Transport Urbain

Projet IPI

Pierre CABEZA
Hanaa DIDI
Adrien DONG
Omar HARBI
Paul IYABI
Wilson JOUËT
Nabil MHENNI

Lundi 10 Mars 2008

Groupe 2.1.1
M. Tarhini

Sommaire

I. Ce que doit faire le système

On est amené à réaliser un logiciel permettant à un utilisateur donné de consulter un itinéraire entre deux stations d'un réseau de transport défini préalablement. Cet itinéraire doit être le plus court possible, en terme de correspondances et/ou de distance, selon le choix de l'utilisateur. On se limitera à deux changements de ligne au maximum pour un itinéraire.

Il faut également noter que l'interconnexion entre le point de départ A et celui d'arrivée B peut être interrompue. Les distances au terminus ne sont que des approximations. On peut aussi ajouter comme option d'utilisation de choisir uniquement les correspondances entre métros ou entre bus.

L'utilisateur pouvant se trompant lors de la saisie des stations, il est indispensable de prévoir un module permettant d'identifier et de corriger les fautes de frappes. Entre autre, il doit pouvoir proposer une ou plusieurs corrections lorsqu'on oublie ou rajoute des lettres (au maximum 2), ou bien qu'on en inverse.

Exemple : Saint-Micel → Saint-michel
Luxembourgh → Luxembourg
Gare de Loyn → Gare de Lyon

Donner la station de départ : Gare de Lyon
Donner la station d'arrivée : Saint-Michel

Chemin (non direct) le plus court :

Gare de Lyon ➔ Châtelet Métro 14
Châtelet ➔ Saint-Michel Métro 4

II. Ses grandes fonctionnalités

1. Initialisation et remplissage de la base de données

En premier lieu, on a dû choisir 3 lignes de métro et 2 lignes de bus dans le plan des transports en commun parisiens, à savoir les lignes 5, 11 et 14 pour le métro, et les lignes 26 et 65 pour les bus.

Deux solutions s'offrent à nous pour stocker les informations concernant les différentes stations (nom, numéro(s) de ligne, distance au terminus) : soit utiliser un fichier texte, soit utiliser une base de données SQL.

La deuxième solution nous apparaît plus appropriée pour la mise en place du projet. En effet, l'initialisation par un fichier texte peut poser des problèmes de mémoire lorsque le nombre de stations augmente, et l'utilisation de la fonction *scanf* sur celui-ci nous oblige à avoir un texte bien formaté car la moindre erreur peut entraîner le non fonctionnement du programme (si l'on change les lignes par exemple).

2. Correction des fautes de frappe

a. Idée générale

L'idée est de mettre en place une correction intuitive qui va proposer le nom de station le plus proche de celui entrée par l'utilisateur, lorsque celui-ci se trompe. S'il y en a plusieurs, on en fait une liste qu'on affiche.

Pour cela, il suffit de comparer le nombre d'opérations à faire pour passer du mot tapé par l'utilisateur à l'ensemble des mots qui se trouvent dans la base de données : ce nombre s'appelle le coût. Il se calcule grâce à la distance de Levenstein.

On considèrera que la marge d'erreur du coût pour un nom erroné est de 4, mais cette valeur peut être amenée à changer.

b. La distance de Levenstein

La distance de Levenshtein mesure la similarité entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut *supprimer, insérer*, ou *remplacer* pour passer d'une chaîne à l'autre. Cette distance est donc d'autant plus grande que le nombre de différences entre les deux chaînes est grand.

Son algorithme général peut se résumer ainsi :

```
entier DistanceDeLevenshtein(caractere chaine1[1..longueurChaine1], caractere
chaine2[1..longueurChaine2])
   // d est un tableau de longueurChaine1+1 rangées et longueurChaine2+1
colonnes
   declarer entier d[0..longueurChaine1, 0..longueurChaine2]
   // i et j itèrent sur chaine1 et chaine2
   declarer entier i, j, coût

   pour i de 0 à longueurChaine1
       d[i, 0] := i
   pour j de 0 à longueurChaine2
       d[0, j] := j

   pour i de 1 à longueurChaine1
       pour j de 1 à longueurChaine2
           si chaine1[i] = chaine2[j] alors coût := 0
                                      sinon coût := 1
           d[i, j] := minimum(
                                      d[i-1, j  ] + 1,     // effacement
                                      d[i , j-1] + 1,     // insertion
                                      d[i-1, j-1] + coût   // substitution
                             )

   retourner d[longueurChaine1, longueurChaine2]
```

La chaîne chaine1 est de longueur longueurChaine1 et chaine2, de longueur longueurChaine2. Cet algorithme renvoie un entier positif ou nul. Il renvoie 0 si les chaînes 1 et 2 sont égales. Si les chaînes 1 et 2 sont très différentes, la fonction renverra au maximum la plus grande longueur des deux chaînes.

3. Appartenance des stations de départ et d'arrivée aux lignes

Les deux stations étant données, on fait les combinaisons de tous les chemins possibles. On commence par faire un produit cartésien des différentes correspondances des deux stations.

Chaque combinaison de deux lignes nécessite un passage par une correspondance (on doit avoir un maximum de deux correspondances). La procédure pour trouver le chemin le plus court est expliquée dans l'exemple suivant.

Exemple:

Départ	Arrivée
1	3
2	6

Dans la station de départ on trouve les lignes 1 et 2 et dans la station d'arrivée les lignes 3 et 6.

Les deux stations n'ayant pas de ligne en commun on passera forcément par au moins une correspondance. Pour cela on fait le produit cartésien des

différentes lignes. On obtient sur cet exemple les combinaisons suivantes.

1	3
1	6
2	3
2	6

Ensuite pour chacune des deux lignes on va récupérer les correspondances possibles et on rejette les solutions à plus de deux correspondances.

Sur le premier cas de l'exemple précèdent on trouve :

1 → 3
1 → 5 →3
1 → 7 → 3

Trois chemins sont donc possibles pour aller de la ligne 1 à la ligne 3. Pour le premier cas, les lignes 1 et 3 se croisent à une station (donc une seule correspondance) et pour les deux autres chemins possibles, il faut passer par une autre ligne soit deux correspondance.

On réitère cette opération pour les 3 autres cas de l'exemple.

4. Choix du chemin le plus court

On a comme données la distance de chaque station par rapport au terminus. On commence donc par faire une fonction qui retourne la distance entre deux stations qui sont sur une même ligne. Ensuite on fait la même chose pour les stations intermédiaires puis on additionne le tout pour avoir la distance totale d'un chemin. Pour avoir la distance en terme de correspondances, il suffit de rajouter un compteur sur les stations dans la base de données.

Ainsi, pour avoir le chemin le plus court en terme de distance, il suffit de prendre le minimum des distances obtenues précédemment. Quant au chemin le plus court au niveau des correspondances, on prend tout simplement le minimum des différents compteurs obtenus pour chaque chemin.

III. Manuel d'utilisation préliminaire

1. Présentation de l'application et fonctionnement

Le logiciel que l'on est amené à réaliser a pour but d'aider les voyageurs utilisant les moyens de transport à obtenir un itinéraire qui soit le plus court en terme de nombre de correspondances et/ou de temps de parcourt.

On étudie la possibilité de développer une interface graphique réalisée à travers un langage objet comme C++, Java ou GTK pour une utilisation sous Unix.

- **Etape 1**

L'utilisateur est amené à saisir la station de départ et celle d'arrivée. Ces deux chaînes de caractères doivent être saisies dans l'ordre : départ puis arrivée. Si on opte pour une interface graphique, on prendra soin de créer deux champs : une fenêtre de saisie assez classique avec un champ pour la station de départ et une autre pour l'arrivée, ainsi qu'un bouton *submit* pour valider notre choix.

- **Etape 2**

Avant d'effectuer la recherche du chemin, le logiciel devra vérifier si les deux stations saisies par l'utilisateur sont correctes et existent bien dans la base de données composée des stations du réseau de transport (métro et bus).

Si au moins une des deux chaînes de caractère n'est pas valide, le logiciel devra proposer une correction et donner un nom de station se rapprochant le plus possible de la station saisie par l'utilisateur qui, par la suite, devra valider cette correction si elle lui convient.

Le logiciel peut également proposer une liste de correction. Dans ce cas, il devra choisir l'une d'entre elles.

- **Etape 3**

Si l'étape 2 est validée, le logiciel proposera un chemin servant d'itinéraire reliant la station de départ et la station d'arrivée. Le chemin donnera les différentes correspondances qu'on devra emprunter pour atteindre la destination, ainsi que le numéro de la ligne utilisée. Pour la solution graphique, on pourra s'inspirer du site www.ratp.fr ou www.transilien.fr.

2. Dépannage / Assistance

Pour tout problème, veuillez contacter les responsables du projet à l'adresse mail suivante : transport.urbain@gmail.com.

IV. Annexe

Voici l'ensemble des stations de métro utilisées dans ce projet. Elles sont ici présentées sous la forme d'un fichier texte, qu'il faudra envoyer vers une base de données. On procédera de la même manière pour les lignes de bus.

Nom de la station	n° de la ligne	Distance au terminus (en m)
Saint-Lazare	14	0
Madeleine	14	500
Pyramides	14	1500
Châtelet	14	2000
Gare de Lyon	14	2500
Cour St-Emilion	14	3000
Bibliothèque François Mitterand	14	3500
Olympiades	14	4000
Chatelet	11	0
Hôtel de Ville	11	500
Rambuteau	11	1000
Arts et Métiers	11	1500
République	11	2000
Goncourt	11	2500
Belleville	11	3000
Pyrénées	11	3500
Jourdain	11	4000
Places des Fêtes	11	4500
Télégraphe	11	5000
Porte des Lilas	11	5500
Mairie des Lilas	11	6000
Place d'Italie	5	0
Campo Formio	5	500
Saint-Marcel	5	1000
Quai de la Rapée	5	1500
Bastille	5	2000
Breguet Sabin	5	2500
Richard Lenoir	5	3000
Oberkampf	5	3500
République	5	4000
Jacques Bonsergent	5	4500
Gare de l'Est	5	5000
Gare du Nord	5	5500
Stalingrad	5	6000
Jaurès	5	6500
Laumière	5	7000
Ourcq	5	7500
Porte de Pantin	5	8000
Hoche	5	8500
Eglise de Pantin	5	9000