Table des matières

Liste des tableaux

Liste des figures

Remerciements

J'aimerais, tout d'abord, remercier le professeur Jonathan Gaudreault, mon directeur de recherche, qui a accepté de diriger mes travaux. Merci d'avoir cru en moi et de m'avoir conseillé judicieusement lors de la réalisation de cet ambitieux projet qu'est la maîtrise. Je souhaite aussi remercier le professeur Claude-Guy Quimper, mon codirecteur de recherche. Ses connaissances et explications m'ont permis d'en apprendre beaucoup sur des aspects plus techniques de la programmation par contraintes.

Je remercie aussi la professeure Josée Desharnais qui a accepté d'être membre du jury lors de l'évaluation de mon mémoire et qui a pris le temps de me fournir des commentaires détaillés afin d'en améliorer la qualité.

Le chapitre 3 n'aurait pas été possible sans la contribution de Marc-André Ménard, étudiant de FORAC, ainsi que Philippe Marier, professionnel de recherche de FORAC. Merci d'avoir pris le temps de m'expliquer le problème de la génération des patrons de chargement dans un séchoir ainsi que d'avoir préparé les données en entrée qui m'ont permis de tester l'approche proposée.

Sur une note plus personnelle, je ne pourrais jamais assez remercier mes parents. Je n'aurais jamais pu me rendre aussi loin dans mes études sans leur soutien inconditionnel lors de toutes les étapes de ma vie. Je souhaite aussi remercier ma famille ainsi que mes amis pour leurs encouragements tout au long de la maîtrise.

Introduction

L'optimisation combinatoire est une branche de la mathématique appliquée et de l'informatique qui s'intéresse à la recherche d'une solution optimale pour un problème où certaines variables ne peuvent prendre que des valeurs discrètes. L'une des méthodes employées pour résoudre ces problèmes est la *programmation par contraintes* (PPC), un paradigme de programmation associé à la grande famille de l'intelligence artificielle [12].

Une citation d'Eugene C. Freuder, dans son article *In Pursuit of the Holy Grail* [11], décrit la PPC comme suit :

> *Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it.*

Le but est d'assigner des valeurs aux différentes variables de manière à ce que les différentes contraintes du problème soient satisfaites. L'ensemble des variables et des contraintes forment un *modèle* représentant le problème.

Le but peut être de trouver [4] :

— une seule solution quelconque satisfaisant toutes les contraintes ;
— toutes ou une partie des solutions ;
— une solution dite optimale qui maximise ou minimise une fonction-objectif qui s'applique sur toutes ou un sous-ensemble des variables.

On trouve plusieurs applications industrielles à ces systèmes, notamment dans le but d'assister les décideurs d'une organisation ou d'une entreprise lors de la prise de décisions qui implique la planification ou l'ordonnancement des opérations [4, 43].

Une gamme variée de problèmes industriels peuvent être modélisés en utilisant la PPC. Cela comprend la planification des trajets des véhicules de livraison (*Vehicle Rou-*

ting) [31], la génération d'horaires pour un ensemble d'employés de la santé [47, 49], l'ordonnancement des processus dans un contexte industriel [3, 9], la planification des opérations de recherche et sauvetage [42], l'affectation d'une ressource telle une piste atterrissage [44] ou du temps pour l'utilisation d'un observatoire [33], la sélection de musique selon des critères spécifiques [46], l'utilisation de la PPC pour retrouver des informations dans une base de données [29, 30], la planification des opérations dans l'industrie forestière [2, 18, 40, 41], etc.

Un logiciel prenant en entrée un modèle et produisant une solution est appelé un *solveur*. Il existe bon nombre de solveurs disponibles au grand public. Très souvent, un solveur supporte plusieurs types de contraintes de base. Il inclut aussi diverses stratégies de recherche qui permettent de guider la recherche d'une solution. En effet, il n'est pas suffisant de simplement déclarer le modèle du problème, il faut généralement aussi indiquer au solveur comment chercher une solution, puisqu'il n'existe pas de stratégie de recherche performante pour tous les problèmes. La configuration du solveur demande, en général, l'expertise d'une personne qui comprend comment bien exploiter le solveur. Le chapitre 1 propose un survol des concepts de bases pour la PPC.

Le Consortium de recherche FORAC regroupe des chercheurs qui utilisent notamment la PPC dans le but de résoudre des problèmes combinatoires vécus par ses partenaires industriels pour la planification des opérations de transformation du bois. Par le passé, d'anciens étudiants de FORAC ont tenté d'intégrer de nouvelles stratégies de recherche *sur mesure* aux solveurs publiquement accessibles, sans grand succès. En effet, bien que l'architecture des solveurs soit souvent conçue de manière à facilement intégrer de nouvelles contraintes et heuristiques, ajouter de nouvelles stratégies de recherche se révèle être plus ardu. Des exemples de stratégies de recherche, créées par un ancien membre de FORAC, qui n'ont pas pu être intégrées facilement dans les solveurs génériques sont les versions parallélisées des stratégies de recherche communes telles que *Depth-First Search* (DFS) [41] et *Limited Discrepancy Search* (LDS) [40]. C'est pour cette raison qu'il a été émis le souhait qu'un solveur puisse intégrer les nouvelles stratégies de recherches créées par les chercheurs de FORAC. Le chapitre 2 décrit ce nouveau solveur. Son architecture permet de facilement intégrer de nouvelles stratégies de recherche. Cela permettra donc d'économiser du temps entre l'idéalisation du nouvel algorithme de recherche et la phase d'essai expérimentale ainsi que d'effectuer plus rapidement du prototypage. De plus, le nouveau solveur permet, comme la majorité des solveurs génériques, de facilement intégrer de nouvelles contraintes et nouvelles heuristiques.

L'industrie de la production du bois d'oeuvre dispose d'un grand nombre de problèmes qui peuvent être résolus à l'aide de la PPC. Un exemple d'un de ces problèmes est la génération de patrons de chargement pour un séchoir à bois. Sécher le bois demande beaucoup de temps (généralement entre 12 et 60 heures [17]). Il est donc nécessaire de soigneusement planifier l'utilisation des séchoirs sur une longue période. La planification des séchoirs sur plusieurs périodes est un sujet traité par Gaudreault et coll. [18] ainsi que Marier et coll. [38]. Par contre, l'industrie forestière évolue dans un contexte de constante compétition où les demandes des clients arrivent parfois tardivement et les besoins changent après la planification initiale de l'utilisation d'un séchoir. C'est pourquoi il est souhaitable d'offrir de la flexibilité aux employés d'une usine en leur permettant de choisir parmi plusieurs patrons de chargement pour remplacer le chargement d'un séchoir déjà planifié dont le démarrage est imminent. La génération d'un patron de chargement est un problème hautement combinatoire. En effet, nous devons respecter des contraintes qui proviennent des limites physiques du séchoir et nous devons nous assurer que les nouveaux patrons de chargement, proposés à l'employé, apportent une contribution minimale quant à la satisfaction des différentes demandes de produits par les clients. À l'aide du solveur proposé au chapitre 2, le chapitre 3 propose une approche PPC dans le but de générer un grand nombre de patrons de chargement alternatifs.

Dans le but de générer avec efficience des solutions pour un problème de PPC, il est critique de déterminer quand et comment réaliser le filtrage des valeurs possibles des variables en fonction des contraintes. Encore une fois, aucune stratégie n'est universelle et il est nécessaire de pouvoir configurer le solveur. Un solveur cherche généralement à effectuer la propagation des contraintes le plus tôt possible. Dans le chapitre 4, nous proposons une nouvelle avenue que nous nommons le *filtrage tardif* (*lazy filtering*). Nous démontrons que, pour le problème du *Balanced Incomplete Block Design* (BIBD) [50], il est possible, en repoussant le filtrage au dernier moment, de conserver le même niveau de filtrage tout en diminuant le temps de résolution. La recherche présentée dans ce chapitre a fait l'objet d'une publication [1] aux Treizièmes journées Francophones de Programmation par Contraintes (JFPC 2017) qui est une conférence avec comité de lecture. Ce chapitre montre, ainsi, comment notre solveur peut être facilement adapté pour intégrer des nouvelles techniques de résolution.

1. Yassine ATTIK, Jonathan GAUDREAULT, Claude-Guy QUIMPER : *Filtrage tardif du BIBD : lorsque la procrastination paie*. Treizièmes Journées Francophones de Programmation par Contraintes (JFPC), 2017.

Chapitre 1

Notions préliminaires

La programmation par contraintes (PPC) est un paradigme qui permet de résoudre des problèmes d'optimisation combinatoire [12]. Il existe deux catégories principales de problèmes que nous pouvons résoudre avec la PPC :

— ***Problème de Satisfaction de Contraintes.*** Le but est de trouver une ou des solutions qui respectent l'ensemble des contraintes.

— ***Problème d'Optimisation sous Contraintes.*** Le but est de trouver une solution qui, non seulement respecte l'ensemble des contraintes, mais qui, en plus, minimise ou maximise une fonction-objectif.

Dans reste de ce mémoire, nous nous concentrons sur la résolution des problèmes de satisfaction de contraintes. Formellement, un problème de satisfaction de contraintes est défini par le tuple $\mathcal{P} = \langle X, dom, C \rangle$. X est un ensemble qui comprend n variables $X = \langle x_1, x_2, ..., x_n \rangle$. $dom(x_i)$ est une fonction qui prend une variable x_i et retourne un ensemble de valeurs pouvant être affectées à cette variable. Il est courant d'utiliser des domaines dont les valeurs appartiennent à l'ensemble des entiers, mais il est aussi possible que les valeurs appartiennent à l'ensemble des réels ou à l'ensemble des nombres binaires. Dans le but de définir un domaine, nous pouvons énumérer spécifiquement toutes les valeurs possibles. Par exemple, c'est utile pour représenter un domaine du genre $\{1, 3, 5\}$. Par contre, il est aussi possible de simplement fournir la valeur minimale, que nous nommons la *borne inférieure*, ainsi que la valeur maximale, que nous nommons la *borne supérieure*, lorsque nous souhaitons représenter un domaine qui peut prendre n'importe quelle valeur entre les bornes (incluant les valeurs des bornes). Nous avons donc un *domaine intervalle*.

Finalement, C est un ensemble de t contraintes $C = \langle C_1, C_2, ..., C_t \rangle$ qui permet de définir des relations entre différentes variables appartenant à un sous-ensemble de X. Ce sous-ensemble est appelé la *portée* d'une contrainte.

Le but de la PPC est de trouver les bonnes valeurs à assigner aux variables de manière à ce que les différentes contraintes soient respectées. Il existe deux types de contraintes :

— **Contraintes à arité fixe.** Ces contraintes sont définies pour ne prendre qu'un nombre fixé d'avance de variables en paramètre. Nous pouvons avoir une arité unaire, binaire, ternaire, etc. Par exemple, la contrainte (1.1) est une contrainte d'arité 2 (binaire). La contrainte prend, en effet, les variables x_1 et x_2 comme paramètres en entrée.

$$x_1 \leq x_2 \tag{1.1}$$

— **Contraintes globales.** Ces contraintes sont définies pour prendre un nombre non déterminé d'avance de variables dans sa portée. Le nombre de variables sur lesquelles la contrainte s'applique est donc aussi un paramètre (la cardinalité de la portée). Un exemple de ce type de contrainte est *AllDifferent*$([x_1, x_2, ..., x_n])$ [1] qui prend un vecteur qui peut contenir autant de variables que souhaitées.

L'ensemble des variables, domaines, ainsi que celui des contraintes forment le *modèle* du problème de PPC. Il est important, pour l'utilisateur, de fournir un bon modèle. En effet, il existe plusieurs manières de modéliser un même problème, mais certains de ces modèles vont permettre au solveur de fournir une solution plus rapidement que d'autres [28]. Par ailleurs, une fois que le modèle est constitué, l'utilisateur doit spécifier quelle stratégie sera employée par le solveur pour chercher une solution.

1.1 Arbres de recherche

L'une des techniques principalement utilisé pour trouver des solutions en PPC est l'utilisation d'un *arbre de recherche* [56]. Un arbre de recherche permet d'affecter des valeurs aux variables de manière structurée. Nous utilisons un algorithme de recherche par retours arrière dans le but d'explorer l'arbre. L'intérêt de ce type d'algorithme est qu'il est considéré comme étant *complet* [65]. Un algorithme complet veut dire que nous avons la garantie que s'il existe une solution, alors nous la trouverons. Nous

1. La contrainte *AllDifferent* permet de s'assurer que les variables de sa portée prennent des valeurs différentes.

avons donc, réciproquement, la garantie que si une solution n'est pas trouvée malgré l'exploration complète de l'arbre, alors il n'existe pas de solution pour le modèle que l'utilisateur a fourni. Un autre intérêt pour ce type d'algorithme de recherche est que nous avons seulement besoin d'une quantité polynomiale d'espace mémoire pour trouver une solution [65]. En effet, nous n'explorons qu'une seule branche de l'arbre à la fois. Puisque souvent un utilisateur ne souhaite pas trouver toutes les solutions [65], alors il est intéressant de constater que l'impact sur la mémoire vive est minimisé.

Dans un arbre de recherche, les noeuds représentent une solution partielle où un certain nombre de variables sont instanciées et les arêtes représentent une nouvelle assignation d'une valeur à une des variables du modèle. Le noeud racine de l'arbre représente un ensemble où aucune variable n'a été instanciée. Lorsque nous atteignons une feuille, alors nous avons une assignation complète de toutes les variables, ce qui correspond à une solution au problème.

Nous illustrons, en premier lieu, une énumération simple de solutions d'un modèle qui ne possède aucune contrainte.

Exemple 1. Soit le problème suivant :

$$dom(x_1) = \{1, 2, 3\} \tag{1.2}$$

$$dom(x_2) = \{1, 2\} \tag{1.3}$$

Le problème génère l'arbre illustré par la figure 1.1. Nous voyons, dans les noeuds, l'état de la solution partielle. Chaque feuille représente une solution au problème. Essentiellement, considérant le fait qu'il n'y avait aucune contrainte, l'arbre généré est une énumération de toutes les combinaisons de valeurs que les variables peuvent prendre.

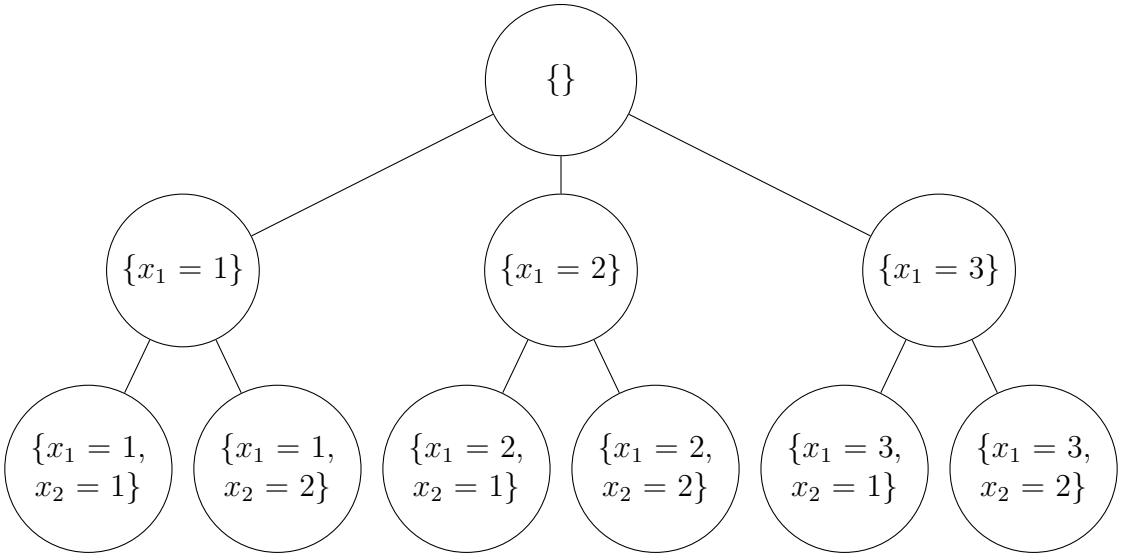


FIGURE 1.1 – Arbre pour l'exemple 1

Voyons maintenant un problème pour lequel il y a des contraintes.

Exemple 2. Soit le problème suivant :

$$x_1 + x_2 = 4 \tag{1.4}$$

$$x_1 > x_2 \tag{1.5}$$

$$5 \leq x_2 + x_3 \leq 6 \tag{1.6}$$

$$dom(x_1) = \{2, 3\} \tag{1.7}$$

$$dom(x_2) = \{1, 2\} \tag{1.8}$$

$$dom(x_3) = \{3, 4, 5, 6\} \tag{1.9}$$

Le problème génère l'arbre illustré par la figure 1.2. Nous voyons que lorsque la variable $x_1 = 2$, il est impossible de trouver une valeur dans $dom(x_2)$ qui permet de respecter la contrainte (1.4) lorsque $Y = 1$ ou encore la contrainte (1.5) lorsque $x_2 = 2$. Nous devons donc retirer de $dom(x_2)$ les valeurs 1 et 2 ce qui fait en sorte que le domaine se retrouve à devenir un ensemble vide. Puisqu'une variable se retrouve avec un domaine vide, il ne sert à rien de continuer l'exploration de l'arbre en deçà de ce noeud. Cette situation est appelée un *échec*. Il faut donc effectuer un retour arrière dans le but d'explorer d'autres sections de l'arbre qui pourraient être plus prometteuses. Nous voyons que nous réussissons à deux reprises à assigner toutes les variables à des valeurs qui respectent toutes les contraintes, ce qui indique que les deux seules solutions au problème sont $\{x_1 = 3, x_2 = 1, x_3 = 4\}$ ainsi que $\{x_1 = 3, x_2 = 1, x_3 = 5\}$.

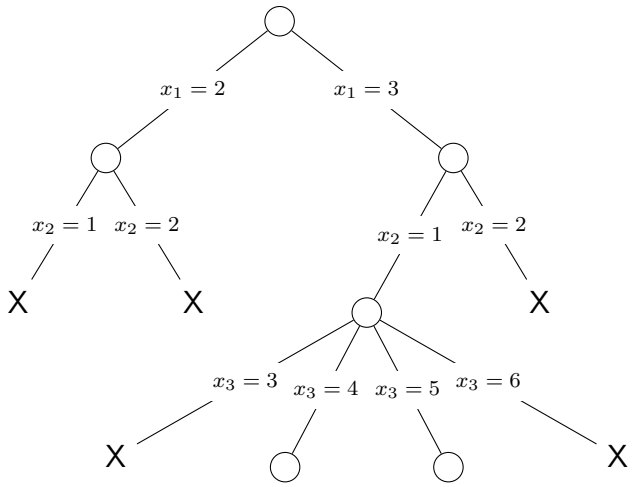


FIGURE 1.2 – Arbre pour l'exemple 2

1.2 Filtrage et propagation

L'exploration d'un arbre de recherche nécessite l'énumération des $\prod_{i=1}^{n} |dom(x_i)|$ combinaisons possibles, ce qui est une approche peu efficient (et impossible) en pratique. Dans le but de faciliter la recherche d'une solution, il est possible d'effectuer du *filtrage*. Le filtrage permet de réduire le nombre de branches à explorer lors de l'exploration d'un arbre de recherche. À chaque fois qu'il y a une modification dans le domaine d'une variable, nous pouvons chercher à éliminer des valeurs des variables lorsque nous savons que leurs affectations ne respectent pas, au minimum, l'une des contraintes. Dans le but de bien définir le filtrage, il faut définir les notions de *support* et de *cohérence*.

Nous définissons deux types de supports :

— **Support de domaine.** Lorsque nous pouvons affecter à chaque variable x de la portée d'une contrainte C une valeur appartenant à $dom(x)$ et que ces affectations satisfont C, alors nous disons que nous avons un *support de domaine*.

— **Support d'intervalle.** Lorsque nous pouvons affecter à chaque variable x de la portée d'une contrainte C une valeur appartenant à l'intervalle $[min(dom(x))..max(dom(x))]$ et que ces affectations satisfont C, alors nous disons que nous avons un *support d'intervalle*.

Une valeur qui n'a pas de support ne participe à aucune solution réalisable du problème. Le filtrage consiste à retirer ces valeurs du domaine dans le but de réduire la taille de

l'arbre de recherche. Nous tentons alors d'établir un *niveau de cohérence*. Il existe différents niveaux de cohérence que nous pouvons atteindre en fonction de l'intensité de filtrage que nous mettons en place. Nous présentons les deux les plus utilisés [58] :

— **Cohérence de domaine.** Considérant une contrainte C, nous disons qu'elle est *cohérente de domaine* si chaque valeur dans le domaine de chaque variable dans la portée de C a un support de domaine [65].
— **Cohérence de borne.** Considérant une contrainte C, nous disons qu'elle est *cohérente de borne* si la valeur minimale ainsi que la valeur maximale dans le domaine de chaque variable dans la portée de C ont un support d'intervalle [27].

La cohérence de domaine est un niveau de cohérence plus fort, mais aussi plus complexe à établir qu'une cohérence de borne. C'est pourquoi, pour chaque type de contrainte, il convient de spécifier un ou plusieurs *algorithmes de filtrage* permettant d'établir le niveau de cohérence désiré. Ces algorithmes de filtrages sont déclenchés dynamiquement pendant la recherche lorsque survient un *événement* déclencheur. Il existe plusieurs événements qui peuvent déclencher le processus de filtrage. Les événements principaux sont [58, 59] :

— l'assignation d'une valeur à une variable ;
— l'augmentation d'une borne inférieure ;
— la diminution d'une borne supérieure ;
— le retrait d'une valeur du domaine d'une variable.

Lorsqu'un événement se produit, les contraintes qui ont dans leur portée la variable modifiée voient déclencher leurs algorithmes de filtrage. Il n'est pas rare que plusieurs événements soient déclenchés en même temps [59]. Par exemple, si nous avons le domaine d'une variable qui est présentement $\{1, 2\}$ et que nous retirons la valeur 1, alors nous déclenchons les événements d'augmentation d'une borne inférieure, l'assignation d'une valeur à une variable (le domaine devient $\{2\}$) ainsi que le retrait d'une valeur. Il faut donc avertir les différentes contraintes que trois événements se sont produits sur la variable. Les algorithmes de filtrage n'ont pas l'obligation de se déclencher à tous les événements. C'est au concepteur de l'algorithme de filtrage de déterminer le traitement à effectuer pour un événement particulier. Puisque les algorithmes de filtrage modifient les variables dans la portée de leur contrainte, d'autres événements vont se produire, ce qui peut déclencher le filtrage pour d'autres contraintes. Cela entraîne donc le déclenchement de plusieurs algorithmes de filtrage de manière répétitive que nous appelons la

propagation des contraintes. Cette propagation va se poursuivre jusqu'à ce qu'il ne soit plus possible de modifier davantage les différents domaines du modèle (lorsque toutes les contraintes deviennent cohérentes). Nous atteignons donc un *point fixe* [5, 58, 59].

Il est possible, dans certaines situations, qu'il existe plusieurs algorithmes de filtrage qui assurent le même *niveau de filtrage* sur les différentes variables du modèle [57]. Avoir le même niveau de filtrage signifie que nous visitons les mêmes solutions ordonnées de la même manière tout en ayant un nombre d'échecs identique lors de l'exploration des arbres. Cela veut donc dire que nous avons exploré le même arbre de recherche pour une version ou une autre. Dans ces cas-là, il est préférable d'utiliser l'algorithme le plus rapide.

1.3 Heuristiques de branchement

Dans la section sur les arbres de recherche (section 1.1), nous instancions les variables dans l'ordre de déclaration et nous choisissons les valeurs en ordre croissant. Il est par contre possible d'influencer la topologie de l'arbre de recherche en utilisant un autre critère. En réalité, le but est de visiter les branches de l'arbre les plus prometteuses le plus tôt possible. Pour ce faire, nous utilisons une heuristique de choix de variables pour influencer l'ordre de sélection des variables et nous utilisons une heuristique de choix de valeurs pour influencer l'ordre de sélection des valeurs dans les variables. Ce qui est intéressant c'est que nous ne sommes pas limités qu'à un seul choix pour l'un ou l'autre type d'heuristique. Un utilisateur peut même créer des heuristiques personnalisées si son problème le requiert. La bonne sélection des heuristiques appropriées dépend de l'expérience des utilisateurs. Par contre, comme Liberatore [36] le démontre, le simple fait de décider avec certitude si une variable est la première variable dans un ordonnancement optimal des variables est au moins aussi compliqué que de décider si le problème de PPC a une solution [65]. Trouver un ordonnancement optimal des valeurs est aussi très clairement au moins aussi complexe du fait que, si une solution existe, alors un ordonnancement optimal des valeurs pourrait être utilisé pour trouver rapidement une solution [65]. C'est pourquoi l'expertise de l'utilisateur est essentielle lorsque vient le temps de décider quelles heuristiques utiliser pour résoudre un problème.

1.3.1 Heuristiques de choix de variables

Nous pouvons distinguer deux types d'heuristiques de choix de variables.

Nous avons les heuristiques à ordonnancement statique. Un ordonnancement statique indique que l'ordre dans lequel les différentes variables seront instanciées lors de la recherche est connu d'avance (avant de débuter la recherche dans l'arbre). Par exemple, dans l'exemple 2 de la section 1.1, nous utilisons l'ordre x_1, x_2, x_3 pour instancier les variables. Par contre, nous aurions pu aussi utiliser l'ordre x_3, x_2, x_1 ou x_2, x_3, x_1 ou un autre pour instancier les variables. C'est au modélisateur de le déterminer selon ce qu'il juge être le plus performant.

Il existe aussi des heuristiques à ordonnancement dynamique qui modifient l'ordre d'instanciation des variables selon les événements qui se produisent lors de la recherche dans l'arbre. Par exemple, Golomb et Baumert [20] propose une heuristique dynamique qui choisit la variable qui contient le plus petit nombre de valeurs restant dans son domaine. Une autre technique, proposée par Haralick et Elliott [23], est de sélectionner la variable qui a le plus de chance de mener rapidement à un échec et donc forcer un retour arrière.

1.3.2 Heuristiques de choix de valeurs

Lorsqu'une variable est sélectionnée par l'heuristique de choix de variables, il faut choisir quelle valeur lui assigner. Certaines des techniques qui existent sont de sélectionner les valeurs dans un ordre croissant, en ordre décroissant ou de prendre une valeur au hasard. Un exemple d'une technique plus évoluée est la technique proposée par Dechter et Pearl [7] qui est une heuristique statique qui permet d'estimer, pour un sous-arbre donné, le nombre de solutions. Cela permet donc de choisir la valeur avec le meilleur potentiel en premier lieu. Dans la même vague d'idées, Pesant et coll. [48] développent des algorithmes qui permettent aussi d'estimer le nombre de solutions, mais en utilisant l'information provenant de chaque contrainte et non l'ensemble du problème.

1.3.3 Utilisation d'un algorithme vorace comme heuristique de branchement

Un algorithme vorace est un algorithme qui permet, si c'est applicable au problème traité, de trouver une solution optimale dans un temps polynomial. L'algorithme fait une succession de meilleurs choix dans tous ceux possibles jusqu'à ce que le problème soit résolu. Les choix que nous faisons doivent répondre à trois caractéristiques [35] :

— **Réalisable.** Il faut satisfaire les contraintes du problème.

— **Optimal localement.** Nous choisissons le meilleur choix local parmi tous les choix possibles.

— **Irrévocable.** Lorsque nous faisons un choix, il n'est plus modifiable par la suite.

Une utilisation possible de l'algorithme vorace est pour le *problème de rendre la monnaie*. Le but est de remettre la monnaie avec le minimum de pièces. Dans l'exemple tiré de [35], une personne souhaite remettre 48 cents avec des pièces de 25 cents, 10 cents, 5 cents et 1 cent énumérées en ordre décroissant. Nous voyons qu'il est très facile de déterminer quelles sont les pièces à remettre. Nous remettons 25 cents une fois, 10 cents deux fois et 1 cent trois fois. C'est la manière la plus optimale (minimisant le nombre de pièces remises) de remettre le change pour 48 cents. L'algorithme est très simple à appliquer puisque nous avons juste besoin de trouver, pour chaque étape, la pièce de valeur maximum à remettre au client jusqu'à ce que le change soit remis au complet. Un exemple où cela ne fonctionne pas, encore une fois tiré de [35], est de remettre 30 cents alors que nous n'avons que des pièces de 25 cents, 10 cents et 1 cent. Nous nous retrouvons à remettre, selon l'algorithme vorace, une pièce de 25 cents et cinq pièces de 1 cent. Par contre, la solution optimale est plutôt de remettre trois pièces de 10 cents. C'est pourquoi il faut vérifier que l'algorithme vorace est bien applicable au cas qu'on souhaite traiter si le but est de trouver une solution optimale, et non pas simplement un assez bon choix. Nous l'utilisons dans le chapitre 3 où nous exploitons ce type d'algorithme pour trouver des patrons de chargement pour un séchoir à bois.

1.4 Stratégies de recherche

Bien que la topologie de l'arbre de recherche soit déterminée à l'aide des différentes heuristiques de choix de variables et de valeurs, il faut également déterminer comment procéder à son exploration. Nous présentons différentes techniques utilisées en PPC.

1.4.1 Fouille en profondeur

Une stratégie très simple dans sa mise en oeuvre est la fouille en profondeur (*Depth-First Search* (DFS)). Lorsque nous explorons un arbre, la première variable à être sélectionnée par l'heuristique de choix de variables représente le niveau racine. La variable va être instanciée à la première valeur de l'heuristique de choix de valeurs. Par la suite, nous sélectionnons la deuxième variable de l'heuristique de choix de variables et nous y

assignons aussi la première valeur choisie par l'heuristique de choix de valeurs. Nous continuons ainsi jusqu'à ce que toutes les variables soient instanciées (toujours en ordre de l'heuristique de choix de variables) auquel cas, nous avons une solution, ou jusqu'à ce que le domaine d'une variable soit vidé de ses valeurs du fait de la propagation des contraintes (donc un échec). Si nous cherchons également d'autres solutions ou que nous avons un échec, il faut revenir en arrière jusqu'à retrouver une variable pour laquelle nous avons une valeur que nous n'avons pas essayée encore (donc la deuxième valeur sélectionnée par l'heuristique de choix de valeurs). Lorsque c'est le cas, alors nous assignons cette valeur à la variable et nous redescendons dans l'arbre à nouveau. Nous continuons ainsi tant et aussi longtemps que nous n'avons pas rempli la condition d'arrêt de l'exploration de l'arbre ou, encore, que l'arbre n'ait pas été exploré au complet. Essentiellement, DFS cherche à essayer toutes les valeurs possibles (dans l'ordre de l'heuristique de choix de valeurs) des variables les plus basses dans l'arbre avant de remettre en question les choix de valeurs des variables plus hautes. Nous pouvons aussi dire que nous sélectionnons toujours, en priorité, le noeud enfant non exploré le plus à gauche. La figure 1.3 montre l'exploration d'un arbre avec DFS.

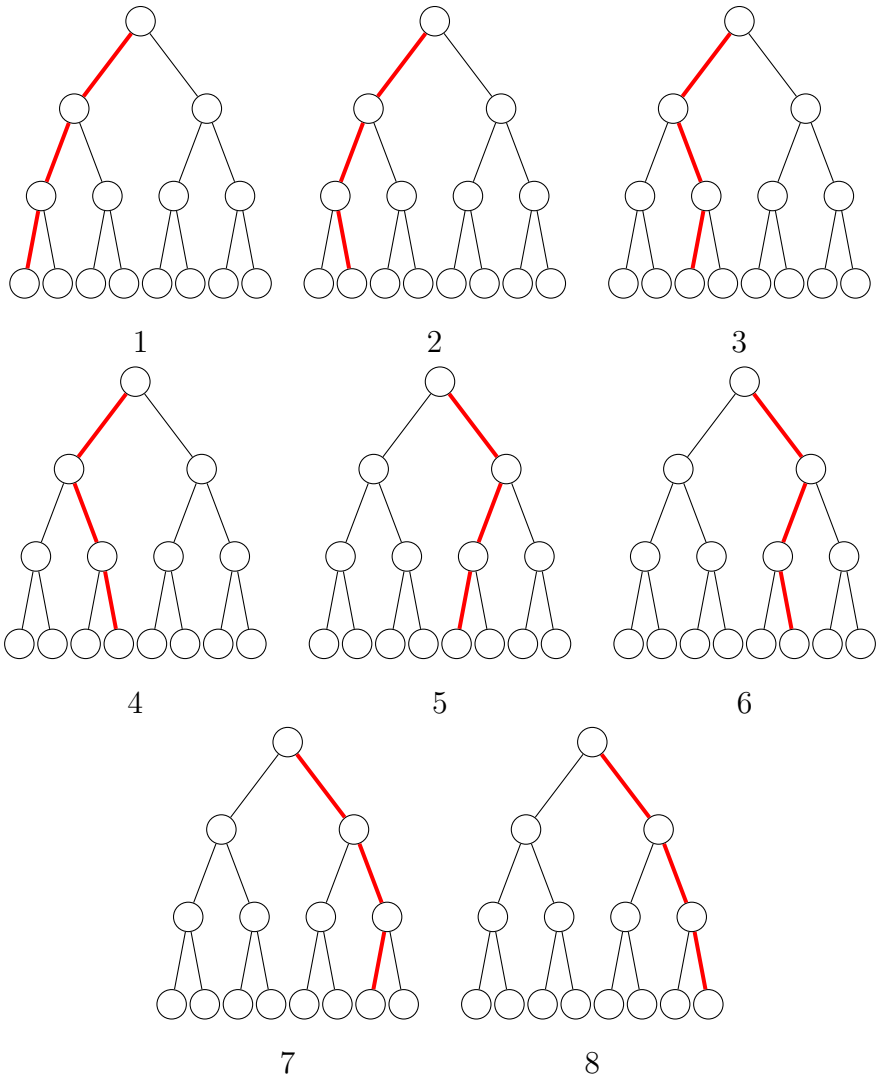


FIGURE 1.3 – Exploration d'un arbre binaire de 3 variables avec DFS.

1.4.2 Stratégies de recherche basées sur les déviations

***Limited Discrepancy Search* (LDS)**

L'un des grands problèmes de DFS est que nous nous retrouvons à passer beaucoup de temps à explorer une même section de l'arbre avant de passer aux autres. Lorsque l'arbre de recherche est de petite taille ou encore que le but est d'explorer l'ensemble de l'arbre (par exemple, dans le cas où nous souhaitons démontrer qu'il n'existe pas de solution), alors DFS est viable comme option considérant sa simplicité de mise en oeuvre et sa rapidité d'exécution. Par contre, très souvent, l'arbre à explorer est gigantesque et le but n'est que de trouver qu'une ou un petit nombre de solutions rapidement. Harvey et Ginsberg sont les premiers à définir et exploiter les *déviations*

pour explorer un arbre de recherche [24, 25]. Nous parlons d'une déviation lorsque la stratégie de recherche sélectionne une branche correspondant à une valeur qui n'est pas préconisée par l'heuristique de choix de valeurs. L'intuition derrière cette stratégie de recherche est que, en supposant que l'utilisateur fournisse de très bonnes heuristiques de branchement [2], la solution devrait se retrouver là où il n'y a pas de déviation. Cela veut dire que nous choisissons, pour chaque variable, la première valeur suggérée par l'heuristique de choix de valeurs. Si c'est le cas, alors nous avons trouvé une solution avec 0 déviation. Si ce n'est pas le cas, alors nous recommençons la recherche à la racine de l'arbre, mais cette fois-ci, en permettant 1 déviation. Cela veut dire que nous permettons, lorsqu'on explore une branche de l'arbre, de faire, dans une des variables, un seul choix qui ne respecte pas l'heuristique de choix de valeurs (donc de choisir une valeur qui n'est pas la préférée de l'heuristique). Nous explorons tout l'arbre qui a jusqu'à 1 déviation et si nous ne trouvons toujours rien, alors nous retournons à la racine et nous permettons jusqu'à 2 déviations. Nous continuons ce processus-là jusqu'à ce que nous trouvions une solution ou jusqu'à ce que toutes les déviations aient été épuisées (dans un arbre binaire, nous avons $|X|$ déviations possibles).

Improved Limited Discrepancy Search (ILDS)

LDS a le désavantage d'explorer plusieurs fois les mêmes feuilles à mesure que le nombre de déviations permises augmente. En effet, si nous sommes arrivés à explorer les feuilles qui ont 2 déviations, l'algorithme va explorer les feuilles de 0, 1 et 2 déviations. Cela amène une redondance inutile qui réduit l'efficience de la recherche dans l'arbre. C'est ce qui a amené Korf [32] à définir ILDS qui est une version améliorée de LDS où seules les feuilles de la $k^{ème}$ déviation sont explorées. Considérant l'amélioration que ILDS apporte, il est courant, dans la littérature scientifique, d'indiquer l'utilisation de LDS alors que c'est ILDS qui est véritablement implémenté. Les algorithmes 1 ainsi que 2 (tirés de [32]) permettent de montrer les différences entre les pseudo-codes de LDS et ILDS pour une itération. Le paramètre *Noeud* fait référence au noeud courant (en débutant par le noeud racine), k indique la déviation des feuilles recherchées et *Profondeur* indique la profondeur d'un arbre binaire qui correspond à $|X|$. La figure 1.4 permet de montrer, pour chaque itération de l'algorithme de ILDS, la recherche des

2. Moisan et coll. [40] montrent que dès que les heuristiques de choix de variables/valeurs font mieux que ce qu'on aurait avec le hasard, LDS est meilleur que DFS.

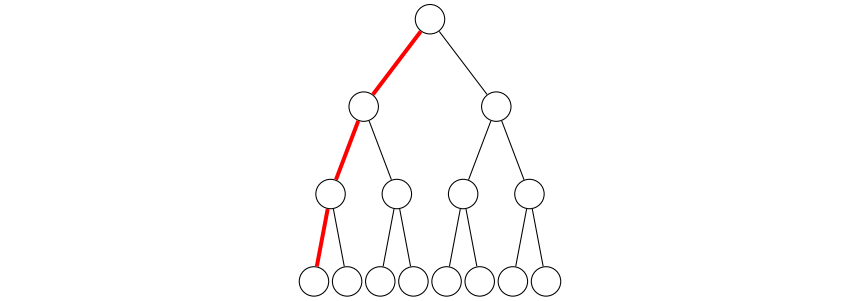
feuilles ayant la déviation k.

Algorithme 1 : LDS(*Noeud*, k) (Tiré de [32])

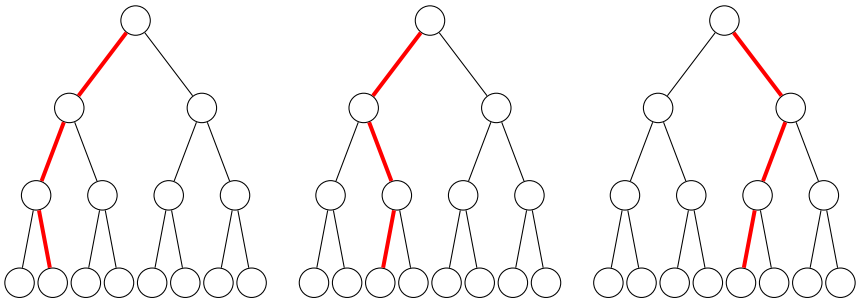
```
si Noeud est une feuille alors
    retourner Noeud
LDS(enfant-gauche(Noeud), k)
si k > 0 alors
    LDS(enfant-droite(Noeud), k − 1)
```

Algorithme 2 : ILDS(*Noeud*, k, *Profondeur*) (Tiré de [32])

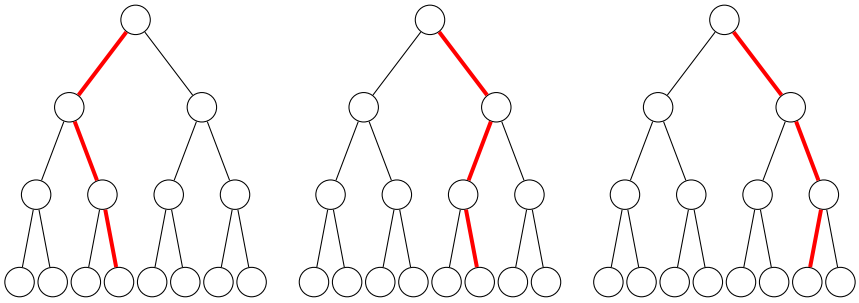
```
si Noeud est une feuille alors
    retourner Noeud
si Profondeur > k alors
    ILDS(enfant-gauche(Noeud), k, Profondeur − 1)
si k > 0 alors
    ILDS(enfant-droite(Noeud), k − 1, Profondeur − 1)
```



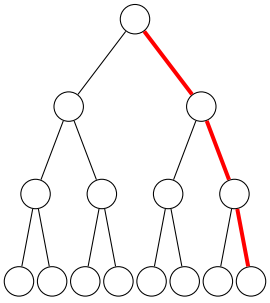
(a) Itération 1 ($k = 0$)



(b) Itération 2 ($k = 1$)



(c) Itération 3 ($k = 2$)



(d) Itération 4 ($k = 3$)

FIGURE 1.4 – Exploration d'un arbre binaire de 3 variables avec ILDS. La variable k indique la déviation permise.

1.4.3 Stratégies de recherche parallélisés

Les stratégies de recherche DFS ainsi que LDS possèdent des versions parallèles. En effet, Moisan et coll. ont effectué la parallélisation de ces algorithmes qui sont connus respectivement sous les noms de *Parallel Depth-First Search* (PDFS) [41] ainsi que *Parallel Limited Discrepancy-based Search* (PLDS) [3] [40]. Le but de ces algorithmes est d'être utilisés dans un contexte massivement parallélisé où plusieurs processeurs explorent simultanément des sections différentes de l'arbre de recherche. Pour y arriver, les algorithmes de recherche prennent, en plus des paramètres standards des algorithmes en version centralisés, les paramètres qui donnent le nombre de processeurs total ainsi que l'identifiant du processeur assigné au travailleur. Ces informations supplémentaires permettent à l'algorithme de déduire si, à partir d'un noeud, le sous-arbre contient des feuilles qui appartiennent au travailleur courant. Si c'est le cas, alors l'algorithme va poursuivre l'exploration de la branche, sinon, l'algorithme va trouver une autre section à explorer.

Nous montrons l'arbre de recherche pour PDFS (figure 1.5) ainsi que PLDS (figure 1.6) si deux processeurs travaillent en parallèle. Dans les deux cas, chaque processeur explore les feuilles qui lui sont assignées dans l'ordre croissant si ces mêmes feuilles avaient été explorées par la version centralisée de l'algorithme (ordre indiqué dans les feuilles).

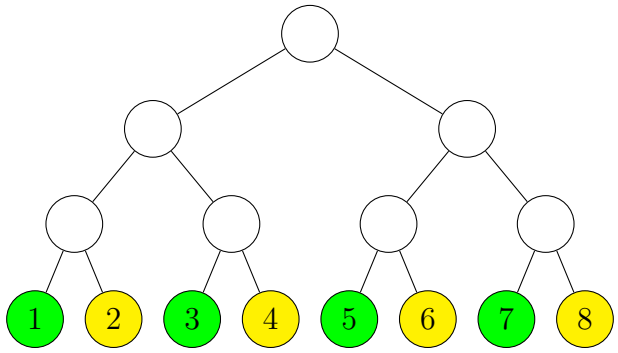


FIGURE 1.5 – Arbre exploré par PDFS où les noeuds en vert sont visités par le processeur 1, les noeuds en jaune sont visités par le processeur 2 et les noeuds en blanc sont visités par les deux processeurs. Les feuilles contiennent l'ordre d'exploration en version centralisé de l'algorithme.

3. À l'origine, dans [40], l'algorithme était nommé PDS, mais il a été renommé PLDS dans [41] dans le but d'assurer la clarté.

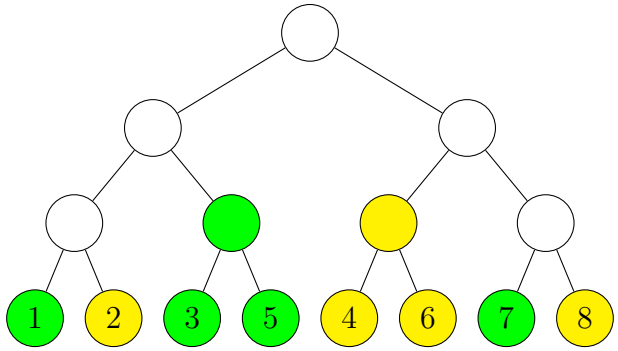


FIGURE 1.6 – Arbre exploré par PLDS où les noeuds en vert sont visités par le processeur 1, les noeuds en jaune sont visités par le processeur 2 et les noeuds en blanc sont visités par les deux processeurs. Les feuilles contiennent l'ordre d'exploration en version centralisé de l'algorithme.

1.5 Solveur

Un solveur est souvent disponible sous forme d'une librairie de programmation qui permet de résoudre des problèmes combinatoires. Il prend en entrée les variables, domaines et contraintes ainsi que les heuristiques et stratégies de recherche choisies par l'utilisateur. Il retourne des solutions si le solveur en trouve. Il existe plusieurs solveurs de PPC disponibles au grand public. Parmi les plus connus, nous avons Choco [54] qui est implémenté en Java, Gecode [63] qui est implémenté en C++, Chuffed [62] qui est implémenté en C++, OscaR [64] qui est implémenté en Scala ainsi que OR-Tools [21] qui est implémenté en C++, mais qui peut être utilisé avec Python, Java ou C#. Bien qu'ils viennent avec un nombre varié de contraintes et d'heuristiques plus ou moins standards, l'architecture des solveurs permet aux utilisateurs de facilement pouvoir ajouter leurs propres contraintes. En effet, dans certains problèmes d'ordre industriels, il est nécessaire de créer des heuristiques et contraintes personnalisées.

1.6 Bris de symétries

En PPC, les symétries causent souvent une perte de temps lors de l'exploration des arbres de recherche. Nous parlons de symétries lorsqu'il est possible de générer une solution équivalente à une autre simplement en effectuant des permutations de valeurs ou de variables. Elles deviennent de plus en plus problématiques au fur et à mesure que l'espace des solutions est grand. En effet, beaucoup de temps est perdu à retrouver plusieurs fois une solution équivalente ou, pire encore, à explorer des parties de

l'arbre de recherche symétriques à des parties que nous savons déjà ne pas mener à des solutions [66].

Exemple 3. L'exemple donné dans [45] est un problème combinatoire où nous devons placer neuf reines et un roi de chaque couleur de manière à ce qu'aucune pièce ne se retrouve sur la même ligne (rangée, colonne ou diagonale) qu'une reine de la couleur opposée. C'est une variante du problème *Peaceably Coexisting Armies of Queens* où le but est de placer le maximum de pièces pour former deux armées de taille égale de reines sur l'échiquier sans qu'aucune reine ne puisse attaquer une reine adverse [60].

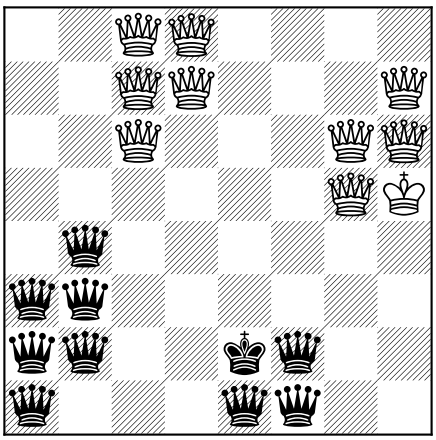


FIGURE 1.7 – Unique solution au problème de l'exemple 3 (Tiré de [45])

Bien que la solution de la figure 1.7 soit unique, il est possible de trouver des solutions symétriques en échangeant les couleurs des pièces, en tournant l'échiquier par n'importe quel multiple de 90 degrés, en effectuant une réflexion par rapport à l'axe horizontal, vertical ou les deux diagonales [45]. Ces opérations donnent 16 symétries possibles incluant l'opération identité [4] [45]. Cet exemple permet de bien comprendre pourquoi il est important d'éliminer le mieux possible les sources de symétries. En effet, ces solutions symétriques peuvent très facilement être obtenues en utilisant des algorithmes qui s'exécutent en temps polynomial. Il est donc souhaitable que le solveur ne se concentre qu'à trouver des solutions qui sont véritablement difficiles. De plus, les symétries qui existent sur les solutions existent aussi sur les non-solutions. Par exemple, si deux reines adverses s'attaquent, elles s'attaqueront encore même si nous tournons l'échiquier à 90

4. L'opération identité est l'opération dans laquelle nous laissons simplement les pièces où elles sont [45].

degrés. C'est vrai aussi pour l'assignation de couleurs. Nous savons qu'il n'est pas utile d'échanger les couleurs des pièces adverses qui s'attaquent puisque le résultat restera le même. Ces symétries sur des assignations que nous savons ne pas mener vers des solutions forcent plusieurs retours arrière (donc perte de temps) qui sont évitables si nous appliquons les bris de symétries appropriés.

1.6.1 Utilisation de la contrainte d'ordre lexicographique pour briser des symétries

Les types de symétries qui arrivent souvent lorsqu'une solution est représentée par un tableau de dimension $l \times c$, où chaque cellule du tableau est associée à une variable, sont les symétries de lignes et les symétries de colonnes. Une symétrie de lignes veut dire que nous pouvons permuter n'importe quelles des l lignes pour trouver une autre solution valide ($l!$ permutations possibles). Une symétrie de colonnes veut dire que nous pouvons permuter n'importe quelles des c colonnes pour trouver une autre solution valide ($c!$ permutations possibles).

Exemple 4. Considérons l'exemple donné par Flener et coll. [10] pour illustrer les symétries de lignes et colonnes. Nous avons une matrice de 4 lignes et 3 colonnes (12 variables) où chaque variable a un domaine appartenant à [0, 1]. Les contraintes sont que $\sum_{ij} x_{ij} = 7 \ \forall i, j$ ainsi que $\sum_i x_{ij} \cdot x_{ik} \leq 1 \ \forall j, k \ \textit{tels que} \ j < k$. Ce problème possède des symétries de lignes et de colonnes.

L'une des solutions à ce problème est :

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

FIGURE 1.8 – Une solution à l'exemple 4 (Tiré de [10])

Une technique utilisée pour éliminer ces symétries est l'utilisation de la contrainte d'ordre lexicographique [13] qui est une contrainte globale qui reçoit en paramètre deux vecteurs de variables et qui s'assure que le premier vecteur soit lexicographiquement plus petit que le deuxième. Cette contrainte impose l'équivalent de l'ordonnancement des mots dans un dictionnaire. Si nous reprenons l'exemple 4, nous pouvons ajouter

des contraintes d'ordre lexicographique entre les différentes colonnes comme montre le résultat à la figure 1.9.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

FIGURE 1.9 – Ordre lexicographique appliqué sur les colonnes pour l'exemple 4 (Tiré de [10])

Nous pouvons aussi ajouter des contraintes d'ordre lexicographique entre les lignes pour donner le résultat de la figure 1.10.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

FIGURE 1.10 – Ordre lexicographique appliqué sur les colonnes pour l'exemple 4 (Tiré de [10])

Grâce à l'ajout de l'ordre lexicographique entre les lignes et entre les colonnes, nous éliminons plusieurs solutions et non-solutions symétriques ce qui permet de réduire d'avantage la taille de l'arbre de recherche.

Chapitre 2

Création d'un nouveau solveur

Dans le contexte des travaux de recherche du Consortium FORAC, nous avions besoin d'un solveur facilement modifiable dans le but d'intégrer de nouvelles idées. Dans ce chapitre, nous présentons un nouveau solveur créé pour les besoins de FORAC. C'est ce solveur qui sera utilisé pour l'ensemble du mémoire. Le solveur permet de facilement faire du prototypage pour de nouvelles stratégies de recherche. Il est arrivé à quelques reprises que des chercheurs souhaitent intégrer de nouvelles stratégies de recherche dans les solveurs disponibles au grand public (voir la section 1.5) sans grand succès. Les algorithmes de parallélisation de Moisan et coll. [40, 41] constituent un exemple de stratégies de recherche que nous souhaiterions pouvoir intégrer facilement dans le nouveau solveur.

Ce solveur permettra aussi d'intégrer un nouvel événement de propagation qui se déclenche lors de la sélection d'une variable (*filtrage tardif*) qui sera décrit en détail au chapitre 4. Ce solveur a été implémenté en C# ce qui permet une utilisation aisée pour les étudiants ainsi que les professionnels de recherche de FORAC habitués avec les langages de programmation .NET. L'architecture du logiciel, présenté dans ce chapitre, est à la base similaire à celui de plusieurs solveurs de PPC. Dans la section 2.1, nous montrerons le diagramme de classe du solveur et expliquerons les différentes classes ainsi que leurs responsabilités. Dans la section 2.2, nous montrerons les différents diagrammes de séquences pour donner un aperçu de l'interaction entre les différentes classes. Dans la section 2.3, nous conclurons ce chapitre.

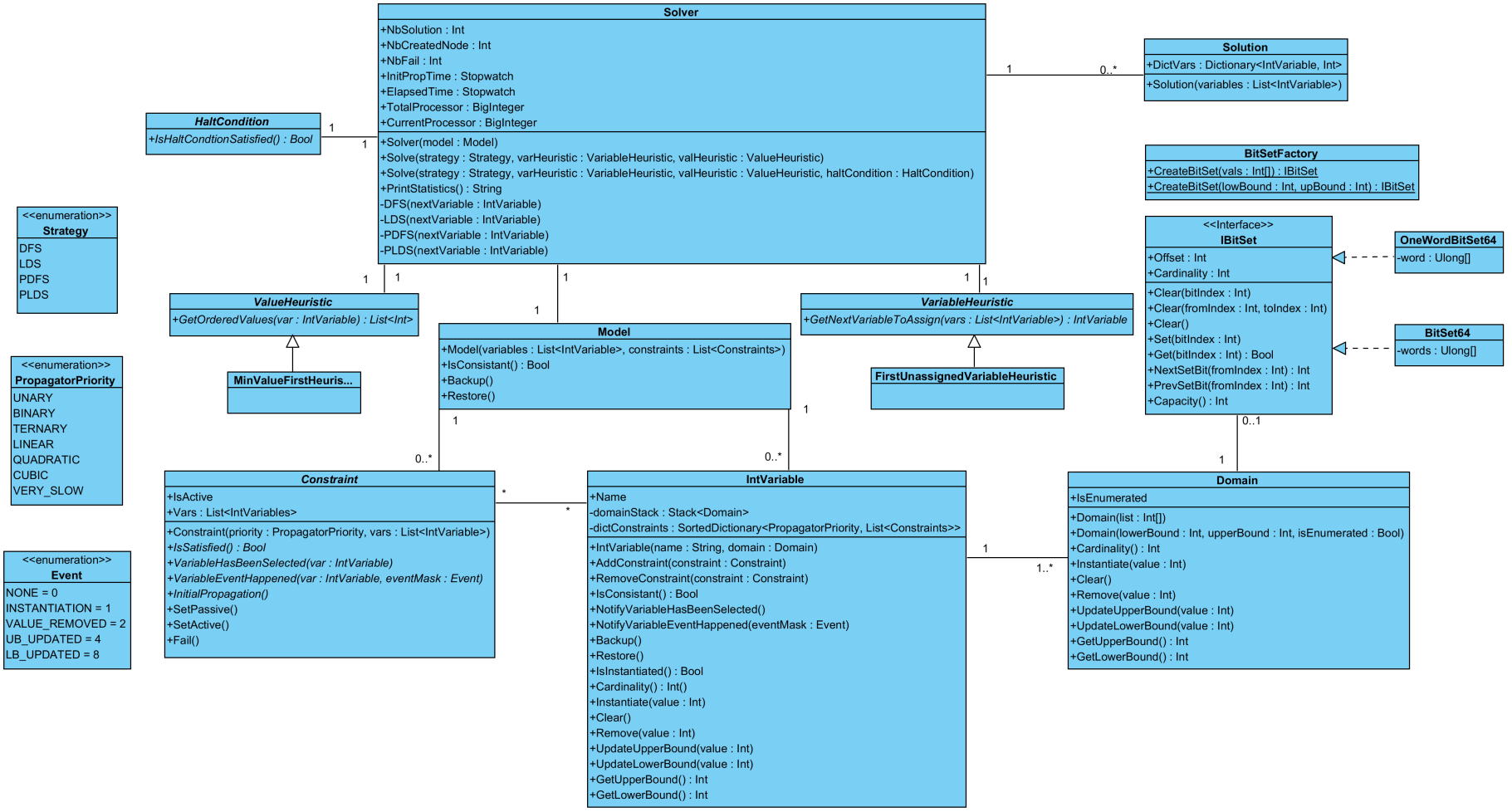


FIGURE 2.1 – Diagramme de classe du solveur

2.1 Présentation du solveur

2.1.1 Diagramme de classe du solveur

La figure 2.1 montre le diagramme de classe du solveur. Compte tenu des objectifs, il est important que son architecture puisse facilement être modifiable de manière à prendre en considération des besoins futurs des utilisateurs. En effet, il arrive souvent que les choix de base offerts par les solveurs génériques en termes d'heuristiques de choix de valeur ou variable ainsi que de contraintes ne soient pas suffisants. Le solveur doit donc pouvoir prendre en considération de nouvelles implémentations. La facilité d'ajout de fonctionnalités est une caractéristique recherchée pour les solveurs. Nous présentons chaque classe du diagramme pour bien comprendre leurs responsabilités dans le fonctionnement du solveur.

Solver

Solver est la classe principale qui permet la résolution d'un problème de PPC. C'est avec cette classe qu'un programmeur va interagir principalement. Elle contient les divers algorithmes des stratégies de recherche disponibles à l'utilisateur. C'est donc le moteur du solveur qui permet de résoudre les problèmes donnés. Il faut obligatoirement lui fournir un *Model* à l'instanciation. Nous pouvons aussi lui fournir d'autres éléments. Par exemple, dans le cas d'une résolution qui utilise une stratégie de recherche parallélisée, il faut aussi lui fournir le nombre total de processeurs (*TotalProcessor* qui est à 1 par défaut) ainsi que le numéro du processeur courant (*CurrentProcessor* qui est à 0 par défaut). Une fois ces informations données, l'utilisateur a seulement besoin d'appeler la méthode *Solve* en y spécifiant la stratégie de recherche souhaitée (une de celles disponibles dans l'énumération *Strategy*), un *ValueHeuristic* ainsi qu'un *VariableHeuristic*. Une autre signature de la méthode *Solve* permet aussi de fournir un *HaltCondition* sans quoi la recherche s'effectuera sur l'arbre au complet. Les méthodes privées *DFS*, *LDS*, *PDFS* ainsi que *PLDS* représentent les implémentations des stratégies de recherche disponibles. La classe *Solver* contient aussi la liste de toutes les solutions retrouvées à n'importe quel moment de l'exploration de l'arbre de recherche. À la fin de la recherche, l'utilisateur pourra récupérer, en plus de solutions trouvées, la valeur *NbSolution* pour connaître le nombre de solutions trouvées, *NbCreatedNode* pour connaître le nombre de noeuds créés pendant l'exploration, *NbFail* pour connaître le nombre d'échecs qui ont forcé des retours arrière, *InitPropTime* ainsi que *ElapsedTime*[1] sont des compteurs qui

1. *InitPropTime* et *ElapsedTime* sont de type *Stopwatch* qui est une classe de C# qui permet de calculer précisément le temps entre le moment où on débute le calcul et la fin.

donnent, pour le premier, le temps pour la propagation initiale des contraintes et, pour le deuxième, le temps total pour l'exploration de l'arbre et *PrintStatistics* qui permet d'imprimer à l'écran les valeurs des différents compteurs du solveur après une recherche.

HaltCondition

HaltCondition est une classe abstraite qui permet d'implémenter ses propres conditions d'arrêt pour mettre fin à une recherche dans un arbre. Nous définissons un certain nombre de conditions d'arrêt comme, par exemple, lorsque nous trouvons un certain nombre de solutions (*NbSolutionHaltCondition*) ainsi que lorsque nous avons dépassé un certain temps de résolution donné par l'utilisateur (*TimeHaltCondition*). Nous avons aussi des *métaconditions d'arrêt* qui permettent de prendre deux conditions d'arrêt et de mettre fin à la recherche si les deux conditions sont rencontrées (*AndHaltCondition*) ou si l'une des deux est rencontrée (*OrHaltCondition*). Pour ce faire, nous utilisons le patron de conception *Composite* [15]. Un utilisateur peut implémenter une condition d'arrêt personnalisée en créant une nouvelle classe qui hérite de *HaltCondition*. Il faut, dans ce cas là, s'assurer de bien implémenter la méthode *IsHaltConditionSatisfied* qui retourne vrai si la condition d'arrêt a été rencontrée (ce qui met fin à la recherche) ou faux sinon.

Solution

Le solveur enregistre les diverses solutions retrouvées, lors de l'exploration, dans la liste *Solution*. *Solution* contient un dictionnaire qui énumère pour chaque variable (*IntVariable*) la valeur assignée.

ValueHeuristic

ValueHeuristic est une classe abstraite qui permet d'implémenter ses propres heuristiques de choix de valeur. Une heuristique de choix de valeur déjà comprise dans le solveur est *MinValueFirstHeuristic* qui retourne la liste des valeurs disponibles d'un domaine ordonnée en ordre croissant pour la variable *var* en paramètre. Un utilisateur, qui souhaite implémenter sa propre heuristique personnalisée, doit implémenter la méthode *GetOrderedValues* qui prend, en entrée, la variable pour laquelle le solveur souhaite connaître l'ordre de choix de valeur et retourner une liste d'entiers où les entiers apparaissent dans l'ordre souhaité.

VariableHeuristic

VariableHeuristic est une classe abstraite qui permet d'implémenter ses propres heuristiques de choix de variable. Une heuristique de choix de variable déjà comprise dans le solveur est *FirstUnassignedVariableHeuristic* qui retourne, dans l'ordre lexicographique de déclaration, la première variable qui n'a pas encore été instanciée à partir de la liste de variables provenant de *Model*. Un utilisateur, qui souhaite implémenter sa propre heuristique personnalisée, doit implémenter la méthode *GetNextVariableToAssign* qui prend en entrée une liste de variables et retourne la prochaine variable que le solveur doit explorer.

Model

Model est la classe qui contient une liste des différentes contraintes et variables du modèle. C'est aussi la classe qui permet de valider que le modèle est toujours consistant ce qui veut dire que toutes les variables possèdent, dans leur domaine, au minimum une valeur. *Backup* permet d'appeler, pour toutes les variables, leur méthode *Backup*. Cela permet, lorsque le solveur effectue un retour arrière, de restaurer les variables à un état antérieur pour explorer d'autres sections de l'arbre. La méthode *Restore* permet d'appeler la méthode *Restore* des différentes variables ainsi que de réactiver toutes les contraintes désactivées précédemment.

Constraint

Constraint est une classe abstraite qui permet d'implémenter ses propres contraintes à appliquer sur les variables. Lorsqu'un utilisateur souhaite utiliser une contrainte, il doit donner la priorité de la contrainte (dans l'énumération *PropagatorPriority*) ainsi que les variables dans sa portée (liste *Vars*). La priorité d'une contrainte permet, lorsqu'un événement se produit, de déclencher les différents algorithmes de filtrage en ordre de priorité. En effet, Schulte et Stuckey [58] ainsi que Schulte et Tack [59] montrent qu'il est préférable de propager les contraintes de celui qui a le plus petit nombre de variables (petite arité) au plus grand (grande arité). Cela permet une propagation plus rapide puisque les contraintes à petite arité ont des algorithmes de filtrage généralement plus rapides qui peuvent contribuer à filtrer les variables dans leur portée pour éviter un travail trop lourd lorsque les algorithmes des contraintes à grande arité seront exécutés plus tard. Le solveur possède la même classification des priorités que [58, 59]. *IsActive* permet d'indiquer si la contrainte est active ou non. Une contrainte est généralement inactivée lorsque son algorithme de filtrage détecte que l'assignation de n'importe quelles

valeurs restantes dans les variables de sa portée satisfait la contrainte. Cela permet donc d'éviter d'appeler l'algorithme de filtrage associé inutilement. Les méthodes *SetPassive* permet de désactiver la contrainte lorsqu'elle est satisfaite pendant la recherche et *SetActive* permet de réactiver la contrainte lorsque *Model* demande un *Restore*. La méthode *Fail* permet d'avertir le modèle qu'une contrainte a échouée dans sa propagation (les valeurs des différentes variables de sa portée ne permettent plus de satisfaire la contrainte). Lorsqu'un utilisateur souhaite ajouter une nouvelle contrainte, il doit implémenter un certain nombre de méthodes. Les méthodes sont *IsSatisfied*, qui permet d'indiquer si la contrainte est satisfaite en considérant l'état actuel des domaines des variables ; *VariableHasBeenSelected*, qui permet d'avertir la contrainte que la variable dans sa portée, en entrée, a été sélectionnée (voir chapitre 4 pour le détail de cet événement) ; *VariableEventHappened*, qui permet d'avertir la contrainte qu'une de ses variables dans sa portée, en entrée, a subie un ou plusieurs événements (énumération *Event*) ce qui permet à l'algorithme de réagir à cette modification si nécessaire et *InitialPropagation*, qui permet d'exécuter un algorithme de propagation sur les variables dans la portée avant de débuter l'exploration de l'arbre de recherche.

IntVariable

IntVariable est la classe qui représente une variable dans un modèle de PPC. Elle est, entre autres, responsable de demander à son domaine courant [2] associé d'effectuer les différentes modifications requises à ses valeurs. À l'instanciation d'une nouvelle variable, l'utilisateur doit fournir le *Name* ainsi que le *Domain*. Le nom de la variable n'a aucune incidence sur le fonctionnement du solveur, il ne sert qu'à l'identification. La classe possède la méthode *AddConstraint*, qui sert à ajouter une contrainte qui a la variable courante dans sa portée (sert à l'application du patron de conception *Observer* [15]) ; *RemoveConstraint*, qui sert à retirer une contrainte ; *IsConsistant*, qui retourne vrai si la variable possède, au minimum, une valeur dans son domaine ; *NotifyVariableHasBeenSelected*, qui permet à la variable de notifier toutes les contraintes dans le dictionnaire *dictConstraints* qu'elle vient d'être sélectionnée par l'heuristique de choix de variable (voir détails dans la chapitre 4) et *NotifyVariableEventHappened*, qui envoie, aux différentes contraintes, un *eventMask* pour leur indiquer le ou les événements (dans l'énumération *Event*) qui se sont produits dans le *Domain* de la variable. La méthode *Backup* permet de conserver l'état actuel du domaine dans le *domainStack* et la méthode *Restore* permet de restaurer le domaine dans un état précédent.

2. Le domaine courant est celui qui se trouve au haut de la pile de *domainStack* (sert à conserver les états courant et précédents d'un domaine lorsque *Model* le demande).

La méthode *Cardinality* ne fait qu'appeler la méthode équivalente dans le *Domain*. *IsInstantiated* retourne vrai si la cardinalité du domaine de la variable est 1. Les méthodes *Instantiate*, *Clear*, *Remove*, *UpdateUpperBound* et *UpdateLowerBound* appellent les méthodes correspondantes dans le *Domain* et elles s'occupent aussi d'effectuer l'appel à *NotifyVariableEventHappened* avec les paramètres appropriés. *GetUpperBound* et *GetLowerBound* permettent d'appeler les méthodes correspondantes dans *Domain*.

Domain

Domain est la classe qui permet de conserver la liste des valeurs possibles d'une variable. Il existe deux constructeurs pour initialiser un domaine. Le premier reçoit une liste d'entiers ce qui permet de représenter un domaine où chaque valeur est énumérée (les valeurs doivent être en ordre strictement croissant, mais elles n'ont pas besoin d'être consécutives). Le deuxième constructeur reçoit la borne inférieure et la borne supérieure du domaine ainsi qu'une valeur booléenne qui indique si le domaine doit être énuméré ou intervalle. Si le domaine est énuméré, alors le constructeur va énumérer toutes les valeurs entre la borne inférieure et supérieure (incluant les bornes) et si le domaine est intervalle, alors seules les bornes seront conservées. Un domaine ne possède une instance d'une classe implémentant *IBitSet* que si les valeurs sont énumérées. Les opérations supportées sur le domaine sont *Cardinality*, qui retourne le nombre de valeurs dans le domaine ; *Instantiate*, qui instancie (assigne) le domaine avec la valeur en paramètre ; *Clear*, qui retire toutes les valeurs du domaine ; *Remove*, qui permet de retirer une valeur spécifique dans un domaine énuméré (cette opération n'est pas disponible pour un domaine intervalle) ; *UpdateUpperBound*, qui permet de modifier la borne supérieure ; *UpdateLowerBound*, qui permet de modifier la borne inférieure ; *GetUpperBound*, qui permet de retourner la borne supérieure courante et *GetLowerBound*, qui permet de retourner la borne inférieure courante.

IBitSet et BitSetFactory

IBitSet est une interface qui sert à représenter la structure de donnée *BitSet*[3]. C'est une structure de données couramment utilisée dans les solveurs[4] pour représenter les différentes valeurs d'un domaine énuméré. Dans le but de bien expliquer son fonctionnement, nous donnons des exemples. Supposons une taille d'un mot w de 8 bits[5] et

3. Cette structure de données est aussi connue sous les noms de *bit array* ou *bit vector*.
4. Choco [54] est un exemple d'un solveur qui l'utilise.
5. 8 bits est une taille utilisée pour faciliter les exemples. Dans le solveur, considérant les processeurs modernes, nous utilisons plutôt $w = 64$ ce qui correspond au nombre de bits dans le type *ulong*.

nous souhaitons représenter l'ensemble de valeurs {2, 4, 5, 8}. La première étape est de conserver la valeur minimum (borne inférieure) de l'ensemble comme étant le *Offset*. Dans le cas présent, la valeur minimum est 2, donc le *Offset* est 2. Par la suite, nous pouvons prendre l'ensemble en entrée et l'encoder dans un nombre binaire de w bits où, à chaque position, 1 indique la présence d'une valeur et 0 son absence. Donc, dans notre exemple, nous nous retrouvons avec le nombre binaire $b = 10110010$ (la valeur *Offset* est représentée à la position 0 qui est celle le plus à gauche dans les exemples). Pour tester si une valeur est présente dans le *BitSet*, nous avons juste besoin de tester si $b[\mathit{ValeurATester} - \mathit{Offset}] = 1$ (le tableau de bits est en base 0). Par exemple, la valeur 4 est présente puisque $b[4-2] = 1$, mais la valeur 6 ne l'est pas puisque $b[6-2] = 0$. C'est la manière dont l'implémentation de l'interface *OneWordBitSet64* fonctionne. Il arrive qu'un mot de taille w ne soit pas suffisant pour tout encoder. Reprenons l'exemple selon lequel nous avons $w = 8$, mais que cette fois-ci nous souhaitons encoder l'ensemble {2, 4, 5, 8, 11, 13, 17}. Puisque $\mathit{borneSupérieure} - \mathit{borneInférieure} + 1 > w$, alors nous aurons besoin de $\lceil \frac{\mathit{borneSupérieure} - \mathit{Offset}}{w} \rceil$ mots de taille w pour encoder l'ensemble. Donc nous nous retrouvons avec le mot $a[0] = 10110010$ pour la première partie et $a[1] = 01010001$ pour la deuxième partie (a est un tableau qui contient des mots). Dans le but de tester la présence d'une valeur, nous devons trouver la valeur binaire à la position $p = \mathit{ValeurATester} - \mathit{Offset}$ qui se trouve dans $a[p \ \mathit{Mod}\ |a|][p - ((p \ \mathit{Mod}\ |a|) * w)]$. *BitSet64* est l'implémentation qui permet d'encoder des ensembles dans un tableau de plusieurs mots de taille w. Dans le but de faciliter la vie des utilisateurs qui créent un *Domain* pour leurs variables, nous utilisons la classe *BitSetFactory* (qui utilise le patron de conception *Factory* [15]) qui permet de décider si le domaine sera modélisé avec *OneWordBitSet64* ou *BitSet64*. Dans tous les cas, les classes implémentant *IBitSet* donnent accès à *Offset*, qui permet de donner la valeur à la borne inférieure de l'ensemble qui a servi à créer le *BitSet* ; *Cardinality*, qui donne la cardinalité de l'ensemble courant ; *Clear(bitIndex)*, qui prend un bitIndex en entrée et met à 0 le bit à cette position-là ; *Clear(fromIndex, toIndex)*, qui permet de placer les bits du *fromIndex* (inclus) jusqu'au *toIndex* (exclus) à 0 ; *Clear*, qui place tous les bits à 0 ; *Set*, qui place le bit indiqué en entrée à 1 ; *Get*, qui indique pour bitIndex en entrée s'il est à 1 (*true*) ou 0 (*false*) ; *NextSetBit*, qui retourne le prochain bit à 1 sur ou après l'index spécifié ou retourne -1 s'il n'existe pas ; *PrevSetBit*, qui retourne le bit précédent sur ou avant l'index spécifié ou retourne -1 s'il n'existe pas et *Capacity*, qui retourne $w \times \mathit{NombreDeMots}$ qui correspond à la capacité totale en nombre de bits dans la structure de données courante.

2.2 Diagramme de séquence de la résolution d'un problème de PPC

Le diagramme de séquence de la figure 2.2 montre la séquence des appels aux différentes méthodes des différentes classes lorsqu'un utilisateur demande la résolution d'un problème (utilisation de la méthode *Solve* dans la classe *Solver*). Au tout début, avant la résolution, il faut ajouter les contraintes dans une liste pour chaque variable de leur portée. Cela permet le fonctionnement de la propagation des événements (patron de conception *Observer* [15]). Ensuite, il faut effectuer la propagation initiale de toutes les contraintes. Finalement, nous cherchons quelle est la première variable à être instanciée et on l'envoie à la stratégie de recherche demandée par l'utilisateur. Nous supposons, dans le cas présent, que l'utilisateur souhaite résoudre le problème avec DFS. Le diagramme de séquence de la figure 2.3 permet de montrer le détail des appels lorsque *Solve* appelle la méthode *DFS*. Nous remarquons l'appel à *NotifyVariableHasBeenSelected* de la variable en entrée. Le chapitre 4 montrera le détail de la séquence qui se produit par la suite. Comme toutes stratégies de recherche, DFS va demander aux différentes variables d'instancier certaines valeurs. Ces instanciations déclenchent donc un événement qui va ultimement permettre la propagation des contraintes. Pour bien voir le détail de comment une variable arrive à avertir les contraintes, dans lesquelles elle est dans leur portée, qu'un événement s'est produit, nous montrons la figure 2.4 qui illustre la séquence d'appels dans ces cas-là. Pour simplifier le diagramme, nous supposons que la variable instanciée est une variable énumérée (qui possède donc un *IBitSet* associé).

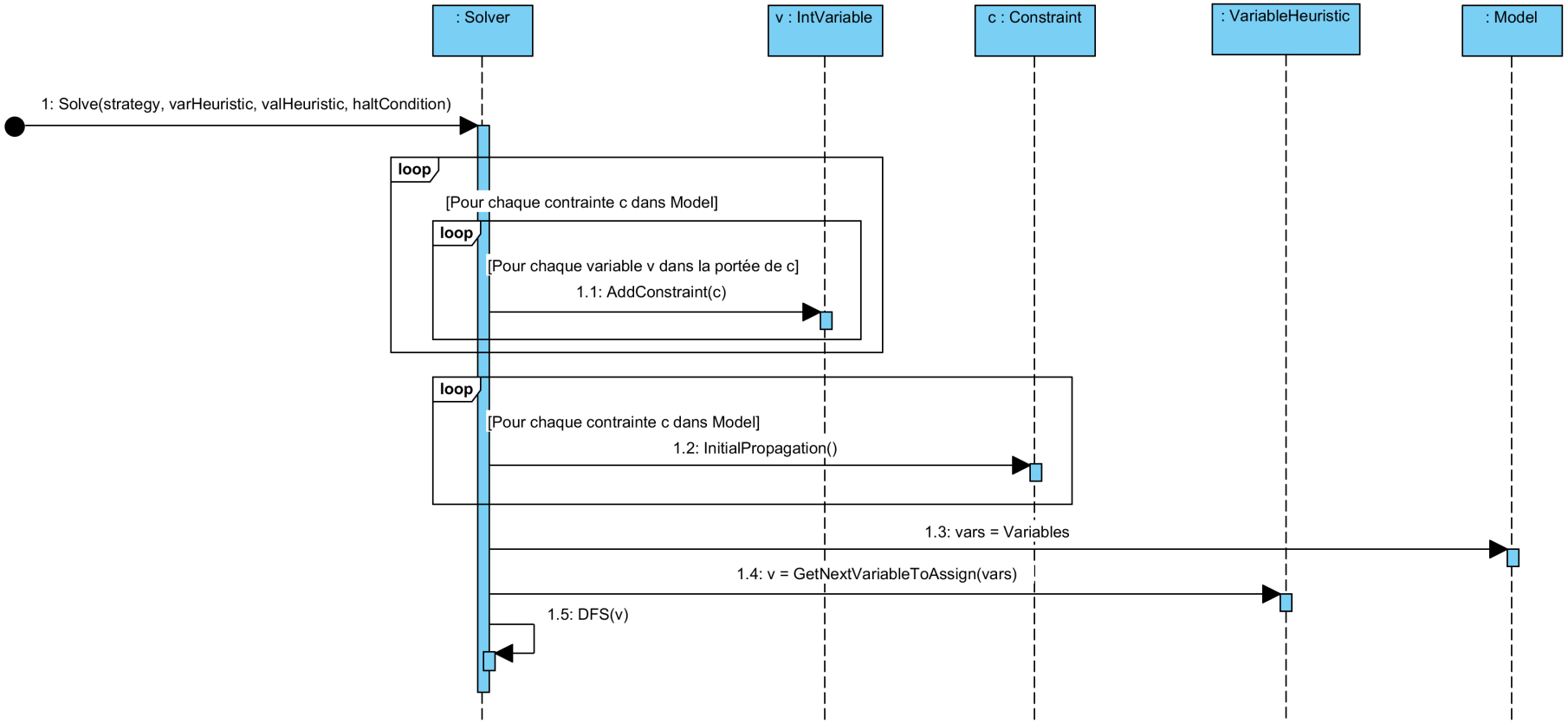


FIGURE 2.2 – Diagramme de séquence lorsqu'un utilisateur souhaite résoudre un problème avec le solveur

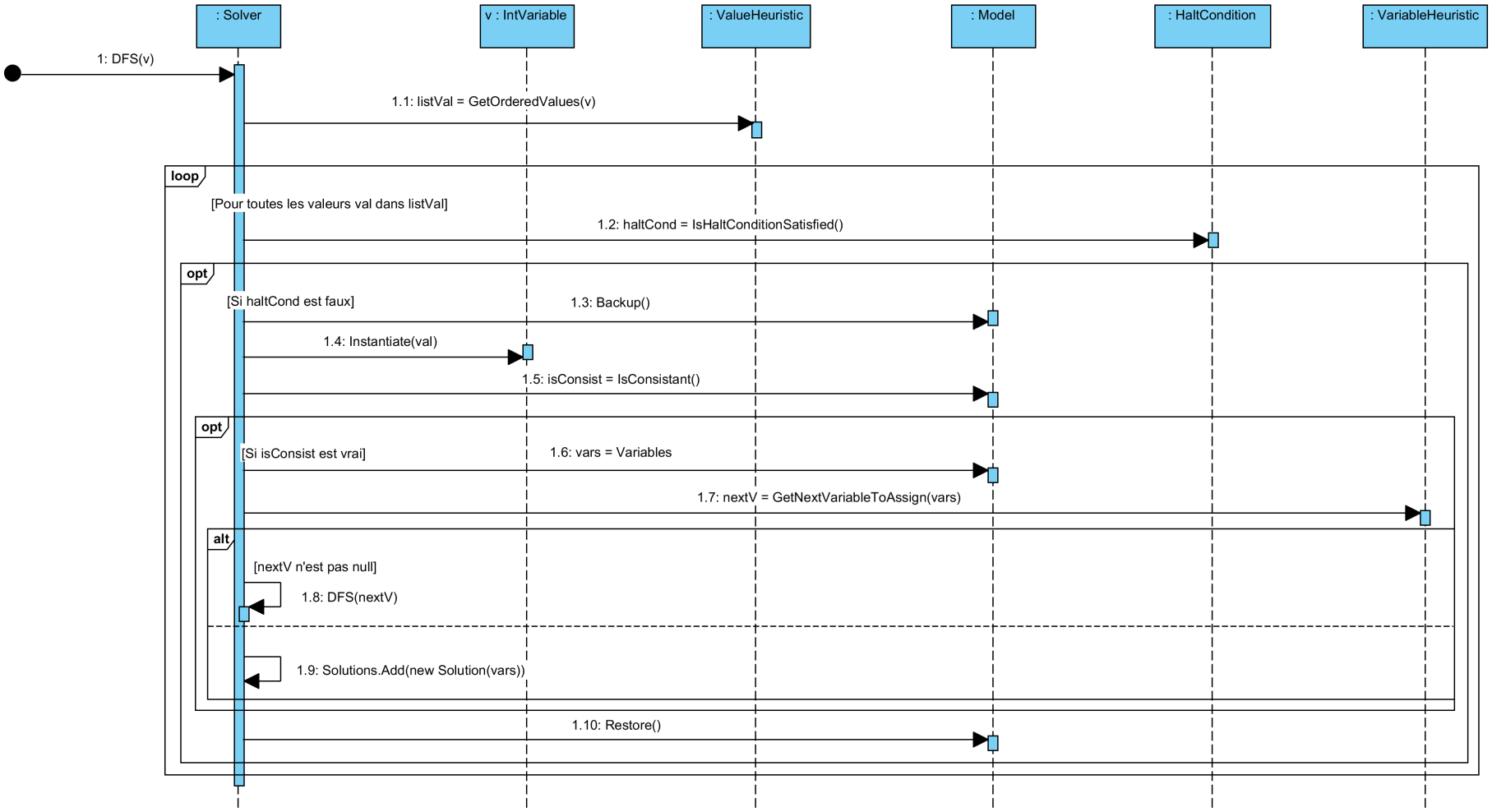


FIGURE 2.3 – Diagramme de séquence lorsque *Solve* appel l'algorithme de DFS

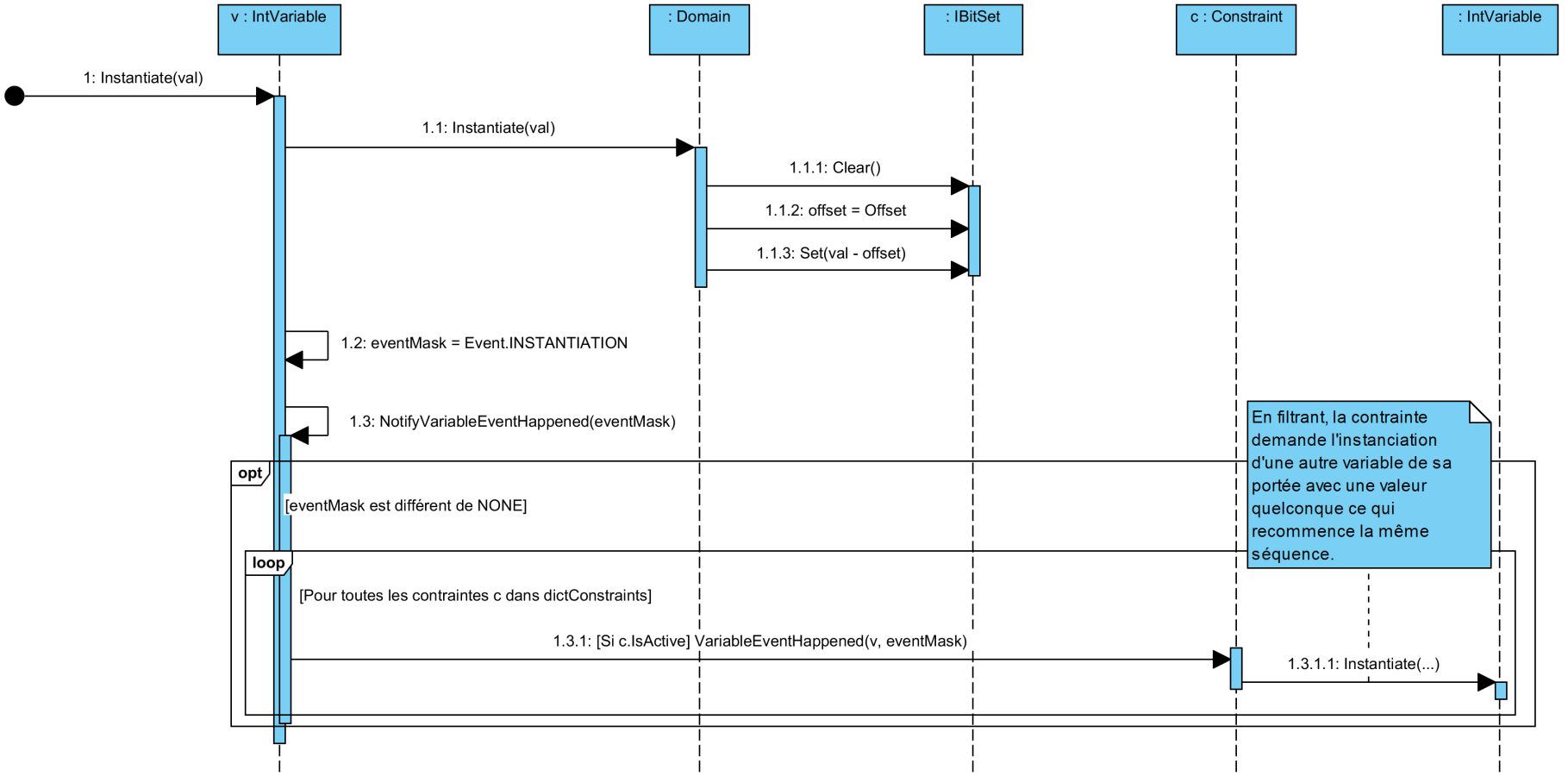


FIGURE 2.4 – Diagramme de séquence lorsque DFS demande l'instanciation d'une variable

2.3 Conclusion

Dans ce chapitre, nous avons décrit le nouveau solveur qui est destiné à permettre aux différents membres de FORAC de rapidement implémenter de nouvelles techniques de résolution des problèmes en PPC. Le langage utilisé, C#, est un langage couramment utilisé pour diverses applications et logiciels de FORAC, ce qui permettra une intégration plus facile dans les différents projets. Dans les prochains chapitres, afin de montrer sa souplesse d'utilisation, nous montrerons comment il peut être utilisé pour la résolution d'un problème provenant de l'industrie forestière (chapitre 3) ainsi que l'implémentation d'un nouvel événement qui permet d'effectuer ce que nous appelons du *filtrage tardif* (chapitre 4).

Chapitre 3

Utilisation du solveur pour la résolution d'un problème industriel : génération de patrons de chargement alternatifs pour les séchoirs à bois

L'industrie forestière fait face à plusieurs problèmes de décision au jour le jour qui peuvent se résoudre grâce à l'optimisation combinatoire. Par exemple, des étudiants de FORAC [1, 39] ont utilisé avec succès diverses techniques pour aider à une meilleure planification du rabotage du bois. Un autre problème pour lequel l'optimisation combinatoire peut aider est celui de générer des patrons de chargement variés. Un employé d'une usine fait face à plusieurs choix en ce qui concerne les paquets de bois qui peuvent entrer dans un séchoir, une ressource limitée en terme d'espace. Puisque les demandes des clients peuvent changer dans le temps, il est souhaitable que l'employé ait accès à plusieurs patrons de chargement dans lesquels il peut choisir tout en étant confiant qu'ils respectent tous les différentes contraintes décrites dans ce chapitre. C'est pourquoi nous proposons l'utilisation du solveur décrit au chapitre 2 pour générer plusieurs de ces patrons de chargement grâce à la PPC. Dans la section 3.1, nous décrirons le problème industriel du séchoir. La section 3.2 décrit le modèle utilisé pour la résolution du problème. La section 3.3 décrit les expérimentations que nous avons effectuées. La section 3.4 discute des résultats obtenus. La section 3.5 conclut ce chapitre.

3.1 Description du problème industriel

Dans une scierie de bois d'oeuvre, nous pouvons décomposer le processus de transformation du bois en trois grandes étapes [16] :

1. Dans un premier temps, nous avons le sciage qui permet de transformer les billots en pièces de bois nommées *sciages* (ci-après nommé *produits verts*[1]).
2. Par la suite, le processus de séchage sert à réduire la teneur en humidité selon les spécifications prédéterminées (transforme les *produits verts* en *produits secs*).
3. Finalement, le rabotage permet de donner un fini lisse, arrondir les arrêtes et ébouter, au besoin, les planches dans le but de retirer les imperfections et augmenter la valeur combinée des sous-produits ainsi créés (transforme les *produits secs* en *produits finis*).

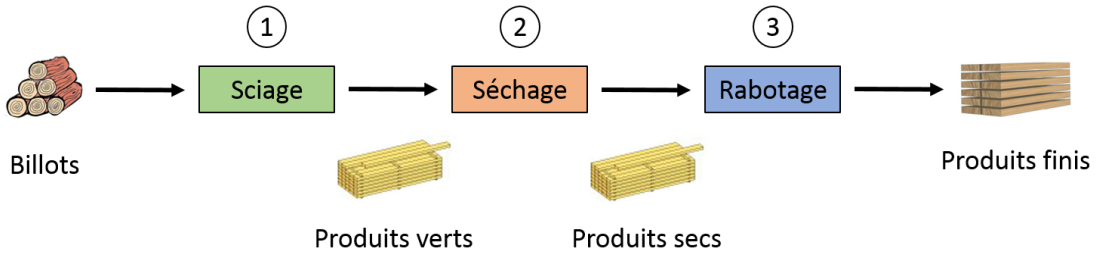


FIGURE 3.1 – Représentation visuelle des trois étapes de transformation dans une scierie

Les défis de coordination entre ces différentes étapes sont étudiés dans la littérature scientifique [17, 18].

3.1.1 Coproduction dans le contexte forestier

Une difficulté qu'il faut considérer est celle que posent les processus *divergents avec coproduction*. Un processus est défini comme divergent lorsque plusieurs produits en sortie sont créés avec un même produit en entrée. Lorsque ces produits sont créés en même temps, alors nous parlons de coproduction [8, 19]. C'est un problème qui pose un grand défi dans le contexte de l'industrie forestière puisqu'un billot de bois va nécessairement générer plusieurs produits verts après l'étape de sciage et un produit sec va générer plusieurs produits finis après l'étape de rabotage. Certains de ces produits sont ceux souhaités pour satisfaire les commandes des clients, alors que d'autres vont

1. Dans l'industrie forestière, un produit est dit *vert* lorsqu'il n'a pas encore été séché.

s'accumuler dans la cour en attendant de pouvoir être vendus. C'est pourquoi, lorsque vient le temps de choisir des produits à placer dans un séchoir, nous devons placer les bons produits verts qui vont ultérieurement générer, au rabotage, les produits finis souhaités. Dans l'industrie, nous utilisons un graphe de conversion qui permet de donner la conversion en pourcentage de PMP[2] d'un produit vert vers les produits finis qu'il va générer, en moyenne, basé sur les historiques de production. Le graphe à la figure 3.2 nous sert d'exemple. À chaque fois que nous décidons de placer un produit vert dans le séchoir, nous générons obligatoirement tous les produits finis possibles en quantité prédéfinie (en moyenne) une fois que le produit est raboté.

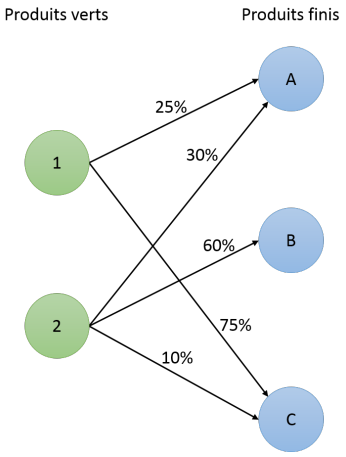


FIGURE 3.2 – Les produits verts 1 et 2 donnent les produits finis A, B et C selon un certain pourcentage de PMP.

Nous pouvons aussi exprimer le graphe de conversion de la figure 3.2 en une matrice de conversion comme le montre le tableau 3.1. C'est sous cette forme que les usines de l'industrie forestière donnent les *recettes* de création des produits finis à partir des produits verts.

Produit	**Fini A**	**Fini B**	**Fini C**
Vert 1	0,25	0,00	0,75
Vert 2	0,30	0,60	0,10

TABLE 3.1 – Matrice de conversion équivalente à la figure 3.2.

Une mise en situation concrète pour expliquer le fonctionnement du graphe de conversion, en considérant la figure 3.2, est un employé d'une usine qui souhaite avoir uni-

2. Le Pied-Planche (mieux connu sous le nom Pied Mesure de Planche (PMP) dans l'industrie) est une unité de mesure du bois égale à 12 pouces sur 12 pouces sur 1 pouce. La définition est tirée de la *Loi de 2006 sur les droits d'exportation de produits de bois d'oeuvre*, Lois du Canada 2006, chapitre 13.

quement le produit fini B. Il devrait donc placer dans le séchoir le produit vert 2 pour lequel 60% de PMP sera converti en produit fini B. Par contre, même s'il n'a pas besoin des produits finis A et C, du fait de la coproduction, ils seront générés en même temps que le produit fini B. Il est donc nécessaire de considérer ce fait lorsqu'il décide quels produits verts il place dans le séchoir. Un autre exemple est qu'il pourrait souhaiter générer le produit fini C. Selon le graphe de conversion (figure 3.2), il serait plus avantageux pour l'employé de placer le maximum de paquets de produits verts 1 (sous réserve de disponibilité en inventaire) pour lequel 75% de PMP est transformé en produit fini C que de placer le produit 2 pour lequel seulement 10% de PMP est converti en produit C.

3.1.2 Description détaillée du processus de séchage

Le bois résineux est utilisé, entre autres, comme bois de construction. Considérant son utilisation dans l'industrie de la construction, nous souhaitons certaines caractéristiques telles que la stabilité dimensionnelle, la réduction ou l'élimination de la décomposition de la matière et l'augmentation de la dureté [67]. Nous y arrivons en réduisant la teneur en humidité du bois grâce au processus de séchage. Par contre, ce processus est long. Il est souvent le goulot d'étrangement dans la chaîne de production, considérant que sécher des produits verts peut prendre 12 à 60 heures [17]. La durée de séchage dépend de règles qui varient selon l'essence du produit vert, la dimension des pièces de bois à sécher et la teneur en humidité qui peut varier selon la période de l'année et le temps qu'a passé le bois vert dans la cours.

Tel que mentionné à la section précédente, le processus de sciage du bois génère des produits verts. Un paquet est composé de plusieurs produits verts ayant les mêmes dimensions (2"×3", 2"×4" et 2"×6" par exemple), mêmes longueurs (8', 10', 12', 14' ou 16') et regroupés selon l'essence du bois ou la teneur en humidité [16]. Ils sont assemblés de manière à créer un prisme rectangulaire ayant une certaine longueur et une certaine hauteur (3' par exemple) [16]. Nous avons donc plusieurs *types de paquets*. Nous plaçons ces paquets dans le séchoir. Par contre, il est important de noter que tous les types de paquets ne peuvent pas être séchés en même temps dans un même séchoir. En effet, les types de paquets doivent avoir des règles de séchage compatibles (typiquement même essence et même dimension).

Une usine possède typiquement plusieurs séchoirs qui seront utilisés simultanément dans le but de satisfaire les demandes des clients. Les demandes sont regroupées dans

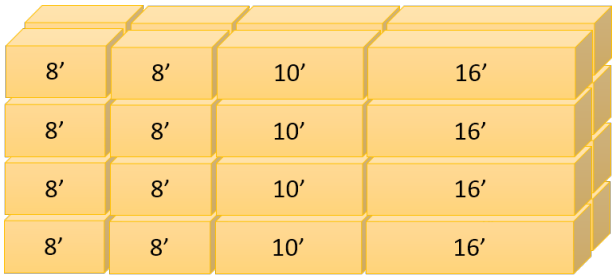


FIGURE 3.3 – Exemple d'un patron de chargement dans un séchoir. Le patron présenté ici est composé de deux *rangées* et chaque *pile* est composée de quatre paquets de mêmes longueurs. (Inspiré de [16])

un carnet de commandes. Dans le but de remplir les commandes, il est nécessaire de planifier l'utilisation des séchoirs sur une longue période de temps. À chaque utilisation d'un séchoir, nous plaçons des paquets de produits verts différents selon la demande. À chaque fois qu'un séchoir est utilisé, nous disons qu'il *contribue* à réduire le retard. En effet, si rien n'était produit pendant la période de planification, nous accumulerions un retard en ce qui concerne la satisfaction des demandes du carnet de commandes. C'est pourquoi il est souhaitable de tenter de maximiser la contribution d'un séchoir à réduire les retards lorsqu'il est utilisé. Le calcul de la contribution d'un séchoir est décrit et utilisé plus tard dans ce chapitre (voir l'algorithme 4). La figure 3.4 donne un exemple de planification sur deux séchoirs différents pendant 20 périodes (dans cet exemple, une période représente 12h). Chaque rectangle de couleur dans cette solution correspond à l'utilisation d'un patron de chargement possible (celui de la figure 3.3 en est un exemple).

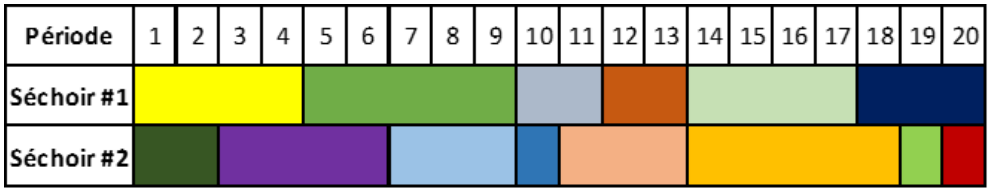


FIGURE 3.4 – Exemple de l'utilisation de deux séchoirs sur 20 périodes. Chaque couleur représente un patron de chargement différent. (Inspiré de [38])

Planifier les opérations sur plusieurs séchoirs et sur plusieurs périodes est un problème qui a été abordé plusieurs fois dans la littérature scientifique. Par exemple, Gaudreault et coll. [18] décrit deux approches différentes pour résoudre ce problème. La première approche utilise le *Mixed-Integer Programming* (MIP) et la deuxième utilise la PPC. Dans tous les cas, à chaque fois qu'il faut remplir un séchoir libre, le choix du meilleur patron de chargement est fait dans une liste de patrons prédéterminés. Les

auteurs montrent que l'approche PPC est celle qui offre le plus rapidement une bonne solution. Du fait que les techniques de [18] utilisent une liste de patrons prédéterminés, il est possible que la planification résultante ne soit pas optimale. C'est pourquoi Marier et coll. [38] proposent une approche qui permet, avec un modèle MIP, de générer dynamiquement un patron de chargement. Ils utilisent ce modèle MIP en combinaison avec une fouille dans un arbre de recherche. À chaque fois que l'arbre doit choisir un chargement, pour un séchoir, ils utilisent le modèle MIP pour générer celui-ci. Cela permet une plus grande flexibilité et une meilleure adéquation par rapport à la satisfaction des besoins exprimés dans le carnet de commandes.

3.1.3 Recherche de chargements alternatifs

Les techniques présentées dans 3.1.2 permettent d'effectuer la planification de plusieurs séchoirs d'une usine sur plusieurs périodes. Par contre, il se peut qu'un employé souhaite modifier le chargement d'un séchoir pour diverses raisons. Par exemple, si nous regardons la planification de la figure 3.5, un employé souhaite modifier le chargement entouré en rouge parce qu'il reçoit la commande d'un client prioritaire qui a besoin de 16 paquets de 16' et non pas seulement 8 comme c'était planifié à l'origine. Nous souhaitons donc permettre à cette personne de modifier le chargement pour ajouter les 8 paquets de 16' qui manquent tout en s'assurant que le nouveau chargement résultant reste possible (ne viole pas de contraintes liées aux dimensions du séchoir ou à la stabilité des piles de paquets par exemple).

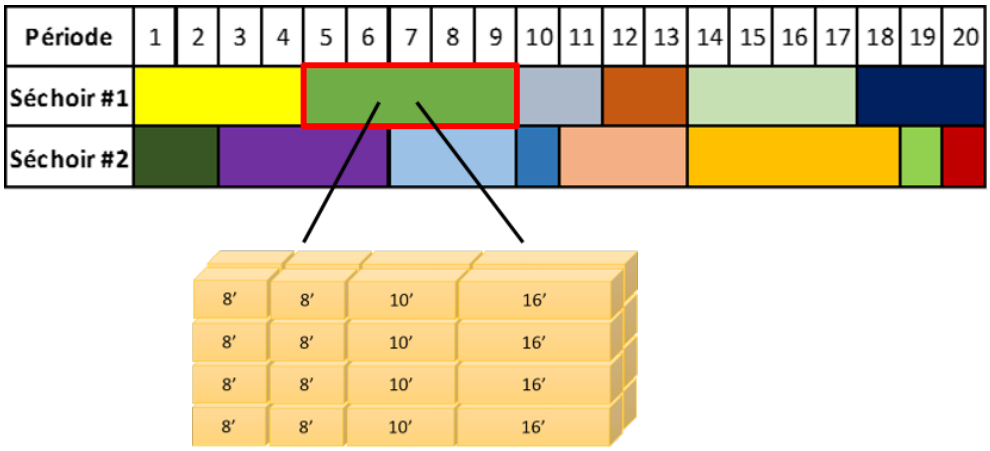


FIGURE 3.5 – Un employé souhaite modifier le patron de chargement du séchoir entouré en rouge. (Inspiré de [38])

Ce changement va probablement avoir une conséquence par rapport à la planification originale en ce qui concerne la satisfaction des besoins exprimés dans le carnet de

commandes. En effet, dans le but de créer de la place pour ces paquets de 16', il faut inévitablement retirer des paquets d'autres formats. Ces paquets retirés permettaient de satisfaire les commandes d'autres clients. Il faut donc remettre à plus tard la satisfaction des commandes de ces autres clients dans le but de satisfaire, en premier, la commande du client prioritaire. Cela entraîne donc, potentiellement, une augmentation du retard par rapport aux commandes exprimées dans le carnet de commandes. En d'autres mots, nous pouvons aussi dire que l'utilisation d'un séchoir avec le chargement ainsi modifié ne contribue, potentiellement, plus autant à la réduction du retard total qu'avec le chargement original. C'est pourquoi il est désirable d'éviter que le retard augmente plus qu'un certain seuil fixé par l'employé (une tolérance quant à l'augmentation du retard total). Il faut donc trouver rapidement un chargement alternatif qui va non seulement inclure les paquets de 16' demandés, mais aussi éviter d'augmenter le retard total de plus que le seuil fixé.

Une manière de procéder pour donner à l'utilisateur la possibilité de trouver très rapidement un nouveau chargement qui respecte les contraintes est d'en générer plusieurs d'avance (à l'aide de la méthode que nous proposerons un peu plus loin). Par exemple, nous pouvons imaginer un logiciel qui permet à un employé d'une usine de demander des modifications dans un patron de chargement. Avant même que l'interface visuelle permettant de sélectionner des paquets apparaisse à l'utilisateur, nous pouvons trouver plusieurs plans de chargement, plus variés les uns que les autres, lorsque l'utilisateur indique ses besoins en termes de paquets. Le système pourra proposer instantanément un chargement ayant, au minimum, les paquets sélectionnés par l'utilisateur. Le présent chapitre montre, en détail, l'approche que nous utilisons dans le but de générer plusieurs chargements alternatifs dans le but de faire fonctionner ce système.

Ce type de système qui combine l'efficacité des techniques d'optimisation automatisée avec l'intuition d'un humain est appelé *Mixed-Initiative Interaction* [26], mieux connu sous le nom de *Mixed-Initiative-System* (Système à initiative partagé) (voir des exemples d'application dans [22] ainsi que [6]). Ce type de système permet à un décideur d'être directement impliqué dans la planification de l'utilisation des séchoirs et ne plus être un simple spectateur. En effet, les humains sont ceux qui, malgré l'usage de techniques informatisées, peuvent réellement prendre en considération le contexte dans lequel évolue leur entreprise. Il existe plusieurs raisons pour lesquelles une modification à la planification originale peut être requise et c'est pourquoi il est crucial de considérer les besoins exprimés par un utilisateur.

3.1.4 Remplissage d'un séchoir

Avant de décrire l'approche utilisée pour générer les chargements alternatifs, nous devons décrire comment un employé peut charger un séchoir (comment il peut concevoir un patron de chargement).

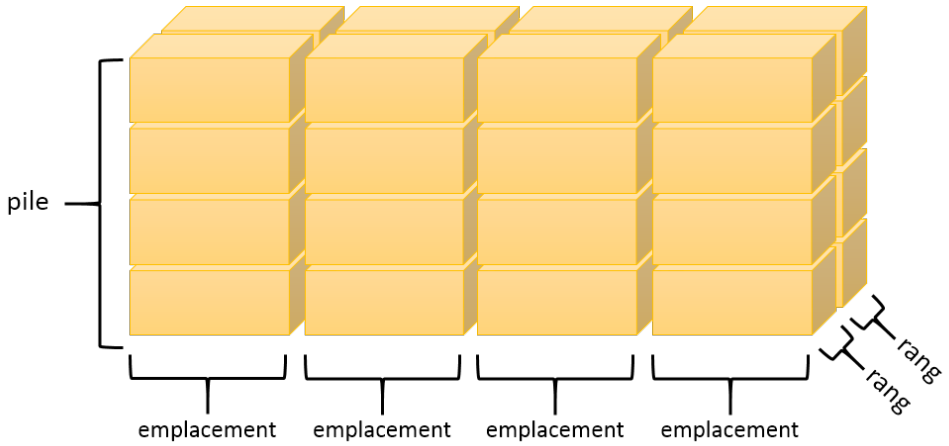


FIGURE 3.6 – Figure illustrant les termes *pile*, *emplacement* et *rang* (ou *rangée*).

Lorsqu'un employé remplit un séchoir, il dispose d'un nombre prédéterminé de *rangées* (ou *rangs*) dans lesquelles il peut placer des paquets de produits verts. Pour chaque rangée, il peut placer, en longueur, un certain nombre de paquets. Il y a une longueur totale maximale de paquets qu'il peut placer dans un rang, mais puisqu'il souhaite aussi maximiser la pleine utilisation du séchoir, il y a aussi une longueur totale minimale de paquets. Il doit donc décider du nombre de paquets qu'il souhaite sur chaque rangée et de la longueur qu'ils auront. Il cherche donc à déterminer les longueurs des différents *emplacements* sur lesquels il va, par la suite, empiler les paquets. Le choix des longueurs, pour chaque emplacement, doit être pris, typiquement, dans l'ensemble {8', 10', 12', 14', 16'}. Évidemment, l'ensemble des longueurs dépend des longueurs des paquets disponibles en inventaire. Il est important de noter qu'un employé n'a pas besoin de placer exactement la même longueur totale de paquets sur les différentes rangées puisqu'elles sont gérées indépendamment l'une de l'autre. Par exemple, un employé peut décider, sur la première rangée, d'utiliser la totalité de sa capacité (longueur totale maximale) pour placer des paquets et, sur la deuxième rangée, n'utiliser que la capacité minimale de celle-ci.

Une fois que l'employé a décidé des longueurs des emplacements sur chaque rangée, il peut commencer à empiler des paquets. Il doit, par contre, s'assurer de la stabilité de chacune des *piles*. On permet donc seulement d'empiler des paquets de mêmes longueurs

sur chaque emplacement. En d'autres termes, il peut placer n'importe quel type de paquet dans une pile tant que la longueur de celui-ci correspond à celui de l'emplacement. Cela empêche les cas où une personne décide d'empiler un paquet de 16' par-dessus un paquet de 8'. En effet, le paquet de 16' risque de tomber. Il y a aussi une hauteur totale maximale de paquets qu'on peut placer dans une pile. Les paquets ont une hauteur standard (3' pour les cas étudiés). Sachant que l'on dispose, par exemple, d'un séchoir de 12' de haut, alors on sait que l'on peut empiler que 4 paquets au maximum. Puisqu'on souhaite, de la même manière que pour la longueur totale des paquets, utiliser au maximum le séchoir, alors on doit s'assurer de toujours former des piles de 4 paquets (le fait de ne pas pouvoir compléter une pile par manque de disponibilité de paquets de la longueur de l'emplacement désigné entraine le rejet du patron de chargement).

3.1.5 Structure d'un carnet de commandes

Nous indiquions, précédemment, que nous pouvons remplir le séchoir avec des paquets de produits verts de plusieurs types. Par contre, dans les faits, nous souhaitons satisfaire des commandes de produits finis passées par les différents clients de l'usine. Ces demandes sont exprimées sous la forme d'un carnet de commandes. Le tableau 3.2 nous donne un exemple d'un carnet de commandes qui donne les demandes de produits finis demandés par les différents clients d'une usine. La colonne *Période* nous donne la période à laquelle le produit est demandé, la colonne *Produit* donne le code du produit fini désiré par un client et la colonne *Quantité* nous donne la quantité, en PMP, requise du produit fini.

Tel que mentionné dans la section 3.1, un produit vert (assemblé dans un paquet) est transformé en produit sec par le processus de séchage et le produit sec est transformé en un ou plusieurs produits finis par le processus de rabotage. Dans le but de décider quels seront les bons produits verts à placer dans le séchoir pour générer les produits finis souhaités, nous utilisons une matrice de conversion (voir tableau 3.1). Par contre, typiquement, la demande de produits finis dépasse très souvent la capacité de traitement du séchoir. Les retards seront donc généralement inévitables. C'est pourquoi nous souhaitons prioriser le séchage des paquets des produits verts qui correspondent aux produits finis demandés le plus tôt. Par exemple, si nous sommes à la période 9 au moment de débuter le séchage de produits verts, alors nous souhaitons remplir le séchoir de produits qui sont demandés à la période 14 en priorité (la période la plus tôt des premiers produits finis demandés selon le tableau 3.2). Il arrive parfois que nous ayons plus de paquets de produits verts requis à une certaine période que de place dans

Période	Produit	Quantité
14	B	502,552
14	C	16 751,918
14	D	16 920,700
14	E	8 232,000
14	F	3 585,000
14	G	13 847,951
14	L	3 136,000
28	A	615,138
28	H	4 065,359
28	I	24 026,211
28	J	13 395,963
28	K	1 764,431
56	C	3 920,000
56	D	25 872,000
56	E	35 672,000
56	F	43 904,000

TABLE 3.2 – Exemple de demandes dans un carnet de commandes. La structure est réelle, mais les demandes sont fictives.

le séchoir. Nous avons, alors, un retard. Dans le but de le minimiser, il faut placer les paquets qui sont en situation de retard aussitôt que le séchoir sera de nouveau libre. Nous cherchons donc à prioriser la satisfaction des commandes les plus pressantes en sélectionnant les *meilleurs paquets* possibles.

3.2 Description de l'approche de résolution proposée

Dans cette section, nous décrivons une approche de PPC qui permet de générer les chargements alternatifs nécessaires au bon fonctionnement du système décrit dans la section 3.1.3. Ce que nous souhaitons de l'approche est de trouver très rapidement des chargements alternatifs qui sont variés. Dans le but de trouver rapidement ces chargements, il faut sacrifier la recherche exhaustive de tous les chargements possibles dans le but de se concentrer sur les chargements qui offrent les meilleures chances de respecter la contrainte de contribution minimale.

3.2.1 Approche proposée de résolution en PPC

L'approche proposée se décompose en quatre grandes étapes qui combinent plusieurs techniques différentes pour arriver à fournir rapidement des patrons de chargement variés.

1. Trouver une assignation valide de longueurs pour les différents emplacements qui composent les différents rangs (section 3.2.3).
2. Détecter si nous n'avons pas déjà trouvé une assignation des longueurs des emplacements équivalente précédemment (section 3.2.4).
3. En utilisant un algorithme vorace, déterminer si nous pouvons empiler des paquets dans le but de trouver un patron de chargement valide qui respecte la contrainte de contribution minimale (section 3.2.5).
4. En s'aidant du patron de chargement trouvé par l'algorithme vorace, trouver des patrons de chargement alternatifs (section 3.2.6).

3.2.2 Paramètres en entrée

— Un ensemble P représentant les différents produits verts p. Les produits verts dans l'ensemble ont tous une règle de séchage compatible avec le séchoir dont on souhaite modifier le patron de chargement.

— Un ensemble F représentant les différents produits finis f.

— Un ensemble D représentant les différentes demandes d du carnet de commandes.

— Un nombre réel *ContributionMin* représentant la contribution minimale que les patrons de chargement générés doivent avoir pour être considérés valides. C'est une borne inférieure de contribution qui est déterminée par l'utilisateur. La contribution d'un patron de chargement est calculée à l'aide de l'algorithme 4.

— Un nombre entier *LongueurRang* donnant la longueur des rangs dans le séchoir.

— Un nombre entier *LongueurMinRang* donnant la longueur totale minimale de paquets qui doivent être sur les rangs.

— Un nombre entier *HauteurSéchoir* donnant la hauteur maximale d'une pile de paquets dans le séchoir.

— Un nombre entier *NbRangs* donnant le nombre de rangs que le séchoir possède.

— Un nombre entier *HorizonPlanification* donnant le nombre de périodes sur lesquelles la planification de l'utilisation des séchoirs doivent se faire.

— Un nombre entier *DuréeSéchage* indiquant la durée, en nombre de périodes, du séchage.

— Un vecteur *ProdDemandé* de taille $|D|$ où l'élément *ProdDemandé*$[d]$ indique le produit fini $f \in F$ demandé.

— Un vecteur *QteDemandée* de taille $|D|$ où l'élément *QteDemandée*$[d]$ indique la quantité requise du produit fini en PMP (nombre réel) de la demande.

— Un vecteur *PériodeDemandée* de taille $|D|$ où l'élément *PériodeDemandée*$[d]$ indique la période (nombre entier) attendue de la demande.

— Un vecteur *QtePMPProdVert* de taille $|P|$ où l'élément *QtePMPProdVert*$[p]$ donne la conversion du nombre de paquets d'un produit vert p en PMP (nombre entier).

— Une matrice *Recette* de taille $|P| \times |F|$ où l'élément *Recette*$[p][f]$ donne le pourcentage de PMP du produit vert p qui sera converti en PMP du produit fini f.

— Un vecteur *LongPqtProdVert* de taille $|P|$ où l'élément *LongPqtProdVert*$[p]$ donne la longueur du paquet de produits verts p (nombre entier).

— Un vecteur *HautPqtProdVert* de taille $|P|$ où l'élément *HautPqtProdVert*$[p]$ donne la hauteur du paquet de produits verts p (nombre entier).

— Un vecteur *QteInventaire* de taille $|P|$ où l'élément *QteInventaire*$[p]$ donne le nombre de paquets du produit vert p en inventaire (nombre entier).

— Un ensemble *LongDiff* qui correspond à $\{LongPqtProdVert[1]..LongPqtProdVert[|P|]\}$. Typiquement, cet ensemble est égal à {8, 10, 12, 14, 16}. Il peut changer selon les longueurs des paquets de produits verts en inventaire.

Contrainte de contribution minimale

Nous recherchons les patrons de chargement du séchoir qui vont satisfaire la contrainte de contribution minimale suivante :

$$ContributionPatron \geq ContributionMin \tag{3.1}$$

Essentiellement, nous souhaitons générer des patrons de chargement qui possèdent, au minimum, une contribution désignée par l'utilisateur. L'algorithme 4 permet de calculer la contribution d'un patron de chargement.

3.2.3 Trouver une assignation valide de longueurs

Dans cette étape, il faut déterminer quelles seront les longueurs des différents emplacements sur les rangs.

Dans le but de le réaliser, il faut calculer d'autres paramètres que voici :

— Un nombre entier *NbEmplacementParRang* qui représente le nombre d'emplacements qu'on a sur un rang. Il est égal à $\left\lceil \frac{LongueurRang}{Min(LongDiff)} \right\rceil$.

— Un ensemble $LongDiff0 = \{0\} \cup LongDiff$. Certains emplacements doivent pouvoir être de longueur 0 pour respecter la longueur maximale du séchoir.

Variables

Les variables sont définies par la matrice $V = (v_j^i) \in [0..NbEmplacementParRang]^{NbRangs \times |LongDiff0|}$ (i rangs et j différentes longueurs de paquets). Les lignes représentent le nombre de rangs du séchoir. Pour les colonnes, nous en avons un par élément du vecteur *LongDiff0*.

Par exemple, si nous avons un séchoir qui contient deux rangs et que le paramètre *LongDiff0* est $\{0, 8, 10, 12, 14, 16\}$, alors nous avons la matrice de variables :

$$V = \begin{pmatrix} v_0^1 & v_8^1 & v_{10}^1 & v_{12}^1 & v_{14}^1 & v_{16}^1 \\ v_0^2 & v_8^2 & v_{10}^2 & v_{12}^2 & v_{14}^2 & v_{16}^2 \end{pmatrix}$$

Chaque variable sert à déterminer combien d'emplacements de longueur j le rang i contient.

Contraintes

Nous énumérons ici les contraintes requises :

— Pour chaque rang, il faut avoir *NbEmplacementParRang* emplacements en utilisant la contrainte (3.2).

$$\sum_{j \in LongDiff0} v_j^i = NbEmplacementParRang \qquad \forall i \in [1..NbRangs] \tag{3.2}$$

— La longueur des emplacements doit être entre *LongueurMinRang* et *LongueurRang*. Nous utilisons la contrainte (3.3) pour l'imposer.

$$LongueurMinRang \leq \sum_{j \in LongDiff0} j\, v_j^i \leq LongueurRang \quad \forall i \in [1..NbRangs] \quad (3.3)$$

En plus de contraintes, nous devons ajouter un bris de symétrie dans le but d'éviter de retrouver des patrons de chargement symétriques à d'autres précédemment trouvées :

— Les rangs au-dessus doivent être lexicographiquement [3] inférieurs ou égaux aux rangs en dessous en utilisant la contrainte (3.4).

$$v^i \preceq_{LEX} v^{i+1} \qquad \forall i \in [1..(NbRangs - 1)] \quad (3.4)$$

Heuristiques

Pour l'heuristique de choix de variables, nous choisissons les variables de la matrice V de gauche à droite et de haut en bas.

Pour l'heuristique de choix de valeurs, nous choisissons les plus petites valeurs en premier.

Stratégie de recherche

Dans le but de trouver des assignations de longueurs d'emplacements les plus variées possible, nous utilisons *Limited Discrepancy Search* (LDS) [24, 25]. Considérant l'utilisation des déviations pour explorer l'arbre de recherche, nous nous retrouvons à explorer plus rapidement des assignations très différentes les unes des autres. Du fait que nous avons un temps limité pour retrouver le maximum de patrons de chargement, nous souhaitons donc privilégier la variété de celles-ci.

3.2.4 Détecter les assignations d'emplacements équivalentes

À chaque fois qu'une assignation d'emplacements est trouvée (voir 3.2.3), il faut vérifier si une assignation équivalente n'a pas déjà été trouvée.

En effet, malgré l'utilisation de la contrainte (3.4), il existe une possibilité de trouver des assignations d'emplacements de séchoir qui soient équivalentes à d'autres trouvées précédemment. Dans le but de bien expliquer cela, nous donnons un exemple concret que voici :

3. Nous utilisons l'algorithme d'ordre lexicographique de Frisch et coll. [13].

Nous avons un séchoir qui possède 2 rangs et les différentes longueurs possibles pour les emplacements se retrouvent dans l'ensemble $\{0, 8, 10, 12, 14, 16\}$.

Supposons que nous avons d'abord trouvé l'assignation des emplacements suivante :

$$V = \begin{pmatrix} 5 & 2 & 4 & 1 & 0 & 3 \\ 5 & 4 & 2 & 1 & 0 & 3 \end{pmatrix}$$

Nous avons donc, pour le séchoir : 10 emplacements de longueur 0, 6 de longueur 8, 6 de longueur 10, 2 de longueur 12, 0 de longueur 14 et 6 de longueur 16.

Plus tard, l'assignation suivante est trouvée :

$$V = \begin{pmatrix} 5 & 3 & 3 & 1 & 0 & 3 \\ 5 & 3 & 3 & 1 & 0 & 3 \end{pmatrix}$$

Nous avons, encore une fois, pour le séchoir : 10 emplacements de longueur 0, 6 de longueur 8, 6 de longueur 10, 2 de longueur 12, 0 de longueur 14 et 6 de longueur 16.

Nous voyons que nous avons, pour les mêmes longueurs, le même nombre d'emplacements malgré l'application d'un bris de symétrie (formule (3.4)). Nous devons donc passer par dessus une assignation équivalente à une autre déjà traitée. Nous utilisons l'algorithme 3 qui permet de comparer l'assignation des longueurs courantes aux assignations passées. Si l'assignation courante est unique, alors nous l'ajoutons à la liste *AssignationsPrécédentes* et continuons avec les étapes suivantes. Sinon, nous devons trouver une autre assignation de longueurs pour les différents emplacements.

3.2.5 Trouver un patron de chargement avec un algorithme vorace

Nous devons, une fois que nous avons confirmé que nous ne sommes pas en présence d'une assignation de longueurs d'emplacement précédemment traitée, tenter d'empiler des paquets de produits verts dans le séchoir dans le but de voir s'il est possible de créer un patron de chargement qui satisfait la contrainte (3.1).

Pour ce faire, nous devons, en premier lieu, définir l'algorithme 4 qui calcule la contribution d'un patron de chargement. Le calcul considère le nombre de PMP de produits finis qui seront générés avec les produits séchés par le patron de chargement ainsi que la période à laquelle chaque produit fini est demandé.

Algorithme 3 : AssignationEquivExiste(*AssignationsPrécédentes*)

entrée : L'ensemble *AssignationsPrécédentes* qui donne toutes les assignations d'emplacements uniques (non équivalentes) retrouvées précédemment

```
// Créer l'ensemble AssignationCourante qui est composé de couples qui donnent,
   comme premier élément, la longueur et, comme deuxième élément, le nombre
   d'emplacements de cette longueur.
```

$AssignationCourante \leftarrow \{(j,k) \mid k = \sum_{i=1}^{NbRangs} v_j^i\} \quad \forall j \in LongDiff0$

```
// Chercher dans AssignationsPrécédentes si nous n'avons pas déjà
   AssignationCourante
```

si AssignationsPrécédentes $\cap$ {AssignationCourante} $= \emptyset$ **alors**

```
    // Insérer AssignationCourante dans AssignationsPrécédentes et poursuivre le
       reste du traitement
```

    $AssignationsPrécédentes \leftarrow AssignationsPrécédentes \cup \{AssignationCourante\}$

sinon

    Retourner chercher une autre assignation d'emplacements avec la méthode présentée à la section 3.2.3.

Une fois que l'algorithme 4 a été défini, nous pouvons l'utiliser avec un algorithme vorace (voir l'algorithme 5).

Explication de l'algorithme 5

Tous les paquets ont une hauteur standard de 3' pour les cas étudiés. Nous pouvons donc facilement déterminer combien de paquets il faut placer dans une pile avec la formule (3.5).

$$NbPaquetsEmpl = \left\lfloor \frac{HauteurSéchoir}{Min(HautPqtProdVert)} \right\rfloor \tag{3.5}$$

Pour qu'un patron de chargement soit valide, tous les emplacements qui ont une longueur désignée plus grande que 0 doivent obligatoirement avoir *NbPaquetsEmpl* paquets dans leur pile. En effet, considérant que le processus de séchage est long, nous souhaitons maximiser son utilisation.

Il est aussi important d'assurer la stabilité des piles de paquets. C'est pourquoi tous les paquets dans une pile doivent avoir la longueur d'emplacement désignée.

Les paquets de produits verts sont disponibles en quantité limitée. C'est pourquoi il faut aussi considérer l'inventaire lorsqu'on place un paquet dans le séchoir.

Il est important aussi de considérer que les produits verts peuvent être coupés de manière à former des produits finis de plus petites longueurs. Par exemple, un produit

Algorithme 4 : CalculerContributionPatron(*NbPaquetProdVert*)

entrée : Le vecteur *NbPaquetProdVert* de taille $|P|$ qui indique, pour le produit p, quelle quantité de paquets nous avons dans le séchoir

sortie : Le nombre réel *ContributionPatron* qui indique la contribution calculée du patron de chargement

```
// Convertir le nombre de paquets de produits verts en PMP
```
$QtePMPProdVert[p] \leftarrow NbPaquetProdVert[p] \times QtePMPParPaquet[p] \quad \forall p \in P$

```
// Déterminer combien de PMP de produits finis ont été produits par le chargement
```
pour tous les $f \in F$ **faire**
  pour tous les $p \in P$ **faire**
    $QtePMPProdFini[f] \leftarrow QtePMPProdFini[f] + QtePMPProdVert[p] \times Recette[p][f]$

```
// Les demandes d dans D sont classées en ordre croissant de périodes dans le
   but de prioriser la satisfaction des demandes les plus urgentes en premier
   lieu
```
pour tous les $d \in D$ **faire**
  si QtePMPProdFini[ProdDemandé[d]] > 0 **alors**
    $ContributionPatron \leftarrow$
     $ContributionPatron + \min(QtePMPProdFini[ProdDemandé[d]], QteDemandée[d]) \times$
     $\min(HorizonPlanification - PériodeDemandée[d], HorizonPlanification - DuréeSéchage)$
    `// Réduire, dans QtePMPProdFini, la quantité de produits finis que nous
       venons d'utiliser`
    $QtePMPProdFini[ProdDemandé[d]] \leftarrow$
     $QtePMPProdFini[ProdDemandé[d]] - QteDemandée[d]$

retourner ContributionPatron

vert de 16' peut être coupé en deux planches de 8' chaque. C'est pourquoi l'algorithme remplit les emplacements dans l'ordre décroissant de longueur.

Nous pouvons maintenant créer un patron de chargement. À chaque fois que nous pouvons placer un nouveau paquet dans une pile qui n'est pas encore complète, nous calculons pour tous les paquets possibles (de la bonne longueur) la contribution au patron de chargement. Le paquet qui permet de maximiser la contribution est sélectionnée. Nous continuons ainsi jusqu'à ce que la pile soit complète. Une fois que la pile est complète, alors nous débutons une nouvelle pile dans un nouvel emplacement. Nous poursuivons ainsi tant et aussi longtemps qu'il reste des emplacements de longueurs plus grandes que 0.

Une fois que nous avons créé un chargement valide avec l'algorithme vorace 5, il faut vérifier, avec l'algorithme 4, si le vecteur *NbPaquetProdVert* satisfait la contrainte contribution minimale de la section 3.2.2. Si c'est le cas, alors nous pouvons continuer à la prochaine étape (section 3.2.6) puisque nous savons qu'il existe au moins un patron de chargement qui respecte la contrainte de contribution pour l'assignation des emplacements, sinon il faut retourner chercher une autre assignation d'emplacements avec la

Algorithme 5 : RemplirSéchoirVorace(V)

entrée : Le matrice V de taille $NbRangs \times |LongDiff0|$ qui donne pour l'élément v^i_j la quantité d'emplacements de longueur j sur le rang i

sortie : Le vecteur $NbPaquetProdVert$ de taille $|P|$ qui donne, pour le produit p le nombre de paquets dans le patron de chargement

```
1  // Le vecteur Emplacements permet d'énumérer toutes les longueurs disponibles
      dans la matrice V
2  pour tous les i ∈ [1..NbRangs] faire
3      pour tous les j ∈ LongDiff0 faire
4          // Nous ajoutons l'emplacement dans Emplacements autant de fois qu'il y
              en a de la longueur j sur le rang i
5          pour tous les k ∈ [1..v^i_j] faire
6              Emplacements.Ajoute(j)
7  // Trier les emplacements dans l'ordre décroissant de longueur.
8  pour tous les e ∈ Emplacements faire
9      // Il faut empiler NbPaquetsEmpl fois des produits dans l'emplacement e. On
          souhaite empiler le paquet qui, localement, maximise la contribution.
10     pour tous les h ∈ [1..NbPaquetsEmpl] faire
           1. Tester successivement tous les paquets de produits verts de l'ensemble
              {p | LongPqtProdVert[p] = e ∧ QteInventaire[p] > 0}  ∀p ∈ P en incrémentant,
              temporairement, l'élément NbPaquetProdVert[p] et en envoyant le vecteur résultant dans la
              fonction CalculerContributionPatron(NbPaquetProdVert).
           2. Le paquet de produits verts p ayant donné la contribution maximale est celui qui est
              sélectionné pour être empilé. On l'ajoute donc dans le vecteur en incrémentant de 1
              NbPaquetProdVert[p].
           3. Réduire de 1 QteInventaire[p].
           4. Dans le cas où il reste encore des emplacements à remplir, mais que l'ensemble des paquets
              candidats du point 1 est vide, nous avons un échec. Il faut donc retourner et chercher une
              autre assignation d'emplacements avec 3.2.3.
11 retourner NbPaquetProdVert
```

méthode décrite dans la section 3.2.3.

3.2.6 Générer des variantes du patron de chargement de l'étape 3.2.5

À cette étape, nous avons un patron de chargement qui respecte la contribution minimale, produit grâce à l'algorithme vorace 4. Nous savons donc qu'il existe au moins un patron de chargement valide pour l'assignation des emplacements de l'étape 3.2.3. Puisqu'il en existe un, alors il est possible qu'il en existe d'autres aussi qui respectent la contrainte de contribution minimale. C'est pourquoi il est important de chercher des patrons de chargement alternatifs pour les mêmes emplacements.

Une manière de procéder pourrait être de tester toutes les combinaisons possibles de

paquets que nous pouvons placer dans le séchoir selon les emplacements disponibles. Par contre, tout dépendant des paquets de produits verts que nous pouvons placer dans le séchoir, cela prendrait beaucoup trop de temps de trouver toutes les permutations possibles. Le but est d'avoir une très grande variété de patrons de chargement qui ont, au minimum, une contribution désignée par le paramètre en entrée *ContributionMin*.

Pour faciliter le travail, nous pouvons tenter de trouver moins de patrons de chargement, mais augmenter la probabilité de trouver des patrons qui vont satisfaire la contrainte de contribution minimale et explorer rapidement l'espace de solutions ainsi restreinte. La stratégie présentée ici consiste à utiliser le patron de chargement de l'algorithme vorace trouvé à l'étape 3.2.5 et tester toutes les combinaisons pour les paquets une longueur à la fois. Par exemple, si nous avons un patron de chargement qui contient des paquets de longueurs 8, 10 et 16, alors nous pouvons tenter de générer toutes les permutations possibles de paquets de longueurs 8 disponibles en inventaire et, pour chaque permutation, remplacer les paquets de longueurs 8 du patron de chargement de l'étape 3.2.5 pour voir si la contrainte de contribution minimale est respectée. Les paquets de longueurs 10 et 16 restent donc inchangés. Par la suite, nous testons toutes les permutations pour les paquets de longueurs 10 tout en conservant les paquets de longueurs 8 et 16 du patron de chargement de l'algorithme vorace et nous faisons la même chose pour ceux de longueurs 16. L'avantage de procéder ainsi est qu'il est très rapide de générer toutes les permutations de paquets pour une longueur à la fois.

L'algorithme 6 permet de sélectionner une longueur l disponible dans le séchoir et d'effectuer, avec l'algorithme 7, la recherche de toutes les combinaisons de paquets de longueurs l qui sont possibles.

Note sur l'unicité des patrons de chargement retrouvés

Grâce à l'étape 3.2.4, nous garantissons que nous ne traiterons pas plusieurs fois des assignations de longueurs équivalentes. Par contre, de l'effet combiné des algorithmes 6 et 7, il existe une dernière source de répétition des patrons. En effet, le patron trouvé à l'étape 3.2.5 sera répété autant de fois par l'algorithme 7 qu'il y a de longueurs différentes à traiter avec l'algorithme 6. C'est pourquoi il faut s'assurer que le patron trouvé avec l'algorithme vorace ne soit ajouté qu'une seule fois à la liste des patrons.

Algorithme 6 : TrouverPatronsAlternatifs(*NbPaquetProdVert*)

entrée : Le vecteur *NbPaquetProdVert* (provenant de l'algorithme vorace 5) de taille $|P|$ qui indique, pour le produit p, quelle quantité de paquets nous avons dans le séchoir

```
// Déterminer les différentes longueurs de paquets dans NbPaquetProdVert
DiffLongSechoir ←
  {énumération des différentes longueurs de paquets utilisées au moins une fois dans NbPaquetProdVert}
pour tous les l ∈ DiffLongSechoir faire
    pour tous les p ∈ P faire
        si LongPqtProdVert[p] = l alors
            // Nous cherchons combien de paquets de la longueur l sont utilisés
               dans NbPaquetProdVert
            QteRequiseLong ← NbPaquetProdVert[p] + QtePqtLong
            // Nous ajoutons le paquet p dans le vecteur PqtLong
            PqtLong.Ajouter(p)
            // Nous assignons 0 comme étant le nombre de paquets dans
               NbPaquetProdVert
            NbPaquetProdVert[p] ← 0
    // Pour les paquets de longueurs l, nous allons tester toutes les
       combinaisons possibles
    BouclesRecursives(1, PqtLong, QteRequiseLong, NbPaquetProdVert)
    Restaurer NbPaquetProdVert à l'état d'origine
```

3.3 Expérimentations

Dans le but d'évaluer la performance de l'approche PPC, nous le testons sur des cas industriels réels. Nous utilisons des données provenant d'une compagnie canadienne de production du bois d'oeuvre. Un cas représente un séchoir pour lequel nous souhaitons générer des chargements alternatifs. Pour chaque cas, nous avons un carnet de commandes de produits finis, un inventaire initial pour les produits verts ainsi qu'un horizon de planification. L'approche PPC est comparée avec une approche MIP semblable à celle décrite dans les articles [37, 38]. En effet, l'approche MIP peut être modifiée légèrement de manière à chercher non pas un patron original de chargement pour un séchoir, mais bien des variantes de celui-ci. CPLEX, qui est un solveur qui sert à résoudre des problèmes MIP, dispose de la fonction *populate* qui permet de trouver plusieurs solutions alternatives à la solution optimale qui satisfont les différentes contraintes du modèle. L'annexe A présente le modèle MIP qui est utilisé.

Algorithme 7 : BouclesRecursives(*ProdCour*, *PqtLong*, *QteRequiseLong*, *NbPaquetProdVert*)

entrée :

- Le nombre entier *ProdCour* qui indique, dans le vecteur *PqtLong*, auquel nous sommes rendus
- Le vecteur *PqtLong* qui donne, pour les positions 1 à $|PqtLong|$, un paquet $p \in P$. Ces paquets ont tous la même longueur.
- Le nombre entier *QteRequiseLong* qui indique combien de paquets du vecteur *PqtLong* il faut placer dans *NbPaquetProdVert*
- Le vecteur *NbPaquetProdVert* de taille $|P|$ qui indique, pour le produit p, quelle quantité de paquets nous avons dans le séchoir

```
1  // Il faut déterminer pour combien de paquets les prochains peuvent contribuer
2  pour tous les i ∈ [(ProdCour + 1)..|PqtLong|] faire
3      Prochains ← Prochains + QteInventaire[PqtLong[i]]
4  pour tous les j ∈ [Max(QteRequiseLong − Prochains, 0)..
     Min(QteRequiseLong, QteInventaire[PqtLong[ProdCour]])] faire
5      ProchaineQteRequise ← QteRequiseLong − j
6      ProchainProd ← ProdCour + 1
7      NbPaquetProdVert[PqtLong[ProdCour]] ← j
8      si ProchainProd < |PqtLong| alors
9          BouclesRecursives(ProchainProd, PqtLong, ProchaineQteRequise, NbPaquetProdVert)
10     sinon
11         Si NbPaquetProdVert n'est pas un patron retrouvé précédemment, alors il faut le tester
             avec l'algorithme 4. Si le chargement satisfait la contrainte de contribution minimale de
             la section 3.2.2, alors nous l'ajoutons à la liste de patrons de chargement.
```

3.3.1 Configuration des instances industrielles

Pour tous ces cas, nous utilisons la même configuration de séchoir, que voici :

- $LongueurRang = 118$
- $LongueurMinRang = 112$
- $HauteurSéchoir = 12$

Nous définissons aussi *DuréeSéchage* a 5 pour les tous les cas.

Nous avons 5 cas différents de séchoir à 2 rangs avec 3 niveaux de tolérance de contribution ainsi que 3 cas différents de séchoir à 4 rangs, aussi avec 3 niveaux de tolérance. La tolérance reflète de combien un utilisateur est prêt à tolérer une diminution de la contribution du séchoir, pour lequel il faut trouver des chargements alternatifs, par rapport au plan multipériodes original (section 3.1.2). Les tolérances sont précalculées par l'utilisateur et sont reflétées dans le paramètre en entrée *ContributionMin*. En effet, plus la tolérance en pourcentage est haute, plus la borne *ContributionMin* est basse. Nous utilisons, pour tous les cas, les tolérances 1, 5 et 10% dans le but d'évaluer sur une

bonne étendue la performance de l'approche PPC. Nous avons donc 24 tests différents. Pour chacun de ces tests, nous comparons l'approche PPC du présent chapitre avec le modèle MIP. Nous utilisons le solveur présenté au chapitre 2 pour l'approche PPC et le solveur CPLEX avec sa fonction *populate* pour le modèle MIP. Tous les tests ont été effectués sur le même ordinateur [4] et dans les mêmes conditions. Le temps limite pour tous les tests est fixé à 1 800 secondes. Le tableau 3.3 présente les résultats pour les cas utilisant un séchoir à 2 rangs et le tableau 3.4 présente les résultats pour les cas utilisant un séchoir à 4 rangs. Pour chaque test, nous soulignons le résultat de la meilleure approche. Nous donnons aussi les graphiques à nuages de points (voir les graphiques dans la section 3.4) des différents cas étudiés dans le but de voir à quel moment chaque solution a été retrouvée. Dans le but de faciliter la génération de certains graphiques, nous ne plaçons un point qu'à toutes les 10 ou 20 solutions. Le titre de l'ordonnée des graphiques affectés l'indique clairement lorsque c'est le cas.

3.3.2 Résultats

Cas	Technique	1%	5%	10%
1	**PPC**	136	7 309	75 498
	MIP	558	2 970	11 610
2	**PPC**	60 076	10 094 028	12 436 766
	MIP	13 959	4 654	30 502
3	**PPC**	237	1 669	3 543
	MIP	1 741	3 003	4 428
4	**PPC**	78 808	2 698 826	3 883 092
	MIP	22 502	2 934	81 970
5	**PPC**	12 102	506 038	4 255 660
	MIP	47 902	1 140	24 725

TABLE 3.3 – Nombre total de patrons trouvées en 1 800 secondes pour 5 cas et selon différentes tolérances. Le séchoir possède 2 rangs.

4. i5-2.60 GHz avec 8 Gb de mémoire vive

Cas	Technique	1%	5%	10%
1	**PPC**	309	165 969	1 793 425
	MIP	20	6 555	4 171
2	**PPC**	900 600	9 346 540	13 032 357
	MIP	12 536	39 439	3 333
3	**PPC**	9 703 307	14 718 614	15 614 287
	MIP	43 858	7 657	40 516

TABLE 3.4 – Nombre total de patrons trouvées en 1 800 secondes pour 3 cas et selon différentes tolérances. Le séchoir possède 4 rangs.

3.4 Discussion

Nous constatons que pour les cas à 2 rangs nous avons 10 tests sur 15 pour lesquels l'approche PPC est meilleure et, pour les cas à 4 rangs, tous les tests montrent l'avantage de la PPC sur MIP. Nous analysons la performance de l'approche PPC pour les cas à 2 rangs et 4 rangs. Par la suite, nous faisons une analyse générale en ce qui concerne l'utilisation de l'approche MIP. Finalement, nous donnons les constatations générales en ce qui concerne l'utilisation de l'approche PPC dans le but de trouver des patrons de chargement alternatifs. Il est important de noter, pour les analyses, qu'un patron de chargement est dit *équivalent* à un autre lorsque nous avons les mêmes paquets de produits verts en même quantité.

3.4.1 Analyse des cas à 2 rangs

Pour les cas à 2 rangs, à l'exception du cas 2 avec les tolérances de 5% et 10%, le solveur a exploré toutes les combinaisons possibles bien avant le temps limite de 1 800 secondes (voir les graphiques 3.7 à 3.11). En regardant la majorité des graphiques, nous voyons que la PPC produit beaucoup plus de solutions très rapidement. Cela s'explique par le fait que, pour 2 rangs, il existe peu de combinaisons différentes d'assignation de longueurs d'emplacements possible pour satisfaire les différentes contraintes de la section 3.2.3. Pour les tests où l'approche PPC trouve moins de solutions que l'approche MIP, un élément intéressant à analyser est combien de temps cela prend aux deux approches pour trouver le même nombre de patrons. En effet, nous avons essentiellement deux buts pour la recherche de patrons de chargement alternatifs : nous souhaitons trouver des patrons variés et nous souhaitons les trouver rapidement. C'est pourquoi, pour les 5 tests pour lesquels l'approche PPC trouve moins de patrons que l'approche MIP, nous montrons, dans le tableau 3.5, le temps que cela a pris pour trouver x patrons où la variable x est le nombre total de patrons trouvés par l'approche PPC.

Test	x^e patron	Technique	Temps (secondes)
Cas 1 - 1%	**136**	**PPC**	10,76
		MIP	16,21
Cas 3 - 1%	**237**	**PPC**	15,02
		MIP	7,18
Cas 3 - 5%	**1 669**	**PPC**	23,06
		MIP	1 345,88
Cas 3 - 10%	**3 543**	**PPC**	23,12
		MIP	188,89
Cas 5 - 1%	**12 102**	**PPC**	47,22
		MIP	101,04

TABLE 3.5 – Temps requis par les deux approches pour trouver le même nombre de patrons. La variable x est égale au nombre total de patrons trouvés par l'approche PPC pour le test indiqué à la première colonne.

Ce que nous voyons est que, sauf pour le cas 3 à 1%, l'approche MIP prend plus de temps que l'approche PPC pour trouver le même nombre de patrons. C'est important puisque la quantité de temps que l'utilisateur laisse aux diverses approches pour trouver des patrons peut varier. Dans nos tests, nous laissons 30 minutes pour chercher le maximum de patrons dans un assez grand temps, mais il ne serait pas réaliste, pour un employé d'une usine, d'attendre tout ce temps-là avant que le programme lui fournisse des patrons de chargement. C'est pourquoi il est intéressant de constater que pour 4 des 5 cas, l'approche PPC trouve plus rapidement des solutions au même point.

Un autre élément à considérer est qu'il existe des tests où la recherche de l'approche PPC est complète, mais qu'on trouve moins de patrons que l'approche MIP. Les tests du cas 3 sont des exemples de cette situation. Cela s'explique par le fait, en premier lieu, qu'il n'existe aucune garantie que l'algorithme vorace de la section 3.2.5 va nécessairement trouver, pour une assignation de longueurs d'emplacements donnée, un chargement qui permette de respecter la contrainte de contribution minimale (s'il en existe un). Nous utilisons un algorithme vorace pour sa rapidité. En deuxième lieu, même si l'algorithme vorace pouvait le faire, il ne faut pas oublier que la génération de patrons de chargement alternatifs de la section 3.2.6 ne produit volontairement qu'un sous-ensemble des combinaisons de paquets que nous pouvons placer dans le séchoir dans le but de favoriser la rapidité et non pas l'énumération exhaustive des patrons. Ces deux raisons expliquent pourquoi malgré le fait que l'approche PPC ait exploré toutes les assignations de longueurs d'emplacements très rapidement, l'approche MIP produit quand même de meilleurs résultats en 30 minutes.

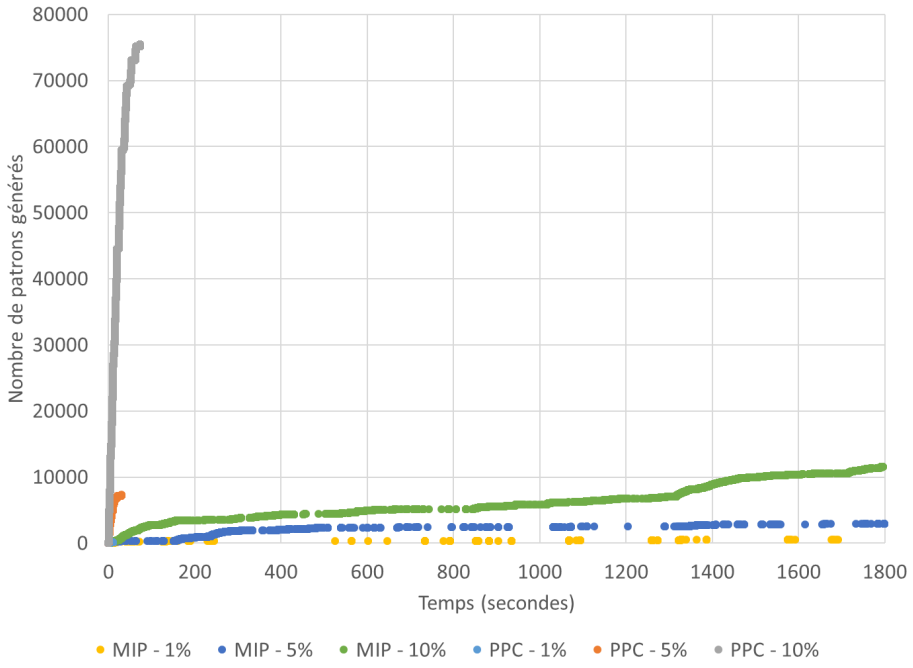


FIGURE 3.7 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #1 - 2 rangs

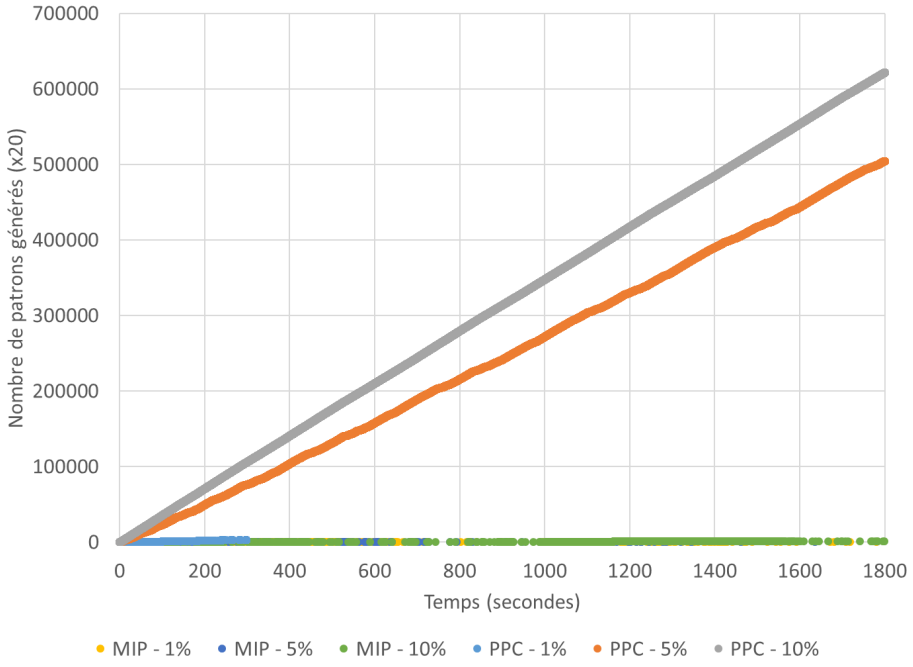


FIGURE 3.8 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #2 - 2 rangs

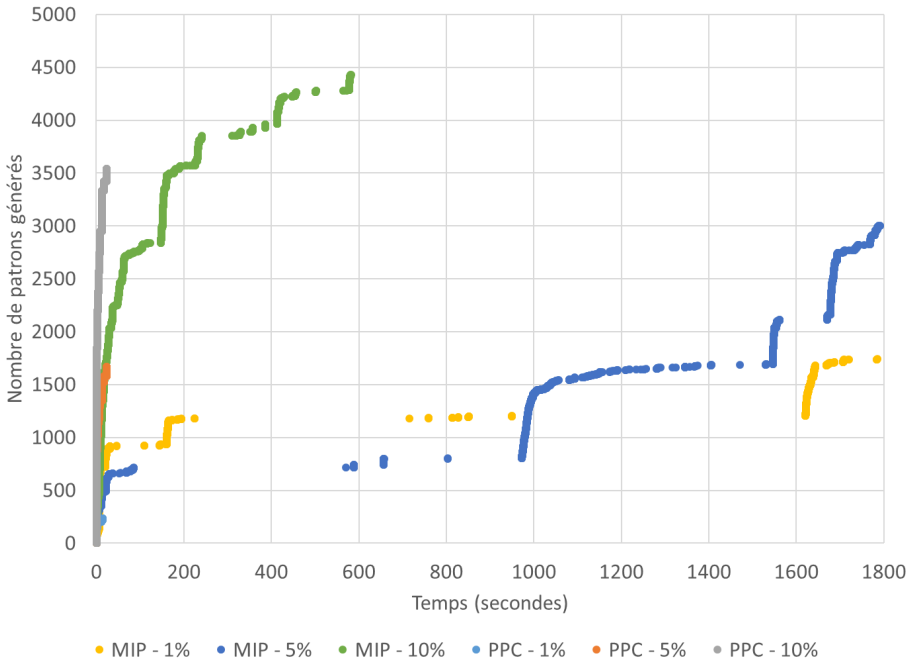


FIGURE 3.9 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #3 - 2 rangs

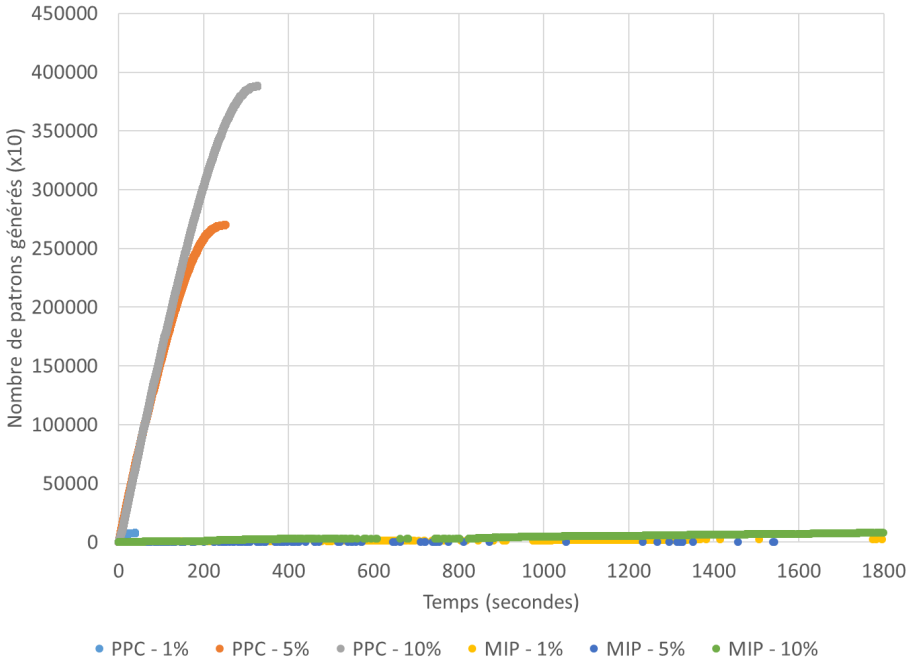


FIGURE 3.10 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #4 - 2 rangs

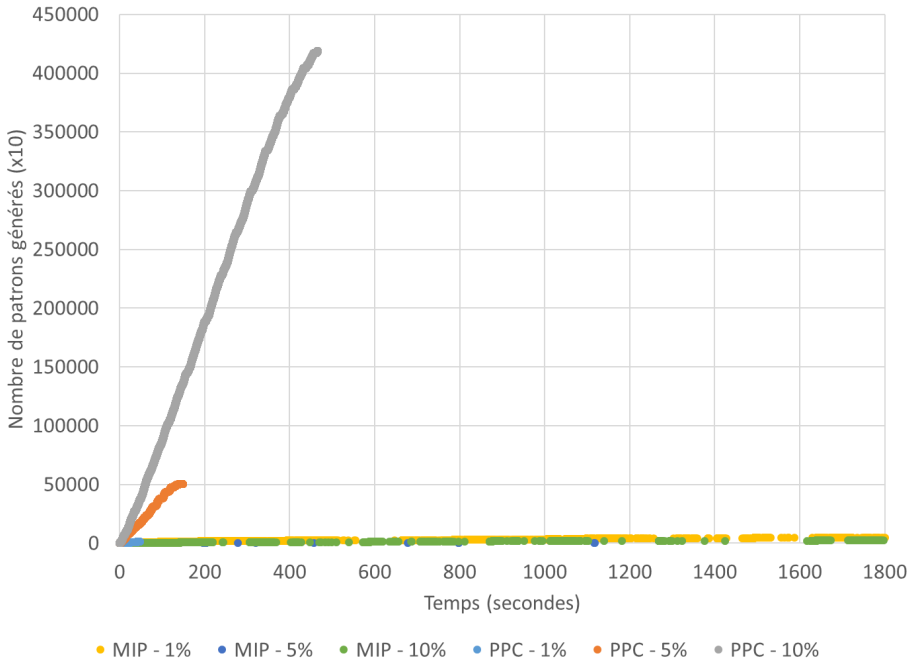


FIGURE 3.11 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #5 - 2 rangs

3.4.2 Analyse des cas à 4 rangs

Pour les cas à 4 rangs, nous voyons que, pour tous les tests, l'approche PPC est meilleure que l'approche MIP. Nous constatons donc que l'approche PPC semble avoir de la facilité à générer des patrons de chargement lorsque nous augmentons le nombre d'emplacements sur lesquels nous pouvons placer des paquets de produits verts (donc une augmentation de combinaisons possibles). En regardant les graphiques 3.13 et 3.14, nous constatons que l'approche PPC trouve des patrons de chargement assez linéairement. Par contre, lorsque nous regardons le graphique 3.12, nous constatons qu'il y a des interruptions. Par exemple, pour les points de PPC - 5%, nous constatons qu'il se passe des moments où l'approche ne trouve aucune solution et d'autres où elle en trouve un certain nombre assez rapidement. Cela s'explique par le fonctionnement même de l'approche PPC présenté dans ce chapitre. En effet, lorsque nous trouvons une assignation de longueurs pour les emplacements avec l'étape 3.2.3, il faut déterminer s'il est possible de trouver au moins un patron de chargement, avec l'algorithme vorace, qui respecte la contrainte de contribution minimale. Cela peut prendre un certain temps avant que l'algorithme trouve une assignation d'emplacements qui permette de construire ce patron de chargement. Par contre, lorsque ça arrive, nous pouvons généralement assez facilement et rapidement trouver des variantes de ce patron de chargement. Nous

pouvons donc dire que l'étape la plus complexe dans le processus est de trouver une assignation de longueurs qui permette de générer, au minimum, un patron de chargement respectant la contrainte de contribution minimale.

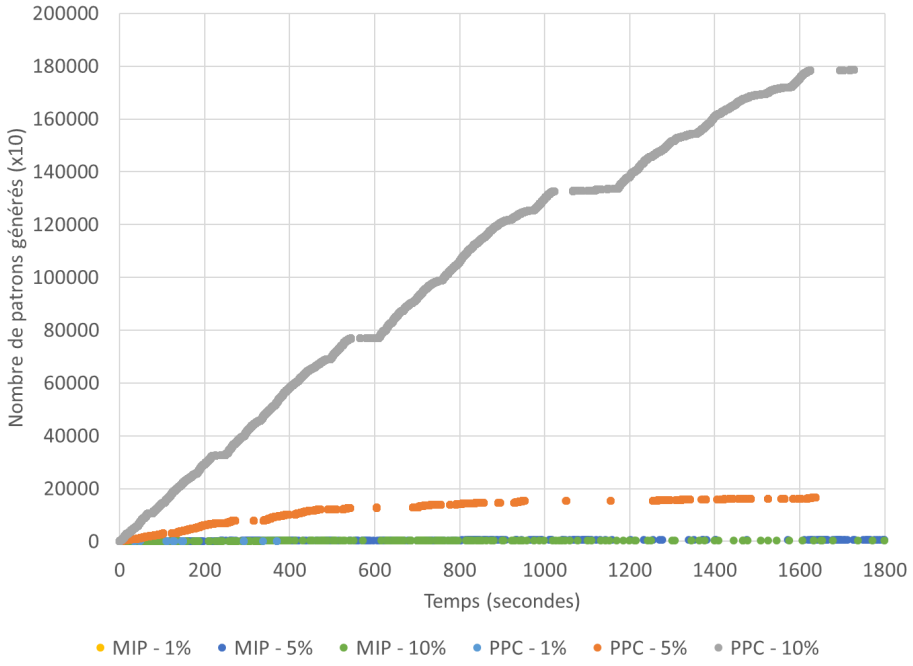


FIGURE 3.12 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #1 - 4 rangs

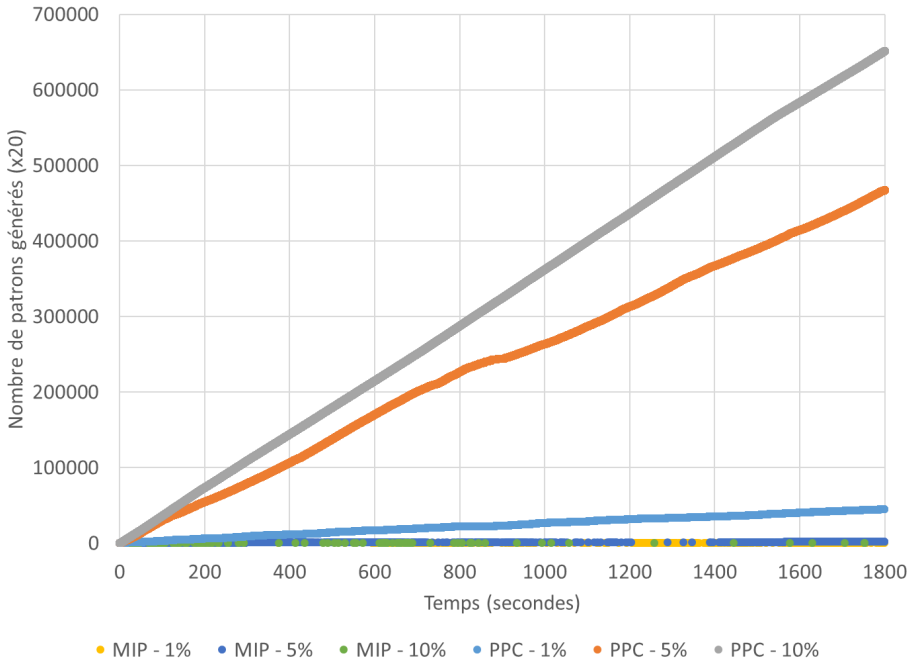


FIGURE 3.13 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #2 - 4 rangs

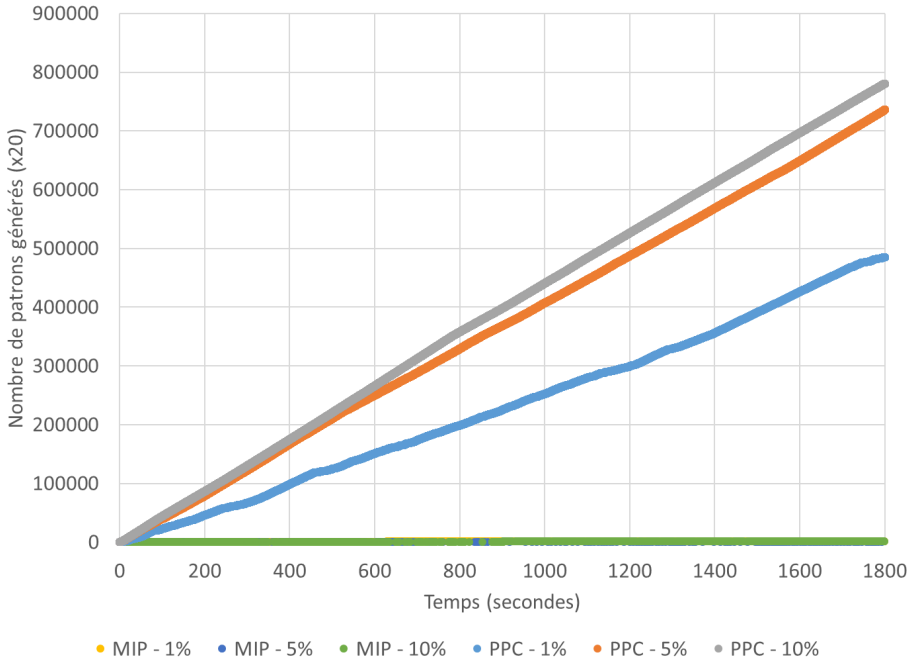


FIGURE 3.14 – Nombre de patrons générés par les approches PPC et MIP en fonction du temps en secondes pour les tolérances 1, 5, 10% pour le cas #3 - 4 rangs

3.4.3 Analyse générale de l'approche MIP

Nous constatons qu'à quelques reprises l'approche MIP donne moins de résultats lorsque le niveau de tolérance augmente. Un exemple est le cas 2 dans le tableau 3.3 où le test avec la tolérance de 1% trouve plus de solutions que le test avec la tolérance de 5%. Cela s'explique par le fonctionnement de *populate* de CPLEX ainsi que le but de la recherche qui est de trouver des patrons de chargement différents (non équivalents). En effet, *populate* cherche à générer des solutions alternatives à la solution optimale en faisant varier toutes les variables du modèle (voir les variables à l'annexe A). Or, nous cherchons des patrons uniques, ce qui est caractérisé par le nombre de paquets de produits verts que nous plaçons dans le séchoir. Il existe des cas où il est possible de changer certaines variables tout en retrouvant un patron équivalent à un autre précédemment retrouvé. C'est donc du temps qui a été perdu puisque ce patron ne sera pas inclus dans la liste des solutions. C'est ce qui explique que pour certains cas, une tolérance supérieure semble générer, en 30 minutes, moins de patrons qu'une tolérance inférieure.

3.4.4 Constatations générales

La tendance qui semble se dégager de tous ces résultats est que l'approche PPC est meilleure que l'approche MIP pour trouver plus de patrons de chargement plus rapidement dans la majorité des tests effectués (19 tests sur 24). Pour les 5 tests restants, nous avons vu que pour 4 d'entre eux l'approche PPC est plus rapide que l'approche MIP pour se rendre au même nombre de patrons. Par contre, cela ne veut pas dire que l'approche MIP ne peut pas apporter de contribution en ce qui concerne la génération de patrons de chargements alternatifs pour un séchoir. Les résultats montrent que l'approche MIP performe très bien lorsque le séchoir possède peu de rangées ou encore que la tolérance est faible, alors que l'approche PPC performe mieux lorsque le séchoir possède plusieurs rangées ou que la tolérance est haute. C'est pourquoi il faut voir l'approche PPC comme étant un outil parmi tant d'autres pour la résolution du problème expliqué dans ce chapitre. Essentiellement, plus il est possible d'utiliser différentes approches en parallèle et plus il y a de chances que chaque approche puisse contribuer à compenser pour les faiblesses des autres.

3.4.5 Pistes pour l'amélioration de l'approche PPC

L'approche présentée ici peut être améliorée afin de la rendre potentiellement plus performante tant en termes de rapidité que de nombre de patrons trouvés. Par exemple, il pourrait être possible d'utiliser des meilleures heuristiques de choix de variables et valeurs pour l'étape d'assignation de longueurs d'emplacements décrite à la section 3.2.3. Ces heuristiques permettraient d'explorer, en priorité, les assignations d'emplacements qui donneraient plus de chance de générer des patrons de chargements respectant la contrainte de contribution minimale. De plus, pour l'étape de génération de patrons alternatifs, il pourrait être possible, en supposant que nous avons plusieurs processeurs, d'utiliser les algorithmes de parallélisation des stratégies de recherche [40, 41] dans le but d'effectuer une recherche exhaustive des paquets de produits verts à placer sur les divers emplacements dans un temps raisonnable (donc remplacer l'étape 3.2.6). Ce sont des pistes de recherche qui permettraient d'améliorer davantage la performance de l'approche PPC.

3.5 Conclusion

Dans ce chapitre, nous avons démontré qu'il est possible d'utiliser le solveur présenté au chapitre 2 dans le but de résoudre un problème concret des partenaires industriels de FORAC. Nous obtenons, règle générale, de très bons résultats en ce qui concerne la génération de patrons de chargement alternatifs pour un séchoir par rapport à l'approche MIP.

Chapitre 4

Implémentation d'un nouvel événement : lorsque la procrastination paie

Dans ce chapitre, nous nous intéressons au meilleur moment pour effectuer le filtrage lorsque nous résolvons un *Balanced Incomplete Block Design* (BIBD) en utilisant les techniques de PPC. Le problème du BIBD possède plusieurs applications importantes telles que le design de plans d'expérience, la planification d'horaires de tournois ou la théorie du codage [34, 61, 68]. Il peut être résolu en utilisant différentes approches telles que la recherche locale [52], la programmation en nombres entiers [68] ou la PPC [51, 53].

Nous essayons une nouvelle approche que nous appelons le filtrage tardif (*lazy filtering*). Nous retardons le filtrage du domaine de chaque variable jusqu'au dernier moment, c'est-à-dire, lorsque la variable a été sélectionnée et qu'elle est sur le point d'être instanciée. Pour ce faire, nous modifions le solveur de manière à ajouter un nouvel événement qui sera décrit plus tard.

Le reste de ce chapitre est organisé ainsi. La section 4.1 présente les concepts préliminaires à propos des BIBDs. La section 4.2 montre comment nous pouvons effectuer un filtrage tardif. La section 4.3 décrit comment les contraintes du BIBD peuvent être adaptées pour le filtrage tardif. La section 4.4 montre les résultats des expériences. La section 4.5 discute des résultats obtenus. La section 4.6 conclut le chapitre.

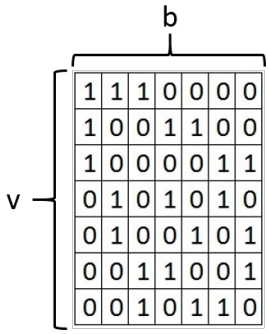


FIGURE 4.1 – Représentation binaire

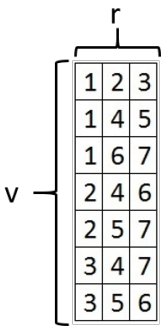


FIGURE 4.2 – Représentation n-aire

4.1 Balanced Incomplete Block Design

Un problème BIBD est défini par le tuple (v, b, r, k, λ). Nous avons v types d'objets qui doivent être assignés à b blocs de manière à ce que chaque bloc contienne k objets de différents types. Chaque type d'objets est présent dans r blocs différents. Pour n'importe quel paire de type d'objets, ils doivent apparaître simultanément dans λ différents blocs [50].

Par définition, les instances BIBDs doivent satisfaire les propriétés suivantes [61] :

1. $b = \frac{\lambda(v^2-v)}{k^2-k}$
2. $r = \frac{\lambda(v-1)}{k-1}$

De ce fait, une instance BIBD peut être définie en utilisant seulement les paramètres (v, k, λ).

Le problème peut être modélisé en utilisant une matrice de variables binaires ayant v lignes et b colonnes $X = (x_{ij}) \in \{0, 1\}^{v \times b}$, où x_{ij} est égal à 1 *ssi* un objet du type i est assigné au bloc j. Il peut aussi être modélisé en utilisant une matrice $Y = (y_{ij}) \in [1..b]^{v \times r}$ où pour chaque ligne, correspondant à un type d'objet i, nous indiquons dans quels r blocs de 1 à b les objets de ce type sont placés. Ces deux représentations, mathématiquement équivalentes, sont illustrées par les figures 4.1 et 4.2 qui montrent une même solution pour l'instance $(7, 3, 1)$.

Les trois premières lignes du tableau 4.1 sont les contraintes requises pour représenter un BIBD sous la forme d'une matrice (versions binaire et n-aire). La représentation d'un BIBD sous forme matricielle amène de la symétrie qui est importante à considérer. En effet, pour n'importe quelle solution que nous trouvons, il existe $v! \times b!$ autres solutions symétriques [10, 14]. C'est pourquoi il est important d'utiliser des stratégies de bris de symétrie comme celles indiquées par les contraintes 4 et 5 du tableau 4.1.

	Binaire	***n*-aire**
1. Chaque type d'objet doit apparaître dans r blocs	$\sum_{j=1}^{b} x_{ij} = r \quad \forall i \in [1..v]$	$y_{ij} < y_{i,(j+1)} \quad \forall i \in [1..v]$ $\forall j \in [1..(r-1)]$ Cette contrainte brise partiellement la symétrie de colonnes.
2. Chaque bloc doit contenir exactement k objets distincts	$\sum_{i=1}^{v} x_{ij} = k \quad \forall j \in [1..b]$	$GCC(Y, [k_1, \ldots, k_b], [k_1, \ldots, k_b])$
3. Deux objets distincts doivent apparaître ensemble dans λ blocs	$\sum_{j=1}^{b} x_{ij}x_{lj} = \lambda$ $\forall i, l \in [1..v]$ *et* $i < l$	$NValue([y_{i1}, \ldots, y_{ir}, y_{l1}, \ldots, y_{lr}],$ $2r - \lambda) \quad \forall 1 \leq l < i \leq v$
4. Ordre lexicographique sur les lignes	$x_{i,1}, \ldots, x_{i,b} \prec_{LEX}$ $x_{(i+1),1}, \ldots, x_{(i+1),b}$ $\forall i \in [1..(v-1)]$	$y_{i,1}, \ldots, y_{i,r} \prec_{LEX}$ $y_{(i+1),1}, \ldots, y_{(i+1),r}$ $\forall i \in [1..(v-1)]$
5. Ordre lexicographique sur les colonnes	$x_{1,j}, \ldots, x_{v,j} \succeq_{LEX}$ $x_{1,(j+1)}, \ldots, x_{v,(j+1)}$ $\forall j \in [1..(b-1)]$	Nous pouvons facilement implémenter une heuristique de choix de valeur qui fait cela. Si nous avons le choix entre plusieurs valeurs différentes qui ont, jusqu'à maintenant, été utilisées sur les mêmes lignes, alors nous utilisons la plus petite valeur possible.

TABLE 4.1 – Contraintes pour BIBD binaire vs n-aire. Les contraintes GCC et NValue sont définies dans [55] et [46] respectivement. La contrainte GCC est définie comme étant $GCC([X_1...X_n], [l_1...l_m], [u_1...u_m])$ où $\forall x \quad l_x \leq |\{i \mid X_i = x\}| \leq u_x$. La contrainte NValue est définie comme étant $NValue([X_1...X_n], x) \iff |\{X_1...X_n\}| = x$. GCC et NValue sont implémentés de manière simplifiée dans 4.3.2 ainsi que 4.3.3 considérant l'ordonnancement statique des variables que nous utilisons.

4.2 Filtrage tardif

Il existe généralement un compromis à faire entre le temps passé à explorer un arbre de recherche et le temps passé à filtrer cet arbre en utilisant les algorithmes de filtrage. Ce n'est pas le sujet de notre recherche présentée ici. Plutôt, nous montrons que, pour un BIBD, il est possible, en remettant le filtrage au dernier moment, de conserver le

même niveau de filtrage tout en diminuant le temps de résolution.

Nous filtrons le domaine d'une variable au tout dernier moment, c'est-à-dire, entre le moment où la variable est sélectionnée et juste avant que l'heuristique de choix de valeur soit appelée. De ce fait, nous ajoutons un événement de filtrage au solveur qui s'appelle *variableAÉtéSélectionnée*. C'est une pratique plutôt inhabituelle puisque les solveurs préfèrent filtrer le plus tôt possible, c'est-à-dire, au moment où le domaine d'une autre variable est modifié.

Pour implémenter cet événement dans un solveur, il faut se souvenir que nous avons une heuristique de choix de variables qui détermine quelle sera la prochaine variable que l'algorithme de recherche doit instancier (voir le chapitre 1). Immédiatement après le moment où la prochaine variable est sélectionnée par l'heuristique, un appel à la méthode *NotifyVariableHasBeenSelected* est effectué par l'algorithme de la stratégie de recherche (se référer à la figure 2.1 du diagramme de classe du solveur). Cette méthode est responsable de notifier toutes les contraintes qui ont la variable sélectionnée dans leur portée pour leur demander de réagir à l'événement. Pour chaque contrainte ainsi notifiée, si un algorithme de filtrage a été implémenté pour répondre à l'événement, alors nous le déclenchons. Le diagramme de séquence de la figure 4.3 montre la séquence des appels aux différentes méthodes du solveur présenté au chapitre 2.

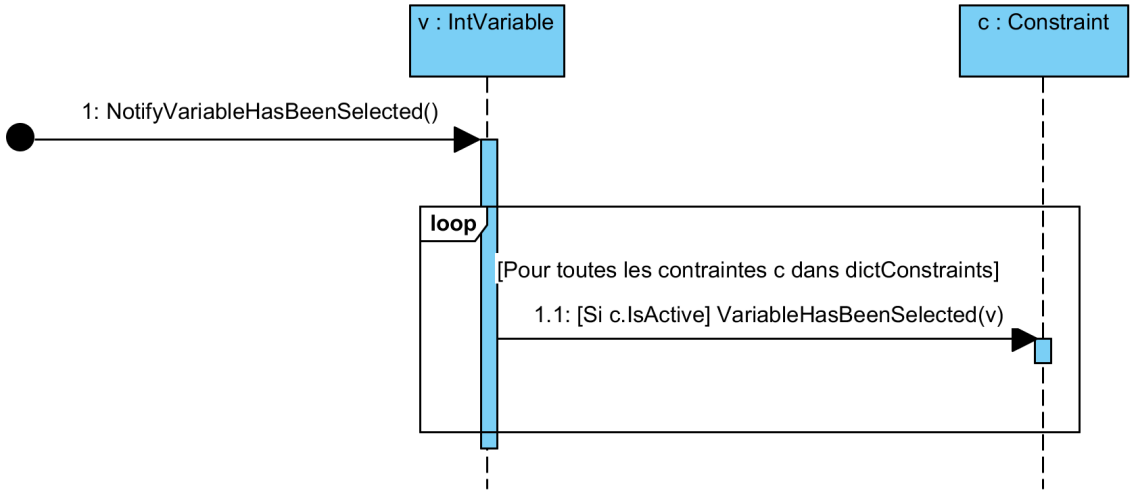


FIGURE 4.3 – Diagramme de séquence lorsqu'une variable est notifiée qu'elle vient d'être sélectionnée

4.3 Filtrage tardif pour le BIBD

Les sections 4.3.1 à 4.3.5 présentent comment les algorithmes de filtrages pour le BIBD peuvent être implémentés. Il est important de remarquer qu'un algorithme est différent selon que l'algorithme de filtrage se déclenche tardivement (*variableAÉtéSélectionnée*) ou avec les événements classiques (bornes de la variable modifiées ou variable instanciée). Cette différence est due au fait que, dans la version traditionnelle, nous pouvons modifier les domaines de n'importe quelle variable de la matrice, alors qu'en version tardive, nous ne modifions que le domaine de la variable qui vient d'être sélectionnée. Dans tous les cas, nous obtenons le même niveau de filtrage, mais avec des temps de résolution différents que nous allons évaluer dans la section 4.4.

Nous présentons les algorithmes et les résultats d'expériences pour la représentation n-aire des BIBD puisqu'ils étaient plus faciles à adapter avec notre technique. Nous donnons aussi l'analyse de la complexité des algorithmes. Des algorithmes équivalents existent pour le modèle binaire. L'implémentation des algorithmes de filtrage décrits suppose que les variables sont instanciées selon une séquence statique (de la gauche vers la droite, du haut vers le bas). Il s'agit d'une pratique courante pour la résolution du BIBD [1]. De même, nous instancions les variables en choisissant d'abord les plus petites valeurs [2].

4.3.1 Chaque type d'objet doit apparaître dans r blocs

Cette contrainte force les valeurs d'une ligne à prendre une valeur différente. On y arrive en imposant un ordre croissant. Cela permet aussi, partiellement, de briser une forme de symétrie.

Filtrage traditionnel

Dans une ligne i où la variable y_{ij} a été instanciée, nous filtrons le domaine de toutes les variables $l > j$ en utilisant la formule (4.1). La complexité de cet algorithme est $O(r)$.

$$y_{ij} < y_{il} \qquad l \in [(j+1)..r] \tag{4.1}$$

1. Choco [54] et Gecode [63] utilisent des heuristiques statiques pour la résolution du BIBD dans les fichiers exemples de leurs solveurs.

2. Par définition, les valeurs ne sont pas réellement *numériques*, mais correspondent en réalité à des identifiants de blocs où on souhaite placer les objets.

Filtrage tardif

Lorsque la variable y_{ij} est sélectionnée, nous avons seulement besoin de retirer du domaine courant les valeurs inférieures à la valeur de $y_{i,(j-1)}$ en utilisant la formule (4.2). La complexité de cet algorithme est $O(1)$.

$$y_{i,(j-1)} < y_{ij} \tag{4.2}$$

Avec cette version, chaque variable de la ligne n'est filtrée qu'une seule fois, alors qu'avec le filtrage traditionnel, les variables de la dernière colonne sont filtrées jusqu'à $r-1$ fois. Cependant, pour les deux algorithmes, le même nombre de valeurs sont retirées.

4.3.2 Chaque bloc doit contenir exactement k objets distincts

Cette contrainte requiert que chaque valeur apparaisse k fois dans la matrice. Dans la version traditionnelle, nous devons vérifier l'ensemble de la matrice lorsqu'une variable est instanciée. Dans la version tardive, nous avons seulement besoin de vérifier les variables précédemment instanciées.

Filtrage traditionnel

Lorsqu'une variable y_{ij} est instanciée, nous devons vérifier, pour toute la matrice, le nombre de fois que la valeur de y_{ij} est utilisée. Si la valeur a été utilisée k fois, alors nous devons retirer du domaine des variables qui suivent la valeur de y_{ij}. La complexité de cet algorithme est $O(vrb)$.

Filtrage tardif

Lorsque la variable y_{ij} est sélectionnée, nous devons vérifier, seulement pour les variables précédemment instanciées, combien de fois chaque valeur est instanciée. Par la suite, nous retirons du domaine de la variable y_{ij} les valeurs qui sont utilisées k fois. La complexité de cet algorithme, en pire cas, est $O(vr+b)$.

4.3.3 Deux objets distincts doivent simultanément apparaître dans λ blocs

Dans la version traditionnelle, il est nécessaire de vérifier les $v-1$ paires de lignes que la ligne courante peut former avec les autres lignes. Dans la version tardive, il est seulement nécessaire de vérifier les paires de lignes que la ligne courante peut former avec les lignes précédentes.

Filtrage traditionnel

À partir de la ligne i à laquelle la variable y_{ij} qui vient d'être instanciée appartient, nous devons considérer les $v-1$ paires de lignes formées avec les autres lignes.

Pour chaque paire, nous devons calculer, pour les variables déjà instanciées, le nombre de fois que chaque valeur est utilisée. Si pour une paire nous avons exactement λ valeurs qui sont utilisées deux fois, alors nous devons retirer de chaque ligne qui forment la paire les valeurs des variables instanciées de l'autre ligne. Si nous avons plus que λ valeurs utilisées deux fois, alors il faut forcer un échec. La complexité de cet algorithme est $O(vrb)$.

Filtrage tardif

Lorsque la variable y_{ij} est sélectionnée, nous devons, pour chaque $i-1$ paires qu'on peut former avec les lignes précédentes, calculer le nombre de fois que chaque valeur est utilisée. Si nous avons exactement λ valeurs utilisées deux fois, alors nous devons retirer du domaine de la variable sélectionnée y_{ij} les valeurs des variables instanciées de l'autre ligne de la paire. La complexité de cet algorithme est $O(r+b)$ dans le meilleur cas lorsqu'on filtre la première ligne avec la deuxième et le pire cas est $O(vr+b)$ lorsqu'on filtre la dernière ligne avec les autres au-dessus.

4.3.4 Ordre lexicographique entre les lignes

Filtrage traditionnel

Pour le filtrage traditionnel, nous utilisons l'algorithme d'ordre lexicographique strict de Frisch et coll. [13] qui se déclenche lorsqu'une borne (supérieure ou inférieure) d'une variable est modifiée. La complexité de cet algorithme est $O(rb)$.

Filtrage tardif

Puisqu'on souhaite uniquement filtrer la variable y_{ij} qui vient d'être sélectionnée, l'algorithme peut être simplifié. Si une variable à la deuxième ligne ou plus bas est sélectionnée, alors il faut vérifier si elle est dans la première colonne ou non. Si elle est dans la première colonne, alors nous filtrons simplement son domaine dans le but de retirer les valeurs plus petites à la valeur de la variable juste au-dessus. Sinon, alors il faut imposer l'ordre lexicographique tant et aussi longtemps que les valeurs des variables à gauche de y_{ij} sont égales aux valeurs de la variable juste au-dessus (ligne $i-1$). Tout

dépendant si y_{ij} est à la dernière colonne ($j = r$) ou non, la manière de filtrer son domaine change légèrement. Cette différence s'explique par le fait que lorsqu'on arrive à la dernière colonne, il ne reste plus aucune chance de pouvoir forcer l'ordre lexicographique. Voir l'algorithme 8 pour le pseudo-code. La fonction *valeur* retourne la valeur instanciée d'une variable. La complexité de cet algorithme est $O(r + b)$.

Algorithme 8 : LexLigneTardif

Entrée : La variable y_{ij} qui vient d'être sélectionnée

```
1  si i > 1 alors
2      si j > 1 alors
3          si valeur(y_{(i-1),l}) = valeur(y_{il})   ∀l ∈ [1..(j − 1)] alors
4              si j ≠ r alors
5                  dom(y_{ij}) = dom(y_{ij}) \ {x | x < valeur(y_{(i−1),j})}
6              sinon
7                  dom(y_{ij}) = dom(y_{ij}) \ {x | x ≤ valeur(y_{(i−1),j})}
8      sinon
9          // La première colonne de la matrice doit être en ordre non décroissant
10         dom(y_{ij}) = dom(y_{ij}) \ {x | x < valeur(y_{(i−1),j})}
```

4.3.5 Ordre lexicographique entre les colonnes

Filtrage traditionnel

Pour chaque variable qui a été instanciée dans la matrice, nous devons vérifier dans quelles lignes chaque valeur a été utilisée. Pour les valeurs qui ont été utilisées sur les mêmes lignes, nous devons seulement conserver la plus petite valeur dans le domaine de la variable suivant la variable qui vient d'être instanciée. La complexité de cet algorithme est $O(v(r + b^2))$.

Filtrage tardif

Pour chaque variable qui a été instanciée, nous devons vérifier dans quelles lignes chaque valeur a été utilisée. Pour les valeurs qui ont été instanciées sur les mêmes lignes, nous devons conserver uniquement la valeur la plus petite dans le domaine de la variable sélectionnée. La complexité de cet algorithme est $O(v(r + b^2))$.

4.4 Expérimentations

Dans cette section, nous comparons le filtrage selon qu'il réponde au filtrage classique ou tardif. Nous procédons d'abord à l'évaluation pour les BIBDs. Ensuite, à l'aide

d'un contre-exemple (remplissage d'une matrice où nous n'appliquons que la contrainte LEX), nous montrons que le filtrage tardif n'est évidemment pas approprié pour tous les problèmes.

4.4.1 BIBDs

Instance	Type algo	Temps (s)	#Échecs	#sols
(6, 3, 2)	Filtrage traditionnel	0,097	486	1
	Filtrage tardif	0,076	486	1
(7, 3, 2)	Filtrage traditionnel	2,227	11 817	12
	Filtrage tardif	1,579	11 817	12
(9, 3, 1)	Filtrage traditionnel	1,077	6 439	2
	Filtrage tardif	0,676	6 439	2
(8, 4, 3)	Filtrage traditionnel	142,989	739 695	92
	Filtrage tardif	102,949	739 695	92
(6, 3, 4)	Filtrage traditionnel	27,280	79 994	21
	Filtrage tardif	21,625	79 994	21
(11, 5, 2)	Filtrage traditionnel	19,511	140 530	2
	Filtrage tardif	13,153	140 530	2
(7, 3, 3)	Filtrage traditionnel	349,753	941 904	220
	Filtrage tardif	266,021	941 904	220

TABLE 4.2 – Résultats selon qu'on filtre traditionnellement ou tardivement

Nous utilisons des instances qui proviennent de Flener et coll. [10]. Les algorithmes ont été implémentés dans le solveur présenté au chapitre 2 étant donné que le filtrage tardif était difficile à implémenter dans un solveur générique.

L'ordinateur utilisé pour les tests est un i5-2.60 GHz avec 8 Gb de mémoire vive.

Pour rappel, les variables sont instanciées en ordre lexicographique (gauche à droite, haut à bas). L'heuristique de choix de valeur sélectionne toujours la plus petite valeur possible en premier. Une stratégie de retours arrière chronologique (DFS) est appliquée en cas d'échec. Le tableau 4.2 montre le temps requis pour explorer complètement les arbres de recherche. Nous présentons aussi le nombre d'échecs ainsi que le nombre de solutions trouvées pour chaque instance BIBD. Comme nous pouvons le voir, le nombre d'échecs est exactement le même selon qu'on utilise une approche ou une autre. Par contre, le temps de résolution se retrouve à être réduit de 20% à 30% lorsque le filtrage tardif est utilisé.

Dans le but de mieux analyser les résultats présentés au tableau 4.2, la figure 4.4 montre le temps requis pour explorer jusqu'à un certain pourcentage de l'arbre de recherche

pour les deux approches. La relation entre les deux approches est linéaire parce que nous ne changeons pas la complexité des algorithmes ou le niveau de filtrage. Nous montrons ces résultats uniquement pour l'instance $(8, 4, 3)$, mais les résultats sont très similaires lorsqu'on analyse les autres instances.

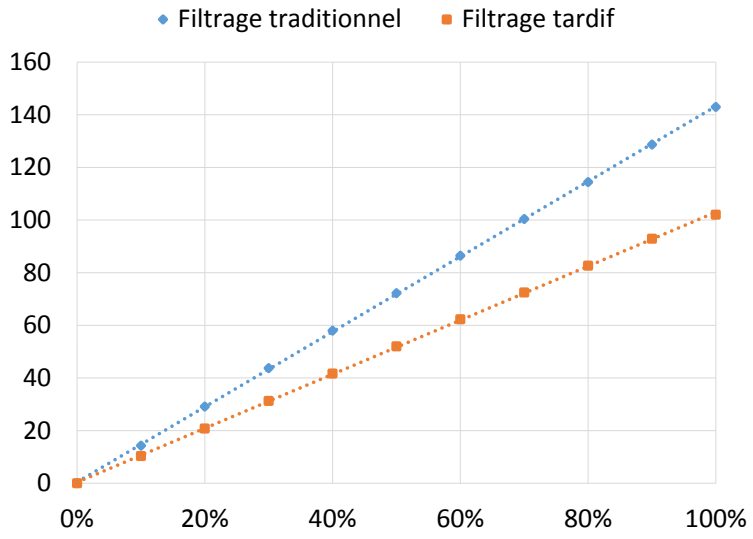


FIGURE 4.4 – Temps requis en secondes à un certain pourcentage d'exploration pour l'instance $(8, 4, 3)$

Nous avons également cherché à déterminer si le gain est distribué uniformément sur l'ensemble des contraintes. Le tableau 4.3 montre comment le temps de calcul est réduit pour chaque contrainte. Nous nous serions attendus à ce qu'il puisse être préférable de filtrer au plus tôt certaines contraintes et d'autres le plus tard possible, mais ce n'est pas le cas pour le problème étudié. La différence est positive pour chaque contrainte. Par contre, l'amélioration diffère grandement d'une contrainte à une autre.

4.4.2 Contre-exemple : conserver seulement la contrainte d'ordre lexicographique

À titre de contre-exemple, nous avons défini un problème qui consiste à remplir une matrice et de lui appliquer la contrainte d'ordre lexicographique strict (LEX) entre les lignes. Nous effectuons le filtrage soit en version traditionnelle (Frisch et coll.) ou tardive (voir 4.3.4).

	Temps filtrage traditionnel (s)	**Temps filtrage tardif (s)**	**Réduction en %**
Contrainte ordre croissant dans ligne	6,094	2,160	64,56%
Contrainte k	11,945	2,438	79,59%
Contrainte λ	23,606	6,247	73,54%
Ordre lex lignes	1,117	0,117	89,56%
Ordre lex colonnes	89,667	75,019	16,34%

TABLE 4.3 – Gain en temps pour les contraintes de l'instance (8, 4, 3)

Plus précisément, nous considérons des tableaux de variables de 2 lignes et c colonnes où, selon l'instance, $c \in \{5, 6, 7\}$. Chaque variable a pour domaine l'ensemble {1, 2, 3}. Il y a une contrainte LEX entre les deux lignes.

#Colonnes	**Type algo**	**Temps (s)**	**#Échecs**	**#sols**
5	Frisch et coll.	0,296	0	29 403
	Filtrage tardif	0,316	81	29 403
6	Frisch et coll.	2,330	0	265 356
	Filtrage tardif	2,438	243	265 356
7	Frisch et coll.	20,648	0	2 390 391
	Filtrage tardif	21,046	729	2 390 391

TABLE 4.4 – Frisch et coll. vs filtrage tardif pour ordre lexicographique strict entre 2 lignes

Nous voyons que l'ordre lexicographique filtré de manière traditionnelle (Frisch et coll.) et le filtrage tardif définissent le même espace de solutions, mais que le filtrage est plus efficient avec le filtrage traditionnel (moins d'échecs et donc moins de retours arrière).

4.5 Discussion

Nous souhaitions savoir quel était le meilleur moment pour exécuter les algorithmes de filtrage lorsqu'on tente de résoudre un BIBD. Nous montrons qu'en retardant le filtrage jusqu'au dernier moment (filtrer le domaine d'une variable uniquement lorsque la variable vient d'être sélectionnée et sur le point d'être instanciée) réduit le temps de

calcul entre 20% et 30% pour les instances présentées. L'approche a préservé le même espace solution et le même niveau de filtrage.

Il va de soi que ces résultats sont spécifiques au problème que nous avons étudié. Par contre, il est possible que d'autres problèmes puissent aussi bénéficier du filtrage tardif. Pour certains problèmes, l'approche optimale pourrait être un mélange de filtrage classique et tardif. Les solveurs génériques actuels ne donnent pas accès à l'événement *variableAÉtéSélectionnée*. Nous croyons qu'ajouter cet événement aux solveurs de PPC pourrait faciliter l'implémentation, le test et finalement l'utilisation d'algorithmes de filtrage tardif pour d'autres problèmes qui pourraient en bénéficier.

4.6 Conclusion

Dans ce chapitre, nous montrons qu'il a été possible de créer très facilement un nouvel événement pour le solveur par une modification très simple du code. Cette flexibilité d'utilisation permettra aux membres de FORAC d'imaginer, à leur tour, des nouveaux événements de propagation et rapidement passer au stade expérimental.

Conclusion

Dans ce mémoire, nous avons présenté, au chapitre 2, un nouveau solveur que nous souhaitions doté d'une grande polyvalence. Ce nouveau solveur devrait permettre de concevoir plus rapidement des nouvelles stratégies de recherche. Le chapitre 3 montre que le solveur permet de résoudre un problème industriel vécu par les partenaires du FORAC qui est le problème de la génération de patrons de chargement alternatifs pour un séchoir spécifique. Le chapitre 4 montre, quant à lui, qu'il est possible de facilement intégrer un nouvel événement dans le but de résoudre le problème du *Balanced Incomplete Block Design* plus rapidement qu'avec l'utilisation des événements classiques.

Malgré ces avantages pour la recherche de nouvelles techniques en PPC, le solveur ne performe pas aussi bien que les solveurs génériques pour bon nombre de problèmes. En effet, la majorité de ces solveurs sont développés et entretenus par des chercheurs expérimentés dans le domaine de la PPC et les choix d'implémentation sont faits en fonction de la vitesse (et non pas de la polyvalence que nous recherchions ici). C'est pourquoi, dans le but que le solveur du chapitre 2 puisse devenir meilleur, il faudrait continuer à le développer et, idéalement, y dédier une équipe de recherche. En résumé, le solveur présenté dans ce mémoire est très bon lorsque vient le temps de faire du prototypage de nouvelles techniques, mais il est possible de l'améliorer encore davantage dans le but que sa performance devienne, un jour, comparable aux solveurs génériques.

Une grande communauté scientifique est active en ce qui concerne la recherche de techniques novatrices de résolution en PPC. Considérant le rôle clé des solveurs dans la résolution des problèmes PPC, il ne fait aucun doute que des meilleures architectures et structures de données seront développées dans le but d'améliorer la rapidité et la facilité d'utilisation ce ceux-ci. Comme Eugene C. Freuder l'a bien spécifié dans son article *In Pursuit of the Holy Grail* [11], la PPC se rapproche tranquillement du jour où l'utilisateur aura simplement à spécifier au solveur le modèle du problème qu'il souhaite voir résoudre puisque le solveur saura automatiquement quelles heuristiques

et stratégies de recherche il doit utiliser. En attendant ce jour, l'expertise des gens ayant des connaissances en PPC restera nécessaire dans le but de bien exploiter les solveurs et pour développer de nouvelles techniques si requises.

Annexe A

Modèle MIP pour le problème du séchoir

Dans cette annexe, nous présentons le modèle MIP qui sert à produire des patrons de chargement pour remplir un séchoir. Ce modèle, originalement conçu par Philippe Marier [37, 38], professionnel de recherche de FORAC, a été implémenté et modifié par Marc-André Ménard, étudiant à la maîtrise de FORAC. Nous l'utilisons dans le but de comparer la performance du modèle MIP par rapport à l'approche PPC du chapitre 3.

Fonction-objectif :

$$maximiser \sum_{d \in D} \sum_{e \in E} \Phi_{d,e} \; min\{T - t_d, T - d_e\}$$

Ensembles :

$\boldsymbol{L}$	Ensemble des longueurs possibles pour les paquets $\cup \{0\}$
$\boldsymbol{P}$	Ensemble des produits verts
$\boldsymbol{E}$	Ensemble des processus de séchage compatibles avec le séchoir
$\boldsymbol{D}$	Ensemble des demandes
$\boldsymbol{F}$	Ensemble des produits finis
$\boldsymbol{D_f}$	Ensemble des demandes pour le produit fini $f \in \boldsymbol{F}$, $\boldsymbol{D_f} \subseteq \boldsymbol{D}$
$\boldsymbol{B}$	Ensemble de tous les couples $(r1, r2)$ tels que $r1, r2 \in \{1..n\}$ et $r1 < r2$

Paramètres :

n	Nombre de rangs dans le séchoir
n_{max}	Longueur maximale des rangs
n_{min}	Longueur minimale des rangs
h_{max}	Hauteur maximale d'une pile
h_{min}	Hauteur minimale d'une pile
h_p	Hauteur d'un paquet du produit p
a	Nombre d'emplacements par rang
a_{min}	Nombre d'emplacements minimum qu'il faut utiliser par rang
i_p	Inventaire du produit p disponible pour séchage (à la date de début prévue du séchage)
u_p	Volume d'un paquet du produit p
l_p	Longueur d'un paquet du produit p
T	Horizon de planification
t_d	Période à laquelle la demande d devrait être satisfaite
d_e	Durée du séchage, en nombre de périodes, lorsque le processus e est appliqué
$o_{p,f}$	Volume du produit fini f obtenu du produit p après séchage et rabotage
w_d	Volume de la demande d
Ψ	Grand nombre (volume total des produits en inventaire)

Variables de décision :

Q_e	Indique si le processus e est utilisé pour le séchoir {0,1}
$Z_{r,s,l}$	Choix de la longueur l pour l'emplacement s du rang r {0,1}
$P_{p,r,s}$	Nombre de paquets de produit p placés sur le rang r et emplacement s {entier positif}
$X_{p,r,s}$	Volume du produit p affecté au rang r et emplacement s {réel positif}
$R_{r,s}$	Indique si l'emplacement s du rang r est utilisé {0,1}
$\Phi_{d,e}$	Volume comblé de la demande d à partir de la production de produits finis faite en suivant le processus de séchage e
J_f	Volume de produit f pouvant être obtenu suite au séchage des produits dans le plan
$B_{r1,r2,s}$	Sert à assurer le respect de l'ordre lexicographique entre les rangs où $(r1, r2) \in \boldsymbol{B}$ et $s \in \{0..a\}$ {0,1}

Contraintes :

— Un seul processus doit être utilisé pour le séchoir.

$$\sum_{e \in \boldsymbol{E}} Q_e = 1 \tag{A.1}$$

— On remplit le séchoir dans les limites de longueur acceptable. Ainsi, pour chaque rang, la somme des longueurs des paquets sur une rangée doit être entre n_{min} et n_{max}.

$$n_{min} \leq \sum_{s=1}^{a} \sum_{l \in \boldsymbol{L}} l(Z_{r,s,l}) \leq n_{max} \qquad \forall r \in \{1..n\} \tag{A.2}$$

— On remplit le séchoir dans les limites de hauteur acceptable. Ainsi, pour chaque pile, la somme des hauteurs des paquets sur une pile doit être inférieure ou égale à h_{max}.

$$\sum_{p \in \boldsymbol{P}} h_p(P_{p,r,s}) \leq h_{max} \qquad \forall r \in \{1..n\}, \forall s \in \{1..a\} \tag{A.3}$$

— On remplit le séchoir dans les limites de hauteur acceptable. Ainsi, pour les piles utilisés, la somme des hauteurs des paquets sur une pile doit être supérieure ou égale à h_{min}.

$$\sum_{p \in \boldsymbol{P}} (h_p(P_{p,r,s})) + \Psi(1 - R_{r,s}) \geq h_{min} \qquad \forall r \in \{1..n\}, \forall s \in \{1..a\} \tag{A.4}$$

— On ne peut pas prendre plus de paquets que ce qui est disponible en inventaire.

$$\sum_{r=1}^{n} \sum_{s=1}^{a} X_{p,r,s} \leq i_p \qquad \forall p \in \boldsymbol{P} \tag{A.5}$$

— On ne peut placer qu'un nombre entier de paquets de chaque produit dans le plan de chargement.

$$P_{p,r,s} = \frac{X_{p,r,s}}{u_p} \qquad \forall p \in \boldsymbol{P}, \forall s \in \{1..a\}, \forall r \in \{1..n\} \tag{A.6}$$

— On ne doit choisir qu'une longueur par emplacement.

$$\sum_{l \in \boldsymbol{L}} Z_{r,s,l} = 1 \qquad \forall r \in \{1..n\}, \forall s \in \{1..a\} \tag{A.7}$$

— On ne doit pas placer des produits qui ne sont pas égaux à la longueur choisie pour un emplacement.

$$P_{p,r,s} \leq \Psi(Z_{r,s,l_p}) \qquad \forall p \in \boldsymbol{P}, \forall s \in \{1..a\}, \forall r \in \{1..n\} \tag{A.8}$$

— Pour chaque rang, on doit utiliser au moins le nombre d'emplacements minimum.

$$\sum_{s=1}^{a} R_{r,s} \geq a_{min} \qquad \forall r \in \{1..n\} \tag{A.9}$$

— Pour un emplacement non utilisé, la longueur est 0.

$$\sum_{l \in \boldsymbol{L}} l(Z_{r,s,l}) \leq max(\boldsymbol{L}) R_{r,s} \qquad \forall r \in \{1..n\}, \forall s \in \{1..a\} \tag{A.10}$$

$$R_{r,s} \leq \sum_{l \in \boldsymbol{L}} l(Z_{r,s,l}) \qquad \forall r \in \{1..n\}, \forall s \in \{1..a\} \tag{A.11}$$

— Contraintes pour la fonction-objectif.

$$J_v = \sum_{r=1}^{n} \sum_{s=1}^{a} \sum_{p \in \boldsymbol{P}} o_{p,v}(X_{p,r,s}) \qquad \forall v \in \boldsymbol{V} \tag{A.12}$$

$$\sum_{e \in \boldsymbol{E}} \Phi_{d,e} \leq w_d \qquad \forall d \in \boldsymbol{D} \tag{A.13}$$

$$\sum_{e \in \boldsymbol{E}} \sum_{d \in \boldsymbol{D_f}} \Phi_{d,e} \leq J_f \qquad \forall f \in \boldsymbol{F} \tag{A.14}$$

$$\Phi_{d,e} \leq \Psi(Q_e) \qquad \forall e \in \boldsymbol{E}, \forall D \in \boldsymbol{D} \tag{A.15}$$

— Les emplacements doivent être assignés en ordre croissant (bris de symétrie).

$$\sum_{l \in \boldsymbol{L}} l(Z_{r,s1,l}) \leq \sum_{l \in \boldsymbol{L}} l(Z_{r,s2,l}) \quad \forall (s1, s2) \in \{1..a\} \textit{ tels que } s1 < s2, \forall r \in \{1..n\} \tag{A.16}$$

— Les rangs doivent être en ordre lexicographique (bris de symétrie).

$$B_{r1,r2,0} = 0 \qquad \forall (r1, r2) \in \boldsymbol{B} \tag{A.17}$$

$$\sum_{l \in \boldsymbol{L}} l(Z_{r1,s,l}) - \sum_{l \in \boldsymbol{L}} l(Z_{r2,s,l}) \leq max(\boldsymbol{L})(B_{r1,r2,s}) \quad \forall (r1, r2) \in \boldsymbol{B}, \forall s \in \{1..a\} \tag{A.18}$$

$$\sum_{l \in \boldsymbol{L}} l(Z_{r2,s,l}) - \sum_{l \in \boldsymbol{L}} l(Z_{r1,s,l}) \leq max(\boldsymbol{L})(B_{r1,r2,s}) \quad \forall (r1, r2) \in \boldsymbol{B}, \forall s \in \{1..a\} \tag{A.19}$$

$$\sum_{l \in \boldsymbol{L}} (l(Z_{r2,s,l})) + 1 \leq \sum_{l \in \boldsymbol{L}} (l(Z_{r1,s,l})) + max(\boldsymbol{L})(B_{r1,r2,s-1} + (1 - B_{r1,r2,s})) \forall (r1, r2) \in \boldsymbol{B}, \forall s \in \{1..a\} \tag{A.20}$$

Bibliographie

[1] Ilyess Bachiri : Résolution des problèmes d'optimisation combinatoire avec une stratégie de retour-arrière basée sur l'apprentissage par renforcement. Mémoire de maîtrise, Université Laval, 2015.

[2] Ilyess Bachiri, Jonathan Gaudreault, Claude-Guy Quimper et Brahim Chaib-draa : RLBS : An adaptive backtracking strategy based on reinforcement learning for combinatorial optimization. *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference*, pages 936–942. IEEE, 2015.

[3] Philippe Baptiste, Philippe Laborie, Claude Le Pape et Wim Nuijten : Constraint-based scheduling and planning. *Handbook of Constraint Programming*, pages 759–797. Elsevier, 2006.

[4] Roman Barták : Constraint programming : In pursuit of the holy grail. *Proceedings of WDS99 (invited lecture), Prague, June*, pages 205–224, 2006.

[5] Christian Bessiere : Constraint propagation. *Handbook of Constraint Programming*, pages 27–81. Elsevier, 2006.

[6] François Chéné, Jonathan Gaudreault et Claude-Guy Quimper : A mixed-initiative system for interactive tactical supply chain optimization. *MOSIM 2014, 10ème Conférence Francophone de Modélisation, Optimisation et Simulation*, 2014.

[7] Rina Dechter et Judea Pearl : Network-based heuristics for constraint-satisfaction problems. *Artificial intelligence*, 34(1):1–38, 1987.

[8] Ludwig Dumetz, Jonathan Gaudreault, André Thomas, Nadia Lehoux, Philippe Marier et Hind El-Haouzi : Evaluating order acceptance policies for divergent production systems with co-production. *International Journal of Production Research*, pages 1–13, 2016.

[9] Hamed Fahimi et Claude-Guy Quimper : *Variants of Multi-resource Scheduling Problems with Equal Processing Times*, pages 82–97. Springer, 2015.

[10] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson et Toby Walsh : Breaking row and column symmetries in matrix models. *Principles and Practice of Constraint Programming-CP 2002*, pages 462–477. Springer, 2002.

[11] Eugene C. Freuder : In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997.

[12] Eugene C. Freuder et Alan K. Mackworth : Constraint satisfaction : An emerging paradigm. *Handbook of Constraint Programming*, pages 13–28. Elsevier, 2006.

[13] Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel et Toby Walsh : Global constraints for lexicographic orderings. *Principles and Practice of Constraint Programming-CP 2002*, pages 93–108. Springer, 2002.

[14] Alan Frisch, Christopher Jefferson et Ian Miguel : Symmetry breaking as a prelude to implied constraints : A constraint modelling pattern. *ECAI*, volume 16, page 171, 2004.

[15] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides : *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] Jonathan Gaudreault : *Algorithmes pour la prise de décision distribuée en contexte hiérarchique*. Thèse de doctorat, École Polytechnique de Montréal, 2009.

[17] Jonathan Gaudreault, Pascal Forget, Jean-Marc Frayret, Alain Rousseau, Sebastien Lemieux et Sophie D'Amours : Distributed operations planning in the softwood lumber supply chain : models and coordination. *International Journal of Industrial Engineering : Theory Applications and Practice*, 17(3):168–189, 2010.

[18] Jonathan Gaudreault, Jean-Marc Frayret, Alain Rousseau et Sophie D'Amours : Combined planning and scheduling in a divergent production system with co-production : A case study in the lumber industry. *Computers and Operations Research*, 38(9):1238–1250, 2011.

[19] Jonathan Gaudreault, Gilles Pesant, Jean-Marc Frayret et Sophie D'Amours : Supply chain coordination using an adaptive distributed search strategy. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1424–1438, 2012.

[20] Solomon W. Golomb et Leonard D. Baumert : Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[21] Google : Or-tools - google optimization tools, 2010. Disponible sur `https://developers.google.com/optimization`.

[22] Simon Hamel, Jonathan Gaudreault, Claude-Guy Quimper, Mathieu Bouchard et Philippe Marier : Human-machine interaction for real-time linear optimization. *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 673–680. IEEE, 2012.

[23] Robert M. Haralick et Gordon L. Elliott : Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[24] William D. Harvey : *Nonsystematic backtracking search*. Thèse de doctorat, University of Oregon, 1995.

[25] William D. Harvey et Matthew L. Ginsberg : Limited discrepancy search. *IJCAI (1)*, pages 607–615, 1995.

[26] Marti A. Hearst, James F. Allen, Curry I. Guinn et Eric Horvitz : Mixed-initiative interaction : Trends and controversies. *IEEE Intelligent Systems*, 14(5): 14–23, 1999.

[27] Willem-Jan van Hoeve et Irit Katriel : Global constraints. *Handbook of Constraint Programming*, pages 205–244. Elsevier, 2006.

[28] John N. Hooker : *Hybrid modeling*, pages 11–62. Springer, 2011.

[29] Alexander Kalinin, Ugur Cetintemel et Stan Zdonik : Searchlight : Enabling integrated search and exploration over large multidimensional data. *Proceedings of the VLDB Endowment*, 8(10):1094–1105, 2015.

[30] Alexander Kalinin, Ugur Cetintemel et Stan Zdonik : Interactive search and exploration of waveform data with searchlight. *Proceedings of the 2016 International Conference on Management of Data*, pages 2105–2108. ACM, 2016.

[31] Philip Kilby et Paul Shaw : Vehicle routing. *Handbook of Constraint Programming*, pages 799–834. Elsevier, 2006.

[32] Richard E. Korf : Improved limited discrepancy search. *AAAI/IAAI, Vol. 1*, pages 286–291, 1996.

[33] M. V. Kornilov : Astronomical observation tasks short-term scheduling using PDDS algorithm. *Astronomy and Computing*, 2016.

[34] Takakazu Kurokawa et Yoshiyasu Takefuji : Neural network parallel computing for BIBD problems. *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing*, 39(4):243–247, 1992.

[35] Anany Levitin : Greedy technique. *Introduction to the Design and Analysis of Algorithms*, pages 315–344. Pearson Education, 2012.

[36] Paolo Liberatore : On the complexity of choosing the branching literal in DPLL. *Artificial intelligence*, 116(1):315–326, 2000.

[37] Philippe Marier, Jonathan Gaudreault et Thomas Noguer : Planification opérationnelle multi-périodes du séchage du bois d'oeuvre avec composition dynamique des patrons de chargement. *CIGI2015, 11e Congrès International de Génie Industriel*, 2015.

[38] Philippe Marier, Jonathan Gaudreault et Thomas Noguer : Kiln drying operation scheduling with dynamic composition of loading patterns. *Information System, Logistics and Supply Chain*, pages 1–4, 2016.

[39] Thierry Moisan : *Minimisation des perturbations et parallélisation pour la planification et l'ordonnancement*. Thèse de doctorat, Université Laval, 2016.

[40] Thierry Moisan, Jonathan Gaudreault et Claude-Guy Quimper : Parallel discrepancy-based search. *Principles and Practice of Constraint Programming*, pages 30–46. Springer, 2013.

[41] Thierry Moisan, Claude-Guy Quimper et Jonathan Gaudreault : Parallel depth-bounded discrepancy search. *Integration of AI and OR Techniques in Constraint Programming*, pages 377–393. Springer International Publishing, 2014.

[42] Michael Morin, Anika-Pascale Papillon, Irène Abi-Zeid, François Laviolette et Claude-Guy Quimper : Constraint programming for path planning with uncertainty. *Principles and Practice of Constraint Programming*, pages 988–1003. Springer, 2012.

[43] Ka Boon Kevin Ng, Chiu Wo Choi et Martin Henz : A software engineering approach to constraint programming systems. *Software Engineering Conference. Ninth Asia-Pacific*, pages 167–175. IEEE, 2002.

[44] Pierre Ouellet et Claude-Guy Quimper : The multi-inter-distance constraint. *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 629, 2013.

[45] Ian P. Gent, Karen E. Petrie et Jean-François Puget : Symmetry in constraint programming. *Handbook of Constraint Programming*, pages 327–374. Elsevier, 2006.

[46] François Pachet et Pierre Roy : Automatic generation of music programs. *International Conference on Principles and Practice of Constraint Programming*, pages 331–345. Springer, 1999.

[47] Gilles Pesant : Balancing nursing workload by constraint programming. *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 294–302. Springer, 2016.

[48] Gilles Pesant, Claude-Guy Quimper et Alessandro Zanarini : Counting-based search : Branching heuristics for constraint satisfaction problems. *J. Artif. Intell. Res.(JAIR)*, 43:173–210, 2012.

[49] Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper et Jason Sweeney : Learning parameters for the sequence constraint from solutions. *International Conference on Principles and Practice of Constraint Programming*, pages 405–420. Springer, 2016.

[50] Steven Prestwich : CSPLib problem 028 : Balanced incomplete block designs. Disponible sur `http://www.csplib.org/Problems/prob028`.

[51] Steven Prestwich : Balanced incomplete block design as satisfiability. *Proceedings of the 12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.

[52] Steven Prestwich : A local search algorithm for balanced incomplete block designs. *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 132–143. Springer, 2002.

[53] Patrick Prosser et Evgeny Selensky : A study of encodings of constraint satisfaction problems with 0/1 variables. *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 121–131. Springer, 2002.

[54] Charles Prud'homme, Jean-Guillaume Fages et Xavier Lorca : *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. Disponible sur `http://www.choco-solver.org`.

[55] Jean-Charles Régin : Generalized arc consistency for global cardinality constraint. *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1*, pages 209–215. AAAI Press, 1996.

[56] Francesca Rossi, Peter Van Beek et Toby Walsh : *Handbook of constraint programming*. Elsevier, 2006.

[57] Christian Schulte et Peter J. Stuckey : When do bounds and domain propagation lead to the same search space ? *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):388–425, 2005.

[58] Christian Schulte et Peter J. Stuckey : Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1):2, 2008.

[59] Christian Schulte et Guido Tack : Implementing efficient propagation control. *TRICS*, 2010.

[60] Barbara M. Smith, Karen E. Petrie et Ian P. Gent : Models and symmetry breaking for peaceable armies of queens. *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 271–286. Springer, 2004.

[61] Douglas R. Stinson : *Combinatorial Designs : Constructions and Analysis*. Springer, New York, 2004.

[62] Chuffed Team : Chuffed - a lazy clause solver, 2017. Disponible sur `https://github.com/chuffed/chuffed`.

[63] Gecode Team : Gecode - generic constraint development environment, 2006. Disponible sur `http://www.gecode.org`.

[64] OscaR Team : OscaR - Scala in OR, 2012. Disponible sur `https://bitbucket.org/oscarlib/oscar`.

[65] Peter Van Beek : Backtracking search algorithms. *Handbook of Constraint Programming*, pages 83–132. Elsevier, 2006.

[66] Toby Walsh : Symmetry breaking constraints : Recent results. *Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*, 2012.

[67] Givon Chuen Kee Yan : Experimental modeling and intelligent control of a wood-drying kiln. Mémoire de maîtrise, University of British Columbia, 2000.

[68] Daisuke Yokoya et Takeo Yamada : A mathematical programming approach to the construction of BIBDs. *International Journal of Computer Mathematics*, 88(5):1067–1082, 2011.