Automatisation de construction de planning

Projet Androide

Kevin Nartz
Ariane Ravier
Anthéa Richaume
Bassem Yagoub

Encadrement : Pierre Fouilhoux & Cédric Herpson
Avec la participation de Diane Redel

9 juin 2020

Table des matières

Introduction **2**

1 État de l'art **2**
1.1 Résolution exacte . . . 3
1.2 Résolution approchée . . . 4
1.3 Construction manuelle et aide à la décision . . . 6
1.4 Précision d'adaptation aux préférences . . . 6

2 Approche du problème et formalisation **7**
2.1 Modèle . . . 7
2.1.1 Service . . . 7
2.1.2 Agent . . . 8
2.1.3 Poste . . . 8
2.2 Contraintes . . . 8
2.2.1 Volumes horaires (C1) . . . 8
2.2.2 Enchaînements de postes interdits (C2 - P2) . . . 9
2.2.3 Accompagnement des agents débutants (C3 - P3) . . . 9
2.2.4 Affectations de postes interdites (C4) . . . 10
2.2.5 Quotas journaliers (C5) . . . 10
2.2.6 Séquences conditionnelles (C6) . . . 10
2.2.7 Nombre de séquences minimum (ou maximum) (C7) . . . 11
2.3 Données . . . 11

3 Approche heuristique **12**
3.1 Méthode gloutonne . . . 12
3.2 Voisinage . . . 14
3.3 Evaluation de la solution . . . 15
3.4 Méthode itérative . . . 15

4 Programmation linéaire **17**
4.1 Outils utilisés . . . 17
4.2 Variables et contraintes . . . 17
4.3 Mise en oeuvre . . . 18

5 Expérimentations **20**
5.1 Instances . . . 20
5.1.1 Génération d'instances . . . 20
5.1.2 Classes d'instances . . . 21
5.2 Critères d'évaluation . . . 21
5.3 Comparaison . . . 22
5.4 Nouveaux résultats . . . 24

6 Conclusion **24**

7 Perspectives **24**

Références **28**

Résumé

Le problème d'automatisation de construction de planning est un problème NP-difficile consistant, à partir de contraintes et d'agents donnés, à déterminer un planning satisfaisant un maximum de contraintes.

Nous proposons une solution heuristique déterminant dans un premier temps une solution par approche gloutonne. Cette solution est ensuite optimisée grâce à une exploration de voisinage dans une méthode itérative qui évaluera les solutions ainsi obtenues, afin de retourner la meilleure.

Dans une deuxième partie, nous proposons une résolution par programmation linéaire, implémentant quelques-unes des contraintes.

Nous présentons enfin différentes classes d'instances générées sur la base de l'aléatoire, permettant de tester les performances des deux approches.

Introduction

Ce projet nous a avant tout été proposé en tant que recherche de solution pour un problème de terrain.

La maternité du CHIC emploie environ 70 sages-femmes, et la construction du planning mensuel des services qui le constituent se fait à ce jour manuellement. Cela représente un temps de travail considérable, pouvant facilement se compter en jours, et implique nécessairement que certains employés prennent sur leurs heures de service pour la réalisation du planning.

L'objectif premier de cette automatisation est d'assister les personnes responsables du planning dans leur travail. Il s'agit de réduire autant que possible la durée de construction du planning, tout en s'assurant de la recevabilité de la solution.

Ce rapport procèdera dans un premier temps à un état de l'art sur le problème de la construction automatique de planning dans les hôpitaux. Nous présenterons ensuite le modèle sur lequel nous nous sommes basés, puis détaillerons l'approche heuristique et l'approche par programmation linéaire développées au cours du projet. Enfin, nous ferons part des résultats obtenus lors de nos expérimentations, et parlerons de nos perspectives sur le projet.

1 État de l'art

Le *Nurse Rostering Problem* (NRP) est un problème combinatoire dont l'enjeu est de trouver une affectation optimale des agents d'un service hospitalier sur une période (*horizon*) donnée. Cette affectation doit respecter un ensemble de contraintes, certaines dites "dures" (*hard constraints*), portant sur des conditions non-négociables car légales ou contractuelles ; les autres dites "souples" ou "préférentielles" (*soft constraints*), portant sur des critères pouvant améliorer les conditions de travail des agents. Ces dernières peuvent constituer un ajout de restrictions quant aux contraintes dures, ou encore une expression des préférences personnelles de chaque agent.

Il existe différentes configurations de services. Certains ne distinguent notamment que deux types de gardes (jour et nuit), et suivent donc une logique *2-shift* ; d'autres séparent la garde de jour en garde de matin, et garde de l'après-midi, passant donc à une logique *3-shift*. Dans certaines approches ([ABL07]), cette distinction est floutée le temps de la résolution par souci de rapidité de résolution.

On distingue également des différences structurelles entre les plannings, qui peuvent impacter la méthode de construction : les plannings cycliques, d'une part, consistent à assigner une séquence de postes (*pattern*) fixe aux agents, et de la répéter sur les semaines et mois qui suivent ; les plannings acycliques, d'autre part, tiennent compte de caractéristiques plus précises qui empêchent une répétition systématique d'un *pattern*, et nécessitent de procéder jour par jour.

Les NRPs sont le sujet, ou simplement l'application, d'une littérature scientifique fournie. Le nombre important de contraintes qui le régissent en font un problème que [TK82] décrit comme plus complexe encore que le problème du Voyageur de Commerce (TSP). De plus, il apparaît qu'il est très difficile de définir un planning optimal, voire très difficile de trouver un planning respectant l'intégralité des contraintes quand celles-ci sont en très grand nombre, et ce même en utilisant l'avis humain du personnel hospitalier. L'objectif n'apparaît donc pas comme la recherche d'une solution optimale, mais comme une amélioration/optimisation de solutions réalisables.

Au fil des années, plusieurs axes de recherche ont été empruntés, et plusieurs méthodes se sont imposées dans la littérature. Ces méthodes ont pu être manipulées dans des cadres théoriques, parfois même au cours de compétitions internationales ([HDCSS14],[CDDC$^+$19]), ou encore autour de cas particuliers et problèmes de terrain ([PRS$^+$11], [BGBB20]). A l'image de [BDCBVL04], nous proposerons ici un tour d'horizon de la littérature sur le sujet, que nous espérons représentative.

Nous distinguerons d'une part les méthodes de résolution exacte, qui emploient des outils comme la programmation linéaire pour obtenir des solutions optimales exactes sur des modèles souvent simplifiés ; d'autre part, nous nous intéresserons aux méthodes de résolution approchée, qui cherchent à satisfaire autant que faire se peut l'intégralité des contraintes souples, en plus des contraintes dures. Puis nous nous pencherons sur le rapport que certaines approches maintiennent avec la construction manuelle de planning (*self-scheduling*) et la possibilité d'interactivité au sein de la résolution.

1.1 Résolution exacte

Résolution par programmation linéaire (PL)

Une première façon d'appréhender le problème est à travers la programmation mathématique. L'emploi de la programmation linéaire est notamment très courant, et ce depuis les premiers travaux sur le NRP. Le *branch-and-bound* a souvent été utilisé pour résoudre le problème instancié sur un modèle de PL ([TW76]). Cependant, on lui préfère depuis l'algorithme de *branch-and-price*, consistant à procéder à une génération de colonnes à chaque noeud d'un arbre généré par *branch-and-bound* ([JSV98], [BC10], [LOR19]). Cette hybridation de méthodes permet d'appliquer le traitement de programmation linéaire à des problèmes de plus grande échelle, et pouvant prendre en compte d'autres contraintes que les *hard constraints*. Dans [TW76], notamment, le *branch-and-bound* employé est efficace parce qu'il ne porte que sur un sous-problème d'échelle réduite : celui de l'affectation d'agents à différents services en cas de manque de personnel.

On note que l'approche par programmation linéaire est souvent préférée pour résoudre une simplification du NRP ([BGBB20]) ou en première étape d'une méthode qui utilise ensuite une heuristique pour améliorer la solution obtenue ([VH00]). En effet, dès lors que le problème contient un nombre important de variables et de contraintes, l'espace de recherche devient rapidement trop grand pour qu'une solution puisse être générée par programmation linéaire.

Optimisation multi-objectifs

Afin de s'assurer de la satisfaction d'un maximum de contraintes, plusieurs articles envisagent le NRP comme un problème d'optimisation multi-objectif. Il s'agit d'organiser un ensemble de contraintes par ordre de priorité et de viser à ce que chacun des objectifs soit respecté, ou du moins ne soit pas négligé au profit d'autres contraintes.

[AR81] et [MS84] commencent par traiter le NRP comme un problème de *goal programming*. Le *goal programming* est une généralisation de la programmation linéaire, adaptée à l'approche multi-critère. Cela leur sert, dans un premier temps, à trouver une bonne solution, qu'ils cherchent ensuite à améliorer à l'aide d'autres outils ([BFM96]). Pour [AR81], le problème de *goal programming* est réduit à une simple affectation de jours de travail ou de congé : la véritable affectation des postes se fait ensuite via une heuristique.

Il est assez courant de proposer une méhode hybride entre optimisation multi-critères et recherche heuristique. [BFM96] introduit, après une résolution multi-objectif, une procédure de recherche tabou pour améliorer le résultat. [Jas97] combine même résolution multi-objectif et recuit simulé au sein d'un seul algorithme, appelé *Pareto-Simulated Algorithm* pour générer un échantillon de solutions approximant l'ensemble des solutions non-dominées. [CY93], enfin, combine même le *goal programming* binaire avec un système expert, pour réduire le temps d'exécution du logiciel tout en préservant la qualité de la solution renvoyée.

Approche par satisfaction de contraintes (CSP)

Une formulation du problème comme un CSP (*Constraint Satisfactin Problem*) paraît assez naturelle, puisque le NRP est défini par l'ensemble de ses contraintes.

[LLR03] utilise un WCSP (*weighted CSP*) pour obtenir le meilleur compromis entre les contraintes, avant de passer à une optimisation de la solution par recherche locale. [AS99] se base sur un CSP partiel (PCSP), où des poids sont associés aux contraintes pour exprimer l'importance de leur satisfaction.

[PRS+11] procède uniquement par CSP pour résoudre le problème d'affectation de l'hôpital de Valparaiso. L'outil décrit est efficace et employé par l'hôpital, mais il se contente d'affecter des postes *jour*, *nuit* ou *repos* sans spécification des rôles ni prise en compte des contraintes préférentielles.

1.2 Résolution approchée

Les diverses approches mathématiques ont donc toujours été de grande importance dans les méthodes de résolution du NRP. Cependant, comme nous l'avons précisé dans l'introduction, il est difficile de considérer qu'il existe une solution optimale au problème de *nurse rostering*. Et puisqu'on cherche avant tout à améliorer autant que possible toute solution réalisable, on comprend qu'il soit courant de considérer l'emploi d'heuristiques. Celles-ci permettent d'appliquer un raisonnement et une méthode plus flexibles, donc peut-être plus proches des méthodes de résolution manuelles du NRP.

Certaines méthodes utilisent uniquement des méthodes heuristiques ; d'autres, dont certaines ont été mentionnées plus haut, partent d'une solution mathématique, qui est ensuite améliorée grâce à des outils heuristiques.

Recherche locale

Les méthodes de recherche locale sont le plus souvent combinées à d'autres méthodes, qu'elles soient mathématiques ([LLR03]) ou heuristiques ([EM93], [ABL07]).

[LH10] propose une recherche locale adaptative seule, avec un processus de multi-démarrage. Dès que la recherche locale cesse d'améliorer la solution courante, la meilleure solution est retenue, est réutilisée comme point de départ d'une nouvelle recherche locale. Cela permet d'envisager plusieurs optima locaux.

Recherche de voisinage (VNS)

Cette méthode consiste, à partir d'une solution donnée ou obtenue par pré-traitement, à explorer des solutions voisines pour trouver des solutions améliorantes. Procéder par recherche sur voisinages réduits (VDNS) permet de réduire l'espace de recherche : il s'agit d'explorer les voisinages modifiant une ou plusieurs variables appartenant à un sous-ensemble des n éléments constituant la solution ([Met10],[MBL09], [BCQVB13]).

Hill-climbing

Parmi les méthodes de recherche locale les plus utilisées, on note le *local hill-climbing* ([EM93]), qui vise à trouver des optima locaux, et peut donc s'avérer particulièrement utile dans cette situation, où il n'y a pas nécessairement de meilleur optimum.

Recuit simulé

On a vu plus haut que le recuit simulé était utilisé par [Jas97], combinée à l'approche multi-objectif dans son *Pareto-Simulated Algorithm*. [BDM+10] utilise également le recuit simulé en parallèle de

l'hyper-heuristique. Si cette méthode ne suffit pas à résoudre le problème heuristiquement, elle s'avère malgré tout très utile pour éviter de rester bloqué sur un optimum local.

Recherche tabou

La technique de la recherche tabou permet d'éviter de revisiter des solutions déjà étudiées en les conservant en mémoire. Elle est souvent qualifiée de recherche locale au sens large, car son fonctionnement en est très proche. On l'associe couramment à des méthodes mathématiques, qui lui fournissent une solution de départ à améliorer, notamment en prenant compte de contraintes préférentielles non modélisées lors de la résolution mathématique ([VH00],[BFM96]).

Hyper-heuristiques

Les hyper-heuristiques sont utilisées pour déterminer l'heuristique à employer sur la résolution. Elles permettent notamment de combiner les avantages et les inconvénients des différentes méthodes, pour les rendre plus robustes.

[ABL07] propose six hyper-heuristiques pour assister la construction et la modification de la solution courante. Chacune a un objectif particulier : l'une est complètement aléatoire et vise à agrandir l'espace de recherche ; une autre ne s'intéresse qu'au coût des solutions potentielles, et ce indépendamment de leur faisabilité, dans le but de minimiser le coût de la solution finale ; deux autres considèrent le nombre de postes non satisfaits ; et les dernières sont orientées vers la qualité de la solution, pour mieux tenir compte des contraintes préférentielles. Ainsi, le recours aux hyper-heuristiques rend les règles plus flexibles, et permet d'explorer davantage de solutions.

[BDM$^+$10] combine l'approche par hyper-heuristique avec un mélange (*shuffle*) glouton. L'hyper-heuristique choisit ici parmi une liste d'heuristiques de bas niveau, mais cela permet quand même d'élargir le nombre de solutions considérées.

Algorithmes génétiques et évolutionnistes

La fonction de $fitness$ employée dans les algorithmes génétiques porte sur les contraintes brisées. En effet, on sait qu'une solution optimale est difficile, voire impossible, à décrire - et donc à évaluer -, et ce particulièrement à mesure que les contraintes et le nombre d'agents augmentent, et que le modèle se complexifie. Dans la plupart des hôpitaux, on sait qu'on ne trouvera pas de solution satisfaisant toutes les contraintes, et on cherche alors à minimiser le nombre de contraintes violées, ainsi que l'effet de ces violations. À ce titre, il paraît plus pertinent d'évaluer les solutions par rapport aux contraintes brisées que de les comparer à un idéal probablement inatteignable et indescriptible.

C'est ainsi que [EM93] procède pour déterminer une solution de $fitness$ minimale via une procédure génétique. À cette solution est ensuite appliqué un algorithme de *local-hill climbing* pour l'améliorer autant que possible.

Le problème de la population de départ est également important, car il est difficile de tirer un planning réalisable à partir d'individus complètement aléatoires. [JYO00] propose de partir d'une population composée de solutions réalisables, déterminées au préalable. À chaque génération, on procède au croisement entre le pire planning et un autre planning tiré aléatoirement.

Aickelin ([ABL07], [AD04]) s'est particulièrement intéressé à l'approche génétique, ainsi qu'évolutionniste. Ses travaux utilisent une combinaison entre algorithme génétique et heuristique permettant de gérer le conflit qui peut apparaître entre détermination d'objectif et respect de contraintes. Son emploi de l'algorithme génétique indirect renvoie de meilleurs résultats que beaucoup de recherches avec tabou, pourtant une heuristique souvent favorisée ([AD04]). Il s'est également penché sur les

algorithmes évolutionnistes (plus particulièrement les algorithmes mémétiques, hybridation entre algorithme évolutionniste et recherche locale) dans le but de répliquer le processus d'apprentissage propre au personnel chargé de mettre en place les plannings manuellement ([ABL07]).

1.3 Construction manuelle et aide à la décision

La tendance naturelle est de baser le fonctionnement de l'algorithme de résolution du NRP sur les méthodes de *self-scheduling* (construction manuelle des plannings hospitaliers). Beaucoup s'appliquent même à reproduire cette construction manuelle ([SS98], [OO88]).

Au-delà de la méthode utilisée par l'humain, et puisqu'on a établi qu'il est difficile d'évaluer l'optimalité ou la qualité d'une solution, plusieurs approches prennent le parti de l'interactivité. [MS84] offrent la possibilité à l'utilisateur de changer les poids associés aux différents objectifs au cours de la procédure, pour mieux inclure des conditions temporelles. [Jas97] fait évaluer à un membre du personnel hospitalier l'ensemble de solutions Pareto-optimales obtenu : c'est à cette personne qu'il incombe de choisir le planning qu'elle considèrera optimal.

[AS99] est l'approche qui mise la plus sur cette interactivité. Plutôt qu'un simple logiciel de création de planning autonome, INTERDIP, dont le fonctionnement est décrit ici,est envisagé davantage comme un outil d'assistance à création de planning. L'utilisateur peut intervenir à chaque étape de la résolution : il fournit les données d'entrée, et peut dès le départ empêcher certaines affectations en les présentant comme des interdictions ; il peut placer des *breakpoints* dans le calcul de résolution, de manière qu'à chacun de ces *breakpoints*, INTERDIP renvoie les solutions retenues à l'étape courante, et l'utilisateur peut alors en supprimer, ou annuler des affectations déjà appliquées ; enfin, il peut modifier une solution et demander au logiciel de la vérifier.

Une dernière observation sur la méthode manuelle ayant inspiré des chercheurs est celle de l'apprentissage et de l'expérience propre à la personne chargée de construire les emplois du temps. Aickelin ([ABL07]) observe qu'une personne ayant l'habitude de construire un planning régulièrement développe des capacités relatives à son expérience. Il tente alors de développer, notamment avec son algorithme mémétique, un apprentissage implicite, puis explicite par l'algorithme, pour que celui-ci apprenne des plannings qu'il a déjà construits les "bons réflexes" à adopter. Dans la même idée, on peut également souligner les travaux de raisonnement par cas (CSB) de [BP06], qui proposent une approche cognitive cherchant à imiter le comportement humain et son expérience.

1.4 Précision d'adaptation aux préférences

L'interactivité peut également se limiter parfois au niveau de la phase de départ, particulièrement pour mieux adresser les contraintes souples associées aux préférences des agents. Le programme proposé par [KJ91] procède, dans une première étape, à un sondage parmi toutes les infirmières, chacune précisant ses préférences. Ces données sont ensuite fournies en entrée au programme, ainsi que les informations plus traditionnelles sur les postes requis. L'un des objectifs de ce sondage a en quelque sorte une visée psychologique : Kostreva et Jennings expliquent avoir observé que lorsque les agents sont impliqués les agents dans la construction du planning, le résultat obtenu leur paraît plus satisfaisant. Cela n'enlève rien au fait que les préférences des agents posent souvent un problème à part entière. Ce sont les premières contraintes à être sacrifiées, et bien qu'elles ne soient pas toutes explicites, elles permettent de rendre la construction de planning plus équitable.

[BS83] ajoute une contrainte d'affectation équitable, visant à répartir au mieux les postes considérés "impopulaires" entre les agents, et à s'assurer que, d'un planning à l'autre, ces postes ne sont pas affectés aux mêmes agents.

L'idée d'un index d'aversion est notamment proposée par [MP79] pour estimer la qualité des plannings précédents pour chaque agent, ce qui permet également d'atteindre une meilleure répartition.

On note enfin que le choix d'une approche favorisant la prise en compte des préférences des agents peut avoir un impact direct sur le déroulement de la résolution du problème. [Dow98], notamment, procède à une recherche tabou associée à une technique stratégique d'oscillation entre des plannings en cohérence avec les besoins d'effectifs et autres contraintes dures, et des plannings focalisés sur les préférences des agents. Cela permet de trouver un équilibre satisfaisant à la fois les exigences de base et les préférences individuelles.

Si certains axes de recherche ont effectivement abouti sur la création de logiciels utilisables en pratique par des hôpitaux ([DFM+95] avec Horoplan, [AS99] avec INTERDIP, [Dow98] avec CARE, ou encore [BDCVB99] avec PLANE), ceux-ci sont le plus souvent faits sur mesure pour un hôpital ou un système hospitalier particulier. Bien que certains de ces logiciels cherchent à maintenir malgré tout une certaine flexibilité, il est toujours difficile d'obtenir une solution qui convienne à tous les hopitaux, notamment parce que chaque pays a sa propre législation sur le fonctionnement hospitalier, par exemple. Ainsi, les logiciels pré-existant notre projet n'étaient pas nécessairement adaptés à toutes nos contraintes.

À partir de cette étude de la littérature sur le *Nurse Rostering Problem*, nous avons pris le parti d'explorer une méthode de résolution exacte d'une part, avec la programmation linéaire, et une méthode de résolution approchée d'autre part, s'appuyant sur des outils heuristiques.

2 Approche du problème et formalisation

2.1 Modèle

Le modèle qui servira de base pour représenter une instance du problème se constitue ainsi :

— Premier jour du mois :
$firstDay \in Day = \{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday, None\}$
— Nombre de jours dans le mois : $nbDays$
— Les jours du mois seront notés $j \in D = \{1, ..., nbDays\}$
— Nombre d'heures supplémentaires autorisées dans le mois : Δ
— Un poste par défaut : $defaultPost \in P$ (cf. 2.1.3)
— Un ensemble S de m services $l \in \{1, ..., m\}$ (cf. 2.1.1)

2.1.1 Service

— Un identifiant
— Un ensemble P de p postes $k \in \{1, ..., p\}$ (cf. 2.1.3)
— Un ensemble A de n agents $i \in \{1, ..., n\}$ (cf. 2.1.2)
— Une liste des agents référents $referents \in A$
— Les postes $k \in P$ requis pour chaque jour $d \in Day$ pour le service $l \in S$ dans $postRequis$ (cf. 2.2.5)
— Un score de priorité pour les postes requis
— Un ensemble C de contraintes (cf. 2.2.2, 2.2.6, 2.2.7)
— Un planning pré-défini pour le service (optionnel)

2.1.2 Agent

— Un identifiant
— Un statut $status \in \{Beginner, Rookie, Confirmed\}$
— Le nombre d'heures autorisées par mois pour l'agent (hors heures supplémentaires) δ_m
— Le nombre d'heures autorisées par semaine (7 jours glissants) δ_w
— Un calendrier *calendar* sous forme de tableau où chaque case d'indice $j \in \{1, ..., nbDays\}$ correspond à un poste $k \in p$ attribué au jour j
— Un calendrier parallèle à *calendar* nommé *calendarLock* pour bloquer les jours définis et ainsi éviter leur modificaiton par l'algorithme
— Une liste *impossiblePost* de postes $k \in P$ interdits pour l'agent (cf. 2.2.5)
— Des scores de priorité pour les postes impossibles, le nombre d'heure au mois et à la semaine

2.1.3 Poste

— Un identifiant
— La durée du poste *time*
— Une liste d'attributs *attributs* où chaque attribut est une chaîne de caractères (cf. 2.2.2, 2.2.6, 2.2.7)

2.2 Contraintes

Lors d'une première réunion introductive, une version "épurée" des contraintes à prendre en compte lors de la construction du planning nous a été fournie. Elle recense les éléments essentiels au bon fonctionnement du département, mais elle ne représente dans les faits qu'une partie de toutes les contraintes prises en compte dans la construction manuelle du planning.

À partir de cette liste furent constituées sept classes de contraintes. Chaque classe servant à recenser les contraintes portant sur un même type de problème.

Les contraintes peuvent être générales, ou s'appliquer à un service particulier. On notera :

1. **SDN** le service "Salle de naissances",
2. **GHR** le service "Grossesse à haut risque",
3. **SDC** et **UK** les services "Suite de couches" et "Unité kangourou" (une section particulière du SDC),
4. **Pool** l'ensemble d'agents pouvant être affecté à n'importe quel service.

On distinguera au sein de chaque classe les contraintes dites "dures" (C) de celles dites "préférentielles" (P).

2.2.1 Volumes horaires (C1)

Toutes les contraintes portant sur le volume horaire travaillé sont des contraintes dures, car elles portent sur des aspects factuels et/ou légaux.

1. Le volume horaire mensuel d'un agent est de 155 heures. On admet une variation Δ (représentant environ une garde de 12 heures) par rapport à ce volume horaire.

2. Si des congés sont posés, le volume horaire travaillé diminue proportionnellement au nombre de jours de congés. Une semaine de congé revient à un régime à 80% (125 heures) ; deux semaines à un régime à 50% (78 heures).
3. Une garde (J ou N) dure 12.25 heures.
4. Tout autre poste représente 7.5 heures de travail.

Modélisation :
Cette contrainte est modélisée par un Δ représentant le nombre d'heures supplémentaires autorisées dans le mois, et δ_m et δ_w représentant respectivement le nombre d'heures autorisées par mois et par semaine pour l'agent.

2.2.2 Enchaînements de postes interdits (C2 - P2)

Contraintes dures :

1. Enchaîner une garde de jour après une garde de nuit (séquence NJ) est interdit.
2. Il est interdit d'enchaîner 3 gardes de nuit. En revanche, enchaîner 3 gardes de jour est possible à titre exceptionnel pour certains services (GHR, SDC et UK).
3. Si un agent a un congé posé, il ne peut pas travailler la nuit précédente.

Contraintes préférentielles :

1. On préfèrera éviter l'enchaînement de 3 gardes de jour.
2. Les enchaînements de deux gardes après un poste de 7.5 heures sont à éviter.
3. On cherche à éviter autant que possible les journées isolées.

Généralisation :
Cette contrainte, dans un cadre plus général, revient à éviter la présence d'une suite (ou d'un enchaînement) de postes (ou de type de postes) dans le calendrier d'un agent.

Modélisation :
Cette contrainte est modélisée par une instance de la contrainte $C2 \in C$ qui contient une liste ordonnée d'attributs où chaque attribut est une chaîne de caractères. Cette liste correspond à l'enchaînement de postes non-autorisé.

2.2.3 Accompagnement des agents débutants (C3 - P3)

Contraintes dures :

1. Un agent débutant doit être "doublé" (i.e. accompagner l'agent sur son poste) par un agent référent lors de journées courtes (i.e. sur des postes de 7.5h). Le doublage est un poste à part entière.
2. Dans le service **SDN**, il n'est pas nécessaire de réquisitionner un agent référent pour doubler un agent débutant. Il suffit de faire travailler un agent référent en même temps et au même poste que l'agent débutant.
3. Deux agents avec peu d'expérience ne peuvent pas travailler le même jour sur le même poste dans le même service.

Contrainte préférentielle :

1. On préfèrera qu'un agent débutant soit accompagné par le même agent référent autant que cela sera possible.

Modélisation :
Cette contrainte est modélisée par la présence dans chaque service de la liste $referents$ ainsi que du *status* dans chaque Agent.

2.2.4 Affectations de postes interdites (C4)

Les postes de nuit sont interdits pour les agents débutants lors du premier mois au sein d'un service.

Généralisation :
Entretenir une liste de postes interdits pour le mois en cours suffit à généraliser cette contrainte.

Modélisation :
Cette contrainte est modélisée par la liste *impossiblePost* présente pour chaque Agent.

2.2.5 Quotas journaliers (C5)

Contrainte dure :
Pour chaque jour de la semaine, dans chaque service et à chaque poste, un quota fixe d'agents doit être affecté.

Par exemple, dans le service **GHR**, chaque jour un agent de jour (Jg), un agent de nuit (NG) et un agent en maturation (Mat) sont nécessaires au bon fonctionnement du service.

Modélisation :
Cette contrainte est modélisée par la liste *postRequis* présente dans chaque service.

2.2.6 Séquences conditionnelles (C6)

Contraintes dures :

1. Après deux gardes d'affilée, un agent doit avoir au moins deux jours de repos.
2. Dans le service **GHR**, lorsqu'un agent travaille de jour (Jg) le samedi, il doit alors être affecté au même poste le dimanche et le lundi suivant.

Contrainte préférentielle :
On préfère accorder 3 jours de repos à un agent ayant enchaîné deux gardes de 12h.

Généralisation :
Cette contrainte peut être traduite par une implication. Siune certaine suite (ou enchaînement) de postes *sp* est affectée, celle-ci doit nécessairement être suivie par une autre suite de postes sp'. Le jour du premier poste peut être précisé pour la suite *sp*, pour satisfaire par exemple le cas du week-end en précisant que le premier jour est un samedi.

Modélisation :
Cette contrainte est modélisée par une instance de la contrainte $C5 \in C$ qui contient deux listes ordonnées d'attributs sp et sp' où chaque attribut est une chaîne de caractères, et un jour $d \in Day$ pour préciser le jour de départ de la séquence sp.

2.2.7 Nombre de séquences minimum (ou maximum) (C7)

Contraintes dures :
Chaque agent doit travailler au moins un week end complet (samedi + dimanche) par mois.

Contrainte préférentielle :
Il est néanmoins préféré d'avoir au moins deux week-ends complets.

Généralisation :
Cette contrainte peut être traduite par la nécessité d'avoir un nombre minimum (ou maximum) d'apparitions d'une suite de postes sp dans le mois. Le jour du premier poste peut être précisé pour la suite sp, pour satisfaire par exemple le cas du week-end en précisant que le premier jour est un samedi.

Modélisation :
Cette contrainte est modélisée par une instance de la contrainte $C7 \in C$ qui contient une liste ordonnée d'attributs où chaque attribut est une chaîne de caractères. Le choix entre le minimum ou le maximum de la présence requise de la séquence sp avec $type \in \{Min, Max\}$ ainsi que du nombre nbS qui correspond à ce choix.

2.3 Données

Nous disposons d'exemples de plannings déjà construits manuellement et validés par Diane Redel, la sage-femme responsable des plannings de la maternité du CHIC, ainsi que des trames vierges à partir desquelles débute la construction. Ces documents nous permettent non seulement de nous assurer qu'il est possible de trouver une solution recevable à partir des données de départ, mais aussi de comparer les solutions de notre logiciel à celle qui fut décidée dans les faits.

3 Approche heuristique

L'approche heuristique dans notre cas est dirigée par une génération de solutions à l'aide d'une méthode gloutonne et de l'exploitation de celle-ci dans un algorithme itératif qui fera évoluer les solutions via des voisinages pour rendre la meilleure solution trouvée.

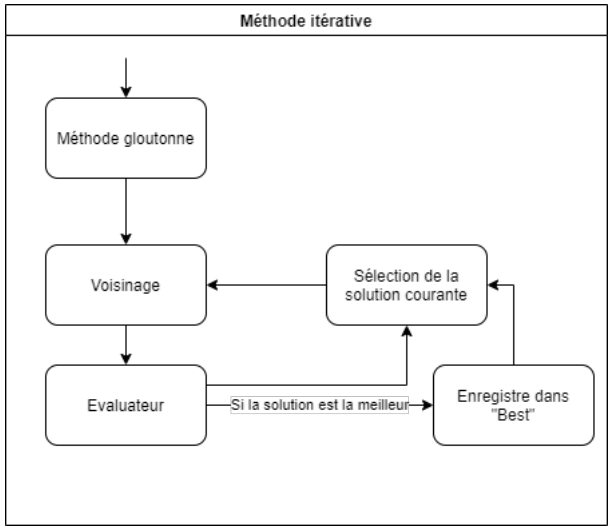

FIGURE 1 – Schéma synthétisant le fonctionnement de la méthode heuristique

3.1 Méthode gloutonne

La méthode gloutonne, dans notre approche, sert à initialiser une première solution de notre modèle afin d'être exploitable par la méthode itérative, qui sera explicitée dans la prochaine partie.
Pour illustrer le processus, nous prendrons l'exemple du service GHR.

1. L'algorithme traite directement à partir d'une instance du modèle les calendriers *calendar* pré-remplis (cf. figure 2) de chaque agent i dans chacun des services l.

   Sur les prochaines figures, chaque case correspond à l'affectation d'un poste k à un agent i le jour j. Les cases déjà remplies correspondent à des postes déjà affectés, des formations professionnelles ou à des congés. Les cases sont vides s'il n'y a aucune affectation.

	1er	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
GHR	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M
1	CA	CA	CA	O/E							O/E							O/E							O/E	CA	CA	CA	CA	CA	CA
6																								FP							
33																									cs	cs	cs				
36																															
40	CA																														
49																															
57	CA	CA																													
63	CA	CA	CA	CA	CA	CA	CA	CA																							

FIGURE 2 – Planning pré-rempli du service GHR.

2. Dans un premier temps, si des trames de planning pré-défini sont indiqués alors l'algorithme attribuera les lignes de ces trames aux agents des services concernés, en attribuant à chaque fois l'agent le moins contraint par rapport à la ligne (le moins de jours non attribuables et le moins de jours avec des postes impossibles), si il y a égalité entre plusieurs agents, on prends dans ce cas là l'agent avec le plus de jours bloqués ($calendarLock$).
3. Les trames peuvent parfois avoir des postes en trop pour certains jours, pour cela on retire les postes attribués en trop, si on a le choix de retirer un poste entre plusieurs agents alors on choisi l'agent avec le plus d'heure de travail dans le mois.
4. Pour chaque jour, les postes requis dans $postRequis$ sont récupérés et attribués aléatoirement aux agents qui n'ont aucun poste affecté. Cette initialisation permet de satisfaire la contrainte C5, sauf dans le cas où le nombre d'agents libres est inférieur au nombre de postes requis.

	1er	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
GHR	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M
1	CA	CA	CA	O/E				Ng			O/E					Mat		O/E	Ng			Mat	Mat	Ng	O/E	CA	CA	CA	CA	CA	CA
6	Ng	Ng	Ng				Mat				Mat		Jg	Jg	Jg		Ng	Jg		Mat				FP		Jg		Mat	Mat		Mat
33		Jg			Mat	Mat		Mat	Ng			Ng		Mat					Mat		Jg		Jg		cs	cs	cs	Jg		Mat	
36	Jg		Jg			Jg	Jg											Mat		Jg	Ng		Ng	**Mat**	Mat	Mat				Ng
40	CA	Mat	Mat	Jg	Jg					Ng	Jg					Jg		Ng			Mat				Ng		Jg	Ng	Ng		
49	Mat			Mat		Ng	Ng					Mat	Ng		Mat	Ng	Jg								Jg		Mat		Jg		Jg
57	CA	CA		Ng	Ng			Jg	Mat	Mat	Ng	Jg	Mat						Jg			Ng					Ng			Jg	
63	CA	CA	CA	CA	CA	CA	CA	CA	Jg	Jg				Ng	Ng		Mat			Ng		Jg		Jg		Ng				Ng	

FIGURE 3 – Planning à l'étape 2 de l'algorithme glouton

5. La dernière étape consiste simplement à remplir toutes les cases non affectées en les affectant au poste $defaultPost$ défini dans le modèle (R correspond à un repos dans la figure 4).

	1er	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
GHR	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M
1	CA	CA	CA	O/E	R	R	R	Ng	R	R	O/E	R	R	R	R	Mat	R	O/E	Ng	R	R	Mat	Mat	Ng	O/E	CA	CA	CA	CA	CA	CA
6	Ng	Ng	Ng	R	R	R	Mat	R	R	R	Mat	R	Jg	Jg	Jg	R	Ng	Jg	R	Mat	R	R	R	**FP**	R	Jg	R	Mat	Mat	R	Mat
33	R	Jg	R	R	Mat	Mat	R	Mat	Ng	R	R	Ng	R	Mat	R	R	R	R	Mat	R	Jg	R	Jg	R	cs	cs	cs	Jg	R	Mat	R
36	Jg	R	Jg	R	R	Jg	Jg	R	R	R	R	R	R	R	R	R	R	Mat	R	Jg	Ng	R	Ng	**Mat**	Mat	Mat	R	R	R	R	Ng
40	CA	Mat	Mat	Jg	Jg	R	R	R	R	Ng	Jg	R	R	R	R	Jg	R	Ng	R	R	Mat	R	R	R	Ng	R	Jg	Ng	Ng	R	R
49	Mat	R	R	Mat	R	Ng	Ng	R	R	R	R	Mat	Ng	R	Mat	Ng	Jg	R	R	R	R	R	R	R	Jg	R	Mat	R	Jg	R	Jg
57	CA	CA	R	Ng	Ng	R	R	Jg	Mat	Mat	Ng	Jg	Mat	R	R	R	R	R	Jg	R	R	Ng	R	R	R	R	Ng	R	R	Jg	R
63	CA	CA	CA	CA	CA	CA	CA	CA	Jg	Jg	R	R	R	Ng	Ng	R	Mat	R	R	Ng	R	Jg	R	Jg	R	Ng	R	R	R	Ng	R

FIGURE 4 – Planning à la fin du déroulement de l'algorithme glouton

Complexité

L'algorithme parcourt une fois le calendrier à l'étape 2 puis une autre fois à l'étape 3, ce qui nous donne pour chaque service l une complexité en $O(n * nbDays)$.

3.2 Voisinage

Afin de pouvoir progresser dans l'ensemble des solutions, nous avons choisi de générer les solutions par voisinage.

Échange de postes

Échange de postes entre deux agents i_1 et i_2 choisis aléatoirement au sein d'un même service l, lui-même choisi alétoirement parmi tout les services de S.

Échange de postes par blocs

Échange d'un bloc de postes (par exemple un bloc de distance 3 comprenant le jour 5,6 et 7) entre deux agents i_1 et i_2 choisis aléatoirement au sein d'un même service l, lui-même choisi alétoirement parmi tout les services de S.

Échange de postes inter-services

Échange de postes entre deux agents i_1 et i_2 choisis aléatoirement, l'un dans un service l aléatoirement choisi et l'autre dans un service indiqué en paramètre. Ce voisinnage permet l'exploitation du pool.

Changement d'un poste

Changement ponctuel du poste k affecté le jour j à l'agent i par un autre poste k', avec $j \in D$, $i \in A$ et $k' \in P$ choisis aléatoirement. Celui-ci n'est pour l'instant pas utilisé car il augmente le nombre de solutions explorables inintéressantes (celles qui brisent la contrainte C5 plus particulièrement).

Distance

On définit la distance d'un voisinage comme étant le nombre d'échanges / changements de poste par rapport à la solution courante.
Comme les solutions voisines à un seul échange de poste de la solution courante sont très souvent trop similaires pour être évaluées différemment, nous avons ajouté à nos méthodes de voisinage un paramètre de distance *range* qui permet alors de générer des voisins à une distance aléatoire choisi entre 1 et *range*.

3.3 Evaluation de la solution

Le checker est un outil permettant d'évaluer la qualité des solutions et ainsi guider la recherche de meilleures solutions.
Afin de donner un score à une solution, chaque contrainte du modèle est vérifiée sur la solution. À chaque fois qu'une contrainte est brisée, on applique un malus au score dépendant du poids de la contrainte (paramètre *priority* dans les contraintes C2, C6 et C7, reglé à 1 pour les autres contraintes pour le moment).
Actuellement le meilleur score atteignable est 0, signifant alors que la solution satisfait toutes les contraintes.

Contraintes prises en compte

Actuellement toutes les contraintes sauf la contrainte C3 (qui demande une coopération entre chaque service) sont prises en compte par l'évaluateur.

Complexité

Le checker vérifie les contraintes localement en ne vérifiant que les éléments qui ont changé dans la solution grâce à un suivi des échange et une mise en mémoire des informations concernant les contraintes brisées auparavant.
Pour chaque échange *swap*, seul 2 agents sont concernés. Dans le pire des cas tout le calendrier des 2 agents vérifié (ce qui n'est pas le cas bien évidement). On se retrouve donc avec une complexité en $O(2 * swap * nbDays)$ pour chacune des contraintes $c \in C$

3.4 Méthode itérative

Cette méthode va permettre de naviguer dans l'ensemble des solutions avec les solutions générées par les algorithmes gloutons et de voisinage et ainsi retourner la meilleure solution trouvée lors de son exécution. Soit *nbI* le nombre d'itération de l'algorithme, *range* la distance de voisinage, *best* la meilleure solution trouvée.

1. L'algorithme va premièrement générer une première solution avec l'algorithme de la méthode gloutonne qui sera la solution courante ainsi que *best*. Puis l'évaluer.
2. On produit un voisin v à une distance maximum *range* de la solution courante.
3. On évalue la solution avec le checker.
4. Si son score est meilleur que celui de la solution courante, alors *best* devient v. v a ensuite une probabilité de 0.9 de devenir la solution courante.

5. Sinon on a une probabilité de 0.002 de prendre v comme solution courante alors qu'elle est moins bonne. On a aussi la même probabilité de 0.0002 de repartir avec une nouvelle solution courante généré avec la méthode gloutonne.
6. On réitère nbI fois à partir de l'étape 2.
7. On retourne alors la meilleure solution trouvée *best*.

	1er	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
GHR	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M
1	CA	CA	CA	O/E	Jg	Ng	R	R	Mat	Ng	O/E	R	Jg	Mat	R	R	Ng	O/E	Ng	R	R	R	R	Mat	O/E	CA	CA	CA	CA	CA	CA
6	Mat	Ng	R	R	Mat	Jg	R	R	R	R	R	R	R	Jg	Jg	Jg	R	R	Jg	Ng	R	R	R	**FP**	Jg	R	R	Jg	Jg	Jg	Jg
33	Ng	R	R	Jg	Ng	R	R	R	R	Jg	Jg	R	R	R	R	Mat	Jg	R	R	R	Mat	Mat	R	R	cs	cs	cs	R	R	R	R
36	Jg	Mat	Jg	R	R	R	Ng	Ng	R	R	Mat	Ng	R	R	R	R	R	R	R	Mat	R	R	R	**R**	Ng	Mat	R	R	Mat	Mat	R
40	CA	R	Mat	Ng	R	R	Jg	Jg	Jg	R	R	R	R	R	Mat	Ng	R	Ng	R	R	R	Ng	Ng	R	R	Ng	Ng	R	R	R	Ng
49	R	Jg	Ng	R	R	Mat	R	R	R	Mat	Ng	R	R	Ng	Ng	R	R	R	Mat	Jg	R	R	R	R	Mat	Jg	R	R	Ng	Ng	R
57	CA	CA	R	Mat	R	R	Mat	Mat	R	R	R	Jg	Ng	R	R	R	Mat	Jg	R	R	Ng	R	Mat	Jg	R	R	Jg	Mat	R	R	R
63	CA	CA	CA	CA	CA	CA	CA	CA	Ng	R	R	Mat	Mat	R	R	R	R	Mat	R	R	Jg	Jg	Jg	Ng	R	R	Mat	Ng	R	R	Mat

FIGURE 5 – Planning retourné par l'algorithme itératif après 100000 itérations

Complexité

Cette méthode partage évidemment les performances des méthodes gloutonne, de voisinage et de l'évaluateur. L'évaluateur étant l'étape la plus coûteuse en temps et la plus utilisée, on peut donc considérer qu'elles partagent la même complexité.

4 Programmation linéaire

4.1 Outils utilisés

Durant ce projet, nous avons pu travailler avec différentes méthodes afin d'obtenir des calendriers répondants à des contraintes.
En effet, nous avons travaillé avec des méthodes heuristiques d'un côté mais nous avons également utilisé la programmation linéaire pour essayer de répondre au problème donné.
Pour ce faire, il nous a fallu dans un premier temps trouver une librairie de PLNE (programmation linéaire en nombres entiers) pour avoir un solveur qui soit libre de droits afin que le CHIC puisse utiliser notre solution sans problème.

Après quelques recherches, nous avons trouvé une librairie qui s'appelle MIPCL répondant à ces critères et semblant être intéressante. Malheureusement, l'installation de celle-ci fut compliquée, certains d'entre nous n'ayant jamais fait de C++, langage dans lequel nous codons tout le projet et notamment à cause de problèmes d'IDE.

Une fois cela fait, nous avons débuté la prise en main de la librairie mais quand nous avons commencé à prendre nos repères dessus le site web de la documentation de la librairie a disparu et nous nous sommes rendu compte que la librairie n'était probablement plus maintenue à jour depuis un moment.

Nous nous sommes donc finalement dirigés vers une autre librairie non commerciale qui s'appelle SCIP et qui est beaucoup plus fiable et complète même si elle est potentiellement moins libre de droits (raison pour laquelle nous n'avons pas opté pour elle directement).

4.2 Variables et contraintes

Qui dit programmation linéaire dit **variables**, **fonction objectif** et **contraintes**.
Penchons-nous tout d'abord sur les variables. Les variables sont binaires et pour un agent a, travaillant un jour j à un poste p, nous avons une variable $x_{a,j,p}$ qui est égale à 1 si l'agent travaille à ce poste à ce moment-là, 0 sinon.

En ce qui concerne la **fonction objectif**, nous avons opté pour une fonction simple :

$$\min \sum_{a=0}^{n} \sum_{j=0}^{nbDays} \sum_{p=0}^{nbPostes} time * x_{a,j,p}$$

La fonction consiste à minimiser la somme de la valeur de toutes les variables binaires multipliées par le nombre d'heures *time* du poste en question dans la variable. Nous avons opté pour cette fonction car il semble intéressant de minimiser le nombre d'heures effectuées par les agents qui seront sélectionnés pour travailler certains jours à certains postes durant un mois de travail.
Une autre fonction pourrait également consister à minimiser la somme de la valeur des variables mais sans le produit avec *time*. En lançant le programme linéaire pour tester les deux fonctions objectifs nous n'avons pas vu de différence notable entre les valeurs des fonctions objectifs ou entre les calendriers finaux, nous utilisons donc la première fonction qui sera sûrement plus informative lorsque le nombre de contraintes dans le programme linéaire sera plus élevé.

Une fois les variables créées, nous pouvons les placer dans des contraintes. À ce stade nous avons implémenté les contraintes suivantes avec la programmation linéaire :

— C1 : Le volume horaire mensuel et hebdomadaire
  — Nombres d'heures par semaine d'un agent $< nbHeuresSemaineAgent$
  — Nombres d'heures par mois d'un agent $< nbHeuresMoisAgent$
— Un seul poste maximum par jour et par personne.

$$\sum_{p=0}^{nbPostes} x_{a,j,p} \leq 1$$

— C2-1 : Enchaîner une garde de jour après une garde de nuit est interdit.
Pour tout agent a, jour j, poste de nuit p et poste de jour p_1 :

$$x_{a,j,p} + x_{a,j+1,p_1} \leq 1$$

— C2-3 : Interdit d'enchaîner 3 gardes de nuit.
Pour tout agent a, jour j, poste de nuit p :

$$x_{a,j,p} + x_{a,j+1,p} + x_{a,j+2,p} \leq 2$$

— C5, service GHR : Un seul agent par jour pour les postes *Jg*, *Mat* et *Ng*

— C2-2, services GHR, SDC et UK : 3 jours d'affilée possible (mais doit être exceptionnel)
Pour tout agent a, poste p en GHR, SDC ou UK :

$$x_{a,j,p} + x_{a,j+1,p} + x_{a,j+2,p} + x_{a,j+3,p} \leq 3$$

4.3 Mise en oeuvre

Premièrement, afin d'avoir une structure qui facilite la création de contraintes nous avons créé une classe $VariableData$ qui englobe, pour chaque variable SCIP l'agent, le jour, le poste en question et un identifiant. En effet, ces données sont "perdues" lors de la création de la variable en question qui n'est à la base représentée que par un identifiant.
Cela est problématique car pour créer des contraintes nous avons besoin de récupérer certaines variables en particulier. On peut par exemple vouloir créer une contrainte qui s'applique sur chaque agent, ce qui implique de récupérer tour à tour les variables représentant un certain agent, hors cela n'est pas possible si on ne sauvegarde pas l'agent à qui est "attribué" la variable lorsqu'on crée la dite variable. C'est pour cela qu'il est impératif d'avoir une structure avec les données associées à chaque variable.

Pour créer ces contraintes et obtenir un calendrier résultat nous avons procédé concrètement ainsi :

1. Nous créons le problème de minimisation SCIP à résoudre.
2. Nous effectuons des boucles imbriquées dans l'ordre suivant : Pour chaque jour du mois, pour chaque agent, pour chaque poste dans ce service nous allons créer les variables SCIP qu'on indique comme binaires et ayant pour coefficient dans la fonction objectif la valeur *time* du poste attribué à l'agent courant.
3. On englobe cette variable dans le type $VariableData$ puis on l'ajoute à une liste afin d'y enregistrer également l'agent, le poste, le jour et l'identifiant de la variable pour les contraintes futures.

4. Après cela, il nous faut boucler d'une certaine façon afin de récupérer les variables dans un ordre précis pour pouvoir les stocker et créer des contraintes qui s'appliqueront à ces variables en particulier. Par exemple, boucler premièrement sur les jours du mois pour appliquer une contrainte liée à des jours donnés, ou bien boucler d'abord sur les agents pour obtenir des contraintes qui limitent le nombre d'heures possibles par semaine (c'est typiquement le cas pour les contraintes C1).
5. Enfin, si le problème a été résolu, on met à jour le modèle de base grâce aux valeurs des variables obtenues auquel cas on sait quel agent doit travailler (ou non) à quel poste pour chaque jour donné puis on libére les variables, les contraintes et le problème SCIP.

5 Expérimentations

Tout au long de nos travaux et essais, l'avis de Diane Redel nous a permis de procéder à une seconde évaluation, portant plus sur la faisabilité de la solution que sur sa simple adéquation aux contraintes.

5.1 Instances

5.1.1 Génération d'instances

Dans l'optique de tester nos algorithmes nous avons jugé intéressant d'avoir un jeu de données autre que les calendriers constitué des agents/postes du CHIC de sorte à généraliser le problème le plus possible.
De ce fait, nous avons créé un générateur d'instances qui se base principalement sur l'aléatoire pour obtenir des instances (de modèles) de tests pratiquement uniques en fonction de paramètres donnés (liste des paramètres en partie 2.5 de la notice).

La fonction se découpe en plusieurs étapes :

1. On crée des contraintes basiques qui concernent pour la plupart le domaine légal (cf. 2.2.1) et qui sont communes à tous les services du CHIC voire aux hôpitaux en France et on les attribuera plus tard aux services.
2. On crée les $nbServices$ services.
3. Pour chaque service :
   — si le paramètre optionnel $nbPostsPerService$ est renseigné on crée $nbPostsPerService$ postes par service, ce qui aura pour conséquence de ne pas prendre en compte la valeur $nbPosts$ comme nombre de postes total dans les services. On obtiendra donc $nbPostsPerService * nbServices$ postes.
   — sinon, on tire aléatoirement un nombre de postes entre 0 et $\frac{nbPostsAvailable}{nbServices}$. Si le nombre est inférieur à 2, on lui donne la valeur 2 car un service avec si peu de postes n'est pas très intéressant. Donc, si jamais certains services sont trop peu riches en postes on se permet de ne pas respecter la valeur $nbPosts$.
4. Une fois le nombre de postes pour le service courant trouvé, on va donner des caractéristiques à ce poste. Il y aura 50% de chance qu'il soit long (12.25h) ou court (7.5h) et $\frac{2}{3}$ de chance qu'il soit de jour, sinon il sera de nuit. On donne ensuite à ce poste (et tous les autres) un nombre de personnel requis de 1. Lorsque tous les postes du service courant sont créés on applique les contraintes créées plus tôt à ce service.
5. On s'occupe ensuite des agents, il y a 80% de chance pour qu'un agent ait un statut `Confirmé`, sinon il sera `Débutant`. Après cela, on vérifie si le paramètre $nbAgentsPerService$ a été indiqué et si c'est le cas on attribue les agents aux services dans lesquels il y a encore de la place. Sinon, on l'attribue de manière aléatoire à un service.
6. Le dernier point concerne le pré-remplissage du calendrier pour simuler des congés et postes déjà connus pour certains agents à des jours donnés.
   Nous attribuons des jours de congés à certains agents de manière aléatoire en se basant sur deux probabilités : $proba_1er_conge$ et $proba_suite_conge$ qui sont soit renseignés dans les paramètres soit avec comme valeurs par défaut 3% pour le premier jour et 70% pour le second. Si un agent obtient un premier jour de congé il aura ainsi $proba_suite_conge$% de chance d'en avoir

un second, puis on diminue de 5% la chance qu'il y ait un congé chaque jour qui suivra. Enfin, on a 5% de chance d'attribuer un poste tiré aléatoirement (hors repos) à cet agent s'il n'a pas posé de congé.

5.1.2 Classes d'instances

Une fois notre générateur fonctionnel, nous avons pu créer des instances diverses que nous pouvons regrouper par classes. En effet, afin de simuler des modèles plus ou moins complexes et avec des propriétés plus ou moins spécifiques, on peut mettre certaines instances dans un même lot. Nous avons donc testé nos algorithmes sur les quelques instances suivantes :

1. `Classe 1` : Les instances constituées d'un seul service. Pratiques pour observer le déroulement des algorithmes lorsqu'on ne s'occupe pas de différents services et qu'on se concentre sur une exécution "locale".
   - $m_{1servicePetit}$: Instance avec un seul service, et seulement quelques agents et postes.
   - $m_{1serviceTresGrand}$: Instance également avec un seul service mais beaucoup plus dense, il est composé de 10 postes (environ, i.e 6.1.1.3.) et 30 agents.
2. `Classe 2` : Les instances constituées de peu de services, qui peuvent servir à analyser le comportement des algorithmes quand on a plus d'un service.
   - $m_{2servicesPetits}$: Instance avec deux services composés au total d'un tout petit plus d'agents et de postes que $m_{1servicePetit}$.
   - $m_{2servicesFevrierBissextile}$: Instance avec 2 petits services mais avec la particularité d'être pendant le mois de février (avec 29 jours) et avec $overtime = 0$, donc aucune heure supplémentaire autorisée dans le mois.
   - $m_{3servicesEte}$: Instance avec 3 petits services et beaucoup de congés afin de simuler un mois pendant les vacances d'été, $overtime = 10$ également.
3. `Classe 3` : Les instances avec beaucoup de services, servant à tester les algorithmes dans des conditions beaucoup plus proches de la réalité par rapport aux instances des classes 1 et 2.
   - $m_{6servicesPetits}$: Instance avec 6 petits services (7 postes, 20 agents)
   - $m_{6servicesGrands}$: Instance se voulant réaliste en ayant des paramètres similaires à ceux du CHIC : 6 grands services composés d'au total 20 postes et 70 agents et un *overtime* de 25h.

5.2 Critères d'évaluation

Maintenant que l'on dispose de données pour tester nos algorithmes il va avant tout falloir déterminer de quelle façon les évaluer.

- Pour la méthode heuristique, le checker est utilisé pour donner une valeur que l'on cherche à minimiser à la solution afin d'atteindre la valeur 0 qui indique qu'aucune contrainte n'a été violée.
- Pour la méthode PLNE, on cherche également à atteindre la valeur 0 car le but est de minimiser la valeur de la fonction objectif. Si on obtenait 0, cela signifierait qu'aucun agent ne travaille pendant le mois, or avec les contraintes mises en place cela est impossible. On va donc vouloir un score qui s'en rapproche le plus possible.

Les points en commun de ces évaluations sont le temps d'exécution des algorithmes et à quel point la solution donnée est proche d'une solution acceptable.

5.3 Comparaison

Ces comparaisons concernent des résultats générés avant juillet 2020.

On a pu voir sur la figure 5 (cf. 3.4) une exécution de l'algorithme itératif sur un service du CHIC. On va maintenant pouvoir tester cet algorithme ainsi que celui de la PL sur des instances générées, mais avant on peut comparer le service GHR obtenu par méthode itérative avec un planning obtenu via l'algorithme de PL :

	1er	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
GHR	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M	Me	J	V	S	D	L	M
1	Ca	Ca	Ca	O/E	Jg	Jg	Jg	Jg	Jg	Jg	O/E	Jg	R	Jg	Jg	Jg	Jg	O/E	R	Jg	R	R	R	R	O/E	Ca	Ca	Ca	Ca	Ca	Ca
6	Jg	Jg	Jg	Ng	R	R	R	Ng	Ng	R	Jg	Ng	R	Ng	Ng	R	Ng	Ng	R	R	R	R	R	FP	R	R	R	R	R	R	R
33	Ng	Ng	R	Jg	Ng	R	R	R	R	Ng	Ng	R	Jg	R	R	Ng	R	Jg	Jg	Ng	Ng	R	R	R	CS	CS	CS	R	R	R	R
36	R	R	Ng	R	R	Ng	Ng	R	R	R	R	R	Ng	R	R	R	R	R	Ng	R	Jg	Jg	Jg	Jg	R	Jg	Jg	Jg	R	R	R
40	Ca	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	Mat	R	R	Ng	Ng	Mat	Jg	Ng	Ng	R	Jg	Jg	R
49	Mat	Mat	R	R	R	R	Mat	R	R	R	R	R	R	R	R	R	R	Mat	R	R	R	Mat	Mat	Ng	Ng	Mat	R	Ng	Ng	R	Jg
57	Ca	Ca	Mat	Mat	Mat	Mat	R	Mat	R	R	R	R	Mat	R	R	R	R	R	R	Mat	Mat	R	R	R	Mat	R	R	Mat	R	Ng	Ng
63	Ca	Ca	Ca	Ca	Ca	Ca	Ca	Ca	Mat	Mat	Mat	Mat	R	Mat	Mat	Mat	Mat	R	R	R	R	R	R	R	R	R	Mat	R	Mat	Mat	Mat

FIGURE 6 – Planning retourné par l'algorithme de PL sur le service GHR

On constate facilement une différence par rapport à la figure 5, car ici les postes sont attribués aux agents de manière beaucoup moins homogène et des postes s'enchaînent sur de plus longues périodes sans repos. Cela ne devrait pas arriver, mais s'explique par le fait qu'il y a moins de contraintes implémentées en PL que dans la méthode heuristique, comme le fait qu'il n'est normalement pas possible d'avoir plus de trois journées de travail de façon consécutive.

Passons maintenant aux exemples de calendriers à partir d'instances générées composées des paramètres suivants :
Le premier jour du mois est mercredi, il y a 30 jours dans le mois, 6 services, 25h d'*overtime* possibles, 20 postes, 70 agents, 60h par semaine par agent et 155h par mois.

Les postes générés se lisent ainsi :
S+"Numéro de service"+P+"numéro de poste"+L (si poste long)+D(ay) ou N(ight) en fonction du moment où est effectué le poste dans la journée.

Itératif																															
Dates	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
Jours	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	
11	R	R	R	R	R	R	R	R	R	R	R	S3P0D	S3P1D	R	S3P0D	S3P1D	S3P0D	R	R	S3P1D	S3P0D	R	R	R	R	R	R	R	R	R	R
19	S3P0D	S3P0D	S3P1D	R	R	R	R	R	R	R	R	R	R	R	R	R	R	CA	R	R	R	R	R	R	R	R	R	R	R	R	R
31	R	R	CA	CA	R	R	CA	CA	R	R	R	R	R	R	R	R	R	R	R	R	CA	R	R	R	R	R	R	R	R	R	S3P0D
41	R	R	R	R	R	R	R	R	R	S3P0D	S3P1D	R	R	S3P1D	S3P1D	R	R	S3P0D	S3P0D	R	R	S3P1D	S3P1D	R	R	R	R	R	R	R	S3P1D
42	R	CA	R	R	R	S3P1D	S3P1D	S3P1D	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	CA	CA	CA	
46	R	R	R	R	S3P1D	S3P0D	S3P0D	S3P0D	R	R	S3P0D	CA	R	R	R	R	S3P1D	S3P1D	R	R	R	R	R	R	R	R	R	R	R	R	
53	R	R	R	R	R	R	R	R	S3P1D	R	R	R	R	R	R	R	R	R	R	CA	CA	R	R	R	R	S3P0D	S3P1D	S3P0D	S3P1D	R	
55	S3P1D	S3P1D	R	CA	R	R	R	R	S3P0D	S3P1D	R	R	R	CA	R	R	R	R	R	R	R	R	R	R	S3P0D	S3P0D	R	R	R	R	
56	R	R	S3P0D	S3P0D	R	R	R	CA	R	R	R	S3P1D	R	R	R	R	R	R	S3P1D	S3P0D	S3P1D	S3P0D	R	R	R	R	R	R	R	R	
69	R	R	R	S3P1D	S3P0D	R	R	R	R	R	R	R	S3P0D	S3P0D	R	S3P0D	R	R	R	R	R	R	S3P0D	S3P1D	S3P1D	S3P1D	S3P0D	S3P1D	S3P0D	R	

FIGURE 7 – Planning retourné par l'algorithme itératif sur un des services du modèle

Pour l'instance sur laquelle on teste le PL, il s'agit de la même classe d'instance et tous les

paramètres sont identiques :

PL																													
Dates	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Jours	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T
9	R	R	S0P0LD	R	R	R	S0P1LD	S0P1LD	S0P1LD	R	R	R	R	R	R	R	R	R	R	S0P1LD	S0P1LD	R	S0P1LD	S0P1LD	R	R	S0P1LD	S0P1LD	S0P1LD	S0P1LD
16	R	R	R	R	S0P1LD	S0P1LD	S0P0LD	R	R	S0P1LD	R	R	S0P1LD	R	R	R	R	R	R	S0P0LD	R	R	R	S0P0LD	R	S0P1LD	R	R	S0P0LD	S0P0LD
19	R	R	R	R	R	R	R	R	S0P0LD	R	R	R	R	R	R	R	S0P1LD	R	R	R	R	S0P1LD	R	S0P1LD	CA	R	R	S0P0LD	R	R
48	S0P0LD	S0P0LD	S0P1LD	S0P0LD	S0P0LD	CA	CA	CA	CA	CA	CA	S0P0LD	S0P0LD	R	R	R	R	R	R	CA	CA	R	S0P1LD	R	R	R	R	R	R	R
49	R	R	R	R	R	R	R	S0P0LD	R	R	R	R	R	R	R	S0P1LD	R	R	S0P0LD	R	S0P0LD	R	S0P0LD	R	R	R	R	R	R	R
50	R	R	R	S0P1LD	R	R	R	R	R	R	S0P1LD	S0P1LD	R	R	S0P1LD	R	R	S0P0LD	R	R	R	S0P0LD	R	R	R	R	R	R	R	R
57	R	S0P1LD	R	R	R	R	R	R	R	R	R	R	R	S0P0LD	S0P0LD	S0P0LD	S0P0LD	S0P1LD	S0P1LD	R	CA	R	R	R	S0P0LD	R	S0P1LD	R	R	R
64	S0P1LD	CA	R	R	R	S0P0LD	R	R	R	S0P0LD	S0P0LD	R	R	S0P1LD	R	R	S0P0LD	R	R	R	R	R	R	R	S0P1LD	S0P0LD	S0P0LD	R	R	CA

FIGURE 8 – Planning retourné par l'algorithme de PL sur un des services du modèle

Ces deux services de tailles similaires sont composés chacun de 2 postes avec un effectif d'une personne à avoir par jour et par poste. Sur le calendrier de la figure 8, on peut voir comme sur la figure 6 plusieurs enchaînements qui durent plus de jours qu'il est normalement autorisé par les contraintes du CHIC. Néanmoins, on remarque par exemple que les deux postes sont tous les jours affectés à des agents comme cela devrait être le cas.

Id	Classe	nbS	nbP	nbAg	Score Init.	Durée	Itérations	Score Sol.	Temps Sol.	Itérations Sol.
1	1	1	4	7	-3361	13 min	100 000	-995	20 sec	2 494
2	1	1	4	7	-1676	12 min	100 000	-71	7 min	56 692
3	2	3	6	15	-3464	30 min	93 335	-818	14 min	47 314
4	3	6	20	70	-3860	2 h	122 582	-980	1 h 45 min	110 407
5	3	6	20	70	-9305	12 h	755 494	-1528	5 h	327 183

TABLE 1 – Tableau des résultats d'exécutions de l'agorithme itératif sur des instances générées

nbS = nombre de services, nbP = nombre de postes, nbAg = nombre d'agents

La solution renvoyée par la méthode heuristique respecte donc plus de contraintes, même sur une instance générée, bien que cette solution (dernière ligne du tableau) ait un score de -1528 (et -9305 au départ), mais il y a une grosse différence qu'on ne voit pas en analysant seulement les calendriers solutions, c'est le temps pour les obtenir. En effet, comme on peut le voir sur le tableau, l'algorithme itératif a tourné pendant 12 heures (et 750 000 itérations) bien que la solution optimale ait été trouvée en 5 heures, tandis que l'algorithme de PL a tourné seulement quelques secondes.

La méthode itérative est relativement coûteuse, principalement à cause du checker qui est la partie la plus coûteuse de l'algorithme avec des possibilités de tomber sur des minima locaux pendant un long nombre d'itérations. On peut notamment voir sur la première instance du tableau que la solution optimale (dans le temps/nombre d'itérations impartis) est trouvée en seulement 20 secondes là où l'algorithme a continué son exécution pendant 13 minutes. Plus l'instance est complexe, plus le nombre d'itérations sera crucial pour trouver une meilleure solution au problème.

D'un côté, si les contraintes obtenues dans la méthode heuristique étaient appliquées dans leur totalité par le programme linéaire, il serait plus intéressant d'utiliser l'algorithme de PL pour obtenir une solution similaire beaucoup plus rapidement. Néanmoins, comme nous l'avons vu dans l'état de l'art (cf. 1), la programmation linéaire est souvent limitée au respect des contraintes simplifiées, et implémenter toutes ces contraintes peut s'avérer compliqué dans la pratique.
D'un autre côté, l'heuristique a l'avantage d'être plus exploitable que la PL, si l'on disposait d'un checker plus rapide, on pourrait effectuer beaucoup plus de changements de voisinage et avoir des solutions ne brisant plus aucune contrainte.

5.4 Nouveaux résultats

Ces résultats ont été générés entre juillet et septembre 2020.

Nous avons ici concentré nos effort sur la méthode heuristique sur le modèle du mois de Mars 2020 du CHIC.
Durant cette période le checker a grandement été amélioré, permettant ainsi de générer des plannings avec plus d'itération en un temps plus raisonnable. De même l'algorithme itératif a été étoffé d'un nouvel algorithme glouton permettant l'exploitation de trames de planning pré-défini ainsi que de nouveaux voisinages.

Nous avons donc généré des plannings avec 1 million d'itérations de l'algorithme itératif avec un temps moyen de 30 minutes. Les derniers plannings générés ont été jugés acceptable par Diane Redel dans le sens où ils permettent un gain de temps dans la réalisation du planning final, ce gain de temps est estimé à une division par 2 du temps initial en comptant le temps de saisie des données.

6 Conclusion

Obtenir un calendrier respectant les contraintes fournies par le CHIC à partir d'un calendrier de base était notre but initial, et comme nous avons pu le voir la tâche fut compliquée.

Nous nous sommes divisés en deux équipes afin d'explorer d'un côté les méthodes heuristiques et de l'autre la programmation linéaire mais nous ne sommes pas parvenus au résultat espéré en début de projet.

En effet, dû à quelques soucis du côté de la programmation linéaire avec les librairies utilisées et leurs installations nous avons perdu du temps et n'avons réellement commencé à travailler sur l'implémentation de contraintes que fin avril. Néanmoins, nous avons pu établir une base qui pourrait être approfondie afin d'arriver à une solution admissible.

D'autre part, comme nous avons pu le voir lors de nos tests sur les instances générées ou le service GHR du CHIC, nous avons pu obtenir de meilleurs résultats via les méthodes heuristiques avec des calendriers ressemblant beaucoup plus à des solutions réalistes même si cela est actuellement plus coûteux en temps.

7 Perspectives

Les perspectives pour ce projet sont nombreuses :
— Développer plus d'algorithmes gloutons et de types de voisinages pour la partie heuristique
— Ajout des contraintes pas encore prises en compte

— Réaliser une interface utilisateur complète : actuellement, l'interface utilisateur passe principalement par l'appel des fonctions dans le fichier *main.cpp*. Il est possible de créer un fichier .xlsx (Excel, Libre Office) affichant les plannings générés.
— Création d'une méthode hybride alliant la résolution par programmation linéaire et la résolution par heuristique

Références

[ABL07] U Aickelin, E K Burke, and J Li. An estimation of distribution algorithm with intelligent local search for rule-based nurse rostering. *Journal of the Operational Research Society*, 58(12) :1574–1585, December 2007.

[AD04] Uwe Aickelin and Kathryn A. Dowsland. An indirect Genetic Algorithm for a nurse-scheduling problem. *Computers & Operations Research*, 31(5) :761–778, April 2004.

[AR81] Jeffrey L. Arthur and A. Ravindran. A multiple objective nurse scheduling model. *IISE Transactions*, 13(1) :55–60, March 1981.

[AS99] Slim Abdennadher and Hans Schlenker. Interdip - an interactive constraint based nurse scheduler. 1999.

[BC10] Edmund K. Burke and Tim Curtois. An ejection chain method and a branch and price algorithm applied to the instances of the first international nurse rostering competition, 2010. 2010.

[BCQVB13] Edmund Burke, Timothy Curtois, Rong Qu, and Greet Vanden Berghe. A time predefined variable depth search for nurse rostering. *Informs Journal on Computing*, 25, 08 2013.

[BDCBVL04] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The State of the Art of Nurse Rostering. *Journal of Scheduling*, 7(6) :441–499, November 2004.

[BDCVB99] Edmund Burke, Patrick De Causmaecker, and Greet Vanden Berghe. A Hybrid Tabu Search Algorithm for the Nurse Rostering Problem. In G. Goos, J. Hartmanis, J. van Leeuwen, Bob McKay, Xin Yao, Charles S. Newton, Jong-Hwan Kim, and Takeshi Furuhashi, editors, *Simulated Evolution and Learning*, volume 1585, pages 187–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Series Title : Lecture Notes in Computer Science.

[BDM+10] Burak Bilgin, Peter Demeester, Mustafa Mısır, Wim Vancroonenburg, GreetVanden Berghe, and Tony Wauters. A hyper-heuristic combined with a greedy shuffle approach to the nurse rostering competition. 2010.

[BFM96] Ilham Berrada, Jacques A. Ferland, and Philippe Michelon. A multi-objective approach to nurse scheduling with both hard and soft constraints. *Socio-Economic Planning Sciences*, 30(3) :183–193, September 1996.

[BGBB20] David Baez, Christelle Guéret, Odile Bellenguez, and Ludovic Billard. Nurse rostering avec prise en compte des remplacements : Application au chu de nantes. EasyChair Preprint no. 3213, EasyChair, 2020.

[BP06] Gareth R. Beddoe and Sanja Petrovic. Selecting and weighting features using a genetic algorithm in a case-based reasoning approach to personnel rostering. *European Journal of Operational Research*, 175(2) :649–671, December 2006.

[BS83] Roger A. Blau and Alan M. Sear. Nurse scheduling with a microcomputer :. *Journal of Ambulatory Care Management*, 6(3) :1–13, August 1983.

[CDDC+19] Sara Ceschia, Nguyen Dang, Patrick De Causmaecker, Stefaan Haspeslagh, and Andrea Schaerf. The Second International Nurse Rostering Competition. *Annals of Operations Research*, 274(1-2) :171–186, March 2019.

[CY93] J. G. Chen and T. W. Yeung. Hybrid expert-system approach to nurse scheduling. *Computers in Nursing*, 11(4) :183–190, August 1993.

[DFM+95] S. J. Darmoni, A. Fajner, N. Mahé, A. Leforestier, M. Vondracek, O. Stelian, and M. Baldenweck. HOROPLAN : computer-assisted nurse scheduling using constraint-based programming. *Journal of the Society for Health Systems*, 5(1) :41–54, 1995.

[Dow98] Kathryn A. Dowsland. Nurse scheduling with tabu search and strategic oscillation. *European Journal of Operational Research*, 106(2-3) :393–407, April 1998.

[EM93] Fred F. Easton and Nashat Mansour. A distributed genetic algorithm for employee staffing and scheduling problems. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 360–367, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[HDCSS14] Stefaan Haspeslagh, Patrick De Causmaecker, Andrea Schaerf, and Martin Stølevik. The first international nurse rostering competition 2010. *Annals of Operations Research*, 218(1) :221–236, July 2014.

[Jas97] Andrzej Jaszkiewicz. A Metaheuristic Approach to Multiple Objective Nurse Scheduling. *Foundations of Computing and Decision Sciences*, 22, January 1997.

[JSV98] Brigitte Jaumard, Frédéric Semet, and Tsevi Vovor. A generalized linear programming model for nurse scheduling. *European Journal of Operational Research*, 107(1) :1–18, May 1998.

[JYO00] Ahmad Jan, Masahito Yamamoto, and Azuma Ohuchi. Evolutionary algorithms for nurse scheduling problem. In *Proceedings of the 2000 Congress on Evolutionary Computation, CEC00, San Diego, ISBN*, pages 196–203. IEEE Press, 2000.

[KJ91] Michael M. Kostreva and Karen S.B. Jennings. Nurse scheduling on a microcomputer. *Computers & Operations Research*, 18(8) :731–739, January 1991.

[LH10] Zhipeng Lü and Jin-Kao Hao. Adaptive local search for the 1rst international nurse rostering competition. 2010.

[LLR03] Haibing Li, Andrew Lim, and Brian Rodrigues. A hybrid AI approach for nurse rostering problem. In *Proceedings of the 2003 ACM symposium on Applied computing - SAC '03*, page 730, Melbourne, Florida, 2003. ACM Press.

[LOR19] Antoine Legrain, Jérémy Omer, and Samuel Rosat. A rotation-based branch-and-price approach for the nurse scheduling problem. *Mathematical Programming Computation*, October 2019.

[MBL09] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni. Solving Nurse Rostering Problems Using Soft Global Constraints. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732, pages 73–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Met10] Jean-Philippe Metivier. *Softening Global Constraints : Implementation & Application*. Theses, Université de Caen, April 2010.

[MP79] Holmes E. Miller and William P. Pierskalla. Nurse scheduling : a case of disaggregation in the public sector. In Larry P. Ritzman, Lee J. Krajewski, William L. Berry, Stephen H. Goodman, Stanley T. Hardy, and Lawrence D. Vitt, editors, *Disaggregation*, pages 611–621. Springer Netherlands, Dordrecht, 1979.

[MS84] A. A. Musa and U. Saxena. Scheduling Nurses Using Goal-Programming Techniques. *IIE Transactions*, 16(3) :216–221, September 1984.

[OO88] Mihoko Okada and Masahiko Okada. Prolog-based system for nursing staff scheduling implemented on a personal computer. *Computers and Biomedical Research*, 21(1) :53–63, February 1988.

[PRS+11] Renzo Pizarro, Gianni Rivera, Ricardo Soto, Broderick Crawford, Carlos Castro, and Eric Monfroy. Constraint-Based Nurse Rostering for the Valparaíso Clinic Center in Chile. In Constantine Stephanidis, editor, *HCI International 2011 – Posters' Extended Abstracts*, volume 174, pages 448–452. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[SS98] Steve Scott and Ron Simpson. Case-bases incorporating scheduling constraint dimensions - Experiences in nurse rostering -. In Jaime G. Carbonell, Jörg Seikmann, G. Goos, J. Hartmanis, J. van Leeuwen, Barry Smyth, and Pádraig Cunningham, editors, *Advances in Case-Based Reasoning*, volume 1488, pages 392–401. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. Series Title : Lecture Notes in Computer Science.

[TK82] James M. Tien and Angelica Kamiyama. On Manpower Scheduling Algorithms. *SIAM Review*, 24(3) :275–287, July 1982.

[TW76] Vandankumar M. Trivedi and D. Michael Warner. A Branch and Bound Algorithm for Optimum Allocation of Float Nurses. *Management Science*, 22(9) :972–981, May 1976.

[VH00] C Valouxis and E Housos. Hybrid optimization techniques for the workshift and rest assignment of nursing personnel. *Artificial Intelligence in Medicine*, 20(2) :155–175, October 2000.