Année 2013–2014

Projet informatique (PI3) – L2

Itinéraires de métro

François Laroussinie
francoisl@liafa.univ-paris-diderot.fr

Résumé : L'objectif de ce projet est de réaliser un programme permettant de chercher des itinéraires dans un réseau de transport comme le métro.

1 Les problèmes à résoudre

Les données de départ seront les lignes de métro. A partir de ces données, le problème à résoudre sera de chercher de "bons" itinéraires pour rejoindre deux stations que fournira l'utilisateur du programme via une interface. Ici un bon itinéraire sera le plus rapide (on estimera un temps de parcours en fonction du nombre de stations et du nombre de correspondances). Dans un deuxième temps, on s'intéressera au même problème lorsque l'utilisateur peut limiter le nombre de correspondances. Le calcul de ces itinéraires peut se faire avec différents algorithmes : pour ce projet, nous en avons retenu deux et ce sont eux qu'il faudra programmer en priorité dans ce projet.

1.1 Description des lignes

La description des lignes de métro sera faite dans des fichiers "texte" en utilisant le format suivant :

```
Ligne 14
Saint-Lazare
Madeleine
Pyramides
Châtelet
Gare de Lyon
Bercy
Cour Saint-Émilion
Bibliothèque François Mitterrand
Olympiades
--
Ligne 12
Porte de la Chapelle
Marx Dormoy
Marcadet - Poissonniers
Jules Joffrin
Lamarck - Caulaincourt
...
```

Remarque : certaines lignes de métro contiennent des embranchements et parfois les stations desservies diffèrent d'une direction à l'autre... Dans un premier temps, on laissera ces aspects de coté et on se contentera de décrire des lignes "simples". Dans un deuxième temps, il est demandé de les intégrer au programme. On pourra noter les embranchements "[$\text{bloc}_1||\text{bloc}_2$]" où chaque bloc_i désigne une suite de stations. Pour les "cycles", on pourra utiliser la notation "[$\text{bloc}_1/\text{bloc}_2$]" le bloc_1 désigne la suite de stations dans le sens haut-bas, et le bloc_2 désigne la suite pour l'autre sens.

Un fichier pourra contenir plusieurs lignes différentes. Et il devra aussi être possible de répartir les lignes dans plusieurs fichiers.

On devra aussi prendre en compte les regroupements de stations permettant des correspondances (par exemple Châtelet-les-Halles avec Châtelet et les Halles ou St-Michel avec St-Michel Notre-Dame,...

Le programme à réaliser devra être capable de lire des fichiers et d'en extraire des informations pour construire une structure de données permettant les recherches d'itinéraires décrits ci-dessous.

Afin de faire des tests, un fichier contenant les lignes du métro parisien sera donné.

1.2 Algorithmes de recherche d'itinéraires

La recherche d'itinéraires repose sur un calcul de *plus courts chemins* dans un graphe valué (c'est-à-dire un graphe où chaque transition est munie d'une valeur qui correspond à la longueur ou durée de la transition). Une solution classique de ce problème est l'algorithme de Dijkstra. Pour calculer nos itinéraires, on utilisera d'abord cette méthode que l'on affinera ensuite pour tenir compte des correspondances. Enfin on va considérer une dernière méthode très différente pour chercher des itinéraires pour lesquels on fixe le nombre maximum de correspondances.

1.2.1 Algorithme de plus court chemin : Dijkstra

Dans un premier temps, on demande de programmer l'algorithme de Dijkstra pour un graphe valué G et deux sommets s et s' de G. Cet algorithme est très classique et il existe une large documentation à son sujet, il est présenté succinctement en annexe.

1.2.2 Algorithme de recherche d'itinéraires simples

A partir des lignes de métro, construire un graphe valué dont les sommets sont les stations et où les arêtes correspondent à des étapes des lignes. On supposera que la durée d'une étape est 1min30. Appeler l'algorithme de Dijkstra sur ce graphe pour en déduire des itinéraires et des temps de parcours minimaux entre deux stations. On affichera aussi l'itinéraire à suivre.

1.2.3 Algorithme de recherche incluant les temps de correspondances

A présent, nous voulons tenir compte du temps de correspondance : on supposera que changer de ligne à une station prend 4 minutes. Déduire un nouveau graphe sur lequel on appliquera l'algorithme de Dijkstra.

1.2.4 Algorithme de recherche avec une borne sur le nombre de correspondances autorisées

Pour cette recherche on suppose que l'on dispose de deux stations s et s' et d'un entier k. Le problème est alors de trouver l'itinéraire le plus court (en temps) en utilisant au plus k correspondances.

Dans la suite on suppose qu'il y a n stations différentes que l'on désignera par les entiers $1, \ldots, n$.

Pour résoudre ce problème, nous allons utiliser l'algorithme suivant :

- Calculer une matrice Direct de dimension $n \times n$ contenant les distances minimales entre chaque station lorsqu'on utilise qu'une seule ligne (sans aucune correspondance) : $\mathsf{Direct}[\alpha, \beta]$ (avec α et β dans $\{1, \ldots, n\}$) sera la durée minimale d'un itinéraire entre la station α et la station β en suivant une ligne directe. Lorsqu'il n'y a pas de trajet possible entre α et β, on utilisera une valeur arbitraire représentant l'infini.
  Pour calculer Direct, on pourra calculer d'abord une matrice Direct_ℓ pour chaque ligne ℓ du réseau.
  Afin de pouvoir retrouver les itinéraires correspondant aux durées de la matrice Direct, il faudra indiquer dans une matrice Ligne_D les numéros de ligne correspondant : $\mathsf{Ligne}_D[\alpha, \beta]$ sera le numéro de la ligne (ou d'une ligne si il y a plusieurs possibilités) permettant d'aller directement de α à β en temps $\mathsf{Direct}[\alpha, \beta]$.
- Ensuite on calculera la matrice D_i de dimension $n \times n$ contenant les distances minimales entre chaque station lorsqu'on utilise au plus i correspondances de la manière suivante : $\mathsf{D}_0 = \mathsf{Direct}$ et

$$\mathsf{D}_{i+1}[\alpha, \beta] = \mathsf{min}\Big(\mathsf{D}_i[\alpha, \beta]\ ,\ \mathsf{min}_{\gamma \neq \alpha, \beta}\{\mathsf{D}_i[\alpha, \gamma] + \mathsf{D}_0[\gamma, \beta] + \Delta\}\Big)$$

  L'idée de cette formule est que la durée minimale entre α et β en autorisant $i+1$ correspondances est soit celle obtenue avec i correspondances, soit elle correspond à la durée minimale d'un itinéraire de α à une station γ en au plus i correspondances PLUS la durée d'un trajet direct entre γ et β PLUS le temps de correspondance Δ (ici 4 minutes)...
  Pour permettre de retrouver le détail des itinéraires, on va utiliser des matrices Via_i : $\mathsf{Via}_i[\alpha, \beta]$ sera le sommet par lequel il faut passer pour aller de α à β en temps $\mathsf{D}_i[\alpha, \beta]$ avec un trajet direct entre γ et β. On initialisera $\mathsf{Via}_0[\alpha, \beta]$ avec α puisque ce trajet se fait sans changement...

1.2.5 Statistiques sur le réseau

A partir des algorithmes de la question précédente, on ajoutera le calcul de plusieurs mesures sur le réseau :

- Nombre minimal de correspondances permettant de se rendre partout dans le réseau depuis n'importe quelle station.
- Nombre minimal de correspondances permettant de se rendre partout dans le réseau depuis n'importe quelle station en *un temps minimal*.
- Les stations les plus éloignées dans le réseau.

1.3 Interface

Le programme devra contenir une interface pour permettre à un utilisateur de :

– Lire un fichier de lignes de métro.
– Chercher un itinéraire simple.
– Chercher un itinéraire prenant en compte le temps de correspondance.
– Chercher un itinéraire permettant de borner le nombre de correspondances.
– Donner des statistiques sur le réseau.

1.4 Extensions

Une fois que le programme sera opérationnel, on pourra s'intéresser aux extensions suivantes :

– Utiliser les files de priorités de Java pour améliorer l'implémentation de l'algorithme de Dijkstra.
– Permettre la gestion de perturbations : une partie d'une ligne pourra être déclarée comme inactive et le calcul des itinéraires devra en tenir compte...
– Proposer d'autres algorithmes...

A Algorithme de Dijkstra

Cet algorithme permet de trouver les plus courts chemins depuis un sommet s dans un graphe valué $G = (S, A, w)$ avec S un ensemble de sommets, A un ensemble de transitions et w une fonction qui associe à chaque transition une valeur positive ou nulle (on note $w(s, s')$ la durée ou la longueur de la transition (s, s').

L'algorithme de Dijkstra consiste à découvrir, en partant de s, tous ses voisins en procédant par distance croissante : on cherche d'abord le plus proche, puis le deuxième plus proche, etc. Pour le premier sommet s_1 à trouver (le plus proche de s), nous savons qu'il est accessible par une seule transition (car w associe des valeurs positives ou nulle aux transitions). Le second sommet s_2 est accessible par une seule transition à partir de s, ou par deux transitions en passant par s_1. Le troisième plus proche sommet de s sera accessible par une, deux ou trois transitions (en passant par s_1 et/ou s_2)... A chaque fois, qu'on découvre un nouveau "plus proche sommet", on voir comment celui-ci permet de rapprocher s d'autres sommets du graphe.

On va utiliser un tableau $\mathsf{d}[-]$ qui donne pour chaque sommet sa distance depuis s en utilisant les sommets déjà découverts (il est facile de voir que $\mathsf{d}[s']$ correspond à une surapproximation de la distance minimale de s à s' et cette distance n'est plus une approximation lorsque s' est découvert). A chaque fois qu'on découvre un sommet s', on doit mettre à jour le tableau $\mathsf{d}[-]$ pour tenir compte des chemins qui passent par s' : il est possible que la distance entre s et s'' soit inférieure en passant par s' et dans ce cas, on aura $\mathsf{d}[s''] = \mathsf{d}[s'] + w(s', s'')$.

On utilise aussi un tableau Pred pour mémoriser le chemin découvert par l'algorithme : $\mathsf{Pred}[x]$ sera le sommet y par lequel on a découvert le sommet x, c'est-à-dire qu'il existe un plus court chemin entre s et x dont la dernière transition est (y, x).

L'algorithme 1 décrit l'algorithme de Dijkstra. Il faut savoir que cet algorithme utilise généralement une file de priorité pour gérer la recherche des sommets, la version présentée ici est donc simplifiée (et moins efficace) que le véritable algorithme de Dijkstra.

Un exemple de l'application de cet algorithme est donné à la figure 1. A gauche se trouve le graphe initiale, et à droite celui avec les distances minimales indiquées en gras (le numéro entre parenthèses indique l'ordre dans lequel les sommets ont été découverts) et les transitions $(\mathsf{Pred}[x], x)$ sont indiquées en gras.

Procédure `PCC-Dijkstra`(G, s)
`//`$G = (S, A, w)$ `: un graphe orienté, valué avec` $w : A \to \mathbb{R}_+$.
`//`$s \in S$ `: un sommet origine.`
begin
  pour chaque $u \in S$ **faire**
    $\mathsf{Pred}[u] := \mathsf{nil}$
    $\mathsf{d}[u] := \begin{cases} 0 & \text{si } u = s \\ \infty & \text{sinon} \end{cases}$
  $E := \mathsf{Ensemble}(S)$
  tant que $E \neq \emptyset$ **faire**
    Soit $u :=$ le plus sommet dans E ayant la valeur $\mathsf{d}[-]$ minimale
    Extraire u de E
    pour chaque $(u, v) \in A$ **faire**
      si $d[v] > d[u] + w(u, v)$ **alors**
        `//on met à jour` $\mathsf{d}[-]$
        $\mathsf{d}[v] := \mathsf{d}[u] + w(u, v)$
        $\mathsf{Pred}[v] := u$
  return *d*, *Pred*
end

Algorithme 1 : algorithme de Dijkstra (simplifié)

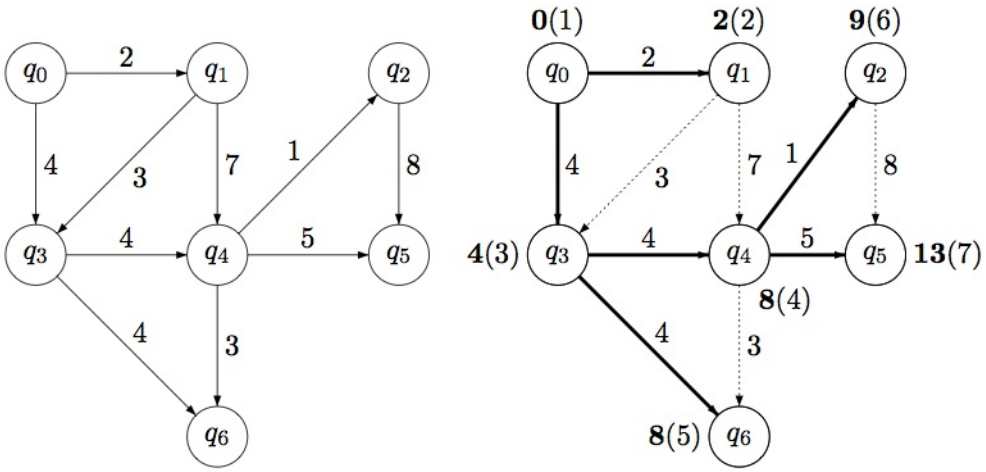

FIGURE 1 – Exemple d'application de la'lgorithme de Dijkstra