spark Master at spark://172.17.0.2:7077

URL: spark://172.17.0.2:7077

Alive Workers: 2

Cores in use: 40 Total, 0 Used

Memory in use: 60.1 GiB Total, 0.0 B Used

Resources in use:

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers (2)

Worker id	Address	State	Cores	Memory	Resources
worker-20240626171406-172.17.0.3-35589	172.17.0.3:35589	ALIVE	20 (0 Used)	30.0 GiB (0.0 B Used)	
worker-20240626171414-172.17.0.2-34357	172.17.0.2:34357	ALIVE	20 (0 Used)	30.0 GiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Voici un code Python utilisant PySpark pour vérifier que le cluster Spark fonctionne correctement :

```
from pyspark.sql import SparkSession
```

```
# Créer une session Spark
```

```
spark = SparkSession.builder \
    .appName("Test Cluster Spark + Hadoop") \
    .master("spark://172.17.0.2:7077") \
    .getOrCreate()
```

```
# Vérifier le fonctionnement en créant un DataFrame simple
```

```
data = [("Ava", 20), ("Nerumir", 25), ("Assistant", 30)]
df = spark.createDataFrame(data, ["Name", "Age"])
```

```
# Afficher le DataFrame
```

```
df.show()
```

```
# Arrêter la session Spark
```

```
spark.stop()
```

```
(venv) root@maitre:~/scripts# python test.py
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/06/26 16:59:59 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
using builtin-java classes where applicable
+-----+
|   Name|Age|
+-----+
|   Ava| 20|
| Nerumir| 25|
|Assistant| 30|
+-----+
(venv) root@maitre:~/scripts#
```

PySpark va utiliser le service qui tourne sur le port 7077 qui est celui du cluster :

```
root@maitre:~# netstat -lp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      6994/sshd: /usr/sb
i
tcp6       0      0 :::22                  :::*                   LISTEN      6994/sshd: /usr/sb
i
tcp6       0      0 :::8080                 :::*                   LISTEN      7082/java
tcp6       0      0 172.17.0.2:7077         :::*                   LISTEN      7082/java
```

Nous voulons utiliser `PySpark` pour structurer les données de la manière la plus maline pour notre besoin, voici le choix qui a été fait :

```
[
  {
    '_id': 1,
    'stop_id': 'identifiant de stop',
    'stop_name': 'exemple stop name',
    'lon': '12',
    'lat': '18',
    'times': [
      {
        'arrival_time': '06:59:00',
        'direction': 0,
        'next_stop': 'next_stop_id',
        'previous_stop': 'previous_stop_id'
      },
    ]
  },
]
```

Voici le script qui nous avons créé pour remplir cet objectif :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import collect_list, struct
from pyspark.sql.window import Window
from pyspark.sql.functions import lag, lead
import os

def recup_doss(dossier):
    tab = os.listdir(dossier)
    res = []
    for i in range(len(tab)):
        if (os.path.isdir(dossier + "/" + tab[i])):
            res.append(tab[i])
    return(res)

def navig_rec(dossier,res,visited):
    subs = recup_doss(dossier)
    for i in range(len(subs)):
        res.append(dossier + "/" + subs[i])
        if ((dossier + "/" + subs[i]) not in visited):
            visited.append(dossier + "/" + subs[i])
        res = navig_rec(dossier + "/" + subs[i],res,visited)
    return(res)

def csv_groups(dossier):
    folders = navig_rec(dossier,[],[])
    res = []
    for folder in folders:
        if os.path.isfile(folder+'/stop_times.csv'):
            res.append(folder)
    return res

def compute_jsons(dossier):

    # Créer une session Spark
    spark = SparkSession.builder \
        .appName("Réunir nos CSV dans un unique JSON") \
        .master("spark://172.17.0.2:7077") \
```

```

.getOrCreate()

for folder in csv_groups('data'):

    if(os.path.isfile(folder+'stops.csv') and os.path.isfile(folder+'stop_times.csv') and os.path.isfile(folder+'trips.csv')):

        # Liste des arrêts
        df_stops = spark.read.option("header", "true").csv(folder+'stops.csv')

        # Liste des horaires des arrêts pour les différentes lignes
        # Déjà trié par trip_id puis arrival_time, donc c'est parfait pour nous.
        df_stop_times = spark.read.option("header", "true").csv(folder+'stop_times.csv')

        # Lister les différentes lignes, on va y récupérer la direction.
        df_trips = spark.read.option("header", "true").csv(folder+'trips.csv')

        # Définir la fenêtre de partition par trip_id et trié par arrival_time
        windowSpec = Window.partitionBy("trip_id").orderBy("arrival_time")

        # Ajouter les colonnes previous_stop et next_stop à notre df
        df_stop_times = df_stop_times.withColumn("previous_stop", lag("stop_id", 1).over(windowSpec)).withColumn("next_stop", lead("stop_id",
1).over(windowSpec))

        # Ajouter la colonne direction à notre df
        combined_df = df_stop_times.join(df_trips, df_stop_times["trip_id"] == df_trips["trip_id"], "inner")
        df_stop_times = combined_df.select(
df_stop_times["trip_id"],
df_stop_times["arrival_time"],
df_stop_times["departure_time"],
df_stop_times["stop_id"],
df_stop_times["previous_stop"],
df_stop_times["next_stop"],
df_trips["direction_id"]
)

        df_stop_times = df_stop_times.withColumnRenamed("direction_id", "direction")

        # Sélectionner les colonnes nécessaires dans les tables
        df_stops_data = df_stops.select("stop_id", "stop_name", "stop_lon", "stop_lat")
        df_stop_times_data = df_stop_times.select("stop_id", "arrival_time", "direction", "next_stop", "previous_stop")

        # Joindre les données des deux tables sur stop_id
        combined_data = df_stops_data.join(df_stop_times_data, "stop_id", "inner")

        # Regrouper les données par stop_id et collecter les valeurs dans une colonne
        grouped_data = combined_data.groupBy("stop_id", "stop_name", "stop_lon", "stop_lat") \
            .agg(collect_list(struct("arrival_time", "direction", "next_stop", "previous_stop")).alias("times"))

        # Convertir les données regroupées en format JSON et afficher le résultat
        json_data = grouped_data.toJSON().collect()

        # Ajout de des stops en JSON dans le fichier de sortie
        with open(output_file_path, 'a') as file:
            for line in json_data:
                file.write("\n" + line)
                file.write(",")

    spark.stop()

#Chemin pour sauvegarder en fichier JSON
output_file_path = "formatted.json"

```

```
# Initialisation du JSON dans le fichier de sortie
with open(output_file_path, 'w') as file:
    file.write("[")

compute_jsons('data')

# Fermeture du JSON dans le fichier de sortie
with open(output_file_path, 'rb+') as file:
    file.seek(-1, os.SEEK_END) # Positionner le curseur sur l'avant-dernier caractère
    last_char = file.read(1).decode() # Lire le dernier caractère
    if last_char == ",":
        file.truncate() # Supprimer le dernier caractère s'il s'agit d'une virgule
with open(output_file_path, 'a') as file:
    file.write("]")
```

🕒 Obtenir les arrêts suivants et précédents


Pour obtenir les `next_stop` et `previous_stop`, j'ai groupé mes données temporairement par `trip_id` et puis j'ai trié cela par `arrival_time`, ce qui fait que les arrêts précédents et suivants dans le `dataframe` sont l'élément précédent et suivant. Le tri ayant lieu qu'une seule fois par `dataframe`, cela sauvegarde un temps considérable de parcours de la table. C'est une recherche par indexation manuelle. (Et oui voilà un exemple original d'utilisation d'indexation autre que la recherche dichotomique !)

J'ai rencontré un bug lorsque j'ai exécuté le script sur le service `Spark`. `PySpark` installe également sa propre version de `Spark` dont il se sert en tant que client pour exécuter ses scripts `python`. Ainsi, il faut s'assurer que la version de `Spark` qu'il installe soit compatible avec celle du serveur. En effet, `PySpark` utilisait `Scala 2.12` et le cluster `Spark` utilisait `Scala 2.13`:

```
6)
5)
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:63)
at java.base/java.lang.Thread.run(Thread.java:840)
Caused by: java.lang.RuntimeException: java.io.InvalidClassException: org.apache.spark.rpc.netty.RpcEndpointVerifier$CheckExistence; local class incompatible: stream classdesc serialVersionUID = 7789290765573734431, local class serialVersionUID = 5378738997755484868
at java.base/java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:597)
at java.base/java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:2051)
```

On exécute alors notre script `Python` avec le `Spark` du cluster directement :

```
spark-submit --master local csv_to_json.py
```


Spark Master at spark://172.17.0.2:7077

URL: spark://172.17.0.2:7077

Alive Workers: 2

Cores in use: 40 Total, 40 Used

Memory in use: 60.1 GiB Total, 20.0 GiB Used

Resources in use:

Applications: 1 Running, 7 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20240628213850-172.17.0.3-42449	172.17.0.3:42449	ALIVE	20 (20 Used)	30.0 GiB (10.0 GiB Used)	
worker-20240628213852-172.17.0.2-41655	172.17.0.2:41655	ALIVE	20 (20 Used)	30.0 GiB (10.0 GiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20240628230644-0007	(kill) Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 23:08:44	root	RUNNING	28 s

Completed Applications (7)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20240628230306-0006	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 23:03:06	root	FINISHED	26 s
app-20240628225931-0005	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 22:59:31	root	FINISHED	16 s
app-20240628224922-0004	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 22:49:22	root	FINISHED	1.0 min
app-20240628221826-0003	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 22:18:26	root	FINISHED	19 s
app-20240628221727-0002	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 22:17:27	root	FINISHED	15 s
app-20240628221511-0001	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 22:15:11	root	FINISHED	19 s
app-20240628220931-0000	Réunir nos CSV dans un unique JSON	40	10.0 GiB		2024/06/28 22:09:31	root	FINISHED	19 s

Le programme plantait à cause des noms de dossier avec les accents et les espaces. On y remédie en renommant les dossiers :

```
#!/bin/bash

# Fonction récursive pour renommer les dossiers
rename_folders() {
    local dir="$1"
    local count=1

    # Parcours de tous les dossiers dans le répertoire spécifié
    for folder in "$dir"/*; do
        if [ -d "$folder" ]; then
            # Renommer le dossier avec un numéro
            new_name="$dir/$count"
            mv "$folder" "$new_name"

            echo "Renommé : $folder en $new_name"
            ((count++))

            # Appel récursif sur chaque sous-dossier
            rename_folders "$new_name"
        fi
    done
}

# Appeler la fonction pour démarrer le processus de renommage
rename_folders "data"
```

```
-- 3
  |-- ARRET_12_technique_T_20240623T230000Z.xml
-- 4
  |-- 1
  |   |-- agency.csv
  |   |-- calendar.csv
  |   |-- calendar_dates.csv
  |   |-- routes.csv
  |   |-- shapes.csv
  |   |-- stop_times.csv
  |   |-- stops.csv
  |   |-- transfers.csv
  |   |-- trips.csv
-- 5
  |-- 1
  |   |-- agency.csv
  |   |-- calendar_dates.csv
  |   |-- feed_info.csv
  |   |-- routes.csv
  |   |-- shapes.csv
  |   |-- stop_times.csv
  |   |-- stops.csv
  |   |-- transfers.csv
  |   |-- trips.csv
-- 6
  |-- 1
  |   |-- agency.csv
  |   |-- calendar_dates.csv
  |   |-- feed_info.csv
  |   |-- routes.csv
  |   |-- shapes.csv
  |   |-- stop_times.csv
  |   |-- stops.csv
  |   |-- transfers.csv
  |   |-- trips.csv
-- 7
  |-- 1
  |-- 2
  |-- 3
-- 8
  |-- 1
  |   |-- agency.csv
  |   |-- calendar.csv
  |   |-- calendar_dates.csv
  |   |-- feed_info.csv
  |   |-- routes.csv
```

```
999922",{"arrival_time":"12:14:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"14:34:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"19:14:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"20:14:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"16:54:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"18:39:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"16:19:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"11:53:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"09:53:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"13:59:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"14:34:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"18:04:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"18:39:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"19:43:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"13:54:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"13:54:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"13:54:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"13:59:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"},{"arrival_time":"20:14:00","direction":"1","next_stop":"MOBIIITI:Quay:100016","previous_stop":"MOBIIITI:Quay:999922"}]}
24/06/28 23:10:04 INFO SparkContext: SparkContext is stopping with exitCode 0
24/06/28 23:10:04 INFO SparkUI: Stopped Spark web UI at http://maître:4040
24/06/28 23:10:04 INFO StandaloneSchedulerBackend: Shutting down all executors
24/06/28 23:10:04 INFO StandaloneSchedulerBackend$StandaloneDriverEndpoint: Asking each executor to shut down
24/06/28 23:10:04 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/06/28 23:10:04 INFO MemoryStore: MemoryStore cleared
24/06/28 23:10:04 INFO BlockManager: BlockManager stopped
24/06/28 23:10:04 INFO BlockManagerMaster: BlockManagerMaster stopped
24/06/28 23:10:04 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
24/06/28 23:10:04 INFO SparkContext: Successfully stopped SparkContext
24/06/28 23:10:05 INFO ShutdownHookManager: Shutdown hook called
24/06/28 23:10:05 INFO ShutdownHookManager: Deleting directory /tmp/spark-a2054fc7-d2e4-4262-a95e-53d6c8f090
24/06/28 23:10:05 INFO ShutdownHookManager: Deleting directory /tmp/spark-43cbc80a-0753-4c2f-b6ea-841623fe08bc
24/06/28 23:10:05 INFO ShutdownHookManager: Deleting directory /tmp/spark-43cbc80a-0753-4c2f-b6ea-841623fe08bc/pyspark-32d27f11-ab87-43df-a33b-b6cc6e142341
root@maître:~/scripts/hosts
```

Nous souhaitons maintenant stocker ces données. L'objectif étant de pouvoir les récupérer de manière optimisée. **MongoDB** possède un algorithme d'indexation par position géographique, nous allons donc l'utiliser.

```
# Installation du container mongodb et mongo-express

docker network create mongo-network

docker run -it -d -p 27017:27017 -e MONGO_INITDB_ROOT_USERNAME=admin -e MONGO_INITDB_ROOT_PASSWORD=password --network mongo-network --name mongo mongo

docker run -d -p 8081:8081 -e ME_CONFIG_BASICAUTH_USERNAME=admin -e ME_CONFIG_BASICAUTH_PASSWORD=password -e ME_CONFIG_MONGODB_SERVER=mongo -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin -e ME_CONFIG_MONGODB_ADMINPASSWORD=password --network mongo-network --name mongo-express mongo-express
```

Nous voilà sur l'interface de mongo-express :

Mongo ExpressDatabase

Mongo Express

Databases

Database Name

+ Create Database

View

admin

Del

View

config

Del

View

local

Del

Server Status









Hostname	bf0839dba3fa	MongoDB Version	7.0.11
Uptime	15 seconds	Node Version	18.20.3
Server Time	Sat, 29 Jun 2024 11:30:43 GMT	V8 Version	10.2.154.26-node.37
Current Connections	3	Available Connections	999997
Active Clients	0	Queued Operations	0
Clients Reading	0	Clients Writing	0
Read Lock Queue	0	Write Lock Queue	0
Disk Flushes		Last Flush	
Time Spent Flushing	ms	Average Flush Time	ms

Créons une collection appelée stops et important notre JSON généré avec PySpark pour remplir la collection des arrêts. La création de la collection a pu se faire sur mongo-express , mais pas l'importation du json pour une raison obscure. On passe donc par le container mongo directement :

```
docker cp formatted.json mongo:/root/
docker exec -it mongo bash
cd /root
mongoimport --authenticationMechanism SCRAM-SHA-256 --username admin --password password --authenticationDatabase admin --jsonArray --collection stops --file formatted.json
```

```
root@4ec12f96e18:/root# mongoimport --authenticationMechanism SCRAM-SHA-256 --username admin --password password --authenticationDatabase admin --jsonArray --collection stops --file formatted.json
connected to: mongodb://localhost/
2024-06-29T11:50:30.659+0000 [.....] test.stops 235MB/6.26GB (3.7%)
2024-06-29T11:50:33.660+0000 [#####] test.stops 459MB/6.26GB (7.2%)
2024-06-29T11:50:36.659+0000 [#####] test.stops 676MB/6.26GB (10.5%)
2024-06-29T11:50:39.660+0000 [#####] test.stops 852MB/6.26GB (13.3%)
2024-06-29T11:50:42.659+0000 [#####] test.stops 1.026GB/6.26GB (16.4%)
2024-06-29T11:50:45.661+0000 [#####] test.stops 1.206GB/6.26GB (19.2%)
2024-06-29T11:50:48.659+0000 [#####] test.stops 1.366GB/6.26GB (21.8%)
2024-06-29T11:50:51.660+0000 [#####] test.stops 1.516GB/6.26GB (24.1%)
2024-06-29T11:50:54.659+0000 [#####] test.stops 1.706GB/6.26GB (27.1%)
2024-06-29T11:50:57.660+0000 [#####] test.stops 1.886GB/6.26GB (30.1%)
2024-06-29T11:51:00.660+0000 [#####] test.stops 2.066GB/6.26GB (33.0%)
2024-06-29T11:51:03.659+0000 [#####] test.stops 2.256GB/6.26GB (36.0%)
2024-06-29T11:51:06.659+0000 [#####] test.stops 2.426GB/6.26GB (38.7%)
2024-06-29T11:51:09.659+0000 [#####] test.stops 2.586GB/6.26GB (41.3%)
2024-06-29T11:51:12.660+0000 [#####] test.stops 2.746GB/6.26GB (43.8%)
2024-06-29T11:51:15.660+0000 [#####] test.stops 2.896GB/6.26GB (46.2%)
2024-06-29T11:51:18.660+0000 [#####] test.stops 3.076GB/6.26GB (49.0%)
2024-06-29T11:51:21.660+0000 [#####] test.stops 3.076GB/6.26GB (49.0%)
```

L'importation a été effectuée avec succès et nous avons nos éléments comme nous le souhaitons. Au final, nous avons plus de 300 000 arrêts et 62 millions d'heures. L'utilisation d'une bonne indexation géographique sur une zone optimisée sera alors crucial :

  667ff507fb7972ec8bfa8399	BLOIS:Quay:CBBOURGA	Bourg	1.255064	47.711201	<pre>e[@{4 items}, @{3 items}]</pre>
  667ff507fb7972ec8bfa839a	BLOIS:Quay:CHCAMPPIR	Camping	1.309862	47.542698	<pre>e[e["arrival_time": "12:47:00", "direction": "1", "next_stop": "BLOIS:Quay:CHMAIRIA", "previous_stop": "BLOIS:Quay:BLGARER"], @{4 items}, @{4 items}, @{4 items}, @{4 items}, @{4 items}, @{4 items}]</pre>
  667ff507fb7972ec8bfa839b	BLOIS:Quay:ONMEUVA	MEUVES	1.149296	47.493038	<pre>e[@{4 items}]</pre>
  667ff507fb7972ec8bfa839c	BLOIS:Quay:BLJOLYR	Joly	1.326825	47.596233	<pre>e[@{4 items},]</pre>

Editing Document: 667ff507fb7972ec8bfa839a

← Back

Save

```
1 {
2   _id: ObjectId('667ff507fb7972ec8bfa839a'),
3   stop_id: 'BLOIS:Quay:CHCAMPPIR',
4   stop_name: 'Camping',
5   stop_lon: '1.309862',
6   stop_lat: '47.542698',
7   times: [
8     {
9       arrival_time: '12:47:00',
10      direction: '1',
11      next_stop: 'BLOIS:Quay:CHMAIRIA',
12      previous_stop: 'BLOIS:Quay:BLGARER'
13    },
14    {
15      arrival_time: '16:25:00',
16      direction: '1',
17      next_stop: 'BLOIS:Quay:CHROCHEA',
18      previous_stop: 'BLOIS:Quay:BLCBLVIE'
19    },
20    {
21      arrival_time: '16:31:00',
22      direction: '1',
23      next_stop: 'BLOIS:Quay:CHBOURGR',
24      previous_stop: 'BLOIS:Quay:CHARCOUR'
25    },
26    {
27      arrival_time: '18:48:00',
28      direction: '1',
29      next_stop: 'BLOIS:Quay:CHBOURGR',
30      previous_stop: 'BLOIS:Quay:CHARCOUR'
31    }
32  ]
33 }
```


Appliquons maintenant un index `2dsphere` sur la clé `stop_lon` et `stop_lat` en créant d'abord notre champ `geo` et ensuite notre index :

```
mongosh
use admin
db.auth("admin", "password")
use test
db.stops.updateMany({}, [{ $set: { "location": { "type": "Point", "coordinates": [{ $toDouble: "$stop_lon" }, { $toDouble: "$stop_lat" } ] } } }])
db.stops.createIndex({ "location": "2dsphere" })
```

🔗 Optimisation

Cela va nous faire gagner un temps considérable. En effet, nous avons plus de 300 000 arrêts. Nous détaillerons tout cela un peu plus tard.

```
test> use admin
switched to db admin
admin> db.auth("admin", "password")
{ ok: 1 }
admin> use test
switched to db test
test> db.stops.updateMany({}, [{ $set: { "location": { "type": "Point", "coordinates": [{ $toDouble: "$stop_lon" }, { $toDouble: "$stop_lat" } ] } } }])
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 339986,
  modifiedCount: 339986,
  upsertedCount: 0
}
test> db.stops.createIndex({ "location": "2dsphere" })
location_2dsphere
test> █
```

On peut vérifier sur `mongo-express` que le champ a été ajouté avec succès, de même pour l'index :

```
    },
    ],
    location: {
      type: 'Point',
      coordinates: [
        1.364638,
        47.57671
      ]
    }
  }
}
```

Indexes

Name	Columns	Size	Attributes	Actions
id	_id ASC	3.88 MB		 DEL
location_2dsphere	location DSC	4.65 MB	2dsphereIndexVersion: 3	 DEL

On peut désormais récupérer de manière optimisée tous les arrêts dans un rayon en kilomètres autour d'une point géographique :

```
from pymongo import MongoClient

# Spécifier les informations d'identification
username = "admin"
password = "password"

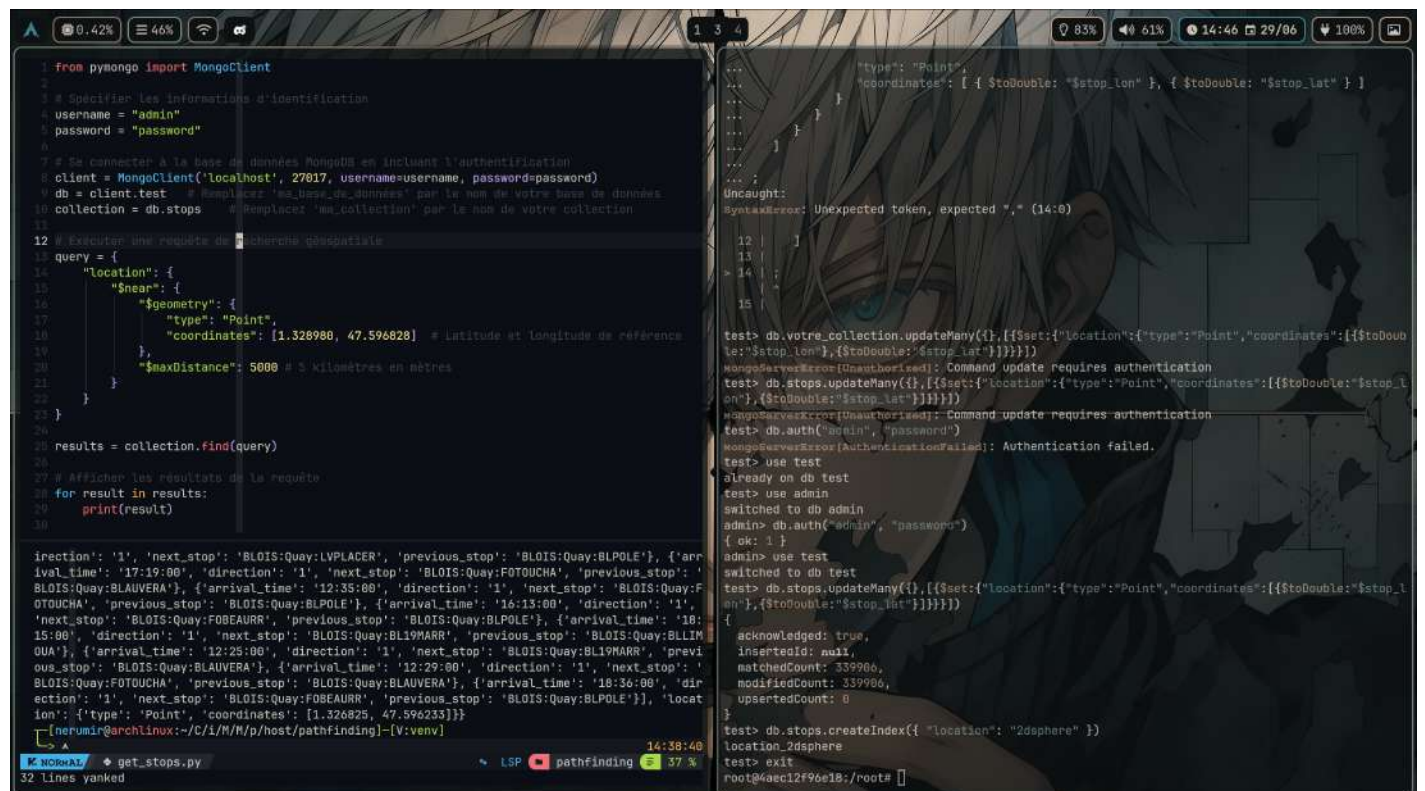
# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test
collection = db.stops

# Exécuter une requête de recherche géospatiale
query = {
    "location": {
        "$near": {
            "$geometry": {
                "type": "Point",
                "coordinates": [1.328980, 47.596828] # Latitude et longitude de référence
            },
            "$maxDistance": 5000 # 5 kilomètres
        }
    }
}

results = collection.find(query)

# Afficher les résultats de la requête
for result in results:
    print(result)

# Fermer la connexion à la base de données
client.close()
```



```
1 from pymongo import MongoClient
2
3 # Spécifier les informations d'identification
4 username = "admin"
5 password = "password"
6
7 # Se connecter à la base de données MongoDB en incluant l'authentification
8 client = MongoClient('localhost', 27017, username=username, password=password)
9 db = client.test # Remplacez 'ma_base_de_donnees' par le nom de votre base de données
10 collection = db.stops # Remplacez 'ma_collection' par le nom de votre collection
11
12 # Exécuter une requête de recherche géospatiale
13 query = {
14     "location": {
15         "$near": {
16             "$geometry": {
17                 "type": "Point",
18                 "coordinates": [1.328980, 47.596828] # Latitude et longitude de référence
19             },
20             "$maxDistance": 5000 # 5 kilomètres en mètres
21         }
22     }
23 }
24
25 results = collection.find(query)
26
27 # Afficher les résultats de la requête
28 for result in results:
29     print(result)
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2
```

Avant de passer à l'étape suivante, il est bien d'avouer que malgré le fait que les données fût d'un très bon état et sans erreur de format, je suis passé à côté d'horaires qui vont au delà de 24 heures. Je ne m'en suis rendu compte que trop tard, donc je fais le nettoyage sur [MongoDB](#) directement :

```
from pymongo import MongoClient

# Spécifier les informations d'identification
username = "admin"
password = "password"
# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test

# Requête mongodb pour modifier les occurrences d'horaires incorrectes. (à mettre juste après l'intégration à mongodb sur le rendu pas ici mdr)
db.stops.update_many(
    { },
    { '$pull': { 'times': { 'arrival_time': { '$gt': '23:59:59' } } } }
)

query = { 'times.time': { '$gt': '23:59:59' } }
count = db.stops.count_documents(query)

print(f"Documents incorrects restants : {count}")
```

Algorithme de pathfinding initial

On va maintenant créer un algorithme de [pathfinding](#) pour calculer le chemin le plus court en tenant compte de ces arrêts pour aller d'un point A à un point B en combinant la marche à pied et le transport en commun. Notons que pour les portions exclusivement à pied ainsi que l'estimation en voiture, nous allons utiliser un algorithme déjà connu qui s'appelle [Dijkstra](#) . Nous sommes cependant obligé d'inventer notre propre algorithme de [pathfinding](#) pour prendre en compte les transports en commun. Afin d'observer les différences entre l'utilisation de la voiture et des transports en commun, nous voulons générer une quantité assez importantes de données sous cette forme là :

```
[
  {
    'depart': {
      'type': 'Point',
      'coordinates': [
        1.45,
        47.3
      ]
    },
    'destination': {
      'type': 'Point',
      'coordinates': [
        1.67,
        47.6
      ]
    },
    'time_car': '280',
    'time_transport': '2790',
    'ratio_car_transport': '9.96',
    'stops_density': '8',
    'steps': [
      {
        'type': 'walk',
        'time': '920'
      },
      {
        'type': 'transport',
```

```

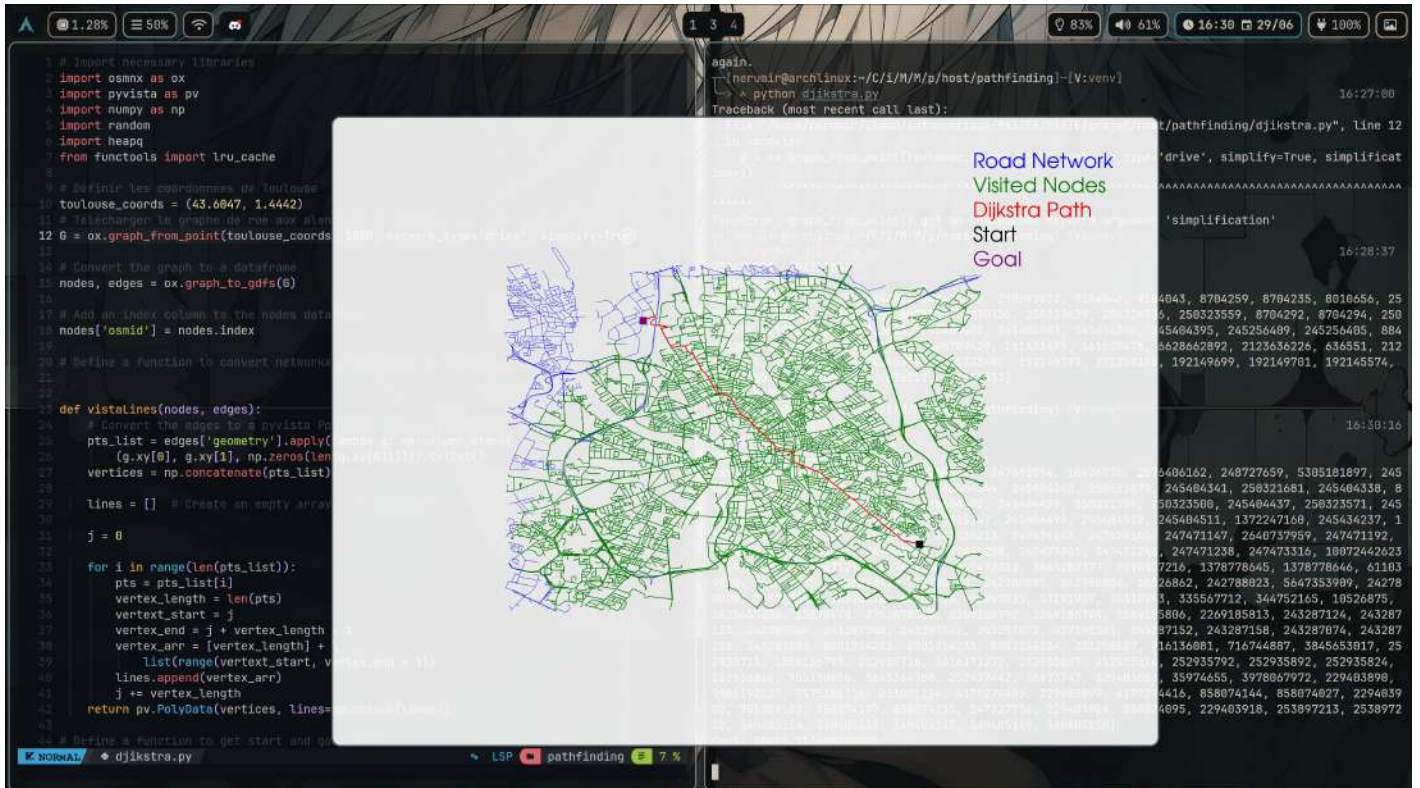
        'time': '1870'
    }
}
],
},
]

```

Notons que je calcule déjà des données intéressantes au moment de la génération de la donnée pour des raisons d'optimisation. J'ai alors enregistré la densité des arrêts (arrêts par kilomètres carrés) autour du point de départ ainsi que le ratio :

$$ratio = \frac{\text{temps transport}}{\text{temps voiture}}$$

Pour réaliser cela, il faut trouver un bon module pour faire du `pathfinding`. Je suis tombé sur un article qui me donne l'algorithme `Dijkstra` en Python (ce n'est pas le plus rapide, il est même peu optimal, mais sa fiabilité est un atout et comme nous allons le voir, sa vitesse est suffisamment satisfaisante) :



J'ai grandement modifié le code proposé par l'auteur afin de correspondre à mes besoins. Voici le script final qui permet de calculer le chemin le plus court d'un point A à un point B :

```

import osmnx as ox
import heapq
from functools import lru_cache

# Fonction pour retrouver le chemin jusqu'au départ une fois l'arrivée atteinte.
def reconstruct_path(start, goal, predecessors):
    path = [goal]
    while path[-1] != start:
        path.append(predecessors[path[-1]])
    path.reverse()
    return path

# Déterminer les prochains nodes à partir du précédent pour la propagation du dijkstra
@lru_cache(maxsize=None)
def successors(G, node):
    neighbors = list(G.neighbors(node))
    successors = [(neighbor, G[node][neighbor][0]['length'])]

```



```

        for neighbor in neighbors]
    return successors

# Fonction de l'algo de Dijkstra
def dijkstra(G, start, goal):
    # Calculer la distance de chaque noeud par rapport à celui de départ
    distances = {node: float('inf') for node in G.nodes}
    distances[start] = 0 # Distance du noeud de départ, c'est forcément 0
    visited = set() # Pour stocker les noeuds déjà visités
    queue = [(0, start)] # Queue de priorité (avec les éléments distance/noeud)
    predecessors = {node: None for node in G.nodes}
    # Tant qu'il y a au moins un élément dans la queue
    while queue:
        # On traite le prochain élément de la queue
        current_distance, current_node = heapq.heappop(queue)
        # Si c'est l'arrivée, on arrête et on rend la distance, donc le résultat
        if current_node == goal:
            return distances[goal]
        # Si le noeud a traité a déjà été visité, on ignore
        if current_node in visited:
            continue
        # On ajoute ce noeud à la liste des noeuds visités
        visited.add(current_node)
        # On boucle sur les prochains noeuds après celui-ci
        for neighbor, weight in successors(G, current_node):
            # On calcul la distance si on atteint le noeud suivant concerné
            new_distance = current_distance + weight
            # Si la distance est inférieur à celle que l'on avait mis pour ce voisin
            # (ça fonctionne car au début on avait mis les valeurs de distances à l'infini)
            if new_distance < distances[neighbor]:
                # Alors on remplit la case des distances et les predecessors.
                distances[neighbor] = new_distance
                predecessors[neighbor] = current_node
            # On ajoute également les prochains noeuds à la pile pour continuer la propagation de l'algo
            heapq.heappush(queue, (new_distance, neighbor))

    return float('inf'), [] # Si on arrive ici, aucun chemin n'a été trouvé

# Renvoie le nombre de mètres à parcourir
def pathf(start,end,G):

    # Trouver le noeud le plus proche
    start_node = ox.nearest_nodes(G, start[1], start[0])
    end_node = ox.nearest_nodes(G, end[1], end[0])
    # Lancer l'algorithme de Dijkstra sur les deux noeuds
    dijkstra_cost = dijkstra(G, start_node, end_node)
    # Rendre le résultat
    if dijkstra_cost != float('inf'):
        return dijkstra_cost
    else:
        return None

```

Si j'utilisais l'algorithme de `pathfinding` proposé par `osmnx`, j'aurais été beaucoup plus lent :

```

import osmnx as ox
import random
import time

start_time = time.time()

# Définir les coordonnées de Toulouse

```

```

toulouse_coors = (43.6047, 1.4442)

# Télécharger le graphe de rue aux alentours de Toulouse
G = ox.graph_from_point(toulouse_coors, 5000, network_type='walk')

# Choisir aléatoirement deux noeuds du graphe pour représenter les points proches
nodes = list(G.nodes())
random.shuffle(nodes)
point_A = nodes[0]
point_B = nodes[1]

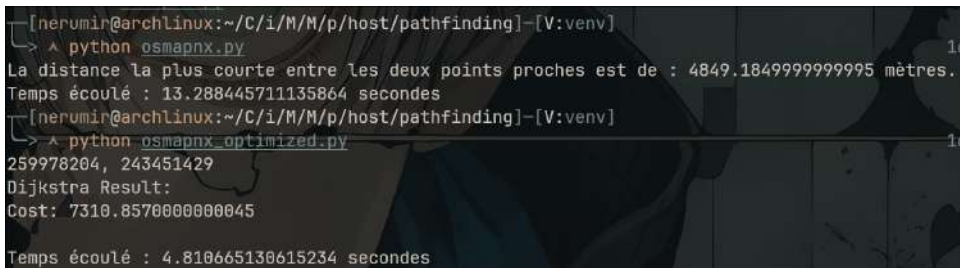
# Calculer le chemin le plus court à pied entre les deux points proches
route = ox.shortest_path(G, point_A, point_B, weight='length')

# Calculer la distance en mètres du chemin le plus court en utilisant la fonction routing.route_to_gdf()
route_gdf = ox.routing.route_to_gdf(G, route)
shortest_distance = route_gdf['length'].sum()

print(f"La distance la plus courte entre les deux points proches est de : {shortest_distance} mètres.")

# Fin du chronomètre
end_time = time.time()
# Calcul du temps écoulé
elapsed_time = end_time - start_time
print(f"Temps écoulé : {elapsed_time} secondes")

```



```

[nerumir@archlinux:~/C/i/M/M/p/host/pathfinding]-[V:venv]
> ^ python osmapnx.py
La distance la plus courte entre les deux points proches est de : 4849.1849999999995 mètres.
Temps écoulé : 13.288445711135864 secondes
[nerumir@archlinux:~/C/i/M/M/p/host/pathfinding]-[V:venv]
> ^ python osmapnx_optimized.py
259978204, 243451429
Dijkstra Result:
Cost: 7310.8570000000045
Temps écoulé : 4.810665130615234 secondes

```

Comme on peut le voir, après plusieurs tests, j'en conclus donc que j'ai divisé le temps de calcul par 3.

📌 Utilité du Dijkstra

Mon objectif est de me servir de cet algorithme de `pathfinding` au sein d'un autre que j'inventerai moi même pour répondre au besoin de trouver le chemin le plus court avec une combinaison de transport en commun et de marche à pied avec les données collectées dans la collection `mongodb stops`. J'aurais alors besoin d'exécuter plusieurs fois cet algorithme autour d'un point `A` et `B` donnés.

Après avoir fait d'autres mesures de complexité de l'algorithme, il semble que ramener le graphe prend un temps qui est comme on pourrait s'y attendre, proportionnel au carré du rayon que l'on indique. Ce calcul est en réalité responsable de presque tout le temps écoulé. Je dois donc optimiser mon chargement du graphe. J'ai essayé de télécharger le graphe de toute la France une bonne fois pour toute et de le réutiliser pour m'épargner ce genre de délais, cependant, c'était beaucoup trop long et ça allait prendre des années. En effet, il n'y a pas que le téléchargement mais également de la segmentation du graphe et des calculs de méta-données et globalement tout le processus de conversion de la carte en graphe.

Mon choix s'est alors finalement porté sur le fait de charger le graphe autour du point de départ dans un rayon r tel que :

$$r = \text{dist}(\text{start}, \text{end}) \times 1.5$$

Avec `dist` qui est la fonction de distance entre les deux nœuds, calculées en fonction des coordonnées géographiques avec la formule de Haversine (puisque la Terre n'est pas plate, en déplaie aux `platistes`) :

```

R = 6371000
dlat = lat2 - lat1
dlon = lon2 - lon1
a = sin2 $\left(\frac{\text{dlat}}{2}\right)$  + cos(lat1) · cos(lat2) · sin2 $\left(\frac{\text{dlon}}{2}\right)$ 
c = 2 · tan-1 $\left(\frac{\sqrt{1-a}}{\sqrt{a}}\right)$ 
distance = R · c

```

Vu que je vais avoir besoin de lancer le calcul de `pathfinding` plusieurs fois sur une même zone entre un point de départ et un point d'arrivée donnés, il vaut mieux que je charge le graphe qu'une seule fois par point de départ et d'arrivée. L'algorithme de `pathfinding` est beaucoup plus rapide, avec environ une seconde de temps, divisant alors encore par 5 le temps écoulé entre les calculs de `pathfinding`, mais en ayant toujours le chargement initial du graphe :

```
from math import radians, sin, cos, sqrt, atan2
import osmnx as ox
import pf_dijkstra as dij
import random

def haversine_distance(lat1, lon1, lat2, lon2):
    R = 6371000 # Rayon de la Terre en mètres

    # Conversion des degrés en radians
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])

    # Calcul des écarts en latitudes et longitudes
    dlat = lat2 - lat1
    dlon = lon2 - lon1

    # Formule de la distance haversine
    a = sin(dlat / 2) ** 2 + cos(lat1) * cos(lat2) * sin(dlon / 2) ** 2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    distance = R * c
    return distance

# Je choisis deux noeuds aléatoires sur le graphe donné
def get_start_goal_nodes(G):
    nodes = list(G.nodes)

    start = random.choice(nodes)
    goal = random.choice(nodes)

    while start == goal:
        goal = random.choice(nodes)

    return start, goal

# Mes points d'arrivée et de départ en exemple
start = (43.6045, 1.4440)
end = (43.5850, 1.4559)
# Je calcule mon rayon selon ma formule
radius = int(haversine_distance(start[0], start[1], end[0], end[1]) * 1.5)
# Je charge le graphe une bonne fois pour toute
G = ox.graph_from_point(start, radius, network_type='walk', simplify=True)

# Je simule plein de calculs de pathfinding aléatoires
for i in range(100):
    start, end = get_start_goal_nodes(G)
    start = (G.nodes[start]['y'], G.nodes[start]['x'])
    end = (G.nodes[end]['y'], G.nodes[end]['x'])
    print(dij.pathf(start, end, G))
```

Algorithme de `pathfinding` pour les transports en commun

Création de l'algorithme

L'idée que j'ai eu pour inventer mon algorithme de `pathfinding` utilisant les arrêts de transports en commun est la suivante :

- Initialiser un tableau vide des chemins complétés `final_paths` .
- Initialiser un tableau des chemins qui sont en train d'être explorés `current_paths` , contenant que le nœud de départ.
- Un tableau contenant tous les arrêts de la zone `stops` .
- Un tableau `stop_nodes` associant les `stop_id` à un temps de chemin minimal pour lequel il a été atteint et son nœud le plus proche sur le graphe.
- On boucle et pour chaque élément de `current_paths` on créé :
 - Un nouveau chemin par arrêt que l'on peut atteindre par transport depuis l'arrêt actuel (position actuelle).
 - Un nouveau chemin par autre arrêt à pied (même si l'arrêt est atteignable via cet arrêt par transport par l'intermédiaire de plusieurs arrêts, il sera éventuellement atteint par un des chemins créé à l'étape d'avant dans les prochaines itérations)
 - Un nouveau chemin qui va directement à l'arrivée à pied.
- Si les arrêts sont atteints en moins de temps que ce qui est indiqué dans `stop_nodes` , alors ils apparaîtront dans le nouveau tableau `current_paths` . Il sera vidé au préalable. Si le chemin est à l'arrivée, alors il sera ajouté dans `final_paths` à la place.
- La boucle s'arrête lorsque `current_paths` est vide.
- On rend finalement le chemin le plus court contenu dans `final_paths` .

Voici mon algorithme de `pathfinding` dans sa version finale :

```
from datetime import datetime, timedelta
import osmnx as ox
import dijkstra as dij

class pf:

    VITESSE_PIETON = 4 # Vitesse de marche en km/h
    VITESSE_VOITURE = 30 # Vitesse moyenne d'une voiture en 30km/h

    def __init__(self, start, end, start_time, stops, radius, G):
        # Initialiser des variables dont on va se servir dans notre programme
        self.start = start
        self.end = end
        self.radius = radius
        self.G = G
        self.start_node = ox.nearest_nodes(self.G, self.start[1], self.start[0])
        self.end_node = ox.nearest_nodes(self.G, self.end[1], self.end[0])
        self.start_time = start_time
        self.stops = stops
        # Initialiser le tableau des chemins finaux avec le chemin où on fait tout à pied
        seconds = int(int(dij.pathf(self.start_node, self.end_node, self.G))/((1/3.6)*self.VITESSE_PIETON))
        time = self.addToTime(self.start_time, seconds)
        self.car_time = int(int(dij.pathf(self.start_node, self.end_node, self.G))/((1/3.6)*self.VITESSE_VOITURE))
        self.final_paths = [
            [
                {
                    'node': ox.nearest_nodes(self.G, self.end[1], self.end[0]),
                    'time': time,
                    'wentBy': 'walk'
                }
            ]
        ]

        # Initialiser les noeuds des arrêts
        self.stop_nodes = {}
        for stop in self.stops:
            elem = {
```

```
'time': '23:59:59', # L'heure la plus tard possible
'node': ox.nearest_nodes(self.G, float(stop['stop_lon']), float(stop['stop_lat']))
}
self.stop_nodes[stop['_id']] = elem
# Initialiser current_paths avec la première étape
self.current_paths = []
actual_node = self.start_node
actual_time = self.start_time
for stop_id, stop in self.stop_nodes.items():
    # Eviter d'aller d'un point A à un point A...
    if(stop['node'] == actual_node):
        continue
    distance = int(dij.pathf(actual_node, stop['node'], self.G))
    time = self.addToTime(actual_time, int(int(distance)/((1/3.6)*self.VITESSE_PIETON)))
    # Ajouter le node seul comme chemin
    if(self.shorterThanEver(stop['node'], time)):
        self.current_paths.append([
            'node': stop['node'],
            'time': time,
            'wentBy': 'walk'
        ])
def pathf(self):
    # Tant que current_paths n'est pas vide
    i = 0
    while(self.current_paths):
        i += 1
        next_paths = []
        for current_path in self.current_paths:
            step = current_path[-1]
            # Si on est à l'arrivée, alors on l'ajoute au final_paths
            if step['node'] == self.end_node:
                self.final_paths.append(current_path)
                continue
            stop_of_step = self.stopById(self.stopByNode(step['node']))
            # On prend en compte les arrêts qu'on peut atteindre par le transport
            new_steps = []
            if(stop_of_step):
                new_steps = self.reachable_stops(stop_of_step, step['time'])
            # On prend en compte le fait de marcher aux autres arrêts mais uniquement si le trajet précédent n'était pas à pied
            if(step['wentBy'] != 'walk'):
                # On prend en compte la suite du chemin faite à pied
                distance = int(dij.pathf(step['node'], self.end_node, self.G))
                time = self.addToTime(step['time'], int(distance)/((1/3.6)*self.VITESSE_PIETON)))
                new_steps.append({
                    'node': self.end_node,
                    'time': time,
                    'wentBy': 'walk'
                })
            # Ne pas considérer les arrêts où l'on est moins rapide que déjà fait précédemment
            for stop_id, stop in self.stop_nodes.items():
                # Eviter d'aller d'un point A à un point A...
                if(stop['node'] == step['node']):
                    continue
                distance = int(dij.pathf(step['node'], stop['node'], self.G))
                time = self.addToTime(step['time'], int(distance)/((1/3.6)*self.VITESSE_PIETON)))
                # Ajouter le node seul comme chemin
                if(self.shorterThanEver(stop['node'], time)):
                    new_steps.append({
                        'node': stop['node'],
                        'time': time,
                        'wentBy': 'walk'
```

```

    })

    # Créer les nouveaux chemins grâce aux nouvelles étapes générées
    for new_step in new_steps:
        next_paths.append(current_path + [new_step])
    self.current_paths = next_paths

# Sélection du chemin le plus rapide parmi tous ceux calculés
better_path = self.final_paths[1]
i = 0
for path in self.final_paths:
    i += 1
    if(self.toTime(path[-1]['time']) <= self.toTime(better_path[-1]['time']) and path != self.final_paths[0]):
        better_path = path

#Calcul du temps de trajet en transport
self.transport_time = self.timeToSec(better_path[-1]['time']) - self.timeToSec(better_path[1]['time'])
# On ajoute le temps de marche à pied pour aller au premier arrêt. On le fait à part pour pouvoir ignorer le temps d'attente du premier arrêt.
# Supposant alors que l'individu part de chez lui à l'heure adaptée au passage à l'arrêt. C'est le comportement réaliste.
self.transport_time += (self.timeToSec(better_path[1]['time']) - self.timeToSec(self.start_time))

# Génération du résultat
res = {
    'depart': {
        'type': 'Point',
        'coordinates': [
            float(self.start[0]),
            float(self.start[1])
        ]
    },
    'destination': {
        'type': 'Point',
        'coordinates': [
            float(self.end[0]),
            float(self.end[1])
        ]
    },
    'time_car': self.car_time + 300,
    'time_transport': self.transport_time,
    'ratio_car_transport': round(self.transport_time/self.car_time, 2),
    # La densité en arrêt par km²
    'stops_density': round(len(self.stops) / (self.radius/1000), 2),
    'steps': better_path
}

return res

# Convertir un temps au format string en secondes
def timeToSec(self, date_str):
    heures, minutes, secondes = map(int, date_str.split(':'))
    return heures * 3600 + minutes * 60 + secondes

# Convertir l'heure en temps comparable
def toTime(self, heure):
    return datetime.strptime(heure, "%H:%M:%S").time()

def addToTime(self, heure, seconds):
    heure_time = datetime.strptime(heure, "%H:%M:%S")
    new_heure_time = heure_time + timedelta(seconds=seconds)
    return new_heure_time.strftime("%H:%M:%S")

# Oui ou non si le node a été atteint en un temps plus court que ce qui a été enregistré jusqu'à présent
# La fonction actualise aussi stop_nodes
def shorterThanEver(self, node, time):

```

```

for stop_id, stop in self.stop_nodes.items():
    if(stop['node'] == node):
        try:
            if(self.toTime(time) <= self.toTime(stop['time'])):
                self.stop_nodes[stop_id] = {
                    'node': node,
                    'time': time
                }
            return True
        except Exception:
            pass
    return False

# Si l'arrêt à été atteint en un temps plus court que ce qui est déjà présent dans la liste, alors on remplace.
# Si il n'est pas dans la liste, alors on l'ajoute si il a été atteint plus rapidement qu'indiqué dans stop_nodes.
# Optimisable car l'ordre est trié par horaire, mais je prends pas de risque.
def addOrReplaceIfQuickest(self, new_stop, stops):
    exist = False
    for index, stop in enumerate(stops):
        if(new_stop['node'] != stop['node']):
            continue
        exist = True
        if(self.toTime(new_stop['time']) <= self.toTime(stop['time'])):
            stops[index] = new_stop
            return stops
    if(not exist):
        if(self.shorterThanEver(new_stop['node'], new_stop['time'])):
            stops.append(new_stop)
    return stops

# Obtenir les informations d'un arrêt avec son id
def stopById(self, stop_id):
    for stop in self.stops:
        if(stop['_id'] == stop_id):
            return stop
    return None

# Obtenir un stop_id à partir du node
def stopByNode(self, node):
    for stop_id, stop in self.stop_nodes.items():
        if(stop['node'] == node):
            return stop_id
    return None

# Obtenir l'horaire d'arrivée à un arrêt à partir du précédent et de l'horaire du précédent
def nextStopTime(self, previous_stop, previous_time, stop_id):
    stop = self.stopById(stop_id)
    for passage in stop['times']:
        try:
            if(self.toTime(passage['arrival_time']) >= self.toTime(previous_time)):
                return passage['arrival_time']
        except Exception:
            pass
    return '23:59:59'

# Obtenir un arrêt par son nom
def stopByName(self, stop_name):
    for stop in self.stops:
        if(stop['stop_id'] == stop_name):
            return stop
    return None

```



```

# Liste des arrêts atteignables par transport à partir d'un arrêt à un temps donné
# On prend pas en compte ceux qui sont atteint plus lentement que d'autres chemins
def reachable_stops(self, stop, time):
    res = []
    for passage in stop['times']:
        new_stop = 'next_stop'
        if('next_stop' not in passage):
            new_stop = 'previous_stop'
            if('previous_stop' not in passage):
                continue
    try:
        # Si il peut arriver à temps pour l'horaire et que le prochain arrêt est dans le scope on essaye de l'ajouter aux arrêts potentiellement atteignables
        if(self.toTime(passage['arrival_time']) >= self.toTime(time) and self.stopByName(passage[new_stop])):
            next_stop_id = self.stopByName(passage[new_stop])['_id']
            time = self.nextStopTime(stop, passage['arrival_time'], next_stop_id)
            if(time != '23:59:59'):
                res = self.addOrReplaceIfQuickest({
                    'node': self.stop_nodes[next_stop_id]['node'],
                    'time': time, # On ajoute le prochain arrêt avec son horaire d'arrivée
                    'wentBy': 'transport'
                }, res)
    except Exception:
        pass
    return res

```

Voici un exemple de résultat :

```

{
  "depart":{
    "type":"Point",
    "coordinates":[
      45.7005864,
      5.9462967
    ]
  },
  "destination":{
    "type":"Point",
    "coordinates":[
      45.6984046,
      5.9482393
    ]
  },
  "time_car":335,
  "time_transport":128,
  "ratio_car_transport":3.66,
  "stops_density":24.74,
  "steps":[
    {
      "node":300532742,
      "time":"11:25:22",
      "wentBy":"walk"
    },
    {
      "node":1766005294,
      "time":"11:52:00",
      "wentBy":"transport"
    },
    {
      "node":1983062779,
      "time":"11:54:08",
      "wentBy":"walk"
    }
  ]
}

```

```
}  
]  
}
```

On voit ici que le chemin le plus court consiste en 5 minutes de marche, environ 30 minutes de transport, en comptant l'attente puis à nouveau un peu plus de 2 minutes de marche.

🕒 Bug Docker

J'ai été confronté à un bug lors du redémarrage de mon container `mongo`, en inspectant les logs, j'ai réalisé qu'il fallait simplement supprimer le fichier suivant :

```
sudo rm /var/lib/docker/overlay2/d145531987d7baf33723d42875e382049d3acde2af105ce1a5ec9ffac24a80e/diff/tmp/mongodb-27017.sock
```

Optimisation de l'algorithme

Optimisation d'utilisation

Pour que l'algorithme se finisse en un temps raisonnable et qu'il ait des résultats exploitables voici ce que j'ai gardé en tête :

- Limiter le nombre d'arrêts à moins de 200 car complexité quadratique sur eux.
- Supprimer le chemin fait entièrement à pied. Il est presque tout le temps favorisé pour des raisons de conception de l'algorithme. En effet, il faut compter le temps entre la deuxième étape et l'arrivée. On ajoutera alors manuellement le temps à pied pour se rendre au premier arrêt. Ce qui permet d'ignorer le temps d'attente à cet arrêt. Pour des raisons évidentes, en situation réelle, nous n'attendons pas au premier arrêt car nous partons de chez nous à un horaire cohérent avec celui-ci. C'est à cause de ce temps qui n'a pas lieu d'être que le chemin fait entièrement à pied est souvent choisi comme le plus court. C'est une subtilité qu'il faut prendre en compte pour que l'algorithme soit cohérent et fonctionnel.
- Nous allons l'utiliser sur plusieurs cœurs de processeur pour collecter nos données. J'en possède 20, cela multipliera alors par 20 la vitesse de calcul, qui passera d'environ 2 minutes pour 150 arrêts à environ 20 secondes. Nous aurons en effet besoin de faire ce même calcul plusieurs fois pour établir des statistiques de moyenne.

🕒 Ce qui peut être amélioré

Pour optimiser l'algorithme et le rendre utilisable sur de grandes quantités d'arrêts, mais je n'ai pas eu le temps ni les ressources en 10 jours :

- Il aurait été possible de fusionner la position de certains arrêts qui sont assez proches et de simplement concaténer leurs horaires, sans oublier de modifier les `stop_id` pour conserver la cohérence de cette fusion.
- Il aurait été possible de supprimer les arrêts de gare par exemple et de garder seulement les bus. Cependant, cela demande des opérations de filtre avancées en fonction par exemple de la distance relative des arrêts entre eux.
- En parallélisant les calculs sur plusieurs serveurs, nous aurions aussi pu diviser le temps de calcul pour les masses et pré-calculer un nombre important de trajets afin de créer un graphe prenant directement en compte les transports.
- Avec plus de temps, il aurait été possible d'apporter ce genre d'optimisation, mais surtout avec beaucoup plus de ressources système. Cependant, je ne pense pas que l'algorithme soit plus améliorable, à cause des horaires, il est nécessaire de parcourir au moins chaque arrêt par étape et le considérer si nous y arrivons en un temps inférieur à ce qui a déjà été fait.

Optimisation de conception

📍 Indexation 2sphere

Notons que notre `dataset` possède plus de 300 000 arrêts dans toute la France. Exploiter les fonctionnalités de l'indexation `2dsphere` est alors une chose qui nous fera gagner un temps monstrueux. `MongoDB` nous permet non seulement de récupérer les données contenues dans un disque mais aussi dans un polygone. Dans notre cas, il est malin de récupérer les arrêts contenus dans une ellipse autour des points de départ et d'arrivée. Cela va presque diviser par deux notre temps de calcul sans endommager sa pertinence. C'est le genre d'optimisations cruciales que nous sommes amenés à effectuer en `Big Data`.

Pour minimiser le nombre d'arrêts considérés en fonction du point de départ et d'arrivée, il faut ruser et prendre les candidats les plus adéquats, pour cela j'utilise la formule de l'ellipse pour faire une interpolation discrète de l'ellipse dont les deux centres sont le point d'arrivée et le point de départ. La taille du grand axe sera 2 fois la distance entre les deux points et la taille du petit axe sera la distance en elle même. Supposons alors que les arrêts concernés ne pourront pas s'éloigner perpendiculairement du trajet à vol d'oiseau d'une distance supérieure à la moitié de la distance. Pour rappel, voici la formule d'une ellipse :

$$\frac{((x - h) \cdot \cos(\theta) + (y - k) \cdot \sin(\theta))^2}{a^2} + \frac{(-(x - h) \cdot \sin(\theta) + (y - k) \cdot \cos(\theta))^2}{b^2} = 1$$

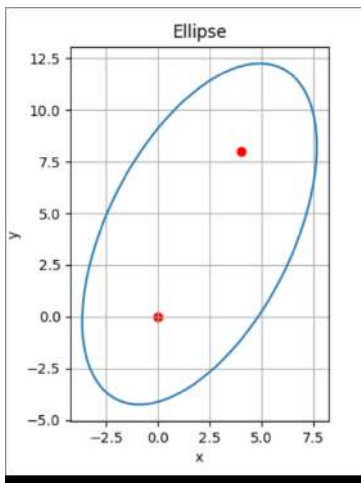
Avec :

- θ qui est la rotation de l'ellipse. On désire aligner cette rotation suivant nos deux points.
- (h, k) le centre de l'ellipse. On désire avoir le milieu du segment formé par le point de départ et le point d'arrivée.
- a et b sont respectivement la moitié de la longueur du grand axe et du petit axe.
- Les points (x, y) qui vérifient cette équations sont donc les points de notre ellipse.

Malheureusement, je n'ai trouvé aucune ressource sur internet pour répondre à ce besoin spécifique, j'ai dû faire l'interpolation discrète moi même en partant de la formule mathématique :

```
import numpy as np

def ellipse(center1, center2, nb):
    # Centres de l'ellipse
    center_x = center1[0]
    center_y = center1[1]
    center_x2 = center2[0]
    center_y2 = center2[1]
    # Distance entre les deux centres
    distance = np.sqrt((center_x2 - center_x) ** 2 + (center_y2 - center_y) ** 2)
    # Centre de l'ellipse est le milieu de nos deux points
    h, k = (center_x + center_x2) / 2, (center_y + center_y2) / 2
    # Longueur du grand axe et du petit axe
    a = distance # la moitié de la distance de chaque côté de nos points.
    b = distance/2 # On peut s'éloigner perpendiculairement du vol d'oiseau à un maximum de la moitié de la distance entre les deux points
    # Rotation de l'ellipse pour s'orienter selon l'axe des points de départ et d'arrivée
    rot = np.arctan2(center_y2 - center_y, center_x2 - center_x)
    # Calcul des points de l'ellipse selon sa formule mathématique
    theta = np.linspace(0, 2*np.pi, nb)
    x = h + a * np.cos(theta) * np.cos(rot) - b * np.sin(theta) * np.sin(rot)
    y = k + a * np.cos(theta) * np.sin(rot) + b * np.sin(theta) * np.cos(rot)
    # On crée la liste des points au format pris par mongodb
    res = []
    for i in range(len(x)):
        res.append([x[i], y[i]])
    return res
```



Génération de notre dataset

Trouvons maintenant 5 lieux candidats avec une densité d'arrêts assez variée tout en restant sur une complexité raisonnable. J'ai fait un script Python pour pouvoir visualiser mes arrêts sur un rayon donné afin de choisir mes lieux plus facilement :

```
import folium
from pymongo import MongoClient

# Spécifier les informations d'identification
username = "admin"
password = "password"
# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test
collection = db.stops

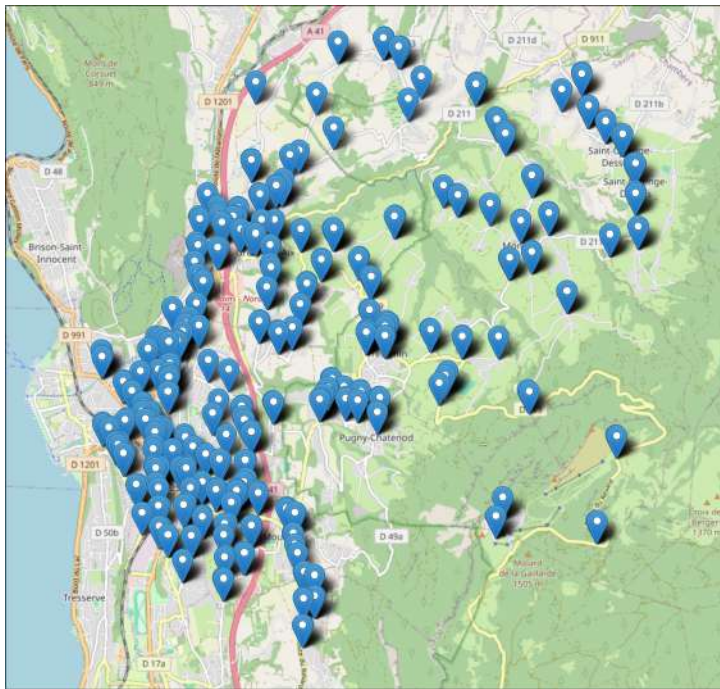
resultats = collection.find({
    "location": {
        "$near": {
            "$geometry": {
                "type": "Point",
                "coordinates": [5.961191, 45.705807] # Latitude et longitude avec l'exemple de Aix les bains
            },
            "$maxDistance": 5000 # 5 kilomètres
        }
    }
})

data = list(resultats)

# Créer une carte centrée
mymap = folium.Map(location=[data[0]['stop_lat'], data[0]['stop_lon']], zoom_start=12)

# Ajouter des marqueurs pour chaque élément dans les données
for point in data:
    folium.Marker([point['stop_lat'], point['stop_lon']], popup=point['stop_name']).add_to(mymap)

# Sauvegarder la carte dans un fichier HTML
mymap.save('map.html')
```



Générons maintenant en utilisant la puissance de nos 20 cœurs de processeur les données produites par l'algorithme de `pathfinding` sur couples aléatoires de points de départ et d'arrivée pour chacun des lieux choisis (ils ont tous été choisis pour avoir entre 150 et 600 arrêts de bus sur des zones de rayons de 1 à 2km soit entre 3 et 12.5km²) :

```
from math import radians, sin, cos, sqrt, atan2
import osmnx as ox
import dijkstra as dij
import random
import multiprocessing as mp
import pf_custom_silent as cust
import get_stops
import json
import os
import sys

# Réunir tous les fichiers json du dossier spécifié en un seul.
def merge_jsons(folder):
    res = []
    for fichier in os.listdir(folder):
        if fichier.endswith(".json"):
            chemin_fichier = os.path.join(folder, fichier)
            with open(chemin_fichier, "r") as file:
                data = json.load(file)
                res.append(data)
    res_file = "result.json"
    with open(res_file, "w") as new_file:
        json.dump(res, new_file, indent=4)

# Pour réunir tous mes json, je décommente ces deux lignes :
# merge_jsons('data')
# sys.exit(0)

nb_chemin = 100 # Nombre de chemins à calculer par lieu

locations = [
    [(45.705807, 5.961191), 2000], # Aix les bains
    [(46.113739, 5.820108), 2000], # Bellegarde sur Valserine
    [(43.330422, 5.424058), 1200], # Marseille nord
    [(43.279111, 5.393982), 1200], # Marseille sud
```

```

[(43.524859, 5.443268), 1000] # Aix en Provence
]

def compute_pf(G):
    # On sélectionne deux points au hasard
    start, end = get_start_goal_nodes(G)
    start = (G.nodes[start]['y'], G.nodes[start]['x'])
    end = (G.nodes[end]['y'], G.nodes[end]['x'])
    # Récupérer les arrêts de manière optimisée
    stops = get_stops.getStops((start[1], start[0]), (end[1], end[0]))
    # On approxime le rayon pour le calcul de densité des arrêts à la distance entre le point d'arrivée et de départ
    radius = int(haversine_distance(start[1], start[0], end[1], end[0]) * 1.5)
    # Calculer le chemin avec l'algorithme de pathfinding
    print(f"Calcul avec {len(stops)} arrêts..")
    path = cust.pf(start, end, '11:20:00', stops, radius, G)
    result = path.pathf()
    print(result)
    print("Calcul terminé !")
    # enregistrer le résultat dans un json
    with open(f"data/part-{str(random.randint(1, 999999)).zfill(6)}.json", "w") as fichier:
        json.dump(result, fichier)

def haversine_distance(lat1, lon1, lat2, lon2):
    R = 6371000 # Rayon de la Terre en mètres
    # Conversion des degrés en radians
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    # Calcul des écarts en latitudes et longitudes
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    # Formule de la distance haversine
    a = sin(dlat / 2) ** 2 + cos(lat1) * cos(lat2) * sin(dlon / 2) ** 2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c
    return distance

# Je choisis deux noeuds aléatoires sur le graphe donné
def get_start_goal_nodes(G):
    nodes = list(G.nodes)
    start = random.choice(nodes)
    goal = random.choice(nodes)
    while start == goal:
        goal = random.choice(nodes)
    return start, goal

# On boucle sur tous les lieux
for location in locations:
    # On calcule le graphe pour le lieu
    G = ox.graph_from_point(location[0], location[1], network_type='walk', simplify=True)
    # On calcule "nb_chemin" chemins en parallèle
    with mp.Pool() as p:
        p.starmap(compute_pf, [(G,) for i in range(nb_chemin)])

```

Nettoyage des données

Nous pouvons maintenant nourrir notre collection avec ces données :

```

docker cp result.json mongo:/root/
docker exec -it mongo bash
cd /root
# On importe notre json dans la collection "paths"

```

```
mongoimport --authenticationMechanism SCRAM-SHA-256 --username admin --password password --authenticationDatabase admin --jsonArray --collection
paths --file result.json
# On créé l'index 2dsphere sur le champ de départ
mongosh
use admin
db.auth("admin", "password")
use test
db.paths.createIndex({ "depart": "2dsphere" })
```

Nous allons apporter quelques modifications à nos données pour les nettoyer, faisons alors un backup de notre collection :

```
from pymongo import MongoClient
from bson.json_util import dumps

# Spécifier les informations d'identification
username = "admin"
password = "password"

# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test
collection = db.paths
```



```

with open('backup_paths.json', 'w') as file:
    i = 0
    for doc in collection.find():
        i += 1
        print(i)
        doc_str = dumps(doc) # Convertir ObjectId en chaînes de caractères
        file.write(doc_str + '\n')
    print("Backup de la collection terminé avec succès!")

```

Maintenant que nous avons récupéré toutes ces données il faut les nettoyer les incohérences au niveau des `ratios` (`ratios` trop gros et trop petits pour être réalistes, c'est souvent dû au chemin qui est trop court pour que la comparaison soit pertinente) :

```

from pymongo import MongoClient

# Spécifier les informations d'identification
username = "admin"
password = "password"
# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test

# La requête mongodb pour virer les occurences avec un ratio trop faible et trop fort
db.paths.delete_many(
    {
        "$or": [
            {"ratio_car_transport": {"$gt": 10}},
            {"ratio_car_transport": {"$lt": 0.1}}
        ]
    }
)

locations = [
    [(45.705807, 5.961191), 5000, 'Aix les bains' ],
    [(46.113739, 5.820108), 5000, 'Bellegarde sur Valserine'],
    [(43.330422, 5.424058), 2500, 'Marseille nord'],
    [(43.279111, 5.393982), 2500, 'Marseille sud'],
    [(43.524859, 5.443268), 5000, 'Aix en Provence']
]

for location in locations:
    # Requête mongodb pour assigner la ville à chacun des éléments en fonction de leur position géographique
    cursor = db.paths.find(
        {
            "depart": {
                "$near": {
                    "$geometry": {
                        "type": "Point",
                        "coordinates": [location[0][0], location[0][1]]
                    },
                    "$maxDistance": location[1]
                }
            }
        }
    )
    for doc in cursor:
        db.paths.update_one(
            {"_id": doc["_id"]},
            {"$set": {"zone": location[2]}}
        )

```

Pour vérifier que tous mes documents ont eu leur zone d'ajoutée, je suis rentré dans la console `mongosh` :

```
mongosh
use admin
db.auth("admin", "password")
use test
# La commande m'a bien rendu 0
db.paths.count({zone: {$exists: false}})
```

📄 Bilan de nettoyage

Nous sommes passés de 249 documents à 238 en triant les `ratios` les moins pertinents. Nous avons également ajouté une colonne de zone pour chacun des documents en fonction de leur point de départ.

Visualisation des données

⚠️ Pertinence statistique

Les statistiques visualisées dans cette partie sont issues des données générées à l'aide de paramètres réels mais aussi de paramètres aléatoires. Ces données ne sont alors pas représentatives de la réalité mais ne sont qu'un échantillon. Vu que le modèle est assez simple mais que la problématique aussi puisqu'elle ne s'encombre pas de conclusions pouvant être nuancées par des biais du comportement humain, les données restent fiables au vu de la modestie des conclusions apportées. Cependant, il faut garder à l'esprit qu'un échantillon n'est jamais une représentation parfaite et que même si l'on écarte ces biais, il reste un intervalle de confiance à considérer. Pour la confiance à 95%, voici la formule :

$$IC = \bar{x} \pm 1.96 \left(\frac{s}{\sqrt{n}} \right)$$

- \bar{x} est la moyenne des valeurs générées.
- s est l'écart type des valeurs. (la manière dont elles sont distribuées)
- n est le nombre de valeurs aléatoires générées.

Par exemple lorsque nous considérons la densité moyenne d'arrêts récoltés sur une zone donnée, cet intervalle est à calculer pour estimer la fiabilité de cette donnée. En effet, dans notre cas, nous générons plusieurs ellipses et elles ne sont pas nécessairement représentatives de la densité sur toute la zone.

Les données qui ont été générées sont assez particulières et ne peuvent être interprétées que de peu de manières différentes. Si nous avions voulu augmenter sa pertinence, il aurait fallu calculer plusieurs autres variables dépendant de chaque chemin, mais aussi des lieux concernés ainsi que des transports concernés. Cependant, pour des raisons de temps mais aussi de puissance de calcul, je n'ai pas pu me permettre de faire une collecte trop riche. Cependant, voici quelques visualisations que nous pouvons mettre en oeuvre :

- Densité d'horaires par ville.
- Diagramme en bâton de la proportion de temps passée en voiture par rapport au transport pour les mêmes trajets, par ville.
- Nuage de point de la densité d'arrêts en fonction du `ratio` de temps `voiture/transport`, coloré par ville.

Pour réaliser cela, j'utilise les requêtes `mongodb` ainsi que `pandas` et `plotly` :

```
from pymongo import MongoClient
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Spécifier les informations d'identification
username = "admin"
```

```

password = "password"
# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test

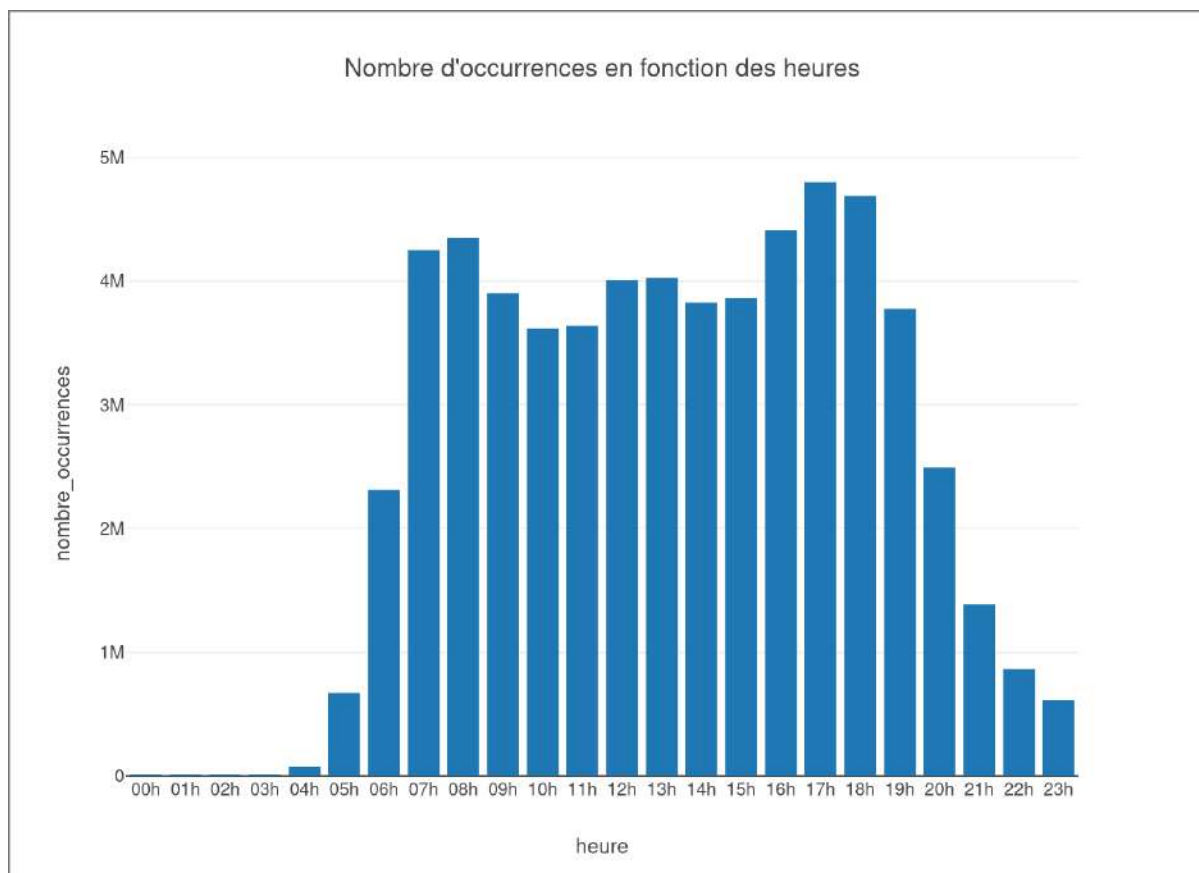
# Unwind les horaires des arrêts et les grouper par "time"
pipeline = [
    {'$unwind': '$times'},
    {
        '$project': {
            'hour': {'$regexFind': {'input': '$times.arrival_time', 'regex': '^(\d{2}):'}}
        }
    },
    {
        '$group': {
            '_id': '$hour.match',
            'total_times': {'$sum': 1}
        }
    },
    {'$sort': {'_id': 1}}
]

results = db.stops.aggregate(pipeline)

df = pd.DataFrame(list(results))
df.rename(columns={'_id': 'heure', 'total_times': 'nombre_occurrences'}, inplace=True)
df['heure'] = df['heure'].str.replace(':', 'h')

# Création du graphique avec Plotly
fig = px.bar(df, x='heure', y='nombre_occurrences', title='Nombre d\'occurrences en fonction des heures', template='none')
fig.update_layout(height=600, width=800)
fig.show()

```



```

from pymongo import MongoClient
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

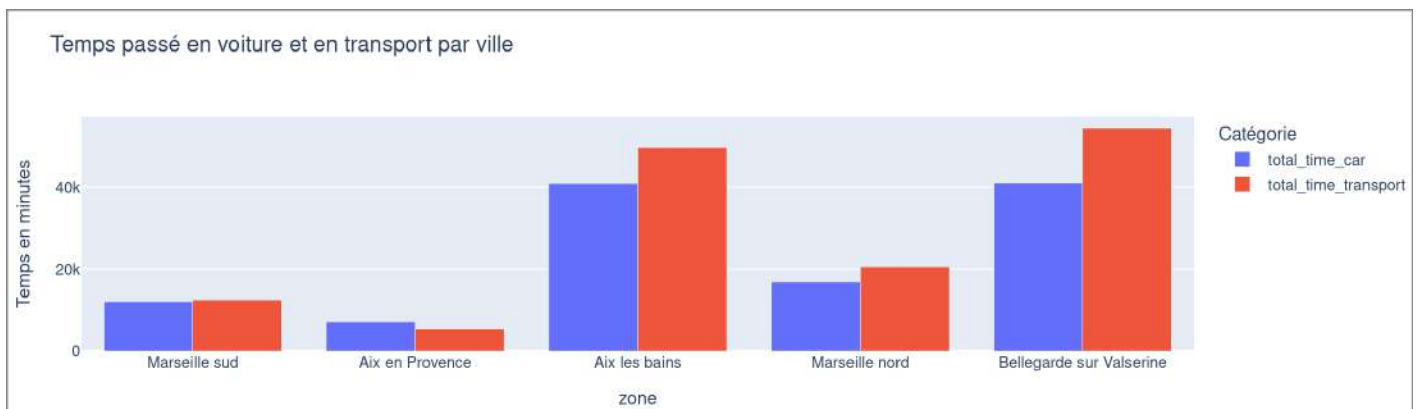
# Spécifier les informations d'identification
username = "admin"
password = "password"
# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test

# Je récupère les résultats groupés par zone avec la somme de chacun des temps (voiture et transport)
# J'applique une projection pour ne récupérer que ces 3 colonnes
pipeline = [
    {
        '$group': {
            '_id': '$zone',
            'total_time_car': {'$sum': '$time_car'},
            'total_time_transport': {'$sum': '$time_transport'}
        }
    },
    {
        '$project': {
            '_id': 0,
            'zone': '$_id',
            'total_time_car': 1,
            'total_time_transport': 1
        }
    }
]

results = db.paths.aggregate(pipeline)
df = pd.DataFrame(list(results))
df
# Création d'un DataFrame empilé pour le graphique
df_stacked = df.set_index('zone').stack().reset_index(name='value').rename(columns={'level_1': 'category'})

# Création du graphique empilé avec Plotly
fig = px.bar(df_stacked, x='zone', y='value', color='category', barmode='group', title='Temps passé en voiture et en transport par ville', labels={'value': 'Temps en minutes', 'category': 'Catégorie'})
fig.show()

```



```

from pymongo import MongoClient
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Spécifier les informations d'identification
username = "admin"
password = "password"

# Se connecter à la base de données MongoDB en incluant l'authentification
client = MongoClient('localhost', 27017, username=username, password=password)
db = client.test

results = db.paths.find({}, {'stops_density': 1, 'ratio_car_transport': 1, 'zone': 1})
df = pd.DataFrame(list(results))
df

# Création du nuage de points avec Plotly
fig = px.scatter(df, x='ratio_car_transport', y='stops_density', color='zone', title='Densité d'arrêts en fonction du ratio voiture/transport', labels=
{'ratio_car_transport': 'Ratio Voiture/Transport', 'stops_density': 'Densité d'arrêts'})
fig.show()

```



🔗 Plotly et HTML5

J'ai choisi `plotly` car il s'intègre très bien en `HTML5`. Je vais donc pouvoir le télécharger et faire une page `HTML` rapidement pour des présentations plus esthétiques. Voici la page `HTML` affichant ces trois graphiques dynamiques (utilisant `JavaScript`) :

```

from plotly.offline import plot
plot(fig, filename='graph1.html')

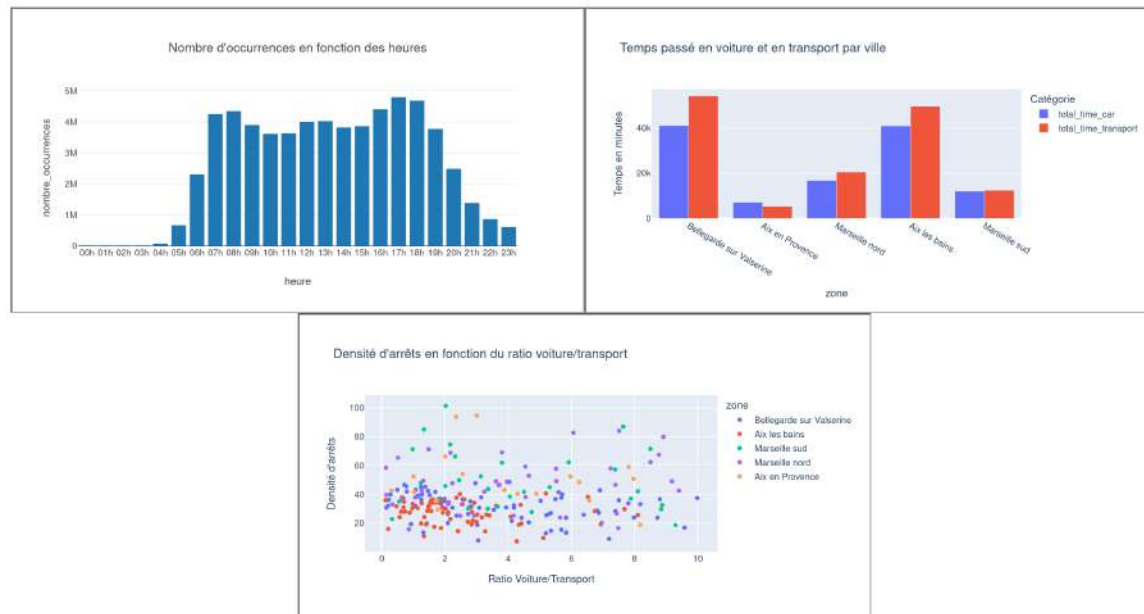
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Intégration d'un graphique Plotly</title>
  </head>
  <body>
    <h2>Statistiques de notre dataset généré</h2>
    <div style="width: 90%; display: flex; flex-wrap: wrap; justify-content:center;">
      <iframe style="width:45%; height:45vh;" src="graph1.html"></iframe>
      <iframe style="width:45%; height:45vh;" src="graph2.html"></iframe>
      <iframe style="width:45%; height:45vh;" src="graph3.html"></iframe>
    </div>
  </body>
</html>

```

Statistiques de notre dataset généré



Conclusion

Nous avons récolté une importante quantité d'arrêts de bus et de train en [Open Data](#) sur le site du gouvernement. Plus de 300 000 arrêts et 62 millions de passages de manière automatisée avec [Python](#). Nous sommes ensuite partie sur l'installation d'une infrastructure de calcul distribué avec [Spark](#) afin de nettoyer et restructurer les données. Celles-ci ont ensuite été stockées sur un serveur [MongoDB](#) afin d'être utilisées de manière optimale en se servant de l'indexation [2sphere](#) de [Mongo](#). Nous avons ensuite généré des données à partir de calculs de [pathfinding](#) à l'aide d'un algorithme que nous avons nous même créé pour répondre à notre besoin spécifique. Les données ont été générées sur une seule machine en [multi-cœurs](#), puis stockées et nettoyées à l'aide de [MongoDB](#). Pour finir, nous avons visualisé les données à l'aide de requêtes [Mongo](#) et des librairies [Pandas](#) et [Plotly](#). Voici l'interprétation que nous pouvons avoir de ces données générées :

- Le premier graphique ne nous donne rien de bien intéressant mis à part le fait évident que les horaires sont surtout réparties la journée et que les passages commencent le matin avec une quantité importante dès le début, tandis que le soir, le nombre de passages diminue progressivement.
- Le second graphique nous montre que finalement, le temps de trajet en voiture n'est pas si rapide par rapport à celui en transport en commun, il n'est qu'un peu plus rapide. Je me dois cependant d'être objectif et de rappeler que les simulations effectuées partent du principe que l'individu se plie aux exigences de l'horaire de passage du premier arrêt. Ignorant alors les contraintes horaires de la vraie vie, mais au vu du premier graphique et de la répartition des passages, il n'y a normalement pas trop de problème. De plus, il a été simulé un temps de 5 minutes pour garer la voiture et se rendre au lieu voulu. En négociant ces deux facteurs, il est possible que l'écart soit plus grand et que la voiture soit d'autant plus avantageuse.
- Le troisième graphique nous montre que pour [Aix les Bains](#), il y a moins de trajet désavantageux que pour les autres villes vis à vis des transports en commun. La ville possède peut de ratios élevés. [Marseille nord](#) en revanche semble posséder une répartition des ratios assez homogène ce qui témoigne des figures de cas avec une grande difficulté d'accès par transport en commun mais aussi de figures de cas avec une grande facilité. [Bellegarde sur Valserine](#) a une répartition qui fait penser à un mélange des deux villes précédentes. Beaucoup de ratios autour de 2 et quelques ratios supérieurs.

⚠ Nuances

Il faut garder en tête que nous avons réussi à récolter que 238 chemins en raison de la puissance de calcul nécessaire pour la collecte. Les statistiques sont alors très incertaines. C'est pour cela que j'ai appuyé l'importance de considérer un intervalle de confiance dans ce genre de cas de figure. Notons que nous aurions pu faire des graphiques comme par exemple une [heatmap](#) de notre nuage de points pour analyser leur densité de répartition afin de clarifier notre interprétation.

🕒 Optimisations possibles

Au final, pour améliorer ce projet plusieurs optimisations plus ou moins ambitieuses peuvent être réalisées :

- Fusionner les arrêts proches ainsi que les horaires.
- Evaluer de manière précise la pertinence de la zone elliptique.
- Amélioration de la qualité du `dataset` de départ.
- Calcul distribué sur plusieurs machines pour une croissance horizontale des performances.
- Création de plusieurs graphes dépendant du temps avec les horaires en tant qu'arêtes.
- Amélioration de l'algorithme de `pathfinding` de marche à pied en implémentant par exemple du bidirectionnel.
- Mise au propre de l'algorithme et optimisation des calculs.
- Ignorer les arrêts isolés de l'ellipse. (pour lesquels aucun trajets ne sont possibles dans la zone)