Problème de tournées avec fenêtres de temps

1 Problème initial

Il s'agit de calculer la tournée la plus courte passant par un certain nombre de points de livraison, chaque livraison devant être effectuée dans une fenêtre de temps donnée (par exemple, entre 8h et 10h). On cherchera en pratique à calculer la tournée la plus courte respectant les contraintes de précédence imposées par les fenêtres de temps : si deux livraisons i et j doivent être effectuées dans deux fenêtres de temps f_i et f_j telles que f_i est antérieure à f_j, alors la tournée devra passer par i avant de passer par j. Si la tournée la plus courte respectant ces contraintes de précédence ne respecte pas les fenêtres de temps, alors la solution devra être adaptée :

- si pour un point i, l'heure d'arrivée sur i est antérieure au début de la fenêtre de temps f_i, alors le livreur pourra faire une pause (ou prendre son temps pendant la livraison précédente, ou décaler son heure de départ...) ;
- si pour un point i, l'heure d'arrivée sur i est postérieure à la fin de la fenêtre de temps f_i, alors il y a un problème qui devra être résolu par l'agent de la société de livraison (qui devra prendre contact avec les clients pour décaler certaines livraisons).

Plus formellement, les données en entrée du problème à résoudre sont :

- un graphe $G = (V, A)$ décrivant le plan de la ville, où V est un ensemble de sommets et $A \subseteq V \times V$ un ensemble d'arcs ;
- une fonction $d : A \to \mathbb{R}^+$ associant une durée d_{ij} à chaque arc (i, j) ;
- un entrepôt $e \in V$;
- un ensemble de points de livraison $L \subseteq V$;
- un nombre de fenêtres de temps $F \in \mathbb{N}$;
- une fonction $f : L \to [1; F]$ associant une fenêtre de temps f_i à chaque point de livraison $i \in L$.

A partir de ces données en entrée, il s'agit de trouver un circuit c de G partant de l'entrepôt e et revenant sur e tel que

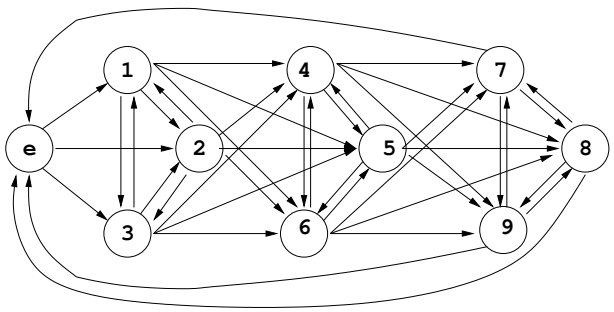
- chaque sommet de L apparaît au moins une fois dans c ;
- pour chaque paire de sommets $(i, j) \in L \times L$ telle que $f_i < f_j$, le circuit c passe par i avant de passer par j ;
- la somme des durées des arcs empruntés par c soit minimale.

2 Prétraitement

Pour simplifier le problème, on peut commencer par calculer un nouveau graphe $G' = (V', A')$ tel que

- $V' = L \cup \{e\}$
- $A' = \{(e, i) | i \in L, f_i = 1\} \cup \{(i, j) \in L \times L | f_i = f_j\} \cup \{(i, j) \in L \times L | f_j = f_i + 1\} \cup \{(i, e) | i \in L, f_i = F\}$

Considérons par exemple le problème initial de la figure 1, pour lequel il y a 3 fenêtres de livraison telles que $f(1) = f(2) = f(3) = 1$, $f(4) = f(5) = f(6) = 2$ et $f(7) = f(8) = f(9) = 3$. Le graphe G' est :



On calcule ensuite, pour chaque arc $(i, j) \in A'$, le plus court chemin π_{ij} de i vers j dans le graphe initial $G = (V, A)$. On définit enfin la fonction $d' : A' \to \mathbb{R}^+$ associant à chaque arc (i, j) de G' la longueur du plus court chemin π_{ij} dans G. Pour résoudre le problème initial, on peut alors chercher un circuit

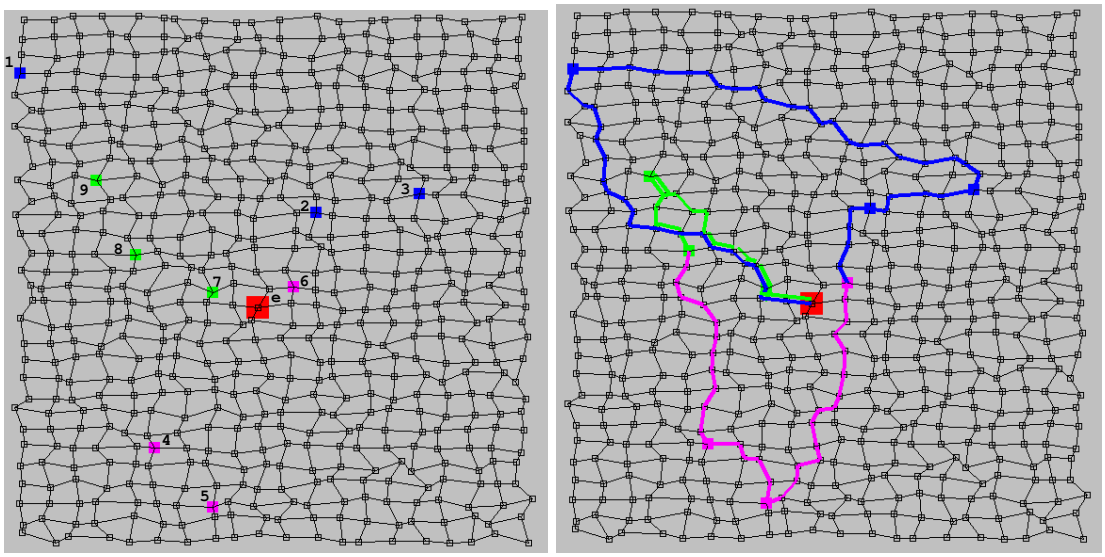


FIGURE 1 – Exemple de problème de tournée (à gauche), et de solution (à droite). Dans cet exemple, on suppose que la durée d_{ij} associée à chaque arc (i, j) est proportionnelle à la distance euclidienne entre i et j.

hamiltonien de G' partant de e et revenant sur e tel que la somme des durées des arcs soit minimale. On reconstruit ensuite la tournée dans G à partir de ce circuit en remplaçant chaque arc du circuit dans G' par le plus court chemin correspondant dans G.

Dans l'exemple précédent, le circuit hamiltonien $< e, 1, 3, 2, 6, 5, 4, 8, 9, 7, e >$ du graphe G' correspond à la solution dessinée dans la partie droite de la figure 1.

3 Modèles mathématiques pour le voyageur de commerce

Le problème de la recherche d'un circuit hamiltonien (passant par chaque sommet une et une seule fois) de coût minimal est bien connu sous le nom de problème du voyageur de commerce.

Modèles linéaires. Il existe pas mal de modèles linéaires en nombres entiers (certains sont des modèles mixtes qui utilisent également des variables définies sur les réels). Le plus ancien date de (Dantzig, Fulkerson and Johnson (1954)). Dans tous les cas, une variable binaire $x_{ij} \in \{0, 1\}$ est associée à chaque arc $(i, j) \in A'$. La variable x_{ij} a pour valeur 1 ssi l'arc (i, j) appartient à la solution. L'objectif est de minimiser la fonction suivante :

$$\sum_{(i,j)\in A'} d'_{ij} \cdot x_{ij}$$

Tous les modèles imposent qu'il y ait exactement un arc sortant et un arc entrant pour chaque sommet, *i.e.*,

$$\forall i \in V' : \sum_{j\in V',(i,j)\in A'} x_{ij} = 1 \text{ et } \sum_{j\in V',(j,i)\in A'} x_{ji} = 1$$

Il faut ensuite ajouter des contraintes pour assurer que la solution soit un circuit élémentaire (élimination des sous-tours). On peut faire cela, par exemple, en ajoutant la contrainte

$$\sum_{(i,j)\in A'\cap U\times U} x_{ij} \leq |U| - 1$$

pour tout sous-ensemble de sommets $U \subset V'$ tel que $|U| \geq 2$. Evidemment, cela implique un nombre exponentiel de contraintes (une par sous-ensemble de V' de taille supérieure ou égale à 2). En pratique, ces contraintes sont ajoutées sous la forme de coupures (cut) dans un processus de résolution par "Séparation et coupure" (Branch & Cut). Plus précisément, appelons P_0 le problème sans aucune contrainte

d'élimination de sous-tour. Dans un premier temps, le problème P_0 est résolu ; si la solution de P_0 ne contient qu'un seul circuit (pas de sous-tour), alors le problème initial est résolu ; sinon, on crée le problème P_1 en ajoutant à P_0 les contraintes nécessaires pour interdire les sous-tours de la solution de P_0. Le problème P_1 est alors résolu. De nouveau, si la solution de P_1 ne contient qu'un seul circuit, alors le problème initial est résolu ; sinon on crée un nouveau problème P_2 en ajoutant à P_1 les contraintes nécessaires pour interdire les sous-tours de la solution de P_1. Ce principe d'ajout de contraintes est itéré jusqu'à trouver une solution ne contenant qu'un seul circuit.

Modèle non linéaire classiquement utilisé en programmation par contraintes. Posons $n = |V'|$ et supposons que les sommets de V' sont numérotés de 0 à $n-1$. Pour chaque sommet $i \in V'$, nous créons 2 variables :

- La variable $next_i$ donne le sommet visité après le sommet i et prend sa valeur dans l'ensemble des sommets successeurs de i dans G'.
- La variable $cost_i$ donne la durée de l'arc $(i, next_i)$.

Les contraintes à satisfaire sont :

- Le coût associé à i est égal à la durée de l'arc $(i, next_i)$: $\forall i \in V, cost_i = d'_{i,next_i}$
- Le tableau $next$ définit un circuit : $circuit(next)$

La fonction objectif à minimiser est la somme des durées, *i.e.* $min \sum_{i \in V'} cost_i$

Sur l'exemple de la figure 1, la solution correspond à

$next$	1	3	6	2	8	4	5	0	9	7
	0	1	2	3	4	5	6	7	8	9

Les langages de programmation par contraintes permettent de décrire ce genre de modèle à base de contraintes, puis de le résoudre en faisant appel à des algorithmes de résolution par séparation, propagation et évaluation (Branch & Propagate & Bound) qui sont intégrés au système. Pour le projet DevOO, vous pouvez utiliser Choco qui est une bibliothèque de programmation par contraintes en Java. Le programme Choco qui résoud le voyageur de commerce est donné ci-dessous.

```
// Création des données
int n = ...; // Nombre de sommets
int minCost = ...; // longueur minimale d'un arc
int maxCost = ...; // longueur maximale d'un arc
int[][] cost = ...; // cost[i][j] = longueur de l'arc (i,j)
int[][] succ = ...; // succ[i] = tableau contenant l'ensemble des sommets successeurs de i

// Création du solveur
Solver solver = new Solver();

// Déclaration des variables
IntVar[] xNext = xNext = new IntVar[n];
for (int i = 0; i < n; i++)
    xNext[i] = VariableFactory.enumerated("Next " + i, succ[i], solver);
IntVar[] xCost = VariableFactory.boundedArray("Cost ", n, minCost, maxCost, solver);
IntVar xTotalCost = VariableFactory.bounded("Total cost ", n*minCost, bound - 1, solver);

// Déclaration des contraintes
for (int i = 0; i < n; i++)
    solver.post(IntConstraintFactory.element(xCost[i], cost[i], xNext[i], 0, "none"));
solver.post(IntConstraintFactory.circuit(xNext,0));
solver.post(IntConstraintFactory.sum(xCost, xTotalCost));

// Résolution
solver.set(IntStrategyFactory.firstFail_InDomainMin(xNext));
solver.findOptimalSolution(ResolutionPolicy.MINIMIZE,xTotalCost);
```