Titre: Title:	Implémentation de la contrainte REGULAR en COMET
Auteur: Author:	Benoit Pralong
Date:	2007
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Pralong, B. (2007). Implémentation de la contrainte REGULAR en COMET [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8102/

UNIVERSITÉ DE MONTRÉAL

IMPLÉMENTATION DE LA CONTRAINTE REGULAR EN COMET

BENOIT PRALONG

DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(MATHÉMATIQUES APPLIQUÉES)

DÉCEMBRE 2007

Your file Votre référence
ISBN: 978-0-494-36932-6
Our file Notre référence
ISBN: 978-0-494-36932-6

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IMPLÉMENTATION DE LA CONTRAINTE REGULAR EN COMET

présenté par : PRALONG Benoit

en vue de l'obtention du diplôme de : Maîtrise és sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. SAVARD Gilles, Ph.D., président

M. ROUSSEAU Louis-Martin, Ph.D., membre et directeur de recherche

M. PESANT Gilles, Ph.D., membre et codirecteur de recherche

M. GALINIER Philippe, Ph.D., membre

pour Nadège.

En essayant continuellement, on finit par réussir.
Donc plus ça rate, plus on a de chances que ça marche.
Les Shadocks

REMERCIEMENTS

Je voudrais remercier toutes les personnes qui m'ont aidé de près ou de loin durant ma maîtrise. Je voudrais remercier évidement Louis-Martin Rousseau et Gilles Pesant pour leur écoute et leur aide durant la maîtrise.

Je remercie aussi Laurent Michel pour avoir répondu à chaque fois et rapidement dès que je me posais une question vis-à-vis de Comet.

Mes remerciements vont ensuite vers tous les membres de l'équipe du Quocessa, Marie-Claude, Gaëtan, Simon et Alessandro, que ce soit pour une aide matérielle, une pause café ou autre.

RÉSUMÉ

La recherche locale basée sur les contraintes se propose de résoudre des problèmes d'optimisation en combinant une modélisation héritée de la programmation par contraintes avec des heuristiques de recherche locale. Le langage de programmation COMET est un langage de programmation récent qui s'inscrit dans cette mouvance. COMET permet notamment à l'utilisateur d'implémenter des contraintes supplémentaires. Ce mémoire décrit l'implémentation de la contrainte globale REGULAR dans le langage de programmation COMET.

La contrainte globale REGULAR contraint une séquence de variables à être reconnue comme appartenant à un langage régulier, où reconnue par un automate fini déterministe. Le langage COMET permet l'utilisation de variables incrémentales - les invariants - qui tiennent à jour une relation entre une ou plusieurs variables. COMET s'assure de maintenir les invariants à jour. Ce mémoire décrit l'implémentation de la contrainte REGULAR dans le langage COMET ; et comment, à l'aide des invariants on peut efficacement évaluer le degré de violation de la contrainte, mais aussi quelques mouvements locaux simples.

La contrainte REGULAR est une contrainte que l'on va utiliser notamment dans les problèmes de création d'emploi du temps. En effet, les emplois du temps doivent souvent respecter un certain type de patron, aisément représentable via un langage régulier. À travers deux problèmes différents de création d'emplois du temps, on va pouvoir discuter des avantages mais aussi des limitations de la contrainte REGULAR dans une optique de recherche locale basée sur les contraintes.

Mots clés : Recherche Locale, Programmation par Contraintes, Recherche Locale Basée sur les Contraintes, Contrainte REGULAR, COMET, Création d'Emplois du Temps.

ABSTRACT

Constraint-based local search intents to solve optimization problems by combining a inherited constraint programming modeling with a local search heuristic. Comet is a recent programming language for constraint-based local search. Comet is an open language allowing users to implement their own constraints. In this masters thesis, we describe an implementation of REGULAR Constraint.

REGULAR is a global constraint. It ensures that a sequence of variables may be recognized as a word from a regular language, or by a finite state automaton. In COMET, invariants are specific incremental variables that maintain a relation between one or more variables. COMET ensures that all invariants maintain their relation during the use of COMET programs. This master thesis describes an implementation of REGULAR constraint in COMET, and the way we use invariants to estimate the constraint violation degree and some simple local moves.

The REGULAR constraint is generally used for scheduling problems. Generally, schedules have to respect a specific pattern, which can be modeled within a finite state automaton. Through two scheduling problems, we will discuss the benefits and the limitations of the use of REGULAR constraint in constraint-based local search.

Keywords : Local Search, Constraint Programming, Constraint-Based Local Search, REGULAR Constraint, COMET, Rotating Schedule, Scheduling problems.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES PROGRAMMES COMET

LISTE DES ALGORITHMES

INTRODUCTION

Les méthodes de programmation par contraintes permettent de résoudre exactement des problèmes de décisions en utilisant les informations données par les contraintes modélisant le problème. Les méthodes de recherche locale décrivent un ensemble d'algorithmes que l'on utilise pour obtenir rapidement une bonne solution à des problèmes d'optimisation, celles-ci n'offrent par contre aucune garantie sur l'optimalité de la solution. Les méthodes de *recherche locale basée sur les contraintes* décrivent un ensemble de méthodes de recherche locale, où l'on va utiliser la programmation par contraintes pour à la fois modéliser les problèmes, mais aussi pour guider la recherche. COMET est un langage informatique qui s'inscrit dans cette mouvance. Il est par ailleurs possible en COMET d'implémenter ses propres contraintes.

Nous allons décrire l'implémentation d'une contrainte globale dans le langage de programmation COMET ; soit la contrainte REGULAR. REGULAR impose à une séquence de variables d'être reconnue par un automate fini déterministe (ou un langage régulier). Contrairement aux méthodes de programmation par contraintes, où l'on utilise des algorithmes afin de restreindre la recherche aux seules solutions réalisables, la version REGULAR pour COMET va autoriser toutes les affectations possibles. Toutefois, on calculera pour chaque affectation différente le degré de violations de la contrainte.

Une fois la contrainte REGULAR implémentée dans le langage COMET, nous allons essayer de résoudre deux problèmes d'optimisation via des méthodes de recherche locale. Le premier problème est un problème de satisfaction de contraintes où il faut créer un emploi du temps rotatifs pour plusieurs employés ou équipes d'employés. On va proposer par la suite un algorithme de post-optimisation qui propose d'améliorer d'un point de vue ergonomique les horaires cycliques proposés. Le deuxième problème que l'on va tester porte lui sur un problème d'optimisation. Il va falloir créer un ensemble d'emplois du temps pour une journée de travail. Ces emplois du temps

doivent être utilisés pour un ou plusieurs employés, et doivent satisfaire du mieux possible une demande. Nous allons présenter quelques résultats que l'on obtient en utilisant COMET et REGULAR pour ces deux problèmes.

Le premier chapitre de ce mémoire va donc se focaliser sur une description brève des méthodes de programmation par contraintes, des méthodes de recherche locale et des méthodes de *recherche locale basée sur les contraintes*. On va aussi décrire dans ce chapitre les caractéristiques du langage de programmation COMET. On va enfin rappeler la définition d'un langage régulier et la contrainte globale REGULAR.

Le deuxième chapitre porte sur les concepts de l'implémentation de la contrainte REGULAR dans le langage COMET. Il va notamment être décrit comment il est possible d'implémenter dans le langage COMET des primitives qui pourront évaluer l'impact de mouvements simples pour la recherche locale. Nous allons ensuite dans le troisième chapitre décrire un problème de satisfaction de contraintes, ainsi qu'un algorithme de recherche tabou pour le résoudre dans le langage COMET. La description de la modélisation en COMET du problème sera évoquée.

Le quatrième et dernier chapitre s'intéressera à un problème d'optimisation. Il s'agit d'un problème de création d'emplois du temps journaliers pour plusieurs employés. Les emplois du temps que l'on crée sont évalués non plus par rapport au degré de satisfaction des contraintes, mais par rapport à la quantité de travail effectuée. On va ainsi présenter une modélisation de ce problème en COMET, ainsi qu'un algorithme tabou pour le résoudre avec le langage COMET.

CHAPITRE 1

REVUE DE LITTÉRATURE

COMET est un langage de programmation orienté objet qui vise à résoudre des problèmes d'optimisations en utilisant des méthodes de recherche locale couplées à une modélisation héritée de la programmation par contraintes. COMET offre la possibilité d'implémenter ses propres contraintes. Nous allons décrire dans le chapitre suivant comment décrire la contrainte globale REGULAR.

Pour pouvoir expliquer le principe de l'implémentation de la contrainte REGULAR en COMET, nous allons rapidement rappeler le principe des méthodes de programmations par contraintes et des méthodes de recherche locale. Nous allons ensuite décrire le langage de programmation COMET et la contrainte globale REGULAR.

1.1 Méthodes de résolutions

COMET résout des problèmes d'optimisation en combinant des algorithmes de recherche locale avec une modélisation des problèmes héritée de la programmation par contraintes. Avant de décrire le langage de programmation COMET, nous allons rappeler rapidement le principe des méthodes de programmation par contraintes, les méthodes de recherche locale, ainsi que les méthodes combinant la programmation par contraintes avec la recherche locale.

1.1.1 La Programmation par contraintes

Le but de ce paragraphe est de proposer une introduction aux méthodes de programmation par contraintes. Plus d'informations sur la programmation par contraintes sont disponibles dans (Apt 2003).

Tableau 1.1 – Comparaison entre les contraintes à domaines finis et les contraintes à domaines infinis

Contrainte sur domaine fini	Contrainte sur domaines continus
$X \geq 9, X \in \{6,7,8,9,10,11\}$	$X + Y \geq 9, X \in \mathbb{N}+, Y \in \mathbb{N}$

Les méthodes de programmation par contraintes désignent à la fois une manière de modéliser les problèmes de décision ou les problèmes d'optimisation, ainsi qu'un ensemble de techniques facilitant la résolution des dits problèmes.

1.1.1.1 Modélisation d'un problème en utilisant la programmation par contraintes

Considérons l'ensemble de m variables $X = \{x_1, x_2, \ldots x_n\}$. Chacune des variables x_i admet des valeurs dans un ensemble de définition D_i. D_i peut être fini ($D_i = \{1,2,3\}$) ou infini ($D_i = \mathbb{R}$) (voir le tableau 1.1). Une contrainte C sur les variables $X = \{x_1, x_2, \ldots, x_n\}$ peut être définie comme étant un sous-ensemble du produit cartésien des domaines de définitions de chacune des variables x_i. On a donc $C \subseteq D_1 \times D_2 \times \ldots \times D_n$. On appellera un tuple $(d_1, d_2, \ldots, d_n) \in C$ une solution de C.

Une valeur $d \in D_i$ est dite inconsistante par rapport à C s'il n'existe pas de tuple $(d_1, d_2, \ldots, d_{i-1}, d, d_{i+1}, \ldots, d_n)$ qui soit solution de C. Sinon, on dira que d est consistante.

Un problème de satisfaction de contraintes $\mathcal{P}$ (ou « *Contraint Satisfaction Problem* » , ou *CSP*) est défini par un triplet $\mathcal{P} = (X, D, C)$. $X = \{x_1, x_2, \ldots, _n\}$ est un ensemble de n variables, $D = D_1 \times D_2 \times \ldots \times D_n$ est le domaine de définition de X et $C = \{C_1, C_2, \ldots, C_m\}$ est un ensemble de m contraintes. Un tuple $d \in D$ est une solution du problème $\mathcal{P}$ si et seulement si on a $d \in C_i, \forall i \in [1..m]$

Un problème d'optimisation de contraintes $\mathcal{P}_\mathcal{O}$ (ou « Constraint Optimization

Probleme » , ou *COP*) est un problème de satisfaction de contrainte $\mathcal{P}$ auquel on aura adjoint une fonction objectif f ($f : D \longmapsto \mathbb{R}$). Le problème n'est plus de trouver un tuple d qui satisfait $\mathcal{P}$, mais un tuple $d \in C$ qui optimise la fonction f. Un problème d'optimisation va donc se décrire de la manière suivante : $\mathcal{P}_{\mathcal{O}} = (\mathcal{P}, f)$.

1.1.1.2 Des outils de résolutions

Il existe plusieurs stratégies pour la résolution des problèmes de satisfaction de contraintes et d'optimisation de contraintes. On va se restreindre ici aux problèmes sur domaines finis.

L'approche naïve L'approche la plus simple pour essayer de résoudre les problèmes de satisfaction de contrainte consiste à tester toutes les valuations possibles des variables X. Bien que *a priori* inefficace, cette approche de résolution est néanmoins la base des méthodes de résolution des contraintes.

Propagation des contraintes Les méthodes de programmations par contraintes fonctionnent en utilisant un arbre de recherche. On va à chaque noeud de l'arbre de recherche fixer une variable, et propager l'information. Par exemple, supposons que l'on ait les variables X et Y, de domaine de définition respectif $D_X = \{1, 3, 4\}$ et $D_Y = \{1, 2, 3, 4\}$ et les contraintes suivantes :

$$X \leq Y$$
$$X + Y \geq 6$$

Si on fixe la valeur de X à 3, on peut réduire le domaine de définition de Y aux valeurs suivantes $\{3, 4\}$. On propage alors l'information des contraintes.

Cohérence des domaines de définitions Reprenons l'exemple précédent, les domaines de définitions de X et de Y sont respectivement $D_X = \{1, 3, 4\}$ et $D_Y =$

$\{1,2,3,4\}$. On remarque que certaines valeurs des domaines de définitions ne sont pas consistantes. En effet, les valeurs $Y = 1, Y = 2$ ne sont pas consistantes. Il est impossible de trouver une valeur de X avec $Y = 1$ (ou $Y = 2$) dans D_X qui va satisfaire les contraintes. De même, pour $X = 1$, il n'existe pas de valeur de Y qui satisfasse les contraintes. Les domaines de définitions de X et de Y sont donc $D_X = D_Y = \{3,4\}$.

Arbre de décision Les méthodes de résolution de problème de programmation par contraintes construisent un arbre de recherche afin de chercher une solution. Chaque noeud de l'arbre est construit de la manière suivante. Une ou plusieurs variables vont être liées à des valeurs, on propage l'information en éliminant les valeurs inconsistantes des domaines de définitions des autres variables, et on continue la recherche. Les contraintes servent donc à élaguer l'arbre de recherche des valeurs qui ne sont pas consistantes. Une feuille de l'arbre de recherche correspond soit à une solution possible du problème, soit à l'ensemble vide (c'est-à-dire qu'il n'y a pas de solution possible avec les assignations précédentes). Un exemple d'arbre de décision est présenté dans la figure 1.1. Il s'agit du problème du décision $\mathcal{P}$ suivant :

$$(\mathcal{P}) \quad \begin{cases} \mathrm{X} \in \{1,2\}, Y \in \{1,2,3\}, Z \in \{1,3\} \\ (\mathrm{X} \neq Y) \wedge (Y \neq Z) \wedge (X \neq Z) \end{cases} \tag{1.1}$$

Il existe plusieurs stratégies pour parcourir un arbre de recherche (par exemple la recherche en profondeur ou la recherche en meilleur d'abord). Il n'existe par contre pas de stratégie qui soit universellement meilleure que les autres. C'est d'ailleurs un axe de recherche important dans le domaine de la programmation par contraintes.

1.1.2 Les méthodes de recherche locale

Les méthodes de recherche locale sont des algorithmes itératifs que l'on va utiliser pour résoudre des problèmes d'optimisation pour lesquels on désire obtenir rapide-

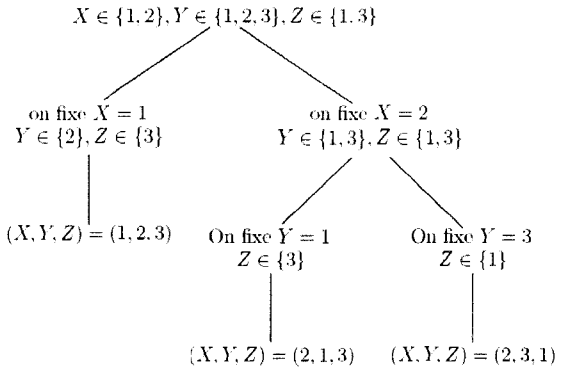

Figure 1.1 – arbre de décision pour le problème $\mathcal{P}$

ment une « bonne » solution . Les méthodes de recherche locale offrent une bonne alternatives aux méthodes exactes pour les problèmes qui sont trop grands en taille ou trop difficiles pour être résolus en un temps raisonnable.

1.1.2.1 Principe d'une méthode de recherche locale

Les méthodes de recherche locale sont des méthodes qui vont chercher, itération par itération, à améliorer la qualité d'une solution. On parle d'amélioration locale car on ne modifie que légèrement la valeur des variables d'une solution d'une itération à l'autre. Les méthodes de recherche locale associent donc à une solution s donnée (c'est-à-dire une assignation des différentes variables du problème) un voisinage $N(s)$ (c'est-à-dire un ensemble de solutions « proche » de la solution s). Le voisinage d'un point $N(s)$ n'est pas unique, il incombe à l'utilisateur de définir son propre voisinage.

Les méthodes de recherche locale fonctionnent toutes en obéissant à un même patron (voir algorithme 1.1). En premier lieu, il faut choisir une solution initiale. Puis, on va tenter à chaque itération de l'algorithme d'améliorer la valeur de l'objectif en choisissant un voisin de la solution courante. Le choix d'une solution voisine peut être fait selon plusieurs critères (meilleur voisin possible, choix aléatoire du voisin). On arrêtera la recherche quand on atteindra un critère d'arrêt, par exemple une limite sur

le nombre d'itérations de l'algorithme sans améliorer la meilleure solution trouvée, ou une limite sur le temps de recherche.

Algorithme 1.1 : Patron d'un algorithme de recherche local

```
s ← solution initiale
s* ← s ;                          // s* est la meilleure solution trouvée
tant que Le critère d'arrêt n'est pas satisfait faire
    Selectionner s' dans N(s)
    s ← s'
    si f(s) ≤ f(s*) alors
        s* ← s
    fin
fin
```

1.1.2.2 Les méta-heuristiques

Les méta-heuristiques sont une classe d'algorithmes particulière que l'on utilise pour résoudre des problèmes d'optimisation. Les méta-heuristiques sont en fait des algorithmes génériques, c'est-à-dire qu'ils ne sont pas crées pour un problème particulier, mais « adaptable à tous les problèmes ». Les méta-heuristiques sont souvent des méthodes rapides que l'on utilise pour obtenir une « bonne » valeur de la solution d'un problème. Néanmoins, à l'opposée des méthodes exactes, il n'existe pas ou peu de garanties sur la convergence des méta-heuristiques. Ce sont donc des méthodes approchées puisqu'on ne peut, en théorie, être sûr de l'optimalité de la solution trouvée. Dans ce paragraphe, on ne va traiter que des méta-heuristiques utilisant la recherche locale. D'autres méta-heuristiques n'emploient pas de manière native la recherche locale (les algorithmes génétique (Holland 1992) et les méthodes de colonies de fourmis (Dorigo and Stützle 2004) en sont des exemples).

La méthode Tabou La méthode de recherche Tabou est une classe de méta-heuristique de recherche locale (Glover 1986). Cette méta-heuristique fonctionne de la manière suivante : après chaque itération, on va interdire d'effectuer le déplacement local inverse (on interdit de « revenir sur ses pas »). On parle alors de mouvement interdit ou tabou. Les méthodes de recherche Tabou fonctionnent donc avec une mémoire à court-terme. On ne se souvient que des derniers pas que l'on a effectués, et c'est pour éviter de revenir directement sur ses pas que l'on les interdit. Les méthodes de recherche Tabou utilisent une structure de voisinage améliorée $N_T(.)$. On va définir cette structure de la manière suivante :

$$N_T(s) = \{s' | s' \in N(s), s' \notin T\} \tag{1.2}$$

T est en fait la liste des mouvements tabous. C'est une liste qui sera maintenue de manière dynamique tout au long de la recherche. Cela signifie qu'un mouvement qui est tabou à un instant t_1 de la recherche ne le sera pas forcément à un instant t_2 (mémoire à court-terme).

Algorithme 1.2 : Algorithme de recherche Tabou

s ← solution initiale
T liste Tabou
s^* ← s ; `// s* est la meilleure solution trouvée`
tant que *Le critère d'arrêt n'est pas satisfait* **faire**
  Selectionner s' dans $N_T(s)$ tel que $\forall s_2 \in N_T(s)$, on ait $f(s') \leq f(s_2)$
  $s \leftarrow s'$
  si $f(s) \leq f(s^*)$ **alors**
    $s^* \leftarrow s$
  fin
  Mettre T à jour
fin

Le principe d'une recherche tabou est relativement simple et décrit dans l'algorithme 1.2. Il s'agit en fait d'une recherche de type « descente » (c'est-à-dire une

recherche où à chaque itération on va vers le meilleur voisin possible) en utilisant la structure de voisinage $N_T(.)$ au lieu de $N(.)$. À chaque itération de l'algorithme, on mettra par contre la liste tabou T à jour.

On peut améliorer un algorithme de recherche tabou en introduisant un critère d'aspiration. On peut en effet s'autoriser à effectuer un mouvement « tabou », si et seulement si ce mouvement améliore la meilleure solution trouvée.

D'autres méta-heuristiques de recherche locale Il existe d'autres méta-heuristiques de recherche locale, comme les méthodes de recuit simulé (Kirkpatrick et al. 1983) ou les méthodes de recherche à voisinage variable (Mladenovic and Hansen 1997). Les méthodes de recuit simulé fonctionnent en choisissant un mouvement dans le voisinage de manière aléatoire. Si le mouvement améliore la solution courante, il sera effectué. S'il dégrade la solution courante, il sera effectué avec une probabilité dépendant de la dégradation et d'un paramètre de l'algorithme : la température. La température va varier avec la recherche. Plus la température sera froide, moins un mouvement dégradant la solution sera accepté. Les méthodes à voisinage variable utilisent plusieurs structures de voisinages différentes $(N_1, N_2, \ldots, N_m)$. Traditionnellement, on emploi des voisinages de plus en plus grands, c'est-à-dire que $\forall s \in S, N_1(s) \subseteq N_2(s) \subseteq \ldots \subseteq N_m(s)$. Les méthodes de recherche à voisinages variables vont chercher dans un voisinage N_i à améliorer la valeur d'une solution. Si la recherche aboutit dans N_i, alors, la recherche continue en repartant de N_1. Si la recherche n'aboutit pas, on recherche alors dans le voisinage suivant N_{i+1}.

1.1.3 Les méthodes de recherche locale basée sur les contraintes

Dans cette section, nous allons décrire quelques méthodes couplant la programmation par contraintes avec les méthodes de recherche locale. En effet, bien que de philosophies différentes, les deux méthodes n'en sont pas moins compatibles.

1.1.3.1 Méthodes couplant la programmation par contraintes et la recherche locale

Plusieurs structures couplant la programmation par contraintes et la recherche locale ont été proposées dans la littérature ces dernières années. Ainsi, par exemple, on peut utiliser les contraintes pour réduire la taille d'un voisinage dans des méthodes de recherche locale (Pesant and Gendreau 1999). La programmation par contraintes facilite alors la recherche, éliminant les voisins inintéressants pour la recherche.

Mais, sans aller jusqu'à utiliser un algorithme de programmation par contraintes pour éliminer les voisinages dans une recherche locale, on peut utiliser les contraintes dans le seul but de guider la recherche. Le rôle des contraintes n'est plus de réduire l'espace de recherche. Elles deviennent partie prenante de la fonction objectif. On évalue non seulement si la contrainte est violée, mais aussi le degré de violation associé à la contrainte.

C'est en tout cas le parti pris par Codognet et Diaz (Codognet and Diaz 2001), qui proposent ainsi, un algorithme de recherche locale fortement inspiré d'un algorithme de recherche Tabou pour résoudre des problèmes de satisfaction de contraintes.

Galinier et Hao (Galinier and Hao 2000) proposent une approche générique pour résoudre des problèmes de satisfaction de contraintes, en utilisant des méthodes de recherche locale. À chaque contrainte du problème que l'on cherche à résoudre, on associe une fonction de pénalité. Pour une assignation de variables données, et pour chaque contrainte, on va aussi définir la notion de variables critiques ; une variable sera dite critique si et seulement si il est possible, en changeant la valeur de cette variable de faire diminuer la fonction de pénalité associée à la contrainte. L'approche générique ensuite consiste à résoudre le problème de satisfaction de contraintes, comme étant la somme (pondérée ou non) de toutes les fonctions de pénalités de chaque contraintes. Enfin, une fonction de voisinage heuristique est proposée en ne considérant à chaque itération dans le voisinage que les variables critiques. Galinier et Hao

utilisent, via cette approche générique, un algorithme de recherche Tabou pour résoudre, par exemple, un problème de k-coloration de graphes (savoir si il est possible de colorier un graphe en au plus k couleurs) (Galinier and Hao 2000).

Enfin, Pascal Van Hentenryck et Laurent Michel (Michel and Hentenryck 2000), (Michel and Hentenryck 2002) proposent une bibliothèque C++ (LOCALIZER) puis un langage informatique (COMET) orienté pour la recherche locale. LOCALIZER et COMET sont deux systèmes qui utilisent néanmoins les contraintes pour guider la recherche locale. Le degré d'information offert par les contraintes atteint un niveau plus élevé que précédemment, puisque outre le degré de violation totale de la contrainte, il est possible aussi de s'enquérir de l'impact de l'assignation courante d'une variable sur le degré de violation, et des différents gains que l'on aurait si on changeait une ou plusieurs variables. De plus, COMET se propose de résoudre non plus des problèmes de satisfaction de contraintes, mais des problèmes « quelconques » d'optimisation combinatoire.

1.1.3.2 Exemple de résolution d'un problème de satisfaction de contraintes

On va montrer le fonctionnement des méthodes de recherche locale basée sur les contraintes à l'aide d'un problème de satisfaction de contraintes relativement simple. Considérons par exemple le problème de satisfaction de contraintes $\mathcal{P}_s$ suivant :

$$(\mathcal{P}_s) \quad \begin{cases} X = Y \\ X \in \{1, 2, 3\} \\ Y \in \{1, 3, 5\} \end{cases} \tag{1.3}$$

Un système de recherche locale basée sur les contraintes (comme proposé par (Codognet and Diaz 2001), (Galinier and Hao 2000) et (Michel and Hentenryck 2002)) va chercher à résoudre le problème $\mathcal{P}_s$ en minimisant la fonction $f(X, Y) = f_1(X, Y) + f_2(X, Y) + f_3(X, Y)$ où f_1, f_2, f_3 sont les trois fonctions associées au trois contraintes.

Tableau 1.2 – Résolution du problème $\mathcal{P}_s$ avec une méthode de recherche locale basée sur les contraintes

itération	opération	X	Y	d_X	d_Y	f(X,Y)
0	solution initiale	4	5	2	1	2
1	amélioration locale (X)	2	5	1	1	1
2	amélioration locale (Y)	2	1	1	1	1
3	amélioration locale (X)	1	1	0	0	0

Pour simplifier, on supposera que $f_1(X,Y)$ vaut 0 quand $X = Y$, 1 sinon. $f_2(X,Y)$ vaut 0 quand $X \in \{1,2,3\}$, 1 sinon. $f_3(X,Y)$ vaut 0 quand $Y \in \{1,3,5\}$, 1 sinon.

Le tableau 1.2 décrit, itération par itération, la résolution de ce problème à l'aide d'une méthode de recherche locale basée sur les contraintes. d_X et d_Y sont les degrés de violation associés aux contraintes. d_X (respectivement d_Y) indique combien de contraintes X (respectivement Y) viole. Initialement, la recherche propose une solution aléatoire : $(X,Y) = (4,5)$. On a $f(4,5) = 2$, mais on remarque que la variable X viole plus de contraintes que la variable Y. La recherche locale se propose donc de modifier la valeur de X en $X = 2$. À l'étape suivante de la recherche, une seule contrainte est violée. Cette fois, l'algorithme employé propose de modifier Y en $Y = 1$. La recherche se poursuit jusqu'en l'obtention d'une paire (X,Y) qui ne viole aucune contrainte.

1.2 COMET

COMET est un langage de programmation orienté objet utilisable pour résoudre des problèmes d'optimisation en utilisant des méthodes de recherche locale couplées à une modélisation du problème héritée de la programmation par contraintes. COMET est téléchargeable gratuitement sur internet sur le site officiel de COMET (site internet

officiel de COMET nd) pour les plates-formes MacOS, Linux et Windows. COMET est un langage de programmation académique jeune, fruit du travail de Laurent Michel et Pascal Van Hentenryck. Actuellement COMET est toujours en version beta, c'est-à-dire qu'il s'agit d'une version non définitive du langage.

1.2.1 La séparation entre le modèle et la recherche

COMET utilise une architecture interne qui lui permet de différencier la modélisation du problème de l'algorithme de recherche utilisé. Pour cela, la modélisation du problème est faite à partir d'objets que l'on nomme « objets différentiables » (voir paragraphe 1.2.3). Ces objets peuvent être par exemple des contraintes ou des objectifs. Les valeurs des paramètres d'une contrainte (par exemple le degré de violation) vont être gardées et mises à jour à l'aide d'invariants.

Les invariants (voir parapraphe 1.2.2) sont une spécification particulière de COMET déjà expérimentée via LOCALIZER ((Michel and Hentenryck 2000), (Hentenryck 1999)). La notion d'invariant est le coeur de l'implémentation et de l'architecture de COMET (voir figure 1.2). Ils permettent d'associer à une variable incrémentale une relation entre plusieurs variables du modèle. La valeur de l'invariant sera toujours maintenue, même si les variables associées à l'invariant évoluent. Le code Comet 1.1 décrit un invariant. Il s'agit ici de la somme des dix premiers éléments d'un tableau **tab**. L'invariant ici est **somme**. À chaque fois qu'une valeur du tableau va changer, la répercution sur **somme** se fera automatiquement.

La modélisation et la recherche sont donc les deux composantes majeures de l'architecture de COMET (Michel and Hentenryck 2002). Ces deux composantes sont indépendantes l'une de l'autre. On peut modifier l'algorithme de recherche sans avoir à modifier la modélisation de notre problème, et inversement, on peut modifier la modélisation du problème sans avoir à modifier l'algorithme de recherche. Cette indépendance entre modélisation et algorithme de recherche est un des principaux intérêts

de COMET.

```
var{int} somme(m) <- sum(i in 1..10) tab[i];
```

Comet 1.1 – Description d'un invariant : somme de 10 valeurs d'un tableau *tab*

1.2.2 Les invariants

Les invariants sont des variables incrémentales décrivant une relation sur une ou plusieurs variables. Les invariants spécifient une relation à maintenir, mais pas comment la maintenir ((Hentenryck and Michel 2005), (Michel and Hentenryck 2002) et (Michel and Hentenryck 2003)). Le chapitre 6 du livre *Constraint-Based Local Search* (Hentenryck and Michel 2005) aborde plus précisément la notion d'invariants en COMET, nous allons ici la décrire de manière succincte. Le caractère incrémentale des invariants est à l'origine de la maintenance des formules qu'ils mettent en exergue. Les invariants vont servir à la fois dans l'utilisation d'objets différentiables (voir paragraphe 1.2.3), mais aussi dans la recherche.

Il existe plusieurs manières de construire les invariants. On peut utiliser des opérateurs arithmétiques classiques comme des opérateurs unaires , binaires , des opérateurs de comparaison . COMET offre aussi la possibilité de créer des invariants sur les ensemble, en utilisant les opérateurs ensemblistes classique. Un exemple d'invariant ensembliste est décrit en Comet 1.2. On peut aussi employer divers aggrégats, la liste des principaux utilisables agrégats est présentée dans le tableau 1.3.

1.2.3 Les objets différentiables

La modélisation d'un problème en COMET nécessite par exemple l'emploi de contraintes. En effet, COMET est un langage qui va utiliser, pour résoudre des problèmes d'optimisation, l'information qu'il va tirer de contraintes à satisfaire et d'objectif à minimiser (ou à maximiser). La création de contraintes et d'objectifs en CO-

Tableau 1.3 – Aggrégats de base supportée par les invariants

opérateur	Aggrégat correspondant en COMET
+	sum(i in E) (valeurNumérique[i])
*	prod(i in E) (valeurNumérique[i])
min	min(i in E) (valeurNumérique[i])
max	max(i in E) (valeurNumérique[i])
&&	and(i in E) (valeurBooléene[i])
||	or(i in E) (valeurBooléene[i])
argmax	argmax(i in E)(valeur[i])
argmin	argmin(i in E)(valeur[i])

```
var{set{int}} ensemble(m) <- setof(i in R)(tab[i] == 0);
```

Comet 1.2 – Description d'un invariant d'ensemble

MET va se faire via les invariants. En effet, on peut interpréter une contrainte comme étant une collection d'invariants. La mise à jour incrémentale des invariants à chaque étape de la résolution de l'algorithme assure le maintien de plusieurs informations propre à la contrainte. Ces informations peuvent être le degré de violation totale de la contrainte, mais aussi la violation associée à chacune des variables de la contrainte. On pourra aussi obtenir le même type d'information pour les objectifs en COMET (voir le paragraphe 1.2.3.2). L'architecture interne de COMET (voir paragraphe 1.2) peut être sommairement décomposée en trois couches. Le noyau sera les invariants, qui permettront la création d'objets différentiables (contraintes et objectifs). Ces objets différentiables pourront être appelés lors de la résolution pour guider la recherche. Les objets différentiables sont appelés ainsi car on peut effectuer des opérations arithmétiques simples avec eux pour enrichir la modélisation du problème (voir le code Comet 1.6 pour la somme de deux objectifs).

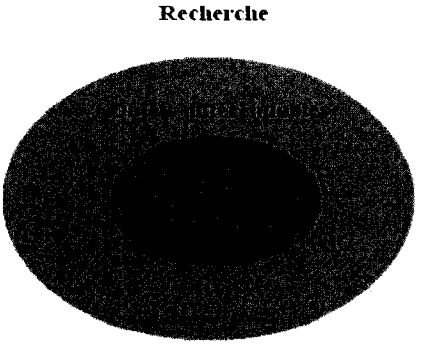

Figure 1.2 – Architecture de COMET : invariants et objets différentiables

1.2.3.1 Les contraintes

COMET s'inspire de la programmation par contraintes et propose l'utilisation de contraintes pour modéliser les problèmes que l'on souhaite résoudre. En particulier, dans chaque contrainte, deux invariants spécifiques vont être tenus à jour. Le premier indique le degré de violation de la contrainte. C'est en effet cette valeur qui permet de savoir comment la contrainte est violée, comparativement à d'autres contraintes. Un deuxième invariant spécifique aux contraintes est le degré de violation associé à une variable particulière de la contrainte. En effet, si connaître le degré de violation de la contrainte est important, connaître l'impact d'une variable sur une violation est aussi important.

Chaque contrainte en COMET dispose d'une interface qui lui est propre (voir Comet 1.3). L'interface des contraintes sera décrite plus en détail lors du chapitre suivant (voir le paragraphe 2.1.1), nous ne citerons ici que deux fonctions spécifiques :

- $getAssignDelta(var\{int\}, int)$ (ligne 6)
- $getSwapDelta(var\{int\}, var\{int\})$ (ligne 7)

```
interface Constraint  {
   var{boolean} isTrue();
   var{int} violations();
   var{int} violations(var{int});
   var{int} decrease(var{int});
   int getAssignDelta(var{int},int);
   int getSwapDelta(var{int},var{int});
}
```

Comet 1.3 – Interface partielle d'une contrainte en COMET

La première permet d'évaluer le gain que l'on aurait sur une contrainte si on changeait la valeur de la variable (premier paramètre) par la valeur passée en second paramètre. La seconde fonction permet d'évaluer le gain sur la contrainte si on échangeait les deux valeurs passées en paramètre.

COMET gère l'utilisation de contraintes simples de la forme $variable == valeur$, $variable \leq valeur$, $variable \neq valeur$, et plus généralement toutes les contraintes que l'on peut spécifier purement numériquement.

COMET permet aussi d'utiliser des contraintes globales, c'est-à-dire des contraintes qui sont dépendantes d'un ensemble de variables, comme par exemple la contrainte *alldifferent* (toute les variables associé à la contrainte *alldifferent* ne peuvent avoir de valeurs identiques). Enfin, COMET permet à l'utilisateur de programmer lui-même sa propre contrainte, l'implémentation de la contrainte *alldifferent* est par exemple décrite dans (Hentenryck and Michel 2005).

COMET permet de regrouper les contraintes entre elles, à l'aide de système de contraintes (Hentenryck and Michel 2005) (Michel and Hentenryck 2002). Les principaux systèmes de contraintes sont décrits dans le site wiki de COMET (wiki internet officiel de COMET nd), ils se différencient entre eux de la manière dont ils gèrent les contraintes. Par exemple, le système de contrainte classique **ConstraintSystem** agit comme une simple collection de contraintes, alors que le système **SatisfactionSystem** agira comme une collection de contraintes à satisfaire (on ne se préoccupe que

du nombre de contrainte violé et non du degré de violation des contraintes).

Les différent systèmes permettent de centraliser l'usage de toute les contraintes. Le code Comet 1.4 décrit l'utilisation d'un système de contrainte simple, pour un problème de Sudoku. La contrainte appelé à chaque fois est la contrainte *alldifferent*. Elle impose à un ensemble de variables d'avoir des valeurs différentes. Les lignes 5,8 et 11 présentent les contraintes que l'on « poste » au système de contraintes. Poster une contrainte au système de contraintes signifie simplement que l'on indique au système de contrainte de prendre la contrainte en compte.

```
// Système de contraintes
ConstraintSystem S(m);
// Pour chaque colonne du sudoku
forall(i in 1..N)
  S.post(alldifferent(all(k in 1..N) sudoku[k,i]));
// Pour chaque ligne du sudoku
forall(i in 1..N)
  S.post(alldifferent(all(k in 1..N) sudoku[i,k]));
// Pour chaque zone du sudoku
forall(i in 1..n, j in 1..n)
  S.post(alldifferent(
    all(k in 1..n, m in 1..n) sudoku[(i-1)*n+k,(j-1)*n+m]));
```

Comet 1.4 – Système de contrainte en COMET : exemple du Sudoku

1.2.3.2 Les objectifs

Les objectifs (Hentenryck and Michel 2005)(Michel and Hentenryck 2003)(Michel and Hentenryck 2002) sont le deuxième type d'objets différentiables principal de COMET. A l'instar des contraintes, les objectifs vont aussi maintenir des informations sur une ou plusieurs variables, à l'aide des invariants. Mais les objectifs ont un rôle différent de celui des contraintes. Les contraintes imposent à des ensembles de variables d'obéir à une propriété, les objectifs eux, ont un aspect qualitatif. Il est aussi possible pour un utilisateur de pouvoir implémenter lui-même ses propres objectifs

(voir le paragraphe 2.1.2). L'interface partielle d'un objectif est décrite dans le code Comet 1.5. L'interface est décrite complètement sur le site internet wiki de COMET (wiki internet officiel de Comet nd). L'implémentation d'un objectif est par ailleurs décrite en (Hentenryck and Michel 2005).

```
interface Objective  {
   var{int} evaluation();
   var{int} increase(var{int});
   var{int} decrease(var{int});
   int getAssignDelta(var{int},int);
   int getSwapDelta(var{int},var{int});
}
```

Comet 1.5 – Interface partielle d'un objectif en Comet

Comme pour les contraintes, les objectifs sont des objets différentiables, il est donc possible de les combiner entre eux, à l'aide d'opérateurs classique pour enrichir la modélisation en Comet (voir le code Comet 1.6). Comet supporte plusieurs opérateurs classiques. Considérons la fonction f qui renvoie l'évaluation d'un objectif. Le tableau 1.4 décrit les valeurs de f, en fonction de différentes combinaisons d'objectifs possibles.

```
// On a deux objectifs O1 et O2
Objective O1 = ...
Objective O2 = ...
// On peut combiner les objectifs entre eux
Objective SommeO = O1 + O2; // somme de deux objectifs
Objective MaxO = max(O1, O2); // maximum des deux objectifs
```

Comet 1.6 – Somme de deux objectifs en Comet Comet

Il est aussi possible d'associer une contrainte à une fonction objectif en Comet. Supposons que l'on ait un système de contraintes S, et un objectif f, il est possible de créer un nouvel objectif F tel que :

$$F = f + S$$

Tableau 1.4 – Combinaisons d'objectif

combinaison	évaluation de l'objectif
$f(o_1 + o_2)$	$f(o_1) + f(o_2)$
$f(o_1 - o_2)$	$f(o_1) - f(o_2)$
$f(\min(o_1, o_2))$	$\min(f(o_1), f(o_2))$
$f(\max(o_1, o_2))$	$\max(f(o_1), f(o_2))$
$f(\|o_1\|)$	$\|f(o_1)\|$
$f(K * o_1)$	K*f(o_1)

1.2.3.3 Les expressions de première classe

Il est possible en Comet d'exprimer des contraintes plus sophistiquées, à l'aide d'expressions de première classe (Hentenryck and Michel 2005)(Hentenryck et al. 2004). Ces expressions sont construites à partir de variables incrémentales, d'opérateurs arithmétiques et logiques. L'utilisation de ces expressions de première classe permet de définir les contraintes et les objectifs non plus à partir de simples constructions d'invariants, mais aussi à partir d'expressions plus riches. Comet 1.7 présente deux modélisations identiques pour une même contrainte en Comet. La première utilise simplement des invariants et doit spécifier un deuxième tableau d'invariants sur lequel s'appliquera la contrainte *alldifferent*. La deuxième utilisation de la contrainte utilise une expression de première classe. Ainsi, l'expression de première classe **all(i in 1..N) (position[i]+i)** (voir le code Comet 1.7) permet de modéliser de manière plus sophistiquée la contrainte.

1.2.3.4 Les invariants différentiables

Comet offre enfin un dernier niveau d'abstractions pour les contraintes et les objectifs : les invariants différentiables (Hentenryck and Michel 2006). Les invariants

```
/* Non-Utilisation d'expressions de première classe */
var{int} diagSup[i in 1..N](m) <- position[i] + i;
S.post(alldifferent(diagSup));
/* Utilisation d'expressions de première classe */
S.post(alldifferent(all(i in 1..N) (position[i]+i)));
```

Comet 1.7 – Modélisation du problèmes des reines avec des expressions de première classe

différentiables permettent d'associer des expressions incrémentales à des objets différentiables. Par exemple, la contrainte décrite en Comet 1.8 est une réécriture de la contrainte qui impose à un sudoku d'avoir sur chaque colonne une et une seule occurrence de chacune des valeurs. Toutefois la contrainte comme elle est posée mélange à la fois une expression mathématique et une expression logique. Les invariants différentiables permettent donc d'utiliser des expressions plus complexes pour les contraintes ou fonctions objectifs.

```
forall(i in 1..N, j in 1..N)
  S.post(sum(k in 1..N)(sudoku[i,j]==sudoku[i,k])<=1);
```

Comet 1.8 – Invariants différentiables pour le problème de Sudoku

1.2.4 La recherche

Comme il a été mentionné précédemment, Comet sépare la modélisation des problèmes de l'algorithme de recherche qui est employée pour les résoudre. Comet est un langage informatique utile pour la résolution de problèmes d'optimisation combinatoire utilisant des méthodes de recherche locale. De ce fait, les outils servant à la recherche en Comet ont été conçus dans cette optique.

1.2.4.1 Un système simple d'utilisation pour la recherche locale : les outils de contrôle

Si l'architecture de Comet facilite l'utilisation de méthode de recherche locale pour résoudre les problèmes d'optimisation, Comet offre en plus la possibilité d'utiliser des outils de contrôle (Hentenryck and Michel 2005)(Michel and Hentenryck 2000)(Michel and Hentenryck 2002). Ces outils de contrôle peuvent être des sélectionneurs, c'est-à-dire des opérateurs qui permettent de sélectionner un objet dans un ensemble. Comet permet aussi de sauvegarder des points précis lors de l'exécution de l'algorithme. Il est de plus possible de spécifier une séquence d'opérations à effectuer avant de mettre à jour les invariants.

Les sélectionneurs

Comet propose l'utilisation de sélectionneurs. Les sélectionneurs sont des routines qui permettent de pouvoir sélectionner un élément dans un ensemble en satisfaisant ou en optimisant certains paramètres. L'utilisation de sélectionneurs permet de pouvoir exploiter de manière simple dans la recherche les différentes structures du modèle. Par exemple, Comet 1.9 emploie deux sélectionneurs gloutons *selectMax* et *selectMin*, qui choisissent respectivement dans la variable la plus violée (*selectMax(i in 1..N, j in 1..N)(S.getViolations(sudoku[i,j]))*), et la transformation locale à effectuer pour transformer du mieux possible cette variable (*selectMin(v in 1..N)(S.getAssignDelta(sudoku[i,j], v))*).

```
while(S.violations() > 0) {
  selectMax(i in 1..N, j in 1..N)(S.getViolations(sudoku[i,j]))
    selectMin(v in 1..N)(S.getAssignDelta(sudoku[i,j], v))
      sudoku[i,j] := v;
}
```

Comet 1.9 – Résolution de Sudoku en Comet

Comet propose aussi l'emploi d'autres types de sélectionneurs, comme des sélectionneurs aléatoires (choisir un élément dans un ensemble de manière aléatoire), le choix aléatoire pouvant être équiprobable ou non.

On peut facilement présenter ici comment améliorer l'algorithme de recherche (voir Comet 1.9) , en le transformant en un algorithme de recherche tabou (voir Comet 1.10). En effet, Comet permet de paramétrer le sélectionneur. On peut alors améliorer le choix de manière simple, en ajoutant un simple critère tabou (voir ligne 5 de Comet 1.10). D'un simple algorithme de descente, on obtient un algorithme de recherche Tabou.

```
int tabou[1..N, 1..N] = -1;
int iter = 0;
while((S.violations() > 0)) {
  selectMax(i in 1..N, j in 1..N : tabou[i,j] < iter)(S.
     getViolations(sudoku[i,j]))
    selectMin(v in 1..N : v != sudoku[i,j])
      (S.getAssignDelta(sudoku[i,j])) {
        sudoku[i,j] := v;
        tabou[i,j] = iter + 7;
      }
  iter++;
}
```

Comet 1.10 – Algorithme de recherche tabou pour le Sudoku en Comet

Les Solutions

Il est possible de pouvoir, à un instant donné, faire une photographie de l'état d'une solution. L'outil de contrôle employé est la structure *Solution*. Le code informatique Comet 1.11 décrit les fonctions utilisées par la structure *Solution*.

Les simulations

Un autre outil intéressant à utiliser est un outil de simulation. On peut évaluer ainsi la valeur d'un paramètre en simulant une série d'actions. Le code Comet 1.12 décrit

```
// Faire une copie de la solution
Solution s = new Solution(m);
// Restaurer la solution copiée
s.restore();
// Faire une copie de l'état des variables de la solution s
// x est une variable entière du modèle, on récupère la valeur
   de x dans la solution s
int xRecup = x.getSnapShot(s);
```

Comet 1.11 – Utilisation des solutions en Comet

l'utilisation de l'outil de simulation *lookahead* pour simuler une simple assignation de variable et évaluer le degré de violation après assignation. Il est important de noter ici que la simulation est une opération plus lourde en temps de calcul que l'opération *getAssignDelta*.

```
// Les deux commandes suivantes sont équivalentes
selectMin(v in 1..N)(S.getAssignDelta( sudoku[i,j], v));
selectMin(v in 1..N)(
  lookahead(m, S.violations()){sudoku[i,j] := v})
```

Comet 1.12 – Simulation en Comet

1.3 La contrainte REGULAR

La contrainte Regular est une contrainte globale proposée par Gilles Pesant (Pesant 2004). Elle impose à un séquence de variables d'appartenir à un langage régulier. Pour pouvoir expliquer le fonctionnement de la contrainte Regular, nous allons dans un premier temps rappeler ce qu'est un langage régulier et un automate fini déterministe.

1.3.1 Les langages réguliers et les automates finis déterministes

Pour définir et expliquer Regular, il faut dans un premier temps rappeler ce qu'est un langage régulier. En parallèle de la définition de langage régulier, nous allons aussi définir la notion d'automate, et plus précisément la notion d'automate fini déterministe.

1.3.1.1 Les langages réguliers

Pour parler de langage, on va définir la notion d'alphabet. Un alphabet est un ensemble de caractères. Généralement, on va noter un alphabet Σ. On va appeler un ensemble de caractères, les uns à la suite des autres, un mot. Ainsi, via l'alphabet $\Sigma = \{a, b, c, d\}$, on peut former le mot **dacba**. On appelle un ensemble de mots un langage. Il existe plusieurs types de langages formels, on ne va s'intéresser qu'aux langages réguliers. Un langage est dit régulier (ou rationnel) s'il peut être représenté au moyen d'une expression rationnelle (ou régulière). Les expressions rationnelles sur un langage Σ sont décrites à l'aide des définitions récursives suivantes :

- le mot composé d'aucune lettre ϵ (ou mot vide) est une expression régulière.
- chaque élément $\alpha \in \Sigma$ est une expression régulière.
- Si α et β sont deux expressions régulières alors $\alpha\beta$ est une expression régulière (concaténation de deux expressions régulières).
- Si α et β sont deux expressions régulières alors $\alpha+\beta$ est une expression régulière (union de deux expressions régulières).
- Si α est une expression régulière, alors α^* est une expression régulière (fermeture de Kleene) ($\alpha^* = \{\epsilon, \alpha, \alpha^2, \alpha^3, ...\}$).

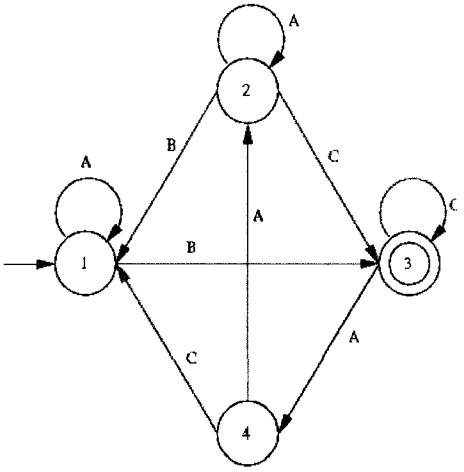

Figure 1.3 – Exemple d'automate fini déterministe

1.3.1.2 Les automates finis déterministes

Un automate est une machine à état dont le comportement est décrit par un ensemble d'états et de transitions. On peut décrire le comportement de l'automate selon le mot (c'est-à-dire un ensemble de caractères) fourni en entrée. On va alors passer d'état en état, selon la lecture de chaque lettre du mot. On dit qu'un mot est reconnu par un automate si et seulement si, à la lecture de chaque nouvelle lettre du mot, on peut se rendre en un état de l'automate et si, lors de la lecture de la dernière lettre du mot, on se rend vers un des états finaux de l'automate. De manière formelle, un automate fini déterministe M est décrit par un quintuplé $M = (\Sigma, Q, q_0, F, \delta)$ tel que :

- Σ est l'alphabet.
- Q est l'ensemble des états de l'automate.
- q_0 est l'état initial.
- F est l'ensemble des états finaux.
- δ est la fonction de transition ($\delta : Q \times \Sigma \longmapsto Q$).

La figure 1.3 représente un automate fini déterministe. On parle d'automate fini déterministe, car il y a un nombre fini d'états (contrairement à une machine de Turing

par exemple) et, partant d'un état et lisant une lettre de l'alphabet, on ne peut se rendre qu'en au plus un état (contrairement aux automates finis non déterministe).

L'ensemble des mots reconnus par un automate $\mathcal{A}$ forme un langage $\mathcal{L}_{\mathcal{A}}$. On sait, via le théorème de Kleene (Kleene 1956), que l'ensemble des langages reconnus par les automates finis déterministes sur un alphabet Σ est égal à l'ensemble des langages réguliers sur le même alphabet Σ.

1.3.2 Description de la contrainte REGULAR

La contrainte Regular (Pesant 2004) est un contrainte globale qui porte sur une séquence de variables prenant leurs valeurs dans un domaine fini. Cette contrainte impose à une séquence de variables de taille fixe d'appartenir à un langage régulier, c'est-à-dire d'être reconnue par un automate fini déterministe.

Durant cette section, on va considérer un alphabet Σ. La notation Σ^* définit l'ensemble de tous les mots pouvant être composés à partir de lettres appartenant à Σ. Plusieurs langages de programmation logique permettent de travailler avec des séquences de caractères pour pouvoir représenter des contraintes portant sur les valeurs des langages réguliers. Par exemple, le langage de programmation logique $CLP(\Sigma^*)$ (Walinsky 1989) permet de représenter des contraintes de la forme : $X \in p$. X ici ne représente qu'une seule variable. Une différence fondamentale avec la contrainte Regular vient du fait que $CLP(\Sigma^*)$ travaille sur un domaine de taille infinie : l'ensemble de toutes les combinaisons possibles de lettres de Σ. Les variables associées à la contrainte Regular ont elles un domaine fini : Σ.

La contrainte Regular est une contrainte sur domaines finis. Ici, les variables vont être chacun des éléments de la séquence de caractères (au lieu d'être la séquence de caractères (Walinsky 1989)). La contrainte Regular porte aussi sur des mots de taille fixe.

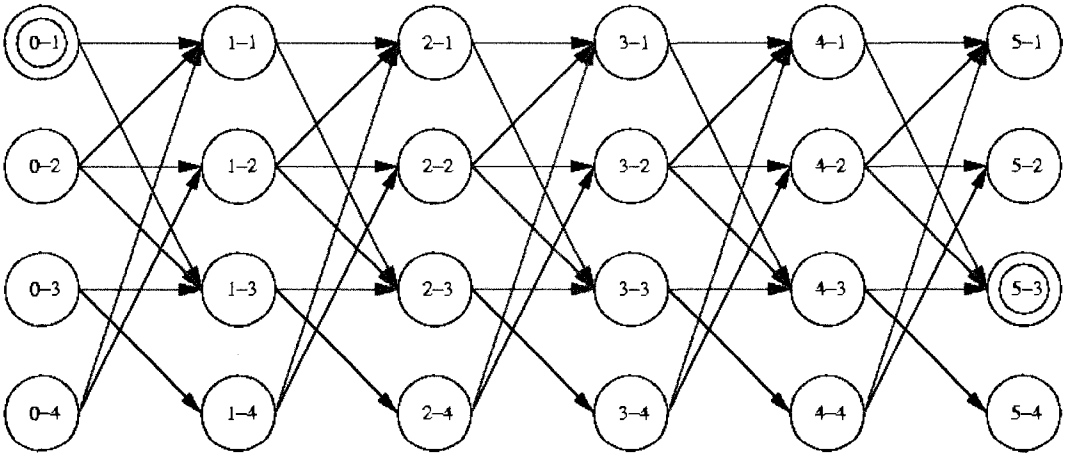

Figure 1.4 – Graphe en couche associé à l'automate fini déterministe décrit en 1.3

1.3.3 Graphe en couche et filtrage

La base de la contrainte Regular est la notion de graphe en couche G_M associé à un automate M. La figure 1.3 présente un automate fini déterministe . Cet automate fini déterministe se compose de 4 états et est décrit sur l'alphabet $\Sigma = \{a, b, c\}$. L'état 1 est l'état initial et l'état 3 est l'état final.

Considérons une séquence de n variables $X = x_1, x_2, \ldots, x_n$. Le graphe en couche G_M associé à l'automate M pour la séquence de variables X va être décrit de la manière suivante. Ce graphe est composé de $n + 1$ couches $N_0, N_1, N_2, \ldots, N_n$ de $|Q|$ sommets ($|Q|$ étant le nombre d'états de l'automate). Chacune des transitions $\delta(q_1, \alpha) = q_2$ de l'automate va être répercutée sur le graphe en couche et relie le sommet associé à l'état q_1 de l'automate pour la couche N_i à l'état associé à l'état q_2 de l'automate pour la couche N_{i+1}, pour toutes valeurs de i allant de 0 à $n - 1$.

Le graphe que l'on obtient est alors appelé graphe en couche associé $\mathcal{G}_M$ à l'automate M.

Le graphe en couche de la figure 1.4 est le graphe en couche associé à l'automate de la figure 1.3, et à un mot de 5 lettres. Pour ne pas surcharger le dessin, les étiquettes de chacune des transitions du graphe en couche ont été supprimées. De plus, pour

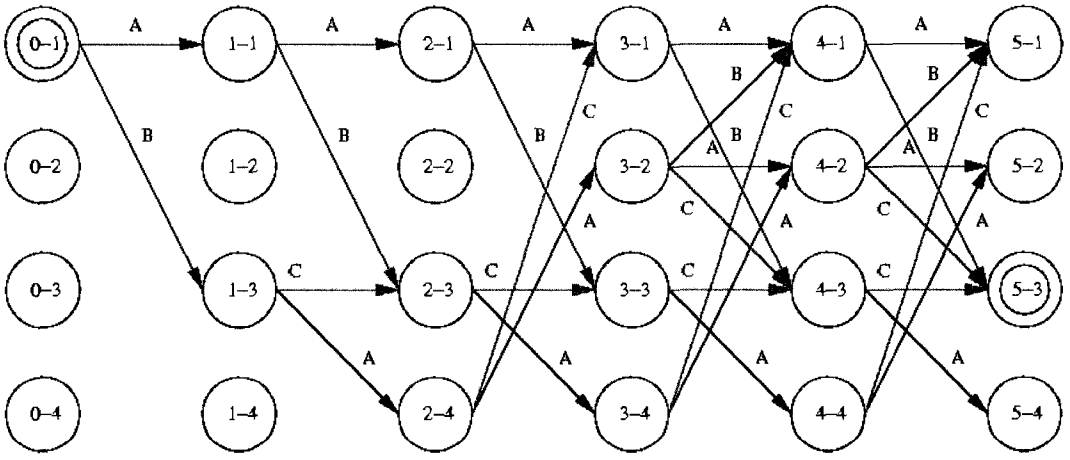

Figure 1.5 – Filtrage avant des valeurs des domaines

faciliter la lecture du graphe en couche dans ce paragraphe, l'état initial de la première couche est « doublé », ainsi que l'état final de la dernière couche.

1.3.3.1 Filtrage du graphe en couche

On se sert du graphe en couche pour représenter les différents mots reconnus par l'automate. Certains arcs deviennent donc obsolètes. Par exemple, les arcs partant d'un état i dans la couche N_0, où i n'est pas l'état initial, sont des arcs qui sont en pratique inutilisés. On se propose donc de filtrer les arcs inutiles du graphe en couche.

Le filtrage des arcs se fait en utilisant en premier lieu un filtrage « avant », c'est-à-dire un filtrage qui élimine les arcs partant de sommets inaccessibles. Il s'agit des sommets appartenant à une couche N_i tels qu'il n'existe pas de chemin partant de l'état initial de l'automate dans la couche N_0 et se rendant vers ces états en ne suivant que des transitions possibles. La figure 1.5 présente le résultat du filtrage avant sur le graphe en couche décrit en la figure 1.4. Certains arcs ont été éliminés, quelques sommets de couche ne sont plus atteignables.

Une fois le filtrage avant effectué, il faut aussi faire un filtrage « arrière ». C'est-à-dire que, de la même manière que l'on a, couche par couche, éliminé les arcs partant

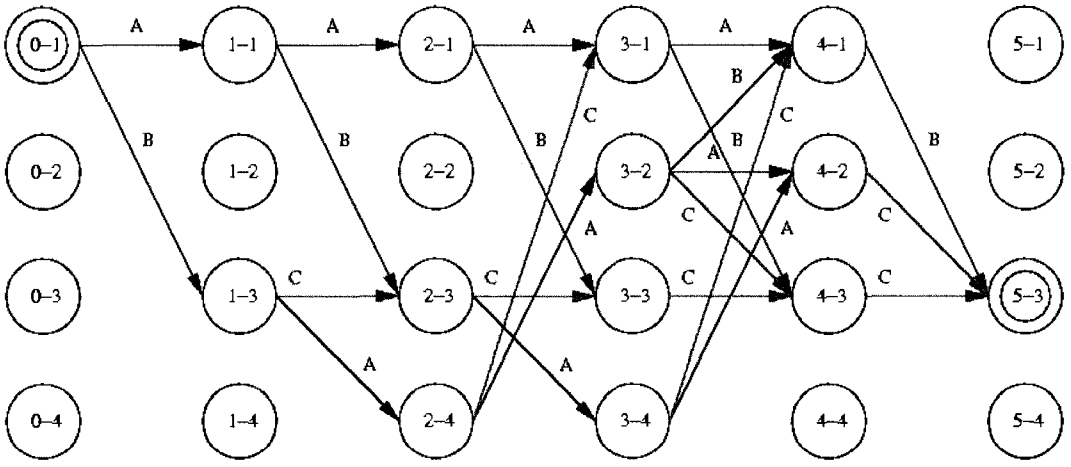

Figure 1.6 – Filtrage arrière des valeurs des domaines

d'états inaccessibles, on va aussi, couche par couche, éliminer les arcs arrivant vers des états inaccessibles. Le filtrage arrière, effectué après le filtrage avant de la figure 1.5, est présenté dans la figure 1.6. À chaque chemin partant de la couche N_0 et allant à la couche N_n est associé un mot de n lettres reconnu par le langage (ou l'automate).

Ainsi, à partir du graphe en couche, on peut déjà commencer à réduire le domaine de définition de chacune des variables. En effet, chaque transition entre deux couches N_{i-1} et N_i est associée à une valeur du domaine de la variable X_i. De ce fait, l'ensemble des valeurs pouvant être affectées à une variable X_i est donc l'ensemble des étiquettes des transitions reliant la couche N_{i-1} à la couche N_i. Ainsi, si on note D_i les domaines de définition des variables X_i, le graphe filtré 1.6 nous renseigne sur les valeurs de chacun des domaines. Les valeurs des domaines sont pour l'exemple du graphe en couche filtré de la figure 1.6 :

- $D_1 = \{a, b\}$
- $D_2 = \{a, b, c\}$
- $D_3 = \{a, b, c\}$
- $D_4 = \{a, b, c\}$
- $D_5 = \{b, c\}$

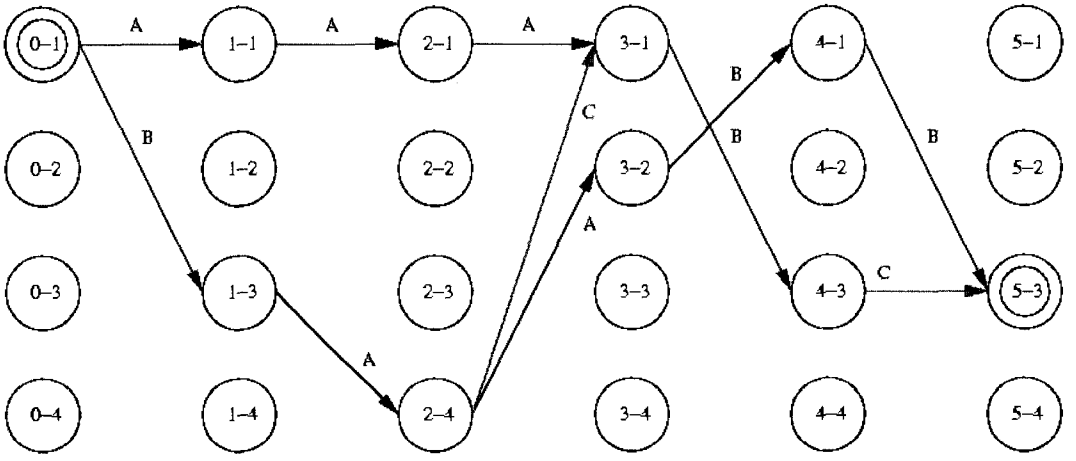

Figure 1.7 – Graphe en couche après avoir fixé la valeur b pour la variable x_4

1.3.3.2 Consistance de domaines

Si on fixe une variable, la contrainte Regular englobe un algorithme qui assure la consistance des domaines de définition des variables. Cet algorithme utilise l'algorithme de filtrage à deux phases évoqué ci-dessus. Ainsi, si par exemple on fixe la valeur de la variable X_4 à la valeur b, l'algorithme de consistance de domaine de la contrainte Regular va répercuter l'information sur les autres domaines de définition. Ainsi, si $D_4 = \{b\}$, le domaine de définition de la variable X_2 va être réduit en un singleton $D_2 = \{a\}$ (voir figure 1.7). La consistance de domaine de la contrainte Regular assure donc, que pour chaque valeur $\alpha \in D_i$, il existe au moins une séquence de variables possible $m_1 m_2 \ldots m_{i-1} \alpha m_{i+1} \ldots m_n$ ou chaque $x_j \in D_j$. L'utilisation de graphe en couche, et l'algorithme de filtrage permet d'assurer cette propriété.

1.3.4 Version souple de REGULAR

Certains problèmes de satisfaction de contraintes tirés d'exemples réels sont souvent sur-contraints, et il n'existe pas de solution faisable à ces problèmes. On peut néanmoins chercher une solution qui, à défaut de satisfaire toute les contraintes, en viole le moins possible. Pour ce type de problèmes, on ne peut utiliser la contrainte

Regular (Pesant 2004) décrite comme précédemment. De ce fait, une version souple de la contrainte Regular existe (van Hoeve et al. 2006). La version souple de Regular calcule le degré de violation associé à une séquence de variables à l'aide d'un algorithme de plus court chemin. L'algorithme de flot est exécuté sur un graphe en couche modifié $\mathcal{G}_M^*$ ressemblant au graphe en couche associé à la contrainte $\mathcal{G}_M$ auquel on aura adjoint d'autres arcs.

La figure 1.8 présente le graphe en couche modifié associé au graphe en couche de la figure 1.6. Pour ne pas surcharger la figure, seules les couches 1 et 2 sont présentées. Pour évaluer la violation d'une valuation des variables associées à la version souple de Regular, on calcule donc le plus court chemin allant de l'état initial à un des états finaux. Les arcs qui apparaissent en pointillés sur la figure 1.8 sont des arcs qui doublent une transition $\delta(q_1, \alpha) = q_2$, autorisant de passer de q_1 vers q_2, sans lire la lettre α. Ce passage n'est pas gratuit, un coût est associé à ces arcs là. Passer par un arc licite a par contre un coût nul.

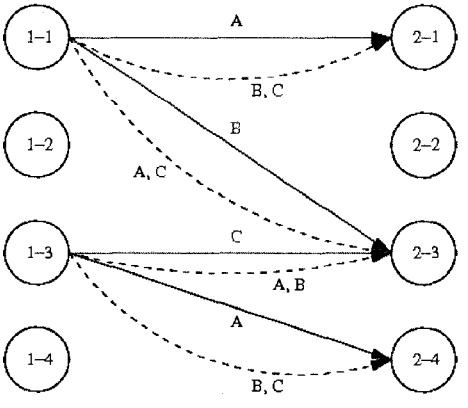

Figure 1.8 – Graphe en couche modifié

La figure 1.8 présente une manière d'obtenir un graphe modifié. Ici, le doublage des arcs permet de calculer la distance de Hamming qui sépare un mot du langage régulier. La distance d'édition qui sépare un mot d'un langage peut aussi être utilisée (van Hoeve et al. 2006). Cette version souple est à la base de l'implémentation de la contrainte Regular en Comet (voir chapitre 2).

CHAPITRE 2

UNE IMPLÉMENTATION DE REGULAR EN COMET

Ce chapitre décrit l'implémentation de la contrainte globale Regular dans le langage de programmation Comet. Comet n'utilise pas des méthodes de programmation par contraintes, mais des méthodes de recherche locale basées sur la programmation par contraintes pour résoudre des problèmes d'optimisation. L'implémentation de la contrainte Regular en Comet est fortement inspirée de l'implémentation de la version souple de la contrainte Regular (van Hoeve et al. 2006).

Durant tout ce chapitre, on va considérer un langage régulier $\mathcal{L}$, et l'automate fini déterministe qui lui est associé $\mathcal{A}_{\mathcal{L}} = (Q, \Sigma, \delta, q_0, F)$.

2.1 Implémentation de contraintes ou d'objectifs en COMET

Comet étant avant tout un langage de programmation orientée objet, chaque contrainte que l'on implémente dérive d'une classe abstraite **Constraint**. Pour pouvoir créer sa propre contrainte, il va falloir récrire les méthodes propres aux contraintes (voir un exemple d'implémentation de la contrainte globale *alldifferent* dans (Hentenryck and Michel 2005)). Il en va de même pour les fonctions objectif, qui dérivent de la classe abstraite **Objective**.

2.1.1 Description de l'interface *Constraint*

La classe abstraite **Constraint** sert de base à toutes les contraintes implémentées en Comet. La portion de code Comet 2.1 décrit l'interface partielle d'une

contrainte. Il s'agit des méthodes qu'il faudra nécessairement implémenter pour toutes les contraintes que l'on veut créer.

```
interface Constraint {
  var{int}[] getVariables();
  var{boolean} isTrue();
  var{int} violations();
  var{int} violations(var{int} x);
  var{int} decrease(var{int} x);
  int getAssignDelta(var{int} x, int v);
  int getSwapDelta(var{int} x1, var{int} x2);
}
```

Comet 2.1 – Interface partielle d'une contrainte en Comet

violations() et violations(var{int} x) Les méthodes $violations_{/0}$ et $violations_{/1}$ renvoient le degré de violation associé à la contrainte. $violations_{/0}$ indique le degré de violation totale de la contrainte, et $violations_{/1}$ indique plus précisément le degré de violation associé à la variable passée en paramètre.

decrease(var{int} x) La méthode $decrease_{/1}$ indique le gain maximum que l'on peut espérer sur le degré de violation de la contrainte si on modifie la variable passée en paramètre.

getAssignDelta(var{int} x, int v) La méthode $getAssignDelta_{/2}$ nous donne une indication sur la variation du degré de violation si on changeait la valeur de la variable x par la valeur v.

getSwapDelta(var{int} x1, var{int} x2) La méthode $getSwapDelta_{/2}$ nous donne une indication sur la variation du degré de violation si on échangeait les valeurs des deux variables x1 et x2.

2.1.2 Description de l'interface *Objective*

La classe abstraite **Objective** sert de base à toutes les fonctions objectifs implémentées en Comet. Le code Comet 2.2 décrit l'interface partielle d'une fonction objectif. Les fonctions de la classe objectif sont principalement les mêmes, à quelques exceptions près. Ainsi, la fonction **evaluation** va renvoyer la valeur de l'objectif. La fonction **increase**, absente pour les contraintes, renvoie l'augmentation maximum que l'on peut espérer sur la valeur de la fonction objectif si on modifie la variable passée en paramètre.

```
interface Objective {
  var{int}[] getVariables();
  var{int} evaluation();
  var{int} increase(var{int} x);
  var{int} decrease(var{int} x);
  int getAssignDelta(var{int} x, int v);
  int getSwapDelta(var{int} x1, var{int} x2);
}
```

Comet 2.2 – Interface partielle d'un objectif en Comet

2.2 Choix de la mesure de violation pour la contrainte REGULAR

La contrainte regular (Pesant 2004) est une contrainte globale qui impose à une chaîne de caractères d' appartenir à un langage régulier. De manière plus simple, regular contraint un ensemble de variables à être reconnu par un automate fini déterministe précis.

2.2.1 Utilisation de REGULAR en COMET

La contrainte regular impose à une chaîne de caractères d'être reconnue comme appartenant à un langage régulier donné. C'est une contrainte globale puisqu'elle touche un ensemble de variables. Néanmoins, pour s'assurer de l'efficacité de la contrainte regular en recherche locale, il faut pouvoir en définir le degré de violation. En Comet, on va autoriser toutes les assignations de variables possibles. En contrepartie, on va juger la pertinence de chacune des assignations. Il faut donc pouvoir évaluer de manière juste le degré de violation de la contrainte, c'est-à-dire avoir une mesure de distance entre le mot (assignation des variables) et le langage sur lequel on travaille. Traditionnellement, les deux mesures de distance les plus utilisées pour évaluer la distance d'un mot à un langage sont :

- La distance de Hamming (utilisée pour les codes correcteurs d'erreurs en informatique).
- La distance de Levenshtein ou distance d'édition (utilisée pour les correcteurs orthographiques).

Ces deux mesures de violations sont les deux mesures de violations qui ont été utilisées pour la version souple de la contrainte Regular(van Hoeve et al. 2006).

La distance de Hamming nous donne une indication sur le nombre minimum de réassignations à faire pour que le mot actuel appartienne au langage. La distance de Levenshtein est une mesure plus fine, elle évalue le nombre minimum d'opérations d'édition à effectuer pour que le mot appartienne au langage régulier étudié. Les opérations d'édition sont la suppression d'une lettre, l'ajout d'une lettre et le remplacement d'une lettre par une autre.

2.2.2 Description des mesures de distance

Dans cette section, on va considérer deux mots a et b, tous les deux de longueur n. Pour faciliter la compréhension, on va noter $a = a_1 a_2 \dots a_n$ et $b = b_1 b_2 \dots b_n$.

Distance de Hamming

Si on considère deux mots a et b, la distance de Hamming qui sépare ces deux mots est le nombre de caractères du mot a qu'il faudrait remplacer pour obtenir b. Ainsi, la distance de Hamming entre deux mots a et b est :

$$d_H(a,b) = \sum_{i=1}^{n} (a_i \neq b_i)$$

On peut alors étendre ainsi la mesure de Hamming à la distance entre un mot a et un langage régulier $\mathcal{L}$. On a alors la mesure suivante $d_H(a,\mathcal{L})$ comme étant la distance entre a et $\mathcal{L}$:

$$d_H(a,\mathcal{L}) = \min_{b\in\mathcal{L}}(d_H(a,b))$$

Distance de Levenshtein

La distance de Levenshtein, ou distance d'édition, est une mesure de violation plus précise que la distance de Hamming. Il s'agit du nombre minimum d'insertions, de suppressions ou de remplacements à effectuer pour changer le mot a en le mot b. Ainsi, par exemple, les mots **abbaabbaabba** et **aabbaabbaabb** auront une distance de Hamming de 6, alors qu'on remarque qu'il suffit d'enlever le dernier a du premier mot et de l'insérer en tête, pour obtenir le deuxième mot. La distance de Levenshtein entre ces deux mots est donc de 2. On va noter la distance de Levenshtein ou distance d'édition $d_E(.,.)$. On peut facilement voir que l'on aura toujours, pour deux mots a et b :

$$\forall a, \forall b, d_E(a,b) \leq d_H(a,b)$$

On peut calculer la distance d'édition entre deux mots a et b en utilisant une récurrence. Ainsi, on obtient la distance d'édition entre les mots $a_1a_2\ldots a_i$ et $b_1b_2\ldots b_j$

en appliquant la formule suivante (page wikipedia de la distance de Levenshtein nd) :

$$d_E(a_1a_2\ldots a_i,\epsilon) = i \tag{2.2}$$

$$d_E(\epsilon, b_1b_2\ldots b_j) = j \tag{2.3}$$

$$\begin{aligned} d_E(a_1a_2\ldots a_i, b_1b_2\ldots b_j) = \min(&d_E(a_1a_2\ldots a_i, b_1b_2\ldots b_{j-1}) + 1,\\ &d_E(a_1a_2\ldots a_{i-1}, b_1b_2\ldots b_j) + 1,\\ &d_E(a_1a_2\ldots a_{i-1}, b_1b_2\ldots b_{j-1}) + (a_i \neq b_j)) \end{aligned} \tag{2.2}$$

On peut ainsi proposer une formule pour décrire la distance d'édition d'un mot a à un langage $\mathcal{L}$.

$$d_E(a, \mathcal{L}) = \min_{b\in\mathcal{L}}(d_E(a, b))$$

2.2.3 Choix de la mesure de distance pour la contrainte REGULAR en COMET

Deux mesures de distance différentes peuvent donc être employées pour la contrainte Regular. Comet couple ensemble plusieurs contraintes. Les contraintes en Comet utilisent des opérateurs simples pour mesurer les gains que l'on aurait si on change la valeur d'une variable ou si on échange les valeurs de deux variables. La distance de Levenshtein mesure la distance d'édition, c'est-à-dire le nombre d'opérations d'édition (remplacement, suppression, insertion) qu'il faudrait effectuer pour obtenir un mot reconnu par un langage régulier. Cette mesure de distance est peu compatible avec les opérations disponibles initialement pour les contraintes. Un mot qui est à une distance 1 d'un langage peut l'être parce qu'il faut insérer une lettre supplémentaire au mot. Or, l'insertion et la suppression de caractères ne sont pas pris en compte lors de la recherche locale. De plus, le degré de violation associé à une variable peut apparaître comme trompeur. Une variable peut violer la contrainte parce qu'il faut la supprimer pour obtenir un mot appartenant au langage.

Comme on ne va travailler que sur des séquences de variables de taille fixe, on ne va pas utiliser la distance d'édition comme mesure de distance pour la contrainte Regular en Comet. On va employer la distance de Hamming. La distance de Hamming mesure pour un mot le nombre de caractères dont il faudrait changer la valeur pour obtenir un mot du langage. Ce type d'opérations entre en adéquation avec les opérations d'évaluation de mouvements associées aux contraintes en Comet.

2.3 Description de l'implémentation de la contrainte REGULAR en COMET

L'implémentation proposée de regular utilise donc la distance de Hamming comme degré de violation. On a expliqué précédemment comment calculer la distance de Hamming d'un mot à un autre mot. On va maintenant montrer une manière simple d'évaluer la distance de Hamming d'un mot à un langage, en utilisant la notion de graphe en couche associé au langage et à l'automate.

2.3.1 Utilisation du graphe en couche

On considère un langage régulier $\mathcal{L}$, et un automate fini déterministe $\mathcal{L}_\mathcal{G}$ lui correspondant. On a expliqué dans le chapitre précédent comment construire à partir de cet automate fini déterministe un graphe en couche qui lui est associé pour un mot de n caractères. La figure 2.1 propose un automate et le graphe en couche associé à l'automate (après filtrage des arcs inutiles). On sait, par construction du graphe en couche que pour chaque chemin allant de la source à la destination correspond un mot de n lettres reconnu par le langage $\mathcal{L}$. Ainsi, le graphe en couche associé à un automate fini déterministe pour un mot de n lettres est composé de $n+1$ couches (numérotées de 0 à n). Chaque couche est elle-même composée de $|Q|$ sommets, ($|Q|$ étant le nombre d'états de l'automate fini déterministe).

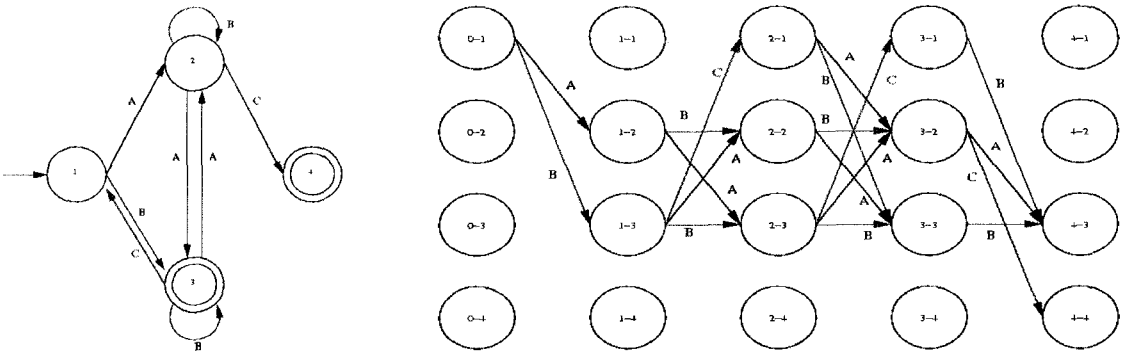

Figure 2.1 – Automate et graphe en couche qui lui est associé

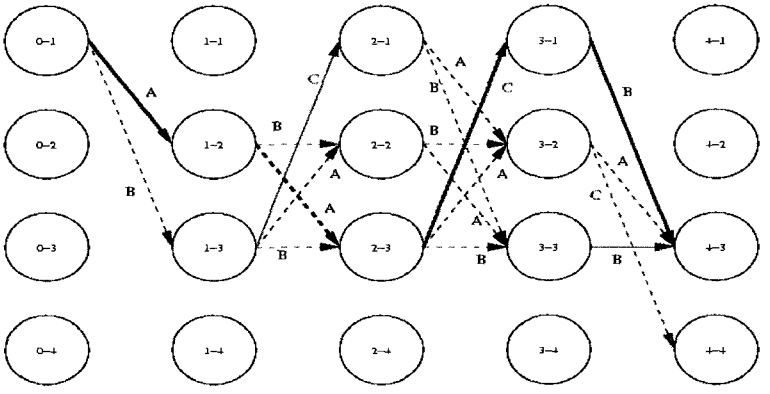

Figure 2.2 – Mesure de la distance de hamming associée au mot *accb*

Considérons le mot *accb* et l'automate décrit en 2.1. Le mot *accb* n'est pas un mot qui est reconnu par l'automate. Néanmoins, on va mesurer avec le graphe en couche associé à l'automate $\mathcal{A}_{\mathcal{L}}$ la distance séparant le mot *accb* du langage $\mathcal{L}$ associé à $\mathcal{A}_{\mathcal{L}}$. La figure 2.2 représente le graphe en couche associé à l'automate $\mathcal{A}_{\mathcal{L}}$ et au mot *accb*. Ainsi, les arcs ne correspondant pas à la lettre a entre la couche 0 et la couche 1 apparaissent en pointillés, les arcs ne correspondant pas à la lettre c entre les couches 1 et 2 apparaissent aussi en pointillés, et ainsi de suite.

Les arcs pleins sont donc les arcs qui ont un coût nul lors de la lecture du mot, les arcs pointillés eux ont un coût de 1. C'est-à-dire qu'à chaque fois qu'il faudra passer par un arc pointillé pour traverser une couche, la distance de Hamming séparant le mot du langage augmentera de 1. Le coût du chemin le plus court partant de l'état

initial de la couche 0 à un état final de la dernière couche est donc la distance de Hamming séparant le mot $m = accb$ du langage $\mathcal{L}$. Les arcs qui apparaissent plus épais sur la figure 2.2 représentent un plus court chemin. Il y a un arc pointillé dans ce chemin. La distance de Hamming séparant le mot *accb* du langage $\mathcal{L}$ est donc de 1.

Une information importante que l'on peut obtenir ici, via le graphe en couche, est « l'identité » de la variable coupable, celle qui est la cause de la violation de la contrainte. En l'occurrence, il s'agit ici de la deuxième variable, puisque il faut passer par un arc pointillé (donc de coût 1) pour passer de la couche 1 à la couche 2.

Il faut toutefois prendre en compte ici qu'on ne considère qu'un des plus courts chemins menant de l'état initial de la première couche à un des états finaux de la dernière couche. On ne considère donc que les « variables coupables » relatives au plus court chemin que l'on a choisi.

2.3.2 Principe de l'implémentation

Dans cette section, on considère un mot m de n lettres : $m = m_1 m_2 \dots m_n$, associé au langage $\mathcal{L}$ décrit par l'automate $\mathcal{A}_\mathcal{L}$ et associé au graphe en couche $\mathcal{G}$.

2.3.2.1 Retour sur les invariants

Le but de ce paragraphe est de décrire les principes de l'implémentation de la contrainte Regular en Comet. Comme on l'a dit dans le chapitre précédent, les invariants sont une spécificité de Comet (Michel and Hentenryck 2002). Les invariants sont en fait des variables incrémentales propre au langage Comet, qui vont tenir à jour une relation entre une ou plusieurs variables (voir le paragraphe 1.2.2). Les invariants de Comet sont la clé de voûte de l'implémentation de la contrainte Regular.

2.3.2.2 Description des invariants utilisés pour la contrainte Regular

On utilise pour la contrainte Regular un graphe en couche associé à l'automate de la contrainte. Néanmoins, Comet n'étant pas un langage de programmation par contraintes mais un langage de recherche locale basé sur les contraintes, on accepte toutes les configurations possibles, moyennant une pénalité. On a déjà expliqué précédemment que l'on utilise la distance de Hamming pour jauger la pertinence d'un mot vis-à-vis du langage associé à la contrainte. Un mot de distance nulle appartient au langage. Un mot de distance k signifie qu'il faut changer k lettres du mot pour obtenir un mot appartenant au langage.

Pour calculer la distance de Hamming d'un mot à un langage, pour la contrainte Regular, nous utiliserons les deux familles d'invariants suivantes :

- $\lambda^s_{i,j}$: plus court chemin du sommet source du graphe en couche $\mathcal{G}$ au sommet (i,j) du graphe en couche $\mathcal{G}$. On désignera λ^s l'ensemble des invariants $\lambda^s_{i,j}, \forall i,j$.
- $\lambda^t_{i,j}$: plus court chemin du sommet (i,j) du graphe en couche $\mathcal{G}$ au sommet destination du graphe en couche $\mathcal{G}$. On désignera λ^t l'ensemble des invariants $\lambda^t_{i,j}, \forall i,j$.

Pour calculer les valeurs des invariants λ^s et λ^t, on va utiliser les notations suivantes :

- N_i désigne l'ensemble des états de la couche i atteignables, c'est-à-dire tous les états tels qu'il existe au moins un chemin de l'état initial vers un état final passant par un de ses états.
- $C_{q_1,q_2,\alpha}$ est une fonction qui indique le coût de la transition allant de l'état q_1 à l'état q_2 en lisant le caractère α. Si on a $\delta(q_1,\alpha) = q_2$, alors $C_{q_1,q_2,\alpha} = 0$, $C_{q_1,q_2,\alpha} = 1$ s'il y a une transition entre les états q_1 et q_2, $+\infty$ sinon.

2.3.2.3 Calcul des invariants $\lambda^s_{i,j}$

Comme on l'a spécifié plus haut, les invariants $\lambda^s_{i,j}$ maintiennent la valeur des plus courts chemins menant de l'état initial de la couche 0 vers l'état j de la couche i. Ces invariants vont servir dans la modélisation de la contrainte Regular en deux moments différents. On va se servir de ces invariants pour calculer le degré de violation total du mot m vis-à-vis du langage $\mathcal{L}$. On va aussi se servir de ces invariants pour estimer la valeur d'un changement d'une variable.

On calcule les invariants $\lambda_{i,j}$ en fonction de la variable m_i et des invariants $\lambda_{(i-1),j}$, en utilisant la récurrence suivante :

$$\lambda^s_{0,q_0} = 0 \tag{2.3}$$

$$\lambda^s_{i,j} = \min_{q \in N_{i-1}} (\lambda^s_{i-1,q} + C_{q,j,m_i}), \forall j \in N_i \tag{2.4}$$

$$\lambda^s_{i,j} = +\infty, \forall j \notin N_i \tag{2.5}$$

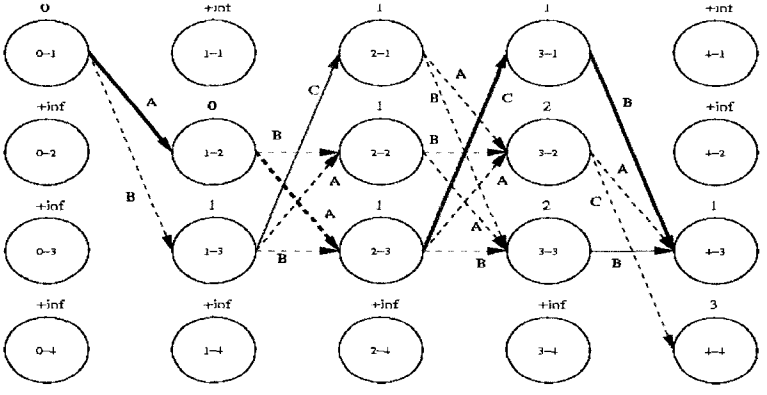

Figure 2.3 – Valeurs de $\lambda^s_{i,j}$

La figure 2.3 présente la valeur des $\lambda^s_{i,j}$ pour tous les états du graphe en couche décrit sur la figure 2.2 associé au mot *accb*.

2.3.2.4 Calcul des invariants $\lambda_{i,j}^t$

Les invariants $\lambda_{i,j}^t$ calculent le plus court chemin menant de l'état j de la couche i à un état final de la dernière couche. Ces invariants vont aussi servir à estimer le degré de violation de la contrainte Regular. Ils vont aussi servir à estimer un mouvement de recherche locale simple, le changement de valeur d'une variable. Les invariants $\lambda_{i,j}^t$ vont être calculés en fonction des invariants $\lambda_{i+1,j}^t$ et de la variable associé à la $(i+1)^{me}$ lettre du mot.

On utilise une récurrence relativement proche de la formulation précédente :

$$\lambda_{n,q}^t = 0 \forall q \in F \tag{2.6}$$

$$\lambda_{i,j}^t = \min_{q \in N_{i+1}} (\lambda_{i+1,q}^t + C_{j,q,m_{i+1}}), \forall j \in N_i \tag{2.7}$$

$$\lambda_{i,j}^t = +\infty, \forall j \notin N_i \tag{2.8}$$

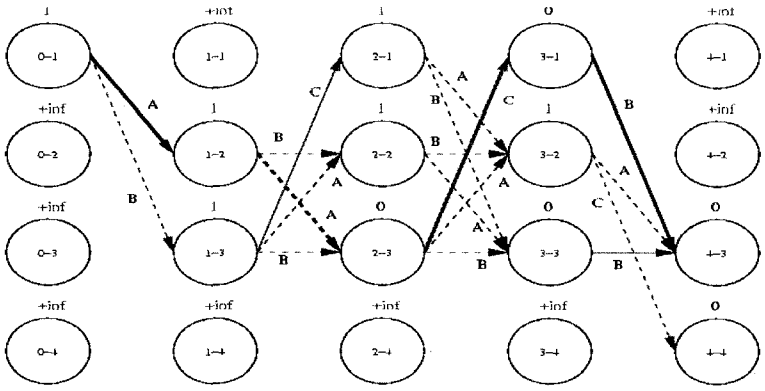

Figure 2.4 – Valeur de $\lambda_{i,j}^t$ sur le graphe en couche associé à l'automate $\mathcal{A}_{\mathcal{L}}$ et associé au mot *accb*

La figure 2.4 présente la valeur des $\lambda_{i,j}^t$ pour tous les états du graphe en couche décrit sur la figure 2.2 associé au mot *accb*.

2.3.2.5 Calcul du degré de violation associé à m_i

On va noter v_i le degré de violation de la contrainte regular associé à m_i. Il s'agit de la différence entre le plus petit coût pour se rendre à la couche i et le plus petit coût pour se rendre à la couche $i-1$. On nomme d_i le plus petit coût pour se rendre à la couche i. Alors on a :

$$d_i = \min_{q \in N_i} (\lambda^s_{i,q}) \tag{2.9}$$

$$v_i = d_i - d_{i-1} \tag{2.10}$$

On crée alors un tableau d'invariants v_i, qui maintiendra la relation décrite ci-dessus.

2.3.2.6 Calcul du degré de violation totale de la contrainte regular

On note V le degré de violation total de la contrainte regular. Alors, on peut calculer la valeur de V de quatre manières différentes :

$$V = \sum_{i=1}^{n} v_i \tag{2.11}$$

$$V = d_n - d_0 \tag{2.12}$$

$$V = d_n \tag{2.13}$$

$$V = \lambda^t_{0,q_0} \tag{2.14}$$

Ces quatre formulations sont équivalentes. Nous utiliserons pour l'implémentation de Comet la première formule. En effet, comme on maintient l'information des différents v_i (degré de violations associé à la variable m_i), on va aussi déclarer un invariant V qui sera la somme des différentes valeur v_i possibles.

2.3.2.7 Utilisation des valeurs de λ^s et λ^t pour estimer les mouvements de recherche locale simple

Comet propose pour ses deux types d'objets différentiables (contraintes et objectifs) des fonctions qui permettent d'évaluer le gain de deux types de mouvements

simples :

- Les ré-assignations de variables
- Les échanges de vàriables entre elles.

La ré-assignation d'une variable Nous expliquons comment avec l'aide des valeurs de λ^s et λ^t on peut calculer de manière simple le gain que l'on a lorsqu'on ré-assigne une variable à une valeur donnée.

Les paragraphes précédents décrivent les deux familles d'invariants que l'on utilise lors de l'implémentation de la contrainte regular en Comet. On se sert de ces invariants pour calculer le gain que l'on aurait si on changeait la valeur d'une variable. Ainsi, si on change la valeur de la i^{me} variable m_i pour la valeur α, la différence $\Delta_{i,\alpha}$ entre le nouveau degré de violation de la contrainte et l'ancien va être estimée de la manière suivante :

$$\Delta_{i,\alpha} = \min_{q_1 \in N_{i-1}} (\min_{q_2 \in N_i} (\lambda^s_{i-1,q_1} + C_{q_1,q_2,\alpha} + \lambda^t_{i,q_2})) - V$$

La valeur $\lambda^s_{i-1,q_1} + C_{q_1,q_2,\alpha} + \lambda^t_{i,q_2}$ correspond au coût du plus court chemin allant de l'état initial dans la couche 0 à un des états finaux dans la dernière couche, tel que l'on se force à passer par un arc reliant les états q_1 dans la couche $i-1$ à l'état q_2 dans la couche i en lisant le caractère α. De ce fait, par extension

$$\min_{q_2 \in N_i} (\lambda^s_{i-1,q_1} + C_{q_1,q_2,\alpha} + \lambda^t_{i,q_2})$$

calcule le plus court chemin passant par l'état q_1 dans la couche i, tel qu'on lit le caractère α entre les couches $i-1$ et i. La formule

$$\min_{q_1 \in N_{i-1}} (\min_{q_2 \in N_i} (\lambda^s_{i-1,q_1} + C_{q_1,q_2,\alpha} + \lambda^t_{i,q_2}))$$

décrit donc la valeur du plus court chemin tel qu'on lit le caractère α entre les couches $i-1$ et i.

La figure 2.5 se focalise sur l'exemple précédent. On a le graphe en couche associé à l'automate et au mot *accb*. On veut connaître la variation du degré de violation si

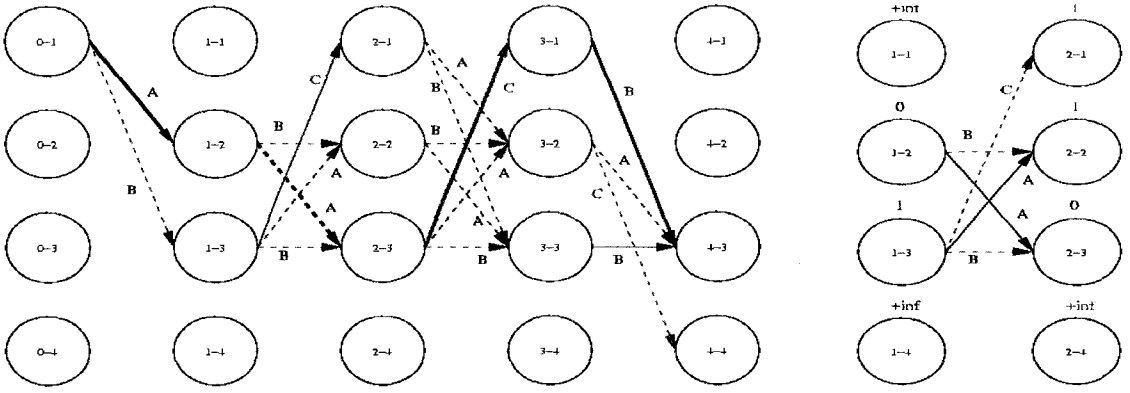

Figure 2.5 – Estimation de $\Delta_{2,a}$ pour l'exemple de la figure 2.2

on change *accb* en *aacb*, c'est-à-dire si l'on ré-assigne la deuxième variable en la valeur a. Au dessus de chaque état i de la couche 1, on donne la valeur de $\lambda^s_{1,i}$, et au sommet de chaque sommet j de la couche 2, on donne la valeur de $\lambda^t_{2,j}$. Le chemin passant de l'état 3 de la couche 1 à l'état 1 de la couche 2 aura un coût total de $\lambda^s_{1,3} + \lambda^t_{2,1} + C_{3,1,a}$ c'est-à-dire de $1 + 1 + 1 = 3$. Le chemin passant de l'état 2 de la couche 1 et allant à l'état 3 de la couche 2 aura un coût de $\lambda^s_{1,2} + \lambda^t_{2,3} + C_{2,1,a} = 0 + 0 + 0 = 0$. Le mot *accb* a un degré de violation 1, changer la valeur du premier c en un a offre un gain de $-1(= 0 - 1)$. La valeur de $\Delta_{2,a}$ est donc de -1.

L'échange de deux variables Un deuxième type de voisinage à proposer est l'échange de valeurs de deux variables entre elles. Il n'existe malheureusement pas de formulation efficace pour estimer de manière juste le gain que l'on aurait à effectuer un tel échange. En effet, si on peut estimer le gain d'une simple assignation sans être trop coûteux en terme de calcul (dans le pire des cas, $|Q|^2$ opérations pour une assignation), il n'existe pas à notre connaissance de formule peu coûteuse pour évaluer précisément un échange de variables. On se propose donc d'utiliser une évaluation optimiste qui consiste à considérer que le gain que l'on a à effectuer un échange de deux variables m_i et m_j de valeur v_i et v_j est équivalent à assigner la variable m_i avec la valeur v_j et la variable m_j avec la valeur v_i. Si on note $\Delta_{i,j}$ le gain que l'on aurait

à effectuer un tel échange, on utilise donc la relation suivante :

$$\Delta_{i,j} = \Delta_{i,v_j} + \Delta_{j,v_i}$$

2.4 Comparaison de deux implémentations de la contrainte en COMET

2.4.1 Une implémentation mettant les invariants à jour de manière « statique »

On a vu précédemment que la clé de voûte de notre implémentation de regular en Comet se base sur l'évaluation des valeurs de λ^s et λ^t. Comme on utilise un algorithme de programmation dynamique, chaque valeur de $\lambda^s_{i,j}$ va dépendre des valeurs de $\lambda^s_{i-1,k}$. On dispose de peu d'informations sur la qualité de l'algorithme qui met à jour les invariants en Comet. De ce fait, comme l'implémentation de la contrainte Regular dépend de la mise à jour des invariants λ^s et λ^t, on propose en parallèle de l'implémentation en Comet une implémentation qui met les invariants à jour de manière « statique ». La mise à jour « statique » consiste donc en un re-calcul des valeur des λ^s et λ^t à chaque itération, selon les valeurs des variables $m_1 m_2 \dots m_n$. Néanmoins, si l'on change la valeur de la i^{eme} variable, il faut prendre en compte le fait que l'on ne doit mettre à jour que les $\lambda^s_{k,j}$, tel que $k \geq i$ et les $\lambda^t_{k,j}$ tel que $k \leq i$.

2.4.2 Une implémentation dans la philosophie de COMET

La deuxième implémentation que l'on propose de la contrainte Regular se base sur l'utilisation des invariants en Comet. Les valeurs de λ^s et λ^t sont mises à jour de manière dynamique par Comet. L'utilisateur n'a donc qu'à décrire les relations décrivant chaque λ^s et λ^t, Comet s'occupe du reste.

2.4.3 Comparaison des deux implémentations

Dans cette section, nous comparons les deux implémentations « statique » et « dynamique » de la contrainte Regular. On va pour cela comparer les implémentations via deux tests différents.

2.4.3.1 Première comparaison des deux implémentations

Le premier test est relativement simple. On considère en effet un tableau de 50 variables que l'on soumet à la contrainte Regular. On mesure le temps que prend Comet pour effectuer une série de 100 assignations de variables consécutives. Ce qui est mesuré ici est donc le temps de mise à jour des invariants selon l'implémentation de la contrainte. Le tableau 2.1 donne en secondes le temps moyen sur un ensemble de 20 tests de chacune des séries de réassignation. La taille des automates va de manière croissante sur les tests, $\mathcal{A}_1$ étant le plus petit et $\mathcal{A}_6$ le plus grand.

Tableau 2.1 – Comparaison des deux implémentations : moyenne du temps en secondes de mise à jour des invariants

automate	version « dynamique »	version « statique »
$\mathcal{A}_1$	0.067	0.278
$\mathcal{A}_2$	0.090	0.443
$\mathcal{A}_3$	0.096	0.596
$\mathcal{A}_4$	0.325	1.629
$\mathcal{A}_5$	0.516	2.403
$\mathcal{A}_6$	0.669	4.126

Empiriquement, on constate que le temps moyen de mises à jour de la version « statique » est 5.5 fois celui de la version « dynamique ». La différence de temps de

calcul est ici dû au fait de la mise-à-jours des invariants. La mise-à-jour Comet est plus efficace que la mise-à-jour statique.

2.4.3.2 Deuxième comparaison des deux implémentations

Le deuxième test comparatif entre les deux implémentations porte sur la résolution d'un problème simple : obtenir à partir d'un mot quelconque un mot reconnu par l'automate. Le problème s'appuie uniquement sur la contrainte Regular.

Étant donné la structure de la contrainte Regular en Comet, le problème peut aussi se poser de la manière suivante : trouver un des mots appartenant au langage décrit par l'automate le plus proche du mot initial au sens de la distance de Hamming.

Le tableau 2.2 donne les temps moyens, médians et minimaux en secondes sur 20 tests. La version « dynamique » est toujours plus rapide que la version « statique ».

L'écart de temps entre « la version statique » et « dynamique » est toutefois moindre. La raison en est simple. En effet, dans le cadre de la recherche, deux composantes sont à prendre en compte :

- La mise-à-jour des invariants.
- L'évaluation d'un mouvement.

La mise-à-jour des invariants est ici dominée par les évaluations de mouvements que l'on fait à chaque itération. Or le temps de calcul pour évaluer un mouvement est identique selon l'implémentation. L'écart de temps est donc uniquement dû au temps de mise-à-jours des différents invariants.

Le code Comet 2.3 décrit l'algorithme Comet employé pour résoudre le problème. Il s'agit d'un simple algorithme de « descente ». À chaque itération, on choisit d'effectuer le mouvement qui diminuera le plus le degré de violation de la contrainte Regular.

Tableau 2.2 – Comparaison des deux implémentations : temps de mise à jour des invariants

	version « dynamique »			version « statique »		
automate	moyenne	médiane	min	moyenne	médiane	min
$\mathcal{A}_1$	0.107	0.092	0.076	0.114	0.096	0.076
$\mathcal{A}_2$	0.192	0.172	0.100	0.214	0.192	0.120
$\mathcal{A}_3$	0.630	0.612	0.408	0.698	0.644	0.432
$\mathcal{A}_4$	0.535	0.468	0.404	0.617	0.572	0.460
$\mathcal{A}_5$	0.920	0.848	0.728	1.269	1.216	0.844
$\mathcal{A}_6$	3.270	3.212	2.500	3.645	3.625	2.980

```
while(S.violations() > 0) {
  selectMin(q in 1..N, v in 1..m )
    (S.getAssignDelta(variable[q], v)) {
      variable[q] := v;
    }
  iter++;
}
```

Comet 2.3 – Résolution du problème d'appartenance à un langage

2.4.3.3 Conclusion

Les deux implémentations de la contrainte Regular en Comet agissent de manière identique. Les invariants, l'évaluation du degré de violation et les mouvements sont calculés de manière identique dans les deux implémentations. La seule différence entre les deux implémentations est la mise à jour des valeurs des invariants. Comet met à jour les invariants lui-même, mais on peut décider nous-mêmes de la manière de les mettre à jour. Les deux manières de mettre à jour les invariants ont été comparées. La mise à jour effectuée par Comet est plus rapide que la version statique (empiriquement environ 5.5 fois plus rapide). On a aussi comparé le temps moyen de

recherche d'une solution à un problème de satisfaction de contraintes relativement simple, cette fois encore la version Comet a été plus rapide. Néanmoins, l'écart est moindre dans ce cas. On peut toutefois conclure que la version purement Comet (mise à jour « dynamique ») est plus efficace en temps de calcul.

Dans la suite de ce mémoire, lorsque l'on se référera à la contrainte Regular, il s'agira de la version « dynamique ».

CHAPITRE 3

CRÉATION D'HORAIRES CYCLIQUES

La contrainte globale Regular impose à une séquence de caractères d'être reconnue par un langage régulier ou un automate fini déterministe. Cette contrainte s'utilise naturellement pour résoudre des problèmes de création d'emploi du temps. En effet, la création d'emploi du temps est régie par un ensemble de règles à satisfaire. Certaines sont modélisables via la contrainte Regular.

Dans le chapitre qui vient, nous allons présenter un problème de satisfaction de contraintes : la création d'emploi du temps cyclique. Ce problème peut faire intervenir dans sa modélisation la contrainte globale Regular. Nous allons présenter comment modéliser le problème en Comet ainsi qu'un algorithme de recherche locale pour le résoudre. Les résultats obtenus seront comparés avec des résultats obtenus en utilisant des méthodes de programmation par contraintes.

3.1 Le problème de création d'emplois du temps circulaires

Dans ce paragraphe, nous allons présenter le problème de création d'emplois du temps cycliques. Nous allons aussi évoquer quelques approches existantes pour le résoudre.

3.1.1 Description du problème

Certaines sociétés de service, les hôpitaux, les postes de police travaillent sept jours sur sept, vingt-quatre heures sur vingt-quatre. Les emplois du temps, élaborés

Tableau 3.1 – Exemple d'emploi du temps cyclique sur 3 semaines

semaine	L	Ma	Me	J	V	S	D
1	A	A	A	A	A	-	-
2	-	-	-	B	B	B	B
3	B	B	B	-	-	A	A

pour chaque semaine de travail, sont souvent divisés en série de quarts de travail (par exemple *journée, soir, nuit*).

Dans ce contexte, les entreprises utilisent des emplois du temps rotatifs (ou cycliques). Les quarts de travail effectués par chaque employé varient selon un calendrier déterminé. Les travailleurs doivent effectuer en alternance tous les différents quarts de travail. L'emploi du temps est dit cyclique car il va être construit pour c employés (ou équipes d'employés) et c semaines. Le premier employé devra effectuer l'emploi du temps des semaines $\{1, 2, \ldots, c\}$, le second effectuera l'emploi du temps de semaines $\{2, 3, \ldots, c, 1\}$, et ainsi de suite jusqu'au dernier employé qui effectuera l'emploi du temps des semaines $\{c, 1, 2, \ldots, c-1\}$.

Ces entreprises ont alors besoin de proposer à leurs employés des emplois du temps qui respecteront à la fois les contraintes de travail (par exemple : *un employé ne peut pas commencer une série de quarts de travail sans avoir pris au préalable au moins deux jours de repos*), mais aussi les charges de travail. On propose en effet la création d'emplois du temps cycliques (ou rotatifs), décrits sur c semaines, pour c employés (ou équipes d'employés). Par exemple, l'emploi du temps présenté dans le tableau 3.1 est un emploi du temps décrit sur 3 semaines pour 3 employés. Cet emploi du temps considère deux types de quarts de travail différents (A et B), ainsi que des jours de repos (-).

Il faut aussi s'assurer que les charges de travail soient respectées. C'est-à-dire

que l'entreprise nécessite un nombre précis d'employés devant effectuer chaque jour chaque quarts de travail. Pour l'exemple du tableau 3.1, on a chaque jour un employé effectuant le quart A et B.

3.1.2 Les différents types de contraintes existantes

Dans ce paragraphe, nous allons décrire les différentes contraintes possibles que l'on peut rencontrer dans le cadre de la résolution des problèmes d'horaires cycliques. On va utiliser la dénomination des différentes contraintes utilisées par Gilbert Laporte et Gilles Pesant (Laporte and Pesant 2004).

Les charges de travail journalières Le problème porte sur la création d'emplois du temps cycliques. Ces emplois du temps sont créées pour c employés ou c équipes d'employés, et se répètent pendant c semaines. Ainsi, après les c semaines, chaque employé aura travaillé le même nombre d'heures et effectué la même quantité de travail. Mais, l'entreprise qui utilise ces emplois du temps a des besoin précis d'employés pour chaque jour de la semaine et pour chaque type de quart de travail. Ainsi, pour chaque jour j de la semaine ($j \in \{Lundi, Mardi, \ldots, Dimanche\}$ et pour chaque quart de travail i (i pouvant être un des quarts de travail ou une journée de repos), on a une demande $d_{i,j}$ d'employés effectuant la tâche i le jour j à satisfaire.

Les contraintes de patrons Comme on l'a expliqué précédemment, les emplois du temps que l'on crée doivent obéir à plusieurs patrons. Nous allons décrire les différents types de patrons que l'on peut rencontrer.

SCP1 La classe de patrons ($SCP1$) n'autorise un changement de quarts de travail que s'il a lieu après une série de jours de repos. On va noter ce genre de patrons $S_a S_o S_b$, où S_a, S_b et S_o représentent les séries de quarts de travail des tâches

a, b et de repos o. On peut autoriser tous les quarts possibles $S_aS_oS_b$ ou seulement un sous-ensemble.

SCP2 Dans certains contextes, on va autoriser certains changements de quarts sans imposer une série de jours de repos. On a alors un patron de type S_aS_b, où S_a et S_b représentent les séries de quarts de travail des tâches a et b.

SCP3 Certains patrons du type $S_aS_oS_b$ vont spécifier un ensemble de jours de la semaine tel que les périodes de repos doivent (ou ne doivent pas, selon l'exemplaire) recouvrir. Par exemple, si une période de repos après une série de quarts de nuit inclut un samedi ou un dimanche, alors il est interdit de commencer une série de quarts du matin immédiatement après la séquence de jours de repos.

Les contraintes portant sur les tailles des séquences de jours de travail

SSC1 Dans le cadres de la contrainte $SCP1$, on va limiter les durées minimales et maximales de chacune des séries de travail S_a et des séries de repos S_o. Par exemple, on va imposer aux séries de quarts de travail de durer entre 2 et 6 jours consécutifs

SSC2 Si on autorise les patrons du type S_aS_b, alors on va aussi limiter la durée de jours consécutifs de travail. Ainsi par exemple, on pourra demander à un employé de travailler entre 4 et 6 jours consécutifs.

SSC3 Dans certains contextes, on va aussi limiter le nombre de jours de semaine identique où l'on effectue la même tâche. Par exemple, on va limiter à 3 le nombre de vendredis consécutifs où un employé sera en repos.

SSC4 Pour tous les types de quarts de travail (en comptant le repos), on va limiter le nombre de séries de quarts identiques d'une longueur donnée dans l'emploi du temps. Par exemple, on va limiter à 2 le nombre de séries de 6 jours consécutifs ou l'on effectuera le quart de matin.

SSC5 On va aussi limiter le nombre de jours minimum et maximum séparant deux longues séries de quarts de travail. Par exemple, si on autorise des séquences de 7 jours de quarts de travail consécutifs, on va interdire à ces séquences d'être consécutives (ou séparé par une séquence de jours de repos) dans l'emploi du temps.

La distribution des jours de repos et de travail

WRD1 On va limiter le nombre de jours de travail (et de repos) dans des séquences de une ou plusieurs semaines. Par exemple, pour une semaine de travail, un employé devra prendre entre 2 et 3 jours de repos.

WRD2 La contrainte $WRD2$ est une variante de la contrainte $WRD1$. On va simplement non plus traiter de séquences de une ou plusieurs semaines, mais des séquences de durées quelconques.

WRD3 On va maximiser le nombre de fins de semaine ou un employé travaillera (ou ne travaillera pas) les deux jours. Cette contrainte signifie simplement que l'on va minimiser le nombre de fins de semaine ou un jour sur les deux serait chômé.

WRD4 La contrainte $WRD4$ impose une distribution des fins de semaines complètes sur plusieurs semaines. Par exemple, chaque séquence de cinq semaines consécutives devra contenir au moins deux fins de semaines complètes de repos.

3.1.3 Approches existantes

Dans ce paragraphe, on va brièvement énumérer quelques approches existantes pour résoudre le problème d'emploi du temps cyclique.

Le problème de création d'emploi du temps cyclique (*Rotating Workforce Scheduling Problem* en anglais) est un problème qui n'est pas récent. Tien et Kamiyama (Tien and Kamiyama 1982) proposent un état de l'art des méthodes datant d'avant les années 1980. La grande majorité des méthodes décrites utilisent soit des approches énumératives exhaustives *ad-hoc*, soit des approches via des problèmes de programmation mathématiques qui ne peuvent résoudre que des versions simplifiées du problème.

La programmation par contraintes (Laporte and Pesant 2004) a été employée pour résoudre ce problème. Les contraintes permettent d'ailleurs de modéliser plus de contraintes différentes que ne le permettent les méthodes précédentes. Les heuristiques ont aussi été employées. Un algorithme génétique (Mörz and Musliu 2004) et un algorithme de recherche Tabou (Musliu 2006) se proposent donc de résoudre le problème d'horaire cyclique.

3.2 La modélisation du problème

Dans ce paragraphe, on va décrire la modélisation en Comet qui a été employée pour résoudre les problèmes d'horaires cycliques ainsi que l'algorithme de recherche locale qui a été employé.

3.2.1 Représentation des variables du problème en COMET

Un emploi du temps cyclique Le problème est la création d'un emploi du temps cyclique sur plusieurs semaines respectant plusieurs contraintes. Le tableau 3.2 est une solution possible à une des instances testées. Il s'agit ici de proposer un emploi

du temps sur 4 semaines, qui obéit entre autres au patron suivant : *Un employé doit prendre au moins un jour de repos avant de commencer un nouveau type de travail.* Il faut s'assurer que l'emploi du temps proposé sera cyclique. C'est-à-dire que passer du dimanche de la première semaine au lundi de la deuxième semaine ne brise pas les règles en vigueur. L'emploi du temps ici est dit cyclique. On peut le répéter plusieurs fois consécutives (voir le tableau 3.3), sans briser le patron.

Tableau 3.2 – Solution possible pour le problème A

semaine	L	Ma	Me	J	V	S	D
1	A	A	A	A	A	-	-
2	-	B	B	-	C	C	C
3	C	C	-	B	B	B	B
4	B	-	C	C	-	A	A

Tableau 3.3 – Emploi du temps 3.2 répété deux fois

semaine	L	Ma	Me	J	V	S	D
1	A	A	A	A	A	-	-
2	-	B	B	-	C	C	C
3	C	C	-	B	B	B	B
4	B	-	C	C	-	A	A
5	A	A	A	A	A	-	-
6	-	B	B	-	C	C	C
7	C	C	-	B	B	B	B
8	B	-	C	C	-	A	A

La représentation des variables Pour ne pas avoir à se préoccuper du caractère cyclique de l'emploi du temps, nous avons décidé d'utiliser un artifice de modélisation qui permet de passer outre cette contrainte. On recopie la première semaine à la fin de l'emploi du temps. Ainsi, on s'assure bien de la continuité entre le dernier jour de la dernière semaine et le premier jour de la première semaine.

Tableau 3.4 – Représentation des variables

semaine	L	Ma	Me	J	V	S	D
1	1L	1Ma	1Me	1J	1V	1S	1D
2	2L	2Ma	2Me	2J	2V	2S	2D
3	3L	3Ma	3Me	3J	3V	3S	3D
4	4L	4Ma	4Me	4J	4V	4S	4D
5 = 1	1L	1Ma	1Me	1J	1V	1S	1D

Ainsi, tout au long de ce chapitre, on présentera l'emploi du temps comme étant un tableau nommé *variable* de taille $(n+1)*7$ ou n est le nombre de semaines pour lesquelles l'emploi du temps doit être prévu. Chaque élément $variable_i$ du tableau correspond au quart de travail qui sera effectué le i^{eme} jour de l'emploi du temps.

3.2.2 Les contraintes

Le problème consiste à chercher un emploi du temps qui satisfait toutes les contraintes. Il y a plusieurs types de contraintes différentes que l'on peut rencontrer : nous allons les énumérer et présenter la modélisation de chacune de ces contraintes en Comet.

3.2.2.1 Le patron à respecter (SCP1 et SCP2)

Les problèmes d'horaires imposent le respect d'un patron pour l'emploi du temps. Ce patron détermine quelles sont les tâches pouvant être effectuées en fonction des

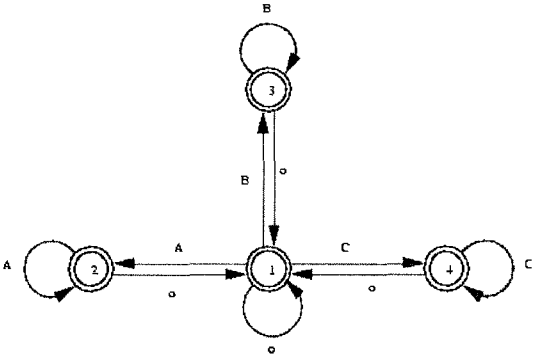

Figure 3.1 – Patron simple - 3 quarts de travails A,B et C, un quarts de repos o

tâches effectuées les jours précédents. Par exemple, une des contraintes du patron pourrait se formuler de la manière suivante : *prendre au moins deux jours de repos après une série de quarts de nuit.*

On peut modéliser le patron à l'aide d'un automate fini déterministe. En effet, un état de l'automate décrit l'état dans lequel nous sommes, (par exemple une série de quarts du matin), les transitions de l'automate vont décrire quels quarts de travail pourront être effectués le lendemain.

La figure 3.1 décrit un patron simple pour un emploi du temps à respecter. Ce patron respecte la contrainte suivante « *Avant d'entreprendre une nouvelle série de quarts de travail, il faut prendre au moins un jour de repos* ». La figure 3.2 décrit le même patron, mais en rajoutant des contraintes sur la durée de chaque série de quarts de travail (et repos). On parlera dans le premier cas de patron simple et dans le second de patron complet.

Lorsque l'on utilise un patron simple, il est quelquefois possible de se passer du patron et d'utiliser des contraintes relativement simples du type : *Si le jour j, on effectue la tâche A, le jour $j+1$ on peut effectuer la tâche A ou la tâche B.*

La portion de code Comet 3.1 présente l'utilisation en Comet de la contrainte Regular et comment la poster à un système de contraintes. Le code Comet 3.2 décrit le remplacement de la contrainte Regular associée à l'automate fini déterministe décrit

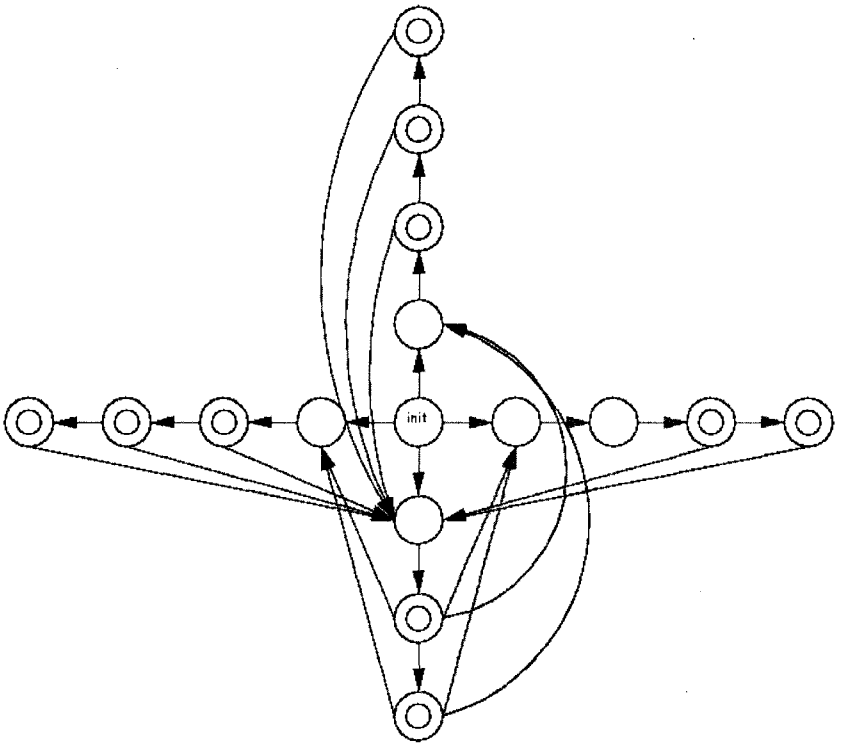

Figure 3.2 – Patron d'emploi du temps prenant en compte la durée des séries de quarts de travail

```
regularINV cstr(variable, N, fName);
S.post(cstr);
```

Comet 3.1 – Utilisation de la contrainte Regular

```
forall(i in 1..(N-1), k in 1..(nbShift-1)) {
  S.post(
     (variable[i] == k) =>
      ((variable[i+1] == k) || (variable[i+1] == repos))
  );
  S.post(
     (variable[i+1] == k) =>
      ((variable[i] == k) || (variable[i]] == repos))
  );
}
```

Comet 3.2 – Contraintes de respect du patron simple

dans la figure 3.1 à l'aide de contraintes simples. *variable* dans les deux exemples renvoie au tableau de variables qui doit respecter les patrons. $fName$ est le nom du fichier dans lequel est décrit l'automate correspondant, N est la taille du tableau de variables. $nbShift$ est le nombre de tâches différentes incluant le repos que peut effectuer un employé.

On ne représente pas la contrainte ($SCP3$). En effet, l'utilisation de cette contrainte équivaut à se passer de l'aspect global de la contrainte Regular. La contrainte ($SCP3$) impose en effet à la période de repos d'un patron $S_a S_o S_b$ autorisée de recouvrir (ou de ne pas recouvrir) un ensemble de jours précis (par exemple, le samedi et le dimanche). On perd ainsi, via cette contrainte, le caractère « global » de Regular, puisqu'on ne considère non plus seulement les patrons mais aussi les jours d'applications des différents patrons.

3.2.2.2 Le nombre de jours consécutifs pour une tâche (SSC1)

Comme on vient de le voir, le fait d'utiliser un patron simple au lieu d'un patron complet retire plusieurs informations. Il convient alors d'exprimer d'autres manières

les contraintes qui disparaissent avec le patron simple.

Dans ce paragraphe, on va présenter deux alternatives pour modéliser les contraintes qui disparaissent. La première alternative est d'utiliser un automate fini déterministe et la contrainte Regular qui lui est associée. La figure 3.3 décrit l'automate fini déterministe associé à cette contrainte. Les arcs étiquetés par la lettre a correspondent à une journée où l'on effectue la tâche associée, les arcs étiquetés par la lettre o renvoient à un changement de quarts de travail. Cet automate fini déterministe signifie ici que la tâche A doit être effectuée au moins 4 jours consécutifs et au plus 5. Il

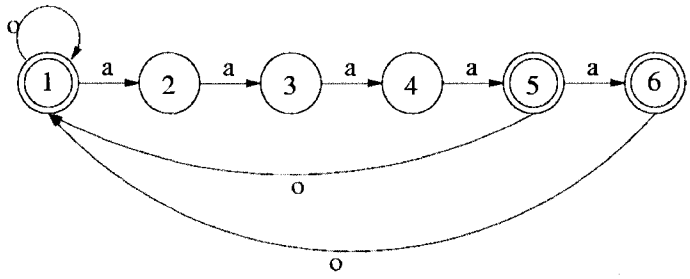

Figure 3.3 – Automate correspondant à la contrainte sur le nombre minimal et maximal de journées consécutives pour une tâche

est donc possible d'utiliser la contrainte Regular pour représenter la contrainte qui a disparu.

Mais, on peut représenter cette contrainte d'une deuxième manière, en la divisant en deux contraintes différentes. On va donc limiter :

- le nombre maximum de journées de travail consécutives à effectuer le quart A.
- le nombre minimum de journées de travail consécutives à effectuer le quart A.

Le nombre de jours consécutifs maximum pour une tâche Pour limiter le nombre de jours consécutifs maximum où un employé effectuera le quart de travail A, on va utiliser la contrainte globale *sequence*. La contrainte $sequence(var, E, p, q)$ assure que, pour chaque séquence consécutive de q valeurs du tableau de variables var, il y ait au plus p variables qui prennent une valeur contenue dans l'ensemble E.

La contrainte *sequence* existe en Comet, on peut donc l'utiliser de manière directe. En effet, dans le cas précis dans lequel nous sommes, il faut s'assurer que dans chaque ensemble de $max_A + 1$ variables consécutives (max_A étant le nombre de jours maximums consécutifs possible pour la variable A), il y ait au plus max_A occurrences de A.

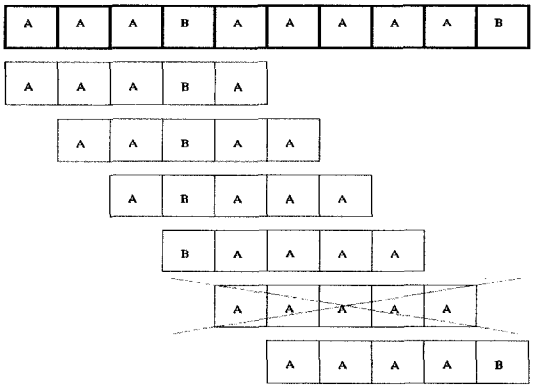

Figure 3.4 – Principe de la contrainte séquence

La figure 3.4 décrit l'idée de l'implémentation de la contrainte *sequence*. Dans cet exemple, on veut limiter le nombre de A consécutifs à 4. On regarde donc toutes les séquences de variables consécutives de taille 5. On s'aperçoit que la contrainte est violée puisque il y a une séquence de 5 A consécutifs. Le code Comet 3.3 décrit deux

```
forall(k in 1..(N-maxShiftStretch[i])){
  S.post(sum(j in 0..maxShiftStretch[i])
    (variable[k+j]==i) <= maxShiftStretch[i]);
}
// est équivalent à
S.post(sequence(variable, {i}, maxShiftStretch[i],
   maxShiftStretch[i]+1));
```

Comet 3.3 – Utilisation de la contrainte *sequence* en Comet

implémentations équivalentes de la contrainte *sequence*, la première réimplémente la contrainte séquence et s'assure juste que pour toutes les séquences de $max_A + 1$ jours consécutifs, il y ait au plus max_A jours de quarts$A$. La seconde poste la contrainte

sequence au système de contraintes. Les deux contraintes ont un fonctionnement identique en Comet. On utilise toutefois la contrainte *sequence*, afin de ne poster au système qu'une contrainte au lieu de N.De plus la contrainte *sequence* en Comet est empiriquement 5 fois plus rapide que l'implémentation directe de la contrainte.

Le nombre de jours consécutifs minimum pour une tâche La contrainte assurant le nombre de jours consécutifs minimum pour une tâche donnée ne s'exprime pas directement à l'aide d'une contrainte globale en Comet, contrairement à la contrainte précédente.

On va donc s'assurer que, pour chaque variable v ayant la valeur A, il y a une séquence de min_A variables consécutives contenant v ayant toutes la même valeur. La figure 3.5 présente cette contrainte. On veut qu'il y ait une séquence d'au moins 3 A consécutifs. On regarde pour cela les 3 séquences contenant la variable centrale. Une telle séquence existe, la contrainte n'est donc pas violée.

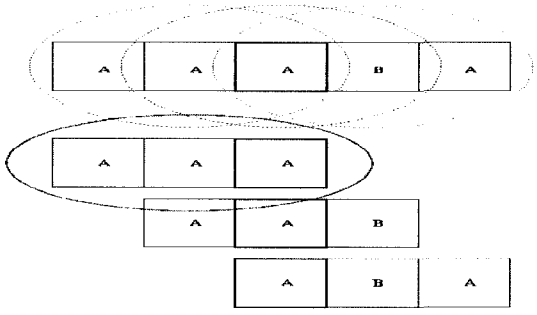

Figure 3.5 – Implémentation de la contrainte limitant le nombre minimum de jours consécutifs pour une tâche

Le code Comet 3.4 décrit l'implémentation de cette contrainte en Comet. Si la variable v associée au k^{eme} jour de l'emploi du temps prend la valeur A, on s'assure qu'il existe au moins une séquence de min_A variables contenant la variable v ne prenant que la valeur A.

```
forall(k in 1..n) {
  S.post(
    (variable[k] == A) =>
      (sum(q in 0..(minShiftStretch[A]-1))
        (
           sum(v in 0..(minShiftStretch[A]-1))(variable[k-q+v]
              == A) == minShiftStretch[A]
        ) >= 1)
  );
}
```

Comet 3.4 – Contraintes assurant le nombre minimum de jours consécutifs à effectuer une tâche donnée

3.2.2.3 Nombre minimum et maximum de journées de travail consécutives (SSC2)

Un contrainte supplémentaire que l'on rencontre porte sur le nombre de journées non chômées consécutives. Ce genre de contraintes sert donc à imposer à un travailleur de travailler au minimum un nombre m de jours consécutifs et un nombre M au maximum. Cette contrainte ne s'intéresse pas à la diversité des tâches à accomplir, mais uniquement à savoir si les tâches sont des tâches de travail ou de repos. Cette contrainte est très proche de la contrainte précédente. La manière de la décrire est donc très similaire. La seule différence provient du fait que l'on s'intéresse à tous les types de quarts de travail différents et non à un seul.

3.2.2.4 Le respect des contraintes « verticales » (SSC3)

Dans le cadre des problèmes d'horaires cycliques, on peut aussi avoir à assurer un nombre maximum de lundis consécutifs (ou n'importe quel autre jour de la semaine) où un employé effectue le même quart de travail. On va utiliser ici la contrainte globale *sequence* pour assurer contrainte. On déclare un tableau d'invariant $vertVariable$ qui va être un tableau à deux entrées grand comme deux fois l'emploi du temps. Le tableau d'invariants vertVariable va uniquement servir pour modéliser cette contrainte

en Comet. Le code Comet 3.5 décrit l'implémentation de cette contrainte en Regular.

```
// On déclare le tableau d'invariants vertVariable
var{int} vertVariable[i in 1..nbdays,
                      j in 1..2*(n/nbdays)] (m, 1..nbShift);
forall(i in 1..nbdays, j in 1..(n/nbdays)){
  vertVariable[i,j] <- variable[(j-1)*nbdays+i];
  vertVariable[i,j + (n/nbdays)] <- variable[(j-1)*nbdays+i];
}
// Contrainte verticale
forall(i in 1..nbShift, l in 1..nbdays) {
  S.post(sequence(all(k in 1..2*(n/nbdays))vertVariable[l,k],
    {i}, maxVertStretchData[i], maxVertStretchData[i]+1));
}
```

Comet 3.5 – Contraintes « verticales »

3.2.2.5 Respect des charges de travail pour chaque journée

Il est primordial, pour l'entreprise (ou le service) qui utilise les emplois du temps d'assurer que chaque jour de la semaine, il y aura un nombre suffisant d'employés qui effectueront les différentes tâches.

Le code Comet 3.6 décrit l'implémentation de cette contrainte. On utilise les contraintes *atMost* et *atLeast*. La contrainte $atMost(tab, variable)$ (respectivement $atLeast(tab, variable)$) assure qu'il y a au plus (respectivement au moins) tab_i occurrences de i dans *variable*. $workload_{i,j}$ indique combien de personnes devront effectuer la tâche j la journée i. *nPersonne* correspond au nombre de personnes (ou de semaines) touchées par l'emploi du temps.

3.2.2.6 La contrainte de fin de semaine (WRD3)

Une autre contrainte que l'on va rencontrer porte sur les fins de semaine (samedi et dimanche). Le respect des fins de semaine consiste à interdire les configurations

```
forall(i in 1..nbdays) {
  // Contraintes atmost
  S.post(
    atmost(
      all(j in 1..nbShift)(workload[i,j]),
      all(j in 1..nPersonne)(variable[(j-1)*nbdays + i])
    )
  );
  // Contraintes atleast
  S.post(
    atleast(
      all(j in 1..nbShift)(workload[i,j]),
      all(j in 1..nPersonne)(variable[(j-1)*nbdays + i])
    )
  );
}
```

Comet 3.6 – Respect des charges de travail pour chaque journée

où un jour sur les deux serait chômé. Le code Comet 3.7 décrit l'implémentation de cette contrainte en Comet.

```
forall(i in 1..nPersonne) {
  S.post(
     ((variable[(i-1)*7 + 6] == nbShift) &&
        (variable[(i-1)*7 + 7] == nbShift)) +
     ((variable[(i-1)*7 + 6] != nbShift) &&
        (variable[(i-1)*7 + 7] != nbShift)) == 1
  );
}
```

Comet 3.7 – Contrainte de fin de semaine

3.2.2.7 Respect des charges de travail par semaine (WRD1 et WRD2)

Il se peut que l'on ait à respecter les charges de travail hebdomadaires pour un employé. Un exemple serait un emploi du temps où chaque semaine, un employé aurait entre deux et trois jours de repos. Cette contrainte sert donc à s'assurer que chaque

employé effectuera dans sa semaine chaque tâche i un nombre de fois compris entre min_i et max_i. On peut exprimer cette contrainte de manière simple avec l'aide des contraintes globales $atMost$ et $atLeast$.

Le code Comet 3.8 décrit l'implémentation de cette contrainte en Comet. Les valeurs $firstWeek$ et $lastWeek$ renvoient à la première et dernière semaine de l'emploi du temps où on impose cette contrainte. Les tableaux de valeurs $maxShift$ et $minShift$ quand à eux renseignent sur le nombre maximum (respectivement minimum) de fois où l'on peut effectuer la i^{eme} tâche.

```
forall( i in firstWeek..lastWeek ) {
  S.post(atmost(maxShift,
    all(k in 1..7)(variable[(i-1)*7 + k])));
  S.post(atleast(minShift,
    all(k in 1..7)(variable[(i-1)*7 + k])));
}
```

Comet 3.8 – Respect des différentes charges de travail par semaine

Cette contrainte est modulable, on peut par exemple imposer des charges de travail sur plusieurs semaines (au lieu d'une) ou même simplement sur plusieurs jours.

3.2.3 La recherche de solution

Comet est un langage de programmation informatique orienté vers l'utilisation des méthodes de recherche locale. Pour résoudre le problème d'horaires cycliques, nous avons implémenté une méthode de recherche Tabou. Deux voisinages distincts ont été testés pour résoudre le problème.

3.2.3.1 Fonction objectif

Le problème est un problème de satisfaction de contraintes. On essaye de le résoudre à l'aide d'un algorithme de recherche locale. Pour guider la recherche, la fonction objectif que l'on cherche à minimiser est la somme des degrés de violations de

toutes les contraintes. Dès lors que la fonction objectif de notre problème aura une valeur nulle, cela signifiera que l'on aura trouvé une solution qui satisfait toutes nos contraintes. La recherche s'arrêtera alors. Puisque l'on pose toutes les contraintes à un même système de contraintes S, la commande Comet $S.violations()$ renvoie au degré de violations total du système. $S.violations()$ est donc la fonction objectif que l'on voudra minimiser tout au long de la recherche.

3.2.3.2 Les structures de voisinage

La première structure de voisinage : (N_1) Les méthodes de recherche locale, *a contrario* des méthodes exactes, peuvent accepter de violer les contraintes. Néanmoins, la première structure de voisinage utilisée ne violera en aucun cas une contrainte particulière : la contrainte assurant les charges de travail pour chaque journée différente (voir 3.2.2.5). En effet, en partant d'une solution initiale qui ne viole pas cette contrainte, le voisinage N_1 d'un emploi du temps s est l'ensemble des solutions s' telles que le passage de s à s' se fait en échangeant deux tâches effectuées le même jour (voir figure 3.6).

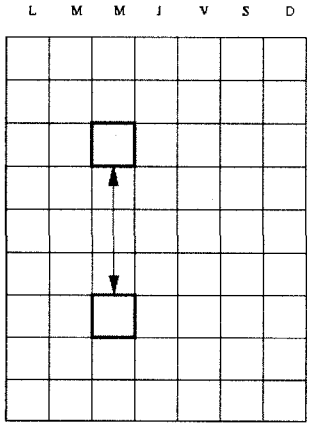

Figure 3.6 – Première structure de voisinage : N_1

Ainsi, si s ne viole pas la contrainte assurant les charges de travail pour chaque journée, aucun voisin $s' \in N_1(s)$ ne violera cette contrainte.

La deuxième structure de voisinage : (N_2) La deuxième structure de voisinage proposée est proche de la précédente, mais plus large. En effet, cette fois, on va s'autoriser de violer la contrainte portant sur les charges de travail pour chaque journée, mais on s'assurera néanmoins que la quantité de tâches différentes devant être effectuées pendant toute la durée de l'emploi du temps sera respectée. Le voisinage N_2 d'un calendrier s sera l'ensemble des calendriers s' tels que le passage de s à s' se fera en échangeant deux tâches (voir 3.7).

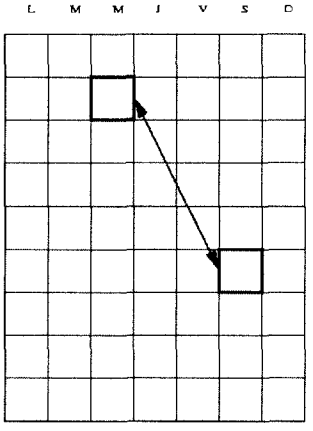

Figure 3.7 – Deuxième structure de voisinage : N_2

3.2.3.3 L'algorithme de recherche

Une fois toute les contraintes postées, on utilise un algorithme de recherche tabou (voir paragraphe 1.1.2.2) pour chercher un calendrier satisfaisant toutes les contraintes. Nous allons décrire les différentes caractéristiques de l'algorithme tabou que l'on utilise.

critère Tabou On utilise une recherche Tabou. Une recherche Tabou est en fait une recherche locale où l'on garde en mémoire les derniers mouvements effectués. On utilise, pour signifier qu'un mouvement est tabou, l'index des deux journées échangées. La liste Tabou va être en fait un tableau à deux dimensions. La valeur $tabou_{i,j}$ du

tableau *tabou* en i et en j signifie en fait à partir de quelle itération lors de la recherche on pourra reffectuer l'échange entre i et j. Le code Comet 3.9 décrit l'implémentation simple d'une méthode tabou pour le problème d'horaire cyclique avec la structure de voisinage N_2. La recherche est relativement simple, en effet, on va seulement chercher la meilleure paire de journées q et v à échanger, tel que l'échange de q et de v soit possible ($tabou[v,q] < iter$).

```
int iter = 0; // on garde en mémoire à quelle itération nous
   sommes
int tabou[1..n,1..N] = -1; // liste tabou
while ((S.violations() > 0) && ( iter<20000)) {
  iter++;
  selectMin(q in 1..n, v in 1..n : tabou[v,q] < iter)
    (S.getSwapDelta(variable[q], variable[v])) {
      variable[q] :=: variable[v];
      tabou[q,v] = iter+7;
      tabou[v,q] = tabou[q,v];
      if (q <= nbdays) variable[n+q] := variable[q];
      if (v <= nbdays) variable[n+v] := variable[v];
    }
}
```

Comet 3.9 – Recherche Tabou pour le problème d'horaire cyclique

Dans cet exemple, lorsque l'on aura effectuer l'échange entre q et v, la possibilité de refaire cet échange sera interdite pendant 7 itérations ($tabou[q,v] = iter + 7$). On s'assure aussi de reporter l'échange sur la copie de la première semaine si q ou v se trouve en première semaine. La taille de la liste tabou ici est fixée à 7 itérations.

critère d'aspiration Lors de la recherche, on peut autoriser certains mouvements tabous, à condition qu'ils améliorent la meilleure solution trouvée. On parle alors de critère d'aspiration. Utiliser un critère d'aspiration en Comet est relativement simple. Le code Comet 3.10 décrit l'implémentation d'un critère d'aspiration en Comet pour une méthode de recherche tabou. On garde en mémoire la meilleure solution trouvée (*best*) ainsi que l'écart entre la meilleure solution et la solution actuelle (*gap*). Savoir si

```
gap = best - S.violations();
selectMin(q in 1..n, v in 1..n,
  d = S.getSwapDelta(variable[q], variable[v]) :
    (tabou[v,q] < iter) || (d<gap))(d) {
      variable[q] :=: variable[v];
      tabou[q,v] = iter+7;
      tabou[v,q] = tabou[q,v];
      if (q <= nbdays) variable[n+q] := variable[q];
      if (v <= nbdays) variable[n+v] := variable[v];
    }
```

Comet 3.10 – Critère d'aspiration pour une recherche tabou

un mouvement est aspiré revient donc à comparer si le gain que l'on aurait à effectuer le mouvement serait plus petit que l'écart entre la meilleure solution et la solution actuelle.

Solution initiale Les algorithmes de recherche que l'on utilise sont des algorithmes de recherche Tabou. Pour chaque test, on va partir d'une solution initiale aléatoire. Toutefois, selon le voisinage employé (N_1 ou N_2), la solution initiale va être proposée de manière différente. En effet, si on utilise le voisinage N_1, on va créer aléatoirement une solution telle que pour chaque jour de la semaine, on respecte les demandes pour chacune des tâches. Pour le voisinage N_2, on va simplement proposer une solution aléatoire telle que l'on s'assure néanmoins de respecter la demande totale de chacun des quarts pour le calendrier.

Caractéristiques de la recherche tabou employée pour résoudre le problème d'horaire cyclique Dans ce paragraphe, nous décrirons les caractéristiques de la recherche *tabou* implémentée. La liste *tabou* n'est pas de taille fixe (contrairement à l'exemple), on va déclarer un mouvement *tabou* pendant une durée p d'itérations, p étant choisi de manière aléatoire entre 4 et 11 itérations. La recherche tabou comportera bien un critère d'aspiration. Enfin, si après plusieurs itérations la meilleure solution n'est pas trouvée, on redémarrera la recherche en partant d'une nouvelle

solution initiale.

3.3 Résultats

Durant cette section, nous allons décrire les résultats obtenus sur treize exemplaires différents. Chacun de ces exemplaires va différer par la taille des automates, la forme des différents patrons, le nombre de tâches différentes, le type de contraintes nécessaires.

Le tableau 3.5 présente les exemplaires testés ainsi que leurs caractéristiques. Quatre séries de 50 tests ont étés effectuées pour chacun de ces exemplaires, en utilisant les voisinages N_1 et N_2 et en utilisant un patron complet ou un patron simple pour la contrainte Regular. Une dernière série de tests a été effectuée pour plusieurs exemplaires tel qu'il est possible de modéliser les patrons de recherche sans utiliser la contrainte Regular.

3.3.1 Résultats utilisant un automate complet

Dans ce paragraphe, on va présenter les résultats médians que l'on obtient sur 50 tests en utilisant un automate complet (c'est-à-dire prenant en compte les contraintes $SCP1, SCP2$ et $SSC1$) et les deux structures de voisinages N_1 et N_2. Le tableau 3.6 présente les résultats obtenus sur les treize exemplaires. La première remarque que l'on peut faire porte sur la pertinence du voisinage N_2. Le voisinage N_2 est plus grand que le voisinage N_1. Or, on s'aperçoit que le nombre d'itérations médian pour chaque instance est du même ordre que l'on utilise N_1 ou N_2 (à quelques exceptions près). Le temps de calcul médian pour obtenir une solution est par contre plus long avec le voisinage N_2. En effet, quand on utilise le voisinage N_2, on visite à chaque itération plus de voisins que N_1. Il est donc normal que N_2 soit plus lent que N_1, puisque que

Tableau 3.5 – Description des différents exemplaires testés

Ref.	Nom	cycles	Quarts	Contraintes
A	Alcan 1	4	3	SCP1, SSC1, WRD3
B	Alcan 2	4	3	SCP1, SSC1, WRD3
C	Montréal Police 1	9	3	SCP1, SSC1, WRD3
D	Montréal Police 2	5	3	SCP1, SSC1, WRD3
E	Edmonton Police	9	3	SCP1, SCP2, SSC1, SSC2
F	Quebec Transportation	12	3	SCP1, SSC1, WRD3
G	Airport	10	8	SCP1, SCP2, SSC1
H	St. Louis Police	17	3	SCP1, SCP3, SSC1, SSC3
I	Non Spécifié	24	3	SCP1, SSC1, WRD1
J	Non Spécifié	12	4	SCP1, SCP2, SSC1, SSC2, WRD3
K	Non spécifié	12	3	SCP1, SSC1, WRD3
L	Non spécifié	12	3	SCP1, SSC1, WRD3
M	Non spécifié	10	3	SCP1, SSC1, WRD3

Tableau 3.6 – Résultats médians obtenus en utilisant un automate complet

	Voisinage N_1			Voisinage N_2		
exemplaire	Temps (s)	Itérations	réussite	Temps (s)	Itérations	réussite
A	0.144	26	100%	0.276	25	100%
B	0.556	126	100%	1.180	114	100%
C	2.376	246	100%	5.288	218	100%
D	0.644	114	100%	1.172	95	100%
E	2.748	188	100%	15.469	431	100%
F	0.792	60	100%	2.464	80	100%
G	1.564	60	100%	4.905	72	100%
H	64.88	2389	100%	439.984	6305	94%
I	7.616	127	100%	21.857	137	100%
J	10.721	576	100%	17.549	362	100%
K	1.132	87	100%	45.759	1402	100%
L	2.844	155	100%	9.224	199	100%
M	0.376	29	100%	0.888	32	100%

le gain que l'on pouvait espérer avoir en terme de nombre d'itérations n'est pas ou peu probant.

On peut en conclure qu'empiriquement, le voisinage N_1 est plus efficace que le voisinage N_2.

Une remarque supplémentaire porte sur l'efficacité de l'algorithme de recherche. En effet, hormis pour l'exemplaire H (St. Louis Police), on obtient 100% de réussite. Le taux d'echec de l'exemplaire H est relativement bas, et est uniquement dû à un des critères d'arrêt de l'algorithme (limite sur le nombre d'itérations de la recherche).

3.3.2 Résultats utilisant un automate simple

On peut aussi utiliser un patron simple pour résoudre les problèmes de création d'emplois du temps. On sait que le patron simple ne considère que les contraintes **SCP1** et **SCP2**. La contrainte **SSC1** est donc considérée à part. Le tableau 3.7 présente les résultats médians obtenus sur 50 tests pour les treize exemplaires en utilisant les voisinages N_1 et N_2. Le nombre médian d'itérations est sensiblement le même, que l'on utilise les voisinages N_1 ou N_2. Le temps de calcul est par contre plus rapide avec N_1, le voisinage N_1 étant plus petit que le voisinage N_2. Le taux de réussite pour chacun des exemplaires est par contre de 100%. On a trouvé une solution satisfaisant les différentes contraintes pour chaque exemplaire.

On remarque aussi que l'utilisation d'un patron simple est plus rapide en terme de temps de calcul que l'utilisation d'un patron complet. Cela vient du fait que lorsque l'on utilise un patron complet, on utilise un automate fini déterministe plus grand que l'automate fini déterministe associé à un patron simple. Or, on sait que le temps d'évaluation d'une solution et le temps de mise-à-jour des invariants dépend en grande partie de la taille des automates. Plus l'automate fini déterministe associé à la contrainte Regular est grand, plus le temps de mise-à-jour et le temps d'évaluation d'une solution est long.

3.3.3 Résultats n'utilisant pas d'automate

Comme on l'a dit dans la section précédente, on peut se passer d'automate fini déterministe dans quelques cas. Ainsi, sur les différentes instances que l'on teste, on peut se passer d'utiliser la contrainte globale Regular pour les instances A, B, C, D, F, I, K et M. Le tableau 3.8 présente les résultats médians obtenus si on utilise pas la contrainte Regular, comparé avec les résultats obtenus en utilisant un automate fini déterministe simple.

Quand il est possible de se passer de la contrainte Regular, le temps médian

Tableau 3.7 – Résultats médians obtenus en utilisant un automate simple

	Voisinage N_1			Voisinage N_2		
exemplaire	Temps (s)	Itérations	réussite	Temps (s)	Itérations	réussite
A	0.020	15	100%	0.060	17	100%
B	0.100	82	100%	0.472	159	100%
C	0.428	165	100%	1.156	169	100%
D	0.220	142	100%	0.452	116	100%
E	0.299	76	100%	0.880	110	100%
F	0.144	52	100%	0.460	61	100%
G	0.356	82	100%	1.012	68	100%
H	5.313	810	100%	30.990	1770	100%
I	1.784	129	100%	5.472	128	100%
J	4.732	828	100%	15.245	878	100%
K	1.712	610	100%	2.888	376	100%
L	0.392	98	100%	1.745	161	100%
M	0.060	27	100%	0.176	29	100%

Tableau 3.8 – Comparaison des résultats médians obtenus en utilisant Regular et en n'utilisant pas Regular pour le voisinage N_1

	Patron simple			Pas de patron		
exemplaire	Temps (s)	Itérations	réussite	Temps (s)	Itérations	réussite
A	0.200	15	100%	0.024	27	100%
B	0.100	82	100%	0.104	128	100%
C	0.428	165	100%	0.256	140	100%
D	0.220	142	100%	0.144	111	100%
F	0.144	52	100%	0.140	77	100%
I	1.784	129	100%	0.735	118	100%
K	1.712	610	100%	0.573	294	100%
M	0.060	27	100%	0.036	26	100%

de calcul est plus rapide pour trouver une solution. Cela s'explique en partie parce que on remplace une contrainte globale, avec une structure relativement importante, par un ensemble de plusieurs contraintes binaires. Dans notre exemple, pour un mot de longueur n, on remplace une contrainte par $\mathcal{O}(n)$ contraintes binaires. Les $\mathcal{O}(n)$ contraintes binaires permettent ici d'obtenir une solution plus rapidement (voir le tableau 3.8), pour un nombre d'itérations équivalent. Néanmoins, comme on l'a dit précédemment, on ne peut pas tout le temps se passer de la contrainte Regular pour modéliser les patrons. Les bons résultats que l'on obtient ici en se passant de la contrainte Regular s'explique en partie à cause de la simplicité du patron, mais aussi à cause des différentes structures de données inhérentes à la contrainte Regular qu'il faut tenir à jour et dont les contraintes binaires se passent.

3.3.4 Comparaison avec la programmation par contraintes

Dans le paragraphe qui suit, nous allons comparer les résultats obtenus avec Comet contre les résultats que l'on obtient avec une méthode de programmation par contraintes. Le programme de programmation par contraintes est celui présenté par Gilbert Laporte et Gilles Pesant (Laporte and Pesant 2004). Ce programme n'utilise pas la contrainte Regular, mais d'autres contraintes.

Le tableau 3.9 présente les résultats comparés obtenus en utilisant le programme de programmation par contrainte et le programme Comet utilisant le voisinage N_1 et la contrainte Regular associé à un automate simple. Les tests comparatifs sont exécutés sur la même machine. La première que l'on va faire porte sur la très nette dominance d'un point de vue qualitatif de la programmation par contraintes. En effet, sur les treize exemplaires testés, l'algorithme de programmation par contrainte est toujours plus efficace.

On va toutefois nuancer la qualité des résultats en faisant l'objection suivante. On utilise en effet une méta-heuristique de recherche locale générique (un algorithme de recherche Tabou) pour trouver une solution à un problème de satisfaction de contraintes. On oppose à un algorithme de recherche tabou relativement générique un algorithme robuste de programmation par contraintes, construit dans l'objectif de résoudre le problème étudié.

3.4 Équilibrage des fins de semaines

L'algorithme de recherche tabou que l'on propose cherche un horaire cyclique qui satisfait toutes les contraintes. Toutefois, il apparaît que les horaires que l'on propose ne sont pas tous ergonomiques pour les employés. Ainsi, on s'aperçoit que les horaires que l'on trouve ont tendance à regrouper les fins de semaine entre elles. Le tableau 3.10 présente deux horaires satisfaisant les contraintes de l'exemplaire F. Le second

Tableau 3.9 – Comparaison entre Comet et la programmation par contraintes

	Patron simple				$\mathcal{CP}$	
	Valeur médiane		Valeur minimale			
exemplaire	temps(s)	itération	temps	itérations	temps	retour arrière
A	0.020	15	0.008	6	0.01	0
B	0.100	82	0.012	10	0.02	37
C	0.428	165	0.060	22	0.01	0
D	0.220	142	0.032	20	0.01	4
E	0.180	76	0.036	14	0.01	1
F	0.144	52	0.064	24	0.02	1
G	0.356	82	0.140	27	0.03	11
H	5.313	810	0.981	158	0.04	17
I	1.784	129	1.472	101	0.14	3
J	4.732	828	0.436	72	0.02	13
K	1.712	610	0.188	66	0.03	36
L	0.392	98	0.152	36	0.04	13
M	0.060	27	0.024	10	0.02	12

emploi du temps décrit dans le tableau 3.10 s'assure en plus du respect des différentes contraintes, d'équilibrer du mieux possible les fins de semaine. C'est-à-dire que l'on va chercher à alterner les fins de semaine où l'on travaille avec les fins de semaine ou l'on ne travaille pas.

3.4.1 Algorithme de recherche

L'équilibrage des fins de semaine est une opération de post-optimisation[1]. Partant d'une solution qui satisfait toutes les contraintes, on va chercher à améliorer l'emploi du temps d'un point ergonomique. Pour cela, on va chercher à équilibrer du mieux possible les fins de semaines dans l'emploi du temps. On considérera qu'un emploi du temps est équilibré pour les fins de semaine si :

- on assure que au plus un dimanche sur deux soit un jour de repos.
- on assure que toutes les k semaines, on ait au plus m dimanches de repos (k et m sont spécifiés pour chaque exemplaires testés).

On va donc créer une fonction objectif qui va évaluer la qualité d'un horaire cyclique en fonction des deux paramètres d'équilibrage que l'on vient de décrire. La première partie de l'équilibrage (au plus un dimanche sur deux comme jour de repos) assure l'alternance entre les fins de semaines chômés et les fins de semaine non chômés. La deuxième partie de l'équilibrage permet de mieux répartir les dimanches chômés dans l'emploi du temps.

Pour modéliser l'équilibrage en Comet, on va employer la contrainte globale Regular pour s'assurer que au plus un dimanche sur deux soit un jour de repos. L'automate fini déterministe correspondant est décrit dans la figure 3.8. On peut par ailleurs remplacer cette contrainte avec la contrainte globale *sequence* (*sequence(ensemble des dimanches, {repos}, 1, 2)*). La deuxième partie de l'équilibrage sera modélisée via la contrainte *sequence*.

[1] On parle ici de post-optimisation car on va chercher à équilibrer les emplois du temps après les avoir trouvés.

Tableau 3.10 – Solutions de l'exemplaire F

semaine	L	Ma	Me	J	V	S	D
1	A	A	A	-	B	B	B
2	B	B	-	C	C	-	-
3	C	C	C	-	B	B	B
4	B	B	B	B	-	C	C
5	C	-	B	B	B	B	B
6	-	C	C	-	-	-	-
7	B	B	-	A	A	-	-
8	B	B	B	B	-	A	A
9	A	A	-	A	A	A	A
10	A	A	A	-	-	-	-
11	C	C	C	-	-	-	-
12	A	A	A	A	A	-	-

semaine	L	Ma	Me	J	V	S	D
1	A	A	A	A	-	B	B
2	B	B	-	B	B	-	-
3	B	B	B	-	A	A	A
4	A	A	A	-	-	-	-
5	C	C	C	C	-	A	A
6	A	A	A	A	A	-	-
7	A	A	-	-	B	B	B
8	B	B	-	A	A	-	-
9	B	B	B	B	B	B	B
10	-	C	C	-	-	-	-
11	C	C	C	-	C	C	C
12	C	-	B	B	-	-	-

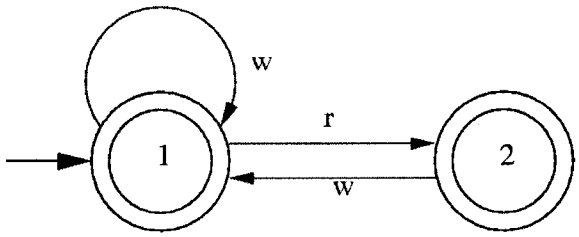

Figure 3.8 – Automate fini déterministe associé à la contrainte « au plus un dimanche de repos sur deux dimanches »

La contrainte *sequence* va alors s'assurer que pour l'ensemble des dimanches, on ait au plus m dimanches de repos dans chaque séquence de k dimanches. Le code Comet 3.11 décrit l'implémentation en Comet de la fonction objectif que l'on cherche à minimiser. Le système de contrainte $S2$ correspond aux contraintes d'équilibrage de l'emploi du temps, le système de contraintes S est le système de contrainte associé à l'horaire cyclique. La fonction objectif (O) est donc conçue comme la somme des deux systèmes de contraintes. On va toutefois pondérer l'impact du système de contrainte S par rapport au système de contraintes $S2$, l'objectif étant avant tout d'avoir un emploi du temps qui respecte les différentes contraintes.

```
// L'équilibrage des fins de semaines
ConstraintSystem S2(m);
regularINV equil(all(k in 1..N:(k%7==0)variable[k],
  nPersonne+1, automate);
S2.post(equil);
S2.post(sequence(all(k in 1..N:(k%7 == 0))variable[k],
  {nbShift}, m, k));
// Fonction objectif
Objective O = S2 + w*S;
```

Comet 3.11 – Fonction objectif à minimiser

Il est possible de remplacer la contrainte Regular par la contrainte *sequence*. La manière d'évaluer l'objectif va donc être modifiée. En effet, le degré de violation associé à la contrainte Regular correspond au nombre de caractères qu'il faut remplacer pour obtenir un mot reconnu par le langage associé. Le degré de violation associé à la

contrainte *sequence* va lui correspondre au nombre de variables violant une séquence. Cette distinction rend plus souple la contrainte Regular vis à vis de la contrainte *sequence*.

Le code Comet 3.12 décrit l'implémentation en Comet d'un algorithme de recherche tabou qui cherche une solution pour le problème d'emploi du temps équilibré. Le voisinage utilisé est le voisinage N_1. L'ensemble de valeur $V[u]$ désigne les indices des jours de l'emploi du temps correspondant au même jour de la semaine que u (par exemple le lundi).

```
best = O.evaluation();
iter = 0;
// Algorithme de Recherche
while((iter<20000) && (best>0)) {
  selectMax(u in 1.n)(O.decrease(variable[u]))
    selectMin(v in V[u] : (variable[u] != variable[v]) &&
      (tabou[u,v]<iter))
        (O.getSwapDelta(variable[u], variable[v])) {
          variable[u] :=: variable[v];
          tabou[u,v] = iter + 7;
          tabou[v,u] = iter + 7;
          if (u<=7) variable[n+u] := variable[u];
          if (v<=7) variable[n+v] := variable[v];
    }
  if(O.evaluation() < best) best = O.evaluation();
  iter++;
}
```

Comet 3.12 – Algorithme de recherche Tabou pour l'équilibrage

Une deuxième nouveauté ici est l'utilisation de la fonction *decrease*(*variable*). La fonction *decrease* indique le gain maximal que l'on pourrait avoir sur l'objectif si on modifiait la valeur de la variable en paramètre. L'algorithme consiste donc en choisir la variable qui pourrait faire évoluer du mieux possible la valeur de l'objectif, et de choisir le meilleur jour de l'emploi du temps échangeable avec qui ne soit pas tabou.

Tableau 3.11 – Solution de l'exemplaire J

semaine	L	Ma	Me	J	V	S	D
1	B	A	A	B	A	-	-
2	-	D	A	A	D	A	A
3	-	-	C	C	B	B	B
4	-	-	-	D	B	A	A
5	A	-	-	-	-	C	C
6	B	B	B	-	-	-	-
7	C	C	C	B	A	-	-
8	C	C	B	A	-	-	-
9	B	B	B	B	B	-	-
10	-	-	-	C	C	C	C
11	D	B	-	-	C	B	B
12	A	A	D	-	-	-	-

3.4.2 Résultats

On va présenter les résultats que l'on a obtenu via notre algorithme de post-optimisation pour les exemplaires C à M. Les tests ont été effectué sur la même machine que précédemment. Pour chaque exemplaire, on a effectué une série de 100 tests différents. Les tableaux 3.11 et 3.12 présentent l'emploi du temps ayant le meilleur équilibrage pour les exemplaires J et M. Les exemplaires A et B n'ont pas été testés car ils sont triviaux (un seul dimanche de repos pour quatre semaines).

Les tableaux 3.13 et 3.14 présentent et comparent les résultats moyens et médians que l'on obtient sur chaque exemplaire, selon que l'on utilise la contrainte Regular ou la contrainte *sequence* pour la première partie de l'objectif. Les résultats utilisant la contrainte Regular sont équivalents en termes de temps de calcul et d'itérations aux

Tableau 3.12 – Solution de l'exemplaire M

semaine	L	Ma	Me	J	V	S	D
1	A	A	A	A	-	C	C
2	C	-	B	B	B	B	B
3	-	C	C	C	C	-	-
4	B	-	A	-	B	B	B
5	B	B	-	A	-	-	-
6	A	A	-	B	-	A	A
7	A	A	-	A	A	A	A
8	-	A	A	A	A	-	-
9	-	B	B	-	A	A	A
10	A	-	A	-	A	-	-

résultats que l'on obtient si on utilise la contrainte *sequence* à la place. Les différences au niveau de l'évaluation des solutions ne sont visibles (sur les résultats finaux) que pour l'exemplaire J. En effet, pour l'exemplaire J, on ne trouve pas de solution qui équilibre « correctement » les fins de semaines. Le tableau 3.11 présente le meilleur exemplaire trouvé par l'algorithme de recherche pour l'exemplaire J.

Le temps moyen nécessaire pour trouver un emploi du temps plus ergonomique que celui que l'on propose est par contre plus important que le temps moyen pour obtenir un emploi du temps valide. Cela vient du fait qu'il s'agit d'une opération de post-optimisation. La solution initiale de l'algorithme est souvent un minimum local duquel il faudra s'extirper lors de la recherche.

On présente donc un algorithme de recherche locale basée sur les contraintes qui cherche, partant d'un emploi cyclique valide, à proposer un emploi du temps amélioré d'un point ergonomique. La recherche locale basée sur les contraintes, et la contrainte Regular se révèle donc être un outil puissant permettant de résoudre ce type de

Tableau 3.13 – Comparaison des résultats moyens selon l'objectif

	Avec Regular			Avec Sequence		
exemplaire	Objectif	Itérations	Temps (s)	Objectif	Itérations	Temps (s)
C	0.22	766.55	1.95	0.12	887.52	2.10
D	1.19	1358.99	2.09	1.12	1237.45	1.75
E	0.39	5634.44	15.90	0.25	5554.26	15.32
F	0.00	1681.66	4.67	0.00	1384	3.45
G	0.00	108.82	0.55	0.00	92.48	0.48
H	3.64	6981.41	45.82	3.46	6920.11	43.27
I	0.00	1124.00	12.31	0.77	2837.56	23.20
J	6.71	4248.17	28.53	8.48	2399.14	15.71
K	0.60	6336.35	18.41	0.56	6859.91	19.50
L	0.07	4053.44	16.37	0.00	2741.49	7.38
M	0.00	91.53	0.21	0.00	72.36	0.16

Tableau 3.14 – Comparaison des résultats médians selon l'objectif

	Avec Regular			Avec Sequence		
exemplaire	Objectif	Itérations	Temps (s)	Objectif	Itérations	Temps (s)
C	0	332	0.83	0	346	0.82
D	1	0	0.00	1	0	0.00
E	0	3142	9.67	0	4170	11.49
F	0	1309	3.49	0	959	2.42
G	0	26	0.12	0	14	0.07
H	4	6097	39.74	3	6325	41.40
I	0	612	7.20	0	907	8.48
J	6	0	0.00	8	0	0.00
K	0	4220	12.15	0	6258	18.03
L	0	3257	15.70	0	1887	7.38
M	0	47	0.108	0	35	0.08

problèmes.

CHAPITRE 4

CRÉATION D'EMPLOIS DU TEMPS JOURNALIERS

Le chapitre précédent décrit l'utilisation de la contrainte Regular pour la création d'horaires cycliques. Il s'agissait d'un problème de satisfaction de contraintes. Il fallait trouver une solution qui ne viole aucune contrainte.

On peut utiliser la contrainte Regular pour résoudre d'autres problèmes de création d'horaires. Le problème que l'on cherche à résoudre ici ne va plus être un problème de satisfaction de contraintes, mais un problème d'optimisation. On va devoir créer un ensemble d'emplois du temps journaliers pour plusieurs employés. Cette série d'emplois du temps devra satisfaire plusieurs contraintes, mais aussi minimiser une fonction dépendant des tâches effectuées par les employés.

4.1 Le problème de création d'emplois du temps journaliers

Dans cette section, nous allons décrire le problème, ainsi que deux approches qui ont été proposées pour le résoudre (Côté et al. 2007) (Demassey et al. 2005).

4.1.1 Description du problème

Le problème est donc la création d'emplois du temps pour plusieurs employés. Les emplois du temps sont construits pour une journée de 24 heures découpées en 96 périodes de quinze minutes et doivent respecter un ensemble de contraintes.

Pour décrire le problème, nous allons définir les notations suivantes :

- I : l'ensemble des employés disponibles.
- W : l'ensemble des différentes tâches de travail que peut effectuer un employé.

- A : l'ensemble des activités que peut effectuer un employé. Les activités peuvent être soit une tâche de travail $a \in W$, soit une pause p, soit une pause repas l, soit un repos o. On a donc $A = W \cup \{p, l, o\}$.
- $T = \{1, 2, \ldots, 96\}$: l'ensemble des périodes de la journée.
- $A_t \subseteq A$: l'ensemble des activités autorisées pour un employé à l'instant $t \in T$.
- $c_{a,t}$: le coût d'utilisation d'un employé pour effectuer à l'instant t la tâche $a \in A_t$.

Contraintes d'emplois du temps Les emplois du temps sont soumis à plusieurs contraintes différentes que nous allons énoncer :

- Un employé $i \in I$ n'a pas le droit d'effectuer à l'instant t une tâche $a \in A$ si $a \notin A_t$.
- Un employé qui travaille doit effectuer entre 3 et 8 heures de travail effectifs, c'est-à-dire sans prendre en compte les différents types de pauses.
- Un employé qui travaillera 6 heures ou plus aura dans sa journée deux pauses p de 15 minutes et une pause repas l d'une heure.
- Un employé qui travaillera moins de 6 heures aura dans sa journée une pause p de 15 minutes et pas de pause repas l.
- L'emploi du temps de la journée devra respecter un patron précis.

Un employé qui commence à effectuer une tâche $a \in A$ doit l'effectuer pendant au moins une heure avant de prendre une pause. Deux tâches a et $b \in A$ ne peuvent pas être effectuées l'une après l'autre. Un employé devra prendre une pause avant de passer de la tâche a à la tâche b. Pour représenter le patron à respecter, on utilise un automate fini déterministe. Les figures 4.1 et 4.2 présentent les patrons à respecter pour une tâche de travail et pour deux tâches de travail.

Demande à respecter et objectif à minimiser L'ensemble des emplois du temps devront être construits pour répondre à une demande. On note $d_{a,t}$ le nombre d'em-

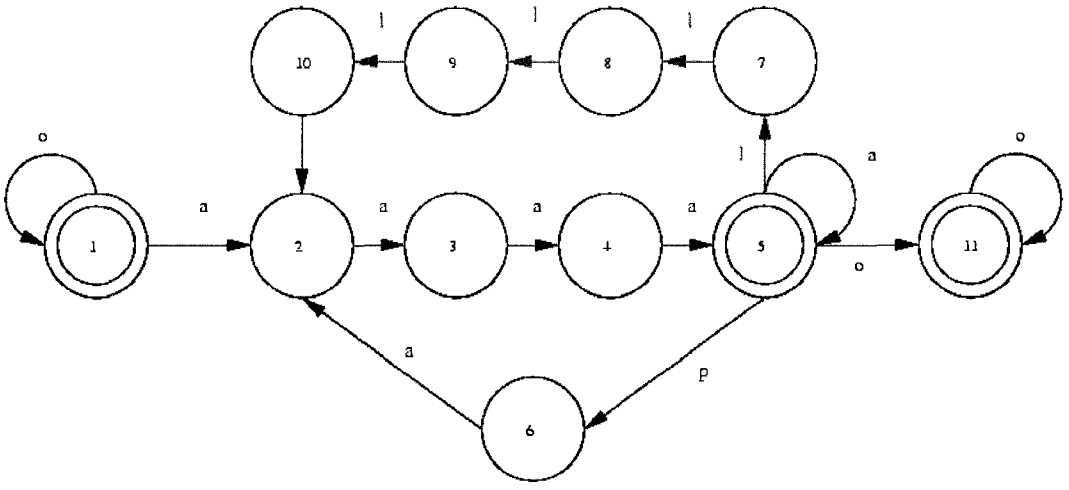

Figure 4.1 – Patron de l'emploi du temps pour une tâche

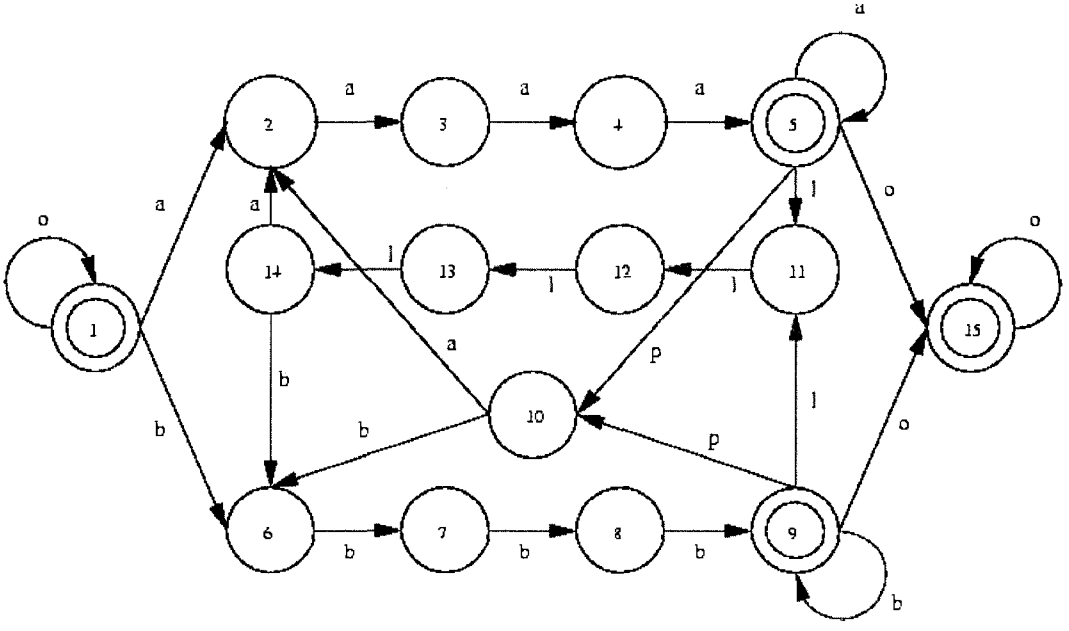

Figure 4.2 – Patron de l'emploi du temps pour deux tâches différentes

ployés devant effectuer la tâche $a \in W$ à l'instant t. Si la demande n'est pas satisfaite, un coût $c^-_{a,t}$ est associé à chaque employé manquant. De même, si la demande est sur-satisfaite, un coût $c^+_{a,t}$ est associé à chaque employé de trop.

Pour faciliter les notations, on va noter $q_{a,t}$ le nombre d'employés effectuant la tâche $a \in W$ à l'instant t. On va noter $q^-_{a,t}$ le nombre d'employés manquant ($q^-_{a,t} = 0$ si la demande $d_{a,t}$ est satisfaite). De la même manière, $q^+_{a,t}$ est le nombre d'employés en trop effectuant la tâche $a \in W$ à l'instant t ($q^+_{a,t} = 0$ si la demande $d_{a,t}$ n'est pas satisfaite, ou si elle l'est exactement).

Le problème consiste donc à minimiser la fonction f suivante, tout en respectant les différentes contraintes :

$$f = \sum_{t \in T, a \in A_t} (c_{a,t} q_{a,t} + c^-_{a,t} q^-_{a,t} + c^+_{a,t} q^+_{a,t})$$

4.1.2 Approches existantes pour résoudre le problème

Le problème que l'on cherche à résoudre peut se modéliser en partie à l'aide de la contrainte Regular (Pesant 2004). En effet, les contraintes portant sur le respect d'un patron se modélisent bien à l'aide d'automate fini déterministe. Marie-Claude Côté, Bernard Gendron et Louis-Martin Rousseau (Côté et al. 2007) présentent une implémentation de la contrainte Regular en programmation mixte en nombre entier (en anglais Mixed Integer Programming ou MIP). La contrainte Regular est notamment utilisée pour résoudre le problème de création d'horaires évoqué ci-dessus.

Une approche couplant la programmation par contraintes et la génération de colonne (Demassey et al. 2005) est aussi proposé pour résoudre ce problème. L'approche en programmation par contraintes sert pour le sous-problème de générations de colonnes.

4.2 Approche de résolution en COMET

Dans ce chapitre, nous allons décrire l'implémentation d'un algorithme de recherche locale basée sur les contraintes en Comet pour résoudre le problème de création d'horaires journaliers. La modélisation Comet se sépare naturellement en deux grandes phases : une phase de modélisation et une phase de recherche.

4.2.1 La modélisation

4.2.1.1 Les variables et les invariants

Dans ce paragraphe, nous allons décrire les variables utilisées dans le modèle, ainsi que les différents invariants. Les invariants en Comet sont en fait des variables incrémentales qui maintiennent à jour une relation entre une ou plusieurs variables. Avec l'aide des invariants, on va pouvoir modéliser certaines contraintes de manières plus simples.

Les variables de notre problème en Comet vont être notées $v_{i,t}, \forall i \in I, t \in T$. $v_{i,t}$ indique la tâche effectuée par l'employé i à l'instant t. On a donc $v_{i,t} \in A_t$.

Les invariants que l'on utilise tout au long de la recherche vont être les suivants :

- σ_i : Pour chaque employé $i \in I$, σ_i indique le nombre de périodes ou l'employé i travaille.
- $\sigma_{p,i}$: Pour chaque employé $i \in I$, $\sigma_{p,i}$ indique le nombre de périodes de pause prises par l'employé i.
- $\sigma_{l,i}$: Pour chaque employé $i \in I$, $\sigma_{l,i}$ indique le nombre de périodes de pause repas prises par l'employé i.
- $q_{a,t}$: $q_{a,t}$ correspond au nombre d'employés effectuant à l'instant $t \in T$ la tâche $a \in A_t$

Le code Comet 4.1 décrit l'implémentation et la description en Comet des variables et des invariants propres au problème.

```
// Variables de décisions
var{int} variable[1..nbEmploye, 1..nbShift](m, 1..nbActTot)
  := nbActTot;
// Nombre de sections de travail pour chaque employé
var{int} sigma[i in 1..nbEmploye](m, 0..nbShift)
  <- sum(k in 1..nbShift)(variable[i,k] <= nbAct);
// Nombre de sections de pause pour chaque employé
var{int} sigmaP[i in 1..nbEmploye](m, 0..nbShift)
  <- sum(k in 1..nbShift)(variable[i,k] == nbAct+1);
// Nombre de section de pause repas
var{int} sigmaL[i in 1..nbEmploye](m, 0..nbShift)
  <- sum(k in 1..nbShift)(variable[i,k] == nbAct+2);
// Nombre de personne effectuant l'activité i à un instant j
var{int} quantite[j in 1..nbAct, i in 1..nbShift](m,
  0..nbEmploye)
    <- sum(k in 1..nbEmploye)(variable[k,i]==j);
```

Comet 4.1 – Descriptions des différentes variables de décisions et des invariants

4.2.1.2 Les différentes contraintes à satisfaire

La contrainte de patron Comme on l'a dit précédemment, on utilise la contrainte Regular pour modéliser la contrainte de respect des patrons pour la création d'un emploi du temps. En effet, pour chaque employé $i \in I$, l'emploi du temps proposé doit respecter un ensemble de contraintes. Un employé qui effectue la tâche $a \in W$ à un instant t ne peut pas effectuer la tâche $b \in W, b \neq a$ à l'instant $t+1$. Cela veut dire qu'un employé ne peut pas basculer d'une tâche à l'autre sans prendre une pause. Un employé qui veut prendre une pause doit avoir préalablement travaillé pendant au moins une heure (4 périodes de temps). Une pause repas dure une heure (4 périodes de temps), la durée d'une pause normale est de quinze minutes (1 période de temps). Enfin, un employé ne peut pas terminer sa journée en pause. L'ensemble de ces contraintes est modélisé via un automate fini déterministe pour $W = \{a\}$ (voir la figure 4.1) et $W = \{a, b\}$ (voir la figure 4.2). Le code Comet 4.2 décrit l'utilisation de la contrainte Regular en Comet. Si on considère Π_W l'automate fini déterministe associé à l'ensemble des tâches W, la contrainte Regular s'utilise pour chaque employé

$i \in I$ de la manière suivante :

$$\text{Regular}(\Pi_W, (v_{i,t})_{t \in T}), \forall i \in I$$

```
regularINV cstr[1..nbEmploye];
forall(i in 1..nbEmploye) {
  cstr[i] = new regularINV((all(k in 1..nbShift) variable[i,k])
     , nbShift, afd);
  S.post(cstr[i]);
}
```

Comet 4.2 – Utilisation de la contrainte Regular pour le respect des patrons

Les contraintes sur les durées de travail Comme on l'a dit dans le paragraphe 4.1.1, le nombre de pauses prises par l'employé i dépend de la quantité de travail qu'il effectue. Les différentes contraintes régissant le nombre de pauses et la durée du travail sont donc les suivantes :

- $(12 \leq \sigma_i \leq 32) \vee (\sigma_i = 0), \forall i \in I$
- $(\sigma_i \geq 24) \Rightarrow ((\sigma_{i,p} = 2) \wedge (\sigma_{i,l} = 4)), \forall i \in I$
- $(\sigma_i < 24) \Rightarrow ((\sigma_{i,p} = 1) \wedge (\sigma_{i,l} = 0)), \forall i \in I$

La première contrainte signifie simplement qu'un employé, s'il travaille, doit travailler entre 3 et 8 heures. La seconde dit que si un employé travaille 6 heures ou plus, il a droit à une pause repas de une heure et deux pauses de quinze minutes dans sa journée de travail. La dernière contrainte dit que si l'employé travaille moins de 6 heures, il ne peut pas prendre de pause repas et juste une pause de quinze minutes dans la journée.

Le code Comet 4.3 décrit l'implémentation de ces contraintes dans le langage de programmation Comet.

```
forall(i in 1..nbEmploye) {
  // Au maximum 8 heures de travail dans la journée
  S.post(sigma[i] <= 32);
  // Au minimum soit 0 (et donc pas de travail) soit 3 heures
  S.post((sigma[i] == 0) + (sigma[i] >= 12)) >= 1);
  // Contraintes relatives à la pause repas
  S.post((sigma[i] >= 12) => (sigmaL[i] >= 0));
  S.post((sigma[i] >= 24) => (sigmaL[i] >= 4));
  S.post((sigma[i] >= 24) => (sigmaL[i] <= 0));
  // Contraintes relatives aux pauses
  S.post((sigma[i] >= 12) => (sigmaP[i] >= 1));
  S.post((sigma[i] >= 24) => (sigmaP[i] >= 2));
  S.post((sigma[i] <= 23) => (sigmaP[i] <= 1));
}
```

Comet 4.3 – Contraintes sur les quantités de travail pour un employé

4.2.1.3 La fonction objectif

La fonction que l'on cherche à minimiser est la suivante :

$$\sum_{t \in T, a \in A_t} (c_{a,t} q_{a,t} + c^-_{a,t} q^-_{a,t} + c^+_{a,t} q^+_{a,t})$$

Pour modéliser la fonction objectif, on va utiliser une classe particulière d'objectif en Comet : **FloatObjectiveSum**. Cette classe d'objectif Comet permet de spécifier une fonction objectif comme la somme de plusieurs fonctions qu'on lui « poste ». À l'instar des systèmes de contraintes, on peut voir **FloatObjectiveSum** comme un système de fonctions objectifs.

$q^-_{a,t}$ désigne le nombre d'employé manquant pour satisfaire la demande $d_{a,t}$ d'employé effectuant à l'instant t la tâche $a \in A_t$, $q^+_{a,t}$ le nombre d'employé en trop. On va calculer $q^-_{a,t}$ de la manière suivante : $q^-_{a,t} = max(0, d_{a,t} - q_{a,t})$. De la même manière, on calcule $q^+_{a,t}$ de la manière suivante : $q^+_{a,t} = max(0, q_{a,t} - d_{a,t})$.

Le code Comet 4.4 décrit en Comet l'implémentation de la fonction objectif.

```
FloatObjectiveSum OS(m);
forall(i in 1..nbShift,  j in 1..nbAct) {
  // Sous-Couverture de la tâche j à l'instant i
  OS.post((undercost[j,i])*max(0.0, demand[j,i]-quantite[j,i]))
     ;
  // Sur-Couverture de la tâche j à l'instant i
  OS.post((overcost[j,i])*max(0.0, quantite[j,i]-demand[j,i]));
  // Coût des employés effectuant la tache j à l'instant i
  OS.post(C[j,i]*quantite[j,i]);
}
```

Comet 4.4 – Objectif à minimiser

4.2.2 La recherche

Le problème que l'on cherche à résoudre est un problème d'optimisation. On cherche à créer un ensemble d'emplois du temps, obéissant à plusieurs contraintes qui minimise une fonction objectif. Cette fonction objectif dépend du nombre de personnes effectuant une tâche donnée à chaque instant de la journée.

4.2.2.1 Solution initiale

Pour tous les exemplaires que l'on testera, l'algorithme partira de la même solution initiale, à savoir un emploi du temps vide. L'algorithme construira et réparera alors itérativement les emplois du temps dans l'optique de trouver celui qui minimisera le plus la valeur de l'objectif en respectant les contraintes.

4.2.2.2 Algorithme de résolution

Pour résoudre le problème, on utilise un algorithme de recherche itératif en trois phases. La première phase consiste à choisir l'employé ayant l'emploi du temps « le plus prometteur » pour la fonction objectif. La deuxième phases consiste à modifier cet emploi du temps, sans prendre en compte les contraintes, de manière à minimiser l'objectif. La dernière phase est une phase de réparation. On répare l'emploi du temps

de manière à ce qu'il ne viole aucune contrainte. L'algorithme 4.1 décrit l'algorithme employé.

Algorithme 4.1 : Algorithme de recherche pour le problème

tant que *le critère d'arrêt n'est pas atteint* **faire**

1 Choisir l'emploi du temps *a priori* le plus prometteur en fonction de l'objectif.

2 Chercher pour cet emploi du temps la meilleure affectation possible de tâches sans prendre en compte les contraintes.

3 Réparer cet emploi du temps, de manière à ce qu'il respecte les contraintes.

fin

Un raffinement que l'on propose est la transformation de cet algorithme en un algorithme de recherche Tabou. En effet, on rajoute un critère tabou sur le choix de l'emploi du temps que l'on va modifier dans la première phase de l'algorithme. On obtient alors un algorithme de recherche Tabou pour la résolution du problème de création d'emplois du temps. Un deuxième raffinement de l'algorithme consiste à chercher à améliorer la valeur de l'objectif après la réparation sans violer aucune contrainte.

4.2.2.3 Implémentation en Comet

Le code Comet 4.5 décrit l'implémentation de l'algorithme 4.1 dans le langage de programmation Comet. Le code insiste notamment sur les trois phases de l'algorithme. La première phase utilise la fonction Comet *decrease(variable)* qui propose le gain maximum que l'on pourrait espérer avoir si on changeait la valeur de la variable passée en paramètre.

Les deuxième et troisième phases utilisent toutes les deux un algorithme de recherche Tabou. L'algorithme de satisfaction de réparation de l'emploi du temps (phase

```
while(/* critère d'arrêt non atteint */) {
  // PHASE 1 : choix de l'emploi du temps le plus prometteur
  selectMax(i in 1..nbEmploye :
    (bigTabou[i] < bigIter) || (tabViol[i] != 0))
    (1000000*tabViol[i] +
      sum(k in 1..nbShift)(OS.decrease(variable[i,k]))) {
        iter = 1;
        // PHASE 2 : proposition du meilleur emploi du temps
           possible sans prendre les contraintes en compte
        while(/* critère d'arrêt non atteint */) {
          selectMin(j in 1..nbShift, q in 1..nbAct :
            (tabou2[j] < iter))
            (OS.getAssignDelta(variable[i,j], q)) {
              variable[i,j] := q;
              tabou2[j] = iter+7;
          }
          iter++;
        }
      // PHASE 3 : réparation pour obtenir un emploi du temps
         valide
      iter = 1;
      int gap; // Pour critère d'aspiration
      int bestTemp = S.violations(); // Pour critère d'
         aspiration
      forall( j in 1..nbShift ) tabou[j] = -1;
      while(/* Emploi du temps non valide */) {
        gap = bestTemp - S.violations();
        selectMin(j in 1..nbShift, q in 1..nbActTot,
          d = S.getAssignDelta( variable[i,j], q ) :
          ((tabou[j] < iter) || (d<gap)) &&
            (variable[i,j] != q))(d) {
              variable[i,j] := q;
              tabou[j] = iter + 7;
        }
        iter++;
      }
      // Mise 0 jour de la meilleure solution trouvée
      bigIter = bigIter + 1;
  }
}
```

Comet 4.5 – Algorithme de recherche locale - méthode Tabou

3) utilise en outre un critère d'aspiration. Une itération de l'algorithme est donc l'exécution des trois phases.

La phase de réparation peut ne pas aboutir rapidement. On arrête alors la réparation après un temps de recherche alloué. Ceci est fait dans le but de ne pas s'enfermer dans la phase de réparation pendant trop longtemps. En contrepartie, le choix de l'emploi du temps à modifier (première phase) dépendra aussi du degré de violations de l'emploi du temps.

4.3 Résultats

4.3.1 Descriptions des instances

Pour chacun des exemplaires testés, lorsque la demande $d_{a,t}$ à l'instant $t \in T$ de la tâche $a \in A$ est nulle, on a $a \notin A_t$. De ce fait, on va réduire T à un ensemble T' tel que $T' = \{t \in T : A_t \cap W \neq \emptyset\}$. T' sera donc l'ensemble des intervalles tels qu'il y ait au moins une tâche $a \in W$ de demande non nulle à l'instant $t \in T'$.

De plus, le coût d'utilisation d'un employé $i \in I$ effectuant à l'instant $t \in T'$ la tâche $a \in A_t$ aura la même valeur que le coût de sur-couverture $c^+_{a,t}$. La fonction objectif f que l'on va chercher à minimiser est donc la suivante :

$$f = \sum_{t \in T', a \in A_t} (c^+_{a,t} q_{a,t} + c^-_{a,t} q^-_{a,t} + c^+_{a,t} q^+_{a,t})$$

4.3.2 Résultats

4.3.2.1 Calcul d'une borne inférieure

Pour chacun des exemplaires testés, nous avons calculé une borne inférieure de la valeur optimale. Le calcul de la borne inférieure Z se fait en utilisant la formule

suivante :

$$Z = \sum_{t \in T', a \in A_t} (\min_{k \in [0...|I|]} (c^+_{a,t}.k + c^-_{a,t}.min(0, d_{a,t} - k) + c^+_{a,t}.min(0, k - d_{a,t})))$$

$|I|$ indique le nombre d'employés disponibles.

Ainsi, lors de la recherche locale, on pourra estimer la valeur d'une solution en fonction de l'écart relatif E_R avec la borne inférieure. On va calculer l'écart relatif E_R de la manière suivante :

$$E_R = (\frac{Sol - Z}{Z}) * 100$$

4.3.3 Première série d'exemplaires

Nous avons dans un premier temps effectué nos tests sur un panel de 20 exemplaires différents. Ces exemplaires ont comme particularité d'avoir des demandes $d_{a,t}, \forall t \in T', \forall a \in A_t$ qui peuvent ne pas être à valeur entière. C'est-à-dire que l'on peut demander $2,5$ employés pour effectuer la tâche a à un instant t.

Le tableau 4.1 donne les meilleurs résultats pour les exemplaires testés obtenus par 3 tests de 60 minutes. Lors de chaque test, on arrêtera la recherche quand on trouvera une solution à 1% de l'optimalité, c'est-à-dire telle que l'on ait la relation suivante : $E_R \leq 1$.

Les résultats obtenus sont relativement bons. En effet, pour la majorité des exemplaires à une tâche, on trouve une solution qui est à moins de 3,5% de la borne inférieure que l'on calcule.

Les exemplaires à deux tâches offrent aussi de bons résultats. La moitié des exemplaires testés offrent des solutions à au plus 5% de l'optimalité (5,27% pour être précis). L'écart relatif avec la borne inférieur pour les autres exemplaires est plus important (entre 12 et 33%). Néanmoins, comme précédemment, on ne connait pas la valeur de la solution optimale. On ne sait pas si les solutions trouvées sont proches ou non de l'optimalité.

Tableau 4.1 – Résultat obtenus pour les instances à une tâche et à deux tâches

	1 tâche			2 tâches		
Instance	Temps	Solution	E_R	Temps	Solution	E_R
1	T_{limite}	179,54	32,17	T_{limite}	291,05	12,95
2	56,58	159,22	0,61	T_{limite}	201,02	2,01
3	374,85	165,17	0,65	T_{limite}	254,89	1,08
4	283,46	129,88	0,96	T_{limite}	248,41	1,83
5	37,98	136,08	0,70	T_{limite}	522,11	33,08
6	T_{limite}	124,08	3,22	537,49	285,72	0,76
7	T_{limite}	140,69	1,52	T_{limite}	276,44	21,94
8	1340,78	163,04	0,52	T_{limite}	682,18	19,37
9	T_{limite}	182,59	28,14	T_{limite}	338,02	13,42
10	T_{limite}	142,18	1,71	T_{limite}	302,67	5,27

4.3.4 Deuxième série d'exemplaires

La version implémentée de la contrainte Regular en programmation en nombre entier (Côté et al. 2007) est utilisée pour résoudre le problème de création d'emplois du temps journaliers, en ayant cette fois des demandes à valeurs entières. La demande $d_{a,t}, \forall t \in T', a \in A_t$ des exemplaires précédents va être modifiée et on ne considère que la valeur seuil des demandes. On remarque par ailleurs que pour un des exemplaires testés, la demande ne change pas, c'était déjà une demande ne comprenant uniquement que des valeurs entières ; il s'agit du neuvième exemplaire à une tâche. La solution trouvée précédemment est à moins de 1% de l'optimalité.

Les résultats obtenus par Côté et al. (2007) sur les instances à une tâche et quelques résultats obtenus sur les instances à deux tâches sont à moins de 1% de la valeur optimale. On va donc se servir de cette information pour affiner notre calcul de borne inférieure. On calcule ainsi une seconde valeur de la borne inférieur Z^2. On a $Z^2 = \frac{100 * sol_{MIP}}{101}$, où sol_{MIP} est la valeur de la solution trouvée à au plus 1% de l'optimum. On calcule ensuite notre nouvelle borne inférieure $Z^* = \max(Z, Z^2)$. On calcule un nouvel écart relatif E_R^* avec la valeur de Z^*.

Les tableaux 4.2 et 4.3 donne les résultats que l'on obtient sur les instances à une et deux tâches, en les comparant avec les résultats obtenus par Côté et al. (2007). Pour les exemplaires à une tâche (voir le tableau 4.2), les résultats sont dans l'ensemble équivalents en terme de qualité de la solution. Le temps de calcul de la méthode en Comet est par contre plus important.

Pour les exemplaires à deux tâches, les résultats obtenus avec Comet sont moins bons. Les résultats trouvés avec Comet sont en moyenne à 7,7% des solutions trouvées en utilisant la programmation mixte en nombre entier. Il faut toutefois relativiser, en effet, on utilise en Comet une heuristique, pour laquelle on n'a aucune garantie d'obtenir la solution optimale du problème. La taille du problème ne permet pas de faire beaucoup d'itérations dans le temps imparti (une itération par seconde dans le

Tableau 4.2 – Comparaison des Résultats obtenus pour les instances à une tâche

	MIP-Regular		Comet		
Instance	Temps	Solution	Temps	Solution	E_R^*
1	1,50	172,67	164,46	172,67	1,00
2	1029,01	164,58	2703,64	163,78	0,51
3	393,96	169,44	T_{limite}	172,59	2,23
4	39,89	133,45	2810,37	133,07	0,70
5	10,41	145,67	8,48	145,60	0,95
6	20,36	135,06	885,12	134,81	0,82
7	6,25	150,36	T_{limite}	150,38	1,01
8	274,92	148,05	469,37	148,05	0,58
9	15,47	182,54	2565,07	182,54	1,00
10	5,50	147,63	T_{limite}	148,48	1,58

meilleur des cas, une itération pour quatre secondes dans le pire). Ceci est dû à la fois à la taille des exemplaires, mais aussi à la grosseur de la contrainte Regular.

4.3.5 Conclusion

Le langage Comet permet via la contrainte Regular de modéliser et de proposer un algorithme de recherche locale pour résoudre le problème d'emplois du temps journaliers évoqué ci-dessus. Comet permet d'avoir un algorithme qui fournit des résultats corrects, via un algorithme de recherche peu élaboré. Les résultats, d'un point de vue qualitatif, sont certes moins bons que la méthode exacte développée par (Côté et al. 2007), mais n'en sont pas moins mauvais. Pour les instances à une tâche, Comet offre des solutions qui sont à 0.14% en moyenne des résultats obtenus par (Côté et al. 2007). Pour les exemplaires à deux tâches, Comet est en moyenne à 7%.

Tableau 4.3 – Comparaison des Résultats obtenus pour les instances à deux tâches

	MIP-Regular		Comet		
Instance	Temps	Solution	Temps	Solution	E_R^*
1	623,00	201,78	T_{limite}	222,34	10,19
2	T_{limite}	214,42	T_{limite}	220,58	4,29
3	784,94	260,80	T_{limite}	261,18	1,14
4	1128,73	245,87	T_{limite}	251,50	3,31
5	T_{limite}	412,31	T_{limite}	500,99	34,59
6	107,75	288,88	2734,54	288,30	0,71
7	45,44	232,14	T_{limite}	254,24	10,61
8	643,73	536,83	T_{limite}	615,83	15,39
9	T_{limite}	309,69	T_{limite}	348,21	13,56
10	T_{limite}	265,67	T_{limite}	275,16	6,18

Comet rivalise donc avec des méthodes performantes pour un problème « relativement difficile ».

CONCLUSION

Dans ce mémoire, nous avons présenté l'implémentation d'une contrainte globale dans le langage de programmation Comet. Comet propose de résoudre des problèmes d'optimisation en combinant une modélisation des problèmes héritée de la programmation par contraintes avec des heuristiques de recherche locale. Comet permet à l'utilisateur d'implémenter lui-même ses propres contraintes.

La contrainte Regular est une contrainte globale qui assure qu'une séquence de variables soit reconnue par un langage régulier et l'automate fini déterministe associé. Pour implémenter la contrainte Regular en Comet, nous avons employé une structure de données particulière, le graphe en couche associé à la contrainte. Via ce graphe en couche, nous avons montré comment estimer le degré de violation de la contrainte pour une assignation de variables, mais aussi comment estimer la qualité de deux mouvements locaux relativement simples.

Nous avons par ailleurs porté l'emphase sur l'utilisation de la contrainte Regular dans le langage Comet. Pour cela, nous avons étudié deux problèmes de recherche opérationnelle, pouvant être modélisés en partie en utilisant la contrainte Regular. Le premier problème est un problème de satisfaction de contraintes. Le problème consiste à fournir un emploi du temps cyclique étalé sur plusieurs semaines obéissant à diverses contraintes. Nous avons présenté une modélisation utilisant la contrainte Regular ainsi qu'un algorithme de recherche Tabou pour résoudre le problème. Nous avons aussi présenté un problème de post-optimisation relatif à ce problème, visant à améliorer la qualité ergonomique des emplois du temps proposés. On peut alors utiliser la contrainte Regular comme membre de la fonction objectif estimant l'ergonomie des emplois du temps. Un algorithme de recherche tabou a aussi été proposé pour résoudre ce problème.

Le second problème était un problème d'optimisation consistant à proposer un ensemble d'emplois du temps pour plusieurs employés respectant un ensemble de

contraintes et minimisant une fonction dépendant de la quantité de travail effectuée par les employés. La contrainte Regular a été utilisée pour assurer que l'emploi du temps de chaque employé respecte un patron précis. Nous avons par ailleurs implémenté un algorithme de recherche locale pour résoudre ce problème.

La contrainte Regular est une contrainte qui, implémentée en Comet, offre une puissance expressive supplémentaire au langage. Ainsi, grâce à la contrainte Regular, on peut implémenter des contraintes de respect de patrons de manière relativement simple. Toutefois, Regular en Comet est une contrainte relativement lourde. Le temps de mise-à-jour dépend des variables associées à la contrainte, mais aussi de la taille de l'automate associé.

De plus, nous avons remarqué dans ce travail que la contrainte Regular perd en efficacité si on l'utilise pour modéliser des contraintes qui ne sont pas à proprement parler des contraintes de respect de patrons. Ainsi, dans le cadre du problème de création d'horaires cycliques, il apparaît que le gain que l'on a à utiliser la contrainte Regular pour un langage plus précis, c'est-à-dire représentant outre le patron à respecter des contraintes annexes ne compense pas en terme de temps de calcul l'utilisation de la contrainte Regular pour des patrons plus simples, les contraintes annexes étant modélisées autrement.

La contrainte Regular est donc un outil de modélisation qui permet de modéliser de manière simple un ensemble de contraintes qui ne le sont pas nécessairement. Nous avons vu que, le temps de calcul lorsque l'on emploie la contrainte Regular dépend en partie de l'automate auquel on l'associe, nous conseillons l'utilisation de la contrainte Regular en recherche locale basée sur les contraintes uniquement si on ne l'utilise que pour modéliser une contrainte de respect de patrons. La contrainte Regular est néanmoins un outil de modélisation offrant une plus grande expressivité du langage et guidant intelligemment la recherche locale vers une solution satisfaisant les contraintes.

La contrainte Regular dépend donc fortement de l'automate fini déterministe qui lui est associé. Pour améliorer le temps de calcul lors de l'utilisation de la contrainte Regular en Comet, une piste de travail pourrait être l'utilisation d'automates finis non déterministes en lieu des automates finis déterministes. En effet, les automates finis non déterministes permettent de représenter les langages réguliers, en réduisant le nombre d'états de l'automate. Or, le temps de mise-à-jour de la contrainte Regular dépend directement de la taille de l'automate. L'utilisation d'automates finis non déterministes peut permettre d'accélérer le temps de mise-à-jour sans pour autant interférer sur la qualité expressive de la contrainte.

BIBLIOGRAPHIE

APT, K. (2003). *Principles of Constraint Programming.* Cambridge University Press 2003.

CODOGNET, P. et DIAZ, D. (2001). Yet another local search method for constraint solving. In STEINHÖFEL, K., editor, *SAGA*, volume 2264 of *Lecture Notes in Computer Science*, pages 73–90. Springer.

CÔTÉ, M.-C., GENDRON, B., et ROUSSEAU, L.-M. (2007). Modeling the regular constraint with integer programming. In HENTENRYCK, P. V. et WOLSEY, L. A., editors, *CPAIOR*, volume 4510 of *Lecture Notes in Computer Science*, pages 29–43. Springer.

DEMASSEY, S., PESANT, G., et ROUSSEAU, L.-M. (2005). Constraint programming based column generation for employee timetabling. In BARTÁK, R. et MILANO, M., editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 140–154. Springer.

DORIGO, M. et STÜTZLE, T. (2004). *Ant Colony Optimization.* MIT Press.

GALINIER, P. et HAO, J. (2000). A general approach for constraint solving by local search. In *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'00), Paderborn, Germany.*

GLOVER, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & OR*, **13**(5), 533–549.

HENTENRYCK, P. V. (1999). Localizer : A modeling language for local search. In FAGES, F., editor, *JFPLC*, pages 143–144. Hermes.

HENTENRYCK, P. V. et MICHEL, L. (2005). *Constraint-Based Local Search.* MIT Press.

HENTENRYCK, P. V. et MICHEL, L. (2006). Differentiable invariants. In BENHAMOU, F., editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 604–619. Springer.

HENTENRYCK, P. V., MICHEL, L., et LIU, L. (2004). Constraint-based combinators for local search. In WALLACE, M., editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 47–61. Springer.

HOLLAND, J. H. (1992). *Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA.

KIRKPATRICK, S., GELATT, D., et VECCHI, M. P. (1983). Optimization by simmulated annealing. *Science*, **220**(4598), 671–680.

KLEENE, S. (1956). *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J.

LAPORTE, G. et PESANT, G. (2004). A general multi-shift scheduling system. *Journal of the Operational Research Society*, **55(11)**, 1208–1217.

MICHEL, L. et HENTENRYCK, P. V. (2000). Localizer. *Constraints*, **5**(1/2), 43–84.

MICHEL, L. et HENTENRYCK, P. V. (2002). A constraint-based architecture for local search. In *OOPSLA*, pages 83–100.

MICHEL, L. et HENTENRYCK, P. V. (2003). Comet in context. In GOLDIN, D. Q., SHVARTSMAN, A. A., SMOLKA, S. A., VITTER, J. S., et ZDONIK, S. B., editors, *PCK50*, pages 95–107. ACM.

MLADENOVIC, N. et HANSEN, P. (1997). Variable neighborhood search. *Computers & OR*, **24**(11), 1097–1100.

MÖRZ, M. et MUSLIU, N. (2004). Genetic algorithm for rotating workforce scheduling. In *Proceedings of second IEEE International Conference on Computational Cybernetics*.

MUSLIU, N. (2006). Heuristic methods for automatic rotating workforce scheduling. *International Journal of Computational Intelligence Research*, **Volume 2**.

PAGE WIKIPEDIA DE LA DISTANCE DE LEVENSHTEIN (n.d.). `http://en.wikipedia.org/wiki/Levenshtein_distance`.

PESANT, G. (2004). A regular language membership constraint for finite sequences of variables. In WALLACE, M., editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer.

PESANT, G. et GENDREAU, M. (1999). A constraint programming framework for local search methods. *J. Heuristics*, **5**(3), 255–279.

SITE INTERNET OFFICIEL DE COMET (n.d.). `http://www.comet-online.org`.

TIEN, J. et KAMIYAMA, A. (1982). On manpower scheduling algorithms. *SIAM Review*.

VAN HOEVE, W. J., PESANT, G., et ROUSSEAU, L.-M. (2006). On global warming : Flow-based soft global constraints. *J. Heuristics*, **12**(4-5), 347–373.

WALINSKY, C. (1989). $CLP(\Sigma^*)$: Constraint logic programming with regular sets. In LEVI, G. et MARTELLI, M., editors, *Logic Programming : Proc. of the Sixth International Conference*, pages 181–196. MIT Press, Cambridge, MA.

WIKI INTERNET OFFICIEL DE COMET (n.d.). `http://willow.engr.uconn.edu/cometPubWiki/index.php/Main_Page`.