Résolution du problème de car-sequencing à l'aide d'une approche de type FC

Simon Boivin, Marc Gravel, Michaël Krajecki, Caroline Gagné

▶ To cite this version:

Simon Boivin, Marc Gravel, Michaël Krajecki, Caroline Gagné. Résolution du problème de car-sequencing à l'aide d'une approche de type FC. Premières Journées Francophones de Programmation par Contraintes, CRIL - CNRS FRE 2499, Jun 2005, Lens, pp.11-20. inria-00000050

HAL Id: inria-00000050
https://inria.hal.science/inria-00000050

Submitted on 25 May 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Résolution du problème de *car-sequencing* à l'aide d'une approche de type FC

Simon Boivin [1] **Marc Gravel** [1] **Michaël Krajecki** [2] **Caroline Gagné** [1]

[1] Département d'informatique et de mathématiques
Université du Québec à Chicoutimi, Chicoutimi, Québec, Canada, G7H 2B1

[2] CReSTIC EA 3804, Département de Mathématiques et Informatique
Université de Reims Champagne-Ardenne, 51687 REIMS, France

{simon_boivin,marc_gravel,caroline_gagne}@uqac.ca, michael.krajecki@univ-reims.fr

Résumé

Le problème de *car-sequencing* consiste à déterminer l'ordre dans lequel un ensemble de voitures seront fabriquées sur une chaîne d'assemblage. Certaines voitures comprennent des options telles que le toit ouvrant, l'air climatisé ou autres pour lesquelles une charge de travail additionnelle est requise à un poste particulier de la chaîne. Pour chacune de ces voitures difficiles, la chaîne d'assemblage possède une capacité maximale indiquant le nombre de voitures q pouvant être produites sur une séquence de p voitures consécutives. Dans cet article, la formulation du problème de *car-sequencing* est présentée sous la forme de problème de satisfaction de contraintes (CSP). Une méthode de filtrage des domaines (FC) ayant pour but la diminution de l'espace de solutions à parcourir est utilisée. Celle-ci intègre les contraintes de capacité de la chaîne d'assemblage en plus de certaines contraintes implicites définies dans la littérature. L'ajout d'heuristiques guidant la recherche de solutions est par la suite proposé. Les résultats obtenus par chacune des méthodes utilisées sont comparés avec les résultats obtenus par ILOG Solver 6.0.

Abstract

The *car-sequencing* problem is the ordering of a set of vehicles on an assembly line. Some cars have options such as sun-roof, air-conditioning or others for which additionnal workload is required at some workstations of the assembly line. For each difficult car, the assembly line has a maximum capacity of q vehicle which can be produced into a sequence of p consecutive cars. In this paper, the formulation of the *car-sequencing* problem is shown as a constraint satisfaction problem (CSP). A domain filtering method (FC) which purpose is to reduce the solution space to be explored is used. The filtering method uses the assembly line capacity constraints and some implicit constraints defined in the literature on the *car-sequencing* problem. The addition of heuristics guiding the solution search is described and the results obtained with each of them are compared. The results for the methods used are compared to the results of ILOG Solver 6.0.

1 Introduction

Les problèmes d'ordonnancement de la production sont des problèmes d'optimisation combinatoire où l'on cherche à déterminer une séquence d'exécution de travaux répondant à certains objectifs pouvant même être parfois conflictuels. Un problème d'ordonnancement particulièrement intéressant est celui du *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. Bien que ce problème concerne plusieurs ateliers présents dans l'ensemble de la chaîne d'assemblage, la littérature l'aborde généralement en ne considérant que la problématique de l'atelier de montage [11]. Dans cet atelier situé en aval de l'atelier de peinture, les différentes composantes sont ajoutées à la voiture et on note qu'une charge de travail supplémentaire est occasionnée à certains postes pour les voitures possédant des options particulières (toit ouvrant, air climatisé,...). Par conséquent, ces voitures doivent être dispersées dans la séquence de production de façon à

lisser la charge de travail.

Plusieurs méthodes ont été proposées dans la littérature pour résoudre ce problème. Parmi celles-ci figurent des heuristiques issues du domaine de l'intelligence artificielle et des méthodes exactes. Dans le cas des heuristiques, on note l'utilisation des réseaux de neurones avec l'algorithme GENET [2], les méthodes de recherche dans le voisinage [3], les méthodes évolutives telles que les algorithmes génétiques [16] et l'optimisation par colonies de fourmis [6] [8]. Dans ces travaux, la problématique est modélisée sous forme de problème d'optimisation en cherchant à déterminer une séquence minimisant le nombre de violations des contraintes de capacité. Toutefois, ces heuristiques ne peuvent garantir l'optimalité de la solution. Pour leur part, les méthodes exactes telles que la programmation linéaire en nombres entiers [7] et les méthodes de résolution de problèmes de satisfaction de contraintes (CSP) [15] [13] [1] représentent des alternatives de résolution pour certaines instances du problème de *car-sequencing*. Tel que défini par Kis [10], ce problème est de la classe des problèmes NP-Difficiles au sens fort et l'énumération de toutes les séquences possibles représente une tâche ardue. Étant donné que l'espace de solution à explorer croît exponentiellement en fonction de la taille du problème, les limites des méthodes exactes sont rapidement atteintes [8].

Dans cet article, un algorithme de type *Forward-Checking* (FC), permettant le filtrage des domaines pour diminuer la taille de l'espace de solutions, est présenté pour la résolution d'instances de problème théoriques de *car-sequencing*. Une comparaison de performance est réalisée entre ces approches et la composante CSP de ILOG Solver 6.0 pour les problèmes de la librairie CSPLib[1] .

2 Le problème de *car-sequencing*

Le problème théorique de *car-sequencing* ne considère que l'ordonnancement de la production à l'atelier de montage. Dans cet atelier, des contraintes de capacité précisent, dans le cas de certaines options, le nombre maximum de voitures q possédant l'option i qui peuvent être produites dans 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. Pour faciliter le traitement, une instance de problème de *car-sequencing* est décrite par un ensemble de voitures à produire où l'on regroupera les voitures semblables dans une même catégorie. À cet ensemble, s'ajoutera les contraintes de capacité du problème ainsi que la demande pour chacune des catégories de voitures. Une solution d'un problème de *car-sequencing* sera une séquence de production de voitures ne violant aucune contrainte de capacité.

Un exemple d'instance de problème tiré de Smith [15] et comportant 25 voitures à ordonnancer est présenté au Tableau 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 voitures. 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		*			*							
	total	3	1	2	4	3	3	2	1	1	2	2	1

TAB. 1 – Instance de problème de *car-sequencing*

Pour établir la difficulté d'une instance, il faut d'abord calculer le taux d'utilisation de chacune des options à l'aide de l'Équation 1. Ce taux correspond au ratio entre le nombre de voitures à produire possédant l'option et le nombre maximum de places disponibles dans la séquence pour cette option. On a donc, pour l'exemple du Tableau 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.

La moyenne des taux d'utilisation des options permet d'obtenir un indicateur du niveau de difficulté de l'instance. Toutefois, un haut taux d'utilisation ne garantit pas assurément un effort de résolution élevé et, à l'inverse, un faible taux d'utilisation ne garantit pas un faible effort de résolution. Les résultats présentés plus loin illustrent ce phénomène de façon claire.

$$TauxUtilisation_i = \frac{(DemandeOpt_i)}{nbVehicule * \frac{q_i}{p_i}} \qquad (1)$$

2.1 Modélisation du problème en CSP

La formulation du problème de *car-sequencing* sous forme de CSP retenue est celle de Smith [15]. Elle consiste à associer chaque variable X du problème à une position de la séquence S. Chaque variable X a alors un domaine D_x correspondant à chacune des catégories de voitures. 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 voitures ne doit ni dépasser, ni être inférieure à la demande.

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

2.2 Résolution du problème de *car-sequencing* sous la forme de CSP

Plusieurs travaux ont été effectués dans le but de résoudre le problème de *car-sequencing* sous la forme d'un CSP. Entre autre, Dincbas, Simonis et al. [4] ont suggéré l'ajout de certaines contraintes implicites afin de permettre 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 peuvent posséder l'option i sur toutes les séquences de taille p_i et au minimum aucune position ne peut posséder l'option i. Ce minimum peut, dans certains cas, être faux. Pour l'instance de problème du Tableau 1 et une affectation de la catégorie de voiture 8 à la première position de la sous-séquence, une situation d'inconsistance doit être détectée et ne le sera que par l'utilisation de contraintes implicites modifiant le nombre minimal de catégories 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'une voiture 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 l'option 2 étant de 2/3, le nombre de voitures produites possédant cette option étant 0 et le nombre de positions non-affectées étant 24, il ne sera pas possible de produire les 17 voitures restantes possédant l'option 2 en ne violant aucune des contraintes. L'ajout de ces contraintes implicites permet de détecter au plus tôt les séquences inconsistantes et ainsi diminuer le temps de calcul nécessaire pour la résolution. L'utilisation de cette contrainte sera détaillée en section 3.1.

Une approche de type MAC (Maintient d'Arc Consistance), utilisant les contraintes globales pour la suppression de régions de l'espace de solutions, a également été proposée dans la littérature par Régin et Puget [13]. Celle-ci utilise une transformation du graphe de contraintes initial pour favoriser le filtrage des domaines. Cette approche a permis la résolution de plusieurs instances de problème suggérées dans la littérature.

2.3 Heuristiques d'ordonnancement de variables et de valeurs

Lors de la descente dans l'arbre de recherche, deux choix doivent être faits par l'algorithme. D'une part, l'algorithme doit déterminer la prochaine variable à affecter (position de la séquence) et, d'autre part, le choix de la valeur à affecter (catégorie de voitures) à la variable doit être précisée. L'ordonnancement de variables et de valeurs a été étudié par Smith [15]. Plus spécifiquement, les effets du *succeed-first* et du *fail-first* ont été expérimentés pour le problème de *car-sequencing*. Le *fail-first* consiste à choisir la variable ou la valeur restreignant le plus possible les domaines des autres variables tandis que le *succeed-first* consiste à choisir la variable ou la valeur restreignant le moins possible les domaines des autres variables. Selon les expérimentations de Smith, une stratégie d'ordonnancement de variables et de valeurs de type *fail-first* obtient généralement de meilleurs résultats pour ce problème. De même, selon Haralick et Elliot [9], 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.

3 Proposition d'une approche FC pour le problème de *car-sequencing*

L'espace de solutions possibles d'un problème de CSP peut être représenté sous la forme d'un arbre de résolution où chacun des noeuds représente l'affectation d'une catégorie de voiture $v \in D_x$ à une position de la séquence et où chacun des niveaux de l'arbre représente une position de la séquence de production. La résolution d'un problème de CSP consiste alors à parcourir en profondeur cet arbre de recherche et l'instanciation d'une feuille de l'arbre représente une solution du problème.

L'algorithme de FC [12] est, dans un premier temps, utilisé pour la résolution du problème de *car-sequencing*. Tel que mentionné par Bessière [1], la résolution de ce type de problème est favorisée par une méthode de filtrage simple. En effet, la n-arité des contraintes fait en sorte qu'une recherche de type MAC alourdit fortement l'effort de résolution. C'est pourquoi une méthode de filtrage simple, exploitant les contraintes de capacité du problème pour éliminer certaines régions de l'espace de recherche de solution, est présentée. Dans un second temps, différentes heuristiques de domaines sont ajoutées pour orienter la recherche de solutions spécifiquement pour ce problème.

3.1 Principe de filtrage des domaines

Le FC utilisé pour la résolution du problème de *car-sequencing* consiste à éliminer des domaines des variables les éléments rendus inconsistants par l'état courant de la séquence de production de voitures. L'ajout de procédures de filtrage de domaines à un algorithme de type *BackTracking* augmente alors le nombre d'opé-

rations nécessaires à l'affectation d'une valeur à une variable. Le but recherché est de prendre de meilleures décisions et ainsi de diminuer l'effort global nécessaire à la résolution. Le filtrage proposé pour le problème de *car-sequencing* est un filtrage à trois phases. Une première phase consiste à supprimer, du domaine des positions non-affectées de la séquence, les catégories de voitures dont la demande a été remplie et de vérifier si cette opération a pour effet de vider le contenu d'un domaine. Une seconde phase consiste à vérifier si, étant donné l'affectation courante, une des contraintes implicites est violée et ainsi permettre la détection d'inconsistances beaucoup plus tôt. Finalement, une troisième phase consiste à supprimer du domaine des variables les éléments violant une des contraintes de capacité.

Pour faciliter le filtrage, le domaine d'une variable X_k est précisé à l'aide 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 de façon implicite l'ensemble des catégories de voitures possibles mais, lors de l'affectation d'une catégorie à une position 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 1. L'algorithme débute par l'initialisation des variables nécessaires au traitement. La variable *Debut* 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 *Premiere* indique la première position de la sous-séquence utilisant l'option sur laquelle le filtrage est effectué.

```
Procédure FiltrerRelation (PositionCourante, v)
    POUR CHAQUE Option i ∈ v FAIRE
        Debut ← max(1, positionCourante − p_{i+1})
        Fin ← positionCourante
        Trouve ← faux
        Saturation ← Debut
        Premiere ← 0
        q ← q_i
        p ← p_i

        // Vérification de la saturation de l'option
        // courante.
        TANT QUE Saturation < Fin ET Trouve = faux
          FAIRE
            SI X_Saturation possède l'option i ALORS
                p ← p − 1
                Premiere ← 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 l'option est saturée.
        SI q = 0 ALORS
            Debut ← positionCourante
            Fin ← min(Premiere + p, nbVehicule)
            Appeler SauvegarderDomaines()

            // Filtrer les domaines des variables suivantes.
            POUR j ← 0 à nb_categorie FAIRE
                POUR k ← 0 à Fin FAIRE
                    SI Categorie_j possède l'option i ALORS
                        Supprimer j de D_k

    // Vérification de la consistance.
    Consistant ← Vrai
    pos ← positionCourante
    TANT QUE Consistant = Vrai ET pos < Fin FAIRE
        SI D_pos = Nul ALORS
            Consistant ← Faux
        pos ← pos + 1
    Retourner Consistant
```

FIG. 1 – Algorithme de filtrage des relations pour le problème de *car-sequencing*

Sommairement, l'algorithme débute en effectuant une boucle sur chacune des options du problème. On vérifie alors la saturation de l'option i relativement 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 positions futures. L'utilisation des contraintes de capacité permet alors d'éviter de parcourir certains chemins inconsistants dans l'arbre de résolution. C'est ainsi que le FC diminue l'effort de calcul lors de la résolution d'instances de problème de *car-sequencing*.

L'algorithme de détection de culs-de-sac induit par les contraintes implicites est décrit à la Figure 2. L'algorithme utilise XN pour représenter l'ensemble des positions non-affectées et DOR_i pour représenter la demande restante pour une option i. L'algorithme stocke dans $prod_possible_i$ le nombre de positions restantes dans la séquence pouvant être affectées à une catégorie de voitures possédant 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.

3.2 Heuristiques d'ordonnancement de variables et de valeurs

Comme suggéré par Haralick et Elliot [9], une recherche utilisant un principe de filtrage de domaines de type FC est favorisée par l'utilisation d'une heuristique d'ordonnancement de variables *fail-first*. En effet, plus les culs-de-sac sont détectés à des niveaux supérieurs de l'arbre de résolution et plus l'effort engendré par la remontée est minimal. C'est pourquoi,

```
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 ∈ Dx ALORS
                incrémenter (prod_possible_i)
        SI prod_possible_i * q_i/p_i < DOR_i ALORS
            Consistant ← faux
        incrémenter ( i )
    RETOURNER Consistant
```

FIG. 2 – Algorithme de vérification des contraintes implicites du problème

les résultats présentés dans cet article sont basés sur une heuristique de choix de variables de type *fail-first*.

Selon Smith [15], ce type de schéma, appliqué au problème de *car-sequencing*, respecte l'ordre total de l'ensemble des variables du problème. En effet, la globalité des contraintes fait en sorte que l'affectation d'une valeur $v \in D_1$ à la première variable du problème (X_1) a un impact sur les m variables consécutives à X_1 tel que m est le p_j maximum pour les options présentes dans la catégorie de voiture v. Cette propriété du problème fait en sorte que l'application d'une heuristique d'ordonnancement de variables de type *fail-first* respecte l'ordre des variables de l'ensemble X, soit *{1, 2, ..., n}*.

En ce qui concerne l'ordonnancement de valeurs, quatre heuristiques de type *fail-first* et trois heuristiques de type *succeed-first* sont testées pour le problème de *car-sequencing* ainsi que des méthodes statiques et dynamiques d'ordonnancement de valeurs. De plus, une heuristique visant le lissage de la production de chacune des options et une autre heuristique d'ordonnancement de valeurs aléatoires sont également évaluées.

Tout d'abord, trois heuristiques d'ordonnancement de valeurs de type *succeed-first* dont deux dynamiques et une statique sont proposées soit :

Min_TauxUT : Favorise les catégories de voitures les plus faciles en définissant la difficulté d'une catégorie comme étant la somme des taux d'utilisation des options qui la composent. Elle prend également en compte la demande restante pour chacune des catégories, ce qui en fait une méthode dynamique.

Max_Véhicule : Heuristique d'ordonnancement dynamique qui utilise l'effectif restant à produire pour chacune des catégories de voitures afin de conserver le plus de catégories de voitures dans les domaines des variables.

Min_Option : Heuristique statique favorisant les catégories possédant le plus petit nombre d'options pour ainsi diminuer l'impact du filtrage des domaines relié aux contraintes de capacité des options.

De plus, quatre heuristiques d'ordonnancement de valeurs de type *fail-first* dont deux dynamiques et deux statiques sont également proposées soit :

Max_TauxUT : Heuristique qui, à l'inverse de l'heuristique *Min_ TauxUT*, cherche à maximiser la somme des taux d'utilisation des options qui la composent et prend également en compte la demande restante pour chacune des catégories ce qui en fait une heuristique dynamique.

Min_Véhicule : Heuristique dynamique qui, inversement à *Max_ Véhicule*, cherche à garder le moins de catégories possibles dans le domaine des variables étant donné les demandes restantes pour chacune de celles-ci.

Max_Option : Heuristique statique qui favorise les catégories comprenant le plus d'options.

Max_P_Q : Heuristique qui ordonne les catégories en cherchant à maximiser les ratios inverses des contraintes de capacité des options présentes à l'intérieur de celles-ci sans toutefois prendre en compte les demandes restantes des voitures ce qui en fait une heuristique statique. L'utilisation des ratios inverses permet de favoriser les options dont le filtrage relié aux contraintes de capacité a un impact sur le plus grand nombre de variables possible.

Finalement, à des fins de comparaisons, une heuristique aléatoire et une heuristique visant à lisser la production de véhicules ont été développées soit :

Random : Tire aléatoirement une catégorie parmi celles disponibles. Cette heuristique est statique étant donné que l'ordre de priorité des catégories est tiré aléatoirement une seule fois au début de la résolution.

Fréquence : Heuristique dynamique qui consiste à maintenir à jour, pour chacune des catégories de voitures du problème, une position de production au plus tôt. Celle-ci correspond à la prochaine position où cette catégorie de voitures peut être insérée dans la séquence de production sans violer aucune des contraintes implicites du problème. L'heuristique favorise ainsi les catégories de voitures se rapprochant le plus de la position courante d'instanciation.

Les heuristiques utilisées sont résumées au Tableau 2. Les calculs des différentes heuristiques utilisent $Demande_i$ qui établit la demande restante pour une

Heuristique	**Calcul**	**Type**	**Statique** ou **Dynamique**
Min_TauxUT	$\sum_{i \in Opt(Cat)} \frac{Demande_i}{nbVehicule * \frac{q_i}{p_i}}$	succeed-first	dynamique
Max_Vehicule	$Demande_{cat}$	succeed-first	dynamique
Min_Option	$nbOption_{cat}$	succeed-first	statique
Max_TauxUT	$\sum_{i \in Opt(Cat)} \frac{Demande_i}{nbVehicule * \frac{q_i}{p_i}}$	fail-first	dynamique
Min_Vehicule	$Demande_{cat}$	fail-first	dynamique
Max_Option	$nbOption_{cat}$	fail-first	statique
Max_P_Q	$\sum_{i \in Opt(Cat)} \frac{p_i}{q_i}$	fail-first	statique
Random	aléatoire	autre	statique
Fréquence	$ProdAuPlusTot_{cat} - PCour$	autre	dynamique

TAB. 2 – Heuristiques d'ordonnancement de valeurs

option i, $nbVehicule$ qui représente le nombre de voitures à produire, $nbOption$ qui définit le nombre d'options présentes dans une catégorie, $Opt(Cat)$ qui est une fonction permettant d'obtenir les options présentes dans une catégorie de voiture, $ProdAuPlusTot$ qui établit la position dans la séquence à laquelle il sera nécessaire de produire une catégorie afin d'éviter de violer une contrainte de capacité et $PCour$ qui représente la position courante dans la séquence ou la profondeur courante dans l'arbre de résolution. Dans la section suivante, la performance de ces différentes heuristiques d'ordonnancement de valeurs est établie pour la résolution du problème de *car-sequencing*.

4 Expériences numériques et résultats obtenus

4.1 Cadre expérimental

Les heuristiques introduites à la section précédente sont évaluées sur deux groupes d'instances de problème de *car-sequencing* de la libraire CSPLib. Le premier groupe comprend 70 instances de 200 voitures regroupées par taux d'utilisation moyen variant entre 60% à 90%. Pour toutes ces instances, il s'agit de problèmes satisfiables car il a été prouvé qu'il existait une séquence ne violant aucune contrainte de capacité. Le deuxième groupe comprend 9 instances de 100 voitures pour lesquels il a été démontré que 4 d'entre elles étaient satisfiables, 4 autres étaient non-satisfiables et une seule demeure avec un statut inconnu.

Pour la résolution de chacune de ces instances, l'algorithme a été restreint à un temps de 900 secondes. Notre algorithme a été exécuté sur une machine *Sunfire 6800* doté de processeurs *UltraSparc III* à 900 MHz avec 1 GO de mémoire vive. Les algorithmes ont été développés en langage C++ en utilisant le compilateur CC fournit par *Sun*. Notons aussi que ILOG Solver 6.0 a été exécuté sur un ordinateur doté d'un processeur Itanium 64 bits cadencé à 1,4 Ghz avec 4 Go de mémoire vive et utilise la contrainte *IlcSequence*, une contrainte développée pour les problèmes comportant des contraintes globales appliquées sur une séquence.

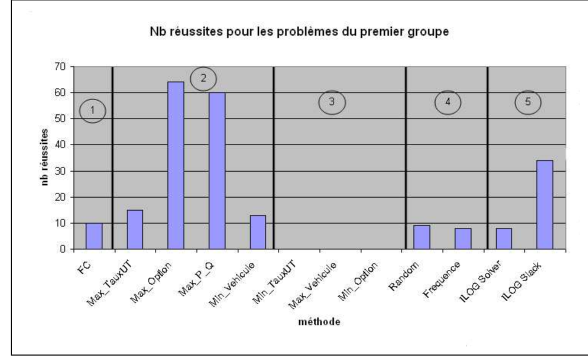

FIG. 3 – Résultats FC, problèmes du premier groupe (satisfiables).

Le lecteur pourra remarquer que cet ordinateur est plus puissant que celui utilisé par notre méthode de résolution. Des contraintes d'ordre technique ont rendu impossible les tests du FC sur la machine Itanium.

4.2 Résultats obtenus pour le premier groupe d'instances de problème

Le nombre d'instances du premier groupe résolues en utilisant les différentes heuristiques d'ordonnancement de valeurs est présenté à la Figure 3. Les étiquettes " F " et " S " précisent respectivement les schémas d'ordonnancement de valeurs *fail-first* et *succeed-first* utilisés par les heuristiques. La première section de la figure présente le nombre de problèmes résolus à l'aide du FC servant de base à chacune des autres heuristiques développées. La deuxième section présente les résultats obtenus à l'aide des heuristiques d'ordonnancement de valeurs de type *fail-first* tandis que le troisième groupe présente ceux de type *succeed-first*. La quatrième section montre la performance des heuristiques d'ordonnancement de valeurs *Fréquence* et *Random*. Pour sa part, la cinquième partie présente les résultats des essais effectués avec ILOG Solver 6.0. À cet effet, une heuristique tirée de Régin et Puget [13] est utilisée pour établir un ordre de priorité pour chacune des options d'un problème. L'Équation 2 définit la marge excédentaire pour chaque option i et favorise ainsi les catégories de voiture possédant la marge excédentaire minimale. Il est à noter que cette méthode, nommée ILOG *Slack*, est une méthode d'ordonnancement de valeurs de type *fail-first*.

$$slack_i = nbVehicule - p_i * \frac{Demande_i}{q_i} \qquad (2)$$

On constate que l'utilisation de la méthode de filtrage de domaines (FC) a permis la résolution de seulement 10 des 70 instances de problème satisfiables tan-

dis que l'utilisation du logiciel ILOG Solver 6.0 a, pour sa part, permis la résolution de seulement 8 de ces instances. L'utilisation d'heuristiques d'ordonnancement de valeurs permet d'augmenter le nombre de problèmes résolus tant lors de l'utilisation du FC qu'avec ILOG Solver. En effet, les heuristiques de type *fail-first*, soit *Max_ TauxUT*, *Max_ Option*, *Max_ P_ Q* et *Min_ Vehicule*, sont parvenues à résoudre respectivement 15, 64, 60 et 13 instances de problème. Par contre, les heuristiques de type *succeed-first*, soit *Min_ TauxUT*, *Min_ Option* et *Max_ Vehicule*, n'ont réussi à résoudre aucune des instances de problème. L'heuristique *Random* a permis de résoudre 9 instances de problème et, pour sa part, l'heuristique *Fréquence* a permis la résolution de 8 instances de problème. De plus, l'utilisation de l'heuristique *Slack*, lors de la résolution du problème par ILOG Solver, a permis de résoudre 34 instances de problème.

On constate donc, comme les travaux de Smith [15] l'avaient souligné, que les schémas d'ordonnancement de valeurs *fail-first* obtiennent de meilleurs résultats que ceux de type *succeed-first*. En particulier, les heuristiques *Max_ Option* et *Max_ P_ Q* obtiennent des résultats intéressants pour ces problèmes. Ces résultats s'expliquent par le fait que les schémas *fail-first* permettent de favoriser les véhicules les plus contraints en début de séquence et d'ainsi supprimer le plus rapidement possible des valeurs de l'espace de recherche de solutions. De plus, la globalité des contraintes de capacité fait en sorte que les choix effectués en début de séquence ont un impact sur tous les choix de la séquence. Le fait de garder les véhicules les moins contraints en fin de séquence s'avère être un choix judicieux puisque les catégories de véhicules disponibles sont moins nombreuses étant donné que plusieurs d'entre elles sont éliminées par le filtrage sur les catégories vides. Les résultats obtenus par les heuristiques statiques permettent également de conclure que celles-ci favorisent la recherche en nécessitant moins de calculs que les heuristiques dynamiques.

Le Tableau 3 présente les résultats détaillés de l'utilisation du filtrage de domaines et de l'heuristique d'ordonnancement de valeurs *Max_ Option* pour la résolution des problèmes du premier groupe. Ceux-ci sont regroupés par taux d'utilisation moyen des options et on compte 10 instances de problème pour chacun d'eux. Les résultats présentés correspondent aux résultats moyens obtenus. Notons que la colonne *Total* temps indique le temps nécessaire pour résoudre le problème et, dans le cas d'un problème non résolu, le temps maximum passé à chercher une solution au problème. La colonne *Temps pr. max* indique le temps nécessaire pour atteindre cette profondeur dans l'arbre de résolution. La colonne *Prof* indique le nombre de positions de la séquence 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 recherche dans l'arbre de résolution. *Noeuds visités* sont le nombre de noeuds de l'arbre de résolution visités pour atteindre la profondeur maximale et 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.

On note une nette amélioration de la performance du FC avec l'ajout de l'heuristique d'ordonnancement de valeurs *Max_ Option*. En effet, le nombre d'instances de problème résolus est passé de 10 à 64. De plus, on constate que le temps nécessaire pour atteindre les profondeurs maximales est grandement inférieur au temps total de recherche. L'heuristique permet donc d'atteindre rapidement une profondeur élevée dans l'arbre de résolution, soit en moyenne 199 voitures de la séquence de production, en un temps moyen très rapide de 22 secondes. On peut aussi constater que, pour les instances de problème de taux d'utilisation de 90%, le nombre de noeuds visités, de retours en arrière et d'appels à la procédure de filtrage est beaucoup plus petit que pour les groupes d'instances de taux d'utilisation inférieur.

Prob	Total Temps	Temps pr max.	Prof.	BT	Noeuds visités	Filtrage	Réussites
60	95	9	199	297 907	596 013	327 293	9
65	138	58	199	2 162 495	4 325 200	2 461 373	9
70	121	53	200	1 965 166	3 930 532	2 075 376	9
75	104	14	198	427 421	855 039	427 705	9
80	94	8	200	243 484	487 167	249 704	9
85	101	11	200	343 656	687 510	431 743	9
90	1	1	200	271	743	492	10
Moy :	93	22	199	777 200	1 554 601	853 384	

Tab. 3 – Résultats de l'heuristique *Max_ Option* pour les problèmes du premier groupe (satisfiables).

Le Tableau 4 présente, pour sa part, les résultats détaillés obtenus par ILOG Solver 6.0 avec l'ajout de l'heuristique d'ordonnancement de valeurs *Slack*. Notons, dans un premier, que l'ajout de cette heuristique permet de faire passer le nombre d'instances résolues de 8 à 34. On peut aussi constater que le temps moyen nécessaire pour la résolution est supérieur pour ILOG Solver en comparaison au FC. Cet élément est d'autant plus important si l'on considère que l'ordinateur utilisé pour la résolution par ILOG Solver possède une puissance de calcul supérieure à celui utilisé pour la résolution par le FC. Le nombre moyen de retour en arrières (BT) est de beaucoup inférieur pour ILOG Solver que pour le FC et peut s'expliquer par le fait que Solver permet un filtrage de domaines plus efficace en utilisant un principe de diminution de domaines de type MAC plutôt qu'un simple filtrage de type FC.

Cependant, cette stratégie nécessite un temps de calcul plus important. Ce filtrage fait également en sorte que le nombre de noeuds visités (*Nb_ choice_ pt*) est inférieur pour Solver comparativement au FC. L'utilisation par Solver d'un filtrage de type MAC ne lui procure aucun avantage pour la résolution des problèmes du premier groupe. Au contraire, le FC avec l'heuristique *Max_ Option* est la plus performante sur ce groupe de problèmes.

Problème	BT	nb choice pt	Total temps	Réussites
60	38 851	39 055	456	5
65	63 460	63 667	720	3
70	55 033	55 250	721	2
75	38 318	38 568	454	5
80	17 539	17 797	365	6
85	29 315	29 593	454	5
90	7 623	7 937	229	8
Moy :	35 734	35 981	486	

TAB. 4 – Résultats de ILOG Solver 6.0 avec l'heuristique *Slack* pour les problèmes du premier groupe (satisfiables).

4.3 Résultats obtenus pour le second groupe d'instances de problème

Les Tableaux 5, 6 et 7 présentent les résultats pour les 9 problèmes du deuxième groupe. La dernière colonne de ces tableaux précise si l'heuristique a réussi à solutionner le problème. On note que, comme pour les problèmes du premier groupe, l'ajout de l'heuristique *Slack* permet d'augmenter le nombre d'instances de problème résolues par ILOG Solver. On constate également que, pour ce groupe d'instances, ILOG Solver 6.0 sans l'utilisation d'heuristiques surpasse les résultats obtenus par le FC. La méthode de filtrage développée par ILOG détecte donc mieux les situations d'inconsistance et élimine ainsi de plus grandes régions de l'espace de solutions.

On note également que, pour ce deuxième groupe, ILOG Solver avec l'heuristique *Slack* surpasse le FC avec l'heuristique *Max_ Option* en solutionnant 6 des 9 problèmes. En effet, 3 des 4 instances connues comme satisfiables et 3 des 4 instances connues comme non-satisfiables sont ainsi solutionnées. Plus particulièrement, cette heuristique a permis de prouver la non satisfiabilité du problème 36_92, problème pour lequel l'état était inconnu auparavant. Cependant, l'état de l'instance 21_90 demeure toujours indéterminé. Il est également possible de constater que le temps moyen nécessaire pour la recherche a augmenté mais demeure toutefois faible pour les problèmes satisfiables.

Les résultats obtenus par le FC montrent que l'heuristique d'ordonnancement de valeurs *Max_ Option* permet d'atteindre, en moyenne, la position 84 de la séquence. Plus spécifiquement, cette profondeur atteint même, en moyenne, 94 pour les 4 problèmes satisfiables de ce groupe. On constate également que les profondeurs maximales sont atteintes très rapidement soit en moyenne en 58 secondes. L'heuristique d'ordonnancement de valeurs utilisée permet donc une descente rapide mais le filtrage développé ne permet pas la détection d'inconsistances et l'élimination de régions de l'espace de solutions assez tôt durant la descente.

Prob	Total Temps	Temps pr max.	Prof.	BT	Noeuds visités	Filtrage	Réussites
16_81[1]	900	3	91	38 960	78 011	42 079	non
26_82[1]	900	0	89	31	151	132	non
41_66[1]	900	0	95	9 443	18 981	11 829	non
4_72[1]	516	516	100	15 856 500	31 713 100	17 445 000	oui
10_93[2]	900	1	74	8 292	16 658	10 372	non
19_71[2]	900	0	88	15 751	31 590	17 410	non
21_90[3]	900	0	90	0	90	96	non
36_92[2]	900	0	64	32	128	115	non
6_76[2]	900	0	69	0	69	70	non
Moy :	857	58	84	1 769 890	3 539 864	1 947 456	

TAB. 5 – Résultats de l'heuristique *Max_ Option* pour les problèmes du deuxième groupe.

Problème	BT	nb choice pt	Total temps	réussites
16_81[1]	87 485	87 614	900	non
26_82[1]	120 382	120 477	900	non
41_66[1]	348 574	348 661	900	non
4_72[1]	15 177	15 274	93	oui
10_93[2]	41	40	1	oui
19_71[2]	74 420	74 537	900	non
21_90[3]	101 197	101 251	900	non
36_92[2]	138 806	138 879	900	non
6_76[2]	83 066	83 146	900	non
Moy :	107 683	107 764	710	

TAB. 6 – Résultats de ILOG Solver 6.0 pour les problèmes du deuxième groupe.

Problème	BT	nb choice pt	Total temps	réussites
16_81[1]	58 675	58 730	900	non
26_82[1]	6 944	7 071	36	oui
41_66[1]	0	111	1	oui
4_72[1]	0	110	1	oui
10_93[2]	758	757	11	oui
19_71[2]	35 214	35 271	900	non
21_90[3]	85 509	85 573	900	non
36_92[2]	13 731	13 730	153	oui
6_76[2]	9 355	9 354	121	oui
Moy :	23 354	23 412	336	

TAB. 7 – Résultats de ILOG Solver 6.0 avec l'heuristique *Slack* pour les problèmes du deuxième groupe.

[1]Instance de problème satisfiable
[2]Instance de problème non-satisfiable
[3]Instance de problème indéterminé

5 Conclusion et perspectives

Dans cet article, une comparaison de la performance de différentes heuristiques d'ordonnancement de valeurs permettant de guider la recherche de solutions pour le problème de *car-sequencing* a été présentée. Les résultats obtenus ont démontré la supériorité d'une stratégie de type *fail-first* par rapport à une stratégie *succeed-first*. De plus, les heuristiques statiques ont permis une meilleure exploration de l'espace de solutions et ont résolu un plus grand nombre d'instances de problème.

La comparaison effectuée entre l'approche de filtrage de domaines développée (FC) et la stratégie utilisée par ILOG Solver (MAC) a fait ressortir un avantage pour l'approche FC en ce qui concerne les problèmes du premier groupe d'instances et pour Solver en ce qui concerne le second groupe d'instances. En effet, l'approche par FC permet d'explorer un plus grand nombre de noeuds de l'espace de solutions et ainsi favorise la recherche de solutions. Solver, pour sa part, mise sur un filtrage plus efficace mais nécessitant un effort de calcul plus important ce qui diminue le temps de calcul alloué à l'exploration et explique les résultats observés sur les instances difficiles ou non-statisfiables.

De plus, les expériences numériques réalisées ont également permis de solutionner l'instance 36_92 à l'aide de ILOG Solver et de l'heuristique *Slack*. Cette instance n'avait jamais été prouvée non-satisfiable jusqu'à ce jour et représente une contribution importante de ces travaux.

Dans le cadre de travaux futurs, la méthode de filtrage de domaines sera améliorée afin de favoriser la détection d'inconsistance plus tôt et éventuellement parvenir à résoudre un plus grand nombre d'instances de problème et, en particulier, celles du deuxième groupe. De plus, l'utilisation de machines parallèles s'avère également une solution intéressante pour la résolution de ce type de problème et des travaux en ce sens sont en cours de développement. Une hybridation entre la méthode proposée dans cet article et une méthode d'optimisation des conflits induits par les contraintes représente une autre avenue de recherche intéressante. Finalement, des essais numériques seront effectués sur des instances de problème de natures différentes. Entre autre, les groupes d'instances de problème suggérées par Gravel et *al* [7] comportant des problèmes de 200, 300 et 400 véhicules à ordonnancer seront utilisés.

Remerciements

Nous tenons à souligner l'apport de "Romeo"[1], le centre de calcul de l'Université de Reims Champagne-Ardenne (FRANCE), dans le cadre des essaies numériques effectués.

Références

[1] C. Bessière, P. Meseguer, E. C. Freuder and J. Larrosa. On Forward Checking for Non binary Constraint Satisfaction. In *Principles and Practice of Constraint Programming*, pages 88–102. 1999

[2] A. Davenport, E. Tsang, K. Zhu and C. Wang. GENET : a connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *American Association for Artificial Intelligence*, 1994

[3] A. Davenport and E. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. In *14th UK Planning and Scheduling Special Interest Group Workshop*, Wivenhoe Park, Colchester, UK. 1995

[4] M. Dincbas, H. Simonis and P. van Hentenryck Solving the car-sequencing problem in logic programming. In *8th European Conference on Artificial Intelligence*, pages 290–295, 1980

[5] I. Gent. Two Results on Car-sequencing Problems. In *Report University of Strathclyde*, APES-02-98, 7. Glasgow, Scotland. 1998

[6] J.Gottlieb, M. Puchta and C. Solnon. A Study of Greedy, Local Search and Ant Colony Optimization Approaches for Car Sequencing Problems. In *Applications of evolutionary computing* 2003

[7] M. Gravel, C. Gagné and W. L. Price. Review and comparaison of three method for the solution of the car-sequencing problem. In *Journal of The Operational Research Society*, 2004

[8] M. Gravel, C. Gagné and W. L. Price. Solving real car sequencing problems with ant-colony optimization. 2005, *to appear*

[9] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. In *Artificial Intelligence*, 14 :263–313, 1980

[10] T. Kis. On the complexity of the car sequencing problem. In *Operation Research Letters*, 32 :331–335, 2004

[11] B.D. Parrello, W.C. Kabat and L. Wos. Job-shop scheduling using automated reasoning : A case of the car sequencing problem. In *Journal of automated reasoning*, 1986

[12] P. Prosser. Forward Checking with Backmarking. In *Constraint Processing, Selected Papers*, pages 185–204. 1995

[1] http ://www.univ-reims.fr/Calculateur

[13] J.C. Régin and J.F. Puget. A filtering Algorithm for Global Sequencing Constraints. In *Constraint Programming*, pages 32–46. 1997

[14] C. Solnon. Ant-P-solveur : un solveur de contraintes à base de fourmis artificielles. In *Journées Francophones sur la Programmation Logique et par Contraintes*, 2000

[15] B.M. Smith. Succeed-first or Fail-first : A Case Study in Variable and Value Ordering. In *Report, University of Leeds*, 96.26, 11. 1996

[16] T. Warwick and E. Tsang. Tackling car sequencing problems using a generic genetic algorithm. In *Evolutionary Computation MIT Press*, 3 : 267-298. 1995