1. **Simplification du graphe**

Comme expliqué dans l’énoncé, toutes les stations sont reliées entre elles.

Ainsi, le graphe obtenu est un graphe complet. Ce fait pose deux problèmes :

* Tout d’abord, le chemin le plus court est d’aller directement d’un point A à un point B puisqu’ils sont forcément reliés. Ce trajet direct n’est pas possible pour une voiture électrique lorsque la distance est trop grande, donc inutile.
* Le stockage des distances se feraient dans une matrice de de dimension n\*n, avec n le nombre de stations. Si l’on utilise des entiers afin de représenter les distances, qui ont chacun une taille de 4 octets, la taille de la matrice sera de n²\*4 octets.  
  Sachant que l’on possède un dataset comportant 17000 stations, cela nous fera une matrice comportant 2 entiers, soit 1,56 Giga-octets.

Comme expliqué précédemment, il est inutile de conserver les distances entre deux stations lorsqu’elles sont trop grandes, puisqu’une voiture électrique ne pourra pas la parcourir en un seul trajet. Ainsi, nous pouvons déjà simplifier le graphe en ne gardant que les arrêtes possédant un poids inférieur à l’autonomie de la voiture.

Si l’on possède six stations, voici le graphe que l’on obtiendrait :

Une image contenant diagramme

Description générée automatiquement

Si la voiture de l’utilisateur possède une autonomie de 200km, il est inutile de conserver les arrêtes qui possèdent un poids supérieur à celle-ci.

En retirant les arrêtes comportant des distances supérieures à 200km, nous obtenons un graphe comportant bien moins d’arrêtes.

Une image contenant diagramme

Description générée automatiquement

Afin d’effectuer cette simplification, il faut tout d’abord calculer la distance entre chaque sommet. Cette opération comporte une complexité en O(n²).

Une image contenant graphique

Description générée automatiquement

Sachant que notre dataset comporte à peu près 17.000 stations, il faut à peu près 49 secondes afin de calculer les distances entre chaque station.

Effectuer ce calcul reste possible mais il doit rester très occasionnel afin de ne pas faire trop patienter l’utilisateur.

Il n’est donc pas envisageable d’effectuer cette opération à chaque fois qu’un trajet est recherché. Une possibilité est de calculer le graphe uniquement lorsque l’utilisateur renseigne son véhicule. Ainsi, la distance maximale des arrêtes peut-être définie grâce à l’autonomie de son véhicule.

1. **Structure de données**

Il faut désormais choisir la structure de données qui permettra de représenter le graphe.

Il y a ici de nombreuses possibilités, mais dont certaines sont plus optimales que d’autres, que ce soit au niveau de la mémoire ou de la vitesse.

Matrice de poids des arcs :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F |
| A | -1 | 100 | -1 | 140 | -1 | -1 |
| B | 100 | -1 | 60 | -1 | -1 | -1 |
| C | -1 | 60 | -1 | 140 | 130 | 120 |
| D | 140 | -1 | 140 | -1 | 100 | -1 |
| E | -1 | -1 | 130 | 100 | -1 | 90 |
| F | -1 | -1 | 120 | -1 | 90 | -1 |

*Les -1 correspondent aux sommets sans adjacences.*

Une matrice de poids des arcs permettrait d’accéder très rapidement à la distance entre chaque sommet, mais elle demande stocker des entiers pour chaque combinaison de sommets, même s’ils ne sont pas adjacents. Ainsi, le problème de la taille de la matrice, mentionné lors de l’explication des problèmes du graphe complet, n’est pas corrigé puisqu’elle sera aussi grande.

Tableau de Maps :

Une représentation permettant d’économiser de la mémoire peut être, pour chaque sommets, de stocker uniquement les sommets auxquels ils sont adjacents et leurs distances.

Il est possible d’obtenir cette représentation grâce à un tableau de Maps. Chaque station est représentée par un identifiant, ce qui permet de les retrouver dans le tableau. Les éléments du tableau sont des Map<int, int>, qui ont pour clé l’identifiant d’une station, et comme valeur la distance entre les deux stations.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stations | 0 (A) | 1 (B) | 2 (C) | 3 (D) | 4 (E) | 5 (F) |
| Stations adjacentes  Et distances | |  |  | | --- | --- | | 1 | 100 | | 3 | 140 | | |  |  | | --- | --- | | 0 | 100 | | 2 | 60 | | |  |  | | --- | --- | | 1 | 60 | | 3 | 140 | | 4 | 130 | | 5 | 120 | | |  |  | | --- | --- | | 0 | 140 | | 2 | 140 | | 5 | 100 | | |  |  | | --- | --- | | 2 | 130 | | 3 | 100 | |  | 90 | | |  |  | | --- | --- | | 2 | 120 | | 4 | 90 | |

Par exemple, en lisant la case colorée, on comprend que la station C (id = 2) est séparée de la station D (id = 3) de 140km.

Les deux représentations efficaces des Maps sont les hashmap et les arbres équilibrés. Voici les avantages et inconvénients des deux solutions :

|  |  |
| --- | --- |
| **Hashmap** | **Arbre équilibré** |
| Insertion et suppression en O(1), pire cas : O(n) | Insertion et suppression en O(n) |
| Recherche en O(1), pire cas : O(n) | Recherche en O(log(n)), pire cas O(log(n)) |
| Représentation mémoire plus compacte | Plus lourd en mémoire |
| Ne préserve pas l’ordre des clés | Préserve l’ordre |
| Nécessite une bonne fonction de hachage |  |
|  | Meilleur pour parcourir les clés dans l’ordre |

Ici, il n’est pas utile de parcourir les clés dans l’ordre, ou même de garder un ordre. Il y a également peu de chances que l’on se retrouve dans le pire cas pour la Hashmap (la fonction de hachage a retourné le même hash pour toutes les clés).

La Hashmap est donc bien plus efficace en terme de rapidité, ainsi qu’en terme de mémoire.

Afin de l’optimiser, il faut encore bien choisir la fonction de hachage, et la taille de la Hashmap.

Fonction de hachage :

Sachant que notre dataset comporte à peu près 17000 stations triées indépendamment de leur position géographique. Ainsi, le hash le plus simple pourrait être de prendre l’id des stations et d’en faire le modulo par la taille de la hashmap. Il faut donc bien ajuster la taille de la hashmap afin de minimiser l’impact en mémoire tout en évitant d’avoir trop de collisions entre les clés.

Taille de la HashMap :

Afin de choisir au mieux la taille de la Hashmap, il faut connaitre le nombre de stations adjacentes en moyenne.

Selon [Range of full electric vehicles cheatsheet - EV Database (ev-database.org)](https://ev-database.org/cheatsheet/range-electric-car), l’autonomie moyenne d’une voiture électrique est de 345km.

Une image contenant graphique

Description générée automatiquement

*Fonction de répartition empirique du degré des sommets pour dmax = 345km*

La moyenne du degré des stations est de 6585. Nous pouvons constater que, globalement, les stations possèdent moins de 8000 stations adjacentes si l’on prend une distance maximale pour une arrête de 345km. Ainsi, la hashmap pourrait être de cette taille, et il y aurait peu de collisions.

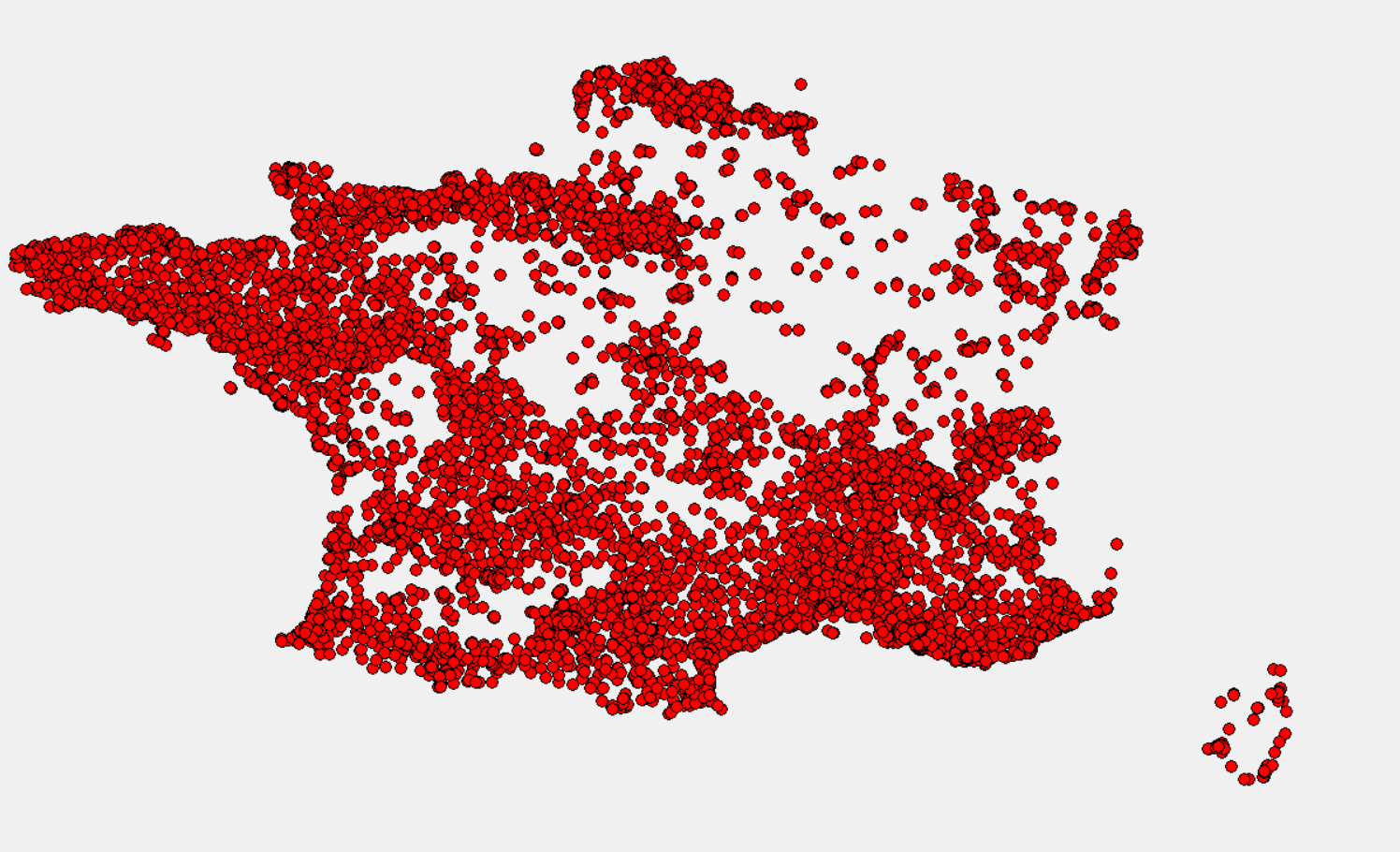
Toujours selon la même source, l’autonomie maximale d’une voiture électrique est de 700km.

Une image contenant graphique

Description générée automatiquement

*Fonction de répartition empirique du degré des sommets pour dmax = 700km*

La moyenne du degré des stations est de 15351. Ainsi, la Map n’aura pas un très grand avantage par rapport à une simple matrice du point de vue de la mémoire. Il est possible de ne pas prendre un dmax trop grand afin de limiter l’impact sur la mémoire, mais cela fera perdre de l’information et ne permettra pas à une voiture ayant une grande autonomie d’effectuer directement le trajet entre deux bornes espacées de 700km.



*Répartitions des bornes sur la france*

1. **Algorithmes de recherche de chemin**

**TODO :** Dijsktra, A\*

**Algorithme de Dijkstra**

Afin de déterminer le plus court chemin entre deux points nous avons implémenté un algorithme de Dijkstra. Cependant, de manière à ne pas avoir besoin de stocker toutes les distances entre chaque point dans une matrice nous avons décidé de calculer les sommets adjacents à chaque itération de l’algorithme.

Les sommets adjacents sont tous les sommets atteignables avec une autonomie donnée.

De ce fait, nous économisons de l’espace mémoire mais en contrefaçon, le temps de calcul d’un trajet est prolongé.

Dans la première version de l’algorithme, voici les résultats obtenus :

|  |
| --- |
| 13042 --> 16405 : 258 km, 38.344920s  9711 --> 6361 : 495 km, 37.036574s  7358 --> 4525 : 327 km, 37.613164s  14185 --> 12756 : 560 km, 37.904909s  6567 --> 6751 : 591 km, 38.116012s  8465 --> 9490 : 482 km, 37.966412s  10751 --> 781 : 237 km, 37.249523s  7034 --> 931 : 626 km, 39.729092s  6261 --> 1107 : 337 km, 38.006598s  5497 --> 5644 : 156 km, 38.065011s |

Les deux premiers nombres correspondent aux stations de départ et d’arrivée. Ils sont choisis au hasard parmi toutes les stations possibles. La distance la plus courte est donnée en km puis le temps de calcul pour trouver le chemin est donnée en seconde.

Dans cet échantillon d’essais, la moyenne est de 37.91s pour trouver le plus court chemin entre deux stations avec une autonomie moyenne de 200km.

Plus l’autonomie est faible, plus le chemin sera long ce qui implique une durée de calcul plus longue.

* Dans un premier temps nous mettons à jour tous les voisins du sommet courant
* Ensuite, le sommet avec le poids le plus petit est bloqué
* Enfin, une fois arrivé à la station finale, l’algorithme s’arrête, il suffit de remonter à l’envers les stations visitées.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Station de départ | Station 1 | Station 2 | … | Station d’arrivée |
| /\* | Maj. station adjacente 1 | Maj. station adjacente 2 | … | … |
|  | Maj. Station\* adjacente 1 |  | … | … |
|  |  |  | Maj. station adjacente n | ...\* |

\*Les cellules encadrées donnent le chemin le plus court (remontée de l’algorithme)

La première liste ne tient pas compte du temps d’arrêt dans les stations, ce qui engendre un grand nombre de stations. Il convient donc de raccourcir cette liste en ne gardant que les stations les plus éloignées atteignables par la voiture et non surchargée par le Traffic.

Ainsi, sur notre exemple nous aurions :

[1,50,26,58,125,459,215,8441,26,855,5,266,2200] 🡪 [1,58,215,855,2200]

En sautant une partie des stations, il devient plus simple de gérer un flux élevé de voitures.

Le découpage aurait ainsi pu être comme suit sans pour autant allonger la taille du trajet :

[1,50,26,58,125,459,215,8441,26,855,5,266,2200]

🡪 [1,26,459,26,266,2200]

🡪 [1,58,125,215,855,2200]

🡪 [1,26,125,8441,5,2200]