PROJET DE
GRAPHES ET COMBINATOIRE

Enseignante: Marie-Eléonore Marmion

DEZE Florian
JONGMANEE Tony
GIS3

2015-2016

Sommaire

Introduction

Dans le cadre de notre formation en Génie Informatique et Statistique, en 3ème année, à Polytech Lille, nous avons suivi un cours de **Graphes et Combinatoire** et **Structure de données**, suivis d'un projet dont l'objectif étant d'appliquer les notions vues lors de ces cours.

Les transporteurs essaient de planifier leurs livraisons pour les rendre optimales en réduisant les coûts de transport. Ces transporteurs ont une contrainte majeure dont ils doivent tenir compte: la capacité de chargement de leurs véhicules. Cela entraîne l'utilisation de plusieurs véhicules pour livrer dans les délais leurs clients.

Le but de ce projet est de trouver une planification optimale qui permettra de réduire les coûts de déplacement de livraisons par un ensemble de véhicules.

Pour cela nous analyserons ce problème dans une première partie comme un problème de graphes. Nous détaillerons la modélisation proposée, avec un exemple, puis nous expliquerons quelles types de structures de données seront utilisées pour problème. Nous terminerons par présenter les algorithmes de résolution de ce problème.

1. Analyse du problème

Le but du problème est de livrer des clients à partir d'un dépôt D. Nous pouvons livrer plusieurs clients avant de repasser au dépôt, donc aller vers n'importe quel client à partir d'un autre sans intermédiaires (autres clients ou dépôt). Nous supposons que le nombre de clients est N et n ce nombre auquel on ajoute le dépôt pour illustrer notre modélisation. Un client est livré en une seule fois. Ainsi, la demande d'un client est toujours inférieurs à la capacité de transport d'éléments à livrer d'un véhicule.

On considère les C1,C2,...,Cn les n clients à livrer de qi unités. Tous les véhicules doivent partir d'un dépôt D et ne peuvent transporter qu'une quantité maximale Q. Nous souhaitons minimiser la distance de transport des marchandises en déterminant l'itinéraire le plus court possible.

1.1. Modélisation

Au vue du nombre important de données que nous possédons, nous allons réaliser une modélisation répondant au problème posé sans prendre en compte la quantité à livrer qi, ni la quantité Q ou le nombre de véhicules. Ces données seront prises en compte dans nos algorithmes de résolutions. Voici une proposition de modélisation :

Soit un graphe orienté G=(X,A,c)
X : Sommets : Le dépôt D
Les clients Ci avec $1<i<N$
A : Arcs : Les routes entre les clients Ci et Cj avec $i \neq j$
Les routes entre le dépôt et les clients
c : Coût : La distance entre tous les clients et entre le dépôt et les clients

1.2. Analyse et simplification du graphe

Pour tout $i,j \in [1..n]$ tel que $i \neq j$, nous avons un arc de Xi vers Xj et un arc de Xj vers Xi de même coût c car les distances sont symétriques. On choisit de simplifier ce graphe en le transformant en un graphe non orienté. Cela permettra de stocker moins d'informations pour l'implémentation.

Nous ne prenons pas en compte pour l'analyse et la modélisation du graphe le nombre de véhicule, sa capacité maximale, ni la quantité qu'il transporte après être passé chez un client. Cela sera pris en compte dans des structures de données ultérieurement.
Nous supposons qu'il existe une route reliant chaque client entre eux et chaque client au dépôt. Ainsi le graphe obtenu est complet.

Nous obtenons donc un graphe non orienté, complet, pondéré positif :
G = (X,E,c)

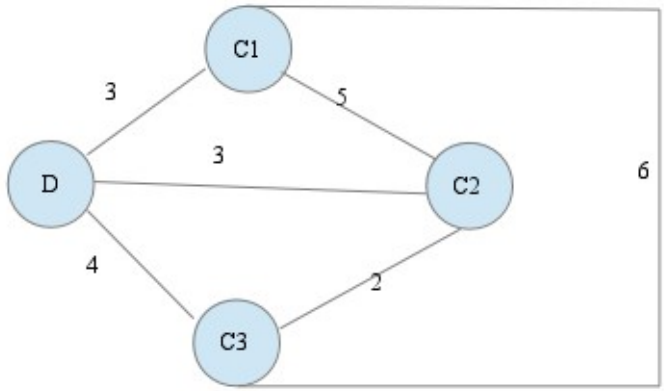
1.3. Exemple de graphe

Nous choisissons de réaliser un exemple avec un nombre réduit de clients afin de visualiser le graphe. Soit le graphe G=(X,E,c) :

X={D,C1,C2,C3}

E={(D,C1);(D,C2);(D,C3);(C1,C2);(C1,C3);(C2,C3)}

c={3,3,4,5,2,6}



Les éléments de c sont dans le même ordre que ceux de E.

1.4. Analyse du sujet sous la forme d'un problème de graphe

Le but est de livrer, en partant d'un dépôt, chaque client, en repassant ou non au dépôt entre un client Ci et Cj. La livraison doit se faire avec un distance minimale de parcours. Cela revient à chercher le plus court chemin entre le dépôt et les clients à livrer. Dans notre modélisation on cherche un chemin de coût minimum entre D et Ci ($i \in [1..N]$) ainsi qu'entre les clients Ci et Cj ($j \neq i$).

2. Structure de données

Nous devons implémenter le graphe avec ces coûts c et la quantité à livrer pour chaque client.

2.1. Sommets et arêtes du graphe

Nous savons que le graphe est complet et non orienté. Cela nous permet de décider plus facilement du type d'implémentation à choisir. En effet il existe quatre méthodes couramment utilisées pour cela : la matrice d'adjacence, la matrice d'incidence, la représentation tabulaire avec la table des arcs et la table des successeur.

Concernant la table des arcs, celle-ci ce compose de deux tableaux SI et ST, correspondant respectivement aux sommets incidents et adjacents. Ils sont de taille m (m=nombre d'arêtes). Le graphe est complet, donc le nombre d'arêtes est égale à n(n-1). Comme le graphe est non orienté, on devrait stocker deux fois la même arête, avec Xi vers Xj et Xj vers Xi (avec X un sommet quelconque). Cependant, il est possible de faire la correspondance entre les sommets de SI et ST, et donc de stocker l'arc une seule fois de Xi vers Xj. En effet, en partant du tableau ST, on peut obtenir l'arc allant des sommets Xj vers Xi. La taille de chacun des tableaux est divisé par deux : n(n-1)/2. De plus, il est nécessaire de stocker le poids des arêtes. On obtient ainsi un coût de stockage de 3n(n-1)/2.

Pour la table des successeurs, elle est composée de deux tableaux Head et Succ, l'un contenant les sommets et l'autre les successeurs. Le premier tableau est de taille n (n=nombre de sommets) et le second de taille m=n(n-1). A partir des sommets du tableau Succ, on ne peut pas retrouver son prédécesseur. Il est ainsi impossible de simplifier son coût comme avec la table des arcs. On stocke en plus les poids des arcs. On crée donc un autre tableau de taille m contenant le poids des arcs. On obtient ainsi un coût de stockage de n+2n(n-1).

La matrice d'adjacence a un coût de stockage de 0(n2), alors que celle d'incidence est de 0(n x m), avec n le nombre de sommet et m le nombre d'arêtes. Or, le graphe est complet, il y a donc plus d'arête que de sommets à partir d'au moins trois sommets. Par conséquent : n2<nxm, ainsi la matrice d'adjacence a un coût moyen moins élevé si le nombre de sommet est supérieur à 2. Cela sera toujours le cas car nous aurons toujours au moins trois sommets : un dépôt et aux moins deux clients. S'il existe seulement un client, une méthode de résolution ne serait pas nécessaire.

Malgré qu'il soit mieux de prendre un tableau dynamique pour le stockage des arêtes, la matrice est mieux afin de rendre plus facile l'implémentation des algorithmes de résolution. Ainsi, la structure de données la plus avantageuse est la matrice d'adjacence avec un coût de stockage n2.

2.2. Distance entre les clients et entre les clients et le dépôt
Quantités de livraisons

Nous savons que le nombre de clients est fixe et qu'il n'y a qu'un seul dépôt. Par conséquent, les nombres de sommets et d'arêtes dans notre graphe sont également fixes, et le coût de stockage des informations dans ce graphe reste inchangé et est égale à n2 (Coût pour aller de Xi à Xj avec i,j∈[1..n]). Nous n'avons pas besoin de ré-allouer de la mémoire pour le stockage de ces informations. On en déduit que l'on peut les stocker dans un tableau à deux dimensions de taille nxn : la matrice sera symétrique et comportera des 0 dans toute sa diagonale.

Le même raisonnement se fait sur le choix de l'implémentation des quantités à livrer à chaque client. Sa taille égal au nombre de clients, ici N. Cette valeur dépend du nombre clients, donc elle est fixe. Les information seront stockées dans un tableau de taille N.

De plus, nous remarquons que la matrice comportant les distances est similaire à la matrice d'adjacence. Celle-ci comporte un 1 lorsque les sommets sont reliés entre eux. Dans le cas avec notre graphe complet, la matrice est remplie de 1 avec sa diagonale composée uniquement de 0. Par conséquent, chaque 1 correspond à un lien entre deux sommets et donc l'existence d'une distance. La matrice d'adjacence est donc inutile avec notre graphe et peut être remplacée par notre matrice de distance. Nous choisissons de supprimer la matrice d'adjacence et d'utiliser la matrice Dist pour représenter le graphe.

2.3. Quantité maximale d'un véhicule et quantité transportée

Notre problème possède quelques contraintes. Un véhicule ne peut transporter qu'une quantité maximale Q d'éléments à livrer. On choisit de définir Q comme une variable de type entier qui sera demandé à l'utilisateur. Puis, nous décidons d'implémenter la quantité qi à livrer à chaque client dans un tableau de taille n. De ce fait, il suffit de planifier quel véhicule peut être envoyé à partir de la quantité restante se trouvant dans le véhicule. Si la quantité restante est inférieur, on supposera qu'il faut un second véhicule. Ce nombre sera dans une variable que nous utiliserons lors de la détermination du plus court chemin.

2.4. Etapes de résolution

Nous avons un graphe complet non orienté G. La résolution du problème se fera en trois étapes :

Etape 1 :

Données : Sous graphe G'(X',E',c) de G
- X' = sommets : clients de G
- E' = arêtes : routes entre chaque client
- c = distances entre chaque client

Sortie : Chaîne Hamiltonienne de coût minimum.
Résolution : Algorithme de Tour Géant

Etape 2 :

Données : Chaîne Hamiltonienne de coût minimum
Sortie : Sous graphe auxiliaire H(X,A,w)
Résolution : Algorithme de SPLIT

H(X,A,w) est un graphe orienté, pondéré et connexe

X = sommets : hi (iϵ[0,n]), correspond au retour au dépôt après la livraison du ième client.

A = arcs : livraisons possibles des clients avec départ et retour au dépôt. On prend en compte la capacité d'un véhicule et de la quantité à livrer à chaque clients. Un client ne peut pas être livré en plus d'une fois.

w = coûts : distance parcouru par le véhicule

Dans l'algorithme de SPLIT, on remarque que les arcs de H sont créés à partir d'un sommet incident d'indice i. Comme on incrémente i et que i≤j, on ne revient jamais sur le sommet incident. Ainsi, par construction le graphe H est sans circuit et possède une source et un puits. Nous avons choisi tout d'abord de représenter le sous graphe auxiliaire H par une matrice contenant les distances comme pour le graphe G. Cependant, nous remarquons que beaucoup de données ne servent pas. La matrice sera remplie de beaucoup de cases vides (valeurs spécifiques indiquant qu'elles ne sont pas des distances). En effet, pour un graphe complet, il est intéressant de prendre une matrice mais pour ce graphe orienté, il sera préférable d'utiliser une table de successeurs avec, en plus, un moyen de stocker les poids.

Etape 3 :

Données : Sous graphe auxiliaire H(X,A,w)
Sortie : Plus court chemin de H allant de H0 au Hn
Résolution : Algorithme de plus court chemin : Djikstra

Le sous graphe H ne comporte pas de circuit et est pondéré positivement. On utilise donc l'algorithme de Bellman ou Djikstra. On choisit celui de Dijkstra car pour utiliser l'algorithme de Bellman, il est nécessaire de définir des niveaux, il faudra effectuer des traitements supplémentaires.

2.5. Choix des structures de données

2.5.1. Graphe de départ G

Comme nous l'avons expliqué précédemment, le graphe G que nous créons est modélisé sous une forme d'une matrice d'adjacence. Lors de la lecture des données, nous connaissons le nombre de clients, donc le nombre de sommets. De plus, nous ne comptons pas modifier le graphe G, seulement exploiter ses informations et créer d'autres graphes. Nous avons besoin de regarder uniquement les successeurs de chaque sommets. On peut donc utiliser soit des listes, soit un tableau à deux dimensions.

La taille et les données de ce graphe ne sont pas modifiées donc on peut utiliser un tableau à deux dimensions. En effet, les listes sont surtout pratiques lorsqu'il y a un changement sur le nombre de données. Cependant nous devons tout de même déclarer les éléments (variables, constante et tableaux) avant la lecture des données. La taille n'est donc pas connu lors de la déclaration du tableau à deux dimensions. Nous décidons de déclarer ces tableaux sous forme dynamique et d'allouer l'espace mémoire nécessaire après la lecture du nombre de clients et la quantité maximale que peut transporter un véhicule. La taille de notre tableau ne sera modifiée qu'une seule fois. L'avantage est que nous utilisons seulement l'espace mémoire nécessaire pour stocker les données.

2.5.2. Tour géant

Le Tour géant renvoie une chaîne Hamiltonienne de coût minimum. Cette chaîne est de taille N. Elle n'est pas modifiée, seulement lue par l'algorithme SPLIT. On en déduit que sa taille est fixe et que l'on peut utiliser un tableau dynamique que l'on dimensionne qu'une fois lors de l'utilisation du tour géant.

La matrice de taille nxn prend en compte le Dépôt. Nous avons choisi d'utiliser cette matrice et non une copie sans dépôt car il nous suffit de changer les indices et nous évite ainsi la recopie de matrice qui est coûteuse. Par conséquent, dans l'implémentation en C, on prendra la matrice Dist de taille nxn.

2.5.3. Graphe H retourner par SPLIT

Nous avons vu que le graphe H est représenté par une table des successeurs avec laquelle on associe les poids.
L'algorithme SPLIT crée les successeurs d'un sommet et Dijkstra regarde les successeurs et non les prédécesseurs. Donc ce graphe est regardé dans un seul sens, de la source vers le puits. Les représentations possibles sont donc les tableaux dynamiques ou les listes. Ce graphe n'est pas modifié lors de sa construction. Si nous choisissons les tableaux dynamiques, il nous faudra trois tableaux : Head, Succ, Poids.

Head : représente les sommets

Succ : Les successeurs de Head

Poids : Les poids associés à chaque arcs (donc à chaque éléments de Succ)
La taille de Head est n (nombre de clients + dépôt)

Le graphe H est connexe (à partir de la source H0 je peux atteindre tous les autres sommets) donc le nombre maximal d'arcs est n(n-1)/2. Cela représenta la taille de Succ et Poids. La taille totale de stockage maximale est $n+2n(n-1)/2 = n^2$.
Deux choix s'offrent à nous, soit des listes chainées pour Head, Succ, et Poids auquels on ajoute un élément un par un, soit trois tableaux dynamiques dont deux potentiellement sur-dimensionnnées que l'on redimensionne à la fin de l'algorithme.

Le nombre d'allocation est beaucoup plus importante pour les listes que pour les tableaux. Pour les tableaux, on a cinq allocations, une seule pour Head, deux pour Succ et Poids (allocation maximale puis redimensionnement). Pour les listes, on aura n allocations pour Head, avec au maximal n^2 allocations au total pour nos tableaux Succ et Poids. En terme de coût, on choisit donc les tableaux. Enfin, les tableaux prennent moins de places en mémoire car les données sont stockées à la suite contrairement aux listes. Ainsi, notre choix final ce porte sur les tableaux dynamiques.

2.5.4. Plus court chemin (PCC)

Pour stocker le plus court chemin, nous avons choisi de le mettre dans un tableau. L'utilisation des listes n'est pas nécessaire car on connaît le nombre maximal que peut prendre ce chemin : chaque voiture peut transporter la marchandise pour un seul client. Ce nombre maximal est donc égal au nombre de sommets du sous graphe auxiliaire. Nous choisissons ainsi de surdimensionner le tableau de d'adapter ensuite celui-ci avec son nombre d'élément comme nous le faisons pour l'algorithme SPLIT.

Ainsi, nous choisissons de créer une structure contenant un tableau dynamique et le nombre d'éléments contenu dans le tableau.

2.5.6. Solution

Pour le stockage de la solution au problème (les différents parcours des voitures), nous choisissons d'utiliser un tableau de liste chaînée. Nous connaissons le nombre de parcours au début de l'algorithme. Le nombre de clients pour un parcours est aussi connu. Nous pouvons utiliser aussi un tableau dynamique à 2 dimensions, cependant il sera plus difficile à l'implémenter. En effet, il est nécessaire de connaître les indices du tableau mais aussi quel entier (client) stocké. Pour cela, nous choisissons de prendre des listes chaînées. Concernant l'implémentation des listes, nous avons besoin des procédures permettant de l'initialisation, l'ajout en queue, la récupération des valeurs d'une liste et la libération de la mémoire d'une liste.

3. Algorithmes de résolution

Dans cette partie, nous allons présenter les différents algorithmes nécessaire à la résolution du problème. Celle-ci est effectuée en quatre étapes :

- la réalisation du tour géant
- la création du sous-graphe auxiliaire à partir du tour géant
- la recherche du plus chemin du sous graphe
- l'étape de décodage et du calcul du coût.

Nous commençons par l'étape du tour géant.

3.1. Algorithme de Tour géant

Dans cet algorithme, on ne prend pas en compte le dépôt et la limitation de chargement du véhicule. Le tour géant permet de connaître un chemin passant par tous les clients une seule fois avec un unique véhicule. On prend le chemin le plus court à partir d'un client choisi préalablement.

Pour cela, on a besoin d'une sous-fonction permettant de déterminer le client le plus proche d'un autre :

Fonction RechercheMinDistMark(N, Dist, i, Mark): entier

Données : N : entier : nombre de clients
Dist : Matrice de taille nxn de réels {Distances entre chaque sommet}
i : entier {Client ayant le plus petite distance avec le dernier client visité}
Mark : tableau de booléen de taille N

Locales : k : entier {indice de parcours}
indi_min : entier {client le plus proche de i}
dist_min : réel {distance entre i et indi_min}

```
{Initialisation}
indi_min=2

{Recherche d'un élément non marqué}
Tant Que (Mark[indi_min-1]=VRAI) Faire
        indi_min=indi_min+1
Fin Tant Que
dist_min=Dist[i+1][indi_min+1]

{Recherche de l'élément de plus petit distance et non marqué}
Pour k de 2 à N+1 Faire
        Si (Dist[i+1][k+1]<>0) et (Mark[k-1]=FAUX) et (dist_min>Dist[i+1][k+1]) Alors
                dist_min=Dist[i+1][k+1]
                indi_min=k
        Fin Si
Fin Pour
```

Retourner (indi_min)
Fin Fonction

Nous déterminons donc le tour géant grâce à l'algorithme RechercheLinDistMark :

Fonction Tour_géant (Ci, Dist, N) : tableau de clients
Données : Ci : entier {Numéro de client initial}
Dist : Matrice de taille nxn de réels {Distances entre chaque sommet}
N : entier : nombre de clients
Locales : k : entier {indice de parcours}
Min : entier {Client ayant le plus petite distance avec le dernier client visité}
Chemin : tableau[N] d'entier {chemin parcouru par le véhicule a partir du sommet Ci est passant par tous les autres sommets}
Mark : tableau de booléen de taille N

{Initialisation}
Chemin[1] = Ci
Min = Ci
Pour k *de* 1 *à* N *Faire*
Mark[k] = FAUX
Fin Pour
Mark[Ci] = VRAI

{Conception}
Pour k *de* 2 à *N*
Min = RechercheMinDistMark(N, Dist, Min, Mark)
Mark[min] = VRAI
Chemin[k] = Min
Fin Pour

Retourner Chemin
Fin Fonction

Nous obtenons ainsi une chaîne hamiltonienne correspondant au Tour Géant. L'ordre des éléments de cet chaîne dépend du premier client fourni au début de l'algorithme Tour_géant.

3.2. Algorithme de SPLIT

Cette fonction permet d'obtenir un sous graphe auxiliaire H nécessaire pour la résolution du problème. Cette algorithme prend en compte la capacité des véhicules et la quantité à livrer à chaque client.

H est représenté par une structure contenant trois tableaux Head de taille n, Succ pour les arcs et Poids pour les distances en les sommets.

Fonction SPLIT(Chemin,Q,N,Dist,Quantite) : table des successeurs

Données : Chemin: tableau de taille n d'entier {Tour géant}
Q: entier {Capacité des véhicules}
N: entier {nombre de clients}
Dist : matrice d'entiers de taille nxn {distances}
Quantite : tableau de taille N d'entier {nombre d'unité à livrer à chaque clients de Chemin}
Locales : H : table des successeurs {sous graphe auxiliaire de sommets Ci associé à Chemin}
cost : réel {coût courant}
load : entier {chargement courant}
i, j : entiers {indices de parcours}
cpt : entier {compteur pour indice de tableaux}

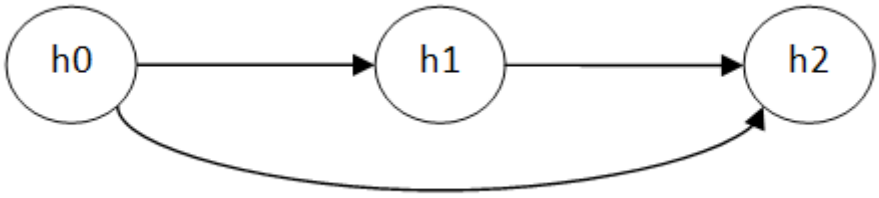
```
{Initialisation}
cpt=1

{Conception}
Pour i de 1 à n Faire
        j=i
        load=0
        Head[i] = cpt
        Faire
                load=load + Quantite[Chemin[j]+1]
                {Calcul du cout}
                Si (j=i) Alors
                        cost= 2*Dist[1][Chemin[i]+1]
                Sinon
                        cost = cost - Dist[1][Chemin[j-1]+1] +
                                Dist[Chemin[j-1]+1][Chemin[j]+1] + Dist[Chemin[j]+1][1]
                Fin Si
                Si (load<=Q) Alors
                        Création d'un nouvel arc}
                        Succ[cpt]=j
                        Poids[cpt] =cost
                        cpt=cpt+1
                Fin Si
                j=j+1
        Tant Que (j<=N) ou (load<Q)
```

Fin Pour
{le dernier sommet est un puit}
Head[N+1] = cpt
Retourner H
Fin Fonction

Nous obtenons donc le sous graphe auxiliaire nécessaire pour résoudre le problème. Chaque sommet du sous graphe représente le dépôt. Les arcs correspondent à un parcours que peut effectuer une voiture avant de retourner au dépôt.

Par exemple, si nous obtenons le sous graphe ci-dessous :



Nous aurons besoin ici une ou deux voitures (deux chemins de h0 à h2). Pour choisir quel chemin il faut prendre, nous utilisons un algorithme de plus court chemin. On obtiendra un tableau contenant des sommets du sous graphe H. Un algorithme de décodage est nécessaire par la suite afin d'obtenir les clients à visiter pour chaque parcours.

3.3. Algorithme de plus court chemin

Tous les coûts sont positifs et le graphe H est orienté et sans circuit. On peut appliquer comme algorithme de plus court chemin un algorithme de Bellmann ou de Dijkstra. Cependant l'algorithme de Bellmann requiert la définition de couches dans le graphe, ce qui entraîne des traitements supplémentaires. Nous choisissons donc d'utiliser l'algorithme de Dijkstra. Celui-ci sera appliqué en respectant l'ordre des tableaux H.Head et H.Succ.

Algorithme de Dijkstra avec nos structures de données :

PCC contiendra le plus court chemin de poids minimum. Il s'agira d'une structure contenant un tableau d'entier dynamique et le nombre d'élément.

Pour appliquer l'algorithme de Dijkstra, nous avons besoin d'une fonction permettant de choisir le sommet de plus peitt potentiel et non marqué.

Fonction Choisir(n,Mark,Potentiel) : entier

Données : n : enter {Nombre de sommets}
Mark : tableau de booléen de taille n
Potentiel : tableau d'entier de taille n
Locales : i :indice de parcours
indice_depart : entier {indice de parcours}
potentiel_courant : entier {poids minimum non marqué dans Mark}
indice_courant : entier {indice dans potentiel du client non marqué ayant le poids minimum}

```
{Initialisation}
indice_depart=1
{Recherche d'un élément non marqué}
Tant Que (indice_depart<=n) et (Mark[indice_depart]) Faire
        indice_depart = indice_depart+1
Fin Tant Que
potentiel_courant = Potentiel[indice_depart]
indice_courant=indice_depart

{Conception}
Pour i de indice_depart à n Faire
        Si (Mark[i]=FAUX) et (potentiel_courant > Potentiel[i]) Alors
                potentiel_courant = Potentiel[i]
                indice_courant = i
        Fin Si
Fin Pour
Si (indice_courant = n) Alors {Tous les éléments sont marqués}
        retourner  (-1)
Sinon
        retourner (indice_courant)
Fin Si
```

Fin Fonction

De plus, nous définissons une constante MAX correspond à l'infini. En C, elle sera égale à 500000.

Fonction Dijkstra (H, n) : Parcours

Données : H : une structure contenant trois tableaux Head, Succ et Poids
n : entier {nombre de sommets}

Locales : Mark :booléen: tableau de taille n
Potentiel :entier: tableau de taille n
Père :entier: tableau de taille n
PCC : tableau d'entier: contenant le plus court chemin
Fini: booléen: indique quand l'algorithme se finie
k : entier : parcours
x, y : entiers

```
{Initialisation}
Parcours[1] = n-1
nbchemin = 1
Pour k de 1 à n Faire
        Mark[k] = FAUX
        Potentiel[k] = MAX
        Père[k] = 0
Fin Pour
Potentiel[1]=0
Père[1]=0
Fini = FAUX
{Conception}
Faire
        Fini=VRAI
         x = Choisir(n, Mark, Potentiel)
        {Successeur trouvé}
        Si (x<>-1) Alors
                Fini=FAUX
                Mark[x]=VRAI
                Si (x<>n-1) Alors
                        Pour k de Head[x] à Head[x+1]-1 Faire
                                y = Succ[k]
                                Si (Potentiel[y]>Potentiel[x]+H.cost[k]) Alors
                                        Potentiel[y]=Potentiel[x]+H.Poids[k]
                                        Père[y]=i
                                Fin Si
                        Fin Pour
                Fin Si
        Fin Si
Tant Que Fini=FAUX
{On récupère le chemin de h0 au dernier}
Parcours[1] = n
{Tant qu'on n'atteint pas h0}
Tant Que (Parcours[nbchemin]≠0) Faire
        Parcours[nbchemin+1] = Père[Parcours[nbchemin]]
```

```
            nbchemin = nbchemin + 1
        Fin Tant Que

        Retourner PCC
Fin Fonction
```

L'algorithme de Dijkstra nous fournit un arbre couvrant de poids minimum, ainsi aucun cycle s'y trouve, il existe donc un unique chemin allant de h0 au dernier.
A la fin de l'algorithme, on obtient un tableau contenant le plus court chemin du sommet 0 à n dans l'ordre décroissant. Celui-ci n'est pas important car nous voulons déterminer combien de voitures et quels parcours sont nécessaires pour livrer tous les clients.

Enfin, il reste à décoder le tableau obtenu par l'algorithme de Dijkstra et calculer le coût total.

3.4. Algorithme de décodage et calcul du coût

L'algorithme de décodage est nécessaire pour déterminer les parcours des voitures. Nous avons besoin du Tour Géant déterminé auparavant afin de connaître l'ordre des clients à visiter. Par exemple, l'algorithme de Dijkstra nous donne le chemin : 4 2 1 0 avec un tour géant 3 2 1 4. Nous avons besoin ici de trois voitures. La première voiture s'occupe du client 3, la deuxième du client 2 et la dernière des clients 3, 4.

L'algorithme détermine le nombre de voitures ainsi que tous les parcours qui doivent être effectués. Il utilise les procédures des listes.

```
Fonction Décodage (PCC, nbliste, chemin) : tableau de liste chaînée d'entier

        Données : PCC : structure parcours
                  chemin : tableau d'entier {Tour géant}

        Données/Résultats : nbliste

        Locales : Solution : tableau de liste chaînée d'entier
                  i, j : indice de parcours

        Pour i de 1 à nbchemin Faire
                {Ordre décroissant}
                Pour j de parcours[nbchemin-i]+1 à parcours[nbchemin-i-1] Faire
                        ajout_element_liste(Solution[i], chemin[j])
                Fin Pour
        Fin Pour
        nbliste = nbchemin-1

        Retourner Solution
Fin Fonction
```

Nous obtenons à la fin de cet algorithme les parcours à effectuer pour chaque voiture. Il ne reste plus qu'à calculer le coût total de ces parcours.

Pour cela, nous utilisons l'algorithme suivant :

```
Fonction calcul_cout(Solution, nbliste, Dist) : réel
    Données : Solution : tableau de liste chaînée d'entier
              nbliste : entier : nombre de liste dans Solution
              Dist : matrice de réel de taille nxn

    Locales : i, j : entiers {indices de parcours}
              cout_total : réel
              val1, val2 : entiers {entiers contenus dans les liste}

    cout_total = 0
    Pour i de 1 à nbliste Faire
        val2 = valeur_liste(solution[i], 1) +1
        cout_total = cout_total + Dist[1][val2]
        Pour j de 2 à nbelem Faire
            val1 = val2
            val2 = valeur_liste(solution[i], j) +1
            cout_total = cout_total + Dist[val1][val2]
        Fin Pour
        cout_total = cout_total + Dist[val2][1]
    Fin Pour

    Retourner cout_total
Fin Fonction
```

A partir de tous nos algorithmes, il nous sera possible de résoudre le problème.

4. Résolution du problème

Nous allons maintenant résoudre le problème à l'aide des algorithmes vus précédemment. Mais tout d'abord, nous avons besoin de récupérer les données. Pour cela, nous créons un algorithme permettant de les lire. Nous verrons ensuite comment résoudre le problème.

4.1. Lecture des données

Nous avons besoin de lire le nombre de clients, les demandes des clients, la capacité maximale d'une voiture et enfin les distances séparant les clients mais aussi celles avec le dépôt et les clients.

Nous partageons cet algorithme en trois étapes :
- le nombre des clients et la capacité des voitures
- la demande des clients
- les distances

Commençons par la lecture des distances. Nous avons réalisé une fonction pour cela :

```
Fonction Lecture_Dist(N) : matrice de réels

    Donnée : N : entier {nombre de clients}

    Locales : i,j : entiers {indices de parcours}
              Dist : matrice de réels de taille (N+1)x(N+1) {distances}

    Pour i de 1 à N+1 Faire
        Pour j de 1 à N+1 Faire
            Lire Dist[i][j]
        Fin Pour
    Fin Pour
    Retourner Dist
Fin Fonction
```

De même pour la lecture de la demande des clients, voici la fonction permettant cette lecture :

```
Fonction Lecture_Quantite(N) : tableau d'entiers

    Donnée : N : entier {nombre de clients}

    Locales : i : entier {indice de parcours}
              Quantite : tableau d'entiers de taille N {demande des clients}

    Pour i de 1 à N Faire
        Lire Quantite[i]
    Fin Pour
    Retourner Quantite
Fin Fonction
```

Pour finir, il faut rassembler ces deux fonctions dans une procédure de lecture. La lecture du nombre des clients et celle de la capacité maximale des voitures se fera dans cette procédure :

Procédure Lecture (N, Q, Quantite, Dist,)

Données/Résultats : N : entier {nombre de clients}
Q : entier {capacité maximale d'une voiture}
Quantite : tableau[N] d'entiers {demande}
Dist : matrice[N+1][N+1] de réels {Distances}

Lire N
Lire Q
Quantite = Lecture_Quantite(N)
Dist = Lecture_Dist(N)

Fin Procédure

Ainsi, nous obtenons les données nécessaires à la résolution du problème.

4.2. Résolution

Pour la résolution du problème, nous aurons besoin des différents algorithmes de résolution. De plus, pour appliquer l'algorithme du Tour Géant, celui-ci a besoin d'avoir un numéro client où il pourra débuter. Pour l'obtenir, nous ajoutons une lecture avant l'appel de l'algorithme du Tour Géant.

A la fin de cette résolution, nous obtiendrons le coût total des déplacements des voitures pour livrer tous les clients tout en réduisant les parcours.

Voici la procédure principale permettant de résoudre le problème :

Procédure Résolution()

Locales : N : entier {nombres de clients}
Q : entier {capacité maximale d'une voiture}
Quantite : tableau[N] d'entiers {demande}
Dist : matrice[N+1][N+1] de réels {Distances}
Ci : entier {premier client du tour géant}
Tour_géant : tableau[N] d'entier
H : table des successeurs {sous graphe auxiliaire}
PCC : Parcours {plus court chemin}
Solution : tableau de liste chaînée d'entier {chaque parcours que doivent effectuer chaque voiture}
nbliste : entier {nombre de parcours dans Solution}
cout_total : réel {coût total}

{Lecture des données}
Lecture(N, Q, Quantite, Dist)
{Tour Géant}
Lire Ci
Tour_géant = Tour_géant(Ci, Dist, N)

{Sous Graphe auxiliaire}
H = SPLIT(Tour_géant, Q, N, Dist, Quantite)

{Recherche du plus court chemin}
PCC = Dijkstra(H, N+1)

{Décodage}
Solution = Decodage(PCC, nbliste, tour_géant)

{Calcul du coût total}
calcul_total = calcul_cout(Solution, nbliste, Dist)
Afficher calcul_total

Fin Procédure

Ainsi, à partir de cette procédure, nous obtenons le cout total de la meilleure solution de parcours.

4.3. Tests effectués

Concernant les tests, nous avons commencé avec le petit fichier reprenant l'exemple du polycopié contenant 5 clients. Nous obtenons donc un coût total de 205 avec 1 comme premier client.

Puis nous avons réalisé nos tests sur de grandes données : nous avons donc essayé notre programme sur l'ensemble des fichiers .dat fournis (cvrp_100).
Nous obtenons pour chacun des fichiers le bon résultat :

```
/home/gis3/tjongman/Graphes/projet_SDGrapheTest$ ./main < cvrp_100_1_det.dat
Cout total : 14162.330000
/home/gis3/tjongman/Graphes/projet_SDGrapheTest$ ./main < cvrp_100_1_r.dat
Cout total : 1014.550000
/home/gis3/tjongman/Graphes/projet_SDGrapheTest$ ./main < cvrp_100_1_c.dat
Cout total : 882.290000
```

Les clients fournis pour le tour géant sont respectivement 3, 1, 1.

Nous obtenons donc les bonnes valeurs.

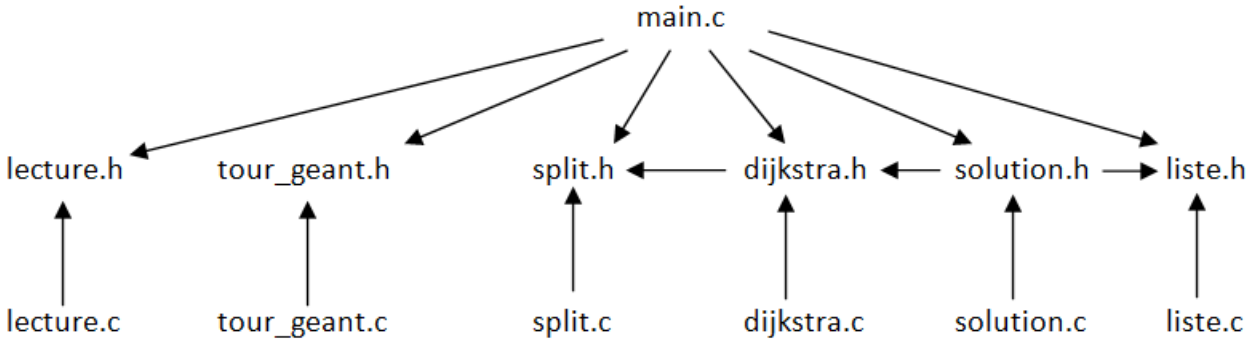
5. Fichiers

5.1. Listing des fichiers

Voici la liste des fichiers que nous avons réalisé durant le projet :

- main.c
- lecture.h
- lecture.c
- tour_geant.h
- tour_geant.c
- split.h
- split.c
- dijkstra.h
- dijkstra.c
- liste.h
- liste.c
- solution.h
- solution.c
- makefile
- exemple.dat
- cvrp_100_1_det.dat
- cvrp_100_1_r.dat
- cvrp_100_1_c.dat

5.2. Schéma des inclusions des fichiers



Une flèche signifie que le fichier pointé est inclus dans celui qui pointe.

5.3. Explication pour la compilation et le résultat de la résolution

Nous avons réalisé un makefile permettant de compiler tous nos fichiers. Il suffit d'entrer la commande : make
Il en sortira un exécutable appelé main. Il ne reste plus qu'à utiliser < afin de rédiriger la sortie des fichiers .dat dans main.
Si des fichiers .o sont déjà présents, il suffit de les supprimer avec la commande : make clean, puis de refaire make

Dans les fichiers dat, il est nécessaire de rajouter à la fin le client qui sera donné au tour géant. Dans les fichiers dat que nous avons fourni, ce nombre y est déjà inclus.

Conclusion

Nous avons modélisé ce projet comme un problème de graphe que l'on a traduit sous la forme d'une recherche de plus court chemin. Avec nos structures et nos algorithmes nous avons déterminé ce chemin et nous connaissons le nombre de véhicules nécessaires ainsi que le coût total parcouru, en fonction de la quantité maximale Q qu'ils peuvent transporter et de la quantité qi à livrer au client Ci ($i \in [1..N]$).
Pour conclure, nous avons répondu à l'ensemble des attentes (obtenir le coût total des véhicules).

Concernant les améliorations, il serait possible de déterminer le coût total minimum. En effet, celui-ci dépend du client fourni à l'algorithme du Tour géant. Il aurait fallu donc d'appliquer notre programme pour chacun des clients afin d'obtenir le coût total minimum malgré que cette méthode soit très coûteuse.

Bilans personnels

Florian :

Le choix des structures de données fut difficile, en prenant en compte les meilleurs structures possibles et celles qui sont les plus faciles à implémenter. Nous avons pu constater lors de l'utilisation du débugger que certaines méthodes était vraiment plus coûteuses, remettant en cause nos choix traitements concernant celles-ci.

Tony :

L'analyse du problème fût longue à comprendre. Cependant une fois comprise, il m'était plus facile de visualiser les étapes de la résolution du problème.
Concernant la choix de la structure de données, il était nécessaire de voir tous les contraintes de chacune d'elles pour choisir la meilleure.