Ecole Polytechnique

Projet 442-7:

Hacker le Metro de Paris

FOUSSOUL Ayoub
AZIZE Achraf
May 21, 2019

Contents

1 Introduction

En 2017, la RATP a rendu public la base de données, en temps réel, de tout le système de transportation publique. Des milliers de données sont produites toutes les minutes, et le but de se projet est de donner sens å cette data, et surtout essayer de l'exploiter en utilisant ce qu'on a pu apprendre comme outils dans le cours Big Data INF442.

Ainsi, les questions traîtés dans ce projet sont les suivant:

- Peut-on estimer le temps d'arrivée prochain d'une ligne mieux que la RATP ?
- Peut-on estimer le trafic dans la ville en utilisant les données de bus ?
- Peut-on estimer la population des quartier d'une ville en utilisant aussi les données de bus ?
- Est-ce que les lignes de transport respectent bien leurs fréquences nominales ?

2 Résultats intermédiares

2.1 Obtenir la data de l'API

Cette partie a necessité la communication avec le "Web Service" de type SOAP de l'RATP, après une familiarisation initiale avec les modalités de communication avec un service web et avec le wsiv de l'RATP, on a décidé de generer des données en utilisant le language PHP, vous trouvrez le programme utilisé dans le fichier generator.php (où il y a qu'a changer les paramettres en bas du fichier pour changer la ligne et le format de l'output). Plusieurs problème ont été recontrés durant cette étape notament le faite de bien preparer la requette SOAP avec un bon header. les données générées sont ceux du (1er mai pour BUS 91 / RER B et 19 mai pour RER A).

2.2 Preprocessing

Afin de stocker toutes les données provenant de l'API, notre choix final a été d'utliser des fichiers 'csv' comme ceux utiliser en TD. Le format adopté est le suivant:

- La première colonne contient le moment de la journée où la data a été enregistrée, sous le format 'aaaammjjhhmm'.
- La deuxième colonne contient le nom de la station
- La troisiéme colonne contient le nombre d'arrivées prochaines dans une direction A, suivi des temps estimés d'arrivés ainsi qu'une description. Pour avoir un document statique, qui ne dépend pas de chaque station, et en se basant sur les données observées, on a décidé de fixer ce nombre à 8.
- On refait la même procedure pour les arrivées prochaines dans l'autre direction R
- Et à la fin le nombre de perturbations, fixé à 4, et leurs descriptions,

Nous avons aussi eu besoin de la structure de chaque ligne (stations

avec leurs stations précedentes dans les deux directions) ce qui a été généré dans un fichier apart "****_structure.csv".

2.3 Création de classes et d'objets adéquats

Cette partie a été la plus difficile, et la plus intéressante à imaginer. Face à la quantité considérable de données qu'il faut traiter, il est indispensable d'utiliser des objets et des classes qui nous faciliterons les taches.

La première classe est nomée Entry.
Elle représentera les entrés réelles dans la data (i.e les vrai arrivées dans les deux directions respectives).
Elle a pour attributs le nom de la station, le numéro d'arrivées dans une direction A suivi d'une liste de moments estimés d'arrivées ainsi que le numéro d'arrivées dans une autre direction suivie d'une liste de moments.
On stocke les moments dans des variables de type tm, car plus facile à manipuler.

```
class Entry{
    protected:
        std::string station_;
        int numArrivals_A_; // A
        int numArrivals_R_; // R

    public:
        tm* arrivals_A_;
        tm* arrivals_R_;
 }
```

Une RawEntry hérite de Entry, et ajoute le moment de la journée où la donnée a été enregistrée, puis le nombre de perturbations suivi d'une liste de String de descriptions, mais aussi deux autres listes de String pour les descriptions des trains dans les deux directions. Cette classe representera les lignes du fichier csv tel qu'ils sont, d'où le nom RawEntry.

```
class RawEntry: public Entry{
    private:
        int numPerturbations_;
        tm moment_;
        std::string* arrivalsDescriptions_A_;
        std::string* arrivalsDescriptions_R_;
        std::string* perturbations_;
}
```

Il est temps de représenter toute les entrées correspondant à une ligne de l'RATP.
On utilise pour cela la classe dataSet.
Elle a pour attributs le nom de la ligne (Rer B, Bus 91.06C, ...), la liste des stations présentes dans cette ligne, un vecteur de RawEntry représentant les entrées nominales, ainsi qu'un vecteur de Entry représentant les arrivées réels.

```
typedef std::tuple <std::string, std::string, std::string> station; //
    (previous station A, station, previous station R) [station = "" if
    does not exist]

class Dataset {
    private:
         std::string line_;
         std::vector<station> stations_; // vector to avoid reding the
             file two
         std::vector<RawEntry> nominalData_;
         std::vector<Entry> realData_;
 }
```

Afin de trouver le chemin optimal entre deux stations, il est necéssaire de combiner différentes lignes (sinon ca n'as pas trop de sens d'utiliser une seule ligne pour calculer un chemin car il y aura jamais de raccourcis) . C'est pour cette raison qu'on implémente la classe Network. Elle contient comme attributs differentes lignes ainsi que les dataSets correspendant à chaque ligne.

```
class Network{
     private:
         std::vector<std::string> lines_;
         std::unordered_map<std::string, Dataset> datasets_;
 }
```

Et finalement, afin de faciliter la recherche du meilleur chemin, on iplémente la classe Graph, qui a pour attributs Edges et Vertices. Les Vertices seront donc les différentes stations des différentes lignes, et les Edges porteront des poids qui seront les distances en minutes pour aller d'une station à l'autre en prenant en considèration le temps d'attente à chaque fois.

```
typedef std::vector<std::tuple<std::string, std::string, char, long>>
    Edges; // (target station,line,direction,distance [in minutes])

class Graph {
     private:
          std::set<std::string> vertices_;
          std::unordered_map<std::string,Edges> edges_;
 }
```

Tous ces structures ont une methode print() qui permet d'afficher l'objet de facon concrète que vous pouvez tester.

3 Réponses aux questions

Avant de répondre aux questions, il est d'abord intéressants d'ajouter des méthodes dans differents classe qui permettent de faciliter le résultat final.

Pour les classes Entry et RawEntry, on ajoute des méthodes "getters" ou de les afficher (print).

De même pour DataSet, sauf pour:

```
void fillRealData();
```

qui est très important, car permet de remplir le vecteur realData_ avec les Entry de data réels.
L'idée est d'aller chercher dans les instants nominaux si le train est à quai, sinon on regarde minute par minute (et on met à jour l'heure

d'arrivé tant qu'on est encore avant l'instant d'arrivé) jusqu'à ce que le train soit à quai oubien que le temps éstimé saute d'une valeure importante ce qui signifie que le prochain train est arrivé.

Sur la classe Network, on implémente plusieurs méthodes indispensables :

```
int duration(std::string line, std::string previousStation_A,
    std::string station);
tm estimateNextArrivalTime(std::string line, char dir, std::string
    station, tm now);
void compareToNominalFrequencices();
void drawEstimatedCityTraffic();
```

La première permet de calculer la durée en minutes dans un trajet d'une station à la suivante, et renvoie 24h si c'est pas possible (Trains sans arrêt pendant tout la journée / pas d'informations sur les arrivées réels .. ect). Ce qui est fait est de regarder l'instant d'arrivé reel dans la station d'avant puis on regarde l'instant le plus proche et plus grand que cet instant dans les arrivées reels de la station en cours, ceci representera le temps pris par le train pour parcourir la distance entre les deux stations, la fonction retourne la moyenne de toutes ces durées sur une journée.

La deuxième prend en parametre la ligne, la direction, la station et le temps actuel et permet de renvoyer le moment d'arrivée du prochain train. Ceci se fait en se basant sur les temps d'arrivée réels du dernier jour (on cherche le plus proche jour qu'on connais dans la data puis on retourne le plus proche horaire plus grand que "now" dans les arrivés réels de cette journée. (notez que les objets tm de la librairie standard sont utilisés pour representées des dates)

Pour la troisieme, on fait le parcours de chaque donnée nominale (ligne du csv) puis on prend à chaque fois la distance entre un horaire et le plus proche horaire dans les arrivées réels, et on fait la moyenne sur tout la donnée norminale (RawEntry), ceci representera en quelque sorte la distance entre le vecteur Entry et RawEntry d'une station, on classe ceci par station et on fait la moyenne sur toute la journée, ceci donne une idée sur l'errure (en norme absolue utilisée ici) entre les

arrivées nominaux et réels d'une station donnée et d'une ligne donnée.

Pour la quatrieme, celle ci est uniquement utilisée pour les bus. Ce qui est fait est de prendre comme précedament l'erreur dans chaque station entre les arrivées réels et nominales et en fonction de ces erreur on peut savoir plus au moins comment est le traffic entre tout deux stations de la ligne, en effet, tout retard du bus est principalement due à un traffic élevé dans la route liant deux stations.

Sur Graph, on implémente la méthode

```
std::unordered_map<std::string, std::tuple<std::string, std::string,
    char, long>> getOptimalPath(std::string source, std::string
    target);
```

qui permet de trouver le meilleur chemin à partir d'une source et du target en minimisant evidemment la somme des edges, l'algorithme standard dijkestra est utilisé ici (vous remarquerez que l'utilisation de Network a permis d'avoir des edges appartenant à plusieurs lignes éventuellement et que si on donne à l'algorithme tout les bus / RER de paris le chemin optimal sera calculé à la sortie).

Et finalement, on implémente le tout sur main.cpp

```
int duration(Network network, string line, string previousStation_A,
    string station){
       return network.duration(line, previousStation_A, station);
}
void estimateOptimalPath(Network network, string from, string to, tm
    now){
       Graph g(network, now);
       unordered_map<string, tuple<string, string, char, long>> path =
           g.getOptimalPath(from, to);
       Graph::print_path(path, from, to);
}
void drawEstimatedCityTraffic(Network network){
       network.drawEstimatedCityTraffic();
}
void compareToNominalFrequencices(Network network){
network.compareToNominalFrequencices();
}
```

qu'on teste sur les fichiers de data de RER A, RER B et bus 91.

Pour les tests voir la fonction int main() du fichier main.cpp où vous pouvez commenter/decommenter les tests que vous voulez.

```
    int main(){

  //****** TESTING STRUCTURES *******

  //testEntry();
  //testRawEntry();
  //testGraph();
  //testDataset();
  //testNetwork();

  //****** TESTING FUNCTIONALITIES ******

  // functions estimateNextArrivalTime() and duration() are used and have
     been test manualy in Graph.cpp but you can test them using the
     following network.

  //city traffic
  unordered_map<string, pair<string, string>> files;
  files["RER B"] = make_pair("RER_B_data.csv","RER_B_structure.csv");
  files["RER A"] = make_pair("RER_A_data.csv","RER_A_structure.csv");
  files["BUS 91"] = make_pair("BUS_91_data.csv","BUS_91_structure.csv");
  Network net(files);
  drawEstimatedCityTraffic(net);
  //Compare to nominal frequencies
  compareToNominalFrequencices(net);
  //path calculation
  time_t now = time(0);
  estimateOptimalPath(net, "Sartrouville", "Denfert Rochereau",
     *localtime(&now));

}
```

L'utilisation de main a été necessaire car sinon Graph incluera Network et Network incluera Graph pour le calcul du chemin ce qui donne une "duplicate symbol error".

4 Difficultés rencontrées

- Des connaissances maigres en API
- Beaucoup de temps pour implémenter la bonne structure de données ainsi que les objets où elles seront stockées.

- Incapacité d'utiliser les codes de Machine Learning vue en cours (clustring, kd-tree etc) dans les questions traités. - Sujet très ouvert ce qui le rend plus compliqué et intéressant à la fois. - et SURTOUT toute fonction / methode implémentée a donné naissance à beaucoup de cas à traiter vu la non uniformité des données colléctées.

5 Conclusion

Ce projet nous a permis de confronter les vrai problèmes que rencontrent un Data Analyst, et de bien concevoir la différence évidente entre la partie académique des algorithmes utilisés, qui sont normalement utilisés dans différents applications du cours et qui est assez claire conceptuellement, et entre la partie pratique qui est moins idéal et plus compliqué.
Ceci dit, notre programme, et en utilisant des algorithmes plus ou moins basiques, permet déjà d'estimer le schedule des trains mieux que la RATP, determine le meilleur chemin pour aller d'une station à une autre, compare les fréquences nominales et réels et permet d'estimer aussi le trafic dans la ville et qu'il est de plus en plus meilleur quand la data à l'entrée devient plus grande. Il est aussi à noté que la facon dont nos fonctions sont implémentés font d'elles une sorte d'online algorithme qui gagne en précision à chaque fois on ajoute des données aux fichers "***_data.csv" .