Planification de livraisons

Rapport d'analyse - 15 Mai 2019

Projet tutoré de SD-Graphes

ROSSET Vincent - TANIEL Rémi

Polytech Lille - GIS2A3

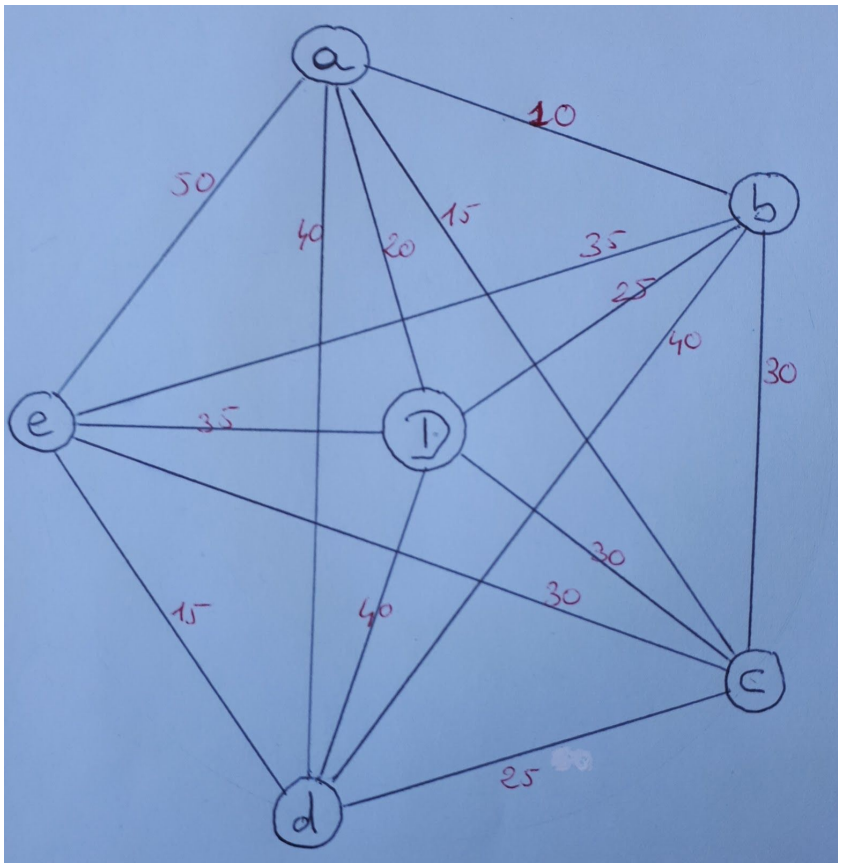
Objectif

Notre objectif est de résoudre un problème de planification de livraison en tenant compte des problématiques de carburant et du nombre de camions qui seront nécessaire pour effectuer la livraison, on recherche donc des coûts minimaux pour les distances en priorité puis le nombre de camions.

Modélisation

Ce problème peut être modélisé par un graphe orienté et plus précisément un réseau avec des coûts positifs, ce graphe possédera N+1 sommets qui représenteront les clients et le point de départ des camions noté D, les arrêtés seront tous les trajets possible entre les différents clients et le dépôt D, les coûts seront les distances entre ces différents points. De plus, on peut remarquer que ce graphe est complet car chaque sommet et relié aux autres sommets par une arête, cela est dû au fait que chaque point est distant d'un autre point.

Exemple avec un graphe avec 6 points (5 clients) :



Nous utiliserons la structure de donnée suivante pour résoudre ce problème :

```
struct donnees_livraison
{
	int n;
	int Q;
	int* q;
	float* distances;
}
```

Cette structure contiendra les informations suivantes :

- n: le nombre de clients à livrer
- Q: la capacité maximale que chaque véhicule pourra transporter
- q: vecteur de taille n contenant les besoins de chaque clients
- distances: la matrice des distances (de taille n+1 x n+1) entre chaque client et le dépôt

Concernant l'implémentation des graphes, nous avons choisi de les représenter avec une représentation tabulaire :

```
struct graphe
{
	int* Head; /* Indice dans le vecteur Succ à partir duquel on peut lire les successeurs */
	char* Succ; /* Liste des successeurs dans l'ordre numérique */
	float* Poids; /* Poids de chaque arête */
}
```

Cette structure contiendra les informations suivantes :

Concernant la modélisation des graphes, nous les représenterons grâce à 3 vecteurs Head, Succ et Poids. Le vecteur Head de taille n+1 représente pour chaque sommet l'indice dans le vecteur Succ à partir duquel on peut lire ses successeurs. Le vecteur Succ représente la liste des successeurs dans l'ordre des sommets, il est de taille n(n-1). Le vecteur poids, de même taille que le vecteur Succ, indique le poids de chaque arête représente par les vecteurs Head et Succ.

En reprenant l'exemple précédent, on obtient le graphe sous cette forme :

Head	0	5	10	15	20	25																							
Succ	a	b	c	d	e	D	b	c	d	e	D	a	c	d	e	D	a	b	d	e	D	a	b	c	e	D	a	b	c	d

Poids	20	25	30	40	35	20	10	15	40	50	25	10	30	40	35	30	15	30	25	30	40	40	40	25	15	35	50	35	30	15

Résolution

Ce problème sera résolu en utilisant l'heuristique route-first/cluster-second qui peut se décomposer en 3 grandes étapes :

- La création d'un *Tour géant* représentant une tournée qui est effectué par un seul véhicule.
- La construction d'un graphe auxiliaire à partir de la procédure SPLIT.
- L'application d'un algorithme de Plus Court Chemin (PCC) sur l'algorithme précédemment construit.

I. *Tour géant*

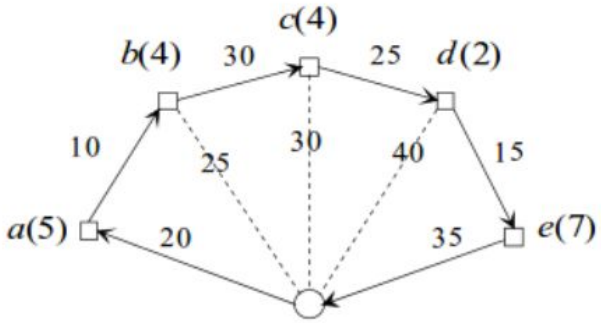
Comme expliqué précédemment, le Tour géant représente une tournée effectué par un seul véhicule sans limitation de chargement, le but de cette étape est de définir un ordre initial de livraison des clients, seulement la tournée ne commence pas au dépôt mais à un client précis.

Puis en partant de ce client initial, nous explorerons les sommets suivant la méthode du plus proche voisin, c'est à dire que d'un sommet x, on regarde tout ses successeurs non marqués et on explore le plus proche, si tous ses successeurs sont marqués, on revient en arrière, voici le pseudo-code associé à cet algorithme :

```
Données:
    dist: matrice des distances entre x, y
    G: graphe de base
    d: numéro du client initial
Résultat:
    T: graphe représentant le Tour géant
Locales:
    Mark: tableau des sommets marqués
    Z: pile de sommet
Code:
    Ajouter les sommets de G à T
    Initialiser Mark à "Faux"
    Mark[d] := "Vrai"
    Empiler d dans Z
    Répéter
        Dépiler x dans Z
```

```
        Chercher le successeur y non marqué de x le plus proche grâce à
dist
        Si y existe Alors
            Créer un arc entre x et y dans T de pois dist(x, y)
            Mark[y] := "Vrai"
            Empiler y dans Z
        FinSi
    Jusqu'à pile Z vide
    Créer un arc entre le dernier sommet marqué et le dépôt dans T
```

Voici le *Tour géant* associé au graphe d'exemple commençant par le client a:



Bien évidemment, l'ordre obtenu dépend du premier client livré.

II. SPLIT

Une fois le Tour géant obtenu, nous appliquons la procédure SPLIT afin d'obtenir le graphe auxiliaire H.

La procédure SPLIT est réalisée de la même manière que l'indique le sujet, voici le pseudo-code associé à cet algorithme :

Procédure SPLIT

Données :
T tour géant
Q capacité des véhicules
n nombre de clients
$dist$ distances entre $x, y \in \{D, C_1, C_2, \cdots, C_n\}$
q vecteur du nombre d'unités à livrer à chaque client
Résultat :
H sous-graphe auxiliaire de sommets h_i $(i \in \{0, \cdots, n\})$ associé à T
Locales :
$cost$ coût courant /* distance parcourue par le véhicule courant */
$load$ chargement courant /* chargement du véhicule courant */
i, j indices utilisés pour parcourir les clients de T

Pour i de 1 à n
$j \leftarrow i$
$load \leftarrow 0$ /* un nouveau véhicule est affrété */
Répéter
$load \leftarrow load + q(T_j)$
Si $(i = j)$ **alors**
/* T_j est le premier client qui sera livré par le véhicule courant */
$cost \leftarrow dist(D, T_i) + dist(T_i, D)$
Sinon
/* T_j est inséré à la fin de la livraison effectuée par le véhicule courant */
$cost \leftarrow cost - dist(T_{j-1}, D) + dist(T_{j-1}, T_j) + dist(T_j, D)$
FinSi
Si $(load \leq Q)$ **alors**
/* la livraison de T_j peut être assurée par le véhicule courant */
Créer un arc (h_{i-1}, h_j) de coût $cost$ dans H
FinSi
$j \leftarrow j + 1$
jusque $(j > n)$ or $(load \geq Q)$ /* La livraison de tous les clients a été planifiée -ou- la livraison du client T_j entraîne une violation de la contrainte de capacité */
FinPour

Ce graphe est retourné selon le "struct graphe" présenté en partie modélisation.

III. PCC

L'algorithme SPLIT nous a donné comme résultats un graphe orienté avec des poids. Nous cherchons l'arborescence des plus courts chemins de ce graphe enracinée en h_0. Notre graphe est sans circuit et avec des poids positifs, nous avons donc le choix d'utiliser l'algorithme de Dijkstra ou de Bellman car notre graphe respecte les condition d'application des deux algorithmes de recherche du plus court chemin. Nous avons choisi de faire

l'algorithme de Dijkstra pour ne pas avoir besoin de réaliser préalablement un tri topologique des sommets. Voici le pseudo-code de cet algorithme :

```
Données:
    H: graphe résultat de l'algorithme SPLIT
    r: indice de racine du graphe H dans le vecteur Head
Résultats:
    cout_min: pot[h_max] coût minimal qui équivaut au potentiel du sommet h_max
    G_pcc: Graphe du plus court chemin reliant h_0 à h_max
Locales:
    Mark: tableau des sommets marqués
    x, y: sommets visités / sommets successeurs du sommet visité
    fini: booléen de fin du programme
Code:
Initialiser Mark à "Faux", pot à "+∞", père à "0" et fini à "Faux"
pot[r] := 0
pere[r] := r
Répéter
      Fini := True
      Choisir x tel que Mark[x]="Faux" et pot[x] minimum
      Si x existe Alors
            Fini := Faux
            Mark[x] := "Vrai"
            Pour k := Head[x] à Head[x+1]-1
                  y := Succ[k]
                  Si pot[y] > pot[x] + Poids(xy) Alors
                        pot[y] := pot[x] + Poids(xy)
                        pere[y] := x
                  FinSi
            FinPour
      FinSi
TantQue fini ≠ "Vrai"
```

Pseudo-code associé au programme principal

Le problème de planification sera décrit avec le fichier suivant :

```
5
10
5 4 4 2 7
0 20 25 30 40 35
```

```
20 0 10 15 40 50
25 10 0 30 40 35
30 15 30 0 25 30
40 40 40 25 0 15
35 50 35 30 15 0
```

On y retrouve les informations suivantes :

- 1 ligne avec le nombre de clients à livre
- 1 ligne avec la capacité maximale en unité des véhicules
- 1 ligne avec les besoins de chaque client
- n+1 lignes avec les n+1 distances entre les différents clients et le dépôt

Notre programme principal utilisera le pseudo-code suivant pour résoudre le problème qui nous a été donné :

```
Données:
    n: Nombre de clients à livrer
    Q: Capacité maximale en unité des véhicules
    q: Demandes de livraison de chaque client
    distances: matrice (n+1)*(n+1) des distances entre tous les sommets
Résultats:
    G_res : Graphe orienté optimal de la livraison
    cout_total : coût total de la livraison du graphe G
Locales:
    G: Graphe initial complet reliant chaque clients entre eux et avec le
dépôt avec les distances en poids (cf exemple en partie modélisation)
    d: numéro client initial
    T: Graphe résultat de la procédure "Tour Géant"
    H: Graphe résultat de la procédure SPLIT
    r: indice de h0 dans H
Code:
Lire n, Q, q et distances dans notre fichier de données
Initialiser G avec les données selon la structure "struct graphe"
Initialiser d au numéro du client initial
T :=  Tour_Geant ( distances, G, d )
H := SPLIT ( T, Q, n, distances, q )
r := indice de h0 dans H
G_pcc, cout_total := Dijkstra ( H, r )
```