Date de rendu : 22 février 2013

PROJET DE C++
NOTE DE SYNTHESE

RECHERCHE DE TRAJET MINIMAL DANS LE METRO

Jean SANTINI - Etienne VERGNAUD

Introduction

Notre projet consiste à rechercher un trajet optimal entre deux points, ici deux stations du métro de Paris. Pour des raisons de simplicité, nous ne considèrerons que les quatre lignes 1, 4, 10 et 13 épurées de tout ce qui peut poser problème, à savoir les boucles (ligne 10) et les branches (ligne 13). Nous avons choisi ces quatre lignes car elles permettent de tester toutes les subtilités de l'algorithme que nous avons utilisé. En effet, chacune des lignes a une correspondance avec deux des autres lignes : nous pouvons donc avoir des trajets avec deux correspondances.

Notre programme utilise l'algorithme de Dijkstra, très souvent utilisé dans les problèmes de plus court chemin. Nous l'avons choisi parmi d'autres (comme l'algorithme A*) car il semblait le plus communément utilisé et relativement simple à mettre en place. Il fonctionne ainsi :

1) A l'état initial, toutes les stations sont non visitées et leur distance à l'origine (« tentative distance ») est -1 sauf pour la station de départ (la première traitée par l'algorithme) pour laquelle la distance est 0.

2) Pour la station en cours de traitement, on calcule la distance à l'origine de tous les voisins encore non visités. Si cette distance est plus petite que celle précédemment enregistrée pour ce voisin (sauf si la distance est -1), elle la remplace et la station en cours devient la « station précédente optimale » pour ce voisin.

3) La station en cours de traitement est maintenant considérée comme visitée. Si la station d'arrivée est visitée, l'algorithme s'arrête.

4) La station suivante à être traitée sera celle qui a la plus petite distance à l'origine (en dehors de -1), et on retourne à l'étape 2.

Le plan de métro correspond à un graphe. Nous avons donc créé trois classes, correspondant respectivement aux stations (« Node »), aux lignes (« Line ») et aux arêtes, c'est-à-dire les rails entre deux stations (« Edge »). A chaque station sont associées les lignes sur lesquelles elle se trouve, et les arêtes auxquelles elle est reliée. Encore une fois pour simplifier l'algorithme, nous avons choisi d'attribuer arbitrairement un poids de 1 à chaque arête, c'est-à-dire un temps de trajet d'une minute entre deux stations. Avec cette structure, nous avons toute l'information nécessaire pour que l'algorithme fonctionne.

Notre programme est composé de deux parties : l'algorithme (en incluant l'instanciation des objets) et l'interface graphique.

L'algorithme d'optimisation du trajet

Le code de la première partie peut se résumer ainsi :

1) A partir de fichiers « .csv », nous avons réalisé l'instanciation des lignes, des stations et des arêtes et des liens entre chacun de ces objets. Nous avons utilisé des « *smart pointers* » pour ne pas avoir à gérer la mémoire. Grâce aux « *smart pointers* », nous avons pu lier les objets entre eux (les lignes et les arêtes aux stations notamment) sans avoir à les copier.

2) Instanciation d'un objet « Dijkstra » qui hérite de la classe Algorithme à laquelle on donne en argument les stations de départ et d'arrivée et qui possède une fonction renvoyant le chemin optimal entre ces deux stations. Cet objet « Dijkstra » prend également en argument un temps de correspondance. Ce temps est ajouté au poids de l'arête lorsque l'on change de ligne. C'est cet argument qui nous permet de proposer des préférences de trajet : choisir le trajet le plus court est modélisé par un temps de correspondance de 1 ; choisir le trajet avec le moins de correspondance par un temps de 1000.

3) Exécution de l'algorithme en lui-même.

4) Récupération du trajet optimal : comme chaque station s'est vue attribué par l'algorithme sa « station précédente optimale », il suffit de partir de la station d'arrivée et de remonter à la station de départ pour obtenir le trajet optimal.

Néanmoins, nous nous sommes heurtés à quelques difficultés pour adapter la structure du graphe à l'algorithme.

Alors qu'il était assez évident de créer trois classes « Node », « Edge » et « Line », leur imbrication a été par contre loin d'être triviale. En effet, il s'agissait de créer des objets adaptés à l'algorithme de « Dijkstra » et qui seraient facilement manipulables. Il s'est notamment posé la question de savoir si à une station nous devions attribuer ses stations voisines ou les arêtes dans lesquelles elle était impliquée. La première solution aurait été plus pratique pour l'algorithme car celui-ci a besoin de connaître précisément les stations voisines, mais posait un inconvénient de taille : nous ne pouvions pas savoir quelle ligne était utilisée pour

rejoindre tel voisin. Nous avons donc opté pour la seconde solution, même si elle nécessitait la création d'une méthode renvoyant un vecteur des voisins d'une station.

Nous nous sommes cependant aperçus d'une faille dans l'algorithme de Dijkstra et qu'il semble difficile de contourner avec notre structure de graphe. Cette faille s'observe lorsque l'on cherche à obtenir le trajet avec le moins de correspondance entre Georges V et Vavin par exemple. L'algorithme renvoie toujours le trajet le plus court, alors qu'il fonctionne très bien entre Georges V et Saint-Placide qui est la station symétrique de Vavin sur la ligne 4 par rapport à Montparnasse. En fait, quel que soit le temps de correspondance, l'algorithme arrive à Montparnasse via la ligne 13 en provenance de Duroc. L'algorithme visite les voisins de Montparnasse, et retient que celle-ci a été visitée. Quand l'algorithme arrive à Saint-Placide sur la ligne 4 en provenance de Saint-Sulpice, il constate que Montparnasse a déjà été visité et ne peut donc plus accéder à Vavin.

Nous avons essayé de modifier l'algorithme de deux manières pour résoudre ce problème, mais aucune n'a abouti. La première tentative a consisté à ajouter aux arêtes deux membres « visitedLR » et « visitedRL » (R et L pour « *right* » et « *left* ») pour indiquer si l'arête a été visité dans un sens ou dans l'autre. Cela n'a pas abouti car nous n'avons pas réussi à adapter notamment la fonction donnant le noeud suivant à traiter par l'algorithme. Pour la seconde tentative, nous avons essayé de modifier la définition d'une station visitée. Au lieu d'un simple booléen, nous avions un « *vector* » de booléen correspondant au « *vector* » des lignes associées à la station. Une station serait alors visitée lorsqu'elle aurait été visitée (selon l'ancien concept) par l'algorithme pour toutes les lignes. Nous n'avons pas eu le temps de terminer cette solution et de vérifier qu'elle fonctionnait bien.

L'interface graphique

Notre première idée a été d'utiliser le logiciel « Qt », dont un excellent tutoriel était disponible sur le SiteDuZéro. De plus, disponible sous Mac, il offrait un très bon compromis. Cependant, après deux jours entiers passés à tenter d'afficher un « Hello World » en vain, nous avons abandonné l'idée de coder avec « Qt » sur Mac. Bien que l'on ait pensé à installer XCode, nous n'avons pas trouvé les bonnes librairies nécessaires au compilateur « G++ ».

Nous nous sommes donc décidés à utiliser le module « Window Form » de l'application « Visual Studio 2010 ». Après une prise en main assez rapide, la première difficulté est apparue : il ne nous a pas été possible de créer une interface

dans laquelle des sous-fenêtres se succédaient dans la fenêtre, au fur et à mesure des clics de l'utilisateur. Aussi avons-nous opté pour un affichage sous la forme d'onglets (nommés « Accueil, « Calcul d'itinéraire » et « Plan interactif » dans notre programme).

Par ailleurs, l'interface graphique « Window form » nous a posé trois autres problèmes, en plus de comprendre son fonctionnement, comment manier les contrôles, etc.

- la vue *Designer* était inutilisable dès lors que du code extérieur était écrit dans « MainWindow.h ». Nous étions obligés de tout mettre en commentaire pour modifier le design facilement.
- nous avons mis du temps à comprendre que la classe « MainWindow » était une classe « managée » et que l'on ne pouvait pas créer n'importe quel type de membre à l'intérieur de la fonction (i.e. il faut des pointeurs).
- le type de *string* lié à l'interface graphique « System::string » est différent de celui que nous avons utilisé dans l'algorithme, « std::string ». Il nous a été difficile de trouver les fonctions de conversion adéquates et à comprendre leur fonctionnement.

Autres problèmes

Nous n'avons pas trouvé de fonction Split pour les « std::string » (qui prend en argument une *string*, un caractère délimiteur, et qui renvoie un *vector* correspondant à la chaîne découpée. Nous avons donc été obligé de coder notre propre fonction Split, ce qui fut loin d'être trivial : nous avons été confrontés à de nombreuses subtilités qu'il a fallu gérer, comme la présence de deux délimiteurs à la suite.

Conclusion

Malgré ces difficultés et les améliorations qui pourraient encore y être apportées, nous sommes satisfaits de notre outil avec son interface agréable à utiliser et ses résultats tout à fait corrects.