UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

OFFERTE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

EN VERTU D'UN PROTOCOLE D'ENTENTE

AVEC L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PAR

SIMON BOIVIN

RÉSOLUTION D'UN PROBLÈME DE SATISFACTION DE CONTRAINTES POUR L'ORDONNANCEMENT D'UNE CHAÎNE D'ASSEMBLAGE AUTOMOBILE

MARS 2005

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

RÉSUMÉ

Plusieurs avenues existent dans la littérature pour la résolution des problèmes d'ordonnancement de la production. La complexité de ces problèmes rend nécessaire l'emploi de stratégies de recherche de solutions évoluées. Parmi celles-ci figurent leur modélisation sous la forme de problème de satisfaction de contraintes (CSP). Ce travail de recherche a pour but d'intégrer les formalismes des méthodes de résolution des CSP pour la résolution d'un problème d'ordonnancement de la production soit le problème de "car-sequencing". Les travaux effectués s'inscrivent dans une optique d'exploration des algorithmes de résolution de CSP et de leur application aux problèmes d'ordonnancement de la production.

Dans un premier temps, les algorithmes de résolution de CSP existants sont étudiés et une comparaison entre ceux-ci est effectuée afin de déceler les avantages et les inconvénients de chacune des différentes méthodes de résolution suggérées dans la littérature. Entre autres, les algorithmes basés sur un principe de retour en arrière lors des situations d'inconsistance tels le *BackTracking* et le *BackJumping* sont étudiés. De plus, l'ajout d'heuristiques guidant la recherche de solutions ainsi que les méthodes d'apprentissage lors de la recherche sont exposées dans ce mémoire. Les algorithmes de diminution de domaines développés dans la littérature tels le *Forward-Checking* et les algorithmes de propagation de contraintes (*AC*) sont décrits et leurs implications sur la recherche sont quantifiées. Finalement, quelques méthodes de parallélisation de la recherche sont définies dans le cadre du présent travail de recherche.

Suite à l'énumération des méthodes de résolution développées et à leur comparaison, ce travail de recherche couvre le développement d'un algorithme de résolution de CSP appliqué au problème de "car-sequencing". Premièrement, une étude de ce problème ainsi que des formulations possibles pour celui-ci est effectuée. Par la suite, le graphe de contraintes associé au problème de "car-sequencing" est défini et une stratégie de résolution du problème est décrite. Plus particulièrement, une méthode de diminution de l'espace de recherche de solutions possibles induit par les contraintes de ce problème est définie. L'algorithme de *Forward-Checking* est alors utilisé pour la résolution de ce problème particulier. De plus, une étude sur l'utilisation d'heuristiques guidant la recherche de solutions est effectuée. Finalement, une méthode de subdivision du problème initial en sous-problèmes indépendants a été développée afin de favoriser la diversification de la recherche de solutions lors d'une résolution concurrente.

Les instances de problèmes suggérées dans la librairie de problèmes *CSPLib* sont utilisées comme base d'évaluation des méthodes développées. Les résultats obtenus dans le cadre de ce projet de recherche ont été comparés aux meilleurs résultats obtenus dans la

littérature sur ces instances de problème. De plus, les résultats ont été comparés avec un outil de résolution commercial soit ILOG Solver 6.0.

Les méthodes développées ont obtenu des résultats intéressants. En effet, pour un premier groupe de 70 instances du problème de "car-sequencing", le *Forward-Checking* développé a permis de surpasser les résultats obtenus par le solveur commercial ILOG Solver 6.0. Par contre, sur un deuxième groupe de problèmes, certaines bonifications des méthodes développées s'avèrent nécessaires. La diminution de l'espace de recherche représente une voie à explorer dans des recherches futures pour ainsi favoriser la résolution d'instances de problèmes non-satisfiables.

REMERCIEMENTS

Au terme de ce travail, je tiens à remercier M. Marc Gravel, mon directeur de maîtrise, pour sa patience, sa disponibilité et son support tout au long de ce projet. De plus, je tiens à remercier M. Michaël Krajecki pour sa supervision lors de mon stage de recherche à l'Université de Reims Champagne-Ardenne. Ses précieux conseils m'ont permis d'orienter mes recherches vers des avenues que je n'aurais su explorer seul.

Par le fait même, je veux remercier les doctorants et le personnel enseignant de cette université. Plus particulièrement, M. Pierre-Paul Mérel, M. Olivier Flauzac et M. Christophe Jaillet pour leur accueil chaleureux. Je tiens également à remercier M. Richard Tremblay que j'ai eu le privilège de côtoyer dans le cadre de ce stage. Merci de m'avoir fait profiter de vos expériences en recherche scientifique.

Je désire également remercier Mme Caroline Gagné pour ses conseils et son support. Le fait d'avoir mis à ma disposition du matériel informatique me fut d'une grande aide lors de mes recherches.

Je veux souligner aussi l'apport du personnel enseignant de l'UQAC ainsi que des membres du Groupe de Recherche en Informatique. Les échanges que j'ai pu avoir avec vous m'ont été d'une grande aide lors des travaux de recherche qui ont mené à l'écriture de ce mémoire.

Je veux également souligner le travail d'évaluation de ce mémoire par M. Roger Villemaire de l'Université du Québec à Montréal.

Sur une note plus personnelle, je désire remercier M. Dany Gravel avec qui j'ai complété la majeure partie de ma formation en informatique. Je veux aussi remercier M. Raphaël Girard que j'ai le plaisir de côtoyer depuis maintenant vingt ans. Je tiens, par le fait même, à souligner le travail de correction de la grammaire et de l'orthographe du présent document par ce dernier.

Finalement, sur une note personnelle, quoique également professionnelle, je tiens à remercier Sara, ma compagne de vie, d'études, de voyage, de bureau ... Merci pour ta compréhension et ton support tout au long de ce projet.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

L'ordonnancement de la production est une problématique complexe. Elle consiste à déterminer une séquence d'exécution de travaux répondant à certains objectifs de l'entreprise pouvant même être parfois conflictuels. Par exemple, l'entreprise peut viser à satisfaire au mieux les demandes des clients en respectant le plus possible les dates de livraison prévues. Cet objectif fera en sorte qu'une séquence d'exécution des travaux sera établie en cherchant à privilégier les travaux exigibles à court terme. Cependant, cet ordonnancement peut engendrer des coûts importants en terme de ressources humaines et matérielles. En effet, les équipements utilisés pour la fabrication des produits doivent souvent être configurés lorsqu'ils passent d'une production à une autre. L'ordonnancement cherchant à optimiser la satisfaction de la clientèle impliquera un nombre élevé de réglages de l'équipement ce qui augmentera, par le fait même, les coûts de production et influencera la rentabilité de l'entreprise. C'est pourquoi, les entreprises qui opèrent dans un environnement à ressources limitées chercheront souvent à optimiser l'utilisation de l'ensemble de leurs ressources. Cet objectif favorisera un ordonnancement du processus généralement différent de celui généré par l'objectif de satisfaction de la clientèle. On aura donc des ordonnancements des travaux qui diffèrent selon l'objectif recherché.

Les exemples d'objectifs à atteindre par une organisation sont nombreux et définissent différentes problématiques d'ordonnancement. Selon Stevenson et Benedetti [2001], le rôle de l'ordonnancement consiste à faire des compromis, à trouver le juste équilibre entre, d'une part, la satisfaction du client, la réduction des coûts et du temps d'attente, du temps de réponse et de livraison et, d'autre part, l'utilisation optimale des ressources de l'entreprise. Un bon ordonnancement permettra donc, pour

plusieurs entreprises, de se démarquer de la compétition. L'ordonnancement fait ainsi partie de la stratégie organisationnelle.

Les problèmes d'ordonnancement de la production peuvent être vus sous la forme de problèmes d'optimisation combinatoire. Un problème d'optimisation combinatoire est un problème tel que l'on a un ensemble discret de solutions possibles ainsi qu'une fonction objectif permettant d'évaluer chacune de celles-ci [Hao, Galinier *et al.* 1999]. On cherchera à trouver la solution optimisant l'objectif. Il peut aussi arriver que le problème comporte plus d'un objectif et on parlera alors d'optimisation combinatoire multi-objectifs.

Un des problèmes d'ordonnancement étudiés dans la littérature est le problème de "car-sequencing". Il correspond à une chaîne d'assemblage automobile où l'on doit déterminer l'ordre dans lequel un ensemble de voitures seront fabriquées. Celles-ci possèdent certaines options pour lesquelles une charge de travail supplémentaire est occasionnée à certains postes de la chaîne d'assemblage. Par exemple, certaines voitures nécessitent l'installation de toit ouvrant, d'air climatisé, de lecteur de disques compacts ou autres. Ces véhicules doivent, par conséquent, être dispersés dans la séquence de production de façon à lisser la charge de travail. Des contraintes de capacité précisent le nombre maximum de voitures q possédant l'option i qui peuvent être produites sur une sous-séquence de taille p. Ces contraintes de capacité sont généralement exprimées par le ratio $C_i = q_i / p_i$ pour chacune des options i du problème. La solution d'un problème de "car-sequencing" sera une séquence de production de véhicules ne violant aucune des contraintes de capacité du problème.

Plusieurs méthodes ont été développées dans la littérature pour résoudre ce problème. Parmi celles-ci figurent des heuristiques issues du domaine de l'intelligence artificielle et des méthodes exactes.

Parmi les heuristiques, on note l'utilisation des réseaux de neurones avec l'algorithme GENET [Davenport, Tsang *et al.* 1994], les méthodes de recherche dans le voisinage [Davenport et Tsang 1995], les méthodes évolutives tel les algorithmes génétiques [Warwick et Tsang 1995] et l'optimisation par colonies de fourmis [Solnon 2000] [Gravel, Gagné *et al.* 2004]. Ces algorithmes traitent le problème comme un problème d'optimisation en cherchant à déterminer une séquence violant le moins possible les contraintes de capacité. Par conséquent, ces heuristiques ne permettent pas, dans certains cas, de déterminer si un problème est satisfiable. Au mieux, elles suggèrent une solution comportant un nombre minimal de contraintes non respectées.

Pour leur part, les méthodes exactes assurent la résolution optimale du problème par le parcours complet de l'espace de recherche pour établir la satisfiabilité ou l'insatisfiabilité du problème. Toutefois, tel que défini par Gent [1998], le problème de "car-sequencing" est un problème NP-Complet ce qui signifie que l'énumération et l'évaluation de toutes les séquences possibles représentent des tâches ardues.

Les méthodes exactes telles la programmation linéaire en nombres entiers [Gravel, Gagné *et al.* 2004] et les méthodes de résolution de problèmes de satisfaction de contraintes (CSP) [Smith 1996] [Régin et Puget 1997] [Bessière, Meseguer *et al.* 1999] représentent des alternatives de résolution pour certaines instances du problème de "car-sequencing". En effet, l'utilisation des formalismes de *Branch and Bound* [Wagner 1975] [Ignizio et Cavalier 1994] pour éliminer de nombreuses parties de l'espace de solution permet à la programmation linéaire en nombres entiers de

solutionner certaines instances de ce problème [Gravel, Gagné *et al.* 2004]. Toutefois, étant donné que l'espace de solution à explorer croît exponentiellement en fonction de la taille du problème, soit le nombre de voitures à produire, les limites d'une telle approche sont rapidement atteintes.

Les méthodes de résolution de CSP permettent aussi la réduction de l'espace de recherche à explorer par l'exploitation des contraintes du problème. C'est ainsi que certaines solutions ne seront pas évaluées puisque l'étude des contraintes du problème indique qu'elles sont impossibles. Les méthodes de modification du problème initial suggérées par Régin [1996] et Régin et Puget [1997] ainsi que les algorithmes spécifiques à ce problème développés par Bessière Meseguer *et al.* [1999] permettent de favoriser également la réduction de l'espace de recherche et d'ainsi résoudre certaines instances de problème. C'est dans cette optique que le présent travail sera réalisé en visant l'étude des algorithmes de résolution de CSP et leur adaptation au problème de "car-sequencing".

Même si les méthodes de résolution de CSP permettent de déterminer s'il existe ou non une solution au problème par le parcours en totalité ou en partie de l'arbre de recherche, la complexité du problème fait en sorte que plusieurs instances de problème ne peuvent être résolues dans un temps jugé raisonnable [Smith 1996]. L'utilisation optimale des ressources informatiques est un élément essentiel lors de la résolution. C'est pourquoi, le présent travail étudiera plus particulièrement les méthodes d'accélération du processus de recherche de solution.

Premièrement, un état de l'art des méthodes de résolution de problèmes de CSP est présenté au Chapitre 2. Entre autres, les méthodes de base de la résolution par l'algorithme de BackTracking, les principes de filtrage de domaines et les méthodes

d'apprentissage en cours de recherche sont abordées. Deuxièmement, l'application des méthodes de résolution de CSP au problème de "car-sequencing" et, plus particulièrement, un algorithme de filtrage de domaines spécifique à ce problème est présenté au Chapitre 3. Des essais numériques et les résultats permettant de déterminer les performances des méthodes développées sont également présentées dans ce chapitre. De plus, les résultats obtenus sont comparés à ceux présentés dans la littérature ainsi qu'à ceux obtenus par ILOG Solver 6.0. Un schéma concurrent de résolution permettant d'alterner les phases de descente en profondeur dans l'arbre de résolution avec une recherche en largeur est également proposé pour repousser les limites de l'algorithme séquentiel. Enfin, la conclusion de ce mémoire permettra de porter un jugement général de la performance des méthodes de résolution de CSP et d'identifier des avenues de recherche intéressantes.

CHAPITRE 2

MÉTHODES DE RÉSOLUTION DE PROBLÈMES DE SATISFACTION DE CONTRAINTES

2.1 Introduction

Un problème de CSP est un triplet $P = \{X, D, C\}$ tel que X est un ensemble $\{X_1, X_2, ..., X_n\}$ de variables, D est un ensemble $\{D_1, D_2, ..., D_n\}$ de domaines tel que D_i est le domaine de valeurs pouvant être prises par X_i et C est un ensemble de contraintes restreignant les valeurs $v \in D_i$ pouvant être prise par X_i.

La portée d'une contrainte C_i est le nombre de variables $E \in X$ tel que les valeurs $v \in D_e$ pouvant être prises par E sont restreintes par C_i [Dechter et Frost 2002]. Une contrainte s'appliquant à seulement une variable sera dite *unaire* tandis que, pour une contrainte s'appliquant à deux variables, on parlera de contrainte *binaire*. On aura donc, pour un graphe de contraintes binaires, un ensemble de contraintes $C_{ij} \in C$ qui restreindra les valeurs pouvant être prises par les variables X_i et X_j simultanément [Cooper, Cohen *et al.* 1994]. Un CSP peut être vu sous la forme d'un graphe de contraintes tel que les sommets sont les variables et les arêtes sont les contraintes du problème.

De nombreux problèmes d'intelligence artificielle peuvent être formulés sous forme de CSP et l'engouement pour ce secteur de recherche est croissant. Oeuvrant dans des secteurs d'activités aussi vastes que les systèmes de vision artificielle, la gestion d'horaire, le raisonnement temporel, les problèmes de graphes, l'affichage de plans, les problèmes de satisfaction (SAT), le développement de circuits, le design manufacturier et nombre d'autres champs d'application, les CSP peuvent servir de cadre pour la résolution de problèmes théoriques ou pratiques. La complexité des CSP rendant inopportun le fait de tenter de les résoudre manuellement, des méthodes d'intelligence artificielle utilisant la force de calcul des ordinateurs ont été développées

pour les solutionner. Quoiqu'en constante évolution, la capacité des ordinateurs modernes n'est cependant pas encore suffisante pour résoudre plusieurs de ces problèmes. C'est pourquoi, des algorithmes orientant le processus de recherche de solutions ont été développés. De plus, certains processus ont été instaurés dans le but d'utiliser la force de calcul d'ordinateurs parallèles ou les principes du calcul distribué [Habbas, Krajecki *et al.* 2000] [Bessière, Maestre *et al.* 2002].

Parmi les problèmes de CSP les plus connus figurent le problème des reines et le problème de coloration de graphe. Le problème des reines consiste à placer *n* reines sur *n* différentes cases d'un échiquier de *n* lignes par *n* colonnes de manière à ce qu'aucune reine ne puisse en éliminer une autre [Tsang 1999]. Une des principales particularités de ce problème est que le nombre de reines détermine à la fois le nombre de variables du problème et la taille du domaine de celles-ci.

Le problème de coloration de graphe est un problème où l'on doit affecter une couleur à chacune des zones d'une carte de manière à ce qu'aucune zone adjacente de cette dernière ne soit de la même couleur [Zhou et Nishizeki 1999].

Enfin, le problème de "car-sequencing" est un autre problème retrouvé dans la littérature. L'étude de ce problème a débuté avec les travaux de Parrello et Kabat [1986]. Par la suite, Dincbas, Simonis *et al.* [1988] ont travaillé sa résolution par les méthodes de programmation de contraintes. D'autres recherches ont été réalisées par Régin [1996] et Régin et Puget [1997] ainsi que par Bessière, Meseguer *et al.* [1999] afin de trouver des algorithmes ou des modifications à des algorithmes existants permettant la résolution de ce problème. L'évaluation de l'efficacité de ces algorithmes se fait par l'exécution de ceux-ci sur différentes instances du problème. C'est pourquoi, la librairie CSPLib (http://4c.ucc.ie/~tw/csplib/) met à la disposition des chercheurs une

bibliothèque d'instances théoriques de problèmes de CSP dont, entre autres, des instances du problème de "car-sequencing". Certaines instances industrielles du problème ont aussi été rendues disponibles par le constructeur automobile *Renault* dans le cadre du *Challenge ROADEF'2005* [1].

Ces instances de problèmes peuvent être résolues en suivant différentes méthodes. Comme mentionné précédemment, ces méthodes permettent l'exploration de l'espace de solution de manière à déterminer la satisfiabilité ou l'insatisfiabilité d'un problème. Cette recherche de solutions peut être guidée pour faciliter la résolution. Dans le cas de problèmes insatisfiables, les formalismes d'étude des contraintes du problème permettront d'éviter certaines parties de l'espace de solutions et ainsi de parcourir celui-ci plus rapidement. Les sections suivantes décrivent les méthodes de résolution développées par les chercheurs du domaine des CSP.

2.2 Les méthodes de résolution de problèmes de CSP

La méthode naïve de résolution de problèmes de CSP consiste à générer et tester tous les ordres possibles d'instanciation (*generate-and-test*). Alors, chaque combinaison de variables et de valeurs est systématiquement générée et testée pour voir si elle satisfait toutes les contraintes du problème. La première combinaison satisfaisant toutes les contraintes est alors la solution du CSP [Tsang 1999]. Résoudre un problème de cette manière devient rapidement impossible lorsque la taille des problèmes augmente et ce, même avec l'utilisation d'outils informatiques. Des méthodes d'intelligence artificielle plus évoluées ont donc été développées dans le but de résoudre

---

[1] http://www.prism.uvsq.fr/~vdc/ROADEF/CHALLENGES/2005/

les CSP de manière plus efficace. Dans les sections suivantes, ces méthodes seront décrites en détails. Parmi celles-ci figure l'algorithme de *BackTracking* qui sert de base à plusieurs des autres méthodes et qui peut aussi être utilisé en simultané avec celles-ci.

2.2.1 Algorithme de BackTracking

Une solution a du graphe de contraintes est une instanciation de valeur pour toutes les variables du problème telle que toutes les contraintes du problème sont satisfaites. On note $\vec{a}_i$ une sous-solution ou une instanciation partielle consistante de i variables du problème. Une instanciation est une affectation de valeur $v \in D_i$ à la variable X_i. On dira d'une instanciation qu'elle est consistante si elle ne viole aucune des contraintes du problème. On définit une sous-séquence d'instanciation ou une instanciation partielle de taille i comme étant une instanciation consistante de i variables de l'ensemble X. Une solution est donc une séquence consistante de la taille du nombre de variables du problème.

L'algorithme de BackTracking (BT), a comme but d'étendre une sous-séquence de taille i à une sous-séquence de taille $i+1$ jusqu'à la découverte d'une solution. Comme l'instanciation d'une variable a comme possible effet de diminuer le domaine des variables lui étant adjacentes dans le graphe de contraintes, il peut arriver que le domaine courant de la variable que l'on tente d'instancier soit vide. On parlera alors de situation d'inconsistance (*dead-end*). Dans ce cas précis, un principe de retour en arrière sera appliqué. Le retour en arrière est l'action de retourner vers une variable précédant la variable courante dans la sous-séquence d'instanciation, de modifier la valeur affectée à celle-ci et de poursuivre la résolution par la suite. L'algorithme de BT est donc un algorithme à deux phases. Premièrement, une phase permettant d'avancer

vers une prochaine variable à instancier et permettant d'étendre la sous-séquence et, deuxièmement, une phase de retour en arrière consistant à choisir la variable vers laquelle sera fait le retour en arrière dans la sous-séquence [Dechter 1990]. L'algorithme est illustré à la Figure 2.1 où l'on peut voir l'étape de choix de la valeur à affecter à la prochaine variable dans la sous-procédure *SelectionnerValeur* [Dechter 2003]. Cette dernière permet de choisir une valeur dans le domaine de la variable courante et cette valeur doit être consistante par rapport au graphe de contraintes. De plus, l'algorithme effectue un retour en arrière vers la dernière variable affectée lors d'inconsistance. Il est à noter l'utilisation de la fonction *EstConsistant* qui permet de déterminer si, pour la séquence d'instanciation présente $\vec{a}_i$, il est possible d'affecter la valeur v à la variable X_i.

```
Procédure BACKTRACKING
Entrée : Un graphe de contraintes : P = (X, D, C)
Sortie : Une solution au problème ou une information tel que le graphe
est inconsistant.
i ← 1
D'i ← Di
TANT QUE 1 ≤ i ≤ n FAIRE
      Instancier Xi ← SélectionnerValeur()
      SI Xi est nul ALORS
            i ← i - 1
      SINON
            i ← i + 1
            D'i ← Di
SI i = 0 ALORS
      Retourner « inconsistant »
SINON
      Retourner les valeurs affectées à {X1, ... Xn}

Sous-procédure SélectionnerValeur
      TANT QUE D'i n'est pas vide FAIRE
            Sélectionner un élément v ∈ D'i et enlever v
            de D'i
      SI EstConsistant(a_{i-1}, Xi = v) ALORS
            Retourner v
```

Figure 2.1 - Algorithme de BackTracking

L'algorithme BT sert de base à de nombreux algorithmes plus élaborés de résolution de problème de CSP. Entre autres, certains chercheurs ont spécialisé la phase de retour en arrière pour en augmenter l'efficacité [Dechter et Frost 2002] [Prosser 1995]. D'autres ont élaboré des procédures de renforcement de la recherche par l'ajout de méthode de filtrage de domaines visant la suppression de valeurs rendues impossibles par l'état courant d'une résolution [Mackworth 1977] [Mohr et Henderson 1986] [Hentenryck, Deville *et al.* 1992] [Bessière 1994] [Bessière, Freuder *et al.* 1995; Bessière et Régin 2001]. Examinons maintenant certains de ces algorithmes ainsi que leurs implications sur la recherche de solutions.

2.2.2 Algorithme de BackJumping

Selon Dechter et Frost [2002], le BT a comme principal problème de retomber sans cesse dans les mêmes situations d'inconsistance. Cet algorithme a donc été raffiné par plusieurs chercheurs dans le but d'accélérer le processus de retour en arrière. Les algorithmes de BackJumping (BJ) ont été développés dans ce but.

L'algorithme BJ conserve, comme le BT standard, une étape de choix de la prochaine variable à instancier et une étape de retour en arrière lors de situations d'inconsistance. Cependant, plutôt que d'effectuer un retour en arrière vers la $i-1^{ième}$ variable affectée, on utilisera un principe de saut vers une variable plus loin dans la sous-séquence d'instanciation. Le BJ a comme but d'identifier la variable ayant causé l'inconsistance. Le tout se fera par l'analyse des raisons de la situation d'inconsistance et la recherche de la variable ayant eu le plus d'impact sur celle-ci. Il sera aussi possible de mettre à profit cette analyse en enregistrant les raisons de l'inconsistance

sous la forme de nouvelles contraintes. On parle alors du principe d'*apprentissage* [Frost et Dechter 1994] qui sera traité à la Section 2.2.4.

Plusieurs algorithmes utilisant le principe de BJ ont été développés. Les quatre principaux sont le *Gaschnig's backjumping* (G-BJ), [Gaschnig 1979] le *Graph-based backjumping* (GB-BJ) [Dechter et Frost 2002], le *Conflict-directed backjumping* (CD-BJ) [Prosser 1995]et le *i-backjumping* (i-BJ) [Dechter et Frost 2002]. Ces algorithmes utilisent certains formalismes pour déterminer vers quelle variable effectuer un saut lors du retour en arrière. A cet effet, les définitions 2.1 à 2.6 sont nécessaires pour assurer la compréhension de la description de ces méthodes de BJ.

Définition 2.1 : *Un **conflit** est une situation telle que l'affectation d'une valeur* $v \in D_i$ *à la variable* X_i *viole une des contraintes C du problème.*

Définition 2.2 : *Un **ensemble de conflits** est lorsque, pour une sous-séquence* $\vec{a} = (a_1,...,a_n)$ *consistante et une variable* X_i *non affectée, il n'existe pas de valeur dans* D_x *qui soit consistante avec* $\vec{a}$. *On dira donc que* $\vec{a}$ *est un ensemble de conflit de* X_i. *S'il n'existe aucun sous-ensemble de* $\vec{a}$ *étant en conflit avec* X_i, *alors* $\vec{a}$ *est un **ensemble de conflit minimal** de* X_i.

Définition 2.3 : *Pour une sous-séquence d'instanciation* $\vec{a}$, *si* $\vec{a}$ *est en conflit* avec la prochaine variable à instancier (X_{i+1}), *on nommera cette variable **feuille de l'inconsistance**.*

Définition 2.4 : *Un **nogood** est, pour un problème P = (X, D, C), toute instanciation partielle qui n'apparaît dans aucune solution de P. Un **nogood minimal** est tout nogood non composé lui-même d'un nogood.*

Définition 2.5 : *Un **retour en arrière sûr** (safe jump) est, pour une séquence* $\vec{a} = (a_1,..., a_i)$ *qui est en une feuille d'inconsistance, un saut vers une variable* X_j *tel que* $j \leq i$ *et que l'instanciation partielle* $\vec{a}_j = (a_1,...,a_j)$ *est un nogood.*

Définition 2.6 : *La **variable coupable** (culprit) de l'inconsistance est, pour une instanciation* $\vec{a} = (a_1,...,a_i)$, *la variable d'indice minimale présente dans* $\vec{a}$ *entrant en conflit avec* X_{i+1} c'est-à-dire vidant le contenu de la prochaine variable à instancier. *Selon la notation habituelle, la variable coupable est notée* X_b.

Le Gaschnig's backjumping (G-BJ) porte le nom du chercheur ayant défini plusieurs concepts à la base des méthodes modernes de résolution des CSP. Il enregistre de l'information durant la génération des sous-séquences d'instanciation et utilise cette information pour déterminer la variable coupable de la situation d'inconsistance. Lors de l'étape de retour en arrière, l'algorithme identifie la variable coupable de l'inconsistance en utilisant la fonction *SélectionnerValeur-GBJ* décrite à la Figure 2.2 [Dechter 2003].

Notons que l'algorithme gardera, tout au long de son exécution, une information relative à la dernière variable précédant la variable courante qui entre en conflit avec une des valeurs possibles de cette dernière. Cette information sera stockée dans la variable $dernier_i$ et gardée pour chaque variable du problème. Ce phénomène fera en sorte, conformément à Dechter [1990], que le G-BJ ne fera que des retours en arrière sûrs et vers des feuilles de la situation d'inconsistance.

```
Procédure SélectionnerValeur-GBJ
TANT QUE D'i n'est pas vide FAIRE
      Sélectionner un élément v ∈ D'i arbitrairement et supprimer v
      dans D'i
      Consistant ← Vrai
      k ← 1
      TANT QUE k < i ET Consistant FAIRE
            SI k > dernier_i ALORS
                  dernier_i ← k
            SI EstConsistant (a_k , X_i = v) ALORS
                  k ← k + 1
            SINON
                  Consistant ← false
      SI Consistant ALORS
            Retourner v
      Retourner Nul
```

Figure 2.2 - Procédure SélectionnerValeur – G-BJ

L'algorithme de retour en arrière par sauts *Graph-based (GB-BJ)* utilise uniquement l'information présente dans la graphe de contraintes pour décider vers quelle variable effectuer le retour en arrière. Lors de la détection de la situation

d'inconsistance tel qu'aucune valeur ne peut être affectée à une variable X_i, l'algorithme choisira de revenir vers la variable X_j la plus récemment affectée telle que X_j est adjacent à X_i dans le graphe de contraintes. Dans le cas où X_j n'a pas d'autres valeurs possibles, l'algorithme retournera vers une variable X_k adjacente à X_i et X_j.

Certains éléments spécifiques à la méthode sont utilisés lors de la résolution tel les principes d'ancêtres, de parents et de session. Les *ancêtres* d'une variable X_i, noté $anc(X_i)$, sont, pour un graphe de contrainte P et un ordre d'instanciation O, les variables précédant X_i dans la séquence O et étant adjacentes à X_i dans P. Le *parent* de X_i, noté $P(i)$ est la variable la plus récemment affectée dans $anc(X_i)$. La session d'une variable X_i, lors d'un algorithme de BT, débute lorsqu'une valeur présente dans D_i est affectée à X_i. Cette session se termine lorsqu'on effectue un retour en arrière vers une variable X_j tel que $j < i$. L'algorithme de GB-BJ est illustré à la Figure 2.3 [Dechter 2003]. Notons que la sélection de valeur est la même que celle décrite à la Figure 2.1 pour le BT.

L'algorithme garde en mémoire une copie du domaine des variables et ce, pour permettre de restaurer l'état initial des variables lors de retour en arrière. L'algorithme utilise, lors de cette même phase, les informations du graphe de contraintes, soit les ancêtres de la variable, pour identifier à quel endroit se déplacer. Notons que la procédure *SelectionnerValeur* renvoie *nul* lorsque le domaine de la variable courante est vide.

```
Procédure Graph-Based-BJ
Entrée : Un graphe de contraintes P = (X, D, C)
Sortie : Une solution au problème ou une information telle que le
graphe est inconsistant.

Calculer anc(X_i) pour tous X_i
i ← 1
D'_i ← D_i
I_i ← anc(X_i)
TANT QUE 1 ≤ i ≤ n FAIRE
      Instancier x_i ← SelectionnerValeur
      SI x_i est nul ALORS
            iprev ← i
            i ← P(i) dans I_i
            I_i ← I_i ∪ I_iprev - {X_i}
      SINON
            i ← i + 1
            D'_i ← D_i
            I_i ← anc(X_i)
SI i = 0 ALORS
      Retourner « inconsistant »
Sinon
      Retourner les valeurs affectées à {X_1, ... X_n}
```

Figure 2.3 - Algorithme Graph-Based-BJ

Le *Conflict-directed backjumping* (CD-BJ) quant à lui utilise de l'information relative à la détection de conflits. Plutôt que d'utiliser l'information du graphe de contraintes comme dans le GB-BJ, il utilise l'état courant de la résolution relativement aux contraintes du graphe pour effectuer ses retours en arrière.

L'ensemble de retour en arrière J_i de l'inconsistance sera donc gardé en mémoire. Lors de la résolution de problème, on cherchera à déterminer si l'instanciation de la variable courante avec la sous-séquence d'instanciation courante est possible par l'utilisation de la méthode *EstConsistant.* Si l'instanciation est impossible, on ajoutera la variable touchée par le conflit, soit la variable dont le domaine est vide, dans l'ensemble de conflits de la variable à instancier. On aura donc, tout au long de la résolution, un ensemble de retour en arrière pour chaque variable du problème. L'algorithme de CD-BJ est illustré à la Figure 2.4 [Dechter 2003].

On note, dans un premier temps, que l'algorithme stocke de l'information relative à l'inconsistance. De plus, l'algorithme modifie le comportement de la méthode de sélection de valeurs.

```
Procédure CONFLICT-DIRECTED-BACKJUMPING
Entrée : Un graphe de contraintes P = (X, D, C)
Sortie : Une solution ou une information telle que le graphe de
      contraintes est inconsistant.
      i ← 1
      D'i ← Di
      Ji ← 0
      TANT QUE 1 ≤ i ≤ n FAIRE
            Instancier Xi ← SelectionnerValeur-CBJ
            SI Xi = null ALORS
                  Iprev ← i
                  i ← index de la dernière variable dans Ji
                  Ji ← Ji ∪ Jiprev - {Xi}
            SINON
                  i ← i + 1
                  D'i ← D'
                  Ji ← 0
      SI i = 0 ALORS
            Return « inconsistant »
      SINON
            Retourner les valeurs affectées à {X1, … Xn}
Sous-Procédure SelectionnerValeur-CBJ
      TANT QUE D'i n'est pas vide FAIRE
            Sélectionner v ∈ D'i et supprimer v dans D'i
            Consistant ← VRAI
            k ← 1
            TANT QUE k < i ET Consistant FAIRE
                  SI EstConsistant (ak, Xi = v) ALORS
                        k ← k + 1
                  SINON
                        Soit Rs la première contrainte à causer le
                        conflit
                        Ajouter les variables présentes dans la portée
                        de S excepté Xi dans Ji
                        Consistant ← FAUX
            SI Consistant ALORS
                  Retourner v
      Retourner NUL
```

Figure 2.4 - Algorithme Conflict-Directed-BJ

On constate également que la sélection de valeurs prend en compte les portées des contraintes ainsi que le niveau de la contrainte puisqu'elle fonctionne selon la contrainte la plus tôt et utilise le principe de portée. Cette information est alors stockée dans J_i pour permettre les retours en arrière vers les variables ayant le plus influencé la

situation d'inconsistance. Plusieurs chercheurs tels Prosser [1995] et Lynce et Marques-Silvia [2002] ont travaillé à des versions évoluées de cette méthode.

Comme mentionné précédemment, les algorithmes de type BT sont des algorithmes comportant deux phases : la phase de regard en avant et la phase de retour en l'arrière. L'algorithme de *i-BackJumping* (i-BJ) implémente ces deux phases en une seule et présente ainsi une hybridation entre la phase de regard en avant et de retour en arrière. L'idée maîtresse de cet algorithme est de tenter d'instancier plus d'une variable à la fois. L'algorithme affectera donc des valeurs à i variables à la fois. Le *i-BJ* vérifie par le fait même si on peut ajouter i variables à la sous-séquence courante tout en préservant la consistance. Pour ce faire, la méthode *EstConsistant* doit être modifiée puisque l'on vérifie si une solution est extensible à i variables. De plus, l'enregistrement des ensembles de conflits sera modifié et on parlera alors d'*ensemble de conflits de niveau i*. Selon Dechter [2003], un ensemble de variables affectées est un *ensemble de conflits de niveau i* s'il n'est pas extensible à i autres variables en préservant la consistance.

2.2.3 Ordonnancement de variables et de valeurs

Selon Bessière, Meseguer *et al.* [1999], un des facteurs essentiels pour l'efficacité d'une recherche en arbre de type BT est le critère utilisé pour sélectionner la prochaine variable à instancier. C'est pourquoi, des méthodes d'ordonnancement de variables ont été élaborées. Ces méthodes visent à déterminer quelle sera la prochaine variable à affecter en utilisant certaines heuristiques. Ces méthodes peuvent aussi être utilisées pour déterminer l'ordre selon lequel les valeurs des domaines sont testées lors de l'affectation des variables.

Un premier type d'ordonnancement des variables se fie uniquement sur la structure du problème ou sur le graphe de contraintes. Étant donné que la structure du graphe de contraintes ne change aucunement lors de la résolution, on parle alors de méthodes d'*ordonnancement de variables statiques* (Statique Variables Ordering). Des heuristiques tel *minwidth* [Freuder 1982] qui choisit un ordre minimisant la largeur du graphe ou *maxdeg* [Dechter et Meiri 1994] qui ordonne les variables selon la taille de leur voisinage peuvent être employées.

Le second type est l'*ordonnancement dynamique de variables* (DVO). Comme son nom l'indique, le DVO choisira une variable dépendamment de l'état courant de la résolution du problème. Une heuristique de ce type est le *dom* [Haralick et Elliott 1980]. Pour U, l'ensemble des variables non affectées, on choisira la variable $X_i \in U$ tel que le domaine de X_i est le plus petit de tous les domaines de U. L'ordre d'affectation des variables sera donc modifié dépendamment des variables déjà affectées.

D'autres méthodes hybrides de sélection ont aussi été suggérées tel *dom+deg* [Frost et Dechter 1995] utilisant la méthode *dom* pour la sélection prioritaire et *deg* pour briser les égalités. Une autre heuristique est le *dom+futdeg* [Frost et Dechter 1995] qui brise les égalités en sélectionnant la variable ayant le plus grand degré futur. Finalement, d'autres heuristiques de type DVO peuvent être utilisées. Ce sont des heuristiques sélectionnant les variables selon la dureté des contraintes associées à celles-ci. Il faudra, dans ce cas précis, vérifier à quel point une contrainte restreint le problème. Cependant, comme mentionné par Bessière, Chmeiss *et al.* [2001], le défaut de ces heuristiques est la nécessité de vérifier la dureté des contraintes, ce qui est très gourmand en tests de contraintes.

L'ordonnancement de variables et de valeurs a également été étudié par Smith [1996]. Plus spécifiquement, les effets du *succeed-first* et du *fail-first* ont été expérimentés pour le problème de "car-sequencing" sur une chaîne de production. Le *fail-first* consiste à choisir la variable restreignant le plus possible les domaines des autres variables. Le *succeed-first* consiste à choisir la variable restreignant le moins possible les domaines des autres variables. Selon les expérimentations de Smith, une stratégie d'ordonnancement de valeurs de type *fail-first* obtient de meilleurs résultats pour ce problème. Cependant, il ne semble pas possible de pouvoir généraliser le phénomène à tous les problèmes de CSP. En effet, aucune de ces heuristiques n'a montré de comportement meilleur pour tous CSP [Bessière, Chmeiss *et al.* 2001]. L'heuristique à utiliser lors de la résolution sera donc grandement dépendante du problème à résoudre.

2.2.4 L'apprentissage

Une première solution au problème du retour en arrière vers les mêmes situations d'inconsistance soulevé par Dechter et Frost [2002] consiste à modifier les schémas de retour en arrière. Une seconde solution consiste, quant à elle, à utiliser des méthodes d'apprentissage des inconsistances (learning). Selon Frost et Dechter [1994], Dechter [1990] et Lynce et Marques-Silva [2002], les méthodes d'apprentissage peuvent favoriser la recherche et permettre de trouver plus rapidement les solutions en évitant de retomber sans cesse dans les mêmes situations d'inconsistance.

L'apprentissage vise à éliminer le plus possible de branches dans l'arbre de résolution. Pour ce faire, le but principal est d'identifier les ensembles de conflits minimaux et de les enregistrer sous la forme de nouvelles contraintes. Lors de la

détection d'une inconsistance tel que l'instanciation $S=(X_1=v_1, ..., X_{i-1}=v_{i-1})$ ne peut être consistante avec X_i, on ajoutera une contrainte qui permettra de rendre explicite l'inconsistance présente dans le graphe [Frost et Dechter 1994].

L'apprentissage peut être de deux formes soit *en profondeur* ou *en surface.* Cependant, l'analyse d'une situation d'inconsistance pour en identifier les ensembles de conflits minimaux est une tâche ardue. C'est pourquoi, on préférera parfois utiliser des méthodes d'apprentissage en surface telles le *value-based learning, le graph-based shallow learning* ou le *jump-back learning.* Ces méthodes d'apprentissage en surface permettent l'enregistrement d'ensembles de conflits non minimaux [Dechter 1990].

Un premier type d'apprentissage est le *value-based learning (VB-l)* [Frost et Dechter 1994]. Le principe consiste à enregistrer, selon *S*, l'ensemble de variables affectées ainsi que leurs valeurs respectives. Ce stockage d'information se fait dépendamment des contraintes du problème. En effet, on enregistre uniquement les couples *variable-valeur* qui entrent en conflit avec la feuille de la situation d'inconsistance. Soit le graphe de contraintes présenté à la Figure 2.5 ainsi que $O=X_1, X_2, X_3, X_4, X_5$ un ordonnancement des variables. Suite à l'instanciation de $X_1=a$, $X_2=b$, $X_3=b$, $X_4=c$, il est impossible d'instancier X_5 tout en préservant la consistance. Le *VB-l* enregistrera $X_1=a$, $X_2=b$, $X_4=c$ mais n'enregistrera pas $X_3=b$ puisque cette valeur du domaine de X_3 n'entre en conflit avec aucune valeur de X_5. La complexité de cet algorithme ne sera que $O(n)$ puisque l'on ne fait que vérifier si la valeur prise par chaque variable entre en conflit avec la feuille de la situation d'inconsistance et, dans l'affirmative, on ne fait que l'enregistrer sous forme de nouvelle contrainte. Cette méthode effectue un apprentissage en surface puisqu'elle ne garantit pas l'enregistrement d'ensembles de conflits minimaux.

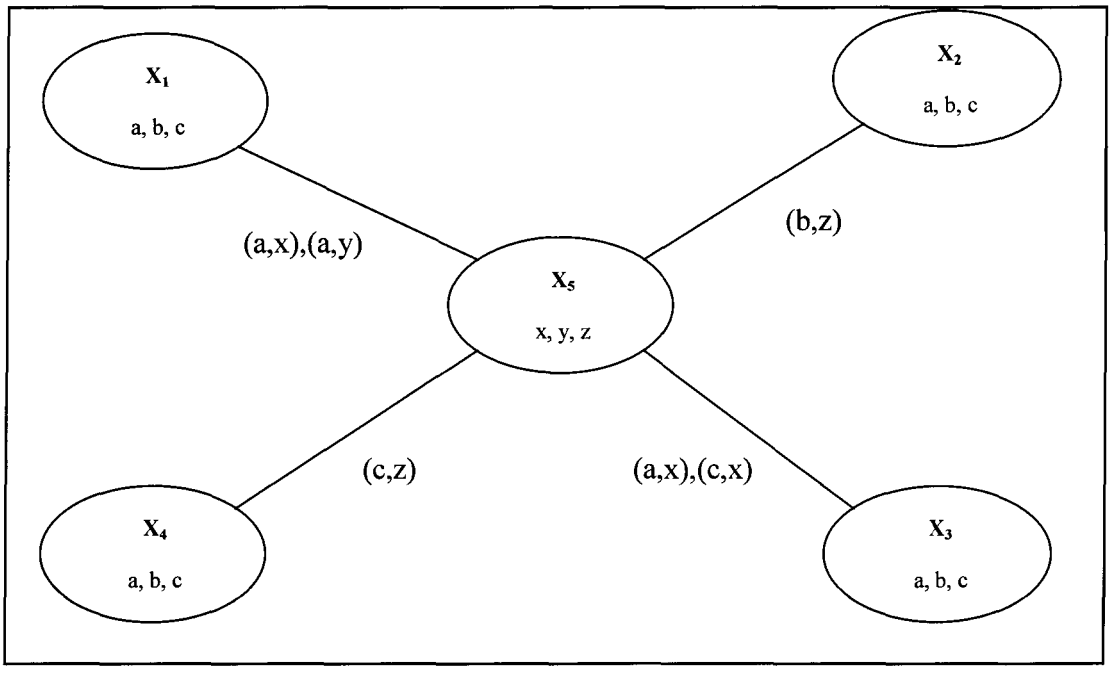

Figure 2.5 - Un CSP. Notons que les contraintes sont illustrées par des paires de variable-valeur impossibles.

Un second type d'apprentissage est le *Graph-based shallow learning (GB-l)*. Le *GB-l* utilise la structure du graphe pour identifier les inconsistances et les enregistrer. Cette méthode est surtout utilisée lors de la résolution d'un problème avec le *GB-BJ*. Lors de la détection d'une feuille de situation d'inconsistance tel que l'instanciation des variables $X_1, \ldots, X_i$ est inconsistante avec la variable X_{i+1}, l'algorithme enregistre le tout sous forme de nouvelles contraintes pour éviter que la situation ne se reproduise. Le *GB-l* est une méthode d'enregistrement en surface. L'algorithme de *GB-BJ* sera tel que décrit à la Figure 2.3, mais on y ajoutera l'enregistrement d'une contrainte lors d'une situation d'inconsistance. Pour l'exemple de la Figure 2.5 et l'instanciation déterminée précédemment, on enregistrera donc la négation de $X_1=a$, $X_2=b$, $X_3=b$, $X_4=c$ comme étant une nouvelle contrainte car ces quatre variables sont adjacentes à X_5. Contrairement au *VB-l*, on enregistre la valeur prise par X_3 car celle-ci est adjacente à X_5 dans le graphe de contraintes.

Le *jump-back learning (JB-l)* utilise, pour sa part, les informations récoltées durant la phase de retour en arrière du *BJ* pour enregistrer de nouvelles contraintes. Lors de la détection du conflit en X_5, l'algorithme vérifiera dans la séquence O s'il existe une sous-séquence $O' \subseteq O$ qui est en conflit avec X_5. Pour l'exemple de la Figure 2.5, on enregistrera donc $X_1=a$ *et* $X_2=b$ puisque, dès l'instanciation de ces deux variables, le domaine de la variable X_5 est vide.

Enfin, le *deep learning* [Dechter 1990] enregistre tous les ensembles de conflits minimaux présents dans une séquence inconsistante. Pour l'exemple de la Figure 2.5, on enregistrera $X_1=a$ *et* $X_2=b$ et $X_1=a$ *et* $X_4=c$ puisque les deux sous-séquences vident le domaine de X_5. Quoique étant la forme d'apprentissage la plus précise, le *D-l* demande un effort de calcul trop grand étant donné son niveau de complexité exponentiel pour la recherche de conflits minimaux [Frost et Dechter 1994].

2.2.5 L'algorithme de Forward-Checking

Les algorithmes de types *BT* ou *BJ* permettent d'instancier les variables une à la fois et appliquent le schéma de retour en arrière lorsque la variable courante a un domaine vide. On prend donc les décisions et on en subit les impacts par la suite. Le *Forward Checking (FC),* quant à lui, calcule l'impact d'une instanciation avant même que la décision ne soit prise. Avant d'affecter une valeur v à une variable X_i, on vérifiera l'impact sur les domaines pour toutes les autres variables non encore affectées. Si la valeur que l'on désire affecter à la variable vide le contenu du domaine d'une autre variable, on changera cette valeur.

L'algorithme de *FC* est décrit à la Figure 2.6 [Bacchus et Grove 1995]. Soit a, la solution courante et X_i, la variable que l'on tente d'instancier, v_l^i est la $l^{ième}$ valeur

présente dans le domaine de X_i. L'algorithme appelle la méthode *Check-Forward* qui permet de vérifier si l'instanciation courante est possible et, dans l'affirmative, passera à l'instanciation suivante. Dans le cas contraire, il appellera *Restaurer* qui restaurera le domaine de la variable X_i puisqu'on est pas parvenu à l'instancier [Bacchus et Grove 1995]. La méthode *Check-Forward* appliquera une procédure de filtrage des domaines qui sera dépendante du problème traité.

```
Procédure FC (X_i)
      POUR CHAQUE v_l^i ∈ D_i FAIRE
            X_i ← v_l^i
            SI D_i = nul ALORS
                  SI i = N ALORS
                        Afficher la solution trouvée
                  SINON
                        SI Check-Forward (X_i) ALORS
                              FC (X_i+1)
                        Restaurer (X_i)
```

Figure 2.6 - Algorithme de Forward Checking

Le FC effectuera donc un filtrage des valeurs de chacune des variables qui sont inconsistantes avec l'instanciation désirée et supprimera ces valeurs. On aura donc une diminution des domaines des variables et on détectera de nombreuses inconsistances plus tôt qu'avec un algorithme de BT conventionnel.

Tel que soulevé par Dent et Mercer [1994], le FC nécessite un grand nombre de vérifications de la consistance. C'est pourquoi, le *minimal forward checking* (MFC) a été développé par Dent et Mercer [1994]. Le MFC conserve la même méthode de détection d'inconsistances que le FC. Cependant, il n'effectuera pas tous les tests de consistance entre les valeurs des domaines des variables. Il poursuivra la résolution dès qu'il rencontrera au moins une valeur consistante dans chacun des domaines des variables adjacentes à la variable courante. Les expérimentations de Dent et Mercer ont démontré que plus la taille des domaines augmentent, plus le MFC surpasse le FC.

Le MFC peut être amélioré par l'utilisation d'une méthode de *Backmarking* (BM) [Dent et Mercer 1994] [Kwan et Tsang 1996]. Le BM consiste à utiliser des structures de stockage des résultats de tests de consistance. Suite à la découverte de la consistance ou de l'inconsistance entre deux valeurs de domaines de variables adjacentes, on enregistrera ce résultat. C'est ainsi que, lors des retours en arrière, ce résultat pourra être utilisé à plusieurs reprises sans vérifier à nouveau la consistance. On évitera ainsi de nombreuses vérifications inutiles.

2.2.6 Les algorithmes de propagation de contraintes

Les algorithmes de propagation de contraintes, aussi appelés algorithmes d'arc consistance, ont pour but de supprimer les valeurs $v \in D_i$ qui sont inconsistantes avec D_j tel que X_i et X_j sont adjacents dans le graphe de contraintes et ce, pour toutes les variables X_i du problème. La propagation définira les arcs (X_i, X_j) sur lesquels on doit vérifier la consistance entre D_i et D_j pour que chacun des arcs du graphe de contraintes soit consistant. Cette opération fera en sorte que, d'un graphe de contraintes G, on passera à un graphe G' dont les domaines sont réduits. Cette réduction des domaines facilitera la recherche de solutions et permettra aussi la détection de situations d'inconsistance au plus tôt.

La propagation de contraintes peut être appliquée à des contraintes de nature diverses. En effet, pour les contraintes unaires, on définira un schéma de vérification spécifique dit de consistance de sommet ou de noeud. De même, les algorithmes d'arc consistance prendront souvent en compte des contraintes de degré deux, c'est-à-dire des contraintes binaires.

La propagation de contraintes est parfois utilisée en phase de prétraitement où l'on supprimera certains membres inconsistants des domaines avant de lancer un algorithme de type BT. Dans d'autres cas, on utilisera l'algorithme de propagation de contraintes lors de la recherche de solutions par l'utilisation d'un algorithme de BackTracking. Dans ce cas précis, on parlera d'algorithme de maintient de l'arc consistance (MAC). Suite à chacune des instanciations de variables, on vérifiera l'impact produit sur les autres variables du problème. Cette vérification fera en sorte que certains membres de domaines seront rendus inconsistants et seront alors supprimés.

On définit deux orientations possibles aux algorithmes de propagation de contraintes. Premièrement, l'algorithme peut être orienté sur les contraintes et maintenir une liste de contraintes telle que l'on doit réviser les domaines des variables comprises dans la portée de celles-ci. Deuxièmement, l'algorithme peut être orienté sur les variables et maintenir une liste de variables dont les domaines doivent être révisés.

La *consistance de sommet ou de noeud* consiste à vérifier, pour chaque sommet du graphe de contraintes, si aucune valeur présente dans le domaine de la variable lui étant associée ne vient enfreindre une contrainte unaire du problème [Mackworth 1977]. L'algorithme de consistance de nœud est illustré à la Figure 2.7. Notons que l'auteur utilise ici $P_i\ (v)$ pour indiquer si l'affectation de la valeur v à la variable X_i est consistante.

```
Procédure NC(i)
      Di ← Di ∩ { v | Pi(v) }
      POUR i ← 1 à n FAIRE
            NC(i)
```

Figure 2.7 - Algorithme de consistance de noeud

L'algorithme de *consistance d'arc* vérifie si, pour tout $v_i \in D_i$ et $v_j \in D_j$, C_{ij} n'est pas enfreinte. Plus clairement, on vérifie si, pour toutes les variables du problème, il existe un membre du domaine de toutes les variables adjacentes qui respecte les contraintes du problème. Le maintient de l'arc consistance est un processus long et répétitif. En effet, lors de chaque instanciation de variable, il devient nécessaire de vérifier l'impact sur le domaine des autres variables du problème. La structure même du graphe de contraintes a donc un effet sur l'effort de vérification de la consistance d'arc.

Les recherches sur le renforcement de la consistance d'arc débutèrent en 1977 avec les travaux de Mackworth [1977] qui élabora quatre algorithmes soit *Revise (ArcConsistance), AC-1, AC-2* et *AC-3*. Par la suite, Mohr et Henderson [1986] développèrent le *AC-4* qui fut une amélioration notable des algorithmes précédents. En 1992, Hentenryck, Deville *et al.* [1992] développèrent le *AC-5* et, en 1994, Bessière [1994] démontra la méthode *AC-6* permettant de pallier aux problèmes de complexité en terme d'espace du *AC-4*. Bessière, Freuder *et al.* [1995] démontrèrent, en 1995, l'efficacité du *AC-7* pour diminuer le nombre de vérifications de consistance inutiles pour les graphes de contraintes bidirectionnelles. Enfin, Bessière et Régin [2001] développèrent le AC2000 et le AC2001 en 2001.

Tel que mentionné précédemment, le tout premier algorithme de vérification de la consistance d'arc fut développé par Mackworth [1977]. Cet algorithme, illustré à la Figure 2.8, supprime les membres du domaine des variables qui ne sont plus consistants avec l'état du graphe de contraintes. L'algorithme prend en entrée deux variables adjacentes dans le graphe de contraintes et renvoie une information sur la consistance ou non de l'arc reliant les deux variables. La complexité de cet algorithme est $O(k^2)$ où

k est la taille du plus grand domaine puisque, dans le pire des cas, on devra réviser toutes les possibilités entre deux domaines de grandeur k.

```
Procédure ArcConsistance (Xi, Xj)
      DELETE ← faux
      POUR CHAQUE vi ∈ Di FAIRE
            S'il n'existe aucun vj ∈ Dj tel que (Xi ← vi, Xj ← vj) est
            consistant ALORS
                  Supprimer vi dans Di
                  DELETE ← vrai
      Retourner DELETE
```

Figure 2.8 - Algorithme de consistance d'arc

Quoique en partie efficace et ne demandant pas un effort de calcul impressionnant, cet algorithme ne garantit pas la consistance globale du graphe puisqu'il ne vérifie qu'un arc du problème [Kumar 1992]. En effet, selon Mackworth [1977], l'algorithme, suite à un appel de la sous-procédure *ArcConsistance* (*Revise*) pour un arc (X_i, X_j), rendra cet arc consistant. Cependant, l'appel de cette même sous-procédure pour un arc (X_j, X_k) pourra rendre l'arc (X_i, X_j) inconsistant ce qui veut dire qu'un simple appel de la procédure *ArcConsistance* pour chaque arc du problème n'est pas suffisant pour garantir la consistance. D'autres algorithmes plus complets ont été développés pour augmenter la consistance locale.

Le *AC-1* est le premier algorithme à garantir la consistance d'arcs du graphe. Cet algorithme s'exécute tant et aussi longtemps que la procédure *ArcConsistance* modifie le domaine d'une des variables. Le *AC-1* est décrit à la Figure 2.9.

La file d'arcs à réviser, notée Q, stocke ici toutes les relations du graphe de contraintes. Par la suite, on utilise la procédure *ArcConsistance* pour vérifier la consistance de tous les arcs du graphe et, lors de la modification du domaine d'une variable, on vérifiera l'implication de ce changement sur toutes les autres variables du problème. Cette méthode rendra le graphe Arc Consistant. Cependant, l'ordre de

complexité passera de $O(k^2)$ à $O(enk^3)$ où e sera le nombre de contraintes du problème et n le nombre de variables. En effet, un cycle de la boucle du *AC-1* implique un traitement dans $O(ek^2)$ puisque le passage à travers toutes les contraintes binaires est dans un temps $O(k^2)$ et que l'on a e contraintes. De plus, dans le pire des cas, l'algorithme ne supprimera qu'une valeur dans le domaine d'une seule variable et, comme il y a nk valeurs possibles, on obtient $O(nk) * O(ek^2)$ qui est égal à $O(enk^3)$ [Kumar 1992]. L'algorithme effectue donc de nombreuses vérifications inutiles. En effet, tel que mentionné par Mackworth, la suppression d'un seul membre d'un domaine suppose la révision de tous les arcs alors, qu'en réalité, seule une fraction de ceux-ci doit être révisée.

```
Procédure AC-1
      Q ← {(Xi, Xj) ∈ arcs(G), i ≠ j}
      RÉPÉTER
            CHANGE ← faux
            POUR CHAQUE (Xi, Xj) ∈ Q FAIRE
                  CHANGE ← (ArcConsistance(Xi, Xj) OU CHANGE)
      TANT QUE NON (CHANGE)
```

Figure 2.9 - Algorithme AC-1

C'est pourquoi, un deuxième algorithme, s'inspirant de la méthode de filtrage de Waltz [1972], fut utilisé pour diminuer le nombre d'appels à la procédure *ArcConsistance*. Cet algorithme effectue un passage sur chacun des nœuds X_i du graphe et vérifie la consistance avec toutes les variables X_j telles que $j < i$.

L'algorithme *AC-2* est décrit à la Figure 2.10. L'algorithme s'exécute sur les n variables du problème et débute par une vérification de la consistance de nœud tel que décrite par Mackworth. Suite à cette vérification, on stocke dans Q tous les arcs sortants de X_i et dans Q' tous les arcs entrants.

Lors de la suppression d'un arc (X_k, X_m) par la sous-procédure *Extraire,* on vérifie par la procédure *ArcConsistance* si des modifications doivent être faites dans le

domaine de cette variable. Si c'est le cas, tous les arcs entrants et sortants de cette variable seront revérifiés. Sinon, l'algorithme continuera la vérification de la consistance pour le prochain arc du problème.

```
Procédure AC-2
      POUR i ← 1 à n FAIRE
            NC(i)
            Q ← { (X_i, X_j) | (X_i, X_j) ∈ arcs(G), j < i }
            Q' ← { (X_j, X_i) | (X_j, X_i) ∈ arcs(G), j < i }
            TANT QUE Q n'est pas vide FAIRE
                  TANT QUE Q n'est pas vide FAIRE
                        Extraire(X_k, X_m) de Q
                        SI ArcConsistance(k, m) ALORS
                              Q' ← Q' ∪ { (X_p, X_k) | (X_p, X_k) ∈ arcs(G),
                              p ≤ i, p ≠ m }
                        Q ← Q'
                        Q' ← vide
```

Figure 2.10 - Algorithme AC-2

Un autre algorithme, le *AC-3*, a été suggéré par Mackworth. Le *AC-3* abandonne l'idée du *AC-2* qui était de rendre le graphe consistant en utilisant une boucle passant sur chacun des noeuds. Il utilise simplement une file Q qui contient tous les arcs du graphe. On appliquera *ArcConsistance* sur chacun des arcs du graphe. Si cette procédure modifie le domaine d'une variable, on ajoutera tous les arcs adjacents à cette variable dans la file Q. L'algorithme se termine lorsque la file des arcs à vérifier Q est vide. Dans le pire des cas, cet algorithme impliquera une complexité de $O(ek^3)$. L'algorithme de *AC-3* est décrit à la Figure 2.11.

Notons que l'algorithme commence avec un même nombre d'arcs à vérifier que le *AC-1*. Cependant, l'algorithme ne vérifiera par la suite que les arcs ayant été modifiés dans un premier temps par l'algorithme *ArcConsistance* et permettra ainsi de diminuer le nombre de vérifications.

```
Procédure AC-3
      Q ← {(Xi, Xj) ∈ arcs(G), i ≠ j}
      TANT QUE Q n'est pas vide FAIRE
            Extraire (Xk, Xm) ∈ Q
            SI ArcConsistance(Xk, Xm) ALORS
                  Q ← «{(Xi, Xk) tel que (Xi, Xk)∈ arcs(G),
                        i ≠ k, i ≠ m}
```

Figure 2.11 - Algorithme AC-3

Suite au *AC-3*, le *AC-4* a été développé par Mohr et Henderson [1986]. Cet algorithme exploite la notion de support pour favoriser le maintient de l'arc consistance. Un membre de domaine v sera déclaré supporté ou viable tant et aussi longtemps que, pour toutes les variables X_j telles que $j \neq i$, il existe au moins un membre de D_j consistant avec l'affectation de v à X_i.

Cet algorithme utilise un compteur, noté *Compteur[(i, j), v]*, entre chaque membre des domaines des variables adjacentes spécifiant, pour un arc (X_i, X_j), le nombre de valeurs $b \in D_j$ consistantes avec l'affectation de v à X_i. On déclarera aussi $Supporté_{jc}$ comme étant l'ensemble des couples *variable-valeur* supportant l'affectation de c à j pour toutes les variables X_i telles que $i \neq j$. On aura donc $(X_i, b) \in Supporté_{jc}$ indiquant que l'affectation de b à X_i est consistant avec l'affectation de c à X_j. L'algorithme utilisera aussi un élément nommé *Non-Supporté* représentant les couples *variable-valeur* inconsistants et donc retirés des domaines. Enfin, une structure de données nommée *Liste* stocke les éléments qui doivent être vérifiés.

L'algorithme, illustré en Figure 2.12, débute par initialiser correctement les éléments *Supporté*, *Non-Supporté*, la liste des couples *variable-valeur* à vérifier et les *Compteurs*. Par la suite, la phase de traitement effectue toutes les vérifications de consistance présente dans *Liste* et supprime toutes les valeurs inconsistantes présentes dans le domaine des variables du problème. Sommairement, la première boucle de la

phase d'initialisation permet de vider le contenu des variables *Supporté* et *Non-Supporté*. La seconde boucle permet de vérifier, pour chaque contrainte du problème impliquant une variable X_i et X_j, la consistance de chaque élément du domaine de X_i avec le domaine de X_j et d'indiquer, pour chacun de ceux-ci, si les valeurs sont supportées ou non. Pour ce faire, on utilise la méthode *EstAdmissible* qui détermine, pour une variable X_i, une valeur $v \in D_i$, une variable X_j et une valeur $b \in D_j$ si l'affectation de v à X_i est consistante avec l'affectation de b à X_j. S'il n'existe aucune valeur consistante possible, c'est-à-dire que $Total = 0$, on enlèvera les valeurs de domaine inconsistantes et on indiquera qu'il n'existe aucune valeur possible pour le couple (v, X_i). Finalement, le *compteur* du nombre de valeurs présentes dans le domaine de X_j et compatibles avec l'affectation de la valeur v à la variable X_i est affecté à la valeur de *Total*.

La phase de traitement permet de vérifier la consistance de chaque couple *variable-valeur* présent dans *Liste*. Il effectue, pour toutes les valeurs du domaine de X_j consistantes avec l'affectation de la valeur v à X_i, des suppressions de valeurs dans le domaine de X_j jusqu'à ce que le couple (X_j, b) ne soit plus présent dans la liste.

Suite au *AC-4*, le *AC-5* fut proposé par Hentenryck, Deville *et al.* [1992]. Cet algorithme est une approche générique au problème de l'arc consistance, c'est-à-dire que son implantation est assez large pour fonctionner sur des problèmes de nature diverse. Cependant, les auteurs ont travaillé sur une spécialisation de la procédure à suivre pour les contraintes fonctionnelles, anti-fonctionnelles et monotones. Ces spécifications permettent de diminuer la complexité de l'algorithme à $O(ed)$ où e est le nombre d'arcs du problème et d est la taille du plus grand domaine.

```
Procédure AC-4
      // Phase d'initialisation :
      Liste ← nul
      POUR chaque Xi ∈ X FAIRE
            POUR chaque v ∈ Di FAIRE
                  Supporté [Xi, v] ← nul
                  Non-Supporté [Xi, v] ← FAUX
      POUR CHAQUE (Xi, Xj) ∈ C FAIRE
            POUR CHAQUE v ∈ Di FAIRE
                  Total ← 0
                  POUR CHAQUE b ∈ Dj FAIRE
                        SI EstAdmissible(Xi, v, Xj, b) ALORS
                              Total ← Total + 1
                              Supporté [Xj, b] ← Supporté [Xj,
                              b] ∪ {(Xi, v)}
                  SI Total = 0 ALORS
                        Di ← Di - v
                        Liste ← Liste ∪ (Xi, v)
                        Non-Supporté [Xi, v] ← VRAI
                  Compteur [(Xi, Xj), v] ← Total
      Liste ← { (Xi, b) tel que Non-Supporté (Xi, b) = 1 }
      // Phase de traitement
      TANT QUE Liste ≠ 0 FAIRE
            Supprimer (Xi, v) de Liste
            POUR CHAQUE (Xj, b) ∈ Supporté [Xi, v] FAIRE
                  Compteur [(Xj, Xi), b] ← Compteur [(Xj, Xi), b] - 1
                  SI Compteur [(Xj, Xi), b] = FAUX ET
                        Non-Supporté[Xj, b] = FAUX ALORS
                        Dj ← Dj - b
                        Liste ← Liste ∪ (Xj, b)
                        Non-Supporté [Xj, b] ← VRAI
```

Figure 2.12 - Algorithme AC-4

L'algorithme utilise deux procédures soit *ArcCons* et *LocalArcCons*. *ArcCons* permet, dans un premier temps, de déterminer, pour chacun des arcs (X_i, X_j), un ensemble de valeurs pour la variable X_i non-supportées par D_j et supprimera celles-ci. *LocalArcCons* permet, dans un second temps, de déterminer l'ensemble de valeurs $v \in D_i$ qui n'est plus supporté étant donné les suppressions effectuées par *ArcCons* dans D_j.

L'algorithme utilise, comme le *AC-3* et le *AC-4*, une liste permettant la propagation de contraintes. Cependant, plutôt que d'utiliser une liste d'arcs (X_i, X_j) ou une liste de paires *variable–valeur* à réviser, il utilise une liste d'éléments de type $<(X_i, X_j), v>$ où (X_i, X_j) est un arc et v est une valeur supprimée du domaine de X_j.

L'élément v sera alors la justification de l'appel à la procédure de révision des domaines sur l'arc (X_i, X_j). Cette structure de données facilitera le traitement du problème par la méthode *LocalArcCons*.

D'autres chercheurs ont tenté d'améliorer le *AC-4* et ce, pour tous les types de contraintes. Selon Bessière [1994], le problème du *AC-4* est sa complexité en terme d'espace et son mauvais temps moyen de résolution. Cette situation fait en sorte que, pour les problèmes comportant beaucoup de contraintes, il devient parfois plus avantageux d'utiliser le *AC-3* même s'il est non optimal. C'est pourquoi, Bessière et Cordier ont suggéré un autre algorithme, nommé *AC-6,* gardant la complexité en terme de temps du *AC-4* tout en diminuant la complexité moyenne en terme d'espace de celui-ci et ce, pour tout type de contraintes.

L'algorithme utilise une table de booléens M indiquant si une valeur du domaine d'une variable X_i est présente dans le domaine courant de celle-ci. On aura donc $M(X_i,v)$=*vrai* pour $v \in D_i$ qui indique que v est encore présent dans le domaine de X_i. On utilise aussi la procédure *first*(D_i) qui retourne la plus petite valeur présente dans D_i, la procédure *last*(D_i), qui retourne la plus grande valeur présente dans D_i et *next*(v, D_i) qui retourne la valeur v' telle que toutes les valeurs plus grandes que v et plus petites que v' ont été supprimées de D_i. On utilise aussi une structure de données nommée S_{jb} qui représente tous les plus petits couples *variable-valeur* présents dans D_j qui supportent l'affectation de la valeur b à la variable j et tels que $j \neq i$. Ce dernier élément est équivalent à l'élément *Supporté* du *AC-4* mais on se contente ici de garder en mémoire la plus petite valeur possible plutôt que toutes les valeurs possibles. C'est cet élément qui fait en sorte que l'espace nécessaire au *AC-6* est inférieur au *AC-4*. Il est à noter que le principe de *compteurs* utilisé dans le *AC-4* n'est plus utilisé dans le *AC-6*.

Enfin, comme pour le *AC-4*, on utilise aussi une liste d'éléments supprimés nommée *Liste* dont l'impact de la suppression n'a pas encore été testé sur le reste du graphe et dont la propagation doit être effectuée. L'algorithme s'exécute tant et aussi longtemps que la liste d'éléments à réviser n'est pas vide.

L'algorithme a été testé sur le problème des reines et il fut démontré par Bessière et Cordier que le *AC-6* nécessite moins d'opérations et de tests que le *AC-3* et le *AC-4*. Il est à noter que cet écart s'amplifie à mesure que la taille des problèmes augmente.

Un autre algorithme d'arc consistance, le AC-7, a été suggéré par Bessière, Freuder *et al.* [1995]. Il est à noter que cet algorithme implique la présence de contraintes bidirectionnelles. Le *AC-7* utilise cette particularité des contraintes pour diminuer le nombre de vérifications à faire lors de la résolution. Selon les auteurs, une redondance des vérifications est notée lors de la résolution de problèmes où les contraintes sont bidirectionnelles. En effet, pour une contrainte C_{ij} bidirectionnelle, la vérification de la consistance de D_i avec D_j décèlera des membres inconsistants que l'on détectera de nouveau lors de la vérification de la consistance de D_j avec D_i. Le AC-7 permet d'éviter le dédoublement des vérifications de consistance entre les variables adjacentes du graphe de contraintes.

Enfin, Bessière et Régin suggèrerent le *AC2000* et le *AC2001* comme méthode de propagation de contraintes [Bessière et Régin 2001]. Le *AC2000* utilise un principe similaire au *AC-3*, c'est-à-dire qu'il ne nécessite aucune structure de stockage des supports. Il améliore le *AC-3* en éliminant de nombreuses vérifications de consistance inutiles. En effet, il fut observé par les auteurs que, avec le *AC-3*, la suppression d'une seule valeur $v \in D_j$ sous-entend l'ajout de la variable X_j dans la file de propagation. Cet ajout fera que la procédure de révision de domaine sera appliquée pour chaque

contrainte C_{ij} impliquant la variable X_j et ce, pour chaque élément de domaine D_i et D_j. Cette situation fait en sorte que, au début de la résolution, le nombre de vérifications de consistance est très élevé puisque la taille des domaines est plus grande. Les auteurs ont donc élaboré une méthode diminuant le nombre de vérifications de consistance nécessaires.

Posons $\Delta(X_j)$, l'ensemble des valeurs supprimées depuis la dernière propagation de X_j et *lazymode* le ratio maximum entre le nombre d'éléments de D_j et le nombre d'élément de $\Delta(X_j)$. Lors de l'appel à la procédure de révision, au lieu de toujours vérifier si la valeur $v \in D_i$ a un support dans D_j, on vérifiera, dans certains cas, si v a perdu un support, c'est-à-dire si un support de v est présent dans $\Delta(X_j)$. On alternera les phases de vérification de support avec les phases de vérification de la perte d'un support dépendamment du ratio *lazymode*. Si le ratio entre le nombre d'éléments de $\Delta(X_j)$ et le nombre d'éléments présents dans D_j est inférieur à *lazymode*, on vérifiera la perte de support. Dans le cas contraire, on vérifiera la présence d'un support dans chacune des variables adjacentes comme dans le *AC-3*. Les auteurs démontrèrent que cette méthode permet de diminuer le nombre de vérifications de consistance tout en ayant une complexité de $O(ed^3)$. La complexité en terme d'espace varie entre $O(nd)$, si on utilise une propagation telle que ce sont les variables qui sont stockées dans la liste de propagation, et $O(ed)$ si on utilise une propagation telle que ce sont les contraintes qui sont présentes dans le liste de propagation.

Le AC2001, quant à lui, implique l'ajout d'une structure de stockage additionnelle. L'algorithme conservera, pour chacune des valeurs $v_i \in D_i$, la dernière valeur $v_j \in D_j$ consistante avec l'affectation de v_i à i. On enregistrera le tout sous le forme *Last* (X_i, v_i, X_j). Lors des tests de consistance, on se contentera uniquement de

vérifier si *Last* (X_i, v_i, X_j) existe toujours dans D_j. Dans l'affirmative, aucun test de consistance ne sera nécessaire. Sinon, on appliquera une recherche dans D_j pour trouver le prochain *Last* (X_i, v_i, X_j).

Les auteurs démontrèrent que le AC2001 est optimal en terme de complexité soit $O(ed^2)$ et que sa complexité en terme d'espace est de $O(ed)$. C'est donc le premier algorithme optimal n'utilisant pas une liste complète des supports.

2.2.7 Méthodes parallèles et distribution des méthodes de résolution

Le temps nécessaire à la résolution d'un problème de CSP peut être diminué par l'utilisation d'ordinateurs parallèles ou encore par la distribution du calcul sur plusieurs ordinateurs partageant un réseau de communication commun. Plusieurs stratégies de résolution ont été décrites dans la littérature. Parmi les méthodes suggérées figurent la distribution des algorithmes d'arc consistance. Nguyen et Deville [1998] ont démontré l'exactitude d'un algorithme AC-4 distribué fonctionnant avec un procédé d'échanges de messages entre les ordinateurs participant à la résolution.

Un autre schéma de résolution consiste à distribuer l'algorithme de BT conventionnel. Un algorithme, développé par Bessière, Maestre *et al.* [2002], nommé Asynchrone BackTracking (ABT) consiste à la distribution du BT ainsi qu'au partage des nogoods pour diminuer l'espace de recherche. Celui-ci détermine les liens de communications entre les différents agents contribuant à la résolution du problème. L'échange de nogoods permet à ces agents d'éviter de nombreuses situations d'inconsistance.

De plus, un algorithme appelé Backtracking distribué (DiBT) a été développé par Hamadi, Bessière *et al.* [1998]. L'approche suggérée n'implique aucun échange de

nogoods mais est basée sur des échanges de sous-solutions consistantes entre les agents. Cependant, le DiBT fut prouvé incomplet, c'est-à-dire qu'il ne garantit pas un parcours complet de l'espace de solutions.

Certains auteurs ont aussi approfondi le côté coopératif de la résolution distribuée ou parallèle. L'enregistrement des nogoods a ainsi été étudié en détail par Dechter [1990] et Frost et Dechter [1994] ainsi que par Terrioux [2002]. Ces algorithmes servent à spécifier les algorithmes de reconnaissance des nogoods ainsi qu'à déterminer quels nogoods sont utiles et doivent être enregistrés. La manipulation de ceux-ci en cours de recherche peut aussi représenter un aspect important de l'efficacité des algorithmes utilisant le principe d'apprentissage.

L'utilisation de schémas parallèles peut aussi favoriser la résolution de problèmes de CSP. Entre autres, Rao et Kumar démontrèrent l'efficacité du BT parallèle [Rao et Kumar 1993]. Enfin, des algorithmes de décomposition de problème peuvent aussi être utilisés. Habbas, Krajecki *et al.* [2000] développèrent une approche de décomposition de problèmes pour la résolution parallèle. Ces méthodes consistent en une décomposition des domaines du problème ou encore en une distribution de l'arbre de recherche sur plusieurs processeurs. Cette décomposition permet de mettre à profit la résolution de plusieurs sous-problèmes du problème initial afin de résoudre ce dernier.

2.3 Objectifs de la recherche

Les méthodes de résolution de problèmes de CSP sont diverses. Le but du présent chapitre était de définir une vue d'ensemble de ces méthodes et d'en déterminer les avantages et les désavantages. Cet état de l'art des méthodes de résolution de CSP

servira de base à l'utilisation de celles-ci pour la résolution d'un problème à connotation industrielle spécifique.

Les méthodes de résolution de CSP sont souvent des schémas généraux qui doivent être adaptés en fonction du problème à résoudre. Plus particulièrement, les méthodes axées sur la diminution de domaines telles la propagation de contraintes ou le *forward-checking* reposent sur des notions de filtrage directement dépendantes du problème à traiter. C'est pourquoi, l'étude des contraintes du problème à résoudre est une étape importante du développement d'algorithmes de résolution de CSP.

L'objectif principal de ce travail de recherche est la résolution du problème de "car-sequencing" par les méthodes de résolution de CSP. C'est pourquoi, un premier objectif spécifique sera de développer un algorithme général de résolution du problème de "car-sequencing" s'inspirant des méthodes de résolution de CSP. Un second objectif spécifique sera de quantifier l'impact de l'ajout d'éléments tels des heuristiques guidant la descente dans l'arbre de résolution et l'utilisation des listes de nogoods. Finalement, un troisième objectif spécifique de la recherche sera le développement d'un algorithme de résolution concurrente dans le but de diversifier celle-ci et, éventuellement, de permettre l'utilisation d'ordinateurs parallèles ou de distribution du calcul.

Le chapitre 2 a permis de définir les principaux algorithmes de résolution de CSP. Il sert de base au chapitre suivant qui décrira, pour un problème spécifique, le développement d'une méthode de résolution. Chacune des phases de développement de l'algorithme seront décrites et des tests seront effectués afin de déterminer l'impact de l'ajout de chacun des éléments de la méthode. De plus, les travaux effectués permettront de définir l'efficacité de la résolution du problème de "car-sequencing" par les méthodes de résolution de CSP.

CHAPITRE 3

RÉSOLUTION DU PROBLÈME DE "CAR-SEQUENCING" PAR LES MÉTHODES DE SATISFACTION DE CONTRAINTES

3.1 Introduction

Le problème de "car-sequencing" est un problème d'ordonnancement pouvant être modélisé sous la forme d'un problème de CSP. C'est pourquoi, les méthodes de résolution décrites au chapitre précédent peuvent être utilisées pour le résoudre. Ce chapitre décrira l'application des formalismes de résolution de problèmes de CSP à ce problème d'ordonnancement à connotation industrielle. Plus particulièrement, les travaux effectués définissent l'utilisation d'une méthode de recherche de solutions utilisant le principe de retour en arrière et de détection d'inconsistances par le filtrage des domaines du problème.

Premièrement, on retrouve dans ce chapitre une étude du problème de "car-sequencing" pour en relever les particularités susceptibles d'en influencer la résolution. La connaissance de la structure du problème permettra de préciser les méthodes qui peuvent être utilisées pour le résoudre. Il est essentiel, lors de la résolution d'un problème, de déterminer le graphe de contraintes associé ainsi que la meilleure formulation à utiliser.

Deuxièmement, une méthode de résolution du problème de "car-sequencing" basée sur les principes de filtrage de domaines induits par les contraintes qui le composent sera décrite. Ce filtrage a pour but de permettre de diminuer l'aire de recherche de solutions afin d'accélérer la résolution. Les résultats démontreront l'efficacité d'un algorithme de FC sur la résolution de problèmes de CSP et plus particulièrement sur la résolution du problème de "car-sequencing".

Troisièmement, l'ajout d'éléments permettant de guider un algorithme de FC lors de la recherche de solutions sera évalué. Une étude des méthodes d'ordonnancement de

variables et de valeurs a été réalisée afin de déterminer certaines heuristiques pouvant être appliquées au problème pour en accélérer la résolution.

Quatrièmement, l'ajout d'une méthode d'utilisation de listes de nogoods permettant d'éviter les situations d'inconsistance au plus tôt est décrit. Cette méthode a pour but de bonifier l'algorithme de FC de base et ainsi d'établir un schéma de résolution plus performant.

Finalement, une méthode de résolution concurrente de plusieurs régions de l'espace de recherche de solution est élaborée. Celle-ci a été développée afin de permettre une meilleure diversification de la recherche de solutions. On présente, dans un premier temps, une méthode de subdivision du problème initial en sous-problèmes et les mécanismes par lesquels ceux-ci sont affectés à plusieurs processus distincts. Le *FC* développé est alors utilisé pour résoudre chacun de ces sous-problèmes. Dans un second temps, une méthode de subdivision des ressources matérielles est décrite. L'algorithme étant exécuté sur un ordinateur doté d'un seul processeur, les ressources doivent être affectées de manière à favoriser les processus les plus performants tout en ne négligeant pas les processus moins performants. Pour ce faire, on utilise le « multiprogramming » qui permet la gestion de plusieurs processus utilisant en concurrence les ressources d'un seul processeur [Stallings 2001]. Le développement de cette méthode concurrente a pour but l'utilisation éventuelle d'ordinateurs parallèles pour la résolution du problème de "car-sequencing". Une analyse des résultats obtenus et une comparaison avec les meilleurs résultats obtenus par le *FC* et avec les résultats obtenus par ILOG Solver 6.0 sont finalement présentées.

3.2 Description générale du problème de "car-sequencing"

Le problème de "car-sequencing" consiste à déterminer l'ordre selon lequel un ensemble de véhicules sont fabriqués sur une chaîne d'assemblage. Certains constructeurs, dont *Renault*[2], subdivisent cette chaîne en trois ateliers de fabrication. Le premier confectionne les carrosseries des véhicules, le second applique la peinture sur celles-ci et le dernier procède à l'assemblage des véhicules.

En ce qui concerne le premier atelier, la technologie utilisée fait en sorte que la séquence de production des voitures n'a pas d'importance. Cet atelier peut donc être négligé dans la modélisation. En ce qui concerne l'atelier de peinture, la séquence devra favoriser des voitures consécutives de même couleur pour éviter les coûts associés au nettoyage des pistolets. Pour sa part, l'atelier d'assemblage comprend plusieurs postes de montage pour les différentes options des véhicules. Certains postes possèdent une capacité limitée de production qui est exprimée sous la forme de contraintes de capacité. De plus, on distingue des contraintes de capacité fortes liées à certaines options pour lesquelles il est très difficile d'accroître le rythme de production et des contraintes de capacité faibles pour lesquelles du personnel additionnel peut permettre un accroissement de la production. Enfin, le problème de "car-sequencing" industriel doit prendre aussi en compte l'état de la chaîne de production laissée par la production de la journée précédente. Cet élément contraindra davantage le choix des voitures présentes au début de la séquence de production.

---

[2] http://www.prism.uvsq.fr/~vdc/ROADEF/CHALLENGES/2005/#sujet

Le problème de "car-sequencing" industriel est donc un problème multi-objectifs où l'on doit déterminer une solution ne violant pas de contraintes de capacité et minimisant les coûts associés aux purges de l'atelier de peinture. Ces objectifs étant contradictoires, le constructeur automobile aborde souvent le problème en fixant un ordre de priorité aux différents objectifs.

Le problème théorique de "car-sequencing" retrouvées dans la littérature est une simplification du problème industriel. En effet, les instances théoriques ne se préoccupe que de l'atelier d'assemblage et considèrent uniquement les contraintes de capacité. Elles visent donc la recherche de solutions pour un seul des objectifs du problème industriel de "car-sequencing". C'est sur les instances théoriques que seront appliquées les approches de résolution de CSP décrites au chapitre précédent. Des précisions supplémentaires sur les données d'une instance de problème sont fournies à la section suivante.

3.3 Description d'instance théorique de problème de "car-sequencing"

Une instance de problème de "car-sequencing" est décrite par un ensemble de voitures à produire qui peuvent être, pour faciliter le traitement, groupées en catégories de voitures semblables. À cet ensemble, s'ajoute les contraintes de capacité du problème ainsi que la demande pour chacune des catégories de véhicule. Un exemple d'instance de problème tiré de Smith [1996] est présenté au Tableau 3.1. Une matrice d'éléments binaires de la taille du nombre de catégories multiplié par le nombre d'options précise quelles sont les options présentes dans chacune des catégories de véhicules. Par exemple,

les voitures de la catégorie 1 possèdent l'option 2 tandis que celles de la catégorie 2 possèdent les options 1, 3 et 5.

		Catégories											
Option	contrainte	1	2	3	4	5	6	7	8	9	10	11	12
1	1/2		•	•				•	•	•			•
2	2/3	•		•	•	•		•				•	•
3	1/3		•					•		•	•	•	
4	2/5				•		•		•				•
5	1/5		•			•							
	nb de voitures	3	1	2	4	3	3	2	1	1	2	2	1

Tableau 3.1 - Instance théorique de problème de "car-sequencing"

Par exemple, certaines voitures nécessitent l'installation de toit ouvrant, d'air climatisé ou de lecteur de disques compacts ou autres. Ces véhicules doivent par conséquent être dispersés dans la séquence de production de façon à lisser la charge de travail. Des contraintes de capacité précisent alors le nombre maximum de voitures q possédant l'option i qui peuvent être produites sur une sous-séquence de taille p. Ces contraintes de capacité sont généralement exprimées par le ratio $C_i = q_i / p_i$ pour chacune des options i du problème. Notons également que la dernière ligne de ce tableau indique la demande pour chacune des catégories de véhicules. Par exemple, la catégorie 1 a une demande de 3 véhicules et la catégorie 2 une demande de 1 véhicule.

Le taux d'utilisation d'une option, représenté par l'Équation (3.1), sera le ratio entre le nombre de véhicules à produire possédant cette option divisé par le nombre maximum de places dans la séquence pour cette option. On aura donc, pour l'exemple du Tableau 3.1, un taux d'utilisation de 64% pour l'option 1, de 100% pour l'option 2, de 96% pour l'option 3, de 90% pour l'option 4 et de 80% pour l'option 5. On calcule, par la suite, la moyenne des taux d'utilisation des options pour obtenir un indicateur du niveau de

difficulté de l'instance. Toutefois, un haut taux d'utilisation ne garantit pas un effort de résolution plus élevé que si le taux d'utilisation est faible. À l'inverse, un faible taux d'utilisation ne garantira pas un faible effort de résolution. Les résultats présentés plus loin dans le présent travail illustrent ce phénomène de façon plus claire.

$$TauxUtilisation_i = (DemandeOpt_i) \ / \ (nbVehicule \times q_i / p_i) \quad (3.1)$$

Un premier groupe d'instances de problèmes de "car-sequencing" de 200 voitures fournies par la librairie CSPlib[3] sont groupées par taux d'utilisation moyen variant entre 60% à 90%. Pour toutes ces instances, il a été prouvé qu'il existait une séquence ne violant aucune contrainte de capacité. On dit alors que ces problèmes sont satisfiables. Un autre groupe de 9 problèmes de 100 voitures à ordonnancer, comportant des problèmes insatisfiables et des problèmes plus difficiles, sont aussi suggérés dans cette même librairie. Les résultats obtenus à ce jour pour ce groupe de problèmes sont présentés dans le Tableau 3.2.

---

[3] http://4c.ucc.ie/~tw/csplib/

Problème	État	Chercheurs	Année
4_72	Satisfiable	Régin, Puget	1997
6_76	Non-satisfiable	Régin, Puget	1997
10_93	Non-satisfiable	Régin, Puget	1997
16_81	Satisfiable	Gottlieb, Putcha, Solnon	2003
19_71	Non-satisfiable	Gent	1998
21_90	Indéterminé		
36_92	Non-satisfiable	Boivin, Gravel, Gagné	2004
41_66	Satisfiable	Gottlieb, Putcha, Solnon	2003
26_82	Satisfiable	Gottlieb, Putcha, Solnon	2003

Tableau 3.2 - Résultats des problèmes de la CSPlib

Les instances de problèmes 4_72, 6_76, 10_93 ont été résolus par Régin et Puget par l'utilisation de méthodes de résolution de CSP [Régin et Puget 1997]. Plus particulièrement, ils ont élaboré un algorithme de filtrage des domaines pour les problèmes comportant des contraintes globales appliquées sur une séquence. Ils ont utilisé ILOG Solver 4.0 pour résoudre ces instances de problèmes auquel une heuristique d'ordonnancement de valeurs a été ajoutée.

Les instances de problèmes 16_81, 41_66 et 26_82 ont été démontrées satisfiables par Gottlieb, Putcha *et al.* [2003] par l'utilisation de la métaheuristique d'optimisation par colonies de fourmis et de recherche locale. Les mêmes auteurs ont aussi déterminé un nombre de violations de contraintes maximal de 2 pour les instances de problèmes 19_71, 21_90 et 36_92, de 3 pour l'instance 10_93 et de 6 pour l'instance 6_76.

L'instance de problème 19_71 fut démontrée non-satisfiable par Gent [1998] sans l'utilisation d'outils informatiques. L'auteur démontra que, étant donné la composition des

catégories de véhicules et les contraintes de capacité associées, il est impossible de solutionner cette instance.

Les algorithmes développés dans le présent travail ont été testés sur les instances de problèmes satisfiables ainsi que sur les problèmes plus difficiles de cette librairie. Les travaux effectués ont permis de déterminer l'état du problème 36_92. Ce problème a été trouvé non-satisfiable par ILOG Solver 6.0 avec l'utilisation de la même heuristique d'ordonnancement de valeurs que Régin et Puget [1997].

3.4 Formulations du problème de "car-sequencing" en CSP

Pour être solvable par les méthodes de résolution de CSP, un problème de "car-sequencing" doit comporter un ensemble de variables X, un ensemble de domaines D et un ensemble de contraintes C. Conformément à Smith [1996], une première formulation possible est d'associer à chaque variable X du problème de CSP une position dans la séquence S. Chaque variable X a ainsi un domaine D_x correspondant à chacune des catégories de véhicules. De plus, pour chaque contrainte de capacité d'option q_i / p_i, aucune partition de la séquence S de taille p_i ne doit comporter plus de q_i voitures possédant l'option i. Enfin, la production totale pour chacune des catégories de véhicules ne doit ni dépasser, ni être inférieure à la demande.

Une autre formulation possible consiste à associer à chacune des variables X du problème une des voitures à produire. Le domaine D du problème est alors toutes les positions présentes dans la séquence de production S. On conservera les mêmes contraintes qu'avec la première formulation. Le désavantage de cette formulation est qu'elle engendre

de nombreuses solutions équivalentes. Par exemple, si la voiture 5 possède l'option 2 et 3 et que la voiture 8 possède l'option 2 et 3, l'affectation de la voiture 5 à la position 20 et de la voiture 8 à la position 32 sera équivalente à l'affectation de la voiture 5 en position 32 et à l'affectation de la voiture 8 en position 20. Cette génération de solutions équivalentes fait en sorte d'évaluer inutilement certaines solutions et augmente la taille de l'espace de solutions à évaluer.

Le choix de la formulation à utiliser dépend de la nature même du problème à traiter. En effet, la première formulation engendre M^N solutions possibles où M est le nombre de catégories possibles et N le nombre de variables du problème. La seconde formulation, quant à elle, engendre N^N solutions possibles. Notons que, pour le problème de "car-sequencing", on a $M <= N$ c'est-à-dire que le nombre de catégories possibles ne peut excéder le nombre de véhicules à produire. C'est pourquoi, plus le nombre de catégories de véhicules à produire est petit et plus la première formulation surpasse la deuxième en terme du nombre de solutions à évaluer. Le présent travail utilise donc la première formulation du problème de "car-sequencing".

Tel que suggéré par Dincbas, Simonis *et al.* [1988], on doit ajouter certaines contraintes implicites qui permettent la détection de situations d'inconsistance au plus tôt lors de la résolution. En effet, les contraintes de capacité impliquent que, pour une contrainte q_i / p_i, au maximum q_i positions de la séquence possèdent l'option i sur toutes les séquences de taille p_i et au minimum aucune position de la séquence ne peut posséder l'option i. Cette valeur peut, dans certain cas, être fausse.

Pour l'instance de problème du Tableau 3.1 et la sous-séquence d'instanciation du Tableau 3.3, une situation d'inconsistance doit être détectée et ne le sera que par l'utilisation de contraintes implicites modifiant le nombre minimal de voitures devant posséder l'option i sur une séquence de taille q_i. En effet, quoique aucune contrainte de capacité ne soit violée, il est impossible que la demande pour l'option 2 soit remplie. L'insertion d'un véhicule de catégorie 8 possédant les options 1 et 4 en position 1 fait en sorte de créer une situation d'inconsistance pour l'option 2 du problème. La contrainte de capacité de l'option 2 étant de 2/3, le nombre de véhicules produits possédant cette option étant 0 et le nombre de positions de la séquence non-affectées étant 24, il ne sera pas possible de produire les 17 véhicules restants possédant l'option 2 en ne violant aucune des contraintes de capacité. Les positions telle qu'il sera possible de produire au plus tôt l'option 2 dans la séquence sont les positions ombragées du Tableau 3.3.

Séquence	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*
Catégories	8																								
Option																									
1, 1/2	•																								
2, 2/3																									
3, 1/3																									
4, 2/5	•																								
5, 1/5																									

Tableau 3.3 - Situation d'inconsistance détectée par les contraintes implicites

Ces contraintes implicites sont décrites à la Figure 3.1. L'algorithme utilise XN pour représenter l'ensemble des positions de la séquence non-affectées, $prod\_possible_i$, pour établir le nombre de fois qu'une option i peut être affectée à une variable $X_j \in XN$ et DOR_i

pour représenter la demande restante pour une option *i*. L'algorithme stocke dans $prod\_possible_i$, pour chacune des options *i* du problème, le nombre de positions de la séquence non-affectées telles que les catégories de véhicules possibles pour celles-ci comprennent cette option. Ce nombre est ensuite multiplié par le ratio q_i / p_i et indique le nombre réel de positions de la séquence non-affectées pouvant être affectées à l'option *i*. On compare ensuite ce nombre à la demande restante pour l'option *i*, soit DOR_i. Si la production possible est inférieure à la demande restante, on détectera une situation d'inconsistance. L'algorithme détecte ainsi les inconsistances beaucoup plus tôt que par la seule utilisation des contraintes de capacité.

```
Procédure VérifierContraintesImp
i ← 0
Consistant ← vrai
TANT QUE Consistant ET i < nb_options FAIRE
      prod_possible_i ← 0
      POUR CHAQUE variable x ∈ XN FAIRE
            SI i ∈ D_x ALORS
                  incrémenter (prod_possible_i)
      SI prod_possible_i * q_i/p_i < DOR_i ALORS
            Consistant ← faux
      incrémenter ( i )
RETOURNER Consistant
```

Figure 3.1 - Algorithme de vérification des contraintes implicites du problème

3.5 Graphe de contraintes du problème de "car-sequencing"

Un CSP peut également être illustré sous la forme d'un graphe de contraintes où les variables sont les nœuds et les contraintes les arêtes. Étant donné que le problème de "car-sequencing" est un problème à contraintes d'arité supérieure à deux, on utilisera un hypergraphe de contraintes où les relations seront illustrées par des quadrilatères et les variables par des cercles. Un graphe de contraintes représentant une instance de problème

de "car-sequencing" comportant 10 véhicules à produire et 2 contraintes de capacité est illustré à la Figure 3.2. On peut voir que la portée d'une contrainte de capacité $C_i = q_i / p_i$ est égale à p_i et ce, pour la totalité de la séquence.

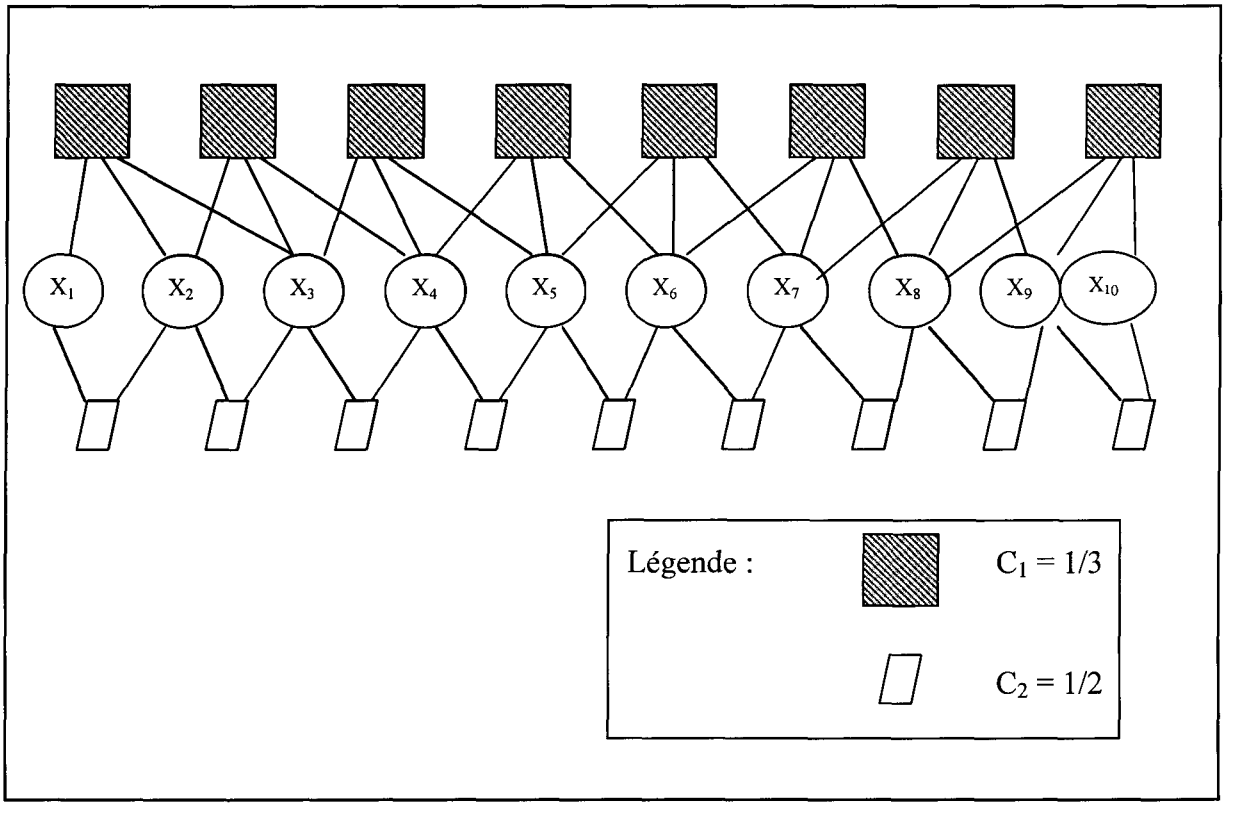

Figure 3.2 – Illustration d'un graphe de contraintes

On parle aussi de contraintes de capacité globale. En effet, la résolution du problème revient à la résolution d'un même problème de taille p_i à plusieurs reprises. Cependant, cette résolution se fait selon l'état de la séquence de production à un temps t et signifie que les domaines des variables ne sont pas toujours les mêmes étant donné que les catégories de véhicules dont la demande a été remplie doivent être retirées des catégories de véhicules possibles pour les positions de la séquence non-instanciées.

3.6 *Résolution du problème de "car-sequencing" en CSP*

La résolution du problème de "car-sequencing" consiste à affecter une catégorie de véhicule $v \in D_x$ à toutes les positions de la séquence représentées par l'ensemble X. La séquence de production doit respecter toutes les contraintes de capacité du problème et remplir la demande de chacune des catégories de véhicule. Par exemple, pour l'instance de problème présentée au Tableau 3.1, une solution valide est décrite au Tableau 3.4.

Catégories / *option*	7	4	8	5	3	10	12	4	2	1	3	10	4	4	9	5	1	6	7	1	6	11	5	6	11
1, 1/2	•		•		•		•		•		•				•				•						
2, 2/3	•	•		•	•		•	•		•	•		•	•		•	•		•	•		•	•		•
3, 1/3	•					•			•			•			•				•			•			•
4, 2/5		•	•				•	•					•	•				•			•			•	
5, 1/5				•					•							•							•		

Tableau 3.4 - Exemple de solution

L'espace de solutions possibles d'un problème de CSP peut être représenté sous la forme d'un arbre de résolution où chacun des nœuds représente l'affectation d'une valeur $v \in D_k$ à une variable X_k et où chacun des niveaux de l'arbre représente une variable du problème. La résolution d'un problème de CSP consiste alors à parcourir en profondeur cet arbre de recherche. L'instanciation d'une feuille de l'arbre représente une solution du problème. La Figure 3.3 illustre un arbre de résolution pour un problème de trois variables de domaine *{1, 2, 3}*.

Lors des situations d'inconsistance, l'algorithme de BT retourne vers la variable précédant la variable courante dans la séquence soit le père du nœud courant de l'arbre de

résolution et modifie la valeur prise par celle-ci. Un problème sera déclaré insatisfiable si la résolution effectue un retour en arrière vers la racine de l'arbre.

Certains logiciels existent sur le marché pour la résolution de problèmes de satisfaction de contraintes. ILOG Solver[4] est un de ces logiciels intégrant les procédures de recherche dans l'arbre de résolution. Les résultats obtenus dans le présent travail pour le problème de "car-sequencing" seront comparés aux résultats obtenus par cet outil commercial de résolution de problèmes de CSP.

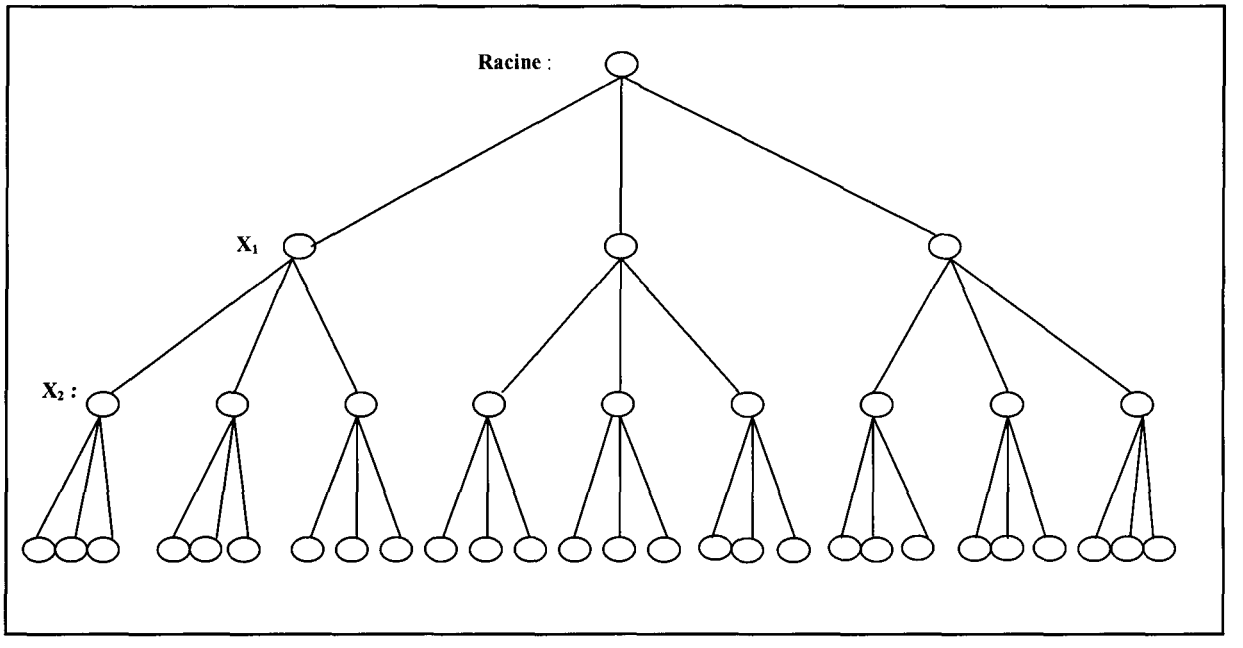

Figure 3.3 - Arbre de recherche

La formulation du problème de "car-sequencing" sous la forme d'un CSP permet donc l'utilisation des algorithmes décrits au chapitre précédent pour solutionner ce problème en particulier. Un algorithme de résolution du problème basé sur les principes de

[4] http://www.ilog.com/products/solver/

filtrage de domaine a donc été développé dans le présent projet. Les résultats présentés permettront d'analyser le comportement des algorithmes de résolution de CSP pour le problème de "car-sequencing" et de préciser l'impact de l'ajout d'éléments tel l'ordonnancement de variables et de valeurs et l'utilisation de liste de nogoods à l'algorithme de forward-checking.

3.6.1 Résolution du problème par l'algorithme de forward-checking

Tel que mentionné par Bessière, Meseguer *et al.* [1999], la résolution du problème de « car-sequencing » est favorisée par l'utilisation d'une méthode de filtrage de domaines simple. En effet, la n-arité des contraintes de capacité fait en sorte qu'une recherche de type MAC (Maintient d'Arc Consistance), utilisant les méthodes de propagation de contraintes, alourdit fortement l'effort de résolution. C'est pourquoi, l'algorithme de FC est utilisé pour la résolution du problème de « car-sequencing ».

Le FC, utilisé pour faciliter la recherche de solutions pour le problème de "car-sequencing", consiste à éliminer des domaines des positions de la séquence les catégories de véhicules rendues inconsistantes par l'état courant de la séquence de production. L'ajout des procédures de filtrage de domaines à un algorithme de BT augmente le nombre d'opérations nécessaires à l'affectation d'une valeur à une variable. Cependant, le but recherché est de prendre de meilleures décisions et ainsi diminuer l'effort global nécessaire à la résolution. C'est pourquoi, un algorithme de filtrage spécifique au problème de "car-sequencing" a été développé.

L'algorithme de FC est décrit à la Figure 3.4. On peut y voir l'appel aux procédures de *Descente* dans l'arbre de résolution et de *Remontée* lors des situations d'inconsistance. La *Descente* consiste à choisir la prochaine catégorie de véhicule à tester et l'appel du filtrage des domaines permet de vérifier que cette affection ne vide pas le contenu d'une des positions de la séquence. La *Remontée* consiste en l'annulation de la plus récente instanciation et à la restauration de son domaine. C'est pourquoi, avant chaque descente dans l'arbre, une sauvegarde de l'état de la résolution doit être effectuée.

```
Procédure FC
Entrée : P, un problème de "car-sequencing"
Sortie : Solution ou information à l'effet que le graphe est inconsistant
Initialisation :
      POUR CHAQUE option i de P FAIRE
            SI DemandeTotal > EffectifPossible ALORS
                  Afficher « aucune solution »
            SINON Aller à Traitement

Traitement :
      TANT QUE etat = 0 FAIRE
            SI consistant Alors
                  consistant = Descente()
            SINON
                  SI positionCourante > profmax ALORS
                        profmax = positionCourante
                        AFFICHER «Sous-Solution »
                  consistant = Remontée()
            SI positionCourante = nb_vehicule ALORS
                  AFFICHER « Solution »
                  etat = 1
            SINON Si positionCourante < 0 ALORS
                  AFFICHER « aucune solution »
```

Figure 3.4 - Algorithme de FC pour le « car-sequencing »

Le filtrage utilisé pour le problème de "car-sequencing" est un filtrage à trois phases. Une première phase consiste à supprimer du domaine des positions de la séquence non-affectées les catégories de véhicules dont la demande a été remplie et de vérifier si cette opération vide le contenu d'un domaine. Une seconde phase consiste à vérifier si, étant

donné l'affectation courante, une des contraintes implicites telle que décrites dans la section 3.4 est violée. Finalement, une troisième phase consiste à supprimer des domaines des positions de la séquence les catégories de véhicule violant une des contraintes de capacité. La procédure de filtrage est décrite à la Figure 3.5.

```
Procédure Filtrer (choix, positionCourante)
Consistant ← vrai

// Phase 1
EffectifCategorie[choix ] ← EffectifCategorie[choix]-1
SI EffectifCategorie[choix]-1 = Nul ALORS
      POUR CHAQUE Xk tel que Xk est non-affectée FAIRE
            Dk ← Dk - { Xk }
            SI Dk = nul ALORS
                  Consistant ← faux
// Phase 2
SI Consistant ALORS
      Consistant ← FiltrerContraintesImplicites()
// Phase 3
SI Consistant ALORS
      Consistant ← FilterRelation(PositionCourante, choix)

Retourner Consistant
```

Figure 3.5 - Algorithme de filtrage pour le « car-sequencing »

Pour faciliter le filtrage, le domaine d'une variable X_k est illustré sous la forme d'un tableau d'éléments booléens T_{ki} indiquant si l'option i peut être affectée à X_k. Le domaine de la variable comprend l'ensemble des catégories de véhicules possibles mais, lors de l'affectation d'une catégorie à un véhicule de la séquence de production, on affecte cette catégorie sous la forme de son ensemble d'options.

Le filtrage induit par les relations du problème est décrit à la Figure 3.6. L'algorithme débute par l'initialisation des variables nécessaires au traitement. La variable *Début* indique la première position de la séquence concernée par l'affectation courante, *Fin* est utilisé pour illustrer la dernière position concernée par le filtrage, *Saturation* est un

compteur permettant de vérifier si l'affectation courante dépasse la capacité d'une option et *Première* indique la première position de la sous-séquence utilisant l'option sur laquelle on effectue le filtrage.

Sommairement, l'algorithme débute par effectuer une boucle sur chacune des options du problème. On vérifie alors la saturation de l'option i relative à la position courante en utilisant les p_i-1 positions précédant celle-ci. Dans le cas où l'affectation courante sature l'option concernée, on filtrera les domaines des variables futures.

Le filtrage des domaines est illustré à la Figure 3.7 avec l'instance de problème du Tableau 3.1. Avant l'affectation d'une catégorie de véhicule $v \in D_k$ à une position de la séquence X_k, un filtrage est effectué et les options dont la capacité est saturée sont retirées des domaines. C'est ainsi qu'il est possible de vérifier si une affectation crée une situation d'inconsistance avant de prendre une décision définitive.

```
Procédure FiltrerRelation (PositionCourante, v)

      POUR CHAQUE Option i ∈ v FAIRE
            Début ← max (1, positionCourante-p_i+1)
            Fin ← positionCourante
            Trouve ← faux
            Saturation ← Début
            Première ← 0
            q ← q_i
            p ← p_i
            TANT QUE Saturation < Fin ET Trouve = faux FAIRE
                  SI X_Saturation possède l'option i ALORS
                        p ← p - 1
                        Première ← Saturation
                        Trouve ←    Vrai
                  Saturation ← Saturation + 1

            TANT QUE Saturation < Fin ET q > 0 FAIRE
                  SI X_Saturation possède l'option i ALORS
                        q ← q - 1
                  Saturation ← Saturation + 1

            SI q = 0 ALORS
                  Début ← positionCourante
                  Fin ← min (Première+p, nbVehicule)
                  Appeler SauvegarderDomaines

                  POUR j ← 0 à nb_categorie FAIRE
                        POUR k ← 0 à Fin FAIRE
                              SI Catégorie_j possède l'option i ALORS
                                    Supprimer j de D_k
      Consistant ← Vrai
      pos ← positionCourante
      TANT QUE Consistant = Vrai ET i < Fin FAIRE
            SI D_pos = Nul ALORS
                  Consistant ← Faux
            pos ← pos + 1
      Retourner Consistant
```

Figure 3.6 - Algorithme de filtrage des relations du problème de "car-sequencing"

L'étape 1 représente l'état initial des domaines. Les domaines sont initialisés à 1 pour toutes les options du problème puisqu'il est possible de produire un véhicule possédant toutes les options du problème. À l'étape 2, l'affectation de la catégorie de véhicule 7 à la position de la séquence X_1 fera en sorte que l'option 1 sera retirée du domaine de la position de la séquence X_2 puisque le ratio de cette option est de ratio 1/2.

Par contre, même si l'option 2 est produite, cette option ne sera pas retirée du domaine des p_2 (3) prochaines positions de la séquence puisque q_2 est égal à 2. Cette option n'est donc pas saturée même si elle est présente dans le véhicule produit. L'option 3, de ratio 1/3, est présente dans le véhicule produit ce qui élimine par le fait même celle-ci du domaine des positions X_2 et X_3. À l'étape 3, le véhicule produit de catégorie 4 comprend les options 2 et 4 ce qui implique un filtrage de l'option 2 sur la position X_3 puisque cette option, de ratio 2/3, est saturée par l'instanciation de X_1 et de X_2. Cependant, l'option 4 n'est pas retirée puisque les instanciations ne saturent pas le ratio de 2/5 associé à cette option. Finalement, à l'étape 4, l'affectation d'un véhicule de catégorie 8 à la position X_3 produit le filtrage de l'option 1 sur la position X_4. De plus, l'option 4, de ratio 2/5, est saturée par l'instanciation de X_2 et de X_3 ce qui produit un filtrage des domaines des positions X_4, X_5 et X_6.

L'utilisation des contraintes de capacité permet ainsi d'éviter de parcourir certains chemins inconsistants dans l'arbre de résolution. C'est ainsi que le FC diminuera l'effort de calcul nécessité lors de la résolution d'instances de problèmes de "car-sequencing".

Étape 1	État initial				
Option \ Variable	X_1	X_2	X_3	X_4	X_5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
5	1	1	1	1	1

Étape2	Affectation : $X_1 \leftarrow 7$				
Option \ Variable	X_1	X_2	X_3	X_4	X_5
1	1	0	1	1	1
2	1	1	1	1	1
3	1	0	0	1	1
4	0	1	1	1	1
5	0	1	1	1	1

Étape 3	Affectation : $X_2 \leftarrow 4$				
Option \ Variable	X_1	X_2	X_3	X_4	X_5
1	1	0	1	1	1
2	1	1	0	1	1
3	1	0	0	1	1
4	0	1	1	1	1
5	0	0	1	1	1

Étape 4	Affectation : $X_3 \leftarrow 8$				
Option \ Variable	X_1	X_2	X_3	X_4	X_5
1	1	0	1	0	1
2	1	1	0	1	1
3	1	0	0	1	1
4	0	1	1	0	0
5	0	0	0	1	1

Figure 3.7 - État des domaines suite au filtrage

La sous-séquence sur laquelle on doit appliquer le filtrage est décrite à la Figure 3.8. Le filtrage est décomposé en deux étapes distinctes, soit le regard en arrière et le regard en avant. On peut voir que la taille de la sous-séquence de filtrage est dépendante du p_i de l'option concernée.

Option 1 - 1/2

Étape 1						Deb : X_2	Fin : X_3
Étape 2						Deb : X_3	Fin : X_4
Variable	X_1	X_2	$\mathbf{X_3}$	X_4	X_5		

Option 4 - 2/5

Étape 1						Deb : X_1	Fin : X_3
Étape 2						Deb : X_2	Fin : X_6
Variable	X_1	X_2	$\mathbf{X_3}$	X_4	X_5		

Figure 3.8 - Portée du filtrage

L'étape du regard en arrière permet de déterminer si, pour chacune des options du problème, une des contraintes de capacité est saturée. Dans l'affirmative, l'étape du regard en avant permet d'éliminer du domaine des positions de la séquence futures les catégories

de véhicule rendues inconsistants. Si la contrainte de capacité n'est pas saturée, l'étape de regard en avant est évitée. Dans ce cas précis, on se contente de vérifier si on viole une contrainte touchant à une position de la séquence passée sans filtrer le domaine des positions futures. Il est à noter que l'étape de regard en arrière n'est pas utilisée dans le cas de contraintes de capacité telles que q_i est inférieur à 2. Si le filtrage des options n'a pas préalablement éliminé cette option du domaine, l'option n'est donc pas saturée par une position précédente dans la séquence. Seule l'étape de regard en avant sera effectuée et cette option est ainsi retirée des p_i prochaines positions de la séquence.

L'exemple de la Figure 3.8 simule la production d'un véhicule possédant l'option 1 et 4 à la position 3 de la séquence. Premièrement, l'option 1 implique un regard arrière permettant de vérifier si cette option n'a pas déjà été affectée à la position 2 de la séquence. Par la suite, la phase de regard en avant permet de supprimer l'option 1 du domaine de la position X_4 puisque cette valeur est rendue inconsistante. La figure présente aussi le filtrage associé à l'option 4 de ratio 2/5. On peut voir que l'affectation de la variable X_3 produit un regard en arrière jusqu'à la position X_1 puisque le p_i associé est 5. De plus, dans le cas d'une saturation de la capacité de la contrainte, on effectuera un regard en avant permettant de supprimer cette valeur du domaine des 5 prochaines positions de la séquence du problème, soit X_4 à X_8. Dans l'exemple présenté, le filtrage est restreint à la position 5 de la séquence puisque c'est la dernière position de la séquence.

L'algorithme de FC a été testé sur les problèmes de la CSPlib. Le Tableau 3.5 présente les résultats obtenus par l'algorithme pour les problèmes satisfiables de la librairie. Ceux-ci sont groupés par taux d'utilisation moyen des options et on compte 10 instances de

problèmes pour chacun d'eux. Les résultats présentés correspondent aux résultats moyens obtenus pour chacun des groupes de 10 instances. Le Tableau 3.6 présente, pour sa part, les résultats obtenus pour la résolution des 9 problèmes plus difficiles de la librairie. Dans les Tableaux 3.5 et 3.6, la colonne *Total temps* indique le temps nécessaire pour résoudre le problème où, dans le cas d'un problème non résolu, le temps maximum passé à chercher une solution au problème. La colonne *Temps prof. max* indique le temps nécessaire pour atteindre la profondeur maximale dans l'arbre de résolution, c'est-à-dire le temps nécessaire pour affecter le plus grand nombre de variables du problème. La colonne *Profondeur* indique le nombre de variables qu'il a été possible d'affecter lors de l'exécution du FC. *BT* représente le nombre de retours en arrière nécessaires pour atteindre la profondeur maximale de la recherche dans l'arbre de résolution. Les *Nœuds visités* sont le nombre de nœuds de l'arbre de résolution visités pour atteindre la profondeur maximale. Le *Filtrage* représente le nombre d'appels à la procédure de filtrage des domaines. Finalement, *réussites* indique le nombre de problèmes résolus par l'algorithme. Pour les résultats du Tableau 3.5, on parlera de moyenne de *Temps, Profondeur, BT, Nœud visités, Filtrage* et *réussites* pour la résolution des 10 problèmes.

L'algorithme a été restreint à un temps de 900 secondes soit 15 minutes. Il a été exécuté sur une machine dotée d'un processeur Pentium 3 à 1 Ghz et de 396 Mo de mémoire vive. L'algorithme a été développé en langage *C++* en utilisant le compilateur *g++* sous *linux*.

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	720	144	193	2 491 235	2 491 427	5 842 187	2
65	720	83	190	1 127 821	1 128 010	3 578 388	2
70	750	71	156	1 106 329	1 106 485	2 848 960	2
75	750	204	155	2 720 274	2 720 430	7 080 188	2
80	900	275	185	5 014 713	5 014 898	12 335 640	0
85	900	277	177	3 418 715	3 418 894	9 231 903	0
90	810	187	172	1 544 589	1 544 760	5 295 757	1
Moyenne :	793	177	175	2 489 096	2 489 272	6 601 860	

Tableau 3.5 - Résultats FC, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussite
16_81	900	240	75	1 133 210	1 133 280	5 750 550	Non
26_82	900	355	70	3 077 980	3 078 050	13 892 700	Non
41_66	900	2	75	32 310	32 385	85 193	Non
4_72	900	217	72	1 634 390	1 634 470	6 854 680	Non
10_93	900	160	86	1 525 900	1 525 980	5 920 110	Non
19_71	900	32	72	259 708	259 780	1 054 220	Non
21_90	900	785	81	5 937 200	5 937 280	29 565 100	Non
36_92	900	191	81	1 026 430	1 026 520	5 865 390	Non
6_76	900	483	77	6 011 550	6 011 620	16 984 900	Non
Moyenne :	900	274	77	2 293 186	2 293 263	9 552 538	

Tableau 3.6 - Résultats FC, deuxième groupe

Notons, premièrement, que l'algorithme atteint sa profondeur maximale rapidement. En effet, pour les problèmes du Tableau 3.5, les profondeurs maximales sont atteintes en moyenne en moins du cinquième du temps alloué. Cependant, après avoir atteint cette profondeur dans l'arbre de résolution, l'algorithme stagne et ne parvient pas à descendre plus bas dans l'arbre. Le reste du temps alloué est donc passé à effectuer des retours en arrière. Deuxièmement, l'algorithme de FC seul a permis de ne résoudre aucun des problèmes non satisfiables ou difficiles présentés au Tableau 3.6. De plus, pour les problèmes plus faciles présentés au Tableau 3.5, l'algorithme a résolu 9 problèmes sur une

possibilité de 70 ce qui indique qu'il n'est pas très performant tant sur les problèmes difficiles que sur les problèmes faciles.

À des fins de comparaison, les deux groupes de problèmes ont été testés sur ILOG Solver. Les Tableaux 3.7 et 3.8 présentent les résultats obtenus. Solver a été exécuté sur une machine dotée d'un processeur Itanium 64 bits à 1,4 Ghz ainsi que de 4 Go de mémoire vive. Solver utilise la contrainte IlcSequence, une contrainte développée pour les problèmes comportant des contraintes globales appliquées sur une séquence. Notons, premièrement, que l'arbre de résolution traité par Solver n'est pas le même que celui utilisé lors du développement du FC puisque le nombre de variables, soit *nb_variables*, du problème n'est pas le nombre de véhicules à ordonnancer. Solver effectue aussi un moins grand nombre de retours en arrière que le FC développé. Le nombre de points de choix, soit *nb_choice_point*, est aussi de beaucoup supérieur pour le FC que pour le calcul effectué par Solver. Pour ce qui est du nombre de réussites, pour les problèmes du premier groupe de la CSPlib, Solver est parvenu à résoudre 8 problèmes comparativement à 9 pour le FC. Cependant, Solver est parvenu à résoudre un problème difficile ainsi qu'un problème non satisfiable. Ce phénomène peut être dû au filtrage utilisé par Solver qui serait plus efficace que celui du FC développé. Cette hypothèse est appuyée par le fait que la majorité des instances de problèmes résolues par Solver ont des taux d'utilisation élevés. L'arbre de résolution traité par Solver produit donc un filtrage plus efficace que celui offert par le schéma de résolution décrit pour le FC. Cependant, au niveau du nombre total d'instances résolues les deux méthodes ont obtenu des résultats équivalents soit 9 pour le FC et 10 pour ILOG Solver.

Le FC n'ayant pas permis de résoudre un grand nombre de problèmes, il est donc nécessaire d'améliorer la recherche de solutions en jumelant le filtrage des domaines à d'autres éléments favorisant la construction de solutions au problème de "car-sequencing". C'est pourquoi, les sections suivantes décrivent les travaux effectués en ce qui a trait à l'ordonnancement de variables et de valeurs ainsi qu'à l'utilisation de listes de nogoods lors de la descente dans l'arbre de résolution.

Problème	Total temps	BT	nb_choice_pt	nb_variables	réussites
60	811	39 338	39 509	2 049	1
65	900	37 374	37 554	1 224	0
70	900	39 023	39 224	1 225	0
75	900	26 627	26 841	1 226	0
80	900	22 813	23 021	1 226	0
85	633	17 985	18 236	3 708	3
90	544	15 808	16 091	4 535	4
Moyenne :	798	28 424	28 639	2 170	

Tableau 3.7 - Résultats ILOG, premier groupe

Problème	Total temps	BT	nb_choice_pt	nb_variables	réussite
16_81	900	87 485	87 614	627	Non
26_82	900	120 382	120 477	625	Non
41_66	900	348 574	348 661	620	Non
4_72	93	15 177	15 274	4780	Oui
10_93	1	41	40	626	Oui
19_71	900	74 420	74 537	624	Non
21_90	900	101 197	101 251	624	Non
36_92	900	138 806	138 879	623	Non
6_76	900	83 066	83 146	623	Non
Moyenne :	710	107 683	107 764	1 086	

Tableau 3.8 - Résultats ILOG, deuxième groupe

3.6.2 Ordonnancement de variables et de valeurs

Lors de la procédure de descente de l'algorithme de FC, deux choix doivent être fait par l'algorithme. Premièrement, l'algorithme doit déterminer la prochaine variable à

affecter. L'ordre d'affectation des variables peut donc être déterminé en suivant certaines heuristiques. Deuxièmement, le choix de la valeur à affecter à la variable a aussi un impact sur la résolution. À la Section 3.6.2.1, l'heuristique d'ordonnancement de variables *fail-first* sera présentée tandis qu'à la Section 3.6.2.2 , on présentera les heuristiques de choix de valeurs *fail-first* et *succeed-first*.

3.6.2.1 Ordonnancement de variables

Comme suggéré par Haralick et Elliot [1980], une recherche utilisant un principe de filtrage de domaines de type FC est favorisée par l'utilisation d'une heuristique de type *fail-first*. En effet, plus les inconsistances sont détectées à des niveaux supérieurs de l'arbre de résolution et plus l'effort engendré par la remontée est minimal. C'est pourquoi, une heuristique d'ordonnancement de variables de type *fail-first* est appliquée au problème de "car-sequencing" dans le présent travail.

Selon Smith [1996], ce type de schéma, appliqué au problème de "car-sequencing", respecte l'ordre total de l'ensemble des position de la séquence du problème. En effet, la globalité des contraintes fait en sorte que l'affectation d'une catégorie de véhicule $v \in D_1$ à la première position de la séquence du problème, soit X_1, a un impact sur les k positions consécutives à X_1 tel que k est le p_i maximum pour les options présentes dans la catégorie de véhicule v. C'est ainsi que l'ordre d'affectation des positions de la séquence respecte l'ordre des variables dans l'ensemble X, soit $\{1, 2, ... n\}$.

En supposant que la variable choisie en cas de domaines de taille égale est la variable d'indice minimum, la première position de la séquence est toujours choisie en

premier. Le Tableau 3.9 présente, pour l'exemple de la Figure 3.7, les étapes de l'heuristique de choix de variables *fail-first*. L'étape 1 représente l'état initial des domaines, c'est-à-dire la possibilité que les véhicules produits comprennent toutes les options. Suite à l'affectation de la catégorie 7 à la position X_1, les domaines des positions X_1, X_2 et X_3 sont diminués. La prochaine position est donc X_2 puisque la taille de son domaine est la plus petite parmi les domaines des positions de la séquence non-affectées. L'affectation du véhicule de catégorie 4 à la position X_2 diminue le domaine de X_3 ce qui fait en sorte qu'elle sera la prochaine position à instancier. Finalement, l'affection d'un véhicule de catégorie 8 à la position X_3 diminue les domaines de X_4 et X_5. La position de la séquence X_4 ayant un domaine de 3 et X_5 de 4, X_4 est la prochaine position de la séquence à affecter puisque de domaine minimal.

On peut voir le respect de l'ordre total d'instanciation des position de la séquence. Ce phénomène est dû au fait qu'une instanciation n'a d'effet que sur les positions suivant la position courante de la séquence. Le nombre de positions de la séquence affectées étant dépendant du p_i de l'option concernée.

	Taille des domaines					
Étapes	**X_1**	**X_2**	**X_3**	**X_4**	**X_5**	**Choix**
1	5	5	5	5	5	**X_1**
2	3	3	4	5	5	**X_2**
3	3	2	3	5	5	**X_3**
4	3	2	2	3	4	**X_4**

Tableau 3.9 - Ordre d'instanciation des variables

C'est pourquoi, pour respecter le schéma d'ordonnancement de variables *fail-first* suggéré pour les algorithmes basés sur le filtrage des domaines, l'ordre d'instanciation des positions de la séquence respectera l'ordre des variables dans l'ensemble X soit *{1, 2, 3, ...*

n}. Les algorithmes présentés dans les sections suivantes considéreront tous le schéma d'ordonnancement de variables *fail-first*.

3.6.2.2 Ordonnancement de valeurs

Le choix de la prochaine valeur à affecter à la variable courante a également un impact majeur sur l'effort de résolution. Les résultats présentés dans cette section illustrent ce phénomène de manière non équivoque. Les heuristiques d'ordonnancement de valeurs *fail-first*, au nombre de 4, et *succeed-first*, au nombre de 3, ont été testées pour le problème de "car-sequencing" ainsi que les méthodes statiques et dynamiques d'ordonnancement de valeurs. De plus, une heuristique s'orientant uniquement sur le lissage de la production de chacune des options et une heuristique d'ordonnancement de valeurs aléatoires ont été développées.

Pour le problème de "car-sequencing", une heuristique de type *fail-first* vise à sélectionner en premier les catégories de véhicules les plus contraintes. Cela permet de supprimer rapidement un plus grand nombre de nœuds inconsistants dans l'arbre de résolution. Ces suppressions éliminent par le fait même des sous-arbres de l'arbre de résolution. La Figure 3.9 illustre l'effet du filtrage suite à l'affectation de la valeur 1 à la position de la séquence X_1. On suppose ici une contrainte de capacité de ratio 1/3 appliquée à la première et à la troisième catégorie de véhicule. On constate que plus les suppressions ont lieu rapidement lors de la résolution et plus les sous-arbres supprimés sont de taille importante. En effet, la suppression d'un nœud implique la suppression de tout le sous-arbre engendré par celui-ci.

Une heuristique de type *succeed-first* vise, quant à elle, à garder le plus possible de catégories de véhicules dans les domaines des positions de la séquence afin d'avoir un choix plus grand lors de l'affectation de celles-ci. Les situations d'inconsistance sont donc souvent détectées à des profondeurs élevées de l'arbre de recherche et implique un effort de traitement considérable afin d'effectuer une remontée dans l'arbre de résolution.

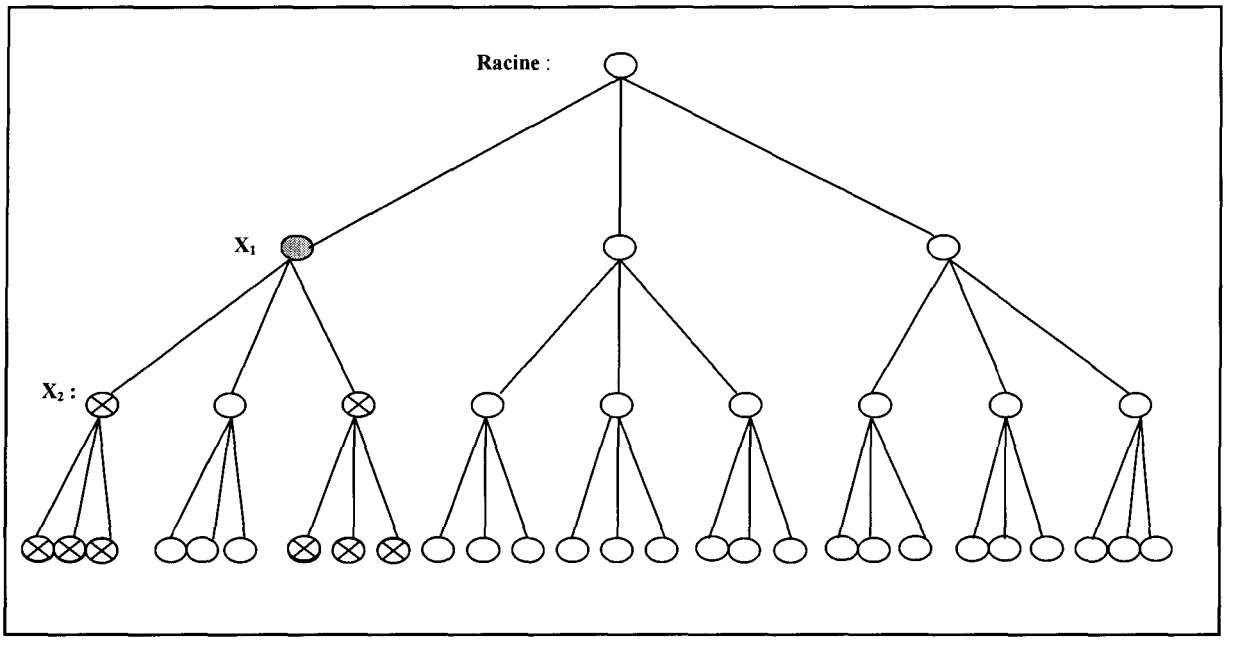

Figure 3.9 - Suppressions de valeurs dans l'arbre de recherche

La problématique soulevée par l'ordonnancement de valeurs lors d'une recherche de solution au problème de "car-sequencing" débute par la définition du principe de difficulté d'une catégorie. Pour ordonner les catégories de véhicules de la plus difficile à la plus facile, il faut déterminer ce qu'est le niveau de difficulté de chacune des catégories.

Une première méthode, permettant de déterminer ce niveau de difficulté, consiste à utiliser la somme des taux d'utilisation des options de cette catégorie (Équation (3.1) page 47). Le niveau de difficulté devient dynamique si on considère le nombre de véhicules

restant à produire pour la catégorie et le nombre de positions restantes à fixer au moment du calcul plutôt que la demande totale de celle-ci. Cette heuristique d'ordonnancement des catégories de véhicules, de type *fail-first*, est appelée *MaxTauxUt* dans le présent travail. Pour l'instance de problème du Tableau 3.1, on aura donc un ordre $o = \{ 5, 4, 7, 2, 11, 6, 3, 10, 12, 9, 1, 8 \}$. Une heuristique d'ordonnancement des catégories de véhicule de type *succeed-first*, nommée *MinTauxUt*, utilisant l'ordre inverse de *MaxTauxUt* a aussi été testée. Ces heuristiques ont comme particularité d'être dynamiques, c'est-à-dire qu'elles sont dépendantes de l'état de la résolution à un temps t. Les Tableaux 3.10, 3.11, 3.12 et 3.13 présentent les résultats pour les heuristiques de choix de catégories de véhicule *MaxTauxUt* et *MinTauxUt* pour les deux types de problèmes de la librairie CSPlib.

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	812	96	174	837 833	1 126 090	2 252 036	1
65	630	283	182	3 241 552	3 241 736	6 077 106	3
70	683	191	183	3 023 547	3 023 730	5 401 816	3
75	720	257	195	4 574 541	4 574 738	9 572 865	2
80	720	351	192	6 486 670	6 486 863	12 644 718	2
85	810	162	190	2 421 364	2 421 553	5 227 242	1
90	900	398	175	4 630 204	4 104 117	10 604 635	0
Moyenne :	754	248	184	3 602 244	3 568 404	7 397 203	

Tableau 3.10 - Résultats heuristique MaxTauxUt, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussite
16_81	900	101	96	1 222 700	1 222 790	4 263 950	Non
26_82	900	95	91	1 234 980	1 235 070	3 760 930	Non
41_66	900	5	94	127 376	127 470	235 476	Non
4_72	900	405	95	7 003 230	7 003 320	18 765 800	Non
10_93	900	66	82	542 046	542 128	2 481 940	Non
19_71	900	754	85	8 249 670	8 249 750	30 968 900	Non
21_90	900	257	88	3 535 290	3 535 380	12 622 000	Non
36_92	900	74	88	930 466	930 554	3 151 120	Non
6_76	900	629	85	10 072 400	10 072 500	25 942 500	Non
Moyenne :	900	265	89	3 657 573	3 657 662	11 354 735	

Tableau 3.11 - Résultats heuristique MaxTauxUt, deuxième groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	900	76	176	988 108	988 284	2 312 876	0
65	900	312	173	3 784 351	3 784 524	8 951 215	0
70	900	174	177	2 215 958	2 216 134	4 971 209	0
75	900	329	172	3 902 743	3 902 918	9 141 275	0
80	900	290	166	2 541 119	2 541 284	6 688 359	0
85	900	410	172	3 266 311	3 266 483	11 034 625	0
90	900	410	155	2 382 503	2 382 658	9 294 625	0
Moyenne :	900	286	170	2 725 870	2 726 041	7 484 883	

Tableau 3.12 - Résultats heuristique MinTauxUt, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussite
16_81	900	608	85	3 487 520	3 487 610	18 048 600	Non
26_82	900	381	84	3 166 920	3 167 000	12 886 400	Non
41_66	900	771	82	8 164 840	8 164 920	27 015 100	Non
4_72	900	707	87	6 429 820	6 429 900	27 421 700	Non
10_93	900	83	67	420 747	420 814	2 337 960	Non
19_71	900	713	88	5 128 050	5 128 140	23 585 000	Non
21_90	900	859	72	4 821 170	4 821 240	20 676 900	Non
36_92	900	353	79	2 263 360	2 263 440	10 933 300	Non
6_76	900	725	81	7 151 740	7 151 820	26 641 700	Non
Moyenne :	900	578	81	4 559 352	4 559 432	18 838 518	

Tableau 3.13 - Résultats heuristique MinTauxUt, deuxième groupe

On constate que l'heuristique *MaxTauxUt* résout un plus grand nombre de problèmes que l'heuristique *MinTauxUt*. Les profondeurs moyennes atteintes (184) par l'heuristique *MaxTauxUt* sur les instances de problèmes du premier groupe sont plus élevées que celles atteintes par l'heuristique *MinTauxUt* (170). De plus, pour ces deux heuristiques d'ordonnancement des catégories de véhicules, le temps nécessaire pour atteindre les profondeurs maximales est peu élevé comparativement au temps alloué à la résolution (15 minutes). La majorité du temps de calcul est donc utilisée à effectuer des remontées dans l'arbre de résolution. Ces remontées se révèlent inutiles puisqu'elles ne permettent pas d'effectuer des descentes significatives dans l'arbre de résolution.

Un second ordonnancement des catégories de véhicules a été testé en utilisant uniquement le nombre d'options présentes dans chacune d'elles comme mesure de difficulté. Cette heuristique, nommée *Option*, définit le niveau de difficulté d'une catégorie comme étant le nombre d'options qui la compose. Cette heuristique a obtenu de bons résultats pour les problèmes de la librairie CSPlib. Cela peut être dû au fait que les instances de problèmes suggérées ont des ratios variant entre 1/5 et 2/3. Pour l'instance de problème du Tableau 3.1, l'ordre généré par une telle heuristique est $o = \{ 2, 7, 12, 3, 4, 5, 8, 9, 11, 1, 6, 10 \}$. Le calcul inverse soit *MinOption* a aussi été développé. Les Tableaux 3.14 et 3.15 présentent les résultats obtenus par l'heuristique *Option*. On retrouve en Annexe 1 les résultats obtenus par l'heuristique *MinOption* qui n'est pas très performante.

On constate, comme pour les heuristiques d'ordonnancement des catégories de véhicules *MaxTauxUt* et *MinTauxUt*, que le schéma d'ordonnancement *fail-first* (*Option)* est supérieur au schéma *succeed-first* (*MinOption)*. En effet, l'heuristique *Option* est

parvenue à résoudre 60 problèmes comparativement à aucun pour *MinOption*. De plus, les profondeurs maximales sont atteintes très rapidement. Pour l'heuristique *Option* et pour les instances de problèmes non-satisfiables, correspondant aux 5 derniers problèmes du Tableau 3.15, les profondeurs maximales sont atteintes en moins de 0 secondes pour trois de ces problèmes. Pour les problèmes du premier groupe, on atteint les profondeurs maximales en moyenne après 55 secondes.

Quoique n'ayant pas permis de résoudre d'instances de problèmes difficiles, l'heuristique *Option* est parvenue à résoudre 52 problèmes faciles de plus que ILOG Solver 6.0 et 51 de plus que le FC sans heuristiques de choix de valeurs. Un bon ordonnancement des catégories de véhicules permet donc de résoudre rapidement les problèmes et nécessite un moins grand nombre de retours en arrière ainsi que d'appels à la procédure de filtrage.

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	96	7	199	190 825	191 023	297 243	9
65	293	101	199	2 530 577	2 530 776	4 216 648	9
70	140	120	200	2 610 829	2 611 029	5 059 408	9
75	183	63	199	1 064 338	1 064 537	2 526 198	8
80	182	24	198	423 514	423 711	830 161	8
85	202	69	199	2 013 643	2 013 842	3 488 983	7
90	0	0	200	504	704	1 034	10
Moyenne :	157	55	199	1 262 033	1 262 232	2 345 668	

Tableau 3.14 - Résultats heuristique Option, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
16_81	900	3	96	55 546	55 642	184 138	Non
26_82	900	93	96	2 352 590	2 352 690	5 345 900	Non
41_66	900	0	98	12 401	12 499	21 643	Non
4_72	900	0	99	18 342	18 441	45 300	Non
10_93	900	461	93	6 891 270	6 891 370	25 580 700	Non
19_71	900	0	91	138	229	496	Non
21_90	900	0	96	2 258	2 354	7 449	Non
36_92	900	0	97	11 889	11 986	27 052	Non
6_76	900	3	89	54 906	54 995	109 182	Non
Moyenne :	900	62	95	1 044 371	1 044 467	3 480 207	

Tableau 3.15 - Résultats heuristique Option, deuxième groupe

Une heuristique nommée *Fréquence* consistant à favoriser le lissage des options sur la séquence a aussi été testée. Le principe utilisé consiste à maintenir à jour, pour chacune des catégories de véhicules, une position de production au plus tôt. Celle-ci correspond à la prochaine position où cette catégorie de voiture peut être insérée dans la séquence de production sans violer une contrainte implicite du problème. L'heuristique favorise ainsi les catégories de véhicules dont la production au plus tôt se rapproche le plus de la position courante. Les tableaux illustrant les résultats obtenus par cette heuristique sont toutefois présentés en Annexe 1 puisqu'elle n'a pas obtenu de bons résultats.

Notons que cette heuristique favorise beaucoup le lissage de la production de chacune des options. Cependant, le calcul utilisé ne permet pas de favoriser les catégories de véhicules les plus contraintes dans le but de supprimer le plus grand nombre de sous-arbres de l'arbre de résolution. C'est pourquoi, cette heuristique ne permet pas de résoudre un très grand nombre d'instances de problème. De plus, les profondeurs atteintes dans l'arbre de résolution ne sont pas non plus très élevées. Le lissage de la production étant

déjà pris en compte par les contraintes implicites du filtrage, cette heuristique favorise uniquement la vérification de celles-ci plus tôt lors du traitement.

Un ordonnancement basé sur l'effectif restant à produire pour une catégorie a été développé. Le but de cette heuristique n'est pas de favoriser les catégories en tenant compte de leur niveau de difficulté. On vise uniquement à garder ou à éliminer le plus de catégories possible des domaines des variables. Pour l'instance de problème du Tableau 3.1, l'ordre généré par une telle heuristique est $o = \{ 4, 1, 5, 6, 3, 7, 10, 11, 2, 8, 9, 12 \}$. L'heuristique *Véhicule*, un schéma de type *succeed-first*, illustre bien l'inefficacité de ce schéma. L'ordre inverse soit *MinVéhicule* obtient un taux d'efficacité supérieur tout en ne parvenant pas à surpasser les résultats obtenus par le FC. Les résultats obtenus par ces deux heuristiques sont présentés en Annexe 1.

Notons, une fois de plus, que le schéma d'ordonnancement de catégories de véhicules *fail-first* surpasse le schéma *succeed-first*. Cependant, l'heuristique *Vehicule* comme *MinVehicule* ne parvient pas à résoudre plus d'instances de problèmes que le FC. Les instances résolues sont, pour la plupart, des instances de bas taux d'utilisation des options.

Une heuristique, nommée *P/Q*, a aussi été testée et utilise la somme des ratios inverses des options présentes dans une catégorie pour en calculer le niveau de difficulté. Celle-ci favorise ainsi les options dont le filtrage relié aux contraintes de capacité a un impact sur le plus grand nombre de positions de la séquence possible. Elle ne prend pas en compte l'effectif restant pour cette catégorie dans le calcul ce qui en fait une heuristique statique. Pour l'instance de problème du Tableau 3.1, l'ordre généré par un telle

heuristique est $o = \{ 2, 5, 7, 12, 9, 8, 11, 4, 3, 10, 6, 1 \}$. Cette heuristique ressemble beaucoup à l'heuristique *Option* mais ajoute la valeur des ratios au calcul. Elle a obtenu des résultats intéressants sur les problèmes satisfiables de la CSPlib. Les Tableaux 3.16 et 3.17 présentent les résultats obtenus par cette heuristique.

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	270	51	192	602 836	603 028	1 361 990	7
65	270	126	175	1 084 160	1 084 288	2 380 999	6
70	360	86	197	1 204 978	1 205 175	2 646 484	6
75	360	116	180	1 559 135	1 715 228	3 507 283	6
80	270	89	199	1 189 068	1 189 268	2 674 110	7
85	90	1	200	12 170	12 369	19 934	9
90	270	159	198	2 796 752	2 796 951	6 181 504	7
Moyenne :	270	90	192	1 207 014	1 229 473	2 681 758	

Tableau 3.16 - Résultats heuristique P/Q, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
16_81	900	708	99	12 119 300	12 119 400	36 394 000	Non
26_82	900	325	95	7 606 890	7 606 980	16 000 700	Non
41_66	900	0	97	13	110	126	Non
4_72	900	1	97	29 770	29 867	82 259	Non
10_93	900	730	94	8 446 240	8 446 340	36 034 100	Non
19_71	900	165	96	3 646 220	3 646 320	8 737 240	Non
21_90	900	301	95	6 401 000	6 401 100	16 874 600	Non
36_92	900	514	96	8 426 650	8 426 750	25 721 200	Non
6_76	900	0	90	9	99	174	Non
Moyenne :	900	305	95	5 186 232	5 186 330	15 538 267	

Tableau 3.17 - Résultats heuristique P/Q, deuxième groupe

Les résultats obtenus par cette heuristique surpassent ceux obtenus par le *FC* ainsi que par ILOG Solver 6.0 tout en étant légèrement inférieurs à l'heuristique *Option*. Le nombre d'instances de problèmes résolus (51) en fait une heuristique d'ordonnancement de catégories de véhicules intéressante pour la résolution du problème de "car-sequencing".

De plus, les profondeurs maximales atteintes pour les instances de problèmes du second groupe sont équivalentes à celles atteintes par l'heuristique *Option*. Cependant, le nombre de nœuds parcourus, le nombre de retours en arrière, le nombre d'appels à la procédure de filtrage ainsi que le temps nécessaire sont plus élevés pour l'heuristique *P/Q* que pour l'heuristique *Option*.

Finalement, à des fins de comparaison, une heuristique d'ordonnancement de catégories de véhicules aléatoire a été développée. Cette heuristique, nommée *Random*, tire aléatoirement un nombre r se situant entre 1 et le nombre de catégories restantes dans le domaine de la variable courante. On affecte à la position de la séquence courante la $r^{ième}$ valeur de son domaine. Les résultats obtenus par la méthode d'ordonnancement de catégories de véhicules aléatoire sont présentés à l'Annexe 1. L'heuristique *Random* a permis de résoudre 8 problèmes du premier groupe. Elle n'est cependant pas parvenue à résoudre de problèmes du second groupe.

L'utilisation d'une heuristique de choix de valeurs a aussi été testée lors de la résolution du problème de "car-sequencing" par ILOG Solver. Tel que mentionné par Régin et Puget [1997], Solver utilise un principe d'heuristique appliqué aux options du problème et non aux catégories. Cet ordre a déjà été prouvé comme essentiel pour la résolution des instances de problèmes 4_72, 6_76 et 10_93. Les auteurs sont en effet parvenus à résoudre ces problèmes en spécifiant l'heuristique à appliquer au problème.

Le calcul suggéré par Régin et Puget est décrit à l'Équation (3.2). Il permet de calculer la marge excédentaire pour chaque option i de la séquence. On peut ainsi favoriser

les options dont la demande se rapproche le plus de la capacité de la séquence de production pour chacune des options, soit celles dont le $slack_i$ est le plus petit.

$$slack_i = nb\_véhicules - p_i \times (Demande_i / q_i) \quad (3.2)$$

Les résultats obtenus par ILOG Solver 6.0 avec l'ajout de l'heuristique d'ordonnancement des options du problème sont présentés aux Tableaux 3.18 et 3.19. Notons que Solver a permis de résoudre 3 problèmes non satisfiables sur une possibilité de 4. De plus, cette heuristique a permis de prouver la non satisfiabilité du problème 26_82, problème pour lequel il était impossible de déterminer l'état auparavant. Le calcul de l'heuristique à appliquer aux options pour ce problème en particulier est $o = \{0, 3, 1, 2, 4\}$. Le nombre de problèmes résolus parmi les problèmes par taux d'utilisation a aussi augmenté comparativement à l'utilisation de Solver sans heuristique passant de 8 à 34 problèmes résolus.

La Figure 3.10 illustre les résultats obtenus par chacune des heuristiques présentées dans cette section sur les 70 problèmes du premier groupe. Les étiquettes « F » et « S » indiquent respectivement les schémas d'ordonnancement de valeurs *fail-first* et *succeed-first*. Le graphique est également subdivisé en groupes de méthodes de résolution. Le premier groupe contient uniquement le *FC*. Cet algorithme sert de base à chacune des autres méthodes développées. Le second groupe est composé des heuristiques d'ordonnancement de catégories de véhicules de type *fail-first*. Le troisième groupe contient les heuristiques d'ordonnancement de catégories de véhicules de type *succeed-first*. Le quatrième groupe contient les heuristiques d'ordonnancement de catégories de

véhicules *Fréquence* et *Random*. Finalement, le cinquième groupe comprend les essais effectués avec ILOG Solver 6.0. Il est à noter que la méthode *ILOG Slack* est une méthode d'ordonnancement de valeurs de type *fail-first*.

Problème	Total temps	BT	Nb_choice_pt	nb_variables	réussites
60	456	38 851	39 055	5 356	5
65	720	63 460	63 667	3 704	3
70	721	55 033	55 250	2 879	2
75	454	38 318	38 568	5 359	5
80	365	17 539	17 797	6 186	6
85	454	29 315	29 593	5 361	5
90	229	7 623	7 937	7 842	8
Moyenne :	486	35 734	35 981	5 241	

Tableau 3.18 - Résultats ILOG Solver 6.0 et heuristique Slack, premier groupe

Problème	Total temps	BT	Nb_choice_pt	nb_variables	réussites
16_81	900	58675	58730	627	Non
26_82	36	6944	7071	4782	Oui
41_66	1	0	111	4777	Oui
4_72	1	0	110	4780	Oui
10_93	11	758	757	626	Oui
19_71	900	35214	35271	624	Non
21_90	900	85509	85573	624	Non
36_92	153	13731	13730	623	Oui
6_76	121	9355	9354	623	Oui
Moyenne :	336	23354	23412	2010	

Tableau 3.19 - Résultats ILOG Solver 6.0 et heuristique Slack, deuxième groupe

On peut voir que l'ajout d'heuristiques d'ordonnancement de catégories de véhicules a permis de bonifier l'algorithme de *FC* conventionnel. L'heuristique *Option* a permis de résoudre 51 problèmes de plus que le *FC*. De plus, on peut constater la dominance des heuristiques d'ordonnancement de catégories de véhicules de type *fail-first* soit *MaxTauxUt, Option, P/Q* et *MinVehicule* sur les heuristiques d'ordonnancement de catégories de type *succeed-first* soit *MinTauxUt, Vehicule* et *MinOption*. Les heuristiques

de type *succeed-first* ont même obtenues des résultats inférieurs au *FC*. L'heuristique aléatoire nommé *Random* a obtenu de meilleurs résultats que les heuristiques *MinTauxUt, Vehicule, Fréquence, MinOption* et *MinVehicule* ce qui démontre leur inefficacité.

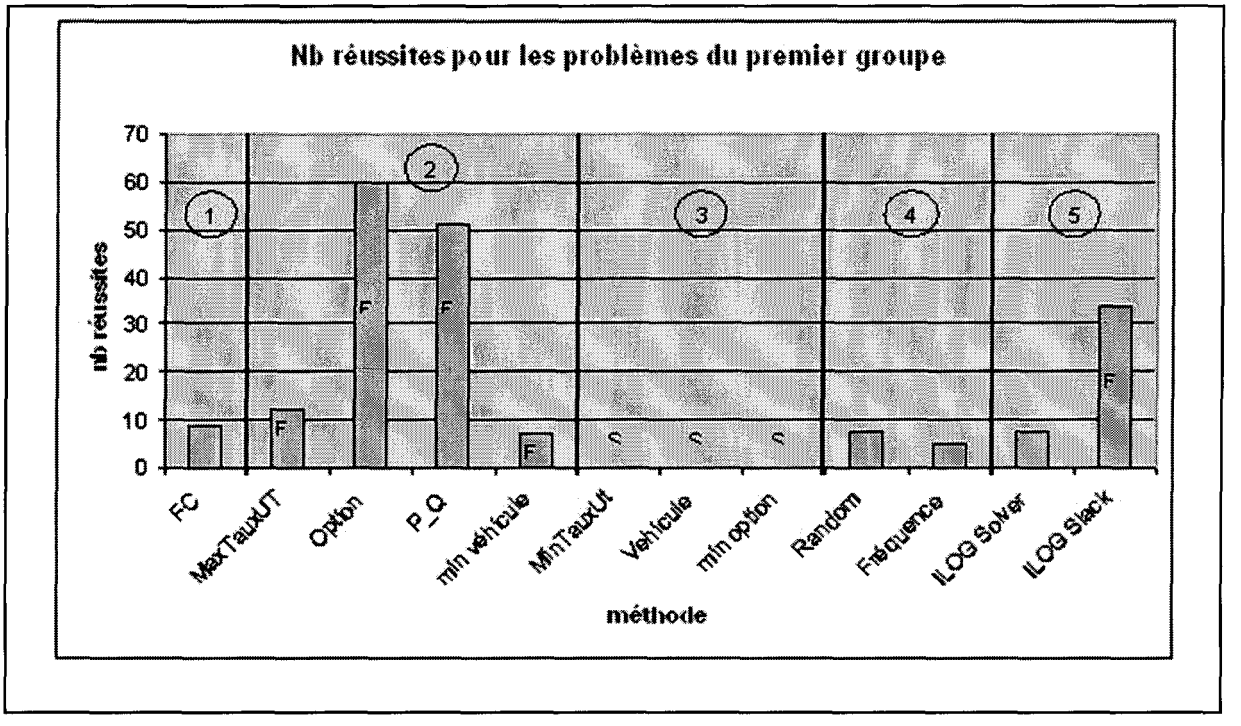

Figure 3.10 - Résultats pour les problèmes du premier groupe

L'utilisation d'heuristiques d'ordonnancement de catégories de véhicules a permis d'augmenter le nombre de problèmes résolus tant lors de l'utilisation du FC que lors de l'utilisation de ILOG Solver. Notons que les heuristiques *Option* et *P_Q* obtiennent des résultats intéressants pour les problèmes du premier groupe. Cet élément n'est donc pas à négliger dans le développement d'algorithmes de résolution de CSP et, plus particulièrement, du problème de "car-sequencing". Le FC, avec l'ajout de l'heuristique *Option*, a permis de résoudre 60 problèmes comparativement à 34 pour ILOG Solver avec l'heuristique *Slack*. Le FC utilisé surpasse donc ILOG Solver en terme de problèmes du

premier groupe résolus. Cependant, ILOG Solver a surpassé le FC lors de la résolution des problèmes plus difficiles ainsi que pour les problèmes non satisfiables.

Le but de cette section était de démontrer l'importance de l'ordonnancement de valeurs lors de la résolution de problème de CSP. Les résultats obtenus par le FC ainsi que par ILOG Solver démontrent que l'ordonnancement de valeurs augmente les capacités d'un algorithme de résolution de CSP pour traiter le problème de « car-sequencing ».

L'heuristique *Option* a obtenu des résultats supérieurs aux autres heuristiques pour les problèmes de la CSPlib. Ce résultat peut cependant être influencé par le fait que tous les problèmes traités comprenaient des contraintes de capacité se situant entre 1/5 et 2/3. Il serait intéressant, dans l'avenir, de tester cette heuristique sur des problèmes dont les contraintes de capacité sont de natures différentes.

3.6.3 Utilisation de nogoods lors de la recherche

Tel que mentionné par Frost et Dechter [1994], l'utilisation de listes de nogoods peut permettre d'éviter de retomber sans cesse dans les mêmes situations d'inconsistance. Cette section décrit une méthode de stockage et d'utilisation des nogoods appliquée lors de la résolution du problème de "car-sequencing" par l'algorithme de FC couplé à l'utilisation d'une heuristique d'ordonnancement de valeurs.

En tenant compte uniquement de la globalité des contraintes de capacité, la résolution du problème de "car-sequencing" revient à la résolution d'un même sous-problème à plusieurs reprises. En effet, chaque position de la séquence est restreinte par les mêmes contraintes de capacité et ce, peu importe où elle se trouve dans la séquence de

production. Le niveau de contrainte d'une position de la séquence est strictement dicté par les positions précédant celle-ci dans la séquence de production. Pour un ensemble de contraintes de capacité $C_i = q_i / p_i$, le nombre de positions affectant la position de la séquence courante sera toujours égal ou inférieur au p_i maximum de l'ensemble C. C'est pourquoi, il est possible d'éviter certains situations d'inconsistance en énumérant toutes les séquences de production de taille $max(p_i)$ et en vérifiant si chacune de ces séquences produit une situation d'inconsistance.

Étant donné l'utilisation de l'algorithme de FC dans le présent travail, il est impossible que l'instanciation courante soit inconsistante avec l'état des positions de la séquence précédentes. En effet, tous les éléments présents dans le domaine de la position de la séquence courante sont consistants avec les catégories de véhicules prises par les positions précédentes étant donné le filtrage. C'est pourquoi, on utilise, dans la présente section, le terme nogood pour indiquer toute séquence d'instanciation vidant le contenu d'une position de la séquence future.

On extrait, en phase de pré-traitement, tous les nogoods présents dans un sous-problème du problème initial afin d'enregistrer toutes ces situations d'inconsistance. Cette étape implique la vérification de M^t sous-séquences tel que t est la taille maximale des nogoods extraits et M le nombre de catégories de véhicules. On effectue donc le parcours complet d'un arbre de profondeur t et d'arité M en utilisant les contraintes du problème pour vérifier les situations d'inconsistance. On restreint, dans le présent travail, la taille maximale des nogoods au p_i maximum du problème. Pour les problèmes de la librairie CSPlib, les nogoods seront restreint à une taille maximale de 5.

Les nogoods extraits permettent de faire des sauts en largeur dans l'arbre de recherche. En effet, le fait de savoir qu'une affectation de $v \in D_k$ à X_k vide le domaine d'une variable non-instanciée X_j permet de ne pas tester cette affectation et d'ainsi passer à la catégories de véhicule suivante. On évite par le fait même certains tests de consistance effectués par le FC.

Pour l'exemple à la Figure 3.11, la catégorie de véhicule 1 est testée et est détectée inconsistante par le FC. La valeur 2 ne fait plus partie du domaine de la position de la séquence courante et est alors ignorée. La troisième catégorie de véhicule est également ignorée puisqu'elle définit un nogood existant dans l'ensemble des nogoods détectés en phase de pré-traitement. Les catégories de véhicules 4, 5 et 6 sont testées et la catégorie 7 ne figure plus dans le domaine. Les catégories 8 et 9 seront évitées puisqu'elles figurent comme nogoods. On définit donc les concepts de nœuds testés et de nœuds évités.

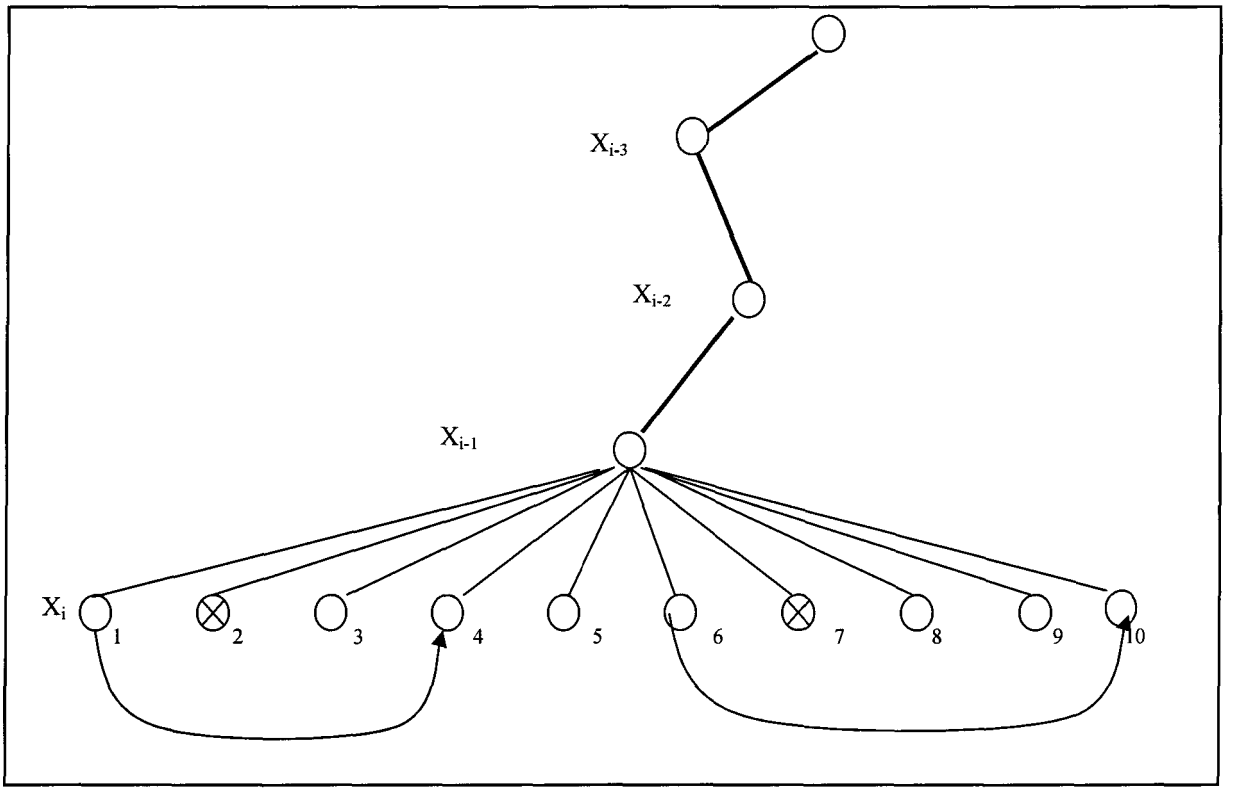

Figure 3.11 - Sauts en largeur induits par l'ensemble de nogoods

Le stockage des nogoods est un élément important de leur utilisation. En effet, comme ils sont utilisés lors de chaque affectation de catégories de véhicules à une position de la séquence, ils doivent être facilement accessibles. C'est pourquoi, les nogoods détectés par la phase de pré-traitement sont stockés sous la forme d'un ensemble d'entiers. Pour un nogood composé des affectations $X_1=12$, $X_2=3$ et $X_3=11$ et une instance de problème comportant 21 catégories de véhicules, on transforme chacune des affectations en un nombre en base 21. On aura donc $11 * 21^1 + 3 * 21^2 + 12 * 21^3$ qui sera égale à 112 686. Chacun des nogoods étant illustré sous la forme d'un entier et stocké dans un ensemble d'entiers, il suffit, lors de la descente, de garder à jour la somme de la position courante *k* et de la variable *k-1* remis en base du nombre de catégorie pour détecter la

présence de nogoods de taille 2. On ajoute la variable *k-2* au calcul pour les nogoods de taille 3 et ainsi de suite pour les nogoods de taille supérieure.

L'impact de l'utilisation de nogoods lors de la recherche se calcule en terme de nœuds évités par le pré-traitement. Les Tableaux 3.20 à 3.25 contiennent le nombre de nœuds évités par les nogoods lors de la descente dépendamment de l'heuristique d'ordonnancement de catégories de véhicules utilisée. Les résultats sont regroupés par schéma d'ordonnancement de valeurs soit *fail-first*, *succeed-first* et les autres heuristiques soit *Random* et *Fréquence*. Ils sont aussi divisés par groupe de problèmes de la librairie CSPlib.

On peut voir que, en terme de nœuds évités, les heuristiques de type *succeed-first*, avec un total de 7 868 652 nœuds évités pour l'ensemble des problèmes, profitent beaucoup plus de l'utilisation des nogoods que les heuristiques de type *fail-first* qui n'ont évités qu'un total de 1 690 114 noeuds. Les autres heuristiques soit *Random* et *Fréquence* n'obtiennent pas de meilleurs résultats avec 2 094 291 nœuds évités. Le fait que les heuristiques *succeed-first* soient plus influencées par l'utilisation de nogoods vient du fait que ces heuristiques parcourent beaucoup plus de branches de l'arbre de résolution. Cela est dû, dans un premier temps, au fait que ces heuristiques coupent beaucoup moins de branches de l'arbre étant donné les catégories de véhicules sélectionnées. Dans un second temps, ces heuristiques ne parviennent pas à résoudre un grand nombre de problèmes et passent donc beaucoup plus de temps en phase de recherche que les heuristiques *fail-first*.

Heuristique	MinTauxUt	Vehicule	MinOpt
Problèmes			
60	130 614	612 401	179 048
65	57 772	132 599	447 702
70	58 182	119 114	693 919
75	275 744	534 478	575 802
80	140 850	180 336	135 946
85	271 404	607 577	187 684
90	490 482	320 629	487 547
Somme	1 425 048	2 507 134	2 707 648
		Total :	**6 639 830**

Tableau 3.20 - Noeuds évités par les nogoods, heuristiques *succeed-first*, premier groupe

Heuristique	MinTauxUt	Vehicule	MinOpt
Problèmes			
16_81	60 841	14 619	107 523
26_82	19 692	3 071	99 962
41_66	49 939	129 891	118 806
4_72	17 950	1 346	5 872
10_93	12 221	34 630	95 894
19_71	41 210	14 213	23 512
21_90	71 767	3 512	55 527
36_92	83 279	21 552	126 801
6_76	0	15 192	0
Somme	356 899	238 026	633 897
		Total :	**1 228 822**

Tableau 3.21 - Noeuds évités par les nogoods, heuristiques *succeed-first*, deuxième groupe

Heuristique	Option	MaxTauxUt	P/N	MinVehicule
Problèmes				
60	9	0	0	3 942
65	22	1	2	5 096
70	10	2	2	12
75	9	12	3	4
80	15	3	0	9
85	11	2 590	2	4
90	18	77 697	10	355 592
Somme :	94	80 305	19	364 659
			Total :	**445 077**

Tableau 3.22 - Noeuds évités par les nogoods, heuristique *fail-first*, premier groupe

Heuristique	Option	MaxTauxUt	P/N	MinVehicule
Problèmes				
16_81	87	13 858	245 308	5 575
26_82	32 829	0	105 821	127 849
41_66	0	0	0	0
4_72	0	0	0	43 463
10_93	151 181	3 982	0	14
19_71	0	26 611	33 335	35 548
21_90	2	0	87 913	331 319
36_92	83	0	0	256
6_76	0	2	0	1
Somme	184 182	44 453	472 377	544 025
			Total :	**1 245 037**

Tableau 3.23 - Noeuds évités par les nogoods, heuristiques *fail-first*, deuxième groupe

Heuristique	Random	Fréquence
Problèmes		
60	1 253	33 960
65	3 116	446
70	202 579	9
75	14 336	4
80	295 920	238
85	320 479	2
90	296 390	69 777
Somme	1 134 073	104 436
	Total :	**1 238 509**

Tableau 3.24- Noeuds évités par les nogoods, heuristiques Random et Fréquence, premier groupe

Heuristique	Random	Fréquence
Problèmes		
16_81	32 459	5 546
26_82	13 785	105 821
41_66	136 486	0
4_72	4 695	690
10_93	111 709	1 593
19_71	1 568	165 890
21_90	80 144	139 577
36_92	52 867	1 113
6_76	1 838	1
Somme	435 551	420 231
	Total :	**855 782**

Tableau 3.25 - Noeuds évités par les nogoods, heuristiques Random et Fréquence, deuxième groupe

L'utilisation des nogoods ne permet pas, peu importe l'heuristique utilisée, d'éviter un grand pourcentage de nœuds visités dans l'arbre de résolution. En effet, comme illustré dans le Tableau 3.26, un pourcentage inférieur à 1 % des nœuds visités est évité lors des résolutions de problèmes par le FC utilisant les listes de nogoods.

Heuristique	**Pourcentage de *nœuds évités***
Option	0,09%
Fail	0,02%
P_Q	0,14%
MinVehicule	0,17%
Random	0,23%
Fréquence	0,08%
Succeed	0,26%
Vehicule	0,36%
MinOpt	0,44%
Moyenne :	0,20%

Tableau 3.26 - Pourcentage de noeuds évités par les nogoods

L'utilisation de méthodes permettant de stocker des nogoods avant la descente dans l'arbre de résolution n'a donc pas obtenu de résultats concluants. C'est pourquoi, cette méthode est abandonnée pour la suite du travail. Un schéma de stockage et de vérification des nogoods prenant en compte la demande restante pour chacune des catégories serait peut-être plus approprié puisqu'il représenterait mieux l'état de la chaîne de production.

3.6.4 Résolution concurrente du problème de "car-sequencing"

Les approches séquentielles ont certaines limites. En effet, les résultats démontrent que ces algorithmes, s'ils ne découvrent pas rapidement une solution, stagnent dans une seule région de l'arborescence et ne parviennent pas à effectuer les retours en arrière permettant de remettre en cause les choix faits aux niveaux supérieurs de l'arbre. Par

exemple, pour le *FC* utilisant l'heuristique d'ordonnancement de catégories de véhicules *Option*, le temps moyen pour atteindre la profondeur maximale dans l'arborescence représente un tiers du temps de recherche total. Les méthodes parallèles de résolution sont analysées dans ce travail de recherche afin de tenter de repousser les limites de l'algorithme séquentiel en favorisant une meilleure diversification de la recherche. Notons que, quoique les méthodes parallèles de résolution soient utilisées, l'environnement de résolution demeure séquentiel. On parle alors de résolution concurrente utilisant plusieurs processus distincts pour la résolution. Plusieurs schémas de résolution parallèle de CSP ont été présentés dans la littérature. La plupart d'entre eux sont des adaptations d'algorithmes séquentiels existants et c'est dans cette optique que le *FC* développé a été parallélisé. L'avenue privilégiée consiste à subdiviser le problème initial en sous-problèmes. Cette subdivision permet de diversifier la recherche de solutions en utilisant les ressources mises à la disposition pour la résolution simultanée de plusieurs régions de l'espace des solutions.

Il est à noter que la meilleure heuristique d'ordonnancement de catégories de véhicules développée soit *Option* est utilisée lors des tests de la méthode. De plus, le schéma d'ordonnancement des positions de la séquence *fail-first* est également privilégié par la méthode concurrente de résolution. Cependant, les listes de nogoods ne sont pas utilisées avec la méthode concurrente de résolution.

3.6.4.1 Subdivision du problème de "car-sequencing"

Tel que mentionné précédemment, l'espace de solutions du problème de "car-sequencing" peut être illustré sous la forme d'une arborescence d'une profondeur égale au

nombre de voitures et d'une arité égale au nombre de catégories de véhicules. C'est cet arbre de résolution qui est subdivisé pour générer les sous-problèmes.

La génération des sous-problèmes est effectuée en phase d'initialisation et consiste à définir les sous-séquences de production de taille *max_t* consistantes avec les contraintes du problème. Dans le présent travail de recherche, cette séquence est restreinte à une taille de 2 étant donné que plusieurs des contraintes de capacité des instances de problèmes utilisées possèdent des ratios tel que le q_i est 1. De plus, cela permet de limiter le temps nécessaire pour la phase d'initialisation.

Supposons l'instance de problème présentée au Tableau 3.27. La génération de sous-problèmes de taille 2 est réalisée en déterminant lesquelles des 9 séquences possibles, à savoir (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2) et (3,3), sont consistantes. On trouve ainsi que seules les séquences (1,3), (2,3), (3,1), (3,2) et (3,3) sont consistantes et génèrent ainsi 5 sous-problèmes. Pour ce faire, l'algorithme utilise le filtrage des domaines défini au chapitre précédent. La Figure 3.12 illustre cette subdivision de l'arbre de résolution en sous-problèmes.

		Catégories		
Option	Contrainte	1	2	3
1	1/3	•		
2	2/3			•
3	1/2	•	•	

Tableau 3.27 - Instance de problème de "car-sequencing"

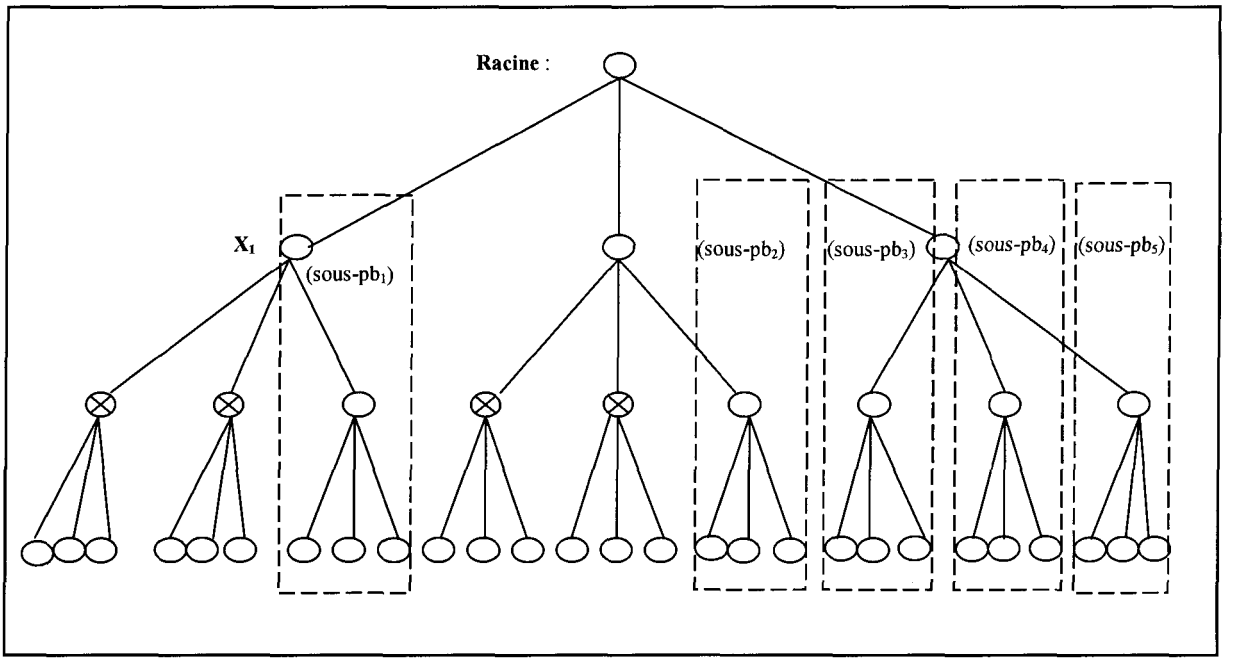

Figure 3.12 - Décomposition de l'arbre de résolution en sous-problèmes

On constate, dans un premier temps, que la génération de ces sous-problèmes permet de restreindre l'espace de recherche. Les Tableaux 3.28 et 3.29 présentent le nombre moyen de sous-problèmes générés pour chacun des groupes de 10 problèmes de la librairie CSPlib. La colonne *max sous pb* indique le nombre moyen maximal de sous-problèmes possibles pour chaque groupe de problèmes. Le nombre de sous-problèmes possibles pour un problème est M^{max_t} où M est le nombre de catégories de véhicules et max_t indique la taille maximale des séquences vérifiées. Par exemple, pour le problème 4_72 et une taille des séquences de 2, le nombre de sous-problèmes maximum est 22^2 soit 484. La colonne *sous pb* indique le nombre de sous-problèmes répertoriés comme consistants et *%* indique le pourcentage de sous-problèmes consistants. On peut constater que la génération des sous-problèmes réduit en moyenne 50% de l'espace de solution.

Problèmes	max sous pb	Sous pb	%
60	456	234,5	52%
65	545,7	277,3	51%
70	583,3	287,5	49%
75	606,6	294,9	49%
80	631,3	308,3	49%
85	699	341,2	48%
90	741,2	368,6	50%

Tableau 3.28 - Impact de la génération des sous-problèmes, premier groupe

Problèmes	max sous pb	Sous pb	%
16_81	676	291	43%
26_82	576	255	44%
41_66	361	211	58%
4_72	484	223	46%
10_93	625	279	45%
19_71	529	267	50%
21_90	529	261	49%
36_92	484	239	49%
6_76	484	259	54%

Tableau 3.29 - Impact de la génération des sous-problèmes, deuxième groupe

Suite à la définition des sous-problèmes, la méthode de résolution démarre un ensemble de processus où chacun d'eux est responsable de la résolution d'un groupe de sous-problèmes. Ces processus effectuent un parcours en profondeur de l'arborescence représentant chacun des sous-problèmes qui lui ont été affectés en utilisant les formalismes de réduction de domaines définis pour le *FC* séquentiel. Chacun de ces processus possède un identificateur numérique nommé *id* qui permet de déterminer les sous-problèmes qui lui sont alloués.

L'algorithme d'allocation des sous-problèmes aux processus prend en entrée un nombre de processus maximum soit *nb_proc* ainsi que le nombre de sous-problèmes

répertoriés soit *nb_sous_pb*. On divise le nombre de sous-problèmes par le nombre de processus pour obtenir le nombre de sous-problèmes à affecter à chacun des processus (*nb_par_proc*). Chaque processus se voit affecter *nb_par_proc* sous-problèmes à résoudre. La Figure 3.13 illustre l'algorithme d'affectation des sous-problèmes au processus. L'algorithme utilise *Début* et *Fin*, deux tableaux de taille *nb_proc*, pour stocker les bornes inférieures et supérieures de chacun des groupes de sous-problèmes.

```
Procédure SubdivisionSousPb
  Entrée : nb_sous_pb, nb_proc
  Sortie : Début, Fin

  nb_par_proc ← ⌊ nb_sous_pb / nb_proc ⌋
  point_surplus ← nb_proc-(nb_sous_pb modulo nb_proc)
  POUR CHAQUE processus p FAIRE
      Début_i ← id * nb_par_proc
      SI id ≥ point_surplus ALORS
            Début_i ← Début_i + i - point_surplus
      Fin_i ← debut_i + nb_par_proc
      SI id ≥ point_surplus ALORS
            Fin_i ← Fin_i + 1
```

Figure 3.13 - Algorithme de subdivision du problème et d'affectation aux processus

Lorsque le nombre de sous-problèmes divisé par le nombre de processus n'est pas entier, l'algorithme affecte un sous-problème supplémentaire à certains processus. On utilise l'élément *point_surplus* pour déterminer le premier processus qui a un sous-problème supplémentaire à traiter. Les processus dont le *id* est égal ou supérieur à *point_surplus* sont affectés à un sous-problème supplémentaire. La Figure 3.14 illustre l'affectation des 5 sous-problèmes de l'exemple précédent à 3 processus. On peut constater que le premier processus est affecté à la résolution du sous-problème pb_1 et que les processus 2 et 3 se voient affecter un sous-problème supplémentaire.

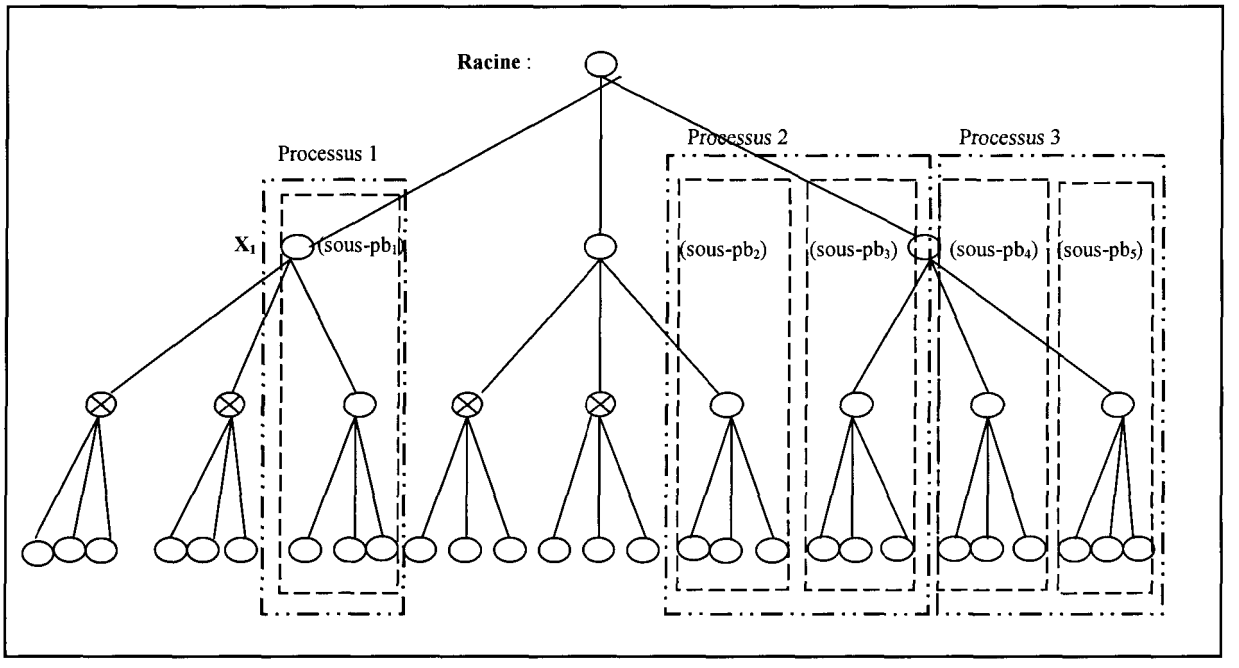

Figure 3.14 - Affectation des sous-problèmes aux processus.

3.6.4.2 L'algorithme de Forward-Checking concurrent

Suite à la subdivision de l'espace de recherche, l'algorithme de *FC* est exécuté sur chacun des sous-problèmes associés aux processus. La Figure 3.15 illustre les passages d'états des processus pendant la résolution concurrente [Stallings 2001]. L'étape de Création (*New*) permet, entre autres, de définir l'ordre de priorité des processus créés. Dans le cadre du présent travail, les processus sont tous considérés de priorité égale. Suite à la création, un processus passe à un état Prêt (*Ready*), c'est-à-dire qu'il est disponible pour débuter la recherche de solutions lorsque des ressources seront mises à sa disposition. La transition de l'état Prêt vers l'état Exécution ainsi que le temps réparti pour l'exécution sont déterminés par le système d'exploitation de l'ordinateur utilisé. Lorsque le temps de traitement alloué à un processus est terminé (*time out),* celui-ci, s'il n'a pas complété toutes ses instructions, retourne à l'état Prêt. Il est alors placé par le système dans une file

d'attente définissant l'ordre d'affectation des ressources aux processus. Un processus passe à un état Attente (*Blocked*) lorsque sa continuation dépend d'un événement ou de la réception d'un signal. La mise en attente est utilisée dans le cadre du *FC* lorsqu'un processus n'est pas suffisamment performant. Finalement, un processus passe à un état Terminé (*Exit*) lorsque les opérations nécessaires à son exécution sont complétées ou lorsqu'il a parcouru tout l'espace de recherche qui lui a été affecté lors de la subdivision du problème initial.

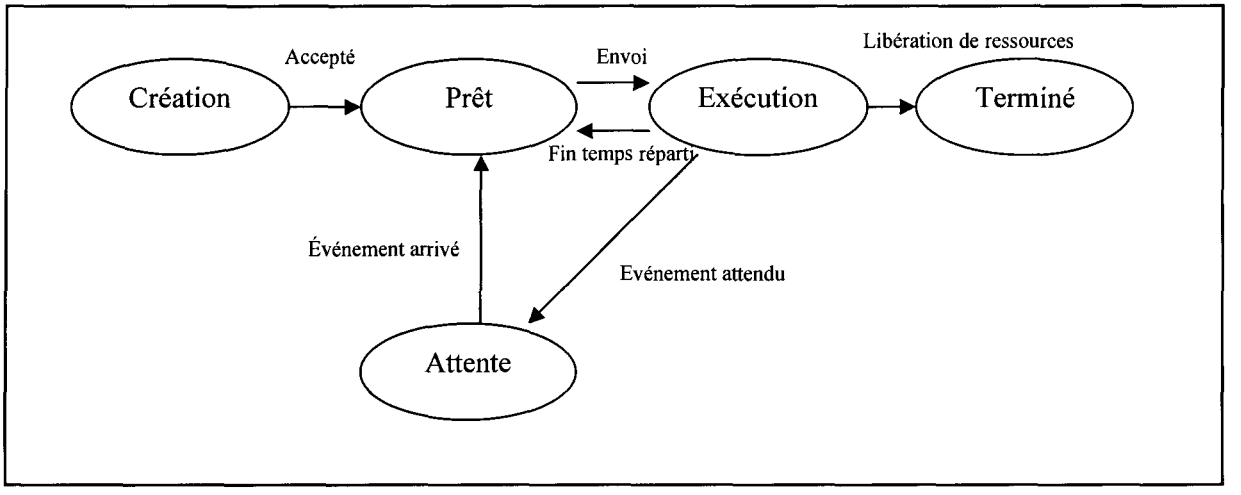

Figure 3.15 - Modèle d'états des processus

Une stratégie d'allocation des ressources aux processus a été mise en place afin d'allouer le plus de ressources possibles aux processus performants tout en offrant du temps de calcul aux processus moins performants. La performance d'un processus est quantifiée, dans le cadre de la résolution d'un sous-problème de "car-sequencing", par la profondeur maximale atteinte dans l'arbre par celui-ci. Cette stratégie fait passer les processus dans un état d'Attente lorsqu'ils ont atteint un certain nombre de retours en arrière. Le redémarrage

d'un processus s'effectue, par la suite, en fonction de sa performance relativement aux autres processus.

L'algorithme de *FC* modifié est décrit en Figure 3.16. Les éléments ajoutés à l'algorithme séquentiel pour permettre la résolution simultanée figurent en caractères gras. Tout d'abord, l'élément 1 permet d'effectuer une boucle sur chacun des sous-problèmes affectés au processus. Cette structure itérative permet de parcourir chacun des sous-problèmes affectés au processus et ce jusqu'à la découverte d'une solution ou jusqu'à la fin du parcours des sous-problèmes. L'algorithme, comme pour la version séquentielle, met à zéro (RAZ) les structures de données nécessaires pour le traitement. Il appelle ensuite, en élément 2, la fonction *instanciation_t* qui permet de guider la descente dans l'arbre pour le sous-problème à résoudre. Par exemple, si la sous-séquence initiale du sous-problème est (3,1), *instanciation_t* affecte 3 à la première variable et 1 à la seconde variable en appliquant les méthodes de filtrage de domaines sur les positions suivantes ainsi qu'en mettant à jour la demande pour les catégories concernées et pour les options produites.

Parmi les éléments ajoutés pour la méthode concurrente figure un tableau en mémoire globale nommé *TabBest* qui stocke les profondeurs maximales atteintes dans l'arborescence par les processus. Lors d'une situation d'inconsistance, l'algorithme vérifie si la profondeur courante est la profondeur maximale atteinte pour le processus courant et, dans l'affirmative, met à jour *TabBest* en appelant la méthode *MAJTabBest* (élément 3) qui reçoit en paramètres la profondeur atteinte ainsi que le *id* du processus concerné.

L'algorithme utilise aussi un principe de tour pour définir le passage d'un état Exécution ou Prêt à un état d'Attente. Un tour est défini par un nombre de retours en

arrière maximal soit NB_BT_MAX. L'élément 4 de l'algorithme vérifie, dans un premier temps, si le nombre de retours en arrière effectués est supérieur à NB_MAX_BT. Lors des tests effectués dans le présent travail, NB_BT_MAX a été initialisé à 10 000. La valeur de ce paramètre est basée sur les résultats obtenus avec le *FC* séquentiel. Celui-ci effectuant plus d'un million de retours en arrière, le nombre de tours de l'algorithme parallèle s'approche donc de 100 ce qui ne représente pas un nombre très élevé d'arrêts et de redémarrages des processus.

Si le processus courant a effectué un tour complet et qu'il est le processus ayant atteint la profondeur maximale dans l'arbre, il reste en état Prêt et envoi un signal de redémarrage aux autres processus (élément 5). Le nombre de signaux de redémarrage envoyé est dépendant du tour courant. Le nombre de processus actifs (*nb_proc_act*) est divisé par deux à tous les tours de l'algorithme. Le nombre de processus actifs peut être divisé par un nombre supérieur à deux. Cependant, plus ce nombre est élevé, plus la méthode utilisée est élitiste c'est-à-dire qu'elle favorise les processus performants au détriment de la diversification de la recherche. Lorsqu'il ne reste qu'un seul processus actif, ce dernier redémarre tous les processus.

Dans le cas où le processus courant n'est pas le processus ayant atteint la profondeur maximale, il se met en état d'attente tel que défini par l'élément 6. Il sera redémarré par la suite dépendamment de ses performances.

Le FC concurrent, suite à un retour en arrière vers une profondeur inférieure à la profondeur initiale des sous-problèmes générés, (*prof_sous_pb*) définit ce sous-problème comme inconsistant et passe au prochain sous-problème qui lui a été affecté. Suite à la

résolution de tous les sous-problèmes affectés à un processus, celui-ci est supprimé des processus actifs et passe alors en état Terminé (élément 8).

```
Procédure FC_Concurrent
  Entrée : id, t, Début, Fin
  Sortie : Solution ou Fin des sous-problèmes du processus
  tc ← Début_id
  tour ← 1, profmax ← 0
  TANT QUE i < Fin ET Etat=0 OU Etat=2 FAIRE    (1)
    Consistant ← Vrai
    Etat ← 0
    RAZ()
      SI (instanciation (t_tc)) ALORS      (2)
        TANT QUE Etat = 0 FAIRE
           SI Consistant ALORS
              Consistant ← Descente()
          SINON
             SI positionCourante > profmax ALORS
              profmax ← positionCourante
                Afficher « Sous-solution »
                MAJTabBest (id, profmax) mémoire globale (3)
                Consistant ← Remontée()
                nb_bt ← nb_bt + 1
                SI positionCourante < prof_sous_pb ALORS
                  Etat ← 2
              SI Etat ← 0 ALORS
                SI nb_bt ≥ NB_BT_MAX * tour ALORS
                  SI TabBestIndMax (id) ALORS
                     SI nb_proc_act = 1 ALORS          } (4)
                       nb_proc_act ← NB_PROC_MAX
                       EnvoyerMessage (Prêt) à tous
                   SINON
                      nb_proc_act ← nb_proc_act / 2
                      EnvoyerMessage (Prêt) aux      } (5)
                     nb_proc_act meilleurs processus
                  SINON
                      Attente()        (6)
          SI positionCourante = nb_vehicule ALORS
            Afficher « Solution »
            Etat ← 1
            EnvoyerMessage (Terminé) à tous      (7)
          SINON SI Etat = 2 ALORS
                    tc ← tc + 1
  SI Etat = 2 ALORS
      Terminé()      (8)
```

Figure 3.16 - Algorithme FC parallèle

3.6.4.3 Résultats obtenus

La méthode de diversification de la recherche par l'exécution concurrente de l'algorithme a été implantée avec l'utilisation de la classe de processus *pthread* répondant aux normes IEEE POSIX 1003.1c sur une machine UNIX [Mueller 1993]. L'ordinateur utilisé est le même que pour les exécutions de l'algorithme séquentiel soit un Pentium 3 à 1 Ghz munie de 396 Mo de mémoire vive. Les ressources matérielles n'ont donc aucunement été augmentées lors de l'exécution de l'algorithme concurrent. Étant donné les ressources matérielles utilisées, le nombre de processus a été fixé à 125 dans le présent travail. Le temps alloué, comme pour la méthode séquentielle, est de 15 minutes.

Les Tableaux 3.30 et 3.31 présentent les résultats obtenus pour les deux groupes de problèmes de la librairie CSPlib. La colonne *Temps proc* a été ajoutée pour définir le temps moyen de traitement mis à la disposition du processus qui a réussi à atteindre la profondeur maximale soit de sa création à sa terminaison. L'initiation et le démarrage des processus s'effectuant en suivant l'ordre des *id* de chacun d'eux, *Temps proc* indique le temps moyen réel de recherche du processus concerné.

L'ajout des méthodes de résolution concurrente améliore les résultats obtenus par le *FC*. En effet, le nombre de problèmes résolus est passé de 60, pour la version du FC utilisant l'heuristique d'ordonnancement de valeurs *Option*, à 67 pour la version concurrente utilisant la même heuristique d'ordonnancement des catégories de véhicules pour les problèmes du premier groupe. De plus, la diversification de la recherche a également permis d'augmenter l'écart entre les résultats obtenus par ILOG Solver 6.0 qui résolvait seulement 34 problèmes du premier groupe. Le temps moyen nécessaire pour la

résolution est aussi passé de 157 secondes pour l'algorithme séquentiel à 53 secondes pour la version concurrente.

Problèmes	Total temps	Temps proc	Profondeur	BT	Nœuds visités	Filtrage	réussite
60	39	8	200	1014	1212	2194	10
65	28	7	200	2605	2803	5628	10
70	50	28	200	7442	7640	15428	10
75	72	30	200	6535	6733	12594	10
80	109	84	199	90902	91099	198216	9
85	55	22	200	14436	14634	30760	8
90	25	2	200	16	214	239	10
Moyenne :	**53,87**	**25,84**	**199,73**	**17564,23**	**17761,96**	**37865,50**	

Tableau 3.30 - Résultats, méthode parallèle, premier groupe

Problèmes	Total temps	Temps proc	Profondeur	BT	Nœuds visités	Filtrage	réussite
16_81	23 196	23 168	100	6 065 710	6 065 800	19 239 600	Oui
26_82	185	85	100	18 628	18 726	48 647	Oui
41_66	24	2	100	139	237	325	Oui
4_72	34	7	100	3 144	3 242	10 534	Oui
10_93	1 238	1 212	94	589 581	589 673	2 223 560	Non
19_71	328	234	95	177 017	177 110	452 240	Non
21_90	213	154	96	27 896	27 990	95 445	Non
36_92	101	11	97	2 784	2 879	6 953	Non
6_76	1 384	1 338	92	1 013 550	1 013 640	1 721 560	Non
Moyenne :	**398,78**	**338,89**	**97,00**	**203 870**	**203 965**	**507 325**	

Tableau 3.31 - Résultats, méthode parallèle, deuxième groupe

Pour les problèmes du second groupe, la méthode concurrente a résolu 4 problèmes comparativement à aucun pour le FC séquentiel utilisant l'heuristique d'ordonnancement de valeurs *Option*. Il est à noter que ces quatre problèmes font partie des quatre problèmes connus comme satisfiables dans la littérature. Pour sa part, ILOG Solver 6.0 avait résolu 3 problèmes satisfiables sur une possibilité de 4 et 3 problèmes non satisfiables sur une possibilité de 4. ILOG Solver 6.0 demeure donc supérieur en terme de performance pour ce groupe de problèmes. Ce phénomène était prévisible et s'explique par le fait que

l'espace de solutions n'est pas diminué par le parcours simultané de plusieurs régions de celui-ci. La résolution de problèmes non satisfiables sous-entend le parcours complet de l'arborescence ce qui ne peut être favorisé que par un meilleur filtrage des domaines. Cependant, la méthode concurrente a augmenté la profondeur maximale moyenne atteinte soit 97 comparativement à 95 pour le FC utilisant l'heuristique d'ordonnancement de valeurs *Option*.

La Figure 3.17 présente les résultats mis à jour pour toutes les méthodes développées dans le cadre de ce travail de recherche. Les sections 1 à 5 correspondent aux résultats des divers algorithmes présentés développés dans les sections précédentes et la section 6 présente le nombre de problèmes résolus par le FC concurrent pour les problèmes du premier groupe. On peut y voir l'évolution de l'algorithme du FC de base, avec 9 problèmes résolus, au FC concurrent, avec 67 problèmes résolus, en passant par les différentes heuristiques d'ordonnancement des catégories de véhicules développées.

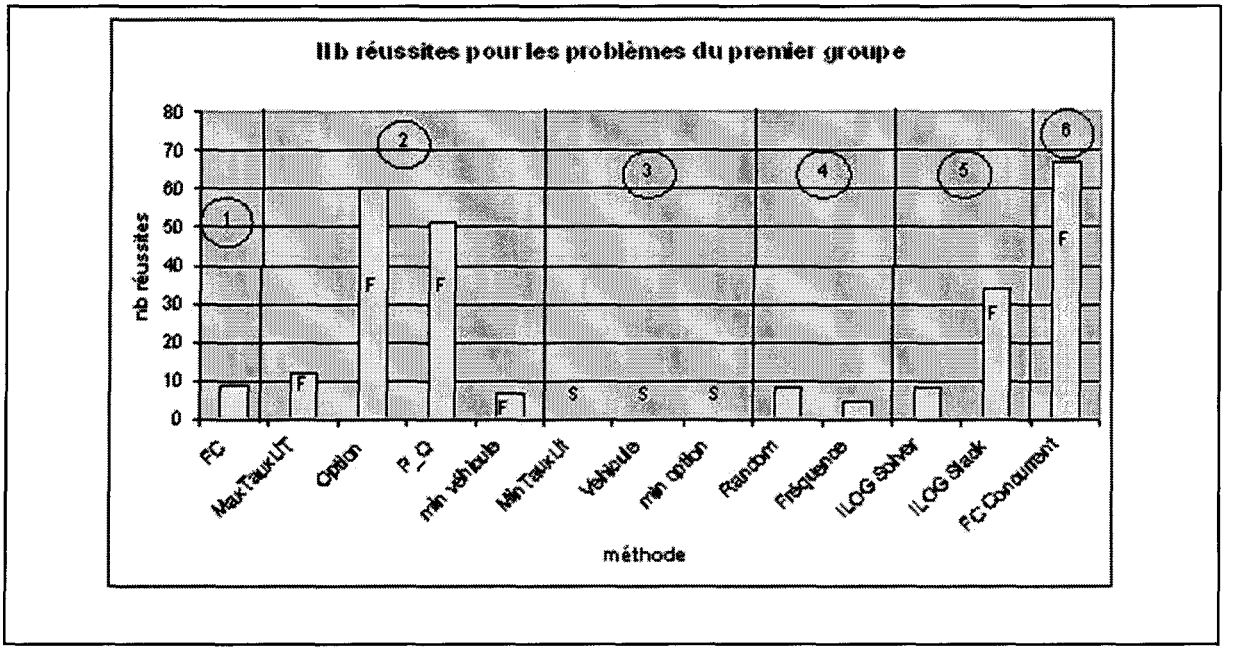

Figure 3.17 - Résultats pour les problèmes du premier groupe

CHAPITRE 4

CONCLUSION

Ce travail de recherche a permis de définir la performance des méthodes de résolution de problèmes de CSP appliquées à un problème d'ordonnancement à connotation industrielle. Les algorithmes présentés au Chapitre 2 ont servis de base au développement d'un algorithme spécifique au problème de "car-sequencing". Les travaux décrits dans ce chapitre ont montré la diversité des algorithmes présents dans la littérature pour la résolution des CSP et le présent travail en a appliqué quelques-uns au problème de "car-sequencing". Les objectifs fixés pour ce travail de recherche ont été atteints et les méthodes présentées dans ce mémoire ont permis d'obtenir des résultats intéressants.

Dans un premier temps, l'objectif visant à développer un algorithme général de résolution du problème de "car-sequencing" par les méthodes de résolution de CSP a été atteint par l'élaboration de l'algorithme *FC*. En effet, l'étude du problème a permis le développement d'une méthode de filtrage des domaines des variables lors du parcours en profondeur de l'arbre de résolution. Ce filtrage utilise les particularités des contraintes de capacité du problème pour diminuer les domaines des variables et, par le fait même, l'espace de recherche de solutions. L'importance de l'étude de la problématique à résoudre lors de l'élaboration de l'algorithme est ressortie de ce travail. En effet, l'ajout des contraintes implicites fut un point tournant pour la résolution du problème de "car-sequencing".

Dans un second temps, les analyses du problème ont permis de définir les particularités de celui-ci afin de répondre au deuxième objectif du travail de recherche qui visait à quantifier l'impact de l'ajout d'éléments, tels des heuristiques, à un algorithme de résolution de CSP. Pour ce faire, la mise en œuvre de l'heuristique d'ordonnancement de

variables *fail-first* a été étudiée. Une particularité du problème, induite par la globalité des contraintes de capacité, est que ce schéma d'ordonnancement de variables est préservé lorsqu'on affecte les variables en suivant l'ordre total de l'ensemble X. De plus, l'ajout d'ordonnancement de valeurs a été étudié et testé sur les instances de problèmes de la CSPlib. Il a été démontré que les schémas d'ordonnancement des catégories de véhicules *fail-first* obtiennent de meilleurs résultats que le *succeed-first* lors de la résolution du problème de "car-sequencing". Cependant, la difficulté de la mise en œuvre des différents schémas d'ordonnancement de valeurs a été de déterminer le calcul du niveau de difficulté d'une catégorie de véhicules. Encore une fois, l'interdépendance entre l'étude de la problématique à résoudre et l'ajout d'éléments à la méthode de résolution fut démontrée.

L'heuristique utilisant le nombre d'options d'une catégorie pour en calculer le niveau de difficulté, soit *Option*, a obtenu de meilleurs résultats que les autres heuristiques. Cependant, les instances de problèmes ayant toutes les mêmes contraintes de capacité, il serait intéressant de tester les heuristiques sur des problèmes de contraintes de capacité de natures différentes. L'ajout de nouveaux problèmes à la librairie existante de problèmes ou l'utilisation d'instances de problèmes réels permettrait de mieux comparer les heuristiques d'ordonnancement de valeurs suggérées dans ce travail.

L'impact de l'utilisation d'heuristiques d'ordonnancement de valeurs lors de la résolution d'instances de problème par ILOG Solver 6.0 a aussi été défini. Les tests effectués ont démontré qu'un bon ordonnancement de valeurs permet d'améliorer les performances du logiciel tant sur les instances de problèmes satisfiables que non-

satisfiables. A ce sujet, les travaux de recherche ont permis de définir l'état du problème 36_92 de la librairie *CSPlib* qui n'avait jamais été solutionné par quiconque jusqu'ici.

Les travaux effectués sur l'utilisation des listes de nogoods ont démontré qu'ils permettent d'éliminer certains sous-arbres de l'arbre de résolution. Cependant, les essais ont montré que la méthode suggérée ne permet pas d'éliminer un grand nombre de nœuds dans l'arbre de résolution du problème de "car-sequencing" et des travaux dans cette direction pourraient être poursuivis.

Enfin, la méthode de résolution concurrente présentée a permis de répondre au troisième objectif de ce travail de recherche. En effet, l'utilisation d'une telle méthode de résolution, consistant à subdiviser le problème initial en sous-problèmes, a permis de diversifier la recherche de solutions et ainsi d'augmenter l'efficacité de l'algorithme de *FC*. Les résultats montrent une amélioration au niveau du nombre d'instances de problèmes de la librairie *CSLlib* résolues tant un niveau des problèmes du premier groupe que du deuxième groupe. De plus, les profondeurs maximales atteintes pour les problèmes non résolus ont aussi augmentées.

La simplicité de la méthode proposée en fait une avenue intéressante pour le développement d'algorithmes sur une architecture à plusieurs processeurs. Il serait également intéressant d'y intégrer une méthode d'échanges de *nogoods* ou de *goods* entre les processus afin de générer de l'apprentissage. La méthode de stockage et de gestion des *nogoods* présentée pourrait alors y être utilisée. De plus, une meilleure méthode d'affectation des sous-problèmes aux processus permettant à un processus qui a terminé son

traitement de prendre en charge un sous-problème affecté à un autre processus pourrait également favoriser la recherche de solutions.

Les problèmes non-satisfiables suggérés dans la librairie *CSLlib* n'ont pu être résolus par les méthodes de résolution développées. La bonification de la méthode de diminution de l'espace de recherche représente une voie à explorer dans des recherches futures pour favoriser la résolution de telles instances. De plus, l'avenue de l'utilisation d'ordinateurs parallèles pour la résolution serait intéressante puisqu'elle favoriserait le traitement en augmentant les ressources informatiques disponibles.

Les instances de problèmes suggérées dans la librairie *CSPlib* étant de nature théorique, il serait intéressant d'appliquer les formalismes étudiés sur des instances de problèmes "car-sequencing" réels. Pour ce faire, une hybridation entre les méthodes exactes de résolution de problèmes de CSP et une heuristique pourrait être développée afin de prendre en compte la totalité des contraintes de la chaîne de montage en plus de tenir compte des coûts associés aux purges de l'atelier de peinture.

BIBLIOGRAPHIE

Bacchus, F. et A. J. Grove (1995). On the Forward Checking Algorithm, Dans *Proceedings* CP, 292-308, Cassis, France.

Bessière, C. (1994). "Arc-Consistency and Arc-Consistency Again." Artificial Intelligence **65**: 179-190.

Bessière, C., A. Chmeiss et L. Saïs (2001). Neighborhood-Based Variable Ordering Heuristics for the Constraint Satisfaction Problem., Dans *Proceedings* CP, 565-569, Paphos, Cyprus.

Bessière, C., E. C. Freuder et J.-C. Régin (1995). Using Inference to Reduce Arc Consistency Computation, Dans *Proceedings* IJCAI, 592-599, Montréal, Québec, Canada.

Bessière, C., A. Maestre et P. Meseguer (2002). La famille ABT, Dans *Proceedings* JNPC, 57-67, Nice, France.

Bessière, C., P. Meseguer, E. C. Freuder et J. Larrosa (1999). On Forward Checking for Non binary Constraint Satisfaction, Dans *Proceedings* CP, 88-102, Alexandria, Virginia, USA.

Bessière, C. et J.-C. Régin (2001). Refining the Basic Constraint Propagation Algorithm., Dans *Proceedings* IJCAI, 309-315, Seattle, USA.

Cooper, M. C., D. A. Cohen et P. Jeavons (1994). "Characterising tractables constraints." Artificial Intelligence **65**: 347-361.

Davenport, A. et E. Tsang (1995). Solving constraint satisfaction sequencing problems by iterative repair, Dans *Proceedings* Planning and Scheduling Special Interest Group Workshop, 16, Colchester, UK.

Davenport, A. J., E. P. K. Tsang, C. J. Wang et K. Zhu (1994). GENET : a connectionist architecture for solving constraint satisfaction problems by iterative improvement., Dans *Proceedings* AAAI, 325-330, Seattle, Washington.

Dechter, R. (1990). "Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition." Artificial Intelligence **41**: 273-312.

Dechter, R. (2003). Constraint Processing, Morgan Kaufmann, 450 pages

Dechter, R. et D. Frost (2002). "Backjump-based Backtracking for Constraint Satisfaction Problems." Artificial Intelligence **136**: 147-188.

Dechter, R. et I. Meiri (1994). "Experimental evaluation of preprocessing algorithms for constraint satisfaction problems." Artificial Intelligence **68**: 211-241.

Dent, M. J. et R. E. Mercer (1994). Minimal Forward Checking., Dans *Proceedings* ICTAI, 432-438, New Orleans, Louisiana, USA.

Dincbas, M., H. Simonis et P. V. Hentenryck (1988). Solving the car-sequencing problem in logic programming, Dans *Proceedings* ECAI, 290-295, Munich, Germany.

Freuder, E. C. (1982). "A Sufficient Condition for Backtrack-Free Search." J. ACM **29**: 24-32.

Frost, D. et R. Dechter (1994). Dead-end driven learning, Dans *Proceedings* AAAI, 294-300, Seattle, Washington.

Frost, D. et R. Dechter (1995). Look-ahead value ordering for constraint satisfaction problems, Dans *Proceedings* IJCAI, 572-578, Montreal, Québec, Canada.

Gaschnig, J. (1979). Performance measurement and analysis of search algorithms.,Carnegie Mellon University, Rapport **CMU-CS-79-124.**

Gent, I. (1998). Two Results on Car-sequencing Problems.,University of Strathclyde, Rapport **APES-02-98, 7.**

Gottlieb, J., M. Puchta et C. Solnon (2003). A Study of Greedy, Local Search, and Ant Colony Optimization Approaches for Car Sequencing Problems., Dans *Proceedings* EvoWorkshops, 246-257, Essex, UK.

Gravel, M., C. Gagné et W. L. Price. (2004). "Review and comparaison of three method for the solution of the car-sequencing problem." Journal of The Operational Research Society.

Habbas, Z., M. Krajecki et D. Singer (2000). Domain Decomposition for Parallel Resolution of Constraint Satisfaction Problem with OpenMP, Dans *Proceedings* European Workshop on OpenMP, 8, Edinburgh, UK.

Hamadi, Y., C. Bessière et J. Quinqueton (1998). Distributed Intelligent Backtracking., Dans *Proceedings* ECAI, 219-223, Brighton, UK.

Hao, J. K., P. Galinier et M. Habib (1999). "Metaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes." Revue d'Intelligence Artificielle **13**: 283-324.

Haralick, R. M. et G. L. Elliott (1980). "Increasing tree search efficiency for contraint satisfaction problems." Artificial Intelligence **14**: 263-313.

Hentenryck, P. V., Y. Deville et C.-M. Teng (1992). "A generic arc-consistency algorithm and its specializations." Artificial Intelligence **57**: 291-321.

Ignizio, J. P. et T. M. Cavalier (1994). Linear Programming, 666 pages

Kumar, V. (1992). Algorithms for Constraint-Satisfaction Problems : A survey. AI Magazine. **13:** 32-44.

Kwan, A. C. M. et E. P. K. Tsang (1996). Minimal Forward Checking with Backmarking, Dans *Proceedings* ICTAI, 290-298, Toulouse, France.

Lynce, I. et J. Marques-Silva (2002). The Effect of Nogood Recording in MAC-CBJ SAT Algorithms, Dans *Proceedings* International Workshop on Constraint Solving and Constraint Logic Programming, 144-158, Cork, Ireland.

Mackworth, A. K. (1977). "Consistency in Networks of Relations." Artificial Intelligence **8**: 99-118.

Mohr, R. et T. C. Henderson (1986). "Arc and path consistency revisited." Artificial Intelligence **28**: 225-233.

Mueller, F. (1993). A Library Implementation of POSIX Threads under UNIX., Dans *Proceedings* USENIX Winter, 29-42, San Diego, California.

Nguyen, T. et Y. Deville (1998). "A Distributed Arc-Consistency Algorithm." Sci. Comput. Program. **30**: 227-250.

Parrello, B. D. et W. C. Kabat (1986). "Job-shop scheduling using automated reasoning : A case of the car sequencing problem." Journal of automated reasoning **2**: 1-42.

Prosser, P. (1995). Forward Checking with Backmarking., Dans *Proceedings* Constraint Processing, Selected Papers, 185-204.

Prosser, P. (1995). MAC-CBJ : maintaining arc consistency with conflict-directed backjumping,University of Strathclyde, Rapport **95/177.**

Rao, V. N. et V. Kumar (1993). "On the Efficiency of Parallel Backtracking." IEEE Trans. Parallel Distrib. Syst. **4**: 427-437.

Régin, J.-C. (1996). Generalized arc consistency for global cardinality constraint, Dans *Proceedings* AAAI/IAAI, 209-215, Portland, Oregon.

Régin, J.-C. et J.-F. Puget (1997). A filtering Algorithme for Global Sequencing Constraints, Dans *Proceedings* CP, 32-46, Linz, Austria.

Smith, B. M. (1996). Succeed-first or Fail-first : A Case Study in Variable and Value Ordering,University of Leeds, England, Rapport **96.26.**

Solnon, C. (2000). Ant-P-solveur : un solveur de contraintes à base de fourmis artificielles, Dans *Proceedings* JFPLC, 189-204, Marseille, France.

Stallings, W. (2001). Operating Systems - Internals and Design Principles, Prentice Hall, 832 pages

Stevenson, W. J. et C. Benedetti (2001). La gestion des opérations. Montréal, Québec, Canada, 785 pages

Terrioux, C. (2002). Approche structurelles et coopératives pour la résolution des problèmes de satisfaction de contraintes. Ecole Doctorale de mathématiques et d'informatique de Marseille. Aix-Marseille, Université d'Aix-Marseille I: 134.

Tsang, E. (1999). "A Glimpse of Constraint Satisfaction." Artificial Intelligence Review **13**: 215-227.

Wagner, H. (1975). Principles of Operations Research, Prentice-Hall, 1039 pages

Waltz, D. L. (1972). Generating semantic description from drawing of scenes with shadows,MIT, Rapport **MAC-AI-TR-271.**

Warwick, T. et E. P. K. Tsang (1995). "Tackling car sequencing problems using a generic genetic algorithm." Evolutionary Computation **3**: 267-298.

Zhou, X. et T. Nishizeki (1999). "Edge-Coloring and f-Coloring for Various Classes of Graphes." Journal of Graph Algorithms and Applications **3**.

ANNEXE 1

RÉSULTATS – HEURISTIQUES D'ORDONNANCEMENT DE VALEURS

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	900	330	188	3 912 587	3 912 775	10 832 072	0
65	900	261	183	3 180 997	3 181 182	8 737 002	0
70	900	342	177	3 212 266	3 212 443	10 204 827	0
75	900	400	139	2 871 064	2 871 204	9 619 397	0
80	900	402	154	2 692 612	2 692 766	9 022 208	0
85	900	259	152	1 960 136	1 960 290	6 891 191	0
90	900	467	139	2 181 167	2 181 305	9 167 033	0
Moyenne :	900	351	161	2 858 690	2 858 852	9 210 533	

Tableau A-1.1 - Résultats heuristique MinOption, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
16_81	900	467	81	2 334 110	2 334 190	11 995 200	Non
26_82	900	200	80	1 977 570	1 977 650	7 195 400	Non
41_66	900	617	67	5 107 900	5 107 960	16 369 500	Non
4_72	900	155	80	1 421 680	1 421 760	5 048 560	Non
10_93	900	871	75	5 146 720	5 146 790	25 237 200	Non
19_71	900	151	77	1 143 480	1 143 560	4 850 180	Non
21_90	900	125	71	977 484	977 555	4 409 130	Non
36_92	900	679	84	4 245 820	4 245 910	19 833 200	Non
6_76	900	29	66	306 038	306 104	916 597	Non
Moyenne :	900	366	76	2 517 867	2 517 942	10 650 552	

Tableau A-1.2 - Résultats heuristique MinOption, deuxième groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	900	173	155	1 316 119	1 316 274	3 337 481	0
65	900	200	171	3 430 384	3 430 554	7 135 580	0
70	810	359	187	5 117 245	5 117 424	10 093 265	1
75	634	198	195	3 806 437	3 806 632	6 746 314	3
80	810	179	193	3 584 180	3 584 373	7 628 382	1
85	900	331	166	4 755 203	4 755 368	10 552 100	0
90	900	258	176	2 532 959	2 533 135	7 785 167	0
Moyenne :	836	242	177	3 506 075	3 506 251	7 611 184	

Tableau A-1.3 - Résultats heuristique Fréquence, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
16_81	900	48	94	566 920	567 014	2 119 750	Non
26_82	900	315	88	3 164 410	3 164 500	11 046 600	Non
41_66	900	73	82	1 342 850	1 342 930	2 714 750	Non
4_72	900	11	83	121 552	121 635	418 965	Non
10_93	900	611	85	6 721 230	6 721 320	23 654 400	Non
19_71	900	614	83	6 360 030	6 360 110	23 573 800	Non
21_90	900	447	89	4 564 450	4 564 540	19 466 900	Non
36_92	900	123	90	1 417 980	1 418 070	5 158 990	Non
6_76	900	100	78	1 427 790	1 427 870	3 499 020	Non
Moyenne :	900	260	86	2 854 135	2 854 221	10 183 686	

Tableau A-1.4 - Résultats heuristique Fréquence, deuxième groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	900	246	185	3 723 452	3 723 639	8 476 017	0
65	900	243	178	3 054 244	3 054 422	7 172 306	0
70	900	237	173	2 606 894	2 607 064	7 816 367	0
75	900	498	172	5 399 805	5 399 980	13 963 577	0
80	900	358	169	2 598 763	2 598 932	9 024 002	0
85	900	430	163	3 149 782	3 149 943	11 462 679	0
90	900	326	155	2 281 854	2 282 009	8 015 158	0
Moyenne :	900	334	171	3 259 256	3 259 427	9 418 586	

Tableau A-1.5 - Résultats heurisitique Véhicule, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussite
16_81	900	109	87	649 773	649 860	3 045 700	Non
26_82	900	14	72	99 325	99 397	447 503	Non
41_66	900	831	81	6 384 900	6 384 980	24 579 300	Non
4_72	900	708	88	8 185 220	8 185 310	30 155 700	Non
10_93	900	365	72	2 037 990	2 038 070	10 316 600	Non
19_71	900	569	67	3 087 700	3 087 770	15 130 000	Non
21_90	900	17	87	129 555	129 642	651 807	Non
36_92	900	73	83	468 946	469 029	2 208 850	Non
6_76	900	239	82	3 461 980	3 462 060	9 501 000	Non
Moyenne :	900	325	80	2 722 821	2 722 902	10 670 718	

Tableau A-1.6 - Résultats heuristique Véhicule, deuxième groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	810	249	154	1 734 733	1 734 885	4 066 403	1
65	783	195	180	2 681 072	2 681 252	5 856 080	2
70	817	189	190	3 227 750	3 227 941	6 275 661	1
75	740	199	192	4 065 100	4 065 292	7 491 978	2
80	900	197	188	3 133 513	3 133 701	6 503 961	0
85	900	196	178	2 035 129	1 950 762	4 985 926	0
90	853	345	170	3 341 548	3 341 718	10 149 753	1
Moyenne :	829	224	179	2 888 406	2 876 507	6 475 680	

Tableau A-1.7 - Résultats heuristique MinVehicule, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussite
16_81	900	22	91	241 099	241 190	926 105	Non
26_82	900	392	85	3 870 140	3 870 230	13 649 100	Non
41_66	900	0	94	10 863	10 957	29 061	Non
4_72	900	453	78	3 462 600	3 462 680	14 752 400	Non
10_93	900	117	82	1 123 900	1 123 980	4 130 920	Non
19_71	900	177	81	1 799 000	1 799 080	6 568 850	Non
21_90	900	686	82	6 729 280	6 729 360	31 529 500	Non
36_92	900	48	85	451 510	451 595	1 671 630	Non
6_76	900	0	75	4 874	4 949	13 254	Non
Moyenne :	900	211	84	1 965 918	1 966 002	8 141 202	

Tableau A-1.8 - Résultats heuristique MinVehicule, deuxième groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
60	777	190	180	2 798 413	2 798 594	6 002 118	2
65	814	156	171	2 765 618	2 765 792	4 838 022	1
70	900	207	158	2 903 633	2 903 785	6 447 071	0
75	629	213	188	4 705 177	4 705 365	8 334 081	4
80	827	250	182	2 919 693	2 919 874	7 931 653	1
85	900	282	179	3 618 823	3 619 004	9 436 152	0
90	900	429	165	3 029 065	3 029 231	10 906 898	0
Moyenne :	821	247	175	3 248 632	3 248 806	7 699 428	

Tableau A-1.9 - Résultats heuristique Random, premier groupe

Problèmes	Total temps	Temps prof. Max	Profondeur	BT	Nœuds visités	Filtrage	réussites
16_81	900	319	81	1 907 330	1 907 410	8 749 450	Non
26_82	900	62	89	594 213	594 302	2 285 510	Non
41_66	900	611	82	9 614 670	9 614 750	23 368 300	Non
4_72	900	22	96	320 104	320 200	991 603	Non
10_93	900	730	70	3 666 400	3 666 470	20 464 200	Non
19_71	900	43	74	366 692	366 766	1 522 940	Non
21_90	900	853	79	6 669 410	6 669 490	35 760 800	Non
36_92	900	498	77	3 459 850	3 459 930	15 031 200	Non
6_76	900	12	81	190 985	191 066	525 780	Non
Moyenne :	900	350	81	2 976 628	2 976 709	12 077 754	Non

Tableau A-1.10 - Résultats heuristique Random, deuxième groupe