RAPPORT D'INGÉNIEUR
PROJET DE TROISIÈME ANNÉE
FILIÈRE F4 : CALCUL ET MODÉLISATION SCIENTIFIQUES

Problème de Gestion de Personnel Soignant en Soins à Domicile

---

Présenté par : Sandy LE PAPE

Tuteurs de projet :
Matthieu GONDRAN &
Marina VINOT

Professeurs référent :
Philippe LACOMME &
Christophe DUHAMEL

Date de la soutenance : mercredi 13 Mars 2019
Durée théorique du projet : 120h

Année scolaire 2018-2019

Remerciements

J'aimerais tout d'abord remercier mon tuteur principal de projet, Matthieu GONDRAN, pour sa disponibilité et les nombreuses aides qu'il a su régulièrement m'apporter. Qu'il m'ait laisser le temps d'appréhender les nouvelles notions et implémenter les algorithmes de base a été déterminant pour bien initier le projet. J'aimerais également remercier Philippe LACOMME. Sa considération pour mon travail et les discussions que j'ai eu ont contribué à la motivation et l'investissement nécessaires à cette dernière année scolaire. Mes remerciements vont également à Marina VINOT et Christophe DUHAMEL, respectivement tutrice associée et tuteur référent du projet. Enfin, j'aimerais remercier l'ISIMA, qui m'a permis non seulement d'acquérir les bases en Recherche Opérationnelle indispensables à la pleine compréhension du projet, mais aussi de travailler de longues heures au sein de ses locaux. Enfin, je tenais à remercier le LIMOS et l'ANR qui m'ont permis de conduire le début de ce projet.

Table des illustrations

Résumé

Le nombre et la diversité des services hospitaliers de soins à domicile grandissant, le problème de planification concernant la gestion du personnel soignant se complexifie grandement. On regroupe communément sous l'appellation *Workforce Scheduling* ce type de problème.

Ce projet m'a permis, entre autres, de me familiariser avec l'implémentation d'un modèle précurseur des problèmes de planification : le TSP. Sa version modèle linéaire devenant vite limitée, un modèle PPC a été jugé plus adapté à la situation. Les contraintes de fenêtres de temps et de minimisation du temps de parcours ont été les principales contraintes visées. Une adaptation de ce travail pour le VRP a ensuite été initié.

Mots-clés : *Workforce Scheduling*, TSP, modélisation linéaire, PPC, VRP.

Abstract

As the number and the diversity of hospital care at home grows by, the planning problem concerning caregivers' management is considerably becoming more complex. We commonly gather under appelation *Workforce Scheduling* this type of problem.

This project enables me, inter alia, to familiarize with the implementation of a precursor model of planning problems : the TSP. The linear model version becoming quickly limited, a PPC model has been judged more convenient to the situation. Time windows contraints and minimization of the travelled time were the main constraints aimed. An adaptation of this work to the VRP has then been initiated.

Mots-clés : *Workforce Scheduling*, TSP, linear model, PPC, VRP.

Table des matières

Introduction

Ce projet m'a été confié par le LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes). C'est une Unité Mixte de Recherche du Centre National de la Recherche Scientifique français. Parmi les thèmes principaux de Recherche du laboratoire, on peut citer l'Optimisation des Systèmes, la modélisation de Problèmes de Transport, les Réseaux & Télécommunications ou encore les Images. Le laboratoire est intégré dans les locaux de l'Ecole d'Ingénieurs ISIMA (Institut Supérieur d'Informatique, de Modélisation et de leurs Applications), se situant sur le Campus des Cézeaux, à Aubière. Ce projet est financé par l'ANR (Agence Nationale pour la Recherche).

Compte tenu du nombre important de patients ne pouvant se déplacer, un personnel soignant à domicile proportionnel doit être mobilisé afin de répondre à toutes les demandes. Dès lors, la gestion de planification des services de soins livrés à domicile est de plus en plus rude à mettre en œuvre. Aussi une automatisation du fonctionnement devient-elle indispensable. Le problème peut ici être vu sous deux angles. D'une part, chaque membre du personnel soignant doit effectuer un certain quota d'heures de travail journalier et de rythme analogue à ses collègues. D'autre part, ce problème se veut avant tout au service des patients : n'étant pas forcément disponibles sur toute la journée, les passages doivent respecter des créneaux horaires précis pendant lesquels le personnel doit passer. D'autant plus que le patient, lorsqu'il est satisfait de l'infirmière, souhaiterait idéalement conserver cette même personne pour les soins à venir (en particulier lorsque plusieurs visites sont requises sur une même journée, comme ce peut être le cas pour des prises de sang, des injections,...). On remarque rapidement que plus le nombre de contraintes que l'on souhaiterait voir respectées est grand, plus la modélisation (et a fortiori son implémentation) sera difficile.

Ainsi, il serait déraisonnable de vouloir prétendre résoudre dans l'immédiat un tel problème : des simplifications du modèle s'imposent donc, dans le but de converger petit à petit vers le final, qui se trouve souvent être le plus complexe. Le premier venant naturellement à l'esprit est le VRP (pour *Vehicle Routing Problem* en anglais). L'autre probème, plus simple encore, est le TSP (pour *Travelling Salesman Problem* en anglais), et qui sera le premier objet de travail.

Dans une première partie, il conviendra donc de présenter les principales modélisations existantes dans la littérature scientifique, pouvant s'appliquer en l'occurrence sur ce type de problème. Dans un second temps, nous exposerons les travaux effectués sur le TSP, modèle le plus simple que l'on puisse trouver. Enfin, nous verrons comment les contraintes et méthodes de calcul utilisées ont pu être utilisées au profit d'un modèle plus complet au vu de notre objectif : le VRP.

Voici le diagrammes de Gantt prévisionnel (en grisé) et réel (en bleu) du projet, qui s'est étalé sur six mois. Les prévisions ont globalement été respectées, à ceci près qu'un effort un peu plus important était attendu sur le VRP.

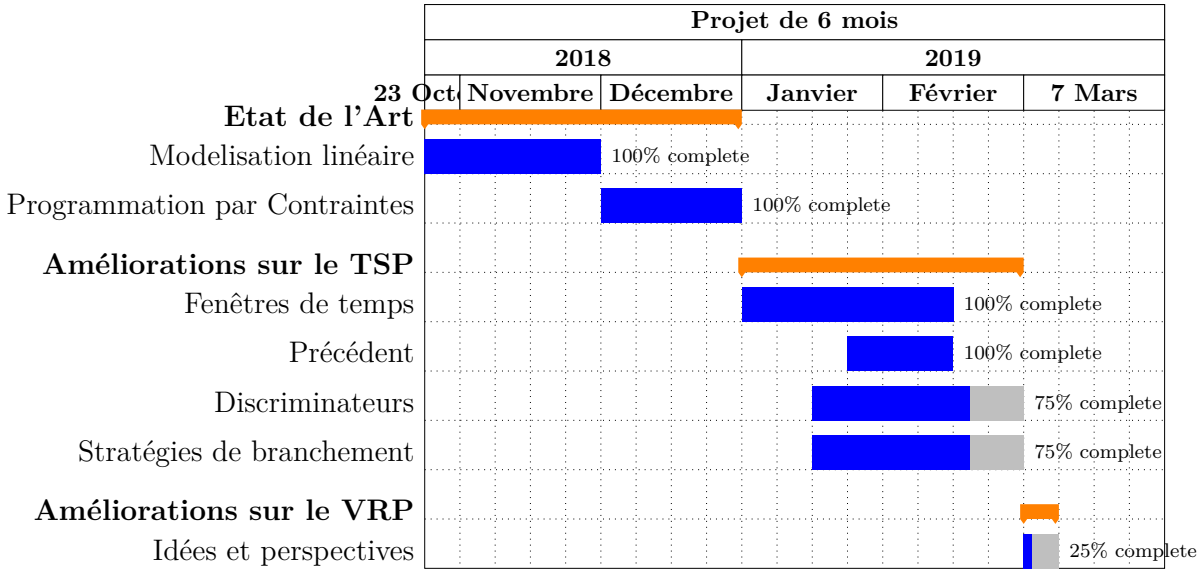


FIGURE 1 – Diagramme de Gantt

Certains extraits du livre [1] ont été précieux pour mes travaux, ainsi que les documentations [2], [3] et [4].

Enfin, le code principal du projet est accessible via le dépôt GIT suivant : `https://github.com/salepape`.

Chapitre 1

Etat de l'Art des principales modélisation

1.1 Modélisation type Programmation Linéaire

1.1.1 Principe

La Programmation Linéaire consiste à optimiser un problème dont la fonction à étudier est linéaire sur un polyèdre convexe. On souhaite soit la maximiser, soit la minimiser. On décrit les programmes linéaires en trois composantes :

— un ensemble de variables,

— un ensemble de contraintes linéaires, chacune exprimée sous forme d'une égalité ou d'une inégalité,

— une fonction linéaire à minimiser ou à maximiser, appelée fonction objectif.

Graphiquement, on visualise ce type de problème sous forme d'un polyèdre, formé par les droites représentant ses contraintes. Prenons pour exemple le programme linéaire suivant, où x_1, x_2 et d sont 3 variables de $\mathbb{R}$:

$$
\begin{aligned}
\max \quad & d = x_1 + x_2 \\
s.c. \quad & x_1 + 2x_2 \leq 7 && (1.1)\\
& 2x_1 + x_2 \leq 8 && (1.2)\\
& x_2 \leq 3 && (1.3)\\
& x_1 \geq 0 && (1.4)\\
& x_2 \geq 0 && (1.5)
\end{aligned}
$$

Comme le montre la FIGURE 1.1, l'ensemble des contraintes sont traduites par des droites, définissant le domaine de faisabilité (ie l'ensemble des solutions possibles que peut prendre le problème).

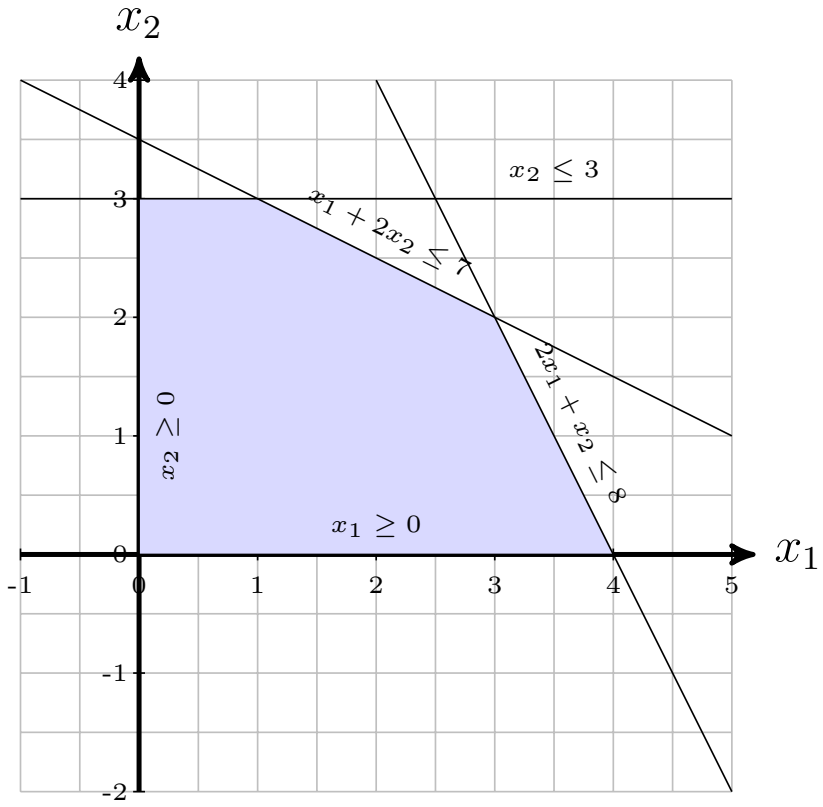


FIGURE 1.1 – Polyèdre des contraintes associé au problème linéaire

Ce type de modélisation a l'avantage d'être relativement intuitive. Il suffit très souvent de poser des variables de décisions associées aux données du problème pour obtenir rapidement une telle modélisation. Plus le nombre de contraintes est faible, plus le programme linéaire aura des chances de tomber sur l'optimum rapidement (on laisse le domaine des variables volontairement assez grand).

Il est maintenant temps de se familiariser avec les outils permettant d'implémenter de tels programmes.

1.1.2 Langage de modélisation, solveurs et outils de travail

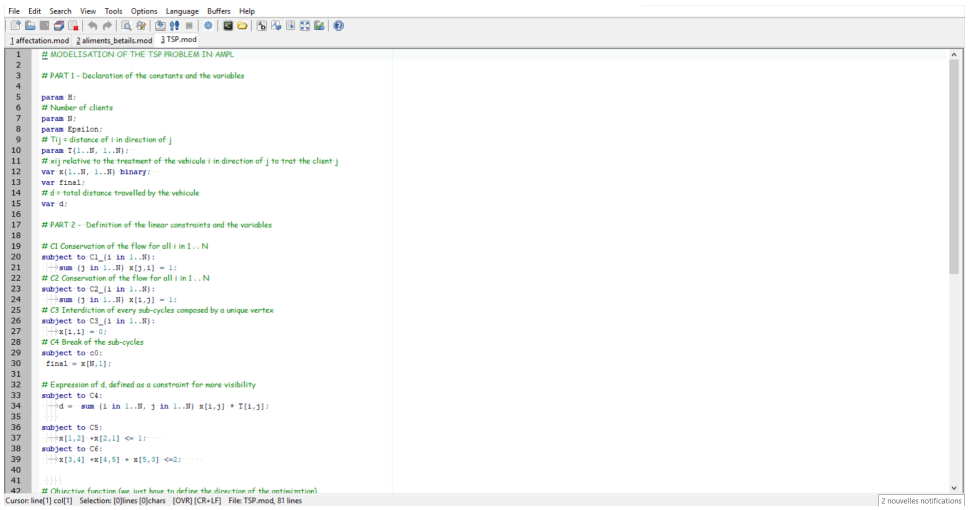
L'un des langages de modélisation (algébrique) les plus connus est l'AMPL (pour *A Modeling Language for Mathematical Programming* en anglais), et permet notamment de modéliser des problèmes linéaires. Il fournit ensuite le travail de résolution à un solveur adapté. Parmi eux, GPLK (pour *GNU Linear Programming Kit* en anglais) est largement

utilisé. C'est un ensemble de routines stockées dans une librairie, pouvant être appelées facilement pour résoudre des problèmes de grande instance.

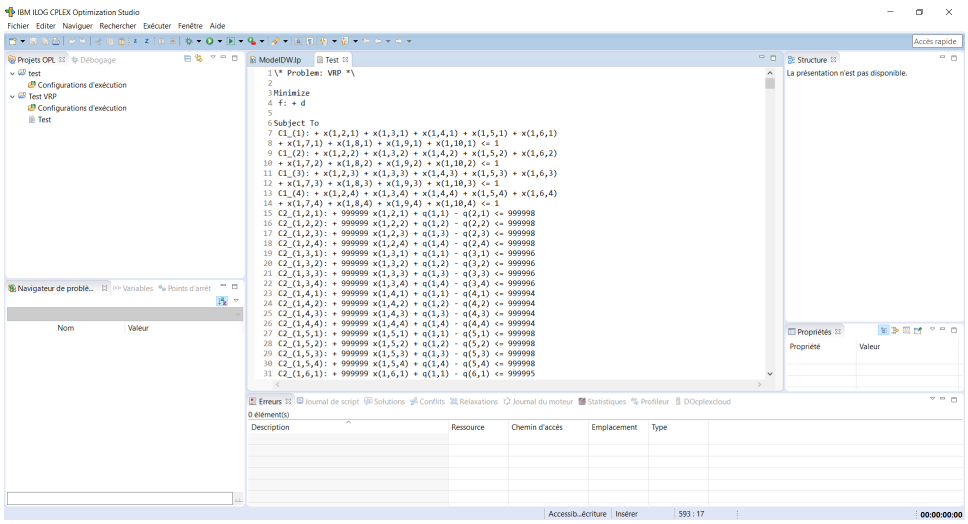
Parmi les outils munis d'une API (voir un aperçu FIGURE 1.2) et d'un IDE qui existent pour modéliser et résoudre les problèmes linéaires, ceux utilisés dans ce projet seront :

— Gusek : package libre de droit, implémentant une partie du langage AMPL, et lié au solveur GPLK (dans notre cas),

— CPLEX Optimizer : conçu par IBM, basé sur les langages OPL et AMPL, et disposant de son propre solveur.



(a) API de Gusek



(b) API *Concert Technology* de CPLEX Optimizer

FIGURE 1.2 – Interfaces Graphiques de Gusek et CPLEX Optimizer

Avec Gusek, les principales étapes pour écrire un programme linéaire sont associées à

des lignes de commandes caractéristiques :

- **Variables associées aux données :** `param` *nomVar* ;
- **Variables de décision :** `var` *nomVar* ;
- **Contraintes :** `subject to` *nomContrainte*_indicesUtilisés : exprContrainte ;
- **Fonction objectif et sens de minimisation :** `minimize` *nomFct* : *nomVar* ; (elle peut être vue comme l'ultime contrainte du problème)
- **Résolution :** `solve` ;
- **Affichage des solutions :** `display` *nomVar* ;
- **Données :** `data` ; puis en-dessous les données chiffrées.

Tout comme Gusek, CPLEX dispose également de sa propre API, intitulée la Concert Technology. Il est possible d'utiliser l'environnement Visual Studio pour profiter d'une palette plus grande de fonctionnalités que sur l'IDE de CPLEX Optimizer, et notamment l'utilisation du C++, via le simple import de répertoires `include` et de bibliothèques dans les propriétés du projet. Voici les principales parties que l'on peut retrouver :

- **Début du programme linéaire :** `ILOSTLBEGIN` ;
- **Déclaration de l'environnement :** instance de la classe `IloEnv` ;
- **Déclaration du modèle :** instance de la classe `IloModel` ;
- **Variables :** instance de la classe `IloNum` ;
- **Variables de décision :** les types de CPLEX `Ilo` ;
- **Contraintes :** stockées dans un tableau du type `IloRangeArray` ;
- **Fonction objectif et sens de minimisation :** `IloMinimize` *nomEnv* : *nomVar* ;
- **Résolution :** méthode `solve` de la classe `IloCP` ;
- **Affichage des solutions :** méthode `getValue` de la classe `IloCP` ;

Le principal inconvénient de la modélisation linéaire est qu'elle est souvent fastidieuse à écrire : de nombreuses sommes selon des indices précis,... Il est donc fréquent de passer sur une modélisation type Propagation de Contraintes : elle est moins intuitive mais a le mérite de s'écrire plus simplement.

1.2 Modélisation type Programmation par Contraintes

1.2.1 Principe

La Programmation par Contraintes consiste à étudier optimiser un problème de nature combinatoire sur un espace de recherche. Les composantes principales d'un tel problème sont les suivantes :

— un ensemble de variables et leur domaine de valeurs respectif,
— un ensemble de contraintes (qui peuvent être redondantes),
— une fonction à minimiser ou à maximiser, appelée fonction objectif,
— un ensemble de stratégies de recherche, que l'on va expliciter.

Pour chaque variable du problème est associé un intervalle de validité (aussi appelé domaine) dans lequel la variable prend sa valeur. Pour ce faire, un solveur type Programmation par Contraintes va choisir de se brancher sur une variable. Par brancher, il faut entendre qu'il va associer à cette variable soit l'une des valeurs du domaines, soit son complémentaire (ie toutes les autres). Ce branchement est dicté par un ensemble de stratégies, indiquant au solveur une manière spécifique de parcourir l'espace de recherche du problème.

Afin d'illustrer ce type de modélisation, prenons comme exemple le carré magique. Ce jeu consiste à placer n^2 nombres uniques, de telle sorte à ce que la valeur d'une case soit comprise entre 1 et n^2, et la somme des valeurs sur chaque ligne et chaque colonne soit égale à une seule et même constante. L'illustration ci-dessous (FIGURE 1.3) présente un résultat possible en dimension 5.

15	16	22	3	9
8	14	20	21	2
1	7	13	19	25
24	5	6	12	18
17	23	4	10	11

FIGURE 1.3 – Exemple de carré magique de taille 5

La formulation PPC pour remplir une case selon les règles stipulées ci-dessus est relativement simple, où on pose $c_{i,j}$ la variable associée à une case du carré magique et

cte la somme à respecter :

domaine de définition

$$c_{i,j} \in [0, (n-1)^2] \tag{1.6}$$

autres contraintes

$$c_{i,j} \neq c_{u,v} \qquad \forall u, v \in [|0, n-1|] \mid (i,j) \neq (u,v) \tag{1.7}$$

$$\sum_{j=0}^{n-1} c_{i,j} = cte \qquad \forall j \in [|0, n-1|] \tag{1.8}$$

$$\sum_{i=0}^{n-1} c_{i,j} = cte \qquad \forall j \in [|0, n-1|] \tag{1.9}$$

$$\sum_{i=0}^{n-1} c_{i,i} = cte \qquad \forall j \in [|0, n-1|] \tag{1.10}$$

$$\sum_{i=0}^{n-1} c_{n-i,n-i} = cte \qquad \forall j \in [|0, n-1|] \tag{1.11}$$

L'interprétation des contraintes est comme suit :

- (1.6) représente le domaine où $c_{i,j}$ va prendre ses valeurs,
- (1.7) impose l'unicité des valeurs dans la matrice,
- (1.8) force la somme des valeurs d'une ligne à valoir *cte*,
- (1.9) force la somme des valeurs d'une colonne à valoir *cte*,
- (1.10) force la somme des valeurs de la diagonale descendant vers la droite à valoir *cte*,
- (1.11) force la somme des valeurs de l'autre diagonale à valoir *cte*.

A l'aboutissement des calculs effectués, le solveur nous retourne la solution optimale, parmi toutes les solutions qu'il a trouvé. On appelle arbre de décisions l'arborescence illustrant le cheminement du solveur selon les stratégies adoptées.

Le principal avantage de la PPC est de pouvoir énumérer l'ensemble des solutions possibles et de trouver l'optimale dans de nombreux cas. Intéressons-nous maintenant au solveur Porgrammation Par Contraintes que nous allons utiliser par la suite.

1.2.2 Le solveur Choco

Choco est une librairie libre de droits (et française !) écrite en langage Java, dédiée à la Programmation par Contraintes, et disposant de son propre solveur. Elle définit

notamment ses propres types de variables et de contraintes (méthodes de classes Java) permettant d'avoir un contrôle important sur l'écriture de contraintes non linéaires, mais également ses stratégies de recherche permettant d'avoir la main sur le branchement de variables.

L'environnement de développement intégré NetBeans convient parfaitement pour construire des projets Java où le code est adapté selon les fonctions Choco. Une image de l'environnement utilisé est proposé en FIGURE 1.4 :

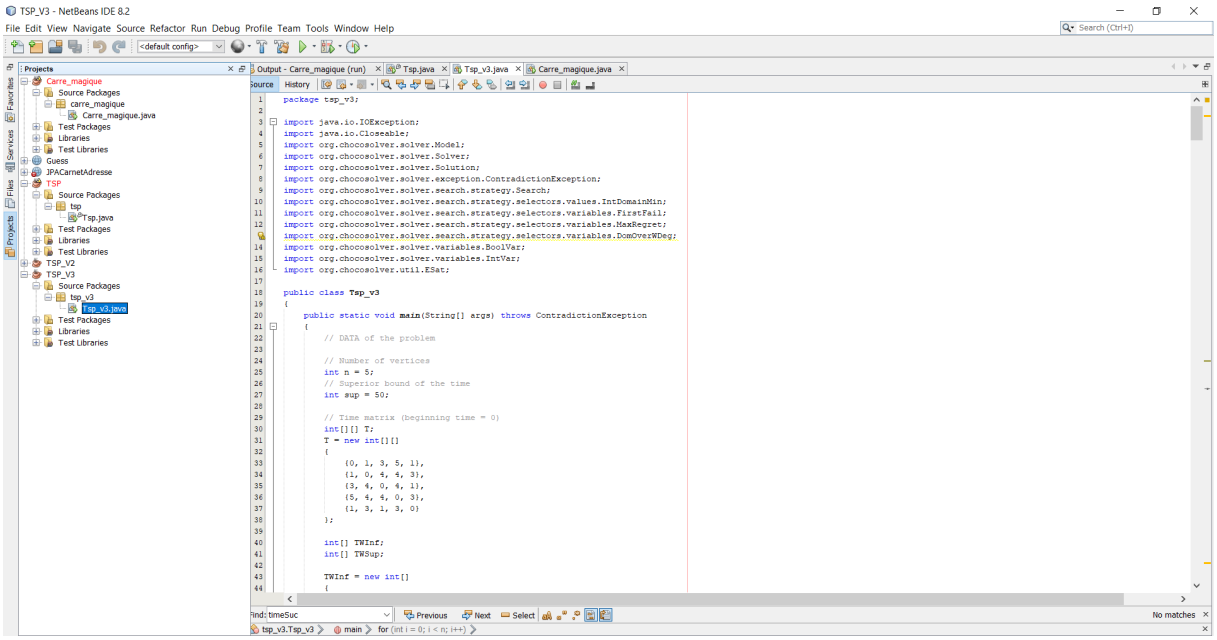


FIGURE 1.4 – Utilisation de Choco sous l'environnement NetBeans

Abordons dès à présent la forme générale d'un programme Choco. Sans se soucier des bonnes manières du Génie Logiciel, toutes les lignes de code s'écriront dans la fonction `main` d'une classe Java publique associée au problème à résoudre.

Ainsi, l'architecture principale d'un programme Choco s'articule de la sorte (un schéma-bilan est exposé en ANNEXE 1) :

- **Données :** comme pour le Java ;
- **Déclaration du modèle :** instance de la classe `Model` ;
- **Variables de décision :** variables Choco ;
- **Contraintes :** allocation de mémoire pour les variables déclarées précédemment puis remplissage des variables via une méthode de `Model` ;
- **Fonction objectif et sens de minimisation :** qui peut être vue comme l'ultime contrainte du problème ;
- **Résolution :** instance de la classe `Solver` ;
- **Affichage des solutions :** instance de la classe `Solution` ;

Il est également possible d'afficher de nombreuses informations, notamment l'historique des décisions prises par le solveur (ie l'arbre de décisions) par `showDecisionTree`. Une illustration de la lecture à avoir lorsque nous appelons cette commande est illustrée en FIGURE 1.5 :

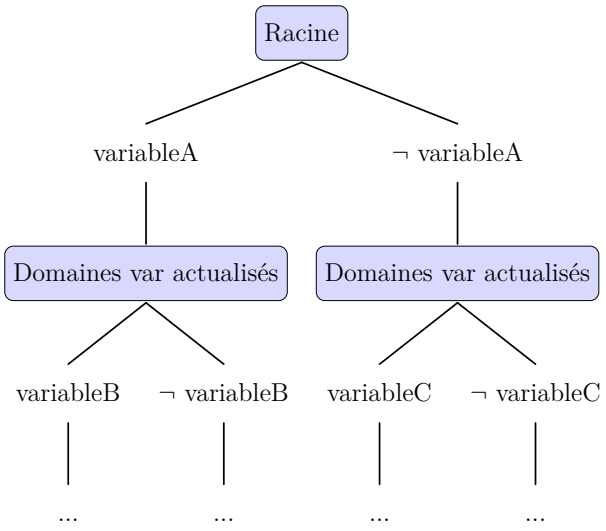


FIGURE 1.5 – Sortie de la commande `decisionTree` et arborescence à visualiser

On peut également afficher les statistiques principales pour chaque solution trouvée (réduites sur une ligne) au problème par `showShortStatistics`, comme illustré en FIGURE 1.6.

```
Model[Model-0], 1 Solutions, MINIMIZE d = 11, Resolution time 0,143s, 22 Nodes (153,4 n/s), 33 Backtracks, 17 Fails, 0 Restarts
d = 11
```

FIGURE 1.6 – Sortie de la commande `showShortStatistics`

D'autre part, certaines astuces de programmation permettent de réduire considérablement le nombre de noeuds parcouru par Choco, notamment la portées de certaines variables du problème. Reprenons le problème du carré magique. Cet exemple illustre en l'occurrence parfaitement la légitimité d'utiliser en priorité des variables globales plutôt que locales, comme l'illustre les résultats ci-dessous :

Model Global Constraints[Carre magique], 1 Solutions, Resolution time 0,038s, 39 Nodes (1 013,8 n/s), 39 Backtracks, 27 Fails, 0 Restarts Solution : $C_{00} = 24, C_{01} = 9, C_{02} = 22, C_{03} = 8, C_{04} = 2, C_{10} = 4, C_{11} = 7, C_{12} = 14, C_{13} = 17, C_{14} = 23, C_{20} = 1, C_{21} = 12, C_{22} = 16, C_{23} = 15, C_{24} = 21, C_{30} = 25, C_{31} = 19, C_{32} = 10, C_{33} = 5, C_{34} = 6, C_{40} = 11, C_{41} = 18, C_{42} = 3, C_{43} = 20, C_{44} = 13,$

Model Local Constraints[Carre magique], 1 Solutions, Resolution time 0,049s, 145 Nodes (2 946,6 n/s), 238 Backtracks, 133 Fails, 0 Restarts Solution : $C_{00} = 24, C_{01} = 8, C_{02} = 21, C_{03} = 10, C_{04} = 2, C_{10} = 4, C_{11} = 5, C_{12} = 20, C_{13} = 23, C_{14} = 13, C_{20} = 1, C_{21} = 19, C_{22} = 14, C_{23} = 9, C_{24} = 22, C_{30} = 25, C_{31} = 15, C_{32} = 7, C_{33} = 6, C_{34} = 12, C_{40} = 11, C_{41} = 18, C_{42} = 3, C_{43} = 17, C_{44} = 16,$

On constate ainsi une différence de noeuds parcourus de 106, ce qui est colossal pour un problème de taille 5. Cet aspect ne devra donc pas être négligé lors de nos futures implémentations (en particulier lorsque de grandes instances seront fournies à des problèmes dont la taille de l'ensemble de contraintes dépasse facilement les 20).

N.B. : il existe également le solveur CPLEX CP Optimizer d'IBM, conçu pour résoudre spécifiquement des problèmes type Propagation par Contraintes. Il ne sera pas utilisé dans ce projet.

Selon les problèmes considérés, le solveur tourne à l'infini et ne parvient pas à converger vers une solution. Cela est principalement dû au fait que la plupart des problèmes que l'on traite sont de classe NP-Complet. C'est pourquoi l'implémentation d'heuristiques et méta-heuristiques réglées pour un groupe d'instances choisi, peut donner des résultats très intéressants.

Abordons maintenant le modèle classique du TSP, que nous allons chercher à améliorer au fur et à mesure, de sorte à se rapporcher du modèle final visé.

Chapitre 2

Modélisation du TSP et ajout de contraintes

2.1 Présentation du TSP et modélisation linéaire associée

2.1.1 Présentation du TSP

Le problème du TSP (pour *Travelling Salesman Problem* en anglais), plus connu sous l'appellation Problème du Voyageur de Commerce en français, est l'un des problèmes les plus connus en Recherche Opérationnelle. Nous allons le présenter à la lumière du contexte du projet.

Considérons une infirmière devant fournir des soins à plusieurs patients au cours de sa journée de travail. Elle dispose de son propre véhicule de fonction pour se déplacer jusqu'au domicile des patients. De plus, on associe un sommet au lieu de l'hôpital (très souvent appelé dépôt dans ce type de problèmes). On supposera ici que :

— le (ou les) soin(s) que nécessite(nt) chaque client sont pour l'instant unitaires et ponctuels (ie sans durée) dans la journée ;
— il existe une route directe entre chaque client, et peut être empruntée dans les deux sens.

L'infirmière commence donc son service en partant de l'hôpital, arrive devant l'habitation d'un patient, le soigne, puis se déplace vers le suivant, et ainsi de suite. Lorsque tous les patients auront été soignés, l'infirmière revient au point de départ de sa course, c'est-à-dire l'hôpital. On appellera tournée un tel voyage, la route empruntée passant une et une seule fois devant les habitations de chaque patient.

Du point de vue logistique de l'hôpital, dans un souci de manque d'argent et de temps, l'objectif est donc de trouver une tournée, dont le temps de trajet entre le début et la fin du voyage est minimal. Formalisons maintenant le problème.

2.1.2 Modélisation mathématique du problème

L'outil mathématique adapté venant naturellement à l'esprit pour représenter les données du problème est un graphe. Soit $G = (V, E)$ un graphe où V représente l'ensemble des clients demandeurs de soins (ou plus précisément la localisation de leur habitation respective), et E l'ensemble des chemins qui les sépare. Voici le graphe obtenu pour 5 patients à soigner (FIGURE 2.1), où les arêtes peuvent être empruntées dans les 2 sens) et une tournée possible dans ce graphe (FIGURE 2.2) :

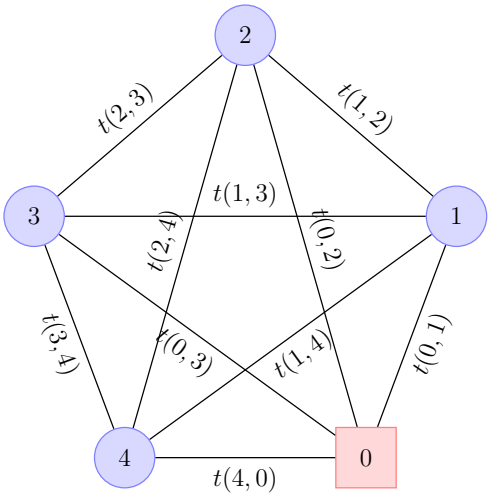


FIGURE 2.1 – Modélisation du TSP

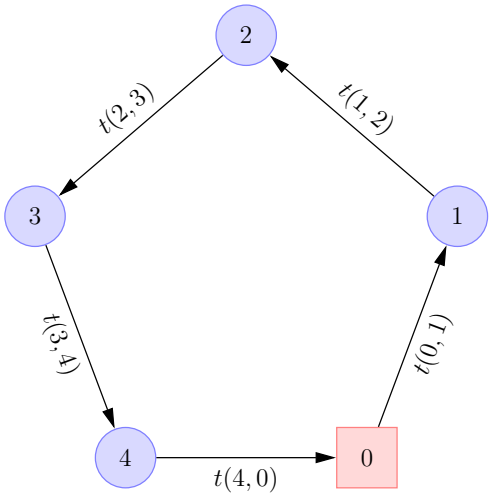


FIGURE 2.2 – Tournée possible

D'après les hypothèses, G devra donc être complet, et il conviendra de trouver le cycle hamiltonien de longueur minimal dans G.

D'un point de vue mathématique, on appellera donc tournée un cycle hamiltonien possible, et sera qualifiée d'optimale si parmi toutes les tournées existantes, elle s'avère être celle se bouclant avec le temps le plus court. Dans toute la suite du rapport, et afin de simplifier la lecture schématique, seule la tournée effectuée par le véhicule sera dessinée et non l'ensemble du graphe complet.

Enfin, d'après les hypothèses, la matrice des temps de trajet notée T sera carrée et symétrique. Les valeurs, notées T_{ij} ou $t(i, j)$, pourront être homogènes ou hétérogènes dans l'ensemble (patients habitant tous à proximité ou certains excentrés).

$$T = \begin{pmatrix} 0 & t(0,1) & t(0,2) & t(0,3) & t(0,4) \\ t(0,1) & 0 & t(1,2) & t(1,3) & t(1,4) \\ t(0,2) & t(1,2) & 0 & t(2,3) & t(2,4) \\ t(0,3) & t(1,3) & t(2,3) & 0 & t(3,4) \\ t(0,4) & t(1,4) & t(2,4) & t(1,3) & 0 \end{pmatrix}$$

Maintenant que les bases mathématiques sont posées, on peut construire le modèle linéaire.

2.1.3 Programme Linéaire associé et limites

L'ensemble des contraintes du programme linéaire sera d'autant plus important que nous rajouterons des contraintes logistiques dans le problème de départ. Avant de donner une écriture possible du programme linéaire concordant avec la modélisation exposée précédemment, fixons les notations et posons le problème :

Données 1

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- **n** le nombre de patients (ie $n = |V|$).

Variables de décision 1

- $\mathbf{x_{ij}} = \begin{cases} 1 & \text{si l'infirmière se déplace de i vers j pour traiter le patient j} \\ 0 & \text{sinon} \end{cases}$
- **d** la durée totale qui a été nécessaire pour que l'infirmière boucle sa tournée.

$$\min \quad d = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_{ij}.T_{ij} + x_{x_N 1}.T_{ij}$$

$$s.c. \quad \sum_{i=0}^{n-1} x_{ij} = 1 \qquad \forall i \in [|0, n-1|] \tag{2.1}$$

$$\sum_{i=0}^{n-1} x_{ij} = 1 \qquad \forall j \in [|0, n-1|] \tag{2.2}$$

$$x_{ii} = 0 \qquad \forall i \in [|0, n-1|] \tag{2.3}$$

$$x_{v_1 v_2} + x_{v_2 v_3} + ... + x_{v_t v_1} \leq t - 1 \qquad \forall \{v_1, ..., v_t\} \subset V \tag{2.4}$$

Voici la signification des contraintes :

- (2.1) impose l'unicité de l'arc entrant pour chaque noeud du graphe,
- (2.2) impose l'unicité de l'arc sortant pour chaque noeud du graphe,
- (2.3) casse les sous-tours de dimension 1,
- (2.4) casse les sous-tours.

▷ La minimisation porte sur la somme des durées de trajet qui ont été nécessaires pour effectuer la tournée.

La contrainte (2.4), la moins évidente à voir, est souvent appelée *contrainte de sous-tours*. Elle permet, comme son nom l'indique, d'empêcher l'algorithme de nous retourner un vecteur qui illustrerait la création de 2 sous-tours ou plus. Voilà ce qu'on pourrait obtenir comme solution (FIGURE 2.3) si la contrainte de sous-tours n'était pas ajoutée au problème linéaire :

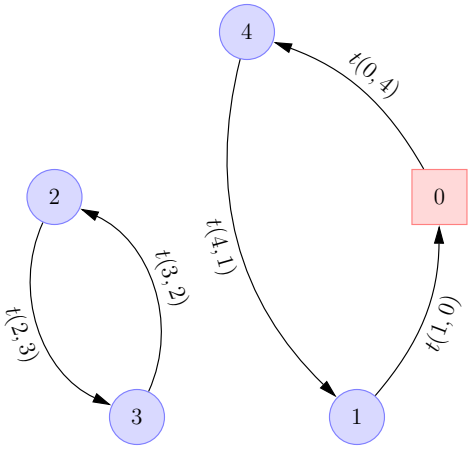


FIGURE 2.3 – Résultat possiblement obtenu sans la contrainte de sous-tours

Seulement, le nombre de contraintes ainsi générées par le problème est exponentiel. Cette formulation implique inévitablement un côut considérable en temps de calcul pour le solveur, si bien qu'il ne peut souvent pas toutes les générer. A part des solutions itératives, qui peuvent être implémentées avec CPLEX sous Visual Studio, ou des heuristiques s'adaptant au problème posé, le solveur est vite dépassé.

Ainsi, une solution itérative consistant à supprimer les sous-tours au fur et à mesure qu'ils apparaissent est efficace, permettant d'arriver à un tour optimal selon les instances. Elle sera plus rapide car en pratique, le nombre de contraintes générées ainsi est bien inférieure au nombre total de sous-tours créés. En Gusek, l'implémentation d'une telle architecture itérative est impossible, ce qui nous force à les préciser explicitement à la main à la fin du programme. Elle est en revanche possible sous CPLEX sous Visual Studio.

Voici les résultats que l'on peut obtenir sous CPLEX (FIGURE 2.4) :

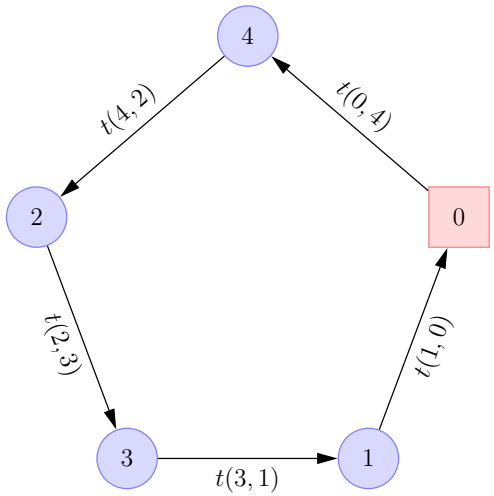


FIGURE 2.4 – Tournée possible obtenue

Les perspectives d'évolution du modèle linéaire devenant vite limitées au vu de tous les efforts fournis pour la seule contrainte de sous-tours, l'idée de passer par une modélisation type Programmation Par Contraintes permet d'avoir davantage d'élasticité au niveau de la gestion et de l'implémentation des contraintes.

2.2 Modélisation par Contraintes du TSP

2.2.1 Programme par Contraintes initial

La vision du problème est ici différente. Plutôt que d'avoir une généralisation importante du problème, on se donne un vecteur où l'on souhaiterait avoir une tournée possible de l'infirmière. A partir du vecteur, on va pouvoir écrire un programme qui va restreindre de plus en plus les valeurs possibles qu'il peut contenir. On introduit de nouvelles notations et on en déduit le programme par contraintes :

Données 2

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- **n** le nombre de patients (ie $n = |V|$).

Variables de décision 2

- $\mathbf{r_i} = k$ si le patient k est visité en position i dans la tournée qu'effectue l'infirmière ;
- **d** le durée totale qui a été nécessaire pour que l'infirmière boucle sa tournée.

$$
\begin{aligned}
\min \quad & d = \sum_{i=1}^{N-1} T_{r_i r_{i+1}} + T_{r_N r_1} & & \\
s.c. \quad & r_i \in [0, n-1] & \forall i \in [|0, n-1|] & \quad (2.5) \\
& r_i \neq r_j & \forall i, j \in [|0, n-1|] & \quad (2.6)
\end{aligned}
$$

Et ci-dessous la signification des contraintes :

- (2.5) représente le domaine où r_i va prendre ses valeurs,
- (2.6) impose l'unicité de tous les numéros de sommets dans le vecteur r.

▷ La minimisation porte sur la somme des durées de voyage entre chaque paire de sommets successifs, en ajoutant le coût de l'arc bouclant le tour (retour à l'hôpital).

Malheureusement, il est impossible en Choco d'écrire une expression où la variable est indicée selon l'indice courant et celui qui suit, d'où la nécessité d'adapter la formulation du problème précédent.

2.2.2 Programme par Contraintes adapté pour Choco

Afin de lui faire connaître l'indice suivant, nous allons donc créer un autre vecteur, qui sera le vecteur de la tournée qui succédera à la courante. Ainsi, on évite la difficulté imposée par Choco, mais au détriment d'une syntaxe plus lourde et moins intuitive. Pour illustrer le fonctionnement de ce type de programmation, nous utiliserons la tournée (extraite du graphe complet associé) ci-dessous :

Voici une représentation graphique (FIGURE 2.5) des vecteurs que l'on va crée et des liens entre eux :

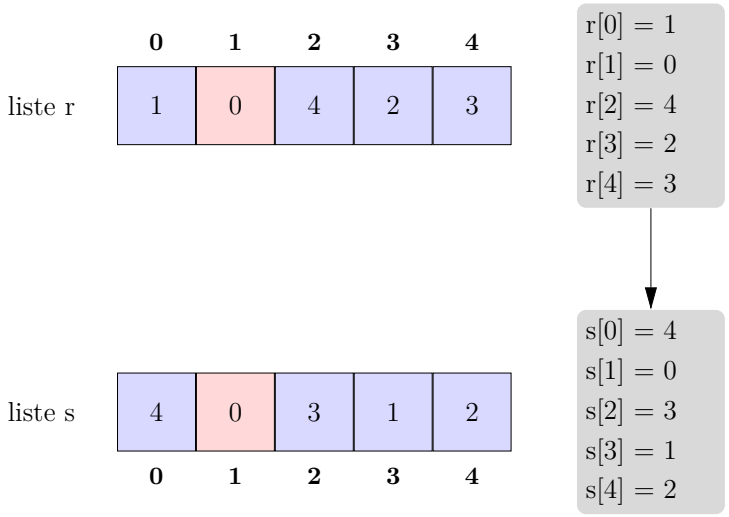


FIGURE 2.5 – Vecteurs tournée et successeur

Dans cette structure de données, on constate que, dans le vecteur tournée, le successeur du sommet 1 sera le sommet 0. Donc, la contenu de la case en position 1 du vecteur successeur sera 0. En aucun on ne définit par le vecteur s une tournée succédant à la tournée courante r : on ne fait qu'indiquer à Choco le sommet suivant du sommet courant dans la tournée qu'effectue l'infirmière via le vecteur s, du fait de ses limitations syntaxiques.

Une fois la structure de données trouvée pour s'adapter à Choco, le programme par contraintes devient :

Données 3

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- **n** le nombre de patients (ie $n = |V|$) ;

- **sup** la borne supérieure de temps.

Variables de décision 3

- $\mathbf{r_i} = k$ si le patient k est visité en position i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{s_i} = k$ si le patient k est visité à la suite du patient i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{tpsSuc_i}$ le temps de voyage entre le patient i et son successeur dans la tournée ;
- **d** la durée totale qui a été nécessaire pour que l'infirmière boucle sa tournée.

$$\min \quad d = \sum_{i=0}^{n-1} tpsSuc_i$$

s.c.

domaines de définition

$$r_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \tag{2.7}$$

$$s_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \tag{2.8}$$

$$tpsSuc_i \in [0, sup] \qquad \forall i \in [|0, n-1|] \tag{2.9}$$

$$d \in [0, (n-1) \times sup] \qquad \forall i \in [|0, n-1|] \tag{2.10}$$

autres contraintes

$$r_i \neq r_j \qquad \forall i, j \in [|0, n-1|] \tag{2.11}$$

$$s_i \neq s_j \qquad \forall i, j \in [|0, n-1|] \tag{2.12}$$

$$r_{i+1} = s_{r_i} \qquad \forall i \in [|0, n-2|] \tag{2.13}$$

$$tpsSuc_i = T_{is_i} \qquad \forall i \in [|0, n-1|] \tag{2.14}$$

$$s_i \neq i \qquad \forall i \in [|0, n-1|] \tag{2.15}$$

Et ci-dessous la signification des contraintes :

- (2.7) à (2.10) représente les domaines où les variables correspondantes vont prendre leurs valeurs,
- (2.11) impose l'unicité de tous les numéros de sommets dans le vecteur r,
- (2.12) impose l'unicité de tous les numéros de sommets dans le vecteur s,
- (2.13) donne au successeur sa définition par rapport au sommet succédant le courant,
- (2.14) fixe tpsSuc comme la distance d'un sommet i à son successeur,

- (2.15) casse les sous-tours de dimension 1.

▷ La minimisation porte sur la somme des durées de voyage entre chaque paire de sommets successifs.

Dans l'intention de se rapprocher vers un modèle de *Workforce Scheduling*, la première idée a été d'introduire dans le code les disponibilités horaires de chaque patient.

2.3 Introduction des fenêtres de temps

La première modification qui a été apportée est l'association à chaque patient d'une fenêtre de temps (*time windows* en anglais), traduisant sa disponibilité dans la journée. Le but serait toujours le même : minimiser le temps de trajet de l'infirmière, sachant qu'elle devra obligatoirement visiter chaque client dans un intervalle de temps prédéfini par chacun d'eux. C'est le problème du TSPTW (pour *Travelling Salesman Problem with Time Windows* en anglais).

N.B. : on souhaite ici calculer le temps de voyage minimal ; il est donc logique de vouloir considérer des dates au plus tôt. Chaque soin sera fourni dès que possible, et dans ce contexte, on supposera qu'aucune action prenant du temps s'effectuera entre la date de fin d'un soin, et la date de départ, d'où l'égalité des deux variables.

De plus, notons la distinction essentielle entre la date au plus tôt à laquelle le patient est disponible et la date d'arrivée de l'infirmière devant le domicile du patient. En effet, si elle arrive à l'avance, elle devra attendre que le patient soit disponible, sinon elle pourra directement le soigner.

Ces constats et l'introduction des fenêtres de temps entraîne un supplément de contraintes non négligeable. Quatre variables de décisions vont ainsi venir compléter le problème, qui s'écrit maintenant :

Données 4

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- **n** le nombre de patients (ie $n = |V|$) ;
- **sup** la borne supérieure de temps ;
- **TWInf** le vecteur contenant les bornes inférieures des fenêtres de temps ;
- **TWSup** le vecteur contenant les bornes supérieures des fenêtres de temps ;

- **dureeSoin** le vecteur contenant les durées de soins que chaque patient nécessite.

Variables de décision 4

- $\mathbf{r_i} = k$ si le patient k est visité en position i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{s_i} = k$ si le patient k est visité à la suite du patient i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{tpsSuc_i}$ le temps de voyage entre le patient i et son successeur dans la tournée ;
- **d** le durée totale qui a été nécessaire pour que l'infirmière boucle sa tournée ;
- **dateArr** la date d'arrivée de l'infirmière devant une habitation ;
- **dateDeb** la date de début du soin fourni au patient ;
- **dateFin** la date de fin du soin fourni au patient ;
- **dateDep** la date de départ de l'infirmière de l'habitation ;
- **dateArrSuc** la date d'arrivée de l'infirmière devant l'habitation suivante celle courante ;
- **dateDebSuc** la date de début du soin fourni au patient suivant celui courant ;
- **TWInfSucVal** la borne inférieure de la fenêtre de temps du patient suivant celui courant.

$$\min \quad d \;=\; \sum_{i=0}^{n-1} tpsSuc_i$$

s.c.

domaines de définition

$$r_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \tag{2.16}$$

$$s_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \tag{2.17}$$

$$tpsSuc_i \in [0, sup] \qquad \forall i \in [|0, n-1|] \tag{2.18}$$

$$d \in [0, (n-1) \times sup] \qquad \forall i \in [|0, n-1|] \tag{2.19}$$

$$dateArr_i \in [0, TWSup_i] \qquad \forall i \in [|0, n-1|] \tag{2.20}$$

$$dateDeb_i \in [TWInf_i, TWSup_i] \qquad \forall i \in [|0, n-1|] \tag{2.21}$$

$$dateFin_i \in [TWInf_i, TWSup_i] \qquad \forall i \in [|0, n-1|] \tag{2.22}$$

autres contraintes

$$r_i \neq r_j \qquad \forall i, j \in [|0, n-1|] \tag{2.23}$$

$$s_i \neq s_j \qquad \forall i, j \in [|0, n-1|] \tag{2.24}$$

$$r_{i+1} \neq s_{r_i} \qquad \forall i \in [|0, n-2|] \tag{2.25}$$

$$tpsSuc_i \neq T_{is_i} \qquad \forall i \in [|0, n-1|] \tag{2.26}$$

$$s_i \neq i \qquad \forall i \in [|0, n-1|] \tag{2.27}$$

$$dateArr_{s_i} = dateFin_i + tpsSuc_i \qquad \forall i \in [|0, n-1|] - r_{n-1} \tag{2.28}$$

$$dateArr_{s_i} = 0 \qquad si\ i = r_{n-1} \tag{2.29}$$

$$dateDeb_{s_i} = \max(dateArr_{s_i}, TWInf_{s_i}) \qquad \forall i \in [|0, n-1|] \tag{2.30}$$

$$dateFin_i = dateDeb_i + dureeSoin_i \qquad \forall i \in [|0, n-1|] \tag{2.31}$$

Et ci-dessous la signification des contraintes :

- (2.16) à (2.22) représente les domaines où les variables correspondantes vont prendre leurs valeurs,
- (2.23) à (2.27) désignent les mêmes contraintes que décrites précédemment,
- (2.28) impose la définition de la date d'arrivée pour tous les sommets sauf l'hôpital,
- (2.29) impose la définition de la date d'arrivée pour l'hôpital,
- (2.30) force l'infirmière à attendre le patient devant sa résidence s'il n'est pas disponible,
- (2.31) impose la définition de la date de fin d'un soin pour tous les sommets.

▷ La minimisation porte toujours sur la somme des durées de voyage entre chaque paire de sommets successifs.

Voici un exemple de schématisation du TSP avec les nouvelles variables entrant en jeu (FIGURE 2.6) :

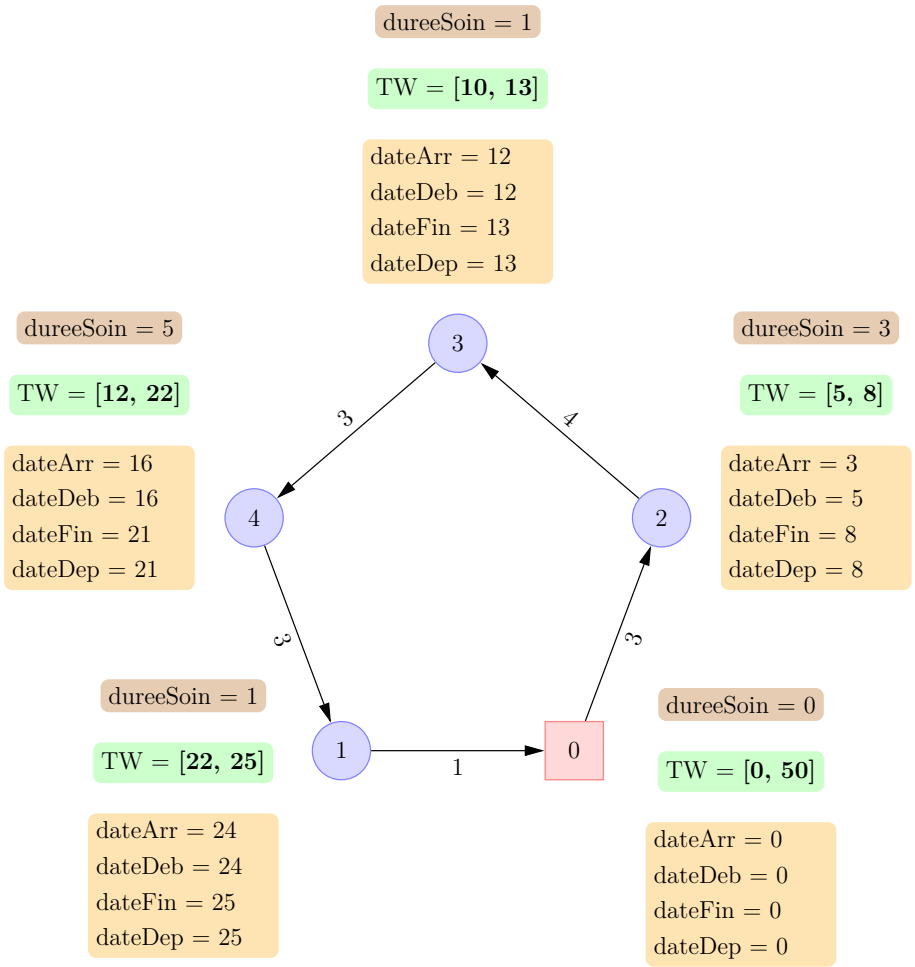


FIGURE 2.6 – Modélisation du TSP avec les fenêtres de temps et les dates caractéristiques

Maintenant que nous avons fixé les nouvelles contraintes du problèmes (associées aux fenêtres de temps), tout l'objectif va être d'aider tant que faire se peut le solveur. Une des idées a été d'implémenter le miroir des successeurs : les prédécesseurs.

2.4 Introduction de prédécesseurs

Encore une fois, afin de faciliter les choix du solveur Choco, il est intéressant d'introduire la notion de prédécesseurs, qu'on l'on définit en miroir par rapport aux successeurs. Le solveur va ainsi pouvoir déduire des décisions à l'aide du successeur et du prédécesseur de chaque sommet courant, et ce dans un ensemble plus restreint. Théoriquement, on augmente ainsi les chances de trouver rapidement une solution. Cette initiative peut

paraître redondante au premier abord, mais il faut garder à l'esprit que plus le problème a de contraintes, plus le solveur est contraint dans ses choix et, a fortiori, plus le solveur convergera vite vers une résolution (sauf si le problème est devenu trop contraint).

Voici la définition des prédécesseurs illustrée (FIGURE 2.7) :

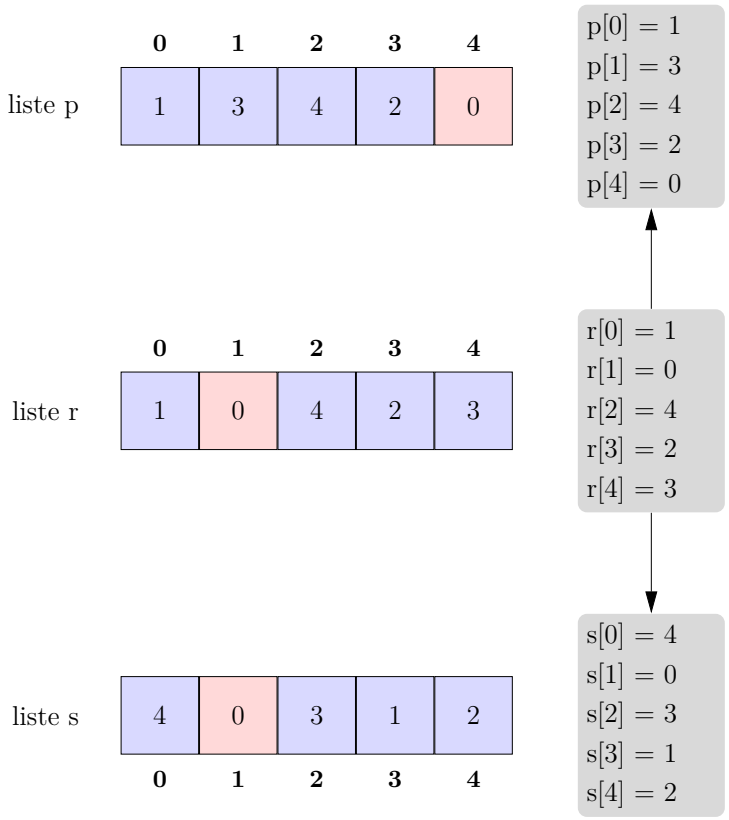


FIGURE 2.7 – Vecteurs tournée, successeur et prédecesseur

Le programme linéaire obtenu est présenté en ANNEXE 2.

Les résultats obtenus ont été concluant et permettent, pour tous les cas testés, de trouver la solution identique en un nombre de noeuds inférieur. Les résultats sont consultables à l'ANNEXE 3.

La suite de l'implémentation va consister à guider le solveur dans ses choix de branchement via des stratégies explicitement indiquées, en se contentant plus de celle par défaut.

2.5 Etude des Stratégies de Branchement

2.5.1 Introduction aux stratégies Choco

Comme détaillé au chapitre 1, le solveur, pour espérer converger vers une solution, doit choisir une première variable sur laquelle se brancher pour commencer à réduire les domaines des variables du problème. Seulement, la stratégie de recherche par défaut implémentée dans le solveur Choco est de choisir la variable du problème ayant le domaine le plus petit. Ainsi, lorsqu'on observe l'arbre de décision, on peut souvent trouver illégitime ces choix, d'autant plus que les enchaînements de branchements ne reflètent aucune logique que l'Homme pourrait avoir.

Ainsi, Choco propose des méthodes permettant de guider le solveur dans les branchements qu'il a à faire. On indique dans le code qu'on souhaite effectuer une recherche personnalisée (surcharge de la stratégie par défaut) grâce à la commande *nomSolveur*.`setSearch`() puis on précise en son sein chaque stratégie à adopter dans autant d'environnements `Search.intVarSearch`(*strat*) qu'on juge nécessaire. Il est impossible d'accumuler sur une seule et même variable plusieurs stratégies mais il est tout à fait possible d'avoir un chaîne de stratégies, chacune portant sur une variable différente. Une stratégie, et en particulier une en Choco, requiert trois informations :

— un sélecteur de variable ;

— un sélecteur de valeur : on prend très souvent la borne inférieure ;

— la variable de branchement.

Parmi les sélecteurs de variables Choco existants, les plus utilisés sont :

- *new* `FirstFail`(*nomModèle*) : sélectionne la variable ayant le domaine le plus petit, et la plus à gauche (les variables instanciées sont ignorées) ;
- *new* `MaxRegret`(*nomModèle*) : sélectionne la variable ayant le plus grand différentiel entre les 2 plus petites valeurs de son domaine (les variables instanciées sont ignorées) ;

N.B. : il est également possible de définir sa propre stratégie au sein d'une classe spécifique. Les stratégies déjà implémentées par Choco ont été suffisantes pour obtenir des résultats intéressants à l'échelle du projet.

Décrivons dès maintenant les stratégies qui ont été mises en place.

2.5.2 Enchaînement de stratégies choisies

Avant de choisir un tel enchaînement, il est intéressant de tester des stratégies simples se ciblant sur une variable afin de voir les éventuelles évolutions. Les premières variables (vecteurs) auxquelles on pense sont *dureeSuc*, r et *dateDebut*. Au vu de comment $dateDebutestdéfini$, il n'est pas très utile de brancher sur lui. En revanche, brancher sur les temps de voyage semble pertinent au vu de la fonction objectif. Vous trouverez les résultats obtenus dans le tableau de l'ANNEXE 3.

On a donc testé plusieurs stratégies :

— brancher uniquement sur la variable *dureeSuc* en utilisant un `MaxRegret` ;

— brancher uniquement sur la variable r en utilisant un `FirstFail` ;

— brancher sur *dureeSuc* puis sur r.

Ainsi, on constate que, sur les instances testées, les stratégies ont apporté une amélioration (ie un ensemble de noeuds parcourus plus petit). La stratégie portant sur le branchement unitaire de *dureeSuc* fonctionne mieux, quelque soit la taille de l'instance.

Suite à la mise en place de ces stratégies, des possibilités d'améliorations existent. Il est possible de guider davantage le solveur en anticipant certaines valeurs.

2.6 Discrimination des choix possibles

L'idée a été d'élaguer les différents chemins dans lequel le solveur pouvait s'engouffrer en lui indiquant par avance certaines informations sur les sommets qu'il pourrait emprunter.

A la fin d'un soin effectué (ie à la date d_{end}), l'infirmière a souvent l'embarras du choix pour choisir sa prochaine destination. Seulement, le nombre de noeuds que le solveur parcourt peut être grandement diminué si des informations lui sont données par avance. Les réflexions se sont portées sur deux discriminations possibles.

2.6.1 Discrimination des distances de voyage

Introduisons cette idée sur un exemple concret présenté en FIGURE 2.8.

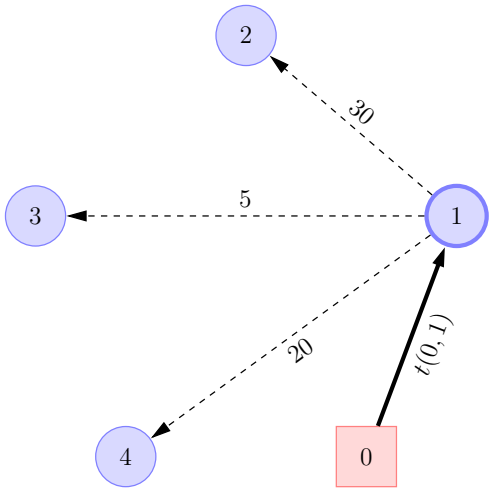


FIGURE 2.8 – Illustration de distances de voyages hétérogènes

On constate rapidement ici qu'il serait malheureux d'emprunter le chemin pour aller vers le patient 4 ou celui vers le patient 2 car ils sont situés à un temps de trajet important du patient 1. Ainsi, emprunter le chemin de temps 5 vers le patient 3 est le plus raisonnable.

Afin d'indiquer au solveur notre préférence, on décide d'implémenter une variable de décision **dureeSucDiscrim** dont le seul but sera de discriminer les sommets sur le critère du temps qui les éloignent du sommet courant. L'une des contraintes qu'elle vérifie est basée sur la formule suivante :

$$\forall i \in V,\ dureeSucDiscrim_i = 100 \times dureeSuc_i + s_i$$

L'autre manière de départager certains sommets a été de se baser sur la borne supérieure des fenêtres de temps.

2.6.2 Discrimination de la borne supérieure des fenêtres de temps

Là aussi, le plus simple est de partir d'une illustration (FIGURE 2.9).

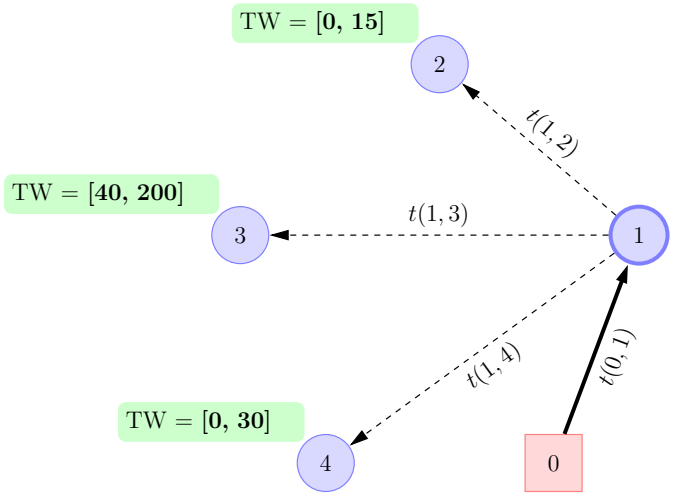


FIGURE 2.9 – Illustration des bornes supérieures de fenêtres de temps hétérogènes

En se rappelant que l'on souhaite les dates au plus tôt (de telle manière à minimiser la durée totale du trajet), il semble naturel de sélectionner comme successeur du sommet 1 celui ayant la borne supérieure de temps la plus faible, car il serait alors plus probable de partir à une date inférieure, au vu des autres sommets suivants 1.

De manière analogue, on va poser une variable **TWDiscrim** afin de séparer les sommets où l'on peut espérer partir le plus rapidement :

$$\forall i \in V,\ TWDiscrim_i = 100 \times TWSup_i + s_i$$

Des tests ont là-aussi été effectués :

— brancher sur la variable *dureeSucDiscrim* en utilisant un `MaxRegret` puis sur la variable r en utilisant un `FirstFail` ;

— brancher sur la variable $TWSupDiscrim$ en utilisant un `MaxRegret` puis sur la variable r en utilisant un `FirstFail` ;

Ces stratégies composites n'ont malheureusement pas porté leurs fruits car ils augmentent de manière importante le nombre de noeuds parcourus, en particulier pour la dernière stratégie proposée.

Suite à ces nombreuses astuces pour guider le solveur dans la résolution du problème PPC, nous allons comparer les résultats initiaux avec les nouveaux obtenus.

2.7 Précisions concernant certains résultats obtenus et conclusions

On rappelle que le sommet 0 est associé à l'hôpital. Ainsi, le temps d'arrivée et de départ de l'infirmière au sommet 0 sont tous deux nuls, en supposant qu'aucune activité intermédiaire soit requise entre ces deux dates. Beaucoup de tests ont été effectués sur le graphe suivant (FIGURE 2.10) :

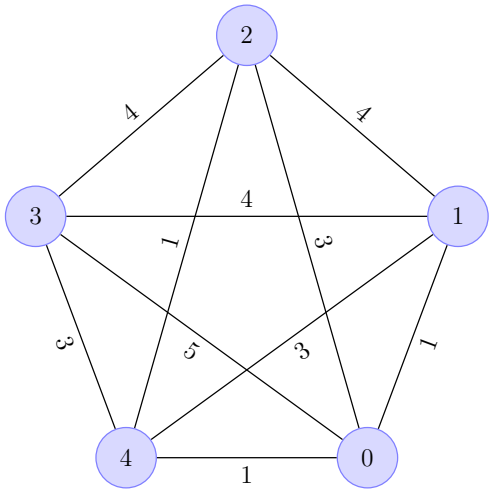


FIGURE 2.10 – Graphe principal de test

2.7.1 Eviter la construction de sous-tours de taille 1

Cette contrainte, traduit en Choco (`TSPModel.arithm(s[i], "!=", i).post();`) a permis une diminution considérable du nombre de noeuds observés, malgré sa redondance. Voici les résultat obtenus sur le graphe test avec application du TSP (fenêtres de temps infinies et soins ponctuels) :

NEW Model av contr[TSP en PPC], 2 Solutions, MINIMIZE d = 11, Resolution time 0,212s, 46 Nodes (216,6 n/s), 89 Backtracks, 43 Fails, 0 Restarts -> O 4 2 3 1

NEW Model ap contr[TSP en PPC], 1 Solutions, MINIMIZE d = 11, Resolution time 0,091s, 9 Nodes (98,6 n/s), 17 Backtracks, 8 Fails, 0 Restarts Cmax = 11 -> 0 4 2 3 1

On obtient des résultats similaires avec le TSP (celui obtenu en fin de partie 2.3). Ainsi, la propagation de contraintes est facilitée et réduit de manière conséquente l'effort de résolution du solveur.

Tous les résultats ci-dessous dépendent d'algorithmes avec comme contrainte supplémentaire celle décrite dans cette partie.

2.7.2 Cas fenêtres de temps infinies et soins ponctuels

Les fenêtres de temps sont fixées à $[0, sup]$ et la durée des soins est égale à 0 ici. Contrairement à ce que l'on pourrait penser, appliquer le TSPTW (celui obtenu en fin de partie 2.3) sur ce cas ne revient pas au même que d'appliquer le TSP traditionnel. En effet, les contraintes portant sur les dates entraînent des branchements malencontreux (sur les variables de dates, pourtant inutiles ici) de la part du solveur Choco, ralentissant la convergence vers la solution.

Model[Model-0], 2 Solutions, MINIMIZE d = 11, Resolution time 0,082s, 7 Nodes (85,8 n/s), 11 Backtracks, 4 Fails, 0 Restarts -> 0 4 2 3 1

NEW Model[TSP en PPC], 1 Solutions, MINIMIZE d = 11, Resolution time 0,146s, 9 Nodes (61,5 n/s), 17 Backtracks, 8 Fails, 0 Restarts -> 0 4 2 3 1

En revanche, ces algorithmes donnent le même résultat (le premier) si on commente les contraintes liées aux dates (sans commenter les grandes fenêtres de temps).

2.7.3 Cas fenêtres de temps restreintes et soins non ponctuels

Tous les résultats ayant été obtenus sont récapitulés dans l'Annexe 3 de ce rapport. Il contient une comparaison entre le TSPTW de base (celui en fin de partie 2.3), et les TSPTW ayant été complétés par les suppléments détaillés en 2.4, 2.5, et 2.6.

Il va de soi que ce problème est bien plus simple que celui à traiter. La première donnée supplémentaire que l'on pourrait ajouter à notre modèle serait de disposer de plusieurs infirmières, effectuant chacune leur tournée. Le problème ainsi modélisé est alors celui du VRP.

Chapitre 3

Adaptation du TSP au VRP

3.1 Présentation du VRP et modélisation linéaire associée

3.1.1 Différences avec le TSP

Le VRP (pour *Vehicle Routing Problem* en anglais) connu en Français sous l'appellation générique Problème de Tournées de Véhicules, est une classe de problèmes en Recherche Opéationnelle proches du TSP, à ceci près que le nombre d'infirmières disponibles pour rendre les services médicaux sur place est supérieur à 1. Ainsi, une gestion logistique supplémentaire s'impose, notamment pour que leurs heures de travail ne se chevauchent pas pour un même client : on commence tout doucement à atteindre les réelles problématiques d'un problème de type *Workforce Scheduling*.

Dans ce modèle, les infirmières disposent de leur propre véhicule de fonction pour se déplacer sur les lieux concernés, et il existe toujours un lieu fixe associé à l'hôpital. Ainsi, une infirmière part de l'hôpital, effectue sa tournée (ie dessert un ensemble de patients qu'elle sera la seule à rencontrer) et puis revient à l'hôpital. Dans un premier temps, on considère les hypothèses suivantes, à savoir que :

— le (ou les) soin(s) que nécessite(nt) chaque client sont pour l'instant unitaires et ponctuels (ie sans durée) dans la journée ;
— il existe une route directe entre chaque client, et peut être empruntée dans les deux sens ;
— les véhicules de fonction sont identiques (ils roulent donc à la même vitesse).

N.B. : dans ce contexte précis, nous allons pour l'instant ne pas considérer de capacité

particulière, que ce soit pour le véhicule ou le service rendu. On pourrait en effet associer un transport de matériel à un soin, matériel qui aurait une capacité (un poids ou un volume), mais on cherche dans un premier temps à résoudre le problème type VRP le plus simple possible.

Maintenant que les hypothèses du modèle ont été posées, et sachant que ce dernier repose sur les mêmes bases mathématiques que celui du TSP, exposons sa formulation linéaire.

3.1.2 Programme Linéaire associé et limites

La liste de notations est légèrement plus importante qu'auparavant, due à l'augmentation de la difficulté du problème, et le problème qui suit également :

Données 5

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- Np l'ensemble des patients et $np = |Np|$ le nombre de patients total à soigner ;
- Nv l'ensemble des véhicules / infirmières et $nv = |Nv|$ le nombre de véhicules / infirmières disponibles.

Variables de décision 5

- $x_{ij}^k = 1$ si l'infirmière k se déplace de i vers j pour traiter le patient j ;
- **d** la durée totale qui a été nécessaire pour que toutes les infirmières bouclent leur tournée respective.

$$\min \quad d = \sum_{i=0}^{np-1} \sum_{j=0}^{np-1} \sum_{k=0}^{nv-1} x_{ij}^k . T_{ij}$$

$$s.c. \quad \sum_{j=1}^{np-1} x_{1j}^k \leq 1 \qquad \forall k \in [|0, nv-1|] \tag{3.1}$$

$$\sum_{j=0}^{np-1} \sum_{k=0}^{nv-1} x_{ij}^k = 1 \qquad \forall i \in [|1, np-1|] \tag{3.2}$$

$$\sum_{i=0}^{np-1} \sum_{k=0}^{nv-1} x_{ij}^k = 1 \qquad \forall j \in [|1, np-1|] \tag{3.3}$$

$$\sum_{j=0}^{np-1} x_{ji}^k = \sum_{j=0}^{np-1} x_{ij}^k \qquad \forall j \in [|1, np-1|], \forall k \in [|0, nv-1|] \tag{3.4}$$

$$x_{ji}^k \neq 1 \qquad \forall i \in [|0, np-1|], \forall j \in [|0, nv-1|] \tag{3.5}$$

Voici la signification des contraintes :

- (3.1) décrit le fait qu'un véhicule peut quitter l'hôpital au plus 1 fois,
- (3.2) impose l'unicité de l'arc entrant pour chaque noeud du graphe,
- (3.3) impose l'unicité de l'arc sortant pour chaque noeud du graphe,
- (3.4) est une facilité d'écriture pour la fonction objectif,
- (3.5) casse les sous-tours de dimension 1.

▷ La minimisation porte sur la somme des durées de trajet qui ont été nécessaires pour effectuer la tournée.

Dans ce modèle, chaque infirmière effectue sa propre tournée. Par analogie avec le TSP, le schéma à considérer se trouve en FIGURE 3.1.

Dans ce modèle, l'intérêt est d'avoir un personnel soignant dont la taille est supérieure à 1. Le type de modélisation à garder en tête est donc plutôt un graphe type celui présenté en FIGURE 3.2.

Une fois de plus, le modèle type programme par contraintes implique certaines modifications d'ordre structurelles.

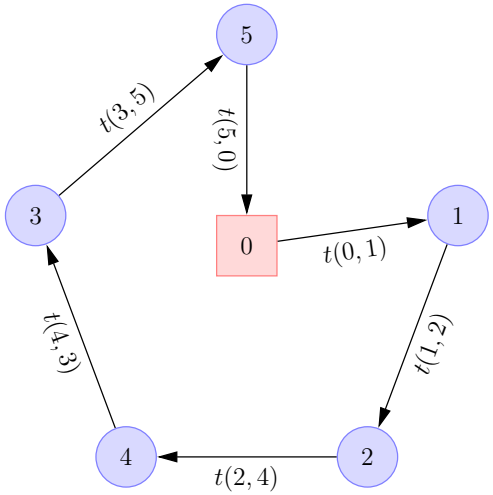


FIGURE 3.1 – Modélisation du VRP avec une seule infirmière en service

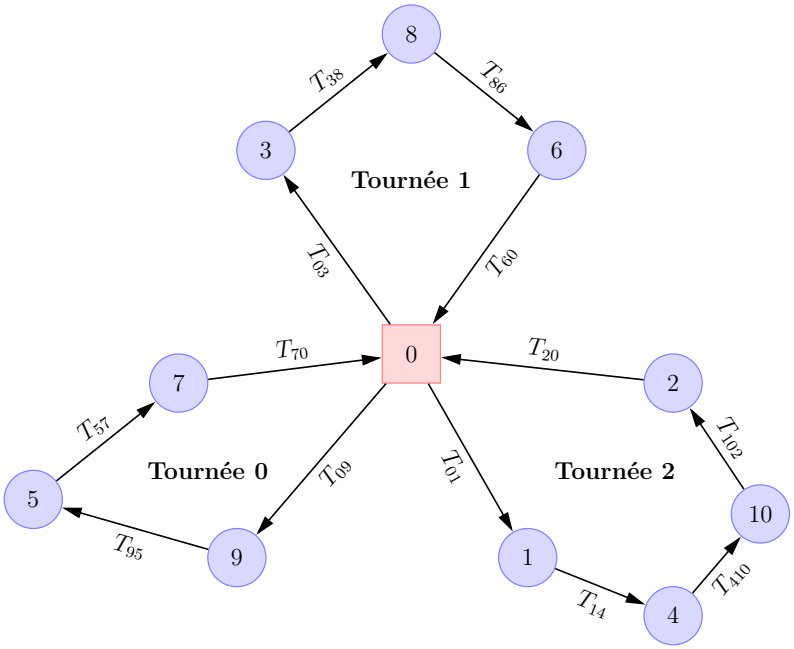


FIGURE 3.2 – Modélisation du VRP avec plusieurs infirmières en service

3.2 Modélisation type Programmation par contraintes

Dans ce type de programmation, on décide d'associer un hôpital d'entrée et un hôpital de sortie pour chaque tournée effectuée par une infirmière du personnel. En réalité, les hôpitaux d'entrée et de sortie désignent la même entité. Voici la représentation sur laquelle nous allons nous baser pour le programme par contraintes (FIGURE 3.3).

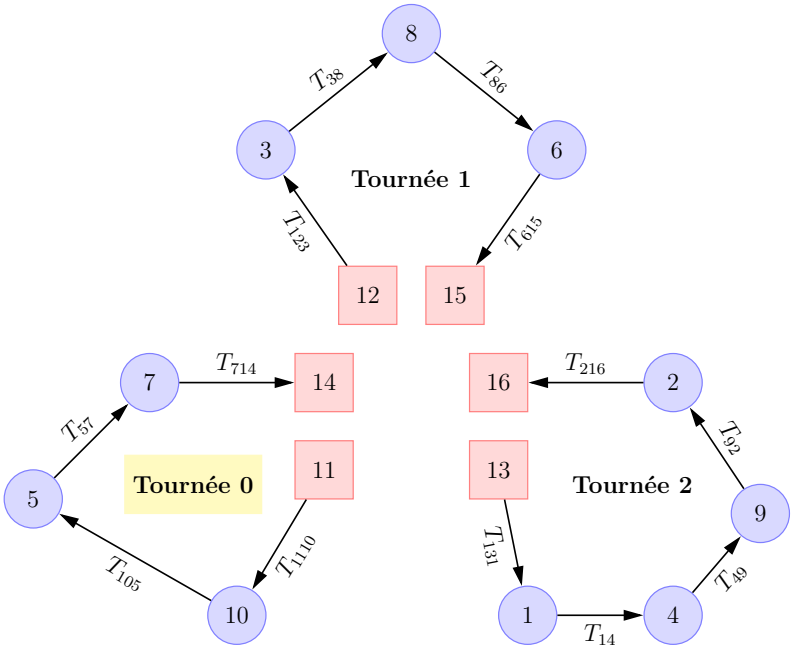


FIGURE 3.3 – Modélisation du VRP pour Choco avec plusieurs infirmières en service

Voici une représentation graphique (FIGURE 3.4) des vecteurs que l'on va crée et des liens entre eux :

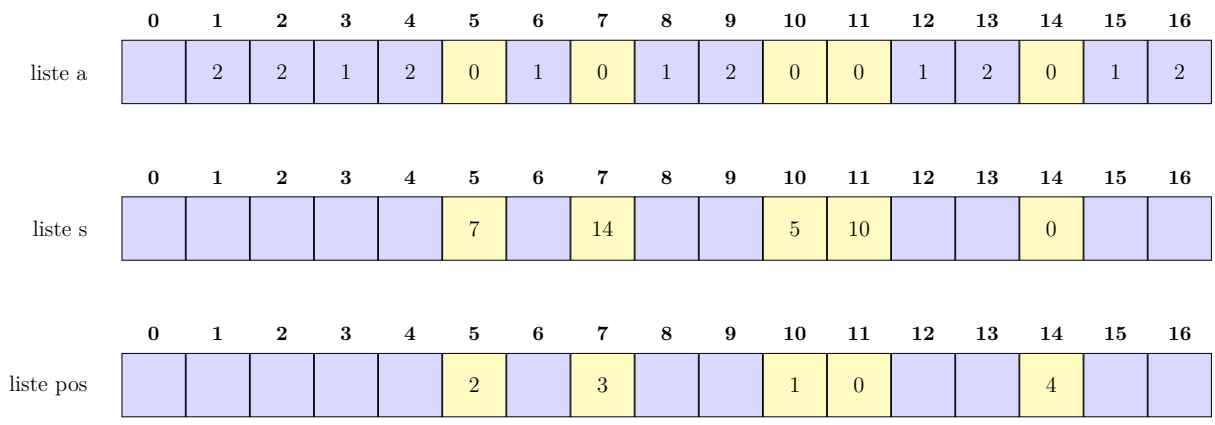


FIGURE 3.4 – Introduction des vecteurs a, s et pos

Toute la différence de cette formulation réside dans la nécessité d'affecter un ensemble

de patients à une infirmière (qui devra donc les soigner). Ainsi, l'introduction de plusieurs vecteurs supplémentaires est indispensable :

— La liste a indique à quel véhicule a été affecté le patient. Par exemple, le patient 7 a été affecté au véhicule 0 sur la figure précédente.

— La liste s indique le patient suivant qui va être traité. Sur la FIGURE 3.4, le patient 14 sera soigné juste après le patient 7.

— Enfin, la liste pos indique la position du client dans la tournée de l'infirmière. Sur la figure ci-dessus, le patient 7 sera le 3^{eme} patient dans la tournée de l'infirmière (en commençant le comptage à 0).

De ces 3 vecteurs, on peut ainsi construire plusieurs partitions de clients en fonction des tournées formées. La partition correspondant à la tournée 0 sera : $b^0 = \{11, 10, 5, 7, 14\}$

L'ensemble des notations qui seront utilisées par la suite sont indiquées ci-dessous :

Données 6

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- V l'ensemble des sommets du graphe, où $|V| = np + |H^d| + |H^f|$;
- Np l'ensemble des patients et $np = |Np|$ le nombre de patients total à soigner ;
- Nv l'ensemble des véhicules / infirmières et $nv = |Nv|$ le nombre de véhicules / infirmières disponibles ;
- H^d l'ensemble d'hôpitaux (dupliqués) de départ de course ;
- H^f l'ensemble d'hôpitaux (dupliqués) de fin de course ;
- **sup** la borne supérieure de temps.

Variables de décision 6

- $\mathbf{tpsSuc_i}$ le temps de voyage entre le patient i et son successeur dans la tournée ;
- $pos_i = k$ si le patient i est visité en position k dans la tournée ;
- s_i le successeur du patient i dans la tournée ;
- $a_i = v$ l'affectation du patient i à une infirmière / un véhicule v ;
- **d** la durée totale qui a été nécessaire pour que toutes les infirmières bouclent leur tournée respective ;
- b^v l'ensemble de clients affectés à la tournée de l'infirmière v.

$$\min \quad d \;=\; \sum_{i=0}^{V-1} T_{i,s_i}$$

s.c.

domaines de définition

$$s_i \in [0, |V| - 1] \qquad \forall i \in Np \;\cup\; H^d \tag{3.6}$$

$$a_i \in [0, nv - 1] \qquad \forall i \in Np \tag{3.7}$$

$$pos_i \in [0, np] \qquad \forall i \in Nv \;\cup\; H^f \tag{3.8}$$

autres contraintes

$$s_i = 0 \qquad \forall i \in H^f \tag{3.9}$$

$$a_i = i \qquad \forall i, j \in H^d \;\cup\; H^f \tag{3.10}$$

$$pos_i = 0 \qquad \forall i, j \in H^d \tag{3.11}$$

$$s_i \neq s_j \qquad \forall i, j \in \{Np \;\cup\; H^d \;\cup\; H^f\}^2 \tag{3.12}$$

$$a_i = a_{s_i} \qquad \forall i \in Np \;\cup\; H^d \tag{3.13}$$

$$s_i \neq i \qquad \forall i \in Np \;\cup\; H^d \tag{3.14}$$

$$pos_{s_i} = p_i + 1 \qquad \forall i \in Np \;\cup\; H^d \tag{3.15}$$

$$b^v = \{u \in N \mid a_u = v\} \qquad \forall v \in Nv \tag{3.16}$$

Voici la signification des contraintes :

- (3.6) à (3.8) représente les domaines où les variables correspondantes vont prendre leurs valeurs,
- (3.9) indique que les hôpitaux finaux ne peuvent avoir de successeur,
- (3.10) affecte un hôpital à un véhicule,
- (3.11) initialise la position de tous les hôpitaux initiaux à 0,
- (3.12) impose l'unicité de tous les successeurs,
- (3.13) affecte le patient i au même véhicule que son successeur a_i,
- (3.14) casse les sous-tours de dimension 1,
- (3.15) évite les sous-tours,
- (3.16) définit une partition de clients en sous-ensembles affectés à une infirmière.

▷ La minimisation porte toujours sur la somme des durées de trajet qui ont été nécessaires pour effectuer la tournée.

Discussions et perspectives

Dans ce type de problème ouvert, les évolutions possibles sont très souvent infinies pour les problèmes de type *Workforce Scheduling* car, étant NP-Complets, il est impossible de trouver un algorithme qui résout le problème en temps polynomial pour n'importe quelle instance d'entrée.

L'une des évolutions essentielles qui n'a pas été prise en compte est l'association pour chaque infirmière d'une palette de compétences. Selon leur formation, elles ne disposent pas des mêmes savoir-faire, ce qui introduit une complexité de modélisation considérable dans la planification selon le soin que nécessite le patient.

L'une des autres évolutions possibles est de privilégier la même infirmière pour un patient nécessitant plusieurs soins dans la journée, soins qui peuvent être fournis par l'infirmière dotée des compétences adéquates.

Enfin, l'implémentation de programmes analogues en Gusek ou en CPLEX sur Visual Studio aurait permis de se faire une idée de la borne inférieure du résultat à obtenir (ie le minimum de ce qu'on est en droit d'attendre de notre nouveau programme).

Conclusion

L'objectif du projet était, dans un premier temps, de se familiariser avec les modélisations classiques des problèmes de tournées et, dans un second temps, de proposer une méthode simple pour un problème de type Workforce Scheduling sur une journée. Il s'est avéré plus raisonnable de concentrer les efforts sur un modèle simple tel que le TSP afin de préparer au mieux le travail sur le VRP, qui est le problème le plus simple de type Workforce Scheduling.

La première difficulté que j'ai dû surmonter a été de comprendre en détail le fonctionnement de la Programmation Par Contraintes à travers une librairie dédiée. Il a fallu maîtriser en profondeur le fonctionnement du solveur afin de le guider dans ses choix de branchements. De plus, ce "paradigme" de programmation est assez perturbant lorsqu'on a été habitué de tout temps à la Programmation Linéaire. Enfin, la gestion de la banque de tests pour ce type de problème a été difficile à stabiliser : il faut donc un sens de l'organisation efficace afin de savoir quelle version de code et sur quelles données se baser pour effectuer les comparaisons de performances.

J'ai pour ma part augmenté considérablement ma maîtrise en programmation linéaire et par contraintes, les travaux pratiques sur ces notions là manquant considérablement dans la filière. Je me suis également rendu compte du potentiel qu'a toujours ce type de réflexion face aux nouvelles visions de l'Intelligence Artificielle basées sur les algorithmes d'Apprentissage Profond, nécessitant des instances de grande taille préparées par des méthodes d'Apprentissage Statistique. Enfin, à la suite de mon stage Recherche & Développement de deuxième année et des projets de filière effectués à l'ISIMA, j'ai pu me faire une idée très claire du Monde de la Recherche et de son fonctionnement.

Bibliographie

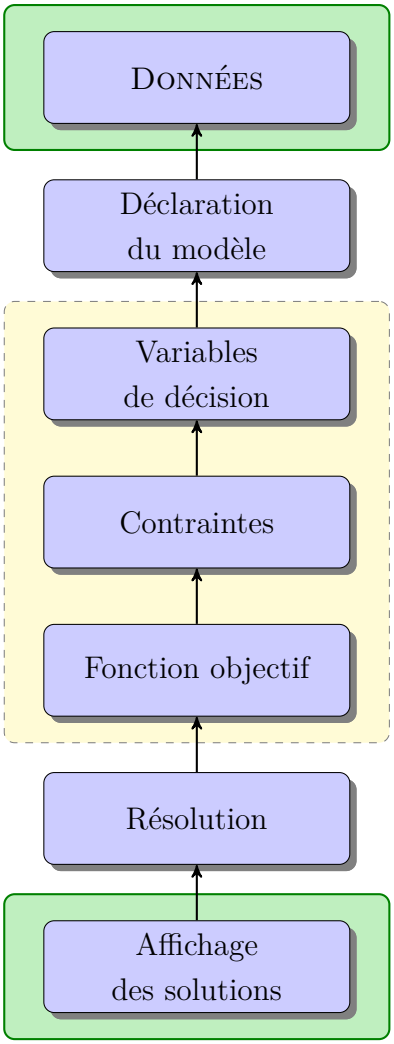
[1] Philippe Lacomme Marina Vinot Eric Bourreau, Matthieu Gondran. De la programmation linéaire a la programmation par contraintes (Chapitres 1, 2 et 5). Ellipses, 2018.

[2] Charles Prud'homme. Documentation choco en ligne. `http://www.choco-solver.org/apidocs/`.

[3] Charles Prud'homme. Documentation choco pdf. `https://media.readthedocs.org/pdf/choco-tuto/latest/choco-tuto.pdf`, Octobre 2018.

[4] Hélène Toussaint. Tutoriel compiler un code c ou c++ utilisant cplex sous visual studio. `http://fc.isima.fr/~toussain/doc/CplexVS15.pdf`, Juillet 2018.

Annexe 1 - Programme Par Contraintes avec prédécesseurs



Annexe 2 - Programme Par Contraintes avec prédécesseurs

Données 7

- **T** la matrice des temps de parcours, où l'élément T_{ij} représente le temps requis pour se déplacer du patient i vers le patient j ;
- **n** le nombre de patients (ie $n = |V|$) ;
- **sup** la borne supérieure de temps ;
- **TWInf** le vecteur contenant les bornes inférieures des fenêtres de temps ;
- **TWSup** le vecteur contenant les bornes supérieures des fenêtres de temps ;
- **dureeSoin** le vecteur contenant les durées de soins que chaque patient nécessite.

Variables de décision 7

- $\mathbf{r_i} = k$ si le patient k est visité en position i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{s_i} = k$ si le patient k est visité à la suite du patient i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{p_i} = k$ si le patient k est visité juste avant le patient i dans la tournée qu'effectue l'infirmière ;
- $\mathbf{tpsSuc_i}$ le temps de voyage entre le patient i et son successeur dans la tournée ;
- **d** le durée totale qui a été nécessaire pour que l'infirmière boucle sa tournée ;
- **dateArr** la date d'arrivée de l'infirmière devant une habitation ;
- **dateDeb** la date de début du soin fourni au patient ;
- **dateFin** la date de fin du soin fourni au patient ;
- **dateDep** la date de départ de l'infirmière de l'habitation ;
- **dateArrSuc** la date d'arrivée de l'infirmière devant l'habitation suivante celle courante ;
- **dateDebSuc** la date de début du soin fourni au patient suivant celui courant ;
- **TWInfSucVal** la borne inférieure de la fenêtre de temps du patient suivant celui courant ;
- **dateFinPrec** la date de fin du soin fourni au patient précédant celui courante ;
- **tempsPrec** le temps de voyage qu'il a fallu pour arriver à ce patient depuis le précédent, ie $tpsSuc_{i-1}$;

- **TWInfVal** la borne inférieure de la fenêtre de temps courante.

$$\min \quad d \;=\; \sum_{i=0}^{n-1} tpsSuc_i$$

s.c.

domaines de définition

$$r_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \qquad (17)$$

$$s_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \qquad (18)$$

$$p_i \in [0, n-1] \qquad \forall i \in [|0, n-1|] \qquad (19)$$

$$tpsSuc_i \in [0, sup] \qquad \forall i \in [|0, n-1|] \qquad (20)$$

$$d \in [0, (n-1) \times sup] \qquad \forall i \in [|0, n-1|] \qquad (21)$$

$$dateArr_i \in [0, TWSup_i] \qquad \forall i \in [|0, n-1|] \qquad (22)$$

$$dateDeb_i \in [TWInf_i, TWSup_i] \qquad \forall i \in [|0, n-1|] \qquad (23)$$

$$dateFin_i \in [TWInf_i, TWSup_i] \qquad \forall i \in [|0, n-1|] \qquad (24)$$

autres contraintes

$$r_i \neq r_j \qquad \forall i, j \in [|0, n-1|] \qquad (25)$$

$$s_i \neq s_j \qquad \forall i, j \in [|0, n-1|] \qquad (26)$$

$$p_i \neq p_j \qquad \forall i, j \in [|0, n-1|] \qquad (27)$$

$$r_{i+1} \neq s_{r_i} \qquad \forall i \in [|0, n-2|] \qquad (28)$$

$$tpsSuc_i \neq T_{is_i} \qquad \forall i \in [|0, n-1|] \qquad (29)$$

$$s_i \neq i \qquad \forall i \in [|0, n-1|] \qquad (30)$$

$$dateArr_{s_i} = dateFin_i + tpsSuc_i \qquad \forall i \in [|0, n-1|] - r_{n-1} \qquad (31)$$

$$dateArr_{s_i} = 0 \qquad si\ i = r_{n-1} \qquad (32)$$

$$dateDeb_{s_i} = \max(dateArr_{s_i}, TWInf_{s_i}) \qquad \forall i \in [|0, n-1|] \qquad (33)$$

$$dateArr_i = dateFin_{p_i} + tpsSuc_{p_i} \qquad \forall i \in [|0, n-1|] - r_{n-1} \qquad (34)$$

$$dateArr_i = 0 \qquad si\ i = r_{n-1} \qquad (35)$$

$$dateDeb_i = \max(dateArr_i, TWInf_i) \qquad \forall i \in [|0, n-1|] \qquad (36)$$

$$dateFin_i = dateDeb_i + dureeSoin_i \qquad \forall i \in [|0, n-1|] \qquad (37)$$

Annexe 3 - Tableau comparatif des résultats

On compare le TSPTW obtenu en fin de question 3 avec d'autres ayant été légèrement modifiés, afin de voir clairement l'évolution pour un groupe de contraintes identifié.

Cas	Avec/sans prec	Nodes	Backtracks	Fails
1	sans	32	59	27
	avec	8	13	5
2	sans	21	39	18
	avec	12	19	7
3	sans	7570	15135	7565
	avec	60	111	51

Cas	Strat testée	Nodes	Backtracks	Fails
1	aucune	8	13	15
	avec tpsSuc	5	9	4
	avec r	7	11	14
2	aucune	12	19	7
	avec tpsSuc	4	7	3
	avec r	5	7	2
3	aucune	60	111	51
	avec tpsSuc	18	35	17
	avec r	122	217	95