Atelier d'approfondissement en informatique: Graphes et Algorithmes

Quentin Garrido

21 avril 2019

Table des matières

1 Introduction

1.1 Objectifs

L'objectif de ce projet est d'implémenter des algorithmes de recherche de plus courts chemins dans un graphe, et plus particulièrement dans un graphe représentant le réseau de métro de Paris.
Nous implémenterons tout d'abord l'algorithme de Dijkstra, puis le modifierons pour qu'ils deviennent l'algorithme A* et nous optimiserons son temps d'éxécution grâce à des structures de données adaptées.

1.2 Utilisation

Tout le code source est disponible à l'adresse suivante : *https ://github.com/garridoq/metro-shortest-path*.

Tout les éxécutables devraient vous être fournis dans le mail et devraient fonctionner sans devoir les recompiler. Dans le cas contraire voici la démarche à suivre :

Un makefile est fourni pour la compilation, il servira à compiler les bibliothèques et les tests. Une fois le code source obtenu il faudra exécuter la commande suivante pour compiler les bibliothèques :

```
> make
```

Vous pourrez alors compiler tous les fichiers de tests de la manière suivante :

```
> make NOM.exe
```

Où NOM est le nom du fichier de test (pour test_heap.c, il faudra entrer make test_heap.exe).

Voici la liste des fichiers de test et leur utilisation :
- test_heap :./test_heap.exe , ce fichier permet de tester l'implémentation du tas binaire et de ses opérations primaires.
- test_dijkstra :./test_dijkstra.exe GRAPH DEBUT FIN , ce fichier va récupérer le graphe dans le fichier GRAPH, puis calculer le plus court chemin de DEBUT à FIN en utilisant l'agorithme de Dijkstra, l'A* et A* avec une file de priorité.
  Un fichier EPS sera créé pour chaque algorithme.

À titre de référence, voici à quoi ressemble le graphe entier du réseau de métro, que nous utiliserons par la suite : Les stations sont réprésentées par les sommets et nous avons une arc entre deux sommets si ils sont reliés par le métro.

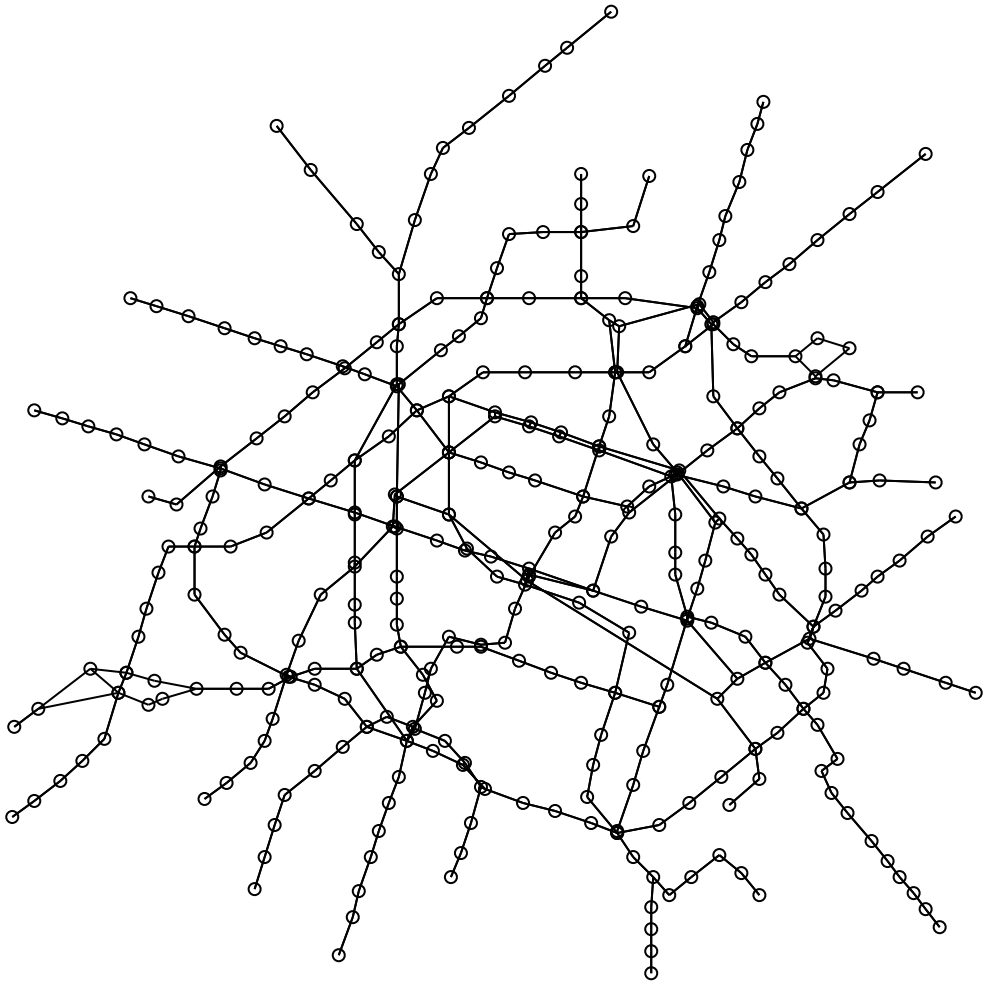

FIGURE 1 – Graphe de référence du métro

2 Algorithme de Dijkstra

2.1 Implémentation

Pour calculer le chemin de plus court d'un point D à A nous allons utiliser la version suivante de l'algorithme de Dijkstra, adaptée depuis le cours de l'unité Graphes et Algorithmes.

Ici nous n'avons pas besoin de calculer les chemins de notre sommet de départ vers tous les autres sommets du graphe et nous nous arrêterons donc dès que nous atteignons notre sommet d'arrivée.

Algorithm 1 Algorithme de Dijkstra

```
1:  procedure DIJKSTRA(E, Γ, l, d ∈ E, a ∈ E)
2:      S = {d}, π(d) = 0, k = 1, x_1 = d
3:      for all x ∈ E\{d} do
4:          π(x) = ∞
5:      end for
6:      while k < n et π(x_k) < ∞ do
7:          for all y ∈ Γ(x_k) tel que y ∉ S do
8:              π(y) = min[π(y), π(x_k) + l(x_k, y)]
9:          end for
10:         Extraire x ∉ S tel que π(x)=min{π(y), y ∉ S}
11:         k = k + 1, x_k = x, S = S ⋃{x_k}
12:         if x_k = a then
13:             break
14:         end if
15:     end while
16:     return π, S
17: end procedure
```

Nous implémentons S avec un tableau de $n = |E|$ éléments, correspondants aux sommets de notre graphe.
Ainsi nous l'initialiserons entièrement à 0 et S = S $\bigcup\{x_k\}$ correspondra à faire S[k] = 1.
Bien que cette implémentation soit plus coûteuse en mémoire qu'une liste chaînée elle permettra d'implémenter l'appartenance à S en temps constant, opération très utilisée aux lignes 7 et 10.
De plus l'ajout d'un élément sera aussi simplifié car nous n'aurons pas à vérifier l'appartenance avant de l'insérer ou non.

2.2 Résultats

Considérons un trajets des stations Alexandre Dumas (1) à Porte Dauphine (256). Nous trouvons alors le chemin le plus court suivant :

Alexandre Dumas − > Philippe-Auguste − > Père Lachaise − > Ménilmontant − > Couronnes − > Belleville − > Colonel Fabien − > Jaurès − > Stalingrad − > La Chapelle − > Barbès Rochechouart − > Anvers − > Pigalle − > Blanche − > Place de Clichy − > Rome − > Villiers − > Monceau − > Courcelles − > Ternes − > Charles de Gaulle, Étoile − > Victor Hugo − > Porte Dauphine

Nous pouvons observer ce chemin avec la figure suivante. Les arcs forment le plus court chemin et les sommets sont uniquement ceux visités. Nous en avons visité 329 sur 376.

Comme nous pouvons le voir nous avons parcouru des sommets qui nous éloignaient grandement du résultat uniquement car le coût pour y aller était plus faible (cf ligne 10 de l'algorithme).

FIGURE 2 – Graphe de référence du métro

Cet effet est exacerbé ici car le trajet que nous avons choisi est particulièrement long.
Ce constat reste le même sur tous les trajets, sauf sur ceux très courts.

Bien que le chemin que nous trouvions ne paraisse pas optimal, nous l'avons vérifié via le service Vianavigo de la RATP, où nous trouvons le même chemin. Cela est du au fait que les deux stations sont sur la même ligne et donc que nous avons aucun changement à faire, d'où la plus courte durée du trajet.

Nous avons vérifié plusieurs autres trajets de la même manière avec le même résultat à chaque fois, ainsi notre implémentation semble être correcte, à condition que l'algorithme utilisé par la RATP pour Vianavigo soit correct, ce que nous pouvons affirmer être vrai.

3 Stratégie A*

Pour utiliser la stratégie A* nous allons appliquer une heuristique lors du choix du prochain sommet à étudier, ce qui correspond à la ligne 10 de l'algorithme de Dijkstra.
En effet nous avons pu voir que choisir toujours le somemt avec le plus court chemin vers lui se révèle sous optimal car nous allons partir dans des directions qui nous éloignent de l'arrivée.
Nous allons donc essayer de choisir une heuristique qui va nous permettre d'explorer moins de sommets.

3.1 Choix de l'heuristique

Dans notre cas, la durée de trajet entre deux stations correspond au poids de l'arc. Cette valeur est calculée pour un métro se déplaçant en moyenne à 10 $m.s^{-1}$ et une unité dans notre graphe correspond à 25.7 m.

L'heuristique choisie repose sur le fait que s'éloigner de l'arrivée est contre productif, et que nous voulons privilégier les stations nous rapprochant de l'arrivée.
Ainsi nous allons utiliser la distance euclidienne entre notre sommet actuel et l'arrivée.
Nous possédons les coordonnées de tous les sommet mais il faut les covnertir en mètres en multipliant par 25.7 puis diviser par 10 pour obtenir une métrique comparable aux valeurs des arcs, et par conséquent des longueurs de chemins.
Cette heuristique devrait s'avérer bonne car très simple à calculer et elle représente le trajet à vol d'oiseau entre nos stations, ce qui est le résultat optimal peu importe le moyen de transport utilisé, aucun trajet ne peut être plus court que cela.

Algorithm 2 Heuristique

```
1: procedure HEURISTIQUE(E, i ∈ E, a ∈ E)        ▷ a est notre point d'arrivée
2:     return distance_euclidienne(i, a) × 25.7/10
3: end procedure
```

Une seule question subsiste, il s'agit de l'optimalité de la solution. En effet si notre heuristique est trop 'forte' par rapport aux valeurs des chemins nous n'explorerons jamais certains sommets qui nous donneraient un plus court chemin.
L'algorithme A* trouvera une solution optimale si l'heuristique est admissible, c'est à dire si l'heuristique ne surestime jamais le coût pour atteindre l'arrivée.
Dans notre cas, la distance euclidienne étant toujours la distance optimale (en supposant que nous sommes dans un plan, en réalité ce n'est pas le cas à cause de la courbure de la Terre, mais l'approximation est acceptable dans notre cas, Paris n'étant pas assez vaste pour que la courbure aie une grande influence) nous pouvons garantir l'optimalité de la solution trouvée.

Nous pouvons bien voir qu'avec cette condition d'optimalité, l'algorithme de Dijkstra trouve une solution optimale car il correspond à avoir une heuristique toujorus égale à 0, or dans un réseau avec des arcs à valeur positive, un chemin est toujours plus long que 0.

3.2 Implémentation de l'algorithme

Algorithm 3 Algorithme A*

```
1: procedure A*(E, Γ, l, i ∈ E)
2:     S = {i}, π(i) = 0, k = 1, x_1 = i
3:     for all x ∈ E\{i} do
4:         π(x) = ∞
5:     end for
6:     while k < n et π(x_k) < ∞ do
7:         for all y ∈ Γ(x_k) tel que y ∉ S do
8:             π(y) = min[π(y), π(x_k) + l(x_k, y)]
9:         end for
10:        Extraire x ∉ S tel que π(x)=min{π(y)+heuristique(y), y ∉ S}
11:        k = k + 1, x_k = x, S = S ⋃{x_k}
12:    end while
13:    return π, S
14: end procedure
```

4 Tas Binaire

4.1 Implémentation classique

4.2 Implémentation pour notre problème

4.3 Autre structures de données possibles

5 Affichage des chemins