Cours de programmation séquentielle

Dijkstra et Floyd - algorithmes de plus cours chemins

1 Préambule

Ce travail pratique est organisé en plusieurs parties traitant la notion de recherche de plus court chemin dans un graphe. Il faudra implémenter les algorithmes de Floyd et de Dijkstra que vous avez vus en cours. Vous appliquerez ces algorithmes sur un graphe représentant un certain nombre de lignes ferroviaires suisses. Finalement, une partie graphique vous sera demandée pour rendre le travail un peu plus ludique.

Dans cette deuxième partie, vous aurez besoin des structures de données que vous avez implémenté ces dernières semaines: la liste et file d'adjacence, ainsi que la file de priorité.

2 L'algorithme de Dijkstra

L'algorithme de Dijkstra permet de déterminer le plus court chemin entre deux points d'un graphe: le sommet de *départ* et le sommet *d'arrivée*.

Dans cette section nous allons décrire l'algorithme en général, puis voir comment l'implémenter (lisez donc toute la section).

2.1 L'algorithme en général

Il s'agit d'un algorithme qui améliore itérativement le calcul de la distance entre les sommets d'un graphe. Nous avons en premier lieu une étape d'initialisation:

1. Nous marquons tous les sommets du graphe comme non visités.
2. Pour chaque sommet, assigner une distance "infinie" à chaque sommet non visité. Le sommet de départ a une distance nulle.

La boucle de mise à jour des distances se passe comme suit:

1. Pour le sommet courant, calculer leurs distances provisoires passant par le sommet courant. Comparer la distance provisoire avec la distance actuellement stockée dans le sommet. Si celle-ci est plus petite remplacer la valeur stockée par celle nouvellement calculée.
2. Lorsque nous avons visité tous les sommets voisins du sommet courant, marquer le sommet comme visité. Un sommet marqué comme visité ne sera jamais plus visité.

3. Si le sommet de destination a été marqué comme visité, ou si les distances avec tous les sommets non visités sont infinies [1] l'algorithme est terminé.
4. Passer au sommet voisin non visité avec la distance la plus faible et recommencer au point 1.

2.2 Implémentation

L'algorithme le plus efficace (et également le plus simple) pour trouver le plus cours chemin entre un sommet de départ et tous les autres sommets d'un graphe utilise une file de priorité min.

Le pseudo-code est donné par

```
void dijkstra(graph g, int src, int num, int dist[num], int prev[num]) {
    dist[src] = 0; // all other distances are INT_MAX
    prev[src] = -1; // previous node is undefined

    queue_init(q, (src, 0));

    while (!queue_is_empty(q)) {
        p = queue_pop(q);
        for v in the neighborhood_of(g, p) {
            d_new = dist[p] + weight(g, p, v);
            if (d_new < dist[v]) {
                dist[v] = d_new;
                prev[v] = p;
                queue_push(q, (v, d_new));
            }
        }
    }
}
```

Il faut noter que la priorité est donnée par la distance totale depuis le sommet de départ. La distance entre `src` et tous les autres sommets est dans le tableau `dist` et le parcours entre `src` et un sommet `v`, est obtenu à l'aide du tableau `prev` en suivant les indices en partant de `prev[v]`.

2.2.1 Remarques

Cet algorithme ne fonctionne que pour les arêtes dont les poids sont positifs. Grâce à cette propriété, il est possible d'ajouter plusieurs fois le même sommet dans la file de priorité (c'est même normal que cela se produise).

3 L'algorithme de Floyd(–Warshall)

L'algorithme de Floyd (ou Floyd–Warshal) est un algorithme qui sert à trouver les plus courts chemins entre toute paire de sommets d'un graphe.

Dans cette section nous allons d'abord décrire l'algorithme puis voir son implémentation.

1. Cela veut dire qu'il n'y a pas de chemin entre le sommet de départ et celui d'arrivée.

3.1 Description de l'algorithme

Considérons un graphe avec N sommets, numérotés de 0 à $N-1$. Soit la fonction `shortest_path(i,j,k)` la fonction retournant le plus court chemin entre le sommet `i` et le sommet `j` en utilisant uniquement les sommets `0, ..., k-1` sommets intermédiaires. En supposant cette fonction connue, nous voulons trouver le plus cours chemin en utilisant les sommet `0` à `N-1`. Pour chaque `shortest_path(i, j, k)` nous avons deux possibilités:

1. le chemin ne passe pas par `k-1` (il ne passe que par les sommets `0` à `k-2`).
2. le chemin passe par `k-1`: d'abord de `i` à `k-1` puis de `k-1` à `j`.

Dans le premier cas, nous savons que le plus court chemin est défini par `shortest_path(i,j,k-1)`. Dans ce second cas, le chemin serait la réunion des chemins de `i` à `k`, puis de `k` à `j`. Ce chemin serait donc la somme des chemins `shortest_path(i, k, k-1)` et `shortest_path(k,j,k-1)`.

De la définition de la fonction `shortest_path(i,j,k)`, on peut déduire que `shortest_path(i,j,0)` est donné par

```
shortest_path(i,j,0)=w(i,j),
```

où `w(i,j)` est le poids de l'arrête entre les sommets `i` et `j`.

Finalement, on peut définir le plus cours chemin récursivement

```
shortest_path(i,j,k) = min(shortest_path(i,j,k-1),
    shortest_path(i,k,k-1) + shortest_path(k,j,k-1)).
```

3.2 Implémentation de l'algorithme de Floyd

L'algorithme de Floyd s'implémente à l'aide de la matrice d'adjacence. Il est décrit en pseudo-code comme

```
void floyd_warshall(int num, int dist[num][num], int next[num][num]) {
    // init dist with INT_MAX
    // init next with -1

    for each edge (i, j) {
        dist[i][j] = w(i, j)  // The weight of the edge (i, j)
        next[i][j] = j
    }
    for each vertex j {
        dist[j][j] = 0
        next[j][j] = j
    }
    for (int k = 0; k < num; ++k) { // standard Floyd-Warshall implementation
        for (int i = 0; i < num; ++i) {
            for (int j = 0; j < num; ++j) {
                if (dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j]
                    next[i][j] = next[i][k]
                }
```

```
            }
        }
    }
}
```

Pour retrouver le chemin il faut simplement suivre les indices dans la matrice `next` de la façon suivante

```
void get_path(int i, int j, int num, int next[num][num], vector *path) {
    if next[i][j] is negative {
        return
    }
    vector_push(path, i);
    while (i != j) {
        i = next[i][j];
        vector_push(path, i);
    }
}
```

4 Tests

Afin de tester le code, vérifier que si vous chercher les temps de parcours entre les villes:

```
Geneve; Sion
Andermatt; Coire
Schaffouse; Lausanne
Lucerne; Delemont
Coire; Geneve
Bale; Bellinzone
Andermatt; Geneve
Lausanne; Delemont
Neuchatel; Bellinzone
Bellinzone; Geneve
Sion; Bale
Lausanne; Geneve
St.-Moritz; Geneve
Sion; Schaffouse
St.-Gall; Lausanne
Neuchatel; Andermatt
Sion; Neuchatel
Bale; Geneve
Geneve; Bale
Berne; Bellinzone
```

on obtient:

```
101 100 188 107 271 205 263 89 257 316
190 34 387 255 212 227 107 157 157 215
```

Puis les chemins entre les villes

```
Geneve; Sion
Andermatt; Coire
Schaffouse; Lausanne
Lucerne; Delemont
Coire; Geneve
Bale; Bellinzone
Andermatt; Geneve
LausanneDelemont
Neuchatel; Bellinzone
Bellinzone; Geneve
Sion; Bale
Lausanne; Geneve
St.-Moritz; Geneve
Sion; Schaffouse
St.-Gall; Lausanne
Neuchatel; Andermatt
Sion; Neuchatel
Bale; Geneve
Geneve; Bale
Berne; Bellinzone
```

sont donnés par

```
[Geneve:Lausanne:Sion]
[Andermatt:Coire]
[Schaffouse:Zurich:Berne:Lausanne]
[Lucerne:Bale:Delemont]
[Coire:Zurich:Berne:Lausanne:Geneve]
[Bale:Lucerne:Bellinzone]
[Andermatt:Sion:Lausanne:Geneve]
[Lausanne:Neuchatel:Delemont]
[Neuchatel:Berne:Lucerne:Bellinzone]
[Bellinzone:Lucerne:Berne:Lausanne:Geneve]
[Sion:Lausanne:Neuchatel:Delemont:Bale]
[Lausanne:Geneve]
[St.-Moritz:Coire:Zurich:Berne:Lausanne:Geneve]
[Sion:Lausanne:Berne:Zurich:Schaffouse]
[St.-Gall:Zurich:Berne:Lausanne]
[Neuchatel:Berne:Lucerne:Andermatt]
[Sion:Lausanne:Neuchatel]
[Bale:Delemont:Neuchatel:Lausanne:Geneve]
[Geneve:Lausanne:Neuchatel:Delemont:Bale]
[Berne:Lucerne:Bellinzone]
```

5 Programme à réaliser

Sur Cyberlearn, vous trouverez un squelette de fichier pour avoir une application complète de recherche de chemin. Il faudra l'adapter à votre code (pas changer le format de sortie par contre) pour pouvoir l'exécuter avec vos différents algorithmes de recherche de plus court chemin et de temps de parcours.