



# Planification de livraisons

Rapport d'analyse - 7 Juin 2019

Projet tutoré de SD-Graphes

---

ROSSET Vincent - TANIEL Rémi

Polytech Lille - GIS2A3



## Table des matières

<b>Table des matières</b>	<b>1</b>
<b>Objectif</b>	<b>3</b>
<b>Principe de résolution</b>	<b>3</b>
<b>Modélisation</b>	<b>3</b>
<b>Pseudo langage des procédures</b>	<b>6</b>
Tour géant	6
SPLIT	8
PCC	9
<b>Pseudo-code associé au programme principal</b>	<b>10</b>
<b>Mode d'emploi</b>	<b>12</b>
<b>Description des exemples traités et résultats obtenus</b>	<b>12</b>
exemple.dat	13
cvrp_100_1_det.dat	13
cvrp_100_1_r.dat	13
cvrp_100_1_c.dat	14
<b>Bilan personnel</b>	<b>14</b>
Objectifs	14
Bilan technique	15
<b>Conclusion</b>	<b>15</b>

## 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, on souhaite aussi savoir le nombre de camions nécessaires à cela et leur parcours.

## Principe de résolution

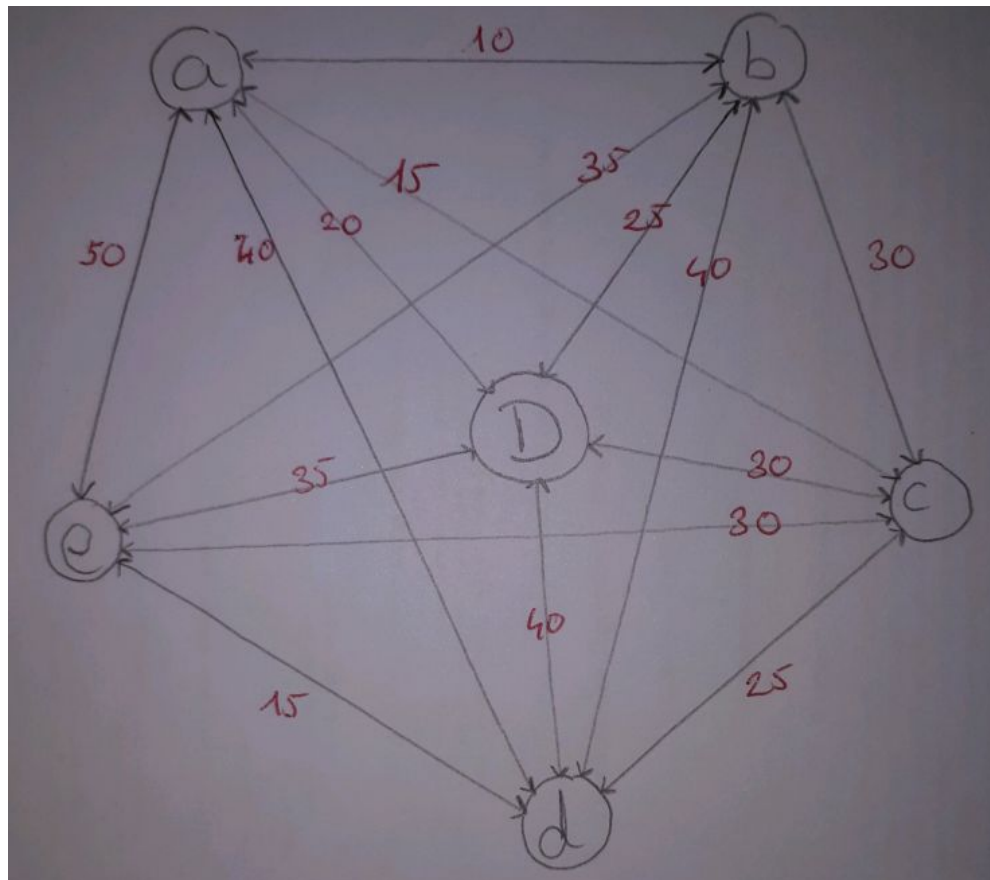
Ce problème sera résolu en utilisant l'heuristique route-first/cluster-second qui peut se décomposer en 3 grandes étapes :

- La création d'un *Tour géant* représentant une tournée qui est effectuée par un seul véhicule. Cette procédure retourne la suite des sommets selon l'ordre gloutonnien du tour.
- La construction d'un graphe auxiliaire à partir de la procédure SPLIT. Cette procédure nous permet d'obtenir un graphe orienté, sans cycles et où les sommets s'enchaînent selon le tri topologique et où il y a un unique sommet par couche du tri topologique.
- L'application d'un algorithme de Plus Court Chemin (PCC) sur l'algorithme précédemment construit. Cette procédure nous permet d'avoir le coût total minimal ainsi que l'ordre des livraisons par véhicules.

## 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 arêtes seront tous les trajets possibles 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 est relié aux autres sommets par une arête, cela est dû au fait que chaque point est distant d'un autre point, les distances aller retour étant les mêmes, nous avons confondu les arêtes.

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



Nous stockerons au départ les données dans la structure suivante pour résoudre ce problème :

```
struct donnees
{
    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  $(n+1)*(n+1)$ ) entre chaque client et le dépôt

Concernant l'implémentation des graphes, nous avons choisi de les représenter selon des structures de données différentes pour chacune des procédures. En effet, la structure de nos graphes évoluera en fonction de ses caractéristiques.

Etant donné que le graphe de départ est complet et que les distances entre 2 clients ne sont pas forcément symétriques, le représenter avec une matrice est la meilleure solution que ce soit en terme de stockage ou de recherche / accès aux données.

La procédure grand tour renvoie une liste d'entier correspondant à l'ordre gloutonnien des livraisons à partir d'un client initial. Nous aurons donc après cette procédure la structure des données ainsi que la liste des entiers correspondant aux sommets.

La procédure SPLIT nous renvoie un graphe auxiliaire orienté sans circuit. L'ajout des arêtes se fait dans l'ordre croissant des sommets. De plus, il sera ensuite parcouru dans ce même ordre car il aura été construit dans l'ordre topologique. Pour ces raisons, nous avons décidé de le représenter sous forme de liste chaînée des vecteurs Head et Succ de taille  $n+1$ , l'ajout en queue et le parcours de la liste étant peu coûteux en complexité et taille mémoire. Pour le stockage des Head et Succ, nous pouvons nous passer de stocker les numéros des sommets car ils correspondent à l'indice du parcours de la boucle. Nous avons tout de même besoin de stocker le sommet de départ de la liste car les indices de départ dans le graphe auxiliaire de SPLIT et du Tour géant sont différents.

Voici les structures maillon et liste utilisées :

```
struct maillon {
    int value, depart;
    float cost;
    struct maillon *next;
};

#define NIL (struct maillon *)0

struct liste {
    struct maillon *tete;
    int nb_elem;
};
```

Nous avons choisi cette représentation plutôt qu'une autre pour des raisons de performances avant tout, en effet ajouter un arc entre 2 points correspond simplement à ajouter un maillon.

Le désavantage de cette modélisation est pour rechercher une arête, ajouter une arête autre part qu'en queue ou tête. Mais nous n'avons pas besoin de ces fonctionnalités, c'est pourquoi cette structure de données nous a semblé idéale.

Enfin, nous avons décidé de passer par une structure "result" pour dans un premier temps stocker les résultats des procédures que nous voulons conserver pour le résultat final et ensuite pour afficher plus facilement l'ensemble des résultats répondant à notre problématique. Nous stockons dans cette structure un nombre réel correspondant au coût total et le maillon tête du plus court chemin du graphe auxiliaire de la procédure SPLIT. Voici donc la structure utilisée en C :

```
struct result {
    float cost;
    struct maillon_point *tete;
};
```

## Pseudo langage des procédures

### I. *Tour géant*

Comme expliqué précédemment, le Tour géant représente une tournée effectuée 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 tous ses successeurs non marqués et on explore le plus proche. Comme le graphe est complet, il n'est pas nécessaire de reculer lors de l'exploration :

#### **Données:**

D: données représentées sont la forme de struct donnees  
d: numéro du client initial

#### **Résultat:**

T: vecteur d'entiers d'ordre de passage des clients

#### **Locales:**

Mark: tableau des sommets marqués  
current : sommet courant  
index : sommet non marqué avec la plus petite distance au sommet

courant

dist : distance entre 2 sommets

i, j : entiers pour réaliser des boucles

**Code:**

Initialiser Mark à "Faux"

$T[0] = d$

$i = 0$

**Tant que**  $i < D.n$  **Faire**

    current =  $T[i]$

    Mark[current] = Vrai

    index = NUL

    dist =  $+\infty$

**Pour** j allant de 1 à n+1 **Faire**

**Si** current  $\neq i$  **et** mark[i] = faux **et** D.distances[current][j]  $\neq$

-1

**et** D.distance[current][j] < dist) **Alors**

            index = j

            dist = D.distances[current][i]

**Fin Si**

**Fin Pour**

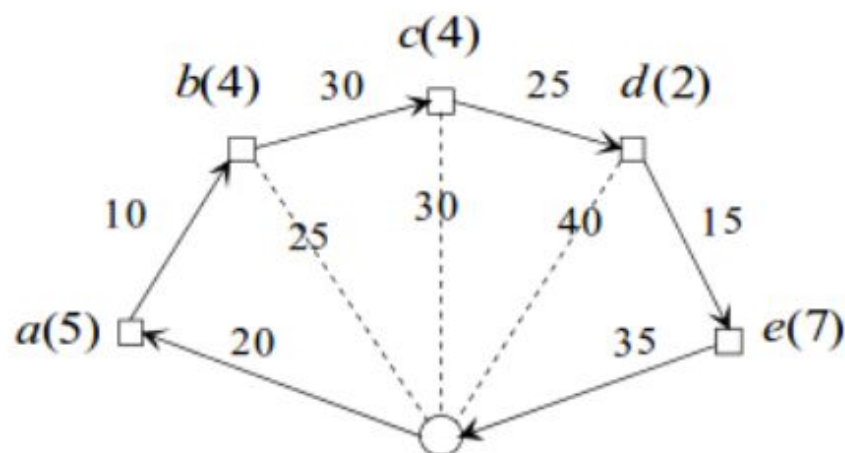
$i = i + 1$

$T[i] = \text{index}$

**Fin Tant Que**

**Retourner** T

Voici le *Tour géant* associé au graphe d'exemple en 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.

Le principe de la procédure SPLIT est la même que sur le sujet, nous l'avons cependant adaptée à notre structure de données. Voici le pseudo-code associé à cet algorithme :

### Données:

D : données initiales (struct donnees)

T : vecteur d'entiers d'ordre de passage des clients du Tour Géant

### Résultat:

Succ : Liste chaînée des successeurs du graphe auxiliaire

### Locales:

cost : coût courant /\* distance parcourue par le véhicule courant \*/

load : chargement courant /\* chargement du véhicule courant \*/

dep : indice du dépôt

i, j : indices utilisés pour parcourir les clients de T

### Code:

Initialiser dep à 0

Initialiser Succ à une liste chaînée vide

**Pour** i allant de 1 à D.n inclu **Faire\***

    j = i

    load = 0

**Tant que** j ≤ D.n **et** load < D.q **Faire**

        load = load + D.q[T[j]-1]

**Si** i = j **Alors**

            /\* Tj est le premier client livré par le véhicule courant

\*/

        cost = D.distances[dep][T[i-1]] + D.distances[T[i-1]][dep]

**Sinon**

            /\*T[j] est ajouté en fin de livraison du véhicule courant

\*/

        cost = cost - D.distances[T[j-2]][dep] +

            D.distances[T[j-2]][T[j-1]] +

D.distances[T[j-1]][dep]

**Fin Si**

**Si** load ≤ D.Q **Alors**

            ajout\_en\_queue\_liste\_succ(succ[i-1], i, j, cost)

**Fin Si**



```

        j = j + 1
    Fin Tant Que
Fin Pour
Retourner T

```

### III. PCC

L'algorithme SPLIT nous a donné comme résultats un graphe orienté sans cycle avec des poids et où les sommets sont triés dans l'ordre topologique. 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 conditions d'application des deux algorithmes de recherche du plus court chemin.

Nous avons choisi d'utiliser l'algorithme de Bellman car le graphe est déjà trié topologiquement et la complexité de l'algorithme de Bellman est en  $O(m)$ , soit une complexité plus faible que l'algorithme de Dijkstra qui a une complexité en  $O(n^2)$ . En effet, on a bien  $m < n^2$  ici.

#### Données:

H: liste des successeurs, résultat de l'algorithme SPLIT  
n: nombre de clients

#### Résultats:

cost:  $\text{pot}[h_{\max}]$  coût minimal qui équivaut au potentiel du sommet  $h_{\max}$   
trajet: Trajet effectué par les véhicules

#### Locales:

pot : vecteur des potentiels pour chaque sommet (taille  $n+1$  avec dépôt)  
pere : vecteur des pères (prédécesseur de l'arborescence des plus courts chemins) pour chaque sommet (taille  $n+1$  avec dépôt)  
start[n] : liste des sommets de départ du véhicule  
finish[n] : liste des sommets de départ des véhicules  
i, j : indices de compteur  
M : maillon de H selon une structure maillon

#### Code:

```

Initialiser pot à "+∞", pere à -1
pot[0] := 0
pere[0] := r
//Propagation des successeurs à la racine  $h_0$ 
M = H[0].tete
Tant Que M non nul Faire
    i = valeur (indice du succ) du maillon M

```

```

    pot[i] = cout (distance avec le succ) de M
    pere[i] = 0
    finish[i] = i
    start[i] = start du maillon M
    M = maillon suivant
Fin Tant Que
Pour i allant de 1 à n inclu Faire
    M = H[i].tete
    Tant Que M est non nul Faire
        j = valeur (indice du succ) du maillon M
        Si pot[j] > pot[i] + cout de M Alors
            pot[j] = cout (distance avec le succ) de M
            pere[j] = i
            finish[j] = j
            start[j] = start du maillon M
        Fin Si
        M = maillon suivant
    Fin Tant Que
Fin Pour
cost = pot[n]
//Sauvegarde du trajet
Tant Que pere[i] ≠ -1 Faire
    ajout_resulat_tab(cost, start[i], finish[i])
    i = pere[i]
Fin Tant Que

```

## Pseudo-code associé au programme principal

Les fichiers d'entrées doivent être formatés comme ci-dessous :

```

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 livrer
- 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:**

D : structure données

d : numéro du client initial

**Locales:**

T: Vecteur d'entier résultat de la procédure "Tour Géant"

H: Liste des Succ résultat de la procédure SPLIT

res: Structure résultat (Coût et maillon tête de la liste des sommets du plus court chemin)

**Code:**

Initialiser D avec les données en entrée

Demander à l'utilisateur le client initial et initialiser d

T := Tour\_Geant ( D, d )

H := SPLIT ( T, D )

res := Dijkstra ( H, D.n )

Afficher(res)

Libérer les espaces alloués aux différentes structure de données

## Mode d'emploi

Un makefile a été créé afin de faciliter la compilation de notre programme, pour le compiler il suffit de donc de lancer la commande suivante dans votre terminal :

```
$ make
```

Puis de le lancer avec :

```
$ ./programme
```

Lors de l'exécution du programme, il vous sera demandé le fichier .dat à analyser ainsi que le numéro du client de départ (> 1)

Voici un exemple d'exécution du programme :

```
Fichier avec lequel lancer le programme:
cvrp 100 1 c.dat

Client de départ:
1

Trajet: [
Camion 1 : (1 2 4 3 5 7 8 9 6 11 10 75)
Camion 2 : (91 89 88 85 84 83 82 86 87 90)
Camion 3 : (63 65 67 66 69 62 74 72 61 64 68)
Camion 4 : (40 41 42 44 45 46 48 50 51 52 49 47 43)
Camion 5 : (20 21 22 23 26 28 27 25 24 29 30)
Camion 6 : (34 36 39 38 37 35 31 32 33)
Camion 7 : (18 17 13 15 16 14 12 19)
Camion 8 : (99 96 95 94 93 92 97 100 98)
Camion 9 : (81 78 76 71 70 73 77 79 80)
Camion 10 : (55 57 59 54 53 56 58 60)
]
Cout total: 882.290039
```

En sortie du programme, nous obtenons la tournée pour chaque camion avec les différents clients qu'ils vont livrer ainsi que le coût total.

## Description des exemples traités et résultats obtenus

Nous avons 4 fichiers d'exemples à notre disposition afin de tester notre programme, voici les résultats que nous avons obtenu pour chaque fichier d'exemple :

### exemple.dat

Ce fichier reprend l'exemple que nous avons dans le sujet du projet, nous obtenons les mêmes données que celles dans le sujet, c'est à dire :

```
Trajet: [  
Camion 1 : (1 2)  
Camion 2 : (3)  
Camion 3 : (4 5)  
]  
Cout total: 205.000000
```

Nous avons donc besoin de 3 camions pour effectuer toutes les livraisons avec un coût total de 205.

### cvrp\_100\_1\_det.dat

Ce fichier simule une liste de 100 clients ainsi que des camions ayant une capacité de 200 unités, seulement les distances entre les différents clients sont très grandes :

```
Trajet: [  
Camion 1 : (1 93 89 49 23 52)  
Camion 2 : (2 18 9 7 29 88)  
Camion 3 : (63 42 16 53 98)  
Camion 4 : (71 72 13 95 26 46 55 90 5)  
Camion 5 : (69 12 51 94 32 6)  
Camion 6 : (14 33 27 74 75 77 35 73 100)  
Camion 7 : (97 65 85 25 47 81)  
Camion 8 : (21 79 43 44 62 19 22 37)  
Camion 9 : (61 48 57 30 68 54 10)  
Camion 10 : (4 17 84 28 67)  
Camion 11 : (83 58 66 80 86 38)  
Camion 12 : (64 20 96 39 78)  
Camion 13 : (34 11 24 8 91 56 3)  
Camion 14 : (76 50 60 99 92)  
Camion 15 : (15 82 70 87 45 41 36)  
Camion 16 : (40 59 31)  
]  
Cout total: 14289.270508
```

Bien évidemment, le nombre de camions nécessaires est beaucoup plus important que précédemment, tout comme le coût total.

### cvrp\_100\_1\_r.dat

Comme le fichier précédent, c'est toujours une base de 100 clients et une capacité par camion de 200 unités, seulement les distances entre les clients sont beaucoup plus faibles:

```

Trajet: [
Camion 1 : (1 69 27 28 53 58 40 21)
Camion 2 : (73 72 74 22 75 56 39 23 67 25 55 54)
Camion 3 : (80 68 77 3 79 33 81 9 51 20 30 70)
Camion 4 : (31 88 7 82 48 47 36 49 19 11)
Camion 5 : (62 10 90 32 63 64 46 8 45 17 84 5 60 83 18 52)
Camion 6 : (89 6 94 95 97 92 59 99 96)
Camion 7 : (93 85 91 100 37 98 61 16 44 14 42 87)
Camion 8 : (2 57 15 43 38 86 13)
Camion 9 : (26 12 76 50 78 34 35 71 66 65 29 24 4 41)
]
Cout total: 1014.550049

```

Comme attendu, le nombre de camions est plus faible, tout comme le coût total.

### cvrp\_100\_1\_c.dat

Dernier fichier pour tester nos données, il présente des caractéristiques similaires au précédent, mais les besoins des clients sont cette fois-ci uniformes et sont quantifiés par des multiples de 10 :

```

Trajet: [
Camion 1 : (1 2 4 3 5 7 8 9 6 11 10 75)
Camion 2 : (91 89 88 85 84 83 82 86 87 90)
Camion 3 : (63 65 67 66 69 62 74 72 61 64 68)
Camion 4 : (40 41 42 44 45 46 48 50 51 52 49 47 43)
Camion 5 : (20 21 22 23 26 28 27 25 24 29 30)
Camion 6 : (34 36 39 38 37 35 31 32 33)
Camion 7 : (18 17 13 15 16 14 12 19)
Camion 8 : (99 96 95 94 93 92 97 100 98)
Camion 9 : (81 78 76 71 70 73 77 79 80)
Camion 10 : (55 57 59 54 53 56 58 60)
]
Cout total: 882.290039

```

Étonnamment le nombre de camion est plus important que pour la précédente base de données, malgré le fait que les besoins des clients soient uniformes. Le coût total est quant à lui plus faible.

## Bilan personnel

Le projet a été réalisé par Rémi TANIEL et Vincent ROSSET sous la supervision de Clarisse DHAENENS. Nous avons eu 12 heures en salle informatique découpée sur 4 séances ainsi qu'une séance préliminaire de 1h30 en salle de cours.

### Objectifs

L'objectif du projet était de résoudre un problème de planification de livraison en tenant compte des problématiques de carburant. Nos camions étant tous les mêmes, nous avons donc cherché une distance minimale parcourue par l'ensemble des camions et le parcours de ces camions.

Le cahier des charges nous détaillait l'heuristique méthode route-first/cluster-second et nous indiquais un pseudo-langage utilisable pour cet algorithme. Nous devions donc faire un choix justifié de structure de données, adapter le pseudo-langage initial et réaliser le code en C.

Nous pensons avoir correctement répondu au cahier des charges, les documents/codes/explications demandés ont été réalisés selon le cahier des charges.

## Bilan technique

Nous avons longuement hésité quant aux structures de données que nous devions utiliser. Notre premier rapport d'analyse faisait référence à des structures de données que nous n'avons finalement pas utilisées. Nous avons également changé le choix de l'algorithme du plus court chemin après avoir remarqué qu'il était plus pertinent d'utiliser l'algorithme de Bellman.

Les principaux changements techniques ont été :

- Simplifier le résultat du Tour Géant en un simple vecteur d'entier
- Modéliser la matrice auxiliaire de SPLIT en liste chaînée des successeurs
- Privilégier l'algorithme de Bellman plutôt que Dijkstra
- Ajouter une structure de résultat pour être affichée plus simplement


La correction du 1er rapport d'analyse nous a beaucoup aidé dans la réalisation de ces modifications.

## Conclusion

Nous avons eu le temps de réaliser les principales fonctions qui nous avaient été demandées, à savoir :

- Le trajet de chaque camion
- Le coût total associé

Il manque peut-être le coût associé à chaque camion qui pourrait être utile afin de déterminer quel camion parcourt le plus de distance.



En ce qui concerne les points d'améliorations, notre programme ne cherche pas le client initial qui permettrait d'avoir un coût total minimal, il se contente d'appliquer les algorithmes avec un client de départ donné.