UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

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

SÉBASTIEN NOËL

MÉTAHEURISTIQUES HYBRIDES POUR LA RÉSOLUTION DU PROBLÈME D'ORDONNANCEMENT DE VOITURES DANS UNE CHAÎNE D'ASSEMBLAGE AUTOMOBILE

OCTOBRE 2007

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

La littérature scientifique propose une grande variété de stratégies pour la résolution des problèmes d'optimisation combinatoire (*POC*). Ces problèmes sont d'une grande complexité et demandent des méthodes évoluées pour les résoudre. Les algorithmes exacts, comme la programmation linéaire en nombres entiers (*PLNE*) à l'aide de l'algorithme *Branch and Bound* (*B&B*), arrivent à trouver une solution optimale pour certaines instances de problèmes. Par contre, plus la taille du problème à résoudre est grande, plus ces algorithmes ont de la difficulté à en venir à bout. Les métaheuristiques représentent alors une alternative intéressante pour trouver une solution de qualité acceptable dans des délais très courts. Toutefois, il est impossible de garantir qu'une métaheuristique trouvera la solution optimale d'un problème. Parmi ces méthodes, on retrouve l'optimisation par colonies de fourmis (*OCF*), qui a su faire ses preuves pendant les dernières années pour la résolution de différents problèmes d'optimisation combinatoire. Une autre avenue consiste à créer des algorithmes hybrides. L'objectif principal de ce mémoire est de proposer trois algorithmes hybridant un *OCF* et la *PLNE* pour résoudre le problème d'ordonnancement de voitures (*POV*).

Le *POV* est un *POC* qui consiste à déterminer dans quel ordre placer un ensemble de voitures à produire sur une chaîne d'assemblage en se soumettant à un ensemble de contraintes. On cherche parfois la séquence minimisant le nombre de conflits, où un conflit représente une surcharge de travail occasionnée à un poste particulier de l'atelier de montage par l'arrivée successive de plusieurs voitures similaires, ou encore minimisant le nombre de changements de couleurs à l'atelier de peinture. Pour simplifier le problème, on ne s'attardera qu'aux contraintes liées à l'atelier de montage où sont installées les différentes options des voitures. Cette version théorique du *POV* que l'on retrouve dans la littérature est une simplification du problème industriel. Différentes méthodes ont été proposées pour solutionner ce problème. Celles qui attirent notre attention sont l'*OCF* et la *PLNE*. On cherchera, dans ce mémoire, à concevoir des approches hybrides exploitant les forces de ces deux approches. Il sera également possible de comparer la performance des algorithmes hybrides avec les résultats obtenus avec l'*OCF* pour établir l'apport de telles hybridations.

Le premier algorithme hybride proposé consiste à créer un sous-problème à partir de la meilleure solution de chaque cycle de l'OCF et de résoudre ce sous-problème avec le *B&B*. Cette méthode ne s'est pas avérée très performante, car aucune intensification n'est effectuée sur une solution. Le second algorithme tente de combler cette lacune en appelant le *B&B* de manière répétitive à un intervalle régulier de cycles de l'*OCF*. Cet appel répété du *B&B* représente, en fait, une recherche locale exacte (*RLE*). Pour l'ensemble des problèmes utilisés pour tester cette hybridation, des résultats de qualité légèrement supérieure ou égale à l'*OCF*, intégrant une recherche locale, ont été obtenus pour environ deux problèmes sur trois. On peut en dire autant de la troisième hybridation proposée, qui

consiste, dans un premier temps, à exécuter l'*OCF* et à fournir la meilleure solution trouvée comme solution de départ à la *RLE*.

Les objectifs fixés dans cette recherche ont été atteints en concevant des méthodes de résolution hybrides, adaptées au *POV*, combinant une métaheuristique et une méthode exacte. On avait aussi pour but d'établir la performance des méthodes hybrides face à leurs contreparties singulières. En règle générale, les hybridations parviennent à donner des résultats de qualité équivalente à celle des résultats de l'*OCF* avec recherche locale mais avec un coût en temps d'exécution. Il s'agit tout de même d'une conclusion réjouissante puisque des améliorations pourraient être apportées à ces algorithmes pour les rendre encore plus performants. On a aussi exploré quelques façons de créer des sous-problèmes plus faciles à résoudre par un algorithme exact. Ceci ouvre donc une porte à une autre approche de la résolution de *POC*.

REMERCIEMENTS

En premier lieu, je désire offrir mes plus grands remerciements à mon directeur de recherche, M. Marc Gravel. Son aide et son soutien ont été très utiles tout au long de ce projet. Grâce à son expérience, sa disponibilité et sa grande patience, il a été un guide hors pair dans la complétion de cette recherche. Merci infiniment.

Je remercie également Mme Caroline Gagné, qui m'a donné de judicieux conseils et apporté un grand support. Merci aux autres enseignants du Département d'informatique et de mathématique de l'UQAC, car vos connaissances, votre savoir-faire et votre professionnalisme se sont avérés une grande source d'inspiration.

Je ne peux passer sous silence la participation de M. Roger Villemaire et M. Djamal Rebaïne, qui, par leurs commentaires constructifs, m'ont permis d'améliorer la qualité de ce mémoire.

Je veux aussi remercier mes collègues du Groupe de Recherche en Informatique de l'UQAC. Merci à Aymen Sioud, pour les suggestions qu'il a apportées à la revue de littérature et son soutien moral. Merci aussi à Arnaud Zinflou, pour l'aide qu'il m'a offerte à tous les moments où j'en ai eu de besoin. Merci aux autres étudiants du GRI, et en particulier à Jérôme Lambert, Pierre Delisle, Anouar Taleb, Jean-Luc Allard et sa conjointe Annie-Marie. Ce fut un plaisir de vous côtoyer quotidiennement. Je vous souhaite la meilleure des chances dans vos projets futurs.

Je profite de l'occasion pour remercier mon parrain et ma marraine, Rémi et Carole, de m'avoir aidé à réaliser mes rêves. Jamais je n'oublierai ce que vous avez fait pour moi.

Finalement, je veux remercier très distinctement mes parents, Alain et Michelle, ainsi que mon frère, Olivier. Aussi loin que je puisse me souvenir, vous m'avez toujours encouragé à terminer ce que j'entreprends et aidé dans les moments les plus difficiles. Merci infiniment.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1 :

INTRODUCTION

On retrouve des problèmes d'optimisation dans presque tous les secteurs économiques, tels la production manufacturière, les transports et la gestion du personnel. Dans une usine, par exemple, on veut chercher à ordonnancer la chaîne de production pour qu'elle satisfasse différents objectifs comme terminer le travail le plus rapidement possible ou encore minimiser le nombre de commandes livrées en retard aux différents clients. Une compagnie de livraison peut chercher à améliorer son service de distribution en minimisant la distance à parcourir pour chacun de ses livreurs. Les grandes institutions scolaires doivent créer des horaires pour les étudiants en assignant des locaux et des enseignants aux différents cours. Dans ce cas, l'objectif est de minimiser le nombre de conflits d'horaires de sorte qu'un étudiant ne se retrouve pas avec deux cours à une même période. La résolution de ce type de problèmes vient mettre à profit différents domaines scientifiques tels que l'informatique, les mathématiques et la recherche opérationnelle.

Un problème d'optimisation combinatoire (*POC*) comprend un ensemble fini de solutions, où chaque solution doit respecter un ensemble de contraintes relatives à la nature du problème. On associe à chaque solution une valeur, nommée valeur de l'objectif, qui est évaluée à l'aide d'une fonction, la fonction objectif. La solution optimale est celle dont la valeur de l'objectif est la plus petite (grande) dans un contexte de minimisation (maximisation) parmi l'ensemble de solutions. Certains types de *POC* consistent à ordonnancer un ensemble d'éléments. Plus le nombre d'éléments à ordonnancer est grand, plus il y a de possibilités de solutions. Par exemple, un problème ayant 200 éléments à ordonnancer compte 200! ($7{,}88 \times 10^{374}$) solutions. Donc, même l'ordinateur le plus

puissant ne pourra pas, dans des temps réalistes, énumérer toutes ces solutions pour trouver la solution optimale.

Deux types d'algorithmes sont fréquemment appliqués pour résoudre les *POC* : les algorithmes exacts et les heuristiques. Les algorithmes exacts garantissent de trouver la solution optimale d'un *POC*, mais celle-ci demeure parfois introuvable, faute de temps. Dans la catégorie des heuristiques, les métaheuristiques se distinguent tout particulièrement dans la résolution des *POC*. Plusieurs d'entre elles sont inspirées de phénomènes naturels, comme des propriétés que possède le métal lorsqu'il se refroidit, le fonctionnement de la mémoire dans le cerveau humain, la théorie de l'évolution de *Darwin* ainsi que le comportement d'une colonie de fourmis. Le recuit simulé (*RS*), la recherche avec tabous (RT), l'algorithme génétique (*AG*) et l'optimisation par colonies de fourmis (*OCF*) sont des exemples de métaheuristiques qui parviennent à trouver des solutions de bonne qualité, parfois même optimales, dans des délais raisonnables.

Il existe une autre approche pour résoudre les *POC* : il s'agit des algorithmes hybrides. Ceux-ci combinent les forces de deux ou plusieurs algorithmes pour en faire un seul. On peut combiner les algorithmes de différentes façons : en les exécutant soit en série, soit en parallèle ou en incorporant un algorithme à un autre, donnant lieu à une méthode maître et esclave. La présente recherche vise à proposer différentes approches hybrides de résolution, pour le problème d'ordonnancement de voitures, en unissant, dans un même algorithme, l'*OCF* proposé par Gravel, Gagné *et al.* [42] pour ce problème et le modèle linéaire en nombres entiers proposé par les mêmes auteurs et résolu par l'algorithme *B&B*.

Le problème d'ordonnancement de voitures (*POV*) consiste à déterminer dans quel ordre fabriquer les différentes voitures sur une chaîne de montage, en se soumettant à un ensemble de contraintes. On cherche parfois la séquence minimisant le nombre de conflits à l'atelier de montage, où un conflit représente une surcharge de travail occasionnée, à un poste particulier de l'atelier, par l'arrivée successive de plusieurs voitures similaires, ou encore minimisant le nombre de changements de couleurs à l'atelier de peinture. Pour simplifier le problème, on ne s'attardera qu'aux contraintes liées à l'atelier de montage. C'est dans l'atelier de montage que les différentes options d'une voiture sont installées, telles que le toit ouvrant, le climatiseur, etc.

L'organisation de ce mémoire est la suivante. On présente, au Chapitre 2, les différentes méthodes utilisées pour la résolution de problèmes d'optimisation combinatoire. Dans un premier temps, on décrit le fonctionnement des principales méthodes exactes ainsi que les heuristiques les plus connues, telles que la recherche locale, le recuit simulé, l'algorithme génétique, la recherche avec tabous et l'optimisation par colonies de fourmis. Par la suite, on présente la revue de la littérature concernant les algorithmes hybrides ainsi que deux façons de classifier ces méthodes. La classification de Talbi [80] classe les algorithmes selon leurs différentes caractéristiques, tandis que celle de Puchinger et Raidl [66] se concentre sur la classification des algorithmes hybridant une méthode heuristique et une méthode exacte. Enfin, un bref survol du problème d'ordonnancement de voitures et de la littérature portant sur les méthodes utilisées pour le résoudre précède la présentation des objectifs de la recherche.

Le Chapitre 3 décrit de façon plus détaillée le problème d'ordonnancement de voitures et présente, par la suite, les méthodes qui seront utilisées pour former un algorithme hybride. Dans un premier temps, on explique le fonctionnement de l'*OCF* proposé par Gravel, Gagné *et al.* [42] et on présente les résultats obtenus par celui-ci. Le modèle linéaire à nombres entiers proposé par les mêmes auteurs est également décrit, tout comme les résultats obtenus sur les divers ensembles de problèmes. Par la suite, on présente le développement d'une méthode de construction de sous-problèmes pour leur résolution par la *PLNE*. L'idée des sous-problèmes est amenée pour faciliter la résolution, comparativement au problème complet. Pour créer ces sous-problèmes, différentes stratégies sont explorées. La performance de chacune de ces stratégies est évaluée, pour ne conserver que celles offrant les meilleurs résultats selon une série de paramètres. Ces paramètres sont, par exemple, le temps accordé au *PLNE* pour résoudre les sous-problèmes, ainsi que la taille de ces derniers. Enfin, quatre approches différentes sont mises à l'épreuve. La première est nommée Recherche Locale Exacte (*RLE*). Elle consiste à exécuter une recherche locale traditionnelle utilisant les sous-problèmes et la *PLNE* pour résoudre ces sous-problèmes. Les trois méthodes suivantes sont l'hybridation intégrative à chaque cycle de l'OCF, l'hybridation intégrative à fréquence espacée et l'hybridation à relais. Les résultats de ces approches hybrides sont comparés sur trois ensembles de problèmes.

Le Chapitre 4 vient conclure notre travail, en récapitulant les objectifs établis au départ et en montrant comment ils ont été atteints. On ouvre aussi la voie vers d'autres

avenues de recherche, par exemple l'hybridation du *PLNE* avec une métaheuristique différente de l'*OCF*.

CHAPITRE 2 :

MÉTHODES DE RÉSOLUTION DE PROBLÈMES D'OPTIMISATION COMBINATOIRE

2.1 Introduction

Selon Dorigo et Stützle [29] et Hao, Galinier *et al.* [46], un problème d'optimisation combinatoire (*POC*) est un problème de minimisation ou de maximisation auquel on associe un ensemble d'instances. Chaque instance possède un ensemble fini de solutions candidates S, une fonction objectif f et un ensemble fini de contraintes Ω qui dépendent de la nature du problème. Les solutions du sous-ensemble $X \subseteq S$, dites réalisables, sont celles qui se soumettent aux contraintes de Ω. La fonction objectif f donne, à chaque solution $s \in X$, la valeur de l'objectif $f(s)$, qui peut être réelle ou entière selon la nature du problème. Un problème de minimisation consiste à trouver, pour une instance donnée, une solution $s^* \in X$ telle que $f(s^*) \leq f(s)$, pour toute solution $s \in X$. On dit alors que s^* est une solution optimale (ou l'optimum global) de cette instance de problème. Pour un problème de maximisation, on veut trouver une solution $s^* \in X$ telle que $f(s^*) \geq f(s)$, pour toute solution $s \in X$.

Les problèmes d'optimisation combinatoire sont présents dans plusieurs domaines d'applications industrielles, économiques et scientifiques. Le problème le plus connu est sans doute celui du voyageur de commerce, qui consiste à visiter chacune des villes d'un ensemble une et une seule fois en empruntant le plus court chemin. On les retrouve aussi dans des applications pratiques telles que la confection d'horaires, l'ordonnancement de tâches dans une usine, l'acheminement de véhicules, l'emballage et le découpage, l'affectation de tâches à des employés, la couverture de réseaux sans fil, la distribution de

fréquences radio, l'ordonnancement de chaînes de montage automobile et plusieurs autres. Ce sont, en général, des problèmes faciles à définir, mais difficiles à résoudre.

Avant de s'avancer dans la théorie sur la complexité des problèmes, il est nécessaire de savoir qu'un *POC* peut être transformé en un problème de décision. Un problème de décision est un problème où le résultat est « oui » ou « non », exclusivement. Dans le cas d'un problème de minimisation, le problème de décision correspondant serait « existe-t-il une solution s telle que $f(s) \leq a$ », où a est un paramètre du problème considéré. Ce problème de décision ne demande pas de trouver la valeur d'une solution inférieure ou égale à a, mais seulement d'indiquer s'il en existe une.

La théorie de la complexité distingue deux classes de problèmes : les problèmes de la classe *P* et ceux de la classe *NP*. Les problèmes appartenant à la classe *P* sont ceux dont le problème de décision correspondant donne la réponse correcte (« oui » ou « non ») en un temps polynomial. Les problèmes de la classe *NP* sont ceux dont on peut obtenir le résultat « oui » à leur problème de décision selon un algorithme non déterministe à temps polynomial (Dorigo et Stützle [29]). Les algorithmes non déterministes sont capables d'effectuer, par un don de clairvoyance, un choix judicieux dans un ensemble d'éléments. Ils ne peuvent pas être implantés sur ordinateur et sont plutôt d'intérêt théorique.

Une réduction en temps polynomial est une procédure qui transforme un problème en un autre problème via un algorithme à temps polynomial. Un problème *pr* est *NP*-difficile si chacun des problèmes dans *NP* peut se réduire en un temps polynomial à *pr*. Donc, un problème *NP*-difficile est au moins aussi difficile que n'importe quel problème de la classe *NP*. Par contre, tous les problèmes *NP*-difficiles n'appartiennent pas

nécessairement à la classe *NP*. Les *POC* appartiennent à la classe *NP*-difficile et il est possible de les transformer en problèmes de décision. Ces derniers appartiennent à la classe *NP*-complet. Si on trouve un algorithme déterministe à temps polynomial pour résoudre un problème *NP*-complet, alors tous les problèmes *NP*-complets peuvent être résolus en temps polynomial, ainsi que les problèmes de la classe *NP*. Un tel algorithme n'a pas encore été trouvé et la communauté scientifique accepte l'hypothèse $P \neq NP$ (Dorigo et Stützle [29]).

Les *POC* sont des problèmes *NP*-difficiles; il existe une multitude de méthodes pour les résoudre. La section suivante présente les différentes méthodes de résolution d'un *POC*.

2.2 Les méthodes de résolution de POC

Les méthodes de résolution de *POC* se subdivisent en trois groupes : les méthodes exactes, les méthodes heuristiques et les méthodes hybrides. Examinons plus en détails les principales méthodes dans chacun de ces groupes.

2.2.1 Les méthodes exactes

Les méthodes (ou algorithmes) exactes garantissent de trouver la solution optimale et de prouver son optimalité pour toutes les instances d'un *POC* (Puchinger et Raidl [66]). Ils permettent aussi de fournir des informations sur les bornes inférieures/supérieures de la solution optimale d'un *POC* dans le cas où l'algorithme est arrêté avant la fin (Dumitrescu et Stützle [31]). Par contre, le temps nécessaire pour trouver la solution optimale d'un *POC* augmente en général fortement avec la taille du problème. De façon pratique, seuls les

problèmes de petites et moyennes tailles peuvent être résolus de façon optimale par des algorithmes exacts. De plus, pour certains problèmes, la consommation de mémoire de ces algorithmes peut être très grande et peut parfois entraîner l'arrêt prématuré de l'application informatique. Parmi les algorithmes exacts, on retrouve l'algorithme du simplexe, le *branch-and-bound* (*B&B*), le *branch-and-cut* (*B&C*), le *branch-and-price* (*B&P*), le *branch-and-cut-and-price* (*B&C&P*), la programmation dynamique, les méthodes basées sur la relaxation lagrangienne et la programmation par contraintes. Les paragraphes suivants décrivent le fonctionnement de chacune de ces méthodes.

La programmation linéaire peut être utilisée pour résoudre de façon exacte les *POC*. En effet, lorsqu'un *POC* peut être modélisé en un programme linéaire, il peut alors être résolu, par exemple, à l'aide de l'algorithme du simplexe (Cormen, Leiserson *et al.* [21]). Lorsqu'on a la contrainte supplémentaire voulant que les valeurs des variables de décision soient entières, on a un programme linéaire en nombres entiers (*PLNE*).

On peut résoudre de façon exacte certaines instances de *PLNE* à l'aide de l'algorithme du *B&B*. Le *B&B*, selon Bouzgarrou [13], utilise le branchement pour diviser l'ensemble des solutions en sous-ensembles et l'évaluation pour mettre des bornes supérieures ou inférieures sur les solutions. La première étape de l'algorithme consiste à résoudre la relaxation du *PLNE* en éliminant la contrainte d'intégralité. Si la solution du programme relaxé est entière, alors elle est aussi une solution optimale du programme non relaxé. Sinon, une opération de branchement est exécutée en choisissant une variable de décision x non entière dans la solution optimale précédemment obtenue. On sépare en deux l'ensemble de solutions en ayant, d'un côté, les solutions dont x prend une valeur

supérieure à celle dans la solution optimale précédente et, de l'autre côté, les solutions où sa valeur est inférieure. Les nouveaux nœuds sont alors évalués et on coupe les branches qui sont jugées inintéressantes. Les branches sont coupées lorsque le programme linéaire n'est pas réalisable, lorsque la solution est de moindre qualité que la meilleure solution trouvée ou lorsque la solution est entière. Le *B&B* s'arrête quand il n'y a plus de nœuds à évaluer. La meilleure solution entière trouvée est la solution optimale du problème. Il est à noter que le choix du prochain nœud à évaluer ainsi que celui de la variable de branchement influence grandement la performance du *B&B*.

Barnhart, Johnson *et al.* [8] décrivent l'algorithme de *B&C* en spécifiant qu'il s'agit d'une généralisation du *B&B*. On peut également utiliser cette technique lorsque le nombre de contraintes est élevé. Des classes de contraintes, sous forme d'inégalités, sont abandonnées pour relaxer le *PLNE*. Ces contraintes sont retirées parce qu'elles sont souvent nombreuses et que la plupart ne sont pas liées à une solution optimale. Si la solution optimale du programme linéaire relaxé est infaisable, un sous-problème, nommé problème de séparation, est résolu pour tenter d'identifier les classes d'inégalités violées. S'il y en a, quelques contraintes sont réinsérées dans le programme linéaire qui est résolu à nouveau. Ceci fait en sorte qu'on coupe certaines solutions non réalisables. Les branchements ont lieu lorsque aucune inégalité n'est violée. Le *B&C* permet la séparation et les coupures dans tout l'arbre *B&B*.

Barnhart, Johnson *et al.* [8] poursuivent en disant que le *B&P* est similaire au *B&C*, sauf qu'il est orienté sur la génération de colonnes plutôt que sur la génération de lignes. Cette technique est utile lorsque le nombre de variables est grand. Le *B&P* enlève des

colonnes du programme linéaire pour en faire une relaxation. Une fois le nouveau problème résolu, on vérifie son optimalité en résolvant le problème de *pricing* pour identifier quelles colonnes ont un coût réduit négatif (Bouzgarrou [13]). Si des colonnes sont trouvées, le programme linéaire est résolu à nouveau. Les branchements sont effectués lorsque plus aucune colonne n'a un coût réduit négatif et que la solution du programme linéaire n'est pas entière. Le *B&P* est aussi une généralisation du *B&B* et permet que la génération de colonnes soit appliquée partout dans l'arbre *B&B*.

La méthode *B&C&P* intègre les éléments du *B&C* et du *B&P*. Elle est utile lorsque le nombre de variables et de contraintes est élevé (Bouzgarrou [13]).

On peut également utiliser la programmation dynamique pour résoudre des *POC* en combinant des solutions de sous-problèmes de façon ascendante (*bottom-up*). Les méthodes diviser-pour-régner, elles, les résolvent de façon descendante (*top-down*). Cormen, Leiserson *et al.* [21] expliquent comment développer un algorithme de programmation dynamique pour résoudre un *POC*. Il faut savoir qu'il est possible de modéliser mathématiquement la structure d'une solution optimale d'un *POC*. Premièrement, il faut trouver la structure d'une solution optimale. Ensuite, on doit définir récursivement la valeur d'une solution optimale à partir de solutions optimales de sous-problèmes. La prochaine étape consiste à calculer la valeur d'une solution optimale de manière ascendante, en partant du plus petit sous-problème jusqu'au problème initial. Ce calcul mène à la solution optimale du problème à résoudre.

Selon Nemhauser et Wolsey [60], la relaxation lagrangienne consiste à retirer certaines contraintes d'un programme linéaire pour le rendre plus facile à résoudre. On

peut ensuite utiliser un algorithme exact comme le *B&B* ou l'algorithme du simplexe pour résoudre le problème relaxé.

Il est aussi possible de représenter certains *POC* sous la forme de problèmes de satisfaction de contraintes (*CSP*). Boivin [11] décrit le *CSP* ainsi que ses méthodes de résolutions. Un *CSP* est un triplet $P = \{X, D, C\}$ où X est un ensemble de variables $\{X_1, X_2, \ldots, X_n\}$, D est un ensemble de domaines $\{D_1, D_2, \ldots, D_n\}$ et C, un ensemble de contraintes. Chaque X_i peut prendre une valeur comprise dans le domaine D_i et les valeurs des X_i doivent respecter les contraintes de C. Plusieurs méthodes existent pour résoudre un *CSP*. Le retour arrière (*Back Tracking*) est une méthode qui consiste à donner une valeur aux variables séquentiellement. Lorsque toutes les variables liées à une contrainte ont été affectées, on vérifie si la contrainte n'a pas été violée. Si c'est le cas, on effectue un retour arrière en donnant une valeur alternative à la dernière variable affectée. Sinon, on continue à affecter les variables. Le saut arrière (*Back Jumping*) est une méthode similaire au retour arrière, sauf que lorsqu'un retour arrière est nécessaire, elle peut retourner plus loin que la dernière variable affectée. Une fonction d'analyse permet de déceler la variable qui a le plus d'incidence sur la violation de la contrainte. Une autre méthode, nommée propagation de contraintes, consiste à réduire certains domaines de D, ce qui rend la recherche de solutions plus facile et permet de détecter des situations de violation de contraintes plus rapidement. La vérification avant (*Forward Checking*) consiste à vérifier l'impact qu'aura l'instanciation d'une variable avant de prendre une décision. Si, en donnant une certaine valeur à une variable, on vide le contenu du domaine des autres variables, alors on change cette valeur. La vérification avant permet donc de diminuer les domaines des variables et

de détecter à l'avance les violations de contraintes, ce que le retour arrière ou le saut arrière ne peuvent faire.

2.2.2 Les méthodes heuristiques

On peut également utiliser des heuristiques pour solutionner un *POC*. Nicholson [61] dit qu'une méthode heuristique peut être définie comme étant une procédure exploitant au mieux la structure du problème considéré dans le but de trouver une solution de qualité raisonnable dans des délais aussi courts que possible. D'autres méthodes, nommées métaheuristiques, sont plus générales et peuvent être appliquées à plusieurs catégories de *POC*. Les métaheuristiques les plus connues sont la recherche locale (*RL*), le recuit simulé (*RS*), la recherche avec tabous (*RT*), les algorithmes génétiques (*AG*) et l'optimisation par colonie de fourmis (*OCF*). Plusieurs autres métaheuristiques existent, mais, plus souvent qu'autrement, elles sont dérivées de celles nommées précédemment. Widmer [85] donne un bon aperçu de ces méthodes et une bibliographie répertoriant une grande quantité de travaux a été réalisée par Osman et Laporte [62]. Cette section explique le fonctionnement de quelques métaheuristiques importantes.

2.2.2.1 La recherche locale

La recherche locale (*RL*) est un algorithme fort simple qui fouille l'espace de solutions d'un problème en visitant pas à pas chaque solution. Une opération élémentaire est effectuée sur la solution s pour donner une solution voisine s'. Le voisinage $N(s)$ comprend toutes les solutions pouvant être obtenues en effectuant une opération élémentaire sur s. Il existe différentes opérations élémentaires, chacune d'elles donnant un

voisinage différent. Pour résoudre un problème de minimisation, un algorithme de descente accepte une nouvelle solution (donnée par l'opération élémentaire définissant le voisinage) seulement si sa valeur de la fonction objectif est inférieure à celle de la solution précédente. Cette méthode a pour désavantage d'être souvent coincée dans un optimum local.

2.2.2.2 Le recuit simulé

Le recuit consiste à faire refroidir un métal le plus lentement possible pour qu'il obtienne une structure moléculaire optimale. L'algorithme recuit simulé (*RS*) est une méthode de recherche locale inspiré de ce phénomène thermodynamique. Cet algorithme a été utilisé pour résoudre des *POC* pour la première fois par Kirkpatrick, Jr. *et al.* [49] et Cerny [16]. Le *RS* se distingue de l'algorithme de descente en acceptant des solutions de moindre qualité dans l'espoir de sortir d'un optimum local. Cette acceptation se fait suivant la probabilité donnée par $exp(-\delta/T)$, où δ est la différence entre la valeur de l'objectif de la nouvelle solution et la valeur de l'objectif de la solution courante et T est la température. La température est un paramètre fixé au début de l'algorithme et diminue graduellement tout au long de l'exécution de l'algorithme. Plus la valeur T est élevée, plus une solution de qualité moindre a de chances d'être acceptée. Plus T s'approche de 0, moins une solution de mauvaise qualité a de chances d'être acceptée. À chaque degré de température, un nombre prédéfini de solutions voisines est visité. Une fois cette quantité atteinte, la température est diminuée. La meilleure façon de diminuer la température

dépend du problème étudié et elle peut être déterminée par essais et erreurs. Par exemple, il est possible de la diminuer de façon linéaire ou exponentielle.

2.2.2.3 La recherche avec tabous

La recherche avec tabous (*RT*), qui est aussi une méthode de recherche locale, a été introduite par Glover [40]. Cette méthode garde temporairement en mémoire les transformations effectuées sur la solution. Cette mémoire s'appelle la liste de tabous et son rôle est d'empêcher l'algorithme de revenir sur ses pas. La nouvelle solution est trouvée en fouillant l'ensemble du voisinage de la solution courante. Pour éviter que la liste de tabous ne devienne trop restrictive, le critère d'acceptation permet de révoquer le statut de tabou d'une transformation. Un exemple de critère d'acceptation serait de retirer de la liste de tabous une transformation qui permettrait d'obtenir une solution de qualité supérieure à la meilleure solution rencontrée.

2.2.2.4 L'algorithme génétique

L'algorithme génétique (*AG*) a été proposé par Holland [47]. Inspiré de la loi de la sélection naturelle, l'*AG* manipule une population d'individus, où un individu est une solution du *POC* traité. Chaque individu est encodé, représentant son matériel génétique. L'algorithme est un processus itératif exécutant séquentiellement quatre opérateurs génétiques : la sélection, le croisement, la mutation et le remplacement. La sélection consiste à choisir les individus qui pourront se reproduire et se fait généralement de façon aléatoire en priorisant les meilleurs individus. Les individus sont regroupés en paires pour l'étape de croisement et celui-ci s'effectue entre deux individus parents pour produire deux

enfants ayant les gènes de leurs parents. Il existe différentes façons de croiser des individus et certaines méthodes sont plus appropriées que d'autres pour certains *POC*. La mutation consiste à effectuer une modification aléatoire sur un enfant immédiatement après le croisement. Par exemple, si un individu est représenté sous forme d'une chaîne de bits, la mutation pourrait être d'inverser un bit choisi aléatoirement. Il est important de s'assurer que la mutation n'engendre pas une solution non réalisable. La mutation s'effectue généralement selon un très faible pourcentage. Finalement, le remplacement consiste à remplacer les individus de l'ancienne population par les individus de la nouvelle population. On peut remplacer l'ancienne population de différentes façons : la remplacer au complet par la nouvelle population, remplacer les pires individus de l'ancienne par les meilleurs de la nouvelle, etc. Éventuellement, les individus de la population finiront par converger vers une solution de bonne qualité. L'*AG* comprend plusieurs paramètres (taille de la population, taille du bassin de reproduction, taux de croisement et de mutation, différentes méthodes de sélection, croisement, mutation et remplacement) qui font converger plus ou moins rapidement la population.

2.2.2.5 L'optimisation par colonies de fourmis

Dorigo, Maniezzo *et al.* [28] proposent l'optimisation par colonies de fourmis (*OCF*). Cette méthode, comme son nom l'indique, est inspirée des colonies de fourmis et est teintée par un algorithme vorace. Dans la nature, les fourmis sont capables de trouver le plus court chemin entre une source de nourriture et leur nid, non pas en utilisant des repères visuels, mais en suivant une piste de phéromone. Une fourmi isolée se déplace

essentiellement au hasard et, lorsqu'elle rencontre une trace de phéromone, la probabilité qu'elle la suive est élevée. Ceci dit, chaque fourmi laisse sur son passage une trace de phéromone; plus il y a de fourmis suivant une trace, plus il y a de phéromone, et plus il y a de phéromone, plus grande sera la probabilité qu'une fourmi suive cette trace. Tôt ou tard, toutes les fourmis suivront cette trace.

L'*OCF* décrit dans ce paragraphe est un *Ant System* orienté pour résoudre le problème du voyageur de commerce. Il consiste à trouver le plus court cycle hamiltonien dans un graphe pondéré, où chaque nœud du graphe représente une ville. Soit d_{ij}, la distance entre les villes i et j, et le couple (i, j), l'arête entre ces deux villes. L'algorithme commence par positionner m fourmis sur n villes au temps $t = 0$. À chaque unité de temps, chaque fourmi k choisit la prochaine ville à visiter parmi l'ensemble des villes V_k à l'aide de la règle de transition définie à l'équation 2.1. L'ensemble de villes V_k contient les villes que la fourmi k n'a pas encore visitées. L'équation 2.1 représente la probabilité qu'une fourmi k se déplace de la ville i à la ville j en considérant la distance à parcourir pour atteindre cette ville à titre d'élément de visibilité ($\eta_{ij} = 1 / d_{ij}$) et la quantité de phéromone ($\tau_{ij}(t)$) accumulée sur le chemin (arête du graphe). α et β sont des paramètres contrôlant respectivement l'importance accordée à la trace de phéromone et à la visibilité.

$$p_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{u \in V_k}[\tau_{iu}(t)]^{\alpha}[\eta_{iu}]^{\beta}} \qquad (2.1)$$

Une fois la tournée construite, chaque fourmi laisse une trace de phéromone sur les arêtes empruntées en mettant à jour la matrice de phéromone selon l'équation 2.2. On a $0 < \rho < 1$ qui est un coefficient, tel que $(1 - \rho)$ représente l'évaporation des phéromones, et $\Delta\tau_{ij}$

$= \sum_{k=1}^{m} \Delta \tau_{ij}^{k}$ cumule la quantité de phéromone ($\Delta\tau_{ij}^{k}$) laissée par unité de longueur sur l'arête (i, j) par l'ensemble des m fourmis. Plus précisément, $\Delta\tau_{ij}^{k} = Q / L_k$ si la $k^{ième}$ fourmi est passée sur l'arête (i, j) dans sa tournée; $\Delta\tau_{ij}^{k} = 0$ dans le cas contraire. Q est une constante (généralement 1) et L_k est la longueur de la tournée de la $k^{ième}$ fourmi calculée à la fin de chaque cycle. Ceci complète un cycle de l'algorithme.

$$\tau_{ij}(t + 1) = \rho\,\tau_{ij}(t) + \Delta\tau_{ij} \qquad (2.2)$$

Différents paramètres doivent être fixés, tels que le nombre de fourmis, le nombre de cycles, les constantes Q, α et β. Il existe plusieurs variantes de l'*OCF*, dont la version A*nt Colony System* qui représente la version la plus évoluée de la métaheuristique. Ses principales particularités sont la mise-à-jour de la matrice de phéromone à l'aide de la meilleure fourmi de chaque cycle, l'utilisation d'une liste de candidats et une procédure de mise-à-jour locale pseudo-proportionnelle au niveau de la règle de transition.

2.2.3 Les méthodes hybrides

Les métaheuristiques hybrides sont apparues en même temps que le paradigme lui-même, mais la plupart des chercheurs n'y accordaient que peu d'intérêt (Cotta, Talbi *et al.* [22]). Elles gagnent maintenant en popularité, car les meilleurs résultats trouvés pour plusieurs *POC* ont étés obtenus avec des algorithmes hybrides. Les méthodes hybrides peuvent être divisées en deux groupes : les métaheuristiques hybrides qui impliquent une combinaison de plusieurs métaheuristiques et les méthodes hybrides impliquant une combinaison d'une méthode exacte et d'une métaheuristique. Décrivons plus en détails ces deux groupes de méthodes hybrides.

2.2.3.1 Les métaheuristiques hybrides

La recherche locale, le recuit simulé, la recherche avec tabous et les algorithmes évolutifs ont déjà été hybridés avec succès dans plusieurs applications. On peut classifier ces différentes hybridations selon la taxonomie proposée par Talbi [80]. Cette classification permet de comparer les métaheuristiques hybrides de façon qualitative. La taxonomie comporte deux aspects. Une classification hiérarchique permet d'abord d'identifier la structure de l'hybridation. Ensuite, une classification générale spécifie les détails des algorithmes impliqués dans l'hybridation.

2.2.3.1.1 Classification hiérarchique

La classification hiérarchique se subdivise en deux classes : l'hybridation de bas niveau et l'hybridation de haut niveau. On a une hybridation de bas niveau lorsqu'une fonction d'une métaheuristique est remplacée par une autre métaheuristique. On obtient une hybridation de haut niveau lorsque deux métaheuristiques sont hybridées sans que leur fonctionnement interne ne soit en relation.

Chacune des deux classes d'hybridation précédentes se subdivise en deux autres classes : à relais et co-évolutionnaire. Lorsque les métaheuristiques sont exécutées de façon séquentielle, l'une utilisant le résultat de la précédente comme entrée, on a une hybridation à relais. L'hybridation co-évolutionnaire se fait lorsque des agents coopèrent en parallèle pour explorer l'espace de solutions.

La combinaison de toutes les classes nommées précédemment donne quatre classes différentes : les hybridations bas niveau à relais, bas niveau co-évolutionnaire, haut niveau

à relais et haut niveau co-évolutionnaire. Voici une description plus détaillée de chacune des classes d'hybridation avec quelques exemples de travaux réalisés.

L'hybridation de bas niveau à relais représente les algorithmes dans lesquels une métaheuristique est incorporée dans une autre métaheuristique à solution unique. Par exemple, Martin et Otto [55] ont incorporé une *RL* dans un algorithme de *RS* pour résoudre le problème du voyageur de commerce et celui de la partition de graphes. Leur méthode a surpassé les algorithmes de recherche locale traditionnels.

L'hybridation de bas niveau co-évolutionnaire consiste à incorporer un algorithme de recherche locale axé sur l'exploitation à une métaheuristique à population axée sur l'exploration. Ces deux types de métaheuristiques ont des forces et faiblesses complémentaires : l'exploitation raffine une solution tandis que l'exploration fouille plus largement l'ensemble des solutions. Pour le problème d'assignation de tâches sur des ordinateurs hétérogènes, Salcedo-Sanz, Xu *et al.* [70] proposent une méthode qui améliore les individus d'un *AG* grâce à un algorithme de réseaux de neurones avant d'utiliser les opérateurs génétiques. Fleurent et Ferland [35] remplacent l'opérateur de mutation d'un *AG* par une *RL* ou une *RT* pour résoudre le problème de partition de graphes. Les *OCF* sont aussi hybridés avec des algorithmes de recherche locale. Règle générale, la phase de recherche locale s'effectue à la fin de chaque cycle de l'*OCF*, c'est-à-dire lorsque les fourmis ont terminé de construire leur solution. Stützle et Hoos [76] incorporent une fonction de recherche locale 2-Opt (Bentley [10]) dans un *OCF* pour le problème du voyageur de commerce et celui de partition de graphes. Gambardella, Taillard *et al.* [38] utilisent un algorithme de descente dans un *OCF* pour résoudre le problème de partition de

graphes. De plus, pour initialiser la matrice de phéromone, la méthode crée aléatoirement un premier groupe de fourmis, qu'elle optimise avec l'algorithme de descente. McKendall et Shang [57] bonifient la méthode de Gambardella, Taillard *et al.* [38] : dans un premier cas, en ajoutant à l'algorithme de descente une stratégie de *look ahead / look back*, dans un deuxième cas, en remplaçant l'algorithme de descente par un *RS*. Pour résoudre le problème d'agencement d'horaires d'examens, Azimi [6] propose une méthode où la solution de la meilleur fourmi de chaque cycle d'un *OCF* est améliorée par une *RT*.

L'hybridation de haut niveau à relais se fait lorsque des métaheuristiques complètes sont exécutées séquentiellement. Par exemple, on peut améliorer la solution d'un algorithme à population en l'utilisant comme solution de départ d'un algorithme de recherche locale. Azimi [6] propose deux hybridations haut niveau à relais pour l'agencement d'horaires d'examens. La première méthode utilise une *RT* dont la solution trouvée servira à initialiser la matrice de phéromone d'un *OCF*. La deuxième méthode prend la meilleure solution trouvée par un *OCF* comme solution de départ d'une *RT*. C'est cette dernière méthode qui s'est le plus démarquée parmi les trois méthodes proposées par l'auteur dans ce même article. Pour résoudre le problème d'emballage d'ensembles (*set packing problem*), Ribeiro, Trindade *et al.* [69] utilisent un algorithme *GRASP* (*Greedy Randomized Adaptive Search Procedures*) pendant un temps *t* pour construire un ensemble de solutions. Ensuite, ils utilisent un algorithme d'extraction de données pour identifier les *patterns* présents dans les solutions de bonne qualité. Finalement, ils utilisent une modification de l'algorithme *GRASP*, qui tient compte des informations trouvées pendant la phase d'extraction de données. Angel, Bampis *et al.* [3] ont combiné un *OCF* et la

programmation par contraintes pour résoudre le problème de pliage de protéines. Les deux méthodes sont exécutées l'une après l'autre et ce, dans un processus itératif. La programmation par contraintes se sert de la trace de phéromones de l'*OCF* pour orienter sa recherche, et l'*OCF* utilise la solution trouvée par la programmation par contraintes pour mettre à jour la trace de phéromone. Pour résoudre le problème d'assignation quadratique, Lin, Kao *et al.* [54] proposent une méthode cyclique à deux étapes, où un *RS* crée une population pour l'*AG* de la deuxième étape.

Finalement, l'hybridation de haut niveau co-évolutionnaire implique un ensemble de métaheuristiques complètes qui travaillent en parallèle et coopèrent pour trouver la solution optimale d'un problème. L'algorithme génétique basé sur le modèle insulaire proposé par Tanese [82] en est un exemple. Dans ce modèle, la population est divisée en petites sous-populations réparties sur les sommets d'un hypercube. Chacune de ces sous-populations évolue selon les règles de l'algorithme génétique. À intervalles réguliers, des migrations ont lieu entre les sous-populations situées sur la même dimension de l'hypercube. Belding [9] a modifié cette méthode en permettant aux immigrants d'aller vers des sous-populations choisies aléatoirement, plutôt que de se limiter à la topologie de l'hypercube. Pour résoudre des problèmes de partitionnement de graphe, Hammami et Ghédira [44] font coopérer une *RT* et un *RS*, de sorte que, à intervalles réguliers, les deux méthodes échangent des informations, à savoir la meilleure solution trouvée ainsi que des pénalités infligées aux éléments de solution inintéressants. Cette méthode performe mieux que si les deux métaheuristiques n'échangent pas d'informations. Dans un autre article, Hammami et Ghédira [43] améliorent leur algorithme en y ajoutant, dans un premier cas,

une fonction de croisement qui combine la meilleure solution trouvée par les deux algorithmes et dans un deuxième cas, la meilleure solution trouvée par chaque agent est améliorée par une fonction d'intensification proposée par Kernighan et Lin [48]. Cette dernière méthode surpasse les deux autres déjà proposées par Hammami et Ghédira.

2.2.3.1.2 Classification générale

La classification générale de la taxonomie de Talbi [80] comporte trois dichotomies : les approches hybrides homogènes versus hétérogènes, globales versus partielles et spécialistes versus générales.

Une hybridation est dite homogène lorsque les métaheuristiques combinées sont identiques. Par exemple, le modèle proposé par Tanese [82], et cité précédemment, est une hybridation de haut niveau co-évolutionnaire homogène. À l'inverse, une hybridation hétérogène combine des métaheuristiques différentes. La méthode proposée par Hammami et Ghédira [44] est une hybridation de haut niveau co-évolutionnaire hétérogène.

Une hybridation globale fait en sorte que toutes les métaheuristiques explorent l'ensemble de l'espace de solutions. Toutes les méthodes présentées dans la section précédente sont des hybridations globales. D'un autre côté, l'hybridation partielle décompose un problème en sous-problèmes ayant leur propre espace de solutions, et chaque sous-problème est donné à un algorithme. Taillard [77] propose une méthode qui décompose un problème de routage de véhicules en sous-problèmes résolus avec une *RT*. Les sous-problèmes sont créés en divisant l'ensemble des villes à visiter en secteurs. Taillard et Voss [79] ont développé *POPMUSIC*, une méthode qui consiste à créer des

sous-problèmes à partir du problème initial et à résoudre ce sous-problème à l'aide d'une métaheuristique ou encore d'un algorithme exact. Cet algorithme a été utilisé pour résoudre des *centroid clustering problems* et des problèmes de balancement de pièces mécaniques, en utilisant la *RT* pour résoudre les sous-problèmes. De la même façon, Taillard et Burri [78] ont utilisé *POPMUSIC* pour résoudre des problèmes de placement de légende sur des plans.

Les hybridations générales sont celles où tous les algorithmes résolvent le même problème d'optimisation. Les méthodes citées dans la section précédente sont toutes des hybridations générales. Les hybridations spécialistes sont celles où chaque algorithme résout un problème d'optimisation différent. Bachelet, Hafidi *et al.* [7] proposent une méthode hybride de haut niveau co-évolutionnaire où une *RT* est utilisée en parallèle pour résoudre un problème d'assignation quadratique, tandis qu'un *AG* effectue une tâche de diversification formulée comme un autre problème d'optimisation. Une mémoire de fréquence sert à emmagasiner des informations relatives aux solutions visitées par la *RT*. L'*AG* utilise la mémoire de fréquence pour générer des solutions appartenant à des régions non explorées. Une autre approche d'hybridation spécialiste à haut niveau co-évolutionnaire consiste à utiliser une heuristique pour optimiser la valeur des paramètres d'une autre heuristique. Par exemple, Krueger [52] optimise les paramètres d'un *RS* à l'aide d'un *AG*, Abbattista, Abbattista *et al.* [1] optimisent ceux d'un *OCF* à l'aide d'un *AG* et Shahookar et Mazumder [71] optimisent les paramètres d'un *AG* à l'aide d'un autre *AG*.

2.2.3.2 L'hybridation de méthodes exactes et de métaheuristiques

Il est possible d'hybrider de plusieurs façons une méthode exacte avec une métaheuristique. Une classification proposée par Puchinger et Raidl [66] permet de différencier la plupart des formes d'hybridations. Cette section présente les détails de cette classification ainsi que quelques travaux réalisés par différents auteurs.

Puchinger et Raidl [66] proposent une classification axée sur l'hybridation de méthodes exactes et approximatives. On divise les méthodes hybrides en deux catégories : les hybridations collaboratives et celles intégratives. Les algorithmes qui échangent des informations de façon séquentielle, parallèle ou entrelacée entrent dans la catégorie des hybridations collaboratives. Les algorithmes à hybridation intégrative font en sorte qu'une technique est une composante incorporée à une autre technique. Autrement dit, il y a un algorithme maître et un algorithme esclave.

2.2.3.2.1 Hybridations collaboratives

L'hybridation collaborative séquentielle est exécutée de façon à ce que la méthode exacte soit un prétraitement de la métaheuristique, ou vice-versa. Applegate, Bixby *et al.* [4] proposent une méthode séquentielle pour résoudre le problème du voyageur de commerce. Dans un premier temps, une *RL* est exécutée pour créer un ensemble de solutions. Un sous-problème est créé à partir des arêtes qui se retrouvent dans les solutions de l'ensemble. Finalement, on trouve la solution optimale du graphe restreint à l'aide d'un algorithme exact. Klau, Ljubic *et al.* [51] proposent une méthode similaire pour résoudre le

prize-collecting Steiner tree problem. Un prétraitement est tout d'abord réalisé pour réduire le graphe, suivi d'un algorithme mémétique. Finalement, un algorithme *B&C* est appliqué au sous-problème donné par la fusion des solutions trouvées par l'algorithme mémétique. Feltl et Raidl [32] appliquent la solution de la relaxation d'un programme linéaire à une fonction d'arrondissement aléatoire pour créer les solutions de départ d'un *AG*, afin de résoudre le problème d'assignation généralisé. Si une solution est infaisable, une fonction de réparation aléatoire est utilisée pour la corriger.

Les prochaines méthodes présentées fixent un ensemble de variables et résolvent le sous-problème donné par les variables libres. Pour résoudre le problème d'ordonnancement de projet à moyens limités, Palpant, Artigues *et al.* [63] construisent, dans un premier temps, une solution de départ à l'aide d'une heuristique. Dans un deuxième temps, grâce à un processus itératif, les sous-problèmes sont créés et résolus de façon optimale. Applegate et Cook [5] ont créé une méthode nommée *Shuffle* pour résoudre les problèmes de *job-shop*. Büdenbender, Grünert *et al.* [14] résolvent des problèmes de *direct flight network design* grâce à un algorithme hybridant une *RT* et le *B&B*. Hansen et Mladenovic [45] créent le *Variable Neighborhood Decomposition Search* pour résoudre des problèmes de *TSP*. Mautor et Michelon [56] ont résolu des problèmes d'affectation quadratique à l'aide de l'algorithme *Mimausa*. La méthode *POPMUSIC* de Taillard et Voss [79] entre aussi dans cette catégorie d'algorithmes, puisque le choix de la fonction d'optimisation de sous-problèmes est laissé à la discrétion de l'utilisateur. La fonction d'optimisation peut donc être un algorithme exact.

Les algorithmes exacts et heuristiques peuvent aussi être exécutés de façon parallèle ou entrelacée. En ce sens, Talukdar, Baerentzen *et al.* [81] ont créé une architecture de résolution de problèmes nommée *asynchronous teams* (*A-Teams*), consistant en un ensemble d'agents et de mémoires connectées à un réseau dirigé fortement cyclique. Ceci signifie que chaque agent fait parti d'une boucle fermée. Chacun des agents est un algorithme d'optimisation approximatif ou exact qui peut travailler sur le problème à résoudre, sur une relaxation du problème ou sur une sous-classe du problème. Une population d'individus est contenue dans la mémoire et les différents agents peuvent ajouter, détruire ou modifier des individus. Différents travaux utilisent cette architecture pour résoudre des problèmes de voyageur de commerce (Souza [74]), de planification d'atelier (Chen, Talukdar *et al.* [17]) ou de planification d'horaires de trains (Tsen [83]). Denzinger et Offermann [26] présentent une architecture similaire. Leur approche, nommée *TECHS* (*Teams for Cooperative Heterogenous Search*), consiste en des équipes d'un ou plusieurs agents utilisant le même paradigme de recherche. La communication entre les agents est contrôlée par des arbitres d'envoi et de réception qui filtrent les données échangées. On y présente une méthode utilisant un système basé sur un *AG* et un *B&B* pour résoudre un problème de planification d'atelier.

2.2.3.2.2 Hybridations intégratives

Les combinaisons d'une méthode exacte et d'une métaheuristique de façon intégrative se font de manière à ce qu'un des deux algorithmes soit une composante

intégrée à l'autre algorithme. On peut donc intégrer une méthode exacte à une métaheuristique, et inversement.

Premièrement, voici comment incorporer un algorithme exact à une métaheuristique. On peut résoudre de façon exacte une relaxation du problème. Ceci peut être pratique pour guider heuristiquement la recherche dans un voisinage, la recombinaison, la mutation, la réparation et/ou l'amélioration locale. Raidl [68] et Chu et Beasley [19] proposent un *AG*, utilisant une relaxation du problème du sac alpin 0-1 multidimensionnel, pour créer la population initiale ainsi que pour réparer les solutions non-réalisables. Une autre méthode consiste à fouiller un voisinage en utilisant une méthode exacte. La recherche *Very Large-Scale Neighborhood* (*VLSN*) présenté par Ahuja, Ergun *et al.* [2] en est un exemple. Burke, Cowling *et al.* [15] proposent une méthode de *RL* à voisinage variable incorporant un algorithme exact pour résoudre le problème du voyageur de commerce asymétrique. Congram, Potts *et al.* [20] proposent une méthode nommée *Dynasearch*, qui utilise la programmation dynamique dans un algorithme de *RL*. Le *Dynasearch* est une structure de voisinage qui permet d'effectuer plusieurs mouvements par itération, contrairement aux structures de voisinages conventionnelles où un seul mouvement par itération est permis. Cette méthode ne s'applique qu'aux problèmes où la solution se représente sous la forme de permutations. Le fusionnement de solutions est une autre méthode qui consiste à fusionner les attributs d'un ensemble de solutions. Par exemple, Cotta et Troya [23] utilisent le *B&B* pour trouver le meilleur croisement possible entre deux individus dans un algorithme évolutif, sans avoir à y appliquer une mutation implicite. Finalement, les algorithmes exacts peuvent être utilisés en tant que décodeur

pour compléter de façon optimale les chromosomes incomplets de certains algorithmes évolutifs. Staggemeier, Clark *et al.* [75] proposent un *AG* comprenant une fonction de décodage pour résoudre le *lot-sizing and scheduling problem*.

Voici comment on peut incorporer une métaheuristique dans un algorithme exact. Premièrement, on peut utiliser une métaheuristique pour calculer les bornes et les solutions sortantes pour les approches *B&B*. Woodruff [86] propose une stratégie qui, à chaque nœud de l'arbre du *B&B*, détermine si une *RT* sera appelée pour trouver de meilleures solutions candidates sortantes (*incumbent solutions*). Dans le même ordre d'idée, Monfroglio [58] propose un algorithme du simplexe hybride pour résoudre des problèmes de couverture d'ensembles (*set covering*), où la sélection du pivot se fait par un algorithme de réseau de neurones. Dans les algorithmes de *B&C* et *B&P*, la séparation dynamique des plans sécants (*cutting-planes*) et l'évaluation des colonnes (*pricing of columns*), peuvent être faites grâce à des métaheuristiques qui accélèrent le processus. Filho et Lorena [33] utilisent une heuristique de génération de colonnes ainsi qu'un *AG* pour résoudre le problème de coloration de graphe. Il est aussi possible de guider le parcours de l'arbre d'un algorithme *B&B* avec une métaheuristique. French, Robinson *et al.* [36] proposent une méthode hybridant un *AG* et un *B&B* pour résoudre le problème de satisfaisabilité maximale (*maximum satisfiability*). Leur méthode commence par le *B&B* en amassant des informations pour créer des chromosomes afin de construire la population initiale de l'*AG*. Lorsqu'un certain critère est atteint, l'*AG* est démarré. Une fois l'*AG* terminé, la meilleure solution trouvée par l'*AG* est greffée à l'arbre du *B&B*, qui reprend son exécution. Finalement, on peut faire fonctionner un algorithme de *B&B* de façon à imiter la recherche

locale. En ce sens, Fischetti et Lodi [34] proposent une méthode qui résout itérativement un sous-problème correspondant au voisinage *k*-OPT en utilisant le logiciel *CPLEX* de *ILOG*. Cette résolution s'effectue en introduisant une contrainte de branchement local sur une solution candidate sortante. Le premier sous-problème est résolu, et tant qu'il y a une amélioration, un nouveau sous-problème est conçu et résolu. Lorsque le processus s'arrête, le problème est résolu en utilisant le branchement standard du *B&B*. Les problèmes résolus sont des programmes linéaires 0-1 généraux.

2.2.3.3 Discussion sur les classifications de méthodes hybrides

Hormis le fait que la classification de Puchinger soit orientée vers les hybridations de méthodes exactes et de métaheuristiques et que celle de Talbi vise les hybridations de façon plus générale, les deux taxonomies se ressemblent beaucoup. On retrouve dans les deux classifications les mêmes classes d'hybridations, telles que l'intégration d'un algorithme dans un autre algorithme (l'hybridation de bas niveau chez Talbi et les combinaisons intégratives chez Puchinger), l'exécution en série (l'hybridation de haut niveau à relais chez Talbi et la combinaison collaborative séquentielle chez Puchinger) et en parallèle des algorithmes (l'hybridation de haut niveau coopérative chez Talbi et la combinaison collaborative parallèle ou entrelacée chez Puchinger). Talbi va un peu plus loin en proposant trois dichotomies pour mieux décrire la nature des algorithmes hybrides (homogène/hétérogène, globale/partielle et générale/spécialiste) ainsi qu'une classification concernant leur implémentation.

Les travaux cités dans la Section 2.2.3 montrent que les méthodes hybrides offrent de meilleurs résultats que les algorithmes conventionnels, autant pour les problèmes pratiques que théoriques (Cotta, Talbi *et al.* [22]).

2.3 Le problème d'ordonnancement de voitures (POV)

Il a été souligné dans l'introduction que les problèmes d'optimisation combinatoire sont présents dans plusieurs domaines d'application industrielle, économique et scientifique. Dans ce mémoire, nous nous intéressons plus particulièrement au problème d'ordonnancement de voitures (*POV*) dans une chaîne d'assemblage. Dans les sections qui suivent, on présente, dans un premier temps, une description générale du *POV* et, dans un deuxième temps les différents travaux réalisés pour résoudre ce problème à connotation industrielle.

2.3.1 Description générale du POV

Dans une usine de fabrication d'automobiles, les voitures passent par trois ateliers principaux : l'atelier de tôlerie, l'atelier de peinture et l'atelier de montage. Le *POV* est un *POC* qui consiste à déterminer dans quel ordre placer les différentes voitures sur la chaîne d'assemblage en se soumettant à un ensemble fini de contraintes. On cherche parfois à minimiser le nombre de conflits, où un conflit représente une surcharge de travail occasionnée par l'arrivée successive de plusieurs voitures similaires à un poste particulier de l'atelier de montage, ou encore à minimiser le nombre de changements de couleurs à l'atelier de peinture. Pour simplifier le problème, on ne s'attardera qu'aux contraintes liées

à l'atelier de montage. C'est dans l'atelier de montage que les différentes options d'une voiture sont installées, telles que le toit ouvrant, le climatiseur, etc.

Le problème théorique du *POV* retrouvé dans la littérature (Dincbas, Simonis *et al.* [27], Parello, Kabat *et al.* [65] et Parello [64]) ne considère que les contraintes et conflits liés à l'atelier de montage et c'est ce problème, défini *NP*-difficile au sens fort par Kis [50], qui sera abordé dans le cadre de ce mémoire. Le problème industriel s'avère beaucoup plus complexe à traiter que le problème théorique, car une séquence optimale de voitures pour l'atelier de peinture peut se révéler catastrophique pour l'atelier de montage, et vice-versa (Gravel, Gagné *et al.* [42]). Ceci augmente grandement le nombre de contraintes à respecter et la nature multi-objectifs du problème augmente considérablement sa difficulté de résolution.

En ce qui concerne le *POV* théorique, trois ensembles de problèmes ainsi que les solutions trouvées par différents chercheurs sont disponibles dans la librairie *CSPlib* (http://4c.ucc.ie/~tw/csplib/prob/prob001/index.html). Le premier ensemble (*ET1*) (Lee, Leung *et al.* [53]) est composé de 70 instances de 200 voitures à ordonnancer pour lesquelles il existe une solution sans conflit pour chacune d'elles. Le deuxième ensemble (*ET2*) proposé par Gent et Walsh [39] est composé de 9 instances de 100 voitures. Il est à noter que pour cinq de ces neuf instances, aucune solution sans conflit n'est connue. Finalement, le troisième ensemble (*ET3*) proposé par Gravel, Gagné *et al.* [42] se compose de 30 instances difficiles de plus grandes tailles (200, 300 et 400 voitures). Ces trois ensembles de problèmes seront utilisés dans le présent travail pour établir la performance des algorithmes proposés.

2.3.2 Les méthodes exactes utilisées pour la résolution du POV

Gravel, Gagné *et al.* [42] proposent une formulation *PLNE* et une résolution à l'aide du logiciel commercial *XPRESS®* de *Dash Optimization*. Pour toutes les instances de problèmes de l'*ET1* et pour les instances de l'*ET2* ayant une solution optimale sans conflits, cette méthode est parvenue à les résoudre relativement facilement. En revanche, cette même méthode n'a pu trouver la solution optimale des instances de problèmes de l'*ET2* dont la solution comporte des conflits. Pour celles-ci, il a été possible de trouver relativement rapidement les meilleurs résultats connus en utilisant la valeur de l'objectif de ces solutions comme borne pour le *PLNE*.

Boivin [11] propose une méthode de vérification vers l'avant (*Forward Checking*) utilisant la programmation par contraintes pour résoudre les *POV*. Avec cette méthode, il a réussi à solutionner 64 des 70 problèmes de l'*ET1* et 1 des 9 problèmes de l'*ET2* (Boivin, Gravel *et al.* [12]). Boivin [11] propose également une approche de résolution concurrente qui consiste à diviser le problème en sous-problèmes. Celle-ci a permis d'augmenter l'efficacité de la vérification avant en diversifiant la recherche de solutions et a mené à la résolution de 67 des 70 problèmes de l'*ET1* et de 4 des 9 problèmes de l'*ET2*. Avec l'utilisation de *ILOG Solver 6.0* et de l'heuristique *Slack*, Boivin [11] a été en mesure de prouver la non-satisfiabilité d'une instance de l'*ET2* pour laquelle l'état était inconnu auparavant.

2.3.3 Les métaheuristiques utilisées pour la résolution du POV

Solnon [73] propose un *OCF*, pour la résolution du *POV*, qui a réussi à trouver la solution optimale de tous les problèmes de l'*ET1*. Elle mentionne aussi que la métaheuristique proposée a résolu de façon optimale, pour la première fois, le problème 19-71 de l'*ET2*.

Gottlieb, Puchta *et al.* [41] proposent des heuristiques voraces statiques et dynamiques, une *RL* à voisinage variable et un *OCF*. Leur étude montre que les heuristiques dynamiques surclassent les heuristiques statiques. Les meilleurs résultats obtenus ont été donnés par l'*OCF* utilisant une heuristique dynamique. La *RL* est équivalente à l'*OCF* lorsque le temps de traitement alloué est grand. Pour l'*ET1*, cette méthode est parvenue à résoudre toutes les instances de problème. Elle a aussi permis de démontrer que le problème 26-82 de l'*ET2* possède une solution sans conflits, fait qui n'avait toujours pas été prouvé. La meilleure solution connue de toutes les autres instances de problème de l'*ET2* a été atteinte.

Warwick et Tsang [84] proposent un *AG* pour résoudre le *POV*. Pour commencer, la population initiale est créée aléatoirement. Le bassin de reproduction comprend uniquement les meilleurs individus de la population. L'enfant issu du croisement de deux individus est ensuite réparé par un algorithme vorace, s'il ne satisfait pas les contraintes du problème, et est amélioré par une *RL* pendant un temps prédéterminé. De plus, l'enfant remplace immédiatement le pire individu du bassin de reproduction; il peut donc se reproduire dans la même génération à l'intérieur de laquelle il a été conçu. Finalement, la population est remplacée par les individus du bassin de reproduction. L'algorithme s'arrête

lorsqu'un critère d'arrêt est atteint. Les instances de problèmes résolues par l'*AG* ont été générées à partir d'un logiciel fourni par Zhu [87]. Les résultats ont montré que l'*AG* proposé performe très bien, autant pour résoudre les problèmes faiblement contraints que pour résoudre ceux qui sont fortement contraints. La métaheuristique peut facilement résoudre des problèmes comptant de 100 à 200 véhicules.

Un *OCF*, dont la trace de phéromone représente l'intérêt de placer deux classes de voitures une à la suite de l'autre dans la séquence, a été proposé par Gravel, Gagné *et al.* [42]. Celui-ci permet de solutionner, en moins d'une seconde, tous les problèmes de l'*ET1* et peut trouver, en moins de 10 secondes, des solutions de très bonne qualité pour tous les problèmes de l'*ET2*. Par la suite, des procédures de *RL* ont été ajoutées à l'*OCF*. La première procédure est appliquée à la meilleure solution de chaque cycle et consiste à inverser l'ordre des voitures dans une sous-séquence. La première voiture de la sous-séquence est choisie aléatoirement parmi celles présentant un conflit. La dernière voiture est choisie aléatoirement parmi toutes les voitures. Une deuxième procédure de *RL*, utilisant le voisinage *Lin2Opt* (Puchta et Gottlieb [67]), est appliquée à la toute fin de l'algorithme sur la meilleure solution globale trouvée par l'*OCF*. Ces procédures de recherche locale ont permis d'améliorer significativement les résultats pour les problèmes de l'*ET2*.

Morin [59] bonifie l'*OCF* proposé par Gravel, Gagné *et al.* [42] en trois points. Premièrement, la matrice de phéromone a été modifiée en matrice tridimensionnelle, où chaque dimension représente la distance entre deux voitures. Cette nouvelle matrice de phéromone représente donc la désirabilité d'espacer deux classes de voitures d'une certaine

distance dans la séquence. La deuxième bonification consiste à faire varier de façon régulière la valeur des exposants donnant l'importance accordée à la désirabilité et à la visibilité dans la règle de transition. Finalement, la phase de construction de solution a été dynamisée en permettant aux fourmis d'insérer une classe de voiture à l'endroit semblant être le plus approprié dans la séquence, au lieu de l'ajouter bêtement à la fin. Cette méthode de construction demande cependant un temps de traitement important. Morin [59] propose plutôt d'utiliser un mode de construction mixte, c'est-à-dire une construction en mode séquentiel qui passe en mode dynamique sous certaines conditions et revient éventuellement en mode séquentiel. Cet *OCF* bonifié permet des performances équivalentes ou supérieures à la version originale, quoique cette dernière reste supérieure pour des instances de plus grandes tailles (300 et 400 voitures). La meilleure solution connue a été atteinte pour toutes les instances de l'*ET1* et l'*ET2*, ainsi que pour 16 des 30 problèmes de l'*ET3*.

Gagné, Gravel *et al.* [37] proposent un *OCF* à plusieurs objectifs pour résoudre des problèmes réels d'ordonnancement de voitures à partir de données fournies par le *Groupe Renault*. Le problème réel prend en considération les ateliers de peinture et d'assemblage. Les options à installer sur les voitures sont divisées en deux : les options de haute priorité et de celles basse priorité. Les conflits liés aux options de haute priorité ont un impact considérable sur la chaîne de montage et sont généralement gérés en assignant temporairement un travailleur additionnel. À l'opposé, les conflits liés aux options de basse priorité ont moins d'impact sur la chaîne de montage et sont gérés plus facilement. Les trois objectifs de l'*OCF* proposé par Gagné, Gravel *et al.* [37] sont la minimisation du

nombre de conflits liés aux options de haute priorité dans l'atelier d'assemblage, la minimisation du nombre de changements de couleurs dans l'atelier de peinture et la minimisation du nombre de conflits liés aux options de basse priorité dans l'atelier d'assemblage. Les résultats obtenus par l'*OCF* multi-objectifs proposé sont meilleurs que ceux obtenus par le *RS* proposé par Chew, David *et al.* [18] et utilisé depuis dix ans dans l'industrie.

Puchta et Gottlieb [67] proposent trois méthodes pour résoudre le problème réel d'ordonnancement de voitures. Plus précisément, ils utilisent deux versions différentes de la métaheuristique nommée acceptation de seuils (*Threshold Accepting*) et une *RL*. L'acceptation de seuils, proposée par Dueck et Scheuer [30], est une méthode dérivée du *RS* où la température est remplacée par un seuil. Une solution voisine de moins bonne qualité que la solution courante est acceptée lorsque l'écart qui les sépare est inférieur au seuil. La valeur du seuil est diminuée tout au long de l'algorithme. L'autre version de l'acceptation de seuils utilisée consiste à faire bondir la valeur du seuil, vers la fin de l'exécution, pour aider l'algorithme à sortir d'un optimum local. Il s'est avéré que les deux versions d'acceptation de seuils ont surpassé la *RL*, et que la version « bondissante » est plus efficace que l'acceptation de seuils standard, quoiqu'elle requière plus de paramétrage. Six instances de problème fournies par un fabriquant allemand d'automobiles ont été utilisées pour les bancs d'essais.

Davenport, Tsang *et al.* [25] proposent une méthode nommée *GENET* pour résoudre le *POV* sous forme d'un problème de satisfaction de contraintes. Un coût est donné à chaque contrainte lorsqu'elle est violée. L'algorithme tente de minimiser le coût total lié à

la violation de contraintes en utilisant une *RL*. Davenport et Tsang [24] bonifient *GENET* pour obtenir une méthode nommée *SWAPGENET*. Le voisinage de cette méthode consiste à échanger la valeur de deux variables. *SWAPGENET* s'est avéré plus performant que son prédécesseur. Des instances de problème générées aléatoirement ont été utilisées pour les bancs d'essais.

2.4 Objectifs de la recherche

L'objectif principal cette recherche vise à proposer des approches efficaces de résolution au problème d'ordonnancement de voitures. Dans cette optique, un premier objectif est de concevoir des méthodes de résolution hybrides adaptées au *POV* en combinant une métaheuristique et une méthode exacte. Plus précisément, on veut combiner la rapidité de l'*OCF* proposé par Gravel, Gagné *et al.* [42] et la force brute du *B&B*. En fait, l'*OCF* fera appel à l'algorithme *B&B* du logiciel *CPLEX* de *ILOG* en envoyant à ce dernier un sous-problème (du problème général) qui se veut plus facile à résoudre par une méthode exacte.

On veut également explorer les deux formes d'hybridation de la classification de Puchinger et Raidl [66] en créant une méthode intégrative et une méthode collaborative. Dans le premier cas, le *B&B* sera exécuté à chaque cycle de l'*OCF*, faisant de cette méthode un algorithme hybride intégratif. Quant à la seconde méthode, l'*OCF* et le *B&B* seront exécutés à tour de rôle, donnant un algorithme collaboratif.

Le deuxième objectif de cette recherche est d'établir la performance de ces méthodes de résolution hybrides. Gravel, Gagné *et al.* [42] ont proposé une version de

l'*OCF* incorporant une procédure de recherche locale. Une comparaison de performances pourra alors être établie à partir des ensembles de problèmes *ET2* et *ET3*. Les instances de problèmes pour lesquelles il existe une solution sans conflits ne seront pas considérées puisqu'elles sont faciles à résoudre, autant par une métaheuristique que par une méthode exacte.

CHAPITRE 3 :

MÉTHODES HYBRIDES POUR LA RÉSOLUTION DU PROBLÈME D'ORDONNANCEMENT DE VOITURES

3.1 Introduction

Comme nous l'avons vu précédemment, il existe plusieurs façons de résoudre les problèmes d'optimisation combinatoire. Parmi celles-ci, il y a les méthodes hybrides, qui consistent à combiner au moins deux méthodes de résolution. Toutefois, il existe une multitude de manières de les combiner. Dans ce chapitre, nous présentons tout d'abord une description détaillée du problème d'ordonnancement de voitures. Ensuite, les deux approches de solution proposées par Gravel, Gagné *et al.* [42] pour la résolution du *POV* sont décrites, en commençant par l'optimisation par colonies de fourmis, suivie de l'utilisation d'un programme linéaire en nombres entiers. Trois hybridations sont ensuite proposées : une hybridation intégrative à chaque cycle de l'*OCF*, une hybridation intégrative à fréquence espacée et une hybridation à relais. Puis, nous discutons de l'influence du *PLNE* sur l'*OCF*. Pour terminer, nous étudions les conséquences du temps accordé au *PLNE* pour résoudre un problème.

3.2 Description du POV

Le *POV* consiste à déterminer l'ordre dans lequel un ensemble de voitures doit être produit, en considérant les options de chacune et les contraintes de capacité de la chaîne de montage. Cette section explique plus en détails en quoi consiste le *POV*. Premièrement, on y décrit une instance d'un *POV* ainsi qu'un algorithme pour évaluer une solution. On explique ensuite comment évaluer la difficulté d'une instance d'un *POV*.

Une instance d'un *POV* comprend un ensemble de variables de décision. La variable *nc* représente le nombre total de voitures à produire, *opt* est le nombre total

d'options disponibles et *cl*, le nombre total de classes de voitures. Deux voitures font partie de la même classe lorsqu'elles requièrent exactement les mêmes options. On représente les classes par une matrice de valeurs booléennes *o* de taille *cl* par *opt*, où o_{ik} est vraie si la classe de voitures *i* nécessite l'option *k*. On a aussi n_i, le nombre de voitures de classes *i* à produire tel que $\sum_{i=1}^{cl} n_i = nc$. Finalement, on exprime la contrainte de capacité de chaque option par le ratio r_k / s_k, où r_k est la quantité maximale de voitures pouvant posséder l'option *k* dans une sous-séquence de longueur s_k.

Le Tableau 3.1, tiré de Smith [72], présente un exemple d'une instance d'un *POV*. Pour cette instance, on a *opt* = 5, *cl* = 12 et $\sum_{i=1}^{cl} n_i = nc = 25$. Les cinq contraintes de capacité sur les options sont exprimées par les valeurs de *r* et *s*. On remarque que l'option 1 a une contrainte de capacité de 1 / 2. Donc, si les classes de voitures assignées aux positions *j* et *j*+1 de la séquence demandent l'option 1, il y a conflit à la position *j*. On peut aussi observer que la classe 1 regroupe les voitures possédant uniquement l'option 2 et que la classe 2 regroupe les voitures possédant les options 1, 3 et 5. On doit produire trois voitures de classe 1, une voiture de classe 2, deux voitures de classe 3, etc.

	Contraintes de capacité		**Classes des voitures**											
Option	***r***	***s***	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	1	2	0	1	1	0	0	0	1	1	1	0	0	1
2	2	3	1	0	1	1	1	0	1	0	0	0	1	1
3	1	3	0	1	0	0	0	0	1	0	1	1	1	0
4	2	5	0	0	0	1	0	1	0	1	0	0	0	1
5	1	5	0	1	0	0	1	0	0	0	0	0	0	0
		n_i	3	1	2	4	3	3	2	1	1	2	2	1

Tableau 3.1 : Instance d'un *POV*

Une solution d'une instance d'un *POV* est représentée par un vecteur de longueur *nc* contenant une et une seule classe de voiture dans chacune de ses cases. On peut construire une matrice *S*, de valeurs binaires de taille *opt* par *nc*, à partir des informations contenues dans le vecteur de la solution dans le but d'évaluer cette dernière. On a $S_{kj} = 1$ si la classe de voiture assignée à la position *j* du vecteur de la solution demande l'option *k*, 0 sinon. L'algorithme de la Figure 3.1 montre comment évaluer une solution en calculant le nombre de conflits à partir de la matrice *S*. La variable *nombre_conflits* contient, comme son nom l'indique, le nombre de conflits dans la solution. Quant à la variable *somme_bloc*, elle contient le nombre total de voitures demandant l'option *k* pour un bloc de taille s_k commençant à la position *j*. Autrement dit, la technique consiste à faire glisser le bloc de taille s_k sur chacune des positions de la séquence de voitures et à vérifier si le nombre d'options *k* assignées dans le bloc excède la valeur r_k. Si c'est le cas, il y a conflit. Il est à noter qu'il existe des algorithmes plus performants que celui-ci, mais ce dernier est utilisé ici pour illustrer clairement comment évaluer une solution.

```
nombre_conflits = 0
pour k = 1 à opt
        pour j = 1 à nc – s_k + 1
                somme_bloc = 0
                pour l = j à j + s_k
                        somme_bloc = somme_bloc + S_kl
                si somme_bloc > r_k
                        nombre_conflits = nombre_conflits + 1
        retourner nombre_conflits
```

Figure 3.1 : Algorithme pour l'évaluation d'une solution

La Figure 3.2 montre un segment d'une solution de l'instance présentée au Tableau 3.1. Les options 1 et 3 ne causent aucun conflit dans le segment de solution présenté. Par contre, pour l'option 2, il existe un conflit à la position i+1, car la contrainte de capacité 2/3 n'est pas respectée. Il en va de même pour l'option 4, à la position i, et pour l'option 5, à la position i+1. On peut conclure qu'il existe trois conflits dans ce segment de la solution.

Segment de la solution pour le problème du Tableau 3.1 :

Positions :	...	i	i+1	i+2	i+3	i+4	i+5	...
Classes :	...	8	5	4	3	6	9	...

Construction de la matrice S à partir de ce segment de solution :

Option 1 (1/2) :	1	0	0	1	0	1
Option 2 (2/3) :	0	1	1	1	0	0
Option 3 (1/3) :	0	0	0	0	0	1
Option 4 (2/5) :	1	0	1	0	1	0
Option 5 (1/5) :	0	1	0	0	0	1

Figure 3.2 : Segment d'une solution

Le nombre de voitures n'est pas suffisant pour évaluer la difficulté d'une instance de *POV*. La moyenne du taux d'utilisation des options est généralement utilisée pour fournir une meilleure idée de cette difficulté. On définit, tout d'abord, le taux d'utilisation d'une option k par le ratio entre le nombre de véhicules requérant l'option k (a_k) et le nombre de véhicules pouvant posséder cette option pour respecter la contrainte de capacité. L'Équation 3.1 exprime le taux d'utilisation d'une option.

$$Util_k = \frac{a_k}{\lceil nc(r_k / s_k) \rceil}, \text{ où } a_k = \sum_{i=1}^{cl} opt_{ki} nc_i \qquad (3.1)$$

Si $Util_k > 1$, on dit que l'option k est sur-contrainte. Une instance comprenant une option sur-contrainte n'a pas de solution sans conflits. Toutefois, une instance n'ayant aucune option sur-contrainte ne garantit pas qu'elle possède une solution sans conflits. On peut alors calculer la difficulté d'une classe de voiture v en additionnant les taux d'utilisation des options qui la composent selon l'Équation 3.2.

$$d_v = \sum_{k=1}^{opt} o_{vk} Util_k \qquad (3.2)$$

Par exemple, la difficulté de la classe de voiture 7 (d_7), pour l'instance décrite au Tableau 3.1, se calcule comme suit : cette classe de voitures comprend les options 1, 2 et 3. Commençons par calculer $Util_1$, le taux d'utilisation de l'option 1. L'instance contient 25 voitures (nc), la contrainte de capacité de l'option 1 est 1 / 2 et on a $a_1 = 8$, le nombre de véhicules demandant l'option 1 à produire. On obtient alors $Util_1 = a_1 / (nc * (r_1 / s_1)) = 8 / (25 * (1 / 2)) = 0{,}62$. De la même façon, on trouve $Util_2 = 1{,}00$ et $Util_3 = 0{,}89$. Finalement, pour calculer d_7, on n'a qu'à faire la sommation des taux d'utilisation des options 1, 2 et 3. Donc, $d_7 = Util_1 + Util_2 + Util_3 = 0{,}62 + 1{,}00 + 0{,}89 = 2{,}51$. Dans le Tableau 3.2, on retrouve la difficulté des 12 classes de voitures de l'instance de *POV* présentée au Tableau 3.1.

	Classes de voitures											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Difficulté (*d*)	1,00	2,31	1,62	1,90	1,80	0,90	2,51	1,52	1,51	0,89	1,89	2,52

Tableau 3.2 : Difficulté des classes de voitures

Lorsqu'on construit une solution, il est préférable de placer, le plus tôt possible, les classes de voitures dont la difficulté est la plus élevée. Moins il reste de positions disponibles dans la séquence, plus les classes fortement contraignantes seront difficiles à placer sans qu'elles ne causent de conflit. Dans l'exemple du Tableau 3.2, il serait préférable de placer les classes de voitures 2, 7 et 12 le plus rapidement possible dans la séquence tout en retardant les classes 6 et 10. Il est également possible de calculer dynamiquement la difficulté des classes de voitures au fur et à mesure que la construction d'une solution progresse. De cette façon, le choix de la prochaine voiture à ajouter à la solution est plus éclairé, car la difficulté reflète la situation actuelle du problème.

3.3 Approches de solution pour le POV

Cette section a pour but de décrire le fonctionnement de deux approches de solution pour le *POV*. Avant de proposer une hybridation intégrant ces deux approches, il est nécessaire de bien comprendre leur fonctionnement individuel. Dans un premier temps, nous présenterons un algorithme d'optimisation par colonies de fourmis et, par la suite, une méthode basée sur la programmation linéaire en nombres entiers (*PLNE*).

3.3.1 Optimisation par colonies de fourmis

L'*OCF* est une métaheuristique efficace pour résoudre une grande variété de problèmes d'optimisation combinatoire. Voyons comment Gravel, Gagné *et al.* [42] l'ont adaptée pour la résolution du *POV*. À chaque cycle de la métaheuristique, les fourmis construisent une solution en ajoutant une classe de voitures à la fois. Le choix de cette classe est fait à l'aide de la règle de transition, qui favorise celles dont la trace de

phéromone est la plus intense et qui sont les plus désirables selon les fonctions de visibilité. Les éléments de la règle de transition sont décrits un peu plus loin dans cette section. Après qu'une voiture ait été ajoutée à la séquence, une mise à jour locale est effectuée à la matrice de phéromone dans le but d'éviter de répéter les mêmes choix. Une fois que toutes les fourmis aient complété leur solution respective et que ces solutions aient été évaluées, on effectue une recherche locale sur S^{bc}, la meilleure solution du cycle. Cette solution sert alors à effectuer la mise à jour globale de la matrice de phéromone. On vérifie si cette solution peut remplacer S^{bg}, la meilleure solution trouvée jusqu'ici par la métaheuristique, et on commence un autre cycle. Une fois tous les cycles de l'algorithme complétés, une autre recherche locale est exécutée sur S^{bg}. Le pseudo-code de la Figure 3.3 présente le fonctionnement de l'OCF. Le paramètre t_{Max} est le nombre de cycles à effectuer et *nb_fourmis* est, comme son nom l'indique, le nombre de fourmis.

Lorsque appliqué sur S^{bg}, la recherche locale utilise un voisinage qui inverse l'ordre d'une sous-séquence de voitures. Premièrement, une position est choisie aléatoirement parmi une séquence de longueur s_k contenant un conflit. Une seconde position est choisie aléatoirement parmi toutes les positions pour ainsi former la sous-séquence. Cette sous-séquence est alors inversée, en incluant les bornes. Ce processus est effectué 2**nc* fois ou est arrêté si une solution sans conflits est trouvée. Le voisinage *Lin2Opt* de Puchta et Gottlieb [67] est utilisé pour réaliser la recherche locale sur S^{bg}. Ce voisinage consiste simplement à sélectionner aléatoirement deux positions et à inverser la sous-séquence. Le processus est répété 2000**nc* fois ou est arrêté si une solution sans conflits est trouvée.

```
t = 0
Initialiser la matrice de phéromone τ(t) à τ0 pour chaque paire de classes uv
Pour t = 1 à tMax
        Choisir une classe aléatoirement pour la 1re position de la séquence de fourmis
        Pour pos = 2 à nc
                Pour m = 1 à nb_fourmis
                        Choisir une classe de voiture v avec l'Équation 2.3
                        Mise à jour locale de la trace de phéromone entre les classes u et v
        Évaluer la solution (S) des nb_fourmis fourmis
        Déterminer la meilleure solution du cycle S^bc et sa valeur F+
        Effectuer une recherche locale sur S^bc
        Mise à jour globale de la trace de phéromone avec S^bc
        Mise à jour de S^bg, la meilleure solution trouvée à ce jour et de sa valeur F*
Effectuer une recherche locale sur S^bg
```

Figure 3.3 : Pseudo-code de l'OCF proposé par Gravel, Gagné *et al.* [42]

La règle de transition utilisée par la métaheuristique est donnée par l'Équation 3.3. Voici une description des variables impliquées dans la règle de transition : τ_{uv} est la quantité de phéromone accumulée entre les classes de voitures u et v, $new\_conflicts_v$ est le nombre de conflits qui surviendront si une voiture de classe v est assignée à la prochaine position libre de la séquence et d_v représente la difficulté de la classe v (voir Équation 3.2). Il faut noter que la difficulté est calculée dynamiquement en fonction du nombre de positions (nc') restant à compléter dans la séquence et de la quantité restante de voitures demandant l'option k (a_k). On a $\ell \notin Tabou_m$ qui signifie que les classes possibles à ajouter dans la séquence sont uniquement celles pour lesquelles il reste au moins une voiture non placée pour la fourmi m. La variable q est un nombre aléatoire et q_0 est un paramètre; toutes deux ont une valeur entre 0 et 1. La valeur de q_0 est un seuil permettant d'orienter la recherche entre l'exploitation des informations accumulées par la matrice de phéromone (Équation 3.3) et l'exploration vers de nouvelles solutions (Équation 3.4). Finalement, α, β, et δ sont

des paramètres exprimant le niveau d'influence de la trace de phéromone et des différentes visibilités sur le choix de la prochaine classe de voiture à assigner.

$$v = \begin{cases} \arg\max_{\ell \notin Tabou_m} \left\{ [\tau_{u\ell}(t)]^{\alpha} \left[\frac{1}{1 + new\_conflicts_{\ell}} \right]^{\beta} [d_{\ell}] \right\} & \text{si } q \leq q_0 \\ V \end{cases} \quad (3.3)$$

où V est choisi selon la probabilité

$$p_{uv}^{m}(t) = \frac{[\tau_{uv}(t)]^{\alpha} \left[\frac{1}{1 + new\_conflicts_{v}} \right]^{\beta} [d_v]^{\delta}}{\sum_{\ell \notin Tabou_m} [\tau_{u\ell}(t)]^{\alpha} \left[\frac{1}{1 + new\_conflicts_{\ell}} \right]^{\beta} [d_{\ell}]^{\delta}} \quad (3.4)$$

Le choix d'une voiture de classe v par la règle de transition s'effectue parmi une liste de candidats. Les candidats sont les classes de voitures qui n'engendreront aucun conflit si elles sont ajoutées à la séquence de voitures. Si toutes les classes de voitures causent un conflit, alors elles sont toutes candidates.

Lorsqu'une voiture de classe v est assignée après une voiture de classe u, on utilise la mise à jour locale pour diminuer la quantité de phéromone entre ces deux classes. Cela évite que trop de fourmis effectuent le même choix de classe de voitures au même moment. Cela permet également d'augmenter la diversité de solutions trouvées. On retrouve la formule de la mise à jour locale à l'Équation 3.5. Le paramètre ρ_{ℓ} représente la persistance locale de la trace de phéromone, c'est-à-dire la rapidité de son évaporation.

$$\tau_{uv}(t) = \rho_{\ell}\tau_{uv}(t) + (1 - \rho_{\ell})\Delta\tau_{uv}, \text{ où } \Delta\tau_{uv} = \tau_0 \quad (3.5)$$

Quant à la mise à jour globale, elle vise à augmenter la quantité de phéromone sur toutes les paires de classes de voitures faisant partie de S^{bc}. Du même coup, une partie de la

phéromone laissée sur les paires de classes de voitures ne faisant pas partie de S^{bc} est évaporée. Cela fait en sorte que les « bonnes » paires de voitures seront choisies plus souvent que les autres. La formule de la mise à jour globale est donnée à l'Équation 3.6.

$$\tau_{uv}(t+1) = \rho_g \tau_{uv}(t) + (1-\rho_g)\Delta\tau_{uv}(t) \qquad (3.6)$$

où $\Delta\tau_{uv}(t) = F^* / F_+$ et ρ_g est la persistance de la trace de phéromone.

Discutons maintenant de la performance obtenue par cette métaheuristique telle que présentée dans Gravel, Gagné *et al.* [42] pour les trois ensembles de problèmes. Les résultats des problèmes de l'*ET1* ne sont pas présentés puisque l'*OCF* proposé résout ceux-ci sans conflit en moins d'une seconde. Le Tableau 3.3 présente les résultats obtenus par la métaheuristique sur les instances de l'*ET2*, en utilisant un ordinateur muni d'un processeur Xeon 3,06 Ghz avec 1024 Mo de mémoire vive. Toutefois, cette version de l'*OCF* n'utilise pas les méthodes de recherche locale présentées précédemment. L'*OCF* a été exécutée 100 fois sur chaque instance de problème et les différents paramètres de l'algorithme ont été assignés comme suit : $\tau_0 = 0{,}005$, $\rho_g = \rho_\ell = 0{,}99$, *nb_fourmis* = 15, $q_0 = 0{,}9$, $t_{Max} = 1000$, $\alpha = 1$, $\beta = 6$ et $\delta = 3$. La solution optimale des problèmes 26-82, 41-66 et 4-72 ainsi que la meilleure solution connue du problème 6-76 est trouvée en tout temps. Quant aux problèmes 16-81, 19-71, 36-92 et 21-90, la solution optimale ou la meilleure solution connue, selon le cas, est trouvée fréquemment, mais pas à tout coup. Le problème 10-93 nous donne plus de fil à retordre, car sa meilleure solution connue n'est trouvée qu'occasionnellement. Les colonnes *minimum* et *maximum* représentent respectivement les valeurs minimale et maximale du nombre de conflits obtenus dans les solutions trouvées.

La colonne *cycle moyen* donne la moyenne du nombre de cycles nécessaires à l'*OCF* pour trouver la solution S^{bg}. On utilise couramment cette mesure pour évaluer la performance d'un *OCF*. On remarque que le cycle moyen est peu élevé par rapport aux 1000 cycles effectués par l'*OCF*. Enfin, la moyenne des temps d'exécution est exprimée en secondes. Il faut noter que l'algorithme s'arrête lorsqu'une solution sans conflits est trouvée.

Problème	Nombre de conflits				Cycle moyen	Moyenne des temps d'exécution (secondes)
	Moyenne	Écart-type	Minimum	Maximum		
16-81	0,1	0,33	0	1	329,10	4,23
26-82	0,0	0,00	0	0	83,72	0,74
41-66	0,0	0,00	0	0	5,84	0,04
4-72	0,0	0,00	0	0	58,72	0,50
10-93	4,2	0,47	3	5	298,93	10,10
19-71	2,1	0,27	2	3	309,48	9,26
6-76	6,0	0,00	6	6	1,01	7,94
36-92	2,3	0,46	2	3	346,02	8,50
21-90	2,6	0,49	2	3	324,39	8,73

Tableau 3.3 : Résultats de l'*OCF* sans recherche locale sur les problèmes de l'*ET2* (Gravel, Gagné *et al.* [42])

Les problèmes de l'*ET2* ont également été résolus par l'*OCF* avec les procédures de recherche locale décrites précédemment. Les résultats de ces essais numériques sont présentés au Tableau 3.4. À l'exception du problème 10-93, l'utilisation de la recherche locale a permis de trouver la meilleure solution connue pour tous les essais et de réduire le nombre de cycles nécessaires pour y parvenir. Étant donné que l'algorithme s'arrête une fois qu'une solution de ce type est trouvée, le temps d'exécution nécessaire a été diminué

pour les problèmes ayant une solution sans conflits. Par contre, le temps d'exécution est plus élevé pour les problèmes n'ayant pas de solution sans conflits.

Problème	Nombre de conflits				Cycle moyen	Moyenne des temps d'exécution (secondes)
	Moyenne	Écart-type	Minimum	Maximum		
16-81	0,0	0,00	0	0	149,03	1,75
26-82	0,0	0,00	0	0	28,33	0,26
41-66	0,0	0,00	0	0	2,98	0,03
4-72	0,0	0,00	0	0	30,54	0,27
10-93	3,8	0,45	3	5	507,22	13,79
19-71	2,0	0,00	2	2	125,61	13,04
6-76	6,0	0,00	6	6	1,00	11,76
36-92	2,0	0,00	2	2	145,27	12,66
21-90	2,0	0,00	2	2	146,81	12,60

Tableau 3.4: Résultats de l'*OCF* avec recherche locale sur les problèmes de l'*ET2* (Gravel, Gagné *et al.* [42])

Les Tableaux 3.5 et 3.6 présentent les résultats obtenus par l'*OCF* pour les problèmes de l'*ET3*, sans et avec recherche locale, respectivement. Encore une fois, l'*OCF* a été exécutée 100 fois sur chaque instance, en utilisant un ordinateur muni d'un processeur Xeon 3,06 Ghz avec 1024 Mo de mémoire vive. On retrouve, dans cet ensemble de problèmes, des instances plus difficiles à résoudre. Les résultats présentés au Tableau 3.5 sont les premiers obtenus pour l'*ET3*; ils établissent une première base de comparaison. On remarque que le cycle moyen où l'on trouve la meilleure solution est relativement bas. Il en va de même pour le temps d'exécution, malgré la grande taille des problèmes. L'utilisation de la recherche locale (Tableau 3.6) permet d'améliorer les résultats précédents et a, entre autres, permis de prouver que les problèmes 200_01, 200_07, 300_01, 300_07, 400_05, 400_06 et 400_10 possèdent une solution sans conflits. À

l'exception du problème 200_08, la recherche locale a permis d'améliorer le nombre de conflits moyen trouvé. De plus, le nombre de conflits maximum trouvé a été diminué pour la majorité des problèmes. Par contre, le cycle moyen où la meilleure solution est trouvée a été augmenté, ainsi que le temps d'exécution moyen. Donc, l'ajout de la recherche locale a une grande influence sur la qualité des solutions trouvées.

Problème	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
	Moyenne	Écart-type	Minimum	Maximum		
200_01	3,80	0,62	2	5	293,44	19,46
200_02	4,14	0,51	3	5	422,62	19,42
200_03	8,90	0,59	8	10	350,44	19,70
200_04	9,86	0,51	8	11	274,59	20,49
200_05	8,81	0,42	8	10	201,67	18,28
200_06	6,87	0,39	6	8	232,09	19,03
200_07	2,99	0,36	2	4	344,48	18,95
200_08	8,00	0,00	8	8	51,06	17,31
200_09	11,85	0,48	11	13	284,31	19,93
200_10	21,44	0,57	20	23	313,96	18,65
300_01	5,33	0,71	4	7	386,80	30,23
300_02	13,15	0,39	13	15	307,19	30,68
300_03	14,54	0,54	13	16	370,63	32,26
300_04	10,33	0,68	9	12	390,43	30,25
300_05	40,55	1,06	36	43	371,53	29,99
300_06	7,59	0,74	6	9	255,48	30,21
300_07	2,89	0,63	1	4	173,35	30,07
300_08	9,17	0,38	9	10	229,22	29,00
300_09	9,05	0,56	8	10	274,34	28,76
300_10	34,63	1,04	32	37	237,32	28,57
400_01	3,01	0,56	2	4	201,01	41,77
400_02	23,28	0,82	21	25	280,48	38,95
400_03	11,65	0,50	10	12	212,39	38,40
400_04	21,96	0,75	20	24	454,46	43,57
400_05	3,48	0,96	1	6	346,67	36,47
400_06	4,20	0,97	1	6	263,12	39,85
400_07	7,65	0,83	5	9	260,60	36,93
400_08	11,54	1,62	6	14	179,93	35,51
400_09	17,98	1,10	15	20	209,19	41,38
400_10	4,24	0,99	2	7	77,19	40,94

Tableau 3.5: Résultats de l'*OCF* sans recherche locale sur les problèmes de l'*ET3* (Gravel, Gagné *et al.* [42])

Problème	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
	Moyenne	Écart-type	Minimum	Maximum		
200_01	1,00	0,67	0	3	985,79	33,42
200_02	2,41	0,49	2	3	962,63	33,44
200_03	6,04	0,82	4	8	998,58	34,45
200_04	7,57	0,56	7	9	968,62	34,66
200_05	6,40	0,51	6	8	986,70	32,01
200_06	6,00	0,00	6	6	314,36	32,52
200_07	0,00	0,00	0	0	986,47	28,65
200_08	8,00	0,00	8	8	45,98	31,10
200_09	10,00	0,00	10	10	988,06	34,00
200_10	19,09	0,29	19	20	998,96	32,68
300_01	2,15	0,77	0	4	990,38	65,29
300_02	12,02	0,14	12	13	749,79	66,38
300_03	13,06	0,24	13	14	858,31	67,32
300_04	8,16	0,81	7	10	966,18	65,99
300_05	32,28	1,10	29	35	1000	66,09
300_06	4,38	0,99	2	7	983,85	65,92
300_07	0,59	0,64	0	2	973,01	62,64
300_08	8,00	0,00	8	8	869,05	63,21
300_09	7,46	0,67	7	9	885,19	63,47
300_10	22,60	0,85	21	25	1000	63,91
400_01	2,52	0,63	1	4	695,43	96,84
400_02	17,37	0,90	16	20	1000	96,70
400_03	9,91	0,43	9	11	954,46	94,87
400_04	19,01	0,10	19	20	994,68	98,98
400_05	0,01	0,10	0	1	979,21	76,85
400_06	0,33	0,53	0	2	997,54	87,56
400_07	5,44	0,80	4	7	989,49	93,29
400_08	5,30	0,81	4	7	1000	93,39
400_09	7,63	0,95	5	10	1000	97,75
400_10	0,95	0,72	0	3	991,09	93,89

Tableau 3.6: Résultats de l'*OCF* avec recherche locale sur les problèmes de l'*ET3* (Gravel, Gagné *et al.* [42])

3.3.2 Programme linéaire en nombres entiers

La programmation linéaire en nombres entiers (*PLNE*) permet de formuler et de résoudre différents problèmes d'optimisation en utilisant, par exemple, une méthode telle le Branch & Bound (*B&B*). Dans cette section, on présente, dans un premier temps, une formulation du *POV* en *PLNE*. On montre, dans un deuxième temps, comment cette méthode exacte performe sur les instances de problèmes.

La formulation du *POV* sous la forme de *PLNE* peut se faire en utilisant deux variables de décision. La première variable de décision, C_{ij}, est de type booléen. Elle définit l'affectation de la classe de voiture i à la position j dans la séquence si C_{ij}=1. La deuxième variable de décision, Y_{kj}, est également de type booléen. Elle permet de vérifier si, pour l'option k, la contrainte de capacité r_k / s_k, dans une sous-séquence de longueur s_k et commençant à la position j de la séquence de voitures, est respectée ($Y_{kj}=0$) ou non ($Y_{kj}=1$). On utilise M, un scalaire pouvant être fixé à $s_k - r_k$, pour formuler adéquatement les contraintes de capacité de façon à identifier le nombre de conflits. Ci-dessous le *PLNE* proposé par Gravel, Gagné *et al.* [42] pour le *POV* :

$$\text{Minimiser} \quad F = \sum_{k=1}^{opt} \sum_{j=1}^{nc-sk+1} Y_{kj} \qquad (3.7)$$

avec les contraintes

$$\sum_{i=1}^{cl} C_{ij} = 1, \text{ pour } j = 1 \text{ à } nc \qquad (3.8)$$

$$\sum_{j=1}^{nc} C_{ij} = n_i, \text{ pour } i = 1 \text{ à } cl \qquad (3.9)$$

$$\sum_{i=1}^{cl} \sum_{l=j}^{j+sk-1} o_{ik} \times C_{il} \leq r_k + M \times Y_{kj},$$

$$\text{pour } k = 1 \text{ à } opt \text{ et } j = 1 \text{ à } nc - sk + 1 \qquad (3.10)$$

L'Équation 3.7 est la fonction objectif du programme linéaire. On tente de minimiser le nombre de fois où la contrainte de capacité liée à une option est dépassée. Autrement dit, pour chaque option k et chaque sous-séquence de longueur s_k, on cherche à respecter autant que possible le ratio r_k / s_k. L'Équation 3.8 nous assure qu'il n'y ait qu'une seule classe de voiture assignée à chacune des positions de la séquence. L'Équation 3.9 fait en sorte que toutes les voitures soient placées dans la séquence. Enfin, c'est l'Équation 3.10 qui formalise la contrainte de capacité de production r_k / s_k et permet d'être dépassée dans le cas où il est impossible de trouver une solution sans conflits.

Le *PLNE* proposé par Gravel, Gagné *et al.* [42] a été utilisé pour résoudre les problèmes de l'*ET1* et de l'*ET2*. Les essais numériques ont été effectués avec le programme *Xpress* sur un ordinateur doté d'un processeur Xeon 3,06 Ghz avec 1024 Mo de mémoire vive (pour plus de détails, consulter Gravel, Gagné *et al.* [42]). Dans le cas de l'*ET1*, le *PLNE* a réussi à trouver une solution sans conflit pour chacun des problèmes. Les résultats concernant l'*ET2* sont présentés au Tableau 3.7. Les problèmes de l'*ET2* ayant

une solution sans conflits (16-81, 26-82, 41-66 et 4-72) ont été résolus rapidement. Pour les problèmes 10-93, 19-71, 6-76 et 36-92, pour lesquels on sait qu'il n'existe pas de solution sans conflits, ainsi que pour le problème 21-90, dont on ignore le statut, le modèle linéaire n'a pu être solutionné. Tout au plus, la meilleure solution connue a été trouvée, après un certain temps de calculs, en fournissant cette valeur de cette solution comme borne supérieure pour accélérer l'exécution de l'algorithme. Pour ces raisons, les problèmes de l'*ET3* n'ont pas été testés avec le PLNE.

Problème	**16-81**	**26-82**	**41-66**	**4-72**	**10-93**	**19-71**	**6-76**	**36-92**	**21-90**
Nombre de conflits	0	0	0	0	3	2	6	2	2
Temps en secondes	30	21	10	15	61135*	4*	75*	195*	34*

* Temps pour trouver la meilleure solution connue

Tableau 3.7 : Résolution des problèmes de l'*ET2* avec un *PLNE* (Gravel, Gagné *et al.* [42])

Étant donné la difficulté à résoudre de façon optimale la plupart des instances citées précédemment, on pourrait, en revanche, s'intéresser à la résolution optimale des sous-problèmes du problème original. À partir d'une solution générée aléatoirement ou par une heuristique quelconque, on utilise alors le PLNE dans le processus itératif pour tenter d'améliorer cette solution en résolvant différents sous-problèmes.

La Figure 3.4 montre comment on peut créer un sous-problème à partir d'une solution. Il s'agit de fixer certaines positions de la séquence et de laisser le *PLNE* « remplir » les cases vides. À chaque essai, on accorde un délai précis au *PLNE* pour la recherche de solutions, ce qui fait que celles-ci peuvent être incomplètes. Si tel est le cas, on prend la meilleure solution trouvée pendant cette recherche. Si, dans le pire cas, aucune

solution n'est trouvée, la solution demeure la même. Pour accélérer la recherche, on spécifie aussi au *PLNE* une borne supérieure, basée sur le nombre de conflits de la solution donnée en entrée. On parlera de « recherche locale exacte » (*RLE*), dans la suite du document, pour désigner cette approche de solution, puisque celle-ci explore entièrement une région particulière de l'espace de solutions.

Segment d'une solution pour le problème du Tableau 3.1 :

Positions :	...	i	$i+1$	$i+2$	$i+3$	$i+4$	$i+5$	$i+6$	$i+7$	...
Classes :	...	8	5	4	3	6	9	1	7	...

Après avoir fixé les positions i, $i+2$, $i+3$ et $i+5$, on obtient le sous-problème suivant :

Positions :	...	i	$i+1$	$i+2$	$i+3$	$i+4$	$i+5$	$i+6$	$i+7$	...
Classes :	...	8	__	4	3	__	9	__	__	...

Les espaces libres doivent être remplis.

Figure 3.4 : Création d'un sous-problème

La Figure 3.5 illustre le processus *RLE* décrit précédemment. On spécifie le nombre d'essais qui seront effectués avec le paramètre *nb_itérations* et le nombre de secondes accordées au *PLNE* pour résoudre le sous-problème avec le paramètre *nb_sec*. Évidemment, on doit aussi fournir la solution de départ *S*. Nous allons maintenant proposer d'autres stratégies pour créer les sous-problèmes. Les sections suivantes expliquent en détails en quoi consistent ces stratégies et comment déterminer la taille des sous-problèmes et donnent une présentation des essais numériques effectués.

```
S = solution donnée en paramètre
Pour i = 1 à nb_itérations
        Générer un sous-problème sp à partir de S
        T = Résolution de sp par le PLNE en nb_sec secondes
        Si T ≤ S
                S = T
Retourner S
```

Figure 3.5 : Pseudo-code de la recherche locale exacte

3.3.2.1 Stratégies pour la construction de sous-problèmes

La création d'un sous-problème à partir d'une solution d'une instance de *POV* se fait en fixant un ensemble de positions dans la séquence. Par conséquent, les classes de voitures assignées aux positions fixées ne peuvent être déplacées par la méthode exacte. Il suffit alors d'ajouter de nouvelles contraintes au *PLNE* pour fixer les positions voulues. Par exemple, si on veut conserver une voiture de classe v située à la position j, il suffit d'ajouter la contrainte $C_{vj} = 1$. Les classes de voitures situées dans les positions restantes, dites libres, peuvent être déplacées à n'importe quelle autre position libre. De cette façon, on obtient un problème plus petit, donc plus facile à résoudre. Les stratégies servant à la création de sous-problèmes reçoivent en paramètre le pourcentage de positions libres par rapport à la taille du problème. Donc, un sous-problème construit à partir d'une solution de 300 voitures, avec un pourcentage de positions libres de 10%, comptera 30 classes de voitures pouvant être déplacées. Lorsqu'on parle de la taille d'un sous-problème, on fait référence au nombre de positions libres. Plus la taille d'un sous-problème est grande, plus il compte de positions libres, et inversement.

Le premier groupe de stratégies examinées comporte trois stratégies principalement basées sur l'aléatoire. Dans la stratégie 1 (S1), toutes les positions libres sont choisies aléatoirement. La stratégie 2 (S2) fonctionne exactement comme la S1, à l'exception que l'on s'assure qu'au moins une position faisant partie d'un conflit soit une position libre. Si toutes les positions faisant partie de conflits étaient fixées, il serait alors impossible d'améliorer la solution. La stratégie 3 (S3) choisit aléatoirement des positions libres parmi les positions faisant partie de conflits. Autrement dit, on tire au hasard les positions libres parmi les positions faisant partie de conflits jusqu'à ce que le nombre requis de positions libres soit atteint ou qu'il ne reste plus de positions faisant partie d'un conflit à libérer. Si la quantité demandée de positions libres n'est pas atteinte, on libère des positions qui ne font pas partie de conflits jusqu'à ce que le critère soit satisfait.

Le deuxième groupe de stratégies examinées comporte deux stratégies principalement basées sur des blocs consécutifs de voitures. La stratégie 4 (S4) vise à laisser une certaine quantité ($max\ s_k$) de positions libres à gauche et à droite d'une position faisant partie d'un conflit. Plus précisément, on choisit aléatoirement une position i faisant partie d'un conflit. On libère les positions en partant de la position $i - max\ s_k$ à $i + max\ s_k$, ou jusqu'à ce que le nombre requis de positions libres ait été atteint. Ce processus est répété tant que le nombre de positions libres n'est pas atteint ou que les positions faisant partie d'un conflit sont libérées. Si nécessaire, on complète la construction du sous-problème en libérant des blocs autour de positions choisies au hasard. Pour sa part, la stratégie 5 (S5) libère un bloc autour d'une position choisie aléatoirement et faisant partie

d'un conflit. Le reste des positions libres est ensuite choisi aléatoirement parmi toutes les autres positions de la séquence.

Le troisième groupe de stratégies examinées comporte quatre stratégies, principalement basées sur la notion de difficulté d'une classe. La stratégie 6 (S6) libère les positions contenant les classes de voitures dont la difficulté (Équation 3.3) est la plus faible. Plus précisément, on balaie du début à la fin la séquence en libérant au passage les positions dont la difficulté est la plus faible. Ensuite, on retourne au début de la séquence et on libère les positions ayant la seconde difficulté la plus faible, et ainsi de suite jusqu'à ce qu'on atteigne le nombre de positions libres requis. Quant à la stratégie 7 (S7), elle procède de la même façon que la S6, mais elle s'assure qu'au moins une position faisant partie d'un conflit devienne libre. La stratégie 8 (S8) fonctionne aussi comme la S6, sauf qu'on commence par libérer les positions faisant partie de conflits dont la difficulté est la plus faible. S'il y a moins de positions faisant partie de conflits que de positions libres, on compense en libérant les positions qui ne font pas partie de conflits. S'il y a plus de positions faisant partie de conflits que de positions libres, les positions libres sont choisies aléatoirement parmi celles en conflit. Finalement, la stratégie 9 (S9) effectue des tournois de trois concurrents pour sélectionner les positions à libérer. Les concurrents du tournoi sont des positions dans la séquence et le gagnant est la position contenant la classe de voitures dont la difficulté est la plus faible. En cas d'égalité, c'est la première position qui est choisie.

Examinons maintenant comment déterminer efficacement la taille des sous-problèmes.

3.3.2.2 Taille des sous-problèmes

La taille des sous-problèmes est un facteur qui influence grandement la capacité d'amélioration d'une solution. Une trop petite taille laisse une moins grande marge de manœuvre pour éliminer les conflits et ainsi améliorer la solution, tandis qu'un trop grand sous-problème risque de prendre beaucoup de temps pour être résolu ou, tout simplement, de ne pas être résolu dans le temps imparti. Il s'agit de trouver un juste milieu entre le pourcentage de positions libres et le temps alloué au *PLNE* pour faire la résolution. Deux méthodes ont été utilisées pour déterminer la taille des sous-problèmes. Une première méthode, dite statique, utilise le même pourcentage de positions libres tout au long du *RLE*. L'autre méthode, dite dynamique, ajuste le pourcentage de positions libres en fonction de la capacité du *PLNE* à résoudre les sous-problèmes donnés.

L'utilisation du pourcentage de positions libres statique fait en sorte que la taille des sous-problèmes demeure la même pendant toute la durée d'une exécution de l'algorithme. Toutefois, pour créer de la variabilité dans les sous-problèmes générés par les différentes stratégies, le pourcentage de positions libres varie aléatoirement dans un intervalle de ±2% autour du pourcentage choisi. Par exemple, si on veut un taux de 35% de positions libres, le pourcentage de positions libres sera choisi aléatoirement dans l'intervalle [33%, 37%] à chaque appel du *RLE*.

Le pourcentage de positions libres dynamique, quant à lui, s'ajuste en fonction de la capacité du *PLNE* à solutionner un sous-problème. Cette méthode comporte deux phases, utilisées de façon cyclique. La première phase consiste à trouver, en commençant par une valeur de 5%, le pourcentage maximum de positions libres que peut supporter la résolution

du *PLNE* pour une stratégie donnée. Si le *PLNE* parvient à trouver une solution faisable dans le temps imparti, on incrémente le pourcentage de 5% au prochain appel du *PLNE*. Sinon, on a un échec et cette stratégie demeure au même pourcentage. Après cinq échecs consécutifs, on considère que le pourcentage précédent permet d'obtenir le plus grand sous-problème pouvant être résolu par le *PLNE* et on passe à la deuxième phase. Il s'agit alors de choisir, à chaque appel du *PLNE*, un pourcentage de positions libres selon une distribution linéaire dans l'intervalle [(2/3) * maximum_pourcentage, maximum_pourcentage]. La distribution linéaire assure un choix diversifié du pourcentage de positions libres tout en favorisant les pourcentages les plus élevés. Après un nombre prédéfini d'appels (100 ou autre si spécifié autrement), on revient à la première phase pour ajuster le pourcentage de positions libres à la hausse ou à la baisse, selon la difficulté des sous-problèmes rencontrés.

3.3.2.3 Performance des stratégies

Cette section présente les différents essais numériques réalisés pour établir la performance des différentes stratégies de création de sous-problèmes. On utilise un ensemble restreint de problèmes ainsi qu'un nombre limité d'essais pour conduire ces tests, car les temps d'exécution à chaque essai sont élevés. Les problèmes utilisés sont le 200_05, 300_05 et 400_02 de l'*ET3*. Ces trois problèmes ont été choisis spécifiquement afin d'avoir un représentant de chaque taille de problème, soit 200, 300 et 400 voitures. Il s'agit également de problèmes où l'OCF réussit plus difficilement à trouver de bonnes solutions. À chaque essai, 1000 appels sont faits au *PLNE* pour solutionner un sous-

problème dans un temps maximum de 5 secondes à chaque appel. Nous discuterons du temps alloué à CPLEX pour la résolution du PLNE à la Section 3.5. Les essais sont répétés à cinq reprises pour établir une moyenne des résultats obtenus. On utilise les librairies C++ d'ILOG CPLEX 9.0 pour résoudre les sous-problèmes.

Dans un premier temps, la performance de l'utilisation d'une taille de sous-problème statique versus dynamique pour une stratégie donnée sera établie. Le Tableau 3.8 présente les résultats obtenus par la méthode *RLE* en utilisant la stratégie 2 (choix arbitraire) pour la création des sous-problèmes pour les problèmes 300_05 et 400_02. Des essais préliminaires, qui ne sont pas présentés dans ce travail, ont montré qu'il n'était pas nécessaire de faire des essais sur le problème 200_05, car la meilleure solution connue était atteinte à chaque essai. On utilise des pourcentages de positions libres statiques allant de 5% à 40%. Les solutions de départ utilisées ont été générées par le premier cycle d'un *OCF*. Ceci permet de commencer avec une solution de meilleure qualité que si elle avait été générée aléatoirement. Pour le problème 300_05 (respectivement 400_02), la solution de départ compte 54 conflits (respectivement 30 conflits).

On peut observer que le nombre moyen de conflits dans les solutions trouvées diminue à mesure que le pourcentage de positions libres augmente. Cependant, une fois le cap des 35% de positions libres dépassé, le nombre moyen de conflits augmente. On remarque aussi que le temps d'exécution augmente de façon exponentielle avec le pourcentage de positions libres. On observe que les petits sous-problèmes sont résolus facilement sans toutefois réussir à améliorer suffisamment les solutions. D'un autre côté, les grands sous-problèmes sont trop difficiles à résoudre dans les délais prescrits. Ici,

l'expression « trop, c'est comme pas assez » prend tout son sens. Il s'avère que 35% semble être le pourcentage de positions libres statique idéal dans les conditions présentes. Il est à noter qu'il pourrait en être tout autrement avec des conditions différentes, par exemples, pour une autre instance de problème, pour une autre stratégie de création de sous-problèmes ou encore pour une autre durée accordée à la résolution du *PLNE*.

Problème	Pourcentage de positions libres	Nombre de conflits				Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum	
300_05	5%	44	1,73	41	45	266,6
300_05	10%	43,2	0,84	42	44	295,4
300_05	15%	40	1,58	38	42	357
300_05	20%	37,4	1,82	35	39	513,6
300_05	25%	35	1,22	34	37	907,2
300_05	30%	34	1,58	32	36	1837,2
300_05	35%	32,4	1,95	30	35	2693
300_05	40%	35,4	1,67	33	37	4672,4
400_02	5%	25,6	0,55	25	26	377,8
400_02	10%	24,2	1,30	23	26	417,6
400_02	15%	21,2	1,30	20	23	513
400_02	20%	19,4	0,89	18	20	795,4
400_02	25%	18,4	0,55	18	19	1609,4
400_02	30%	18	1,41	17	20	2980,2
400_02	35%	17,8	0,45	17	18	4196
400_02	40%	19	0,71	18	20	5203,6

Tableau 3.8 : Méthode *RLE* avec la stratégie 2 et pourcentage de positions libres statique

Une série de 10 essais a été réalisée sur les problèmes 200_05, 300_05 et 400_02 en utilisant la méthode *RLE* avec la stratégie 2 pour un pourcentage de positions libres déterminé dynamique. Les mêmes solutions de départ ont été utilisées pour les problèmes 300_05 et 400_02. La solution de départ du problème 200_05 a aussi été générée en

prenant la meilleure solution du premier cycle d'une exécution de l'OCF et compte 11 conflits. Les résultats sont présentés dans le Tableau 3.9.

Pour le problème 200_05, on atteint, à tous les essais, 6 conflits, ce qui représente la meilleure solution connue jusqu'à maintenant. Dans le cas des deux autres problèmes, le nombre de conflits moyen est suffisamment près de celui obtenu avec le meilleur pourcentage statique pour qu'on puisse adopter le pourcentage dynamique dans les prochains essais et ainsi réduire le nombre d'essais numériques.

Problème	Pourcentage de positions libres	Nombre de conflits				Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum	
200_05	Dynamique	6	0	6	6	3366,75
300_05	Dynamique	33	0,94	32	35	3465,19
400_02	Dynamique	18	0,66	17	19	3646,12

Tableau 3.9 : Méthode *RLE* avec la stratégie 2 et pourcentage de positions libres dynamique

Il faut maintenant établir lesquelles des neuf stratégies sont les plus efficaces. Ces stratégies ont alors été testées à cinq reprises sur les problèmes 200_05, 300_05 et 400_02 en utilisant la méthode *RLE* avec le pourcentage de positions libres dynamique et en se servant des mêmes solutions de départ que celles utilisées dans les essais numériques précédents. Les résultats sont présentés au Tableau 3.10. Le pourcentage statique de positions libres n'est pas considéré car il nécessite une trop grande quantité d'essais à réaliser.

En ce qui concerne le problème 200_05, seules les stratégies 6, 7 et 8 n'arrivent pas à atteindre la meilleure solution connue, contrairement aux autres stratégies qui l'atteignent

à chaque essai. Le problème 300_05 offre des résultats très différents. Dans le premier groupe de stratégies, la S1 offre la meilleure moyenne de conflits, suivie de près par la S2. Quant à la S3, elle se situe loin derrière avec une moyenne de plus de 10 conflits par rapport à la S2. Les stratégies 4 et 5 offrent environ les mêmes performances, la première étant quelque peu favorisée. Comme pour le problème précédent, les stratégies 6, 7 et 8 sont à la queue du peloton. La S8 a réussi à se démarquer des deux précédentes, mais c'est la S9 qui offre les meilleurs résultats pour le troisième groupe de stratégies. Pour le problème 400_02, les stratégies 1, 2 et 3 donnent des résultats quasi-identiques. Toutefois, c'est la S2 qui a trouvé la meilleure solution pour ce problème (16 conflits), toutes stratégies confondues. La meilleure stratégie du deuxième groupe est la S5, avec une différence de 3 conflits par rapport à la moyenne de la S4. Encore une fois, les stratégies 6, 7 et 8 donnent le même genre de résultats qu'auparavant, c'est-à-dire que la même solution est trouvée à chaque essai dans des temps d'exécution relativement bas par rapport aux autres stratégies. La S8 fait un peu mieux que les précédentes, mais elle est loin derrière la S9, qui est, encore une fois, la meilleure stratégie du troisième groupe.

Problème	Stratégie	Nombre de conflits				Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum	
200_05	1	6	0,00	6	6	3309,6
	2	6	0,00	6	6	3360,8
	3	6	0,00	6	6	3379,8
	4	6	0,00	6	6	4835,6
	5	6	0,00	6	6	3105,4
	6	8	0,00	8	8	1309,6
	7	8	0,00	8	8	1602
	8	8	0,00	8	8	2795,6
	9	6	0,00	6	6	2376
300_05	1	33,8	1,30	32	35	3420,8
	2	34,4	0,89	34	36	3468,8
	3	45,2	2,17	42	48	3498
	4	35,2	2,39	32	38	3813,4
	5	35,8	1,92	34	39	3234
	6	52	0,00	52	52	2968,8
	7	52	0,00	52	52	2955,4
	8	44,6	0,89	44	46	3009,6
	9	37,6	1,14	36	39	3035,8
400_02	1	17,6	0,55	17	18	3676,8
	2	17,6	1,14	16	19	3784,4
	3	17,8	0,84	17	19	3866,8
	4	21,2	0,84	20	22	3850,8
	5	18,2	0,45	18	19	3540,6
	6	29	0,00	29	29	970
	7	29	0,00	29	29	964,4
	8	25	1,73	24	28	2365,4
	9	19	1,22	18	21	3529,6

Tableau 3.10 : Méthode *RLE* avec les stratégies 1 à 9 et pourcentage de positions libres dynamique

À la lumière des résultats précédents, on retient une seule stratégie par groupe de stratégies, pour un total de trois. Dans le premier groupe, c'est la S2 qui est conservée. Il est vrai que la S1 a mieux performé que la S2 au problème 300_05 du Tableau 3.10. Par

contre, la S2 assure d'avoir toujours au moins une position en conflit parmi les positions libres et élimine le risque de gaspiller un cycle en tentant d'améliorer un sous-problème où il n'y a pas de position en conflit parmi les positions libres, ce qui risque d'arriver avec la S1. La S3 est rejetée pour sa très mauvaise performance sur le problème 300_05.

Dans le deuxième groupe de stratégies, la S4 a offert une moyenne à peine meilleure que la S5 pour le problème 300_05. Tout le contraire a été observé sur le problème 400_02, où la S5 dominait la S4 avec une moyenne de 3 conflits en moins. De plus, les temps d'exécution de la S5 sont nettement inférieurs à ceux de la S4, ce qui laisse croire que les sous-problèmes générés par la première sont plus faciles à résoudre que ceux suggérés par la seconde. Ceci représente un net avantage, donc la S5 est conservée.

Dans le troisième groupe, on élimine immédiatement les stratégies 6 et 7. Le problème avec ces deux stratégies c'est qu'elles n'intègrent pas d'éléments aléatoires. Pour ces deux stratégies, il n'existe qu'un seul sous-problème pouvant être généré à partir d'une solution donnée en entrée. Éventuellement, elles vont générer un sous-problème que le *PLNE* ne sera pas en mesure d'améliorer. Ce sous-problème sera soit résolu facilement par le *PLNE* et donnera des temps d'exécution bas (tel qu'observé dans le Tableau 3.10), soit le *PLNE* ne sera jamais capable de résoudre le sous-problème dans les 5 secondes allouées et donnera un temps d'exécution avoisinant les 5000 secondes (1000 cycles multipliés par 5 secondes). La S8 souffre moins de ce problème, car elle comprend un élément aléatoire lorsque le nombre de positions en conflit est supérieur au nombre de positions libres. Autrement, elle se comporte comme la S6 et la S7. C'est donc la S9 qui gagne la compétition pour le troisième groupe de stratégies, non seulement par défaut, mais aussi

parce qu'elle a obtenu les meilleurs résultats de ce groupe. Finalement, ce sont les stratégies 2, 5 et 9 qu'on utilisera pour l'hybridation entre l'*OCF* et le *PLNE*.

3.3.2.5 Performance du RLE en comparaison avec les résultats de l'OCF

Suite aux essais numériques, comparons les résultats obtenus par l'*OCF* proposée par Gravel, Gagné *et al.* [42] (Tableau 3.5) et la méthode *RLE* avec les stratégies 2, 5 et 9. Pour le problème 200_05, l'*OCF* obtient une moyenne de 8,90 conflits comparativement à une moyenne de 6 conflits pour la méthode *RLE* avec chacune des 3 stratégies. Quant au problème 300_05, l'*OCF* donne une moyenne de 40,55 conflits, alors qu'il est possible d'obtenir 34,4 conflits avec la *RLE* avec la stratégie 2. Finalement, l'*OCF* trouve une moyenne de 23,28 conflits pour le problème 400_02, alors que la *RLE* avec la stratégie 2 obtient une moyenne de 17,6 conflits. Il est à noter que pour les deux derniers problèmes, les stratégies 5 et 9 offrent aussi de meilleurs résultats que l'*OCF*, mais sont de qualité inférieure à ceux donnés par la stratégie 2.

On peut aussi comparer les résultats obtenus par la méthode *RLE* avec les stratégies 2, 5 et 9 (Tableau 3.10) avec les résultats trouvés par l'*OCF* jumelée à la recherche locale (Tableau 3.6) proposée par Gravel, Gagné *et al.* [42]. Dans le cas du problème 200_05, la méthode *RLE* trouve à chaque essai la meilleure solution connue de 6 conflits, ce qui n'est pas le cas pour l'*OCF* avec recherche locale qui obtient une moyenne de 6,4 conflits. Par contre, c'est l'*OCF* avec recherche locale qui obtient la meilleure moyenne pour le problème 300_05, avec 32,28 conflits. C'est avec la stratégie 2 que la méthode *RLE* obtient, pour sa part, la meilleure moyenne pour ce problème avec 34,4 conflits.

Finalement, pour le problème 400_02, les deux approches obtiennent sensiblement les mêmes résultats avec des moyennes de 17,37 conflits pour l'*OCF* avec recherche locale et de 17,6 conflits avec la *RLE* et la stratégie 2.

À la lumière de ces résultats, on peut affirmer que la méthode *RLE* performe mieux que l'*OCF* sans recherche locale. Par contre, l'ajout d'une phase de recherche locale à l'*OCF* ne permet pas de tirer de véritables conclusions sur la performance des deux méthodes. On peut alors se demander si on ne pourrait pas tirer avantage des deux méthodes par une hybridation de l'*OCF* avec la méthode *RLE*. La section suivante explore différentes formes d'hybridations entre les deux méthodes.

3.4 Hybridations entre l'OCF et la PLNE

Dans cette section, trois différentes formes d'hybridation entre l'*OCF* proposée par Gravel, Gagné *et al.* [42] et la résolution de sous-problèmes avec la programmation linéaire en nombre entiers sont présentées. Premièrement, une hybridation intégrative est proposée, où un sous-problème est résolu par le *PLNE* à chaque cycle de l'*OCF*. Ensuite, une autre hybridation intégrative est définie en espaçant les appels à la méthode *RLE* et en intensifiant cette forme de recherche. Finalement, nous mettons à l'essai une hybridation à relais, où l'*OCF* fournit la meilleure solution obtenue par son processus à la méthode *RLE* qui démarre un processus itératif de recherche tel que présenté à la Section 2.3.2.3.

3.4.1 Hybridation intégrative à chaque cycle de l'OCF

Cette forme d'hybridation consiste faire un appel au *PLNE* à chaque cycle de l'*OCF*. Autrement dit, on remplace la phase de recherche locale dans l'OCF par le *PLNE*.

Pour alléger le texte, nous désignerons cette hybridation *HICC*. Deux options s'offrent à nous : on peut donner au *PLNE*, comme solution d'entrée, la meilleure solution du cycle (*BC*) ou la meilleure solution globale (*BG*). Examinons tout d'abord le fonctionnement de cette hybridation avant de présenter les expérimentations numériques.

Avant d'expliquer le fonctionnement du *HICC*, voici un bref rappel du fonctionnement de la méthode *RLE*. La méthode *RLE* effectue une série d'appels au *PLNE* pour améliorer la qualité d'une solution. Le sous-problème donné au $i^{ème}$ appel du *PLNE* est formé à partir de la solution trouvée au $(i\text{-}1)^{ième}$ appel du *PLNE*. Dans le cas du *HICC*, le sous-problème donné au $i^{ème}$ appel du *PLNE* est créé à partir de la solution du $i^{ème}$ cycle de l'*OCF*.

La Figure 2.6 présente, sous forme de pseudo-code, le fonctionnement de l'hybridation. L'algorithme est semblable à celui de la Figure 2.3, à l'exception des phases de recherche locale qui sont remplacées. On a *sp*, qui représente le sous-problème créé à partir de la meilleure solution du cycle (*HCCC-BC*) ou de la meilleure solution globale (*HCCC-BG*), S^{sp}, qui est la solution trouvée par le *PLNE* après avoir résolu le sous-problème *sp*, et F_{sp}, qui représente le nombre de conflits de cette solution. À la fin de chaque cycle de l'*OCF*, on construit le sous-problème *sp* et on le résout avant de mettre à jour la trace de phéromone avec la solution S^{sp}.

```
t = 0
Initialiser la matrice de phéromone τ(t) à τ0 pour chaque paire de classes uv
Pour t = 1 à tMax
        Choisir une classe aléatoirement pour la 1re position de la séquence des fourmis
        Pour pos = 2 à nc
                Pour m = 1 à nb_fourmis
                        Choisir une classe de voitures v avec l'Équation 2.3
                        Mise à jour locale de la trace de phéromone entre les classes u et v
        Évaluer la solution S des nb_fourmis fourmis
        Déterminer la meilleure solution du cycle (S^bc) et sa valeur F+
        Construire un sous-problème (sp) à partir de S^bc (ou S^bg)
        Résoudre sp avec le PLNE pour trouver S^sp et F_sp
        Mise à jour globale de la trace de phéromone avec S^sp
        Mise à jour de S^bg, la meilleure solution trouvée à ce jour et de sa valeur F*
```

Figure 3.6 : Pseudo-code du *HICC*

De plus, on introduira la « stratégie aléatoire » (SA), où on choisit aléatoirement une stratégie parmi les stratégies 2, 5 et 9 à chaque appel du *PLNE*. La progression dynamique du pourcentage de positions libres se fait de la même façon que pour la méthode *RLE*. Toutefois, on réajuste le pourcentage à tous les 200 cycles, plutôt qu'aux 100 cycles, pour la stratégie aléatoire, étant donné que trois stratégies entrent en jeu au lieu d'une seule et que le pourcentage de positions libres de chaque stratégie évolue de façon indépendante. De cette façon, le pourcentage de positions libres des stratégies est ajusté à cinq reprises pendant l'exécution de l'algorithme.

Le Tableau 3.11 présente les résultats obtenus par l'hybridation *HICC-BC* sur les problèmes 200_05, 300_05 et 400_02. Les moyennes sont calculées sur cinq essais, et on accorde 5 secondes au *PLNE* pour résoudre un sous-problème. Pour le premier problème, les quatre stratégies arrivent à trouver la meilleure solution connue. Seule la stratégie 5

n'est pas arrivée à trouver la meilleure solution à tous les coups. Pour le problème 300_05, la stratégie 9 obtient la meilleure moyenne avec 40,2 conflits, suivie de près par la « stratégie aléatoire » avec 40,6 conflits. Il faut remarquer que ces deux stratégies trouvent le même minimum et maximum. Enfin, c'est la stratégie 9 qui semble la moins bien efficace pour le problème 400_02, avec une moyenne de 23,4 conflits. Le *HICC-BC* apporte une petite contribution par rapport à l'*OCF* sans recherche locale. Par contre, cette hybridation n'arrive pas à concurrencer l'*OCF* avec recherche locale, sauf pour le problème 200_05.

Problème	**Méthode**	**Nombre de conflits**				**Cycle moyen**	**Temps d'exécution (secondes)**
		Moyenne	**Écart-type**	**Minimum**	**Maximum**		
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF + RL	6,40	0,51	6	8	986,70	32,01
	HICC + S2	6	0	6	6	315,2	3455,4
	HICC + S5	6,6	0,55	6	7	487	3236,6
	HICC + S9	6	0	6	6	105	3250,8
	HICC + SA	6	0	6	6	254,6	3155,8
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF + RL	32,28	1,10	29	35	1000	66,09
	HICC + S2	41,8	0,45	41	42	150,8	4442,2
	HICC + S5	41	0,00	41	41	385,2	4203,6
	HICC + S9	40,2	0,45	40	41	494,6	4427,4
	HICC + SA	40,6	0,55	40	41	206,8	4181,2
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF + RL	17,37	0,90	16	20	1000	96,70
	HICC + S2	22,8	0,84	22	24	313	4454,4
	HICC + S5	22,8	0,84	22	24	294,4	4461,8
	HICC + S9	23,4	0,55	23	24	293,8	4566,4
	HICC + SA	22,6	0,55	22	23	442,6	4448,6

Tableau 3.11 : Hybridation *HICC-BC* sur des problèmes de l'*ET3*

On retrouve, au Tableau 3.12, les résultats obtenus par l'hybridation *HICC-BG*. Les essais numériques ont été effectués sous les mêmes conditions que pour l'hybridation *HICC-BC*. Toutes les stratégies, à l'exception de la stratégie 5, arrivent à trouver la meilleure solution connue pour le problème 200_05. Pour le problème 300_05, c'est la stratégie aléatoire qui donne les meilleurs résultats, avec une moyenne de 35,8 conflits. Finalement, la stratégie qui a le mieux performé pour le problème 400_02 est la stratégie 5, avec une moyenne de 19 conflits. De plus, cette stratégie trouve une solution à 17 conflits et cette solution se démarque grandement par rapport aux solutions trouvées par les autres stratégies pour ce problème. Cette hybridation apporte une contribution plus significative sur l'*OCF* sans recherche locale que le *HICC-BC*. Par exemple, pour le problème 300_05 (respectivement 400_02), la moyenne de conflits a été améliorée d'environ 5 points (respectivement 4 points). Mais tout comme le *HICC-BC*, le *HICC-BG* n'arrive pas à faire mieux que l'*OCF* avec recherche locale.

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF + RL	6,40	0,51	6	8	986,70	32,01
	HICC + S2	6	0	6	6	18,6	3306,2
	HICC + S5	6	0	6	6	57	3076,4
	HICC + S9	6,2	0,45	6	7	30	2713,4
	HICC + SA	6	0	6	6	47,2	2919,2
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF + RL	32,28	1,10	29	35	1000	66,09
	HICC + S2	37,8	1,92	35	40	420,8	3357,2
	HICC + S5	38	1,41	36	39	382,6	3244,6
	HICC + S9	39,4	2,88	35	42	495,6	3087,4
	HICC + SA	35,8	1,30	34	37	447,4	3122,6
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF + RL	17,37	0,90	16	20	1000	96,70
	HICC + S2	20,6	1,52	19	23	356,8	3758,6
	HICC + S5	19	1,41	17	20	604,4	3589,4
	HICC + S9	20,4	1,14	19	22	342,8	3810,8
	HICC + SA	20,4	0,55	20	21	446,6	3612,8

Tableau 3.12 : Hybridation *HICC-BG* sur des problèmes de l'*ET3*

Les Tableaux 3.13 et 3.14 rapportent les résultats obtenus par l'*HICC-BG* avec la stratégie aléatoire (*SA*) sur les problèmes de l'*ET2* et l'*ET3*, respectivement. Cette méthode atteint une moyenne de conflits de qualité supérieure ou égale à l'*OCF* avec recherche locale 8 fois sur 9 pour les problèmes de l'*ET2* et 13 fois sur 30 pour ceux de l'*ET3*. Le temps nécessaire pour obtenir ces résultats de qualité douteuse appuie le fait que la méthode *HICC-BG* ne rivalise pas avec l'*OCF* avec recherche locale.

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
16_81	*OCF*	0,1	0,33	0	1	329,10	4,23
	OCF+RL	0,0	0,00	0	0	149,03	1,75
	HICC-BG	0,2	0,45	0	1	147,6	91,86
26_82	*OCF*	0,0	0,00	0	0	83,72	0,74
	OCF+RL	0,0	0,00	0	0	28,33	0,26
	HICC-BG	0,0	0,00	0	0	18,2	57,8
41_66	*OCF*	0,0	0,00	0	0	5,84	0,04
	OCF+RL	0,0	0,00	0	0	2,98	0,03
	HICC-BG	0,0	0,00	0	0	2,4	2,6
4_72	*OCF*	0,0	0,00	0	0	58,72	0,50
	OCF+RL	0,0	0,00	0	0	30,54	0,27
	HICC-BG	0,0	0,00	0	0	114,4	279,6
10_93	*OCF*	4,2	0,47	3	5	298,93	10,10
	OCF+RL	3,8	0,45	3	5	507,22	13,79
	HICC-BG	3,8	0,45	3	4	499,6	2199,6
19_71	*OCF*	2,1	0,27	2	3	309,48	9,26
	OCF+RL	2,0	0,00	2	2	125,61	13,04
	HICC-BG	2,0	0,00	2	2	69,8	2815,6
6_76	*OCF*	6,0	0,00	6	6	1,01	7,94
	OCF+RL	6,0	0,00	6	6	1,00	11,76
	HICC-BG	6,0	0,00	6	6	1,0	4252,4
36_92	*OCF*	2,3	0,46	2	3	346,02	8,50
	OCF+RL	2,0	0,00	2	2	145,27	12,66
	HICC-BG	2,0	0,00	2	2	54,6	2460,4
21_90	*OCF*	2,6	0,49	2	3	324,39	8,73
	OCF+RL	2,0	0,00	2	2	146,81	12,60
	HICC-BG	2,0	0,00	2	2	37,8	2713,0

Tableau 3.13 : Hybridation *HICC-BG* sur les problèmes de l'*ET2*

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
200_01	*OCF*	3,8	0,62	2	5	293,44	19,46
	OCF+RL	1,0	0,67	0	3	985,79	33,42
	HICC-BG	0,4	0,55	0	1	193,6	1588,4
200_02	*OCF*	4,14	0,51	3	5	422,62	19,42
	OCF+RL	2,41	0,49	2	3	962,63	33,44
	HICC-BG	3,4	0,89	2	4	46,4	2968,8
200_03	*OCF*	8,90	0,59	8	10	350,44	19,70
	OCF+RL	6,04	0,82	4	8	998,58	34,45
	HICC-BG	8,4	1,14	7	10	388,2	3319,4
200_04	*OCF*	9,86	0,51	8	11	274,59	20,49
	OCF+RL	7,57	0,56	7	9	968,62	34,66
	HICC-BG	7,4	0,55	7	8	543,4	3124,0
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF+RL	6,40	0,51	6	8	986,70	32,01
	HICC-BG	6	0	6	6	47,2	2919,2
200_06	*OCF*	6,87	0,39	6	8	232,09	19,03
	OCF+RL	6,0	0,00	6	6	314,36	32,52
	HICC-BG	6,0	0,00	6	6	38,8	3330,8
200_07	*OCF*	2,99	0,36	2	4	344,48	18,95
	OCF+RL	0,0	0,00	0	0	986,47	28,65
	HICC-BG	0,0	0,00	0	0	407,0	1261,2
200_08	*OCF*	8,0	0,00	8	8	51,06	17,31
	OCF+RL	8,0	0,00	8	8	45,98	31,10
	HICC-BG	8,0	0,00	8	8	39,8	3622,8
200_09	*OCF*	11,85	0,48	11	13	284,31	19,93
	OCF+RL	10,0	0,00	10	10	988,06	34,00
	HICC-BG	10,0	0,00	10	10	243,2	3220,0
200_10	*OCF*	21,44	0,57	20	23	313,96	18,65
	OCF+RL	19,09	0,29	19	20	998,96	32,68
	HICC-BG	19,8	0,45	19	20	454,4	3212,2
300_01	*OCF*	5,33	0,71	4	7	386,80	30,23
	OCF+RL	2,15	0,77	0	4	990,38	65,29
	HICC-BG	3,0	1,00	2	4	255,6	3405,4
300_02	*OCF*	13,15	0,39	13	15	307,19	30,68
	OCF+RL	12,02	0,14	12	13	749,79	66,38
	HICC-BG	12,0	0,00	12	12	147,6	3291,4
300_03	*OCF*	14,54	0,54	13	16	370,63	32,26
	OCF+RL	13,06	0,24	13	14	858,31	67,32

	HICC-BG	14,0	1,00	13	15	218,0	3520,4
300_04	*OCF*	10,33	0,68	9	12	390,43	30,25
	OCF+RL	8,16	0,81	7	10	966,18	65,99
	HICC-BG	8,8	0,84	8	10	347,4	3403,2
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF+RL	32,28	1,10	29	35	1000	66,09
	HICC-BG	35,8	1,30	34	37	447,4	3122,6
300_06	*OCF*	7,59	0,74	6	9	255,48	30,21
	OCF+RL	4,38	0,99	2	7	983,85	65,92
	HICC-BG	5,4	0,55	5	6	262,0	3459,8
300_07	*OCF*	2,89	0,63	1	4	173,35	30,07
	OCF+RL	0,59	0,64	0	2	973,01	62,64
	HICC-BG	0,6	0,55	0	1	351,8	2508,8
300_08	*OCF*	9,17	0,38	9	10	229,22	29,00
	OCF+RL	8,0	0,00	8	8	869,05	63,21
	HICC-BG	8,2	0,45	8	9	108,0	3744,6
300_09	*OCF*	9,05	0,56	8	10	274,34	28,76
	OCF+RL	7,46	0,67	7	9	885,19	63,47
	HICC-BG	8,4	0,89	7	9	307,6	3427,0
300_10	*OCF*	34,63	1,04	32	37	237,32	28,57
	OCF+RL	22,6	0,85	21	25	1000	63,91
	HICC-BG	26,8	1,92	24	29	686,6	3228,8
400_01	*OCF*	3,01	0,56	2	4	201,01	41,77
	OCF+RL	2,52	0,63	1	4	695,43	96,84
	HICC-BG	1,6	0,89	1	3	481,4	3579,6
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF+RL	17,37	0,90	16	20	1000	96,70
	HICC-BG	20,4	0,55	20	21	446,6	3612,8
400_03	*OCF*	11,65	0,5	10	12	212,39	38,40
	OCF+RL	9,91	0,43	9	11	954,46	94,87
	HICC-BG	10,4	0,55	10	11	225,2	3976,6
400_04	*OCF*	21,96	0,75	20	24	454,46	43,57
	OCF+RL	19,01	0,1	19	20	994,68	98,98
	HICC-BG	19,4	0,55	19	20	190,4	3977,4
400_05	*OCF*	3,48	0,96	1	6	346,67	36,47
	OCF+RL	0,01	0,10	0	1	979,21	76,85
	HICC-BG	0,6	0,89	0	2	459,6	2827,6
400_06	*OCF*	4,20	0,97	1	6	263,12	39,85
	OCF+RL	0,33	0,53	0	2	997,54	87,56
	HICC-BG	0,8	0,45	0	1	232,8	2827,2
400_07	*OCF*	7,65	0,83	5	9	260,60	36,93
	OCF+RL	5,44	0,80	4	7	989,49	93,29

	HICC-BG	4,8	0,84	4	6	391,4	3598,2
400_08	*OCF*	11,54	1,62	6	14	179,93	35,51
	OCF+RL	5,30	0,81	4	7	1000	93,39
	HICC-BG	4,8	0,84	4	6	534,0	3468,2
400_09	*OCF*	17,98	1,10	15	20	209,19	41,38
	OCF+RL	7,63	0,95	5	10	1000	97,75
	HICC-BG	8,8	1,79	6	11	413,2	3564,8
400_10	*OCF*	4,24	0,99	2	7	77,19	40,94
	OCF+RL	0,95	0,72	0	3	991,09	93,89
	HICC-BG	0,2	0,45	0	1	33,2	765,0

Tableau 3.14 : Hybridation *HICC-BG* sur les problèmes de l'*ET3*

Si on compare les résultats du *HICC,* sans distinction entre le *BC* et le *BG*, avec l'*OCF* sans recherche locale pour les problèmes étudiés, on remarque que l'hybridation apporte une contribution sur la moyenne de conflits et stabilise l'écart entre la meilleure et la pire solution trouvée. Mais entre les deux hybridations, c'est le *HICC-BG* qui est la plus efficace. Malgré tout, le *HICC* ne peut rivaliser l'*OCF* avec recherche locale.

La piètre performance du *HICC-BC* par rapport au *HICC-BG* s'explique par le fait que la meilleure solution du cycle change généralement à chaque cycle, tout en ayant une qualité variable. En plus, aucune intensification n'est effectuée sur cette solution, car on ne fait qu'un seul appel au *PLNE*. Par contre, avec le *HICC-BG*, le *PLNE* est appelé avec la même solution tant que la meilleure solution globale n'est pas modifiée. De ce fait, une intensification est ainsi réalisée autour d'une même solution. Avec le *HICC-BC*, il aurait été possible d'effectuer plusieurs appels du *PLNE* à chaque cycle de l'*OCF*, mais le temps d'exécution en aurait été grandement augmenté. Une autre forme d'hybridation pourrait faire des appels consécutifs du *PLNE*, mais seulement à certains cycles de l'*OCF*

lorsqu'une situation particulière se présente. Cette forme d'hybridation est étudiée à la section suivante.

3.4.2 Hybridation intégrative à fréquence espacée

L'hybridation intégrative à fréquence espacée (*HIFE*) consiste à faire une intensification sur la meilleure solution globale lorsque certains critères sont réunis. L'intensification se fait en effectuant une série d'appels au *PLNE*. On effectue une intensification lorsque l'*OCF* améliore la meilleure solution globale ou lorsque cette dernière n'a pas été améliorée depuis 100 cycles. Pour conserver le même nombre d'appels au *PLNE* que dans les essais précédents, on réalise une intensification après le dernier cycle de l'*OCF*. Aucune intensification n'est effectuée avant le 101^{e} cycle, pour laisser le temps à l'*OCF* de trouver une meilleure solution globale de qualité acceptable. Le nombre d'appels successifs du *PLNE* correspond au nombre de cycles écoulés depuis la dernière intensification.

La Figure 3.7 illustre, sous forme de pseudo-code, le fonctionnement du *HIFE*. La variable *stagnation* contient le nombre de cycles écoulés depuis la dernière fois où la meilleure solution globale a été améliorée ou le nombre de cycles depuis la dernière intensification.

```
t = 0
stagnation = 0
Initialiser la matrice de phéromone τ(t) à τ0 pour chaque paire de classes uv
Pour t = 1 à tMax
        Choisir une classe aléatoirement pour la 1e position de la séquence des fourmis
        Pour pos = 2 à nc
                Pour m = 1 à nb_fourmis
                        Choisir une classe de voitures v avec l'Équation 2.3
                        Mise à jour locale de la trace de phéromone entre les classes u et v
        Évaluer la solution (S) des nb_fourmis fourmis
        Déterminer la meilleure solution du cycle (S^bc) et sa valeur F+
        Si F+ < F*
                Mise à jour de S^bg et F*
                Si t ≥ 101
                        Effectuer une intensification de stagnation itérations sur S^bg
                        Mise à jour de S^bg et F* suite à l'intensification
                Mise à jour globale de la trace de phéromone avec S^bg
                stagnation = 0
        Sinon Si stagnation = 100 et t ≥ 101
                        Effectuer une intensification de stagnation itérations sur S^bg
                        Mise à jour de S^bg et F* suite à l'intensification
                        Mise à jour globale de la trace de phéromone avec S^bg
                        stagnation = 0
                Sinon
                        stagnation = stagnation + 1
                        Mise à jour globale de la trace de phéromone avec S^bc
Effectuer une intensification de stagnation itérations sur S^bg
Mise à jour de S^bg et F* suite à l'intensification
```

Figure 3.7 : Pseudo-code du *HIFE*

Pour mettre à l'essai cette forme d'hybridation, nous utilisons uniquement la « stratégie aléatoire » (SA) pour la création des sous-problèmes, car celle-ci s'est avérée relativement performante dans l'hybridation *HICC-BG*. Un total de cinq essais est effectué sur chaque problème, en accordant 5 secondes à chaque appel du *PLNE*.

On retrouve, au Tableau 3.15, les résultats du *HIFE* sur les problèmes 200_05, 300_05 et 400_02. Cette série de tests est effectuée pour voir le potentiel de cette hybridation avant de résoudre l'ensemble des problèmes de l'*ET2* et de l'*ET3*. On remarque immédiatement que cette hybridation donne des résultats semblables à ceux donnés par l'*OCF* avec recherche locale. De plus, le nombre de cycles moyens pour y arriver est nettement inférieur. Il est donc envisageable de tester cette hybridation sur l'ensemble des problèmes de l'*ET2* et de l'*ET3*.

Problème	**Méthode**	**Nombre de conflits**				**Cycle moyen**	**Temps d'exécution (secondes)**
		Moyenne	**Écart-type**	**Minimum**	**Maximum**		
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF + RL	6,40	0,51	6	8	986,70	32,01
	HIFE + SA	6	0	6	6	130,2	2785,2
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF + RL	32,28	1,10	29	35	1000	66,09
	HIFE + SA	33	1,22	32	35	390,6	3151,8
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF + RL	17,37	0,90	16	20	1000	96,70
	HIFE + SA	17,4	0,9	16	18	353,8	3531,4

Tableau 3.15 : Hybridation *HIFE* sur des problèmes de l'*ET3*

Les Tableaux 3.16 et 3.17 présentent les résultats obtenus par l'hybridation *HIFE* sur tous les problèmes de l'*ET2* et de l'*ET3*, respectivement. En comparant les performances de l'*OCF* sans recherche locale au *HIFE*, on note immédiatement que la moyenne du nombre de conflits a été améliorée pour tous les problèmes. Cette forme d'hybridation apporte bel et bien une contribution à la performance de l'*OCF*. De plus, l'hybridation *HIFE* surclasse ou atteint au moins le même résultat que l'*OCF* avec recherche locale pour tous les problèmes de l'*ET2* et pour 21 des 30 problèmes de l'*ET3*.

Seuls les problèmes 200_02, 200_10, 300_02 à 300_05, 300_09, 300_10 et 400_02 ont résisté au *HIFE*. La moyenne du cycle moyen où le *HIFE* trouve la meilleure solution pour l'*ET3* est de 270,35, comparativement à 903,79 pour l'*OCF* avec recherche locale. Donc, en plus d'obtenir une meilleure moyenne pour environ deux problèmes sur trois, le *HIFE* trouve plus rapidement, en terme de cycles, la meilleure solution que l'*OCF* avec recherche locale. Ceci ne s'applique pas pour les problèmes de l'*ET2*, car la moyenne des cycles moyens des deux méthodes est à peu près la même. Toutefois, le temps d'exécution nécessaire est nettement supérieur à celui de l'*OCF* avec recherche locale pour les problèmes où une solution sans conflit n'existe pas. Le *HIFE* se présente donc comme une alternative intéressante à la recherche locale, à la condition que le temps ne soit pas une contrainte.

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
16_81	*OCF*	0,1	0,33	0	1	329,10	4,23
	OCF+RL	0,0	0,00	0	0	149,03	1,75
	HIFE	0,0	0,00	0	0	131,6	37,4
26_82	*OCF*	0,0	0,00	0	0	83,72	0,74
	OCF+RL	0,0	0,00	0	0	28,33	0,26
	HIFE	0,0	0,00	0	0	25,0	0,4
41_66	*OCF*	0,0	0,00	0	0	5,84	0,04
	OCF+RL	0,0	0,00	0	0	2,98	0,03
	HIFE	0,0	0,00	0	0	2,4	0,2
4_72	*OCF*	0,0	0,00	0	0	58,72	0,50
	OCF+RL	0,0	0,00	0	0	30,54	0,27
	HIFE	0,0	0,00	0	0	78,4	3,8
10_93	*OCF*	4,2	0,47	3	5	298,93	10,10
	OCF+RL	3,8	0,45	3	5	507,22	13,79
	HIFE	3,4	0,55	3	4	389,8	1757,6
19_71	*OCF*	2,1	0,27	2	3	309,48	9,26
	OCF+RL	2,0	0,00	2	2	125,61	13,04
	HIFE	2,0	0,00	2	2	111,2	2796,4
6_76	*OCF*	6,0	0,00	6	6	1,01	7,94
	OCF+RL	6,0	0,00	6	6	1,00	11,76
	HIFE	6,0	0,0	6	6	1,00	4180,2
36_92	*OCF*	2,3	0,46	2	3	346,02	8,50
	OCF+RL	2,0	0,00	2	2	145,27	12,66
	HIFE	2,0	0,0	2	2	108	2589,4
21_90	*OCF*	2,6	0,49	2	3	324,39	8,73
	OCF+RL	2,0	0,00	2	2	146,81	12,60
	HIFE	2,0	0,0	2	2	118,4	2758,6

Tableau 3.16 : Hybridation *HIFE* sur les problèmes de l'*ET2*

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
200_01	*OCF*	3,8	0,62	2	5	293,44	19,46
	OCF+RL	1,0	0,67	0	3	985,79	33,42
	HIFE	0,0	0	0	0	327,8	788,8
200_02	*OCF*	4,14	0,51	3	5	422,62	19,42
	OCF+RL	2,41	0,49	2	3	962,63	33,44
	HIFE	2,8	0,45	2	3	232,4	2942,2
200_03	*OCF*	8,90	0,59	8	10	350,44	19,70
	OCF+RL	6,04	0,82	4	8	998,58	34,45
	HIFE	6,0	1	5	7	497	3086,2
200_04	*OCF*	9,86	0,51	8	11	274,59	20,49
	OCF+RL	7,57	0,56	7	9	968,62	34,66
	HIFE	7,0	0	7	7	166	2853,2
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF+RL	6,40	0,51	6	8	986,70	32,01
	HIFE	6,0	0	6	6	130,2	2785,2
200_06	*OCF*	6,87	0,39	6	8	232,09	19,03
	OCF+RL	6,0	0,00	6	6	314,36	32,52
	HIFE	6,0	0	6	6	142,6	3360
200_07	*OCF*	2,99	0,36	2	4	344,48	18,95
	OCF+RL	0,0	0,00	0	0	986,47	28,65
	HIFE	0,0	0	0	0	157,4	199,8
200_08	*OCF*	8,0	0,00	8	8	51,06	17,31
	OCF+RL	8,0	0,00	8	8	45,98	31,10
	HIFE	8,0	0	8	8	93	3609,2
200_09	*OCF*	11,85	0,48	11	13	284,31	19,93
	OCF+RL	10,0	0,00	10	10	988,06	34,00
	HIFE	10,0	0	10	10	124,6	3167,6
200_10	*OCF*	21,44	0,57	20	23	313,96	18,65
	OCF+RL	19,09	0,29	19	20	998,96	32,68
	HIFE	19,6	0,55	19	20	310,4	3205
300_01	*OCF*	5,33	0,71	4	7	386,80	30,23
	OCF+RL	2,15	0,77	0	4	990,38	65,29
	HIFE	1,0	0	1	1	423,0	3125
300_02	*OCF*	13,15	0,39	13	15	307,19	30,68
	OCF+RL	12,02	0,14	12	13	749,79	66,38
	HIFE	12,4	0,55	12	13	139,0	3153
	OCF	14,54	0,54	13	16	370,63	32,26

300_03	*OCF*	14,54	0,54	13	16	370,63	32,26
	OCF+RL	13,06	0,24	13	14	858,31	67,32
	HIFE	13,2	0,45	13	14	170,4	3398,2
300_04	*OCF*	10,33	0,68	9	12	390,43	30,25
	OCF+RL	8,16	0,81	7	10	966,18	65,99
	HIFE	8,2	0,84	7	9	332,2	3459,6
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF+RL	32,28	1,10	29	35	1000	66,09
	HIFE	33,0	1,22	32	35	390,6	3151,8
300_06	*OCF*	7,59	0,74	6	9	255,48	30,21
	OCF+RL	4,38	0,99	2	7	983,85	65,92
	HIFE	3,6	1,34	3	6	404,6	3187,4
300_07	*OCF*	2,89	0,63	1	4	173,35	30,07
	OCF+RL	0,59	0,64	0	2	973,01	62,64
	HIFE	0,0	0,00	0	0	196,0	434,2
300_08	*OCF*	9,17	0,38	9	10	229,22	29,00
	OCF+RL	8,0	0,00	8	8	869,05	63,21
	HIFE	8,0	0,00	8	8	150,0	3497,2
300_09	*OCF*	9,05	0,56	8	10	274,34	28,76
	OCF+RL	7,46	0,67	7	9	885,19	63,47
	HIFE	7,6	0,55	7	8	278,2	3310,4
300_10	*OCF*	34,63	1,04	32	37	237,32	28,57
	OCF+RL	22,6	0,85	21	25	1000	63,91
	HIFE	23,4	1,14	22	25	797,8	3265
400_01	*OCF*	3,01	0,56	2	4	201,01	41,77
	OCF+RL	2,52	0,63	1	4	695,43	96,84
	HIFE	1,6	0,5	1	2	201,8	3383,6
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF+RL	17,37	0,90	16	20	1000	96,70
	HIFE	17,4	0,9	16	18	353,8	3531,4
400_03	*OCF*	11,65	0,5	10	12	212,39	38,40
	OCF+RL	9,91	0,43	9	11	954,46	94,87
	HIFE	9,6	0,5	9	10	396,6	3716,2
400_04	*OCF*	21,96	0,75	20	24	454,46	43,57
	OCF+RL	19,01	0,1	19	20	994,68	98,98
	HIFE	19,0	0,00	19	19	181,4	3790,4
400_05	*OCF*	3,48	0,96	1	6	346,67	36,47
	OCF+RL	0,01	0,10	0	1	979,21	76,85
	HIFE	0,0	0,00	0	0	164,4	149,6
400_06	*OCF*	4,20	0,97	1	6	263,12	39,85
	OCF+RL	0,33	0,53	0	2	997,54	87,56
	HIFE	0,0	0,00	0	0	179,2	416,8

400_07	*OCF*	7,65	0,83	5	9	260,60	36,93
	OCF+RL	5,44	0,80	4	7	989,49	93,29
	HIFE	4,2	0,4	4	5	232,4	3405,4
400_08	*OCF*	11,54	1,62	6	14	179,93	35,51
	OCF+RL	5,30	0,81	4	7	1000	93,39
	HIFE	4,0	0,00	4	4	341,6	3164
400_09	*OCF*	17,98	1,10	15	20	209,19	41,38
	OCF+RL	7,63	0,95	5	10	1000	97,75
	HIFE	7,2	1,1	6	9	471,8	3421
400_10	*OCF*	4,24	0,99	2	7	77,19	40,94
	OCF+RL	0,95	0,72	0	3	991,09	93,89
	HIFE	0,0	0,00	0	0	124,2	98

Tableau 3.17 : Hybridation *HIFE* sur les problèmes de l'*ET3*

3.4.3 Hybridation à relais

L'hybridation à relais consiste, dans un premier temps, à exécuter l'*OCF* sans recherche locale et à utiliser, dans un deuxième temps, de manière itérative, la méthode *RLE* avec la meilleure solution trouvée de l'*OCF*. La Figure 3.8 illustre, sous forme de pseudo-code, le fonctionnement de cette hybridation.

```
t = 0
Initialiser la matrice de phéromone τ(t) à τ0 pour chaque paire de classes uv
Pour t = 1 à tMax
        Choisir une classe aléatoirement pour la 1ière position de la séquence des fourmis
        Pour pos = 2 à nc
                Pour m = 1 à nb_fourmis
                        Choisir une classe de voitures v avec l'Équation 2.3
                        Mise à jour locale de la trace de phéromone entre les classes u et v
        Évaluer la solution (S) des nb_fourmis fourmis
        Déterminer la meilleure solution du cycle (S^bc) et sa valeur F+
        Mise à jour globale de la trace de phéromone avec S^bc
        Mise à jour de S^bg, la meilleure solution trouvée à ce jour et sa valeur F*
Exécuter itérativement la méthode RLE avec S^bg comme solution de départ
Mise à jour de S^bg et F* suite à l'intensification
```

Figure 3.8 : Pseudo-code de l'hybridation à relais

On retrouve les résultats de cette hybridation à relais au Tableau 3.18. On obtient, pour tous les essais effectués sur le problème 200_05, la meilleure solution connue à ce jour. Pour le problème 300_05, c'est la stratégie aléatoire qui trouve la meilleure moyenne, avec 31,6 conflits. Finalement, c'est encore la stratégie aléatoire qui donne la meilleure performance pour le problème 400_02, avec 17,2 conflits.

Problème	**Méthode**	**Nombre de conflits**				**Cycle moyen**	**Temps d'exécution (secondes)**
		Moyenne	**Écart-type**	**Minimum**	**Maximum**		
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF + RL	6,40	0,51	6	8	986,70	32,01
	HIFE + SA	6,0	0	6	6	130,2	2785,2
	OCF + S2	6,0	0,0	6	6	1000	3278,8
	OCF + S5	6,0	0,0	6	6	1000	3243,2
	OCF + S9	6,0	0,0	6	6	1000	2481,2
	OCF + SA	6,0	0,0	6	6	1000	2906
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF + RL	32,28	1,10	29	35	1000	66,09
	HIFE + SA	33	1,22	32	35	390,6	3151,8
	OCF + S2	34,6	1,34	33	36	1000	3688,4
	OCF + S5	33,2	0,84	32	34	1000	3224,8
	OCF + S9	34,8	1,1	34	36	1000	3044,2
	OCF + SA	31,6	0,55	31	32	1000	3057,6
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF + RL	17,37	0,90	16	20	1000	96,70
	HIFE + SA	17,4	0,9	16	18	353,8	3531,4
	OCF + S2	17,8	1,48	16	20	1000	3626,4
	OCF + S5	18,6	1,34	17	20	1000	3615,4
	OCF + S9	18,2	0,84	17	19	1000	3572,8
	OCF + SA	17,2	0,84	16	18	1000	3639,8

Tableau 3.18 : Hybridation à relais sur des problèmes de l'*ET3*

En comparant ces résultats avec ceux obtenus par l'*OCF* avec recherche locale et l'hybridation intégrative à fréquence espacée, on note que l'hybridation à relais permet d'améliorer le nombre de conflits moyens de quelques points. Le temps d'exécution nécessaire est toutefois très élevé par rapport à l'*OCF* avec recherche locale. Les Tableaux 3.19 et 3.20 présentent les résultats obtenus par l'hybridation à relais sur tous les problèmes de l'*ET2* et de l'*ET3*, respectivement. L'hybridation à relais obtient une moyenne de conflits de qualité égale ou supérieure aux autres méthodes testées pour la majorité des problèmes testés, sauf pour les problèmes 200_03, 300_02, 300_04, 300_06, 300_09 et 400_03. Par contre, l'écart entre la meilleure moyenne obtenue et celle obtenue par l'hybridation à relais n'est pas assez significatif pour affirmer que cette dernière éprouve des difficultés sur ces problèmes.

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
16_81	*OCF*	0,1	0,33	0	1	329,10	4,23
	OCF + RL	0,0	0,00	0	0	149,03	1,75
	HIFE + SA	0,0	0,00	0	0	131,6	37,4
	OCF + SA	0,0	0,00	0	0	1000	165,2
26_82	*OCF*	0,0	0,00	0	0	83,72	0,74
	OCF + RL	0,0	0,00	0	0	28,33	0,26
	HIFE + SA	0,0	0,00	0	0	25,0	0,4
	OCF + SA	0,0	0,00	0	0	616	80,8
41_66	*OCF*	0,0	0,00	0	0	5,84	0,04
	OCF + RL	0,0	0,00	0	0	2,98	0,03
	HIFE + SA	0,0	0,00	0	0	2,4	0,2
	OCF + SA	0,0	0,00	0	0	2	0
4_72	*OCF*	0,0	0,00	0	0	58,72	0,50
	OCF + RL	0,0	0,00	0	0	30,54	0,27
	HIFE + SA	0,0	0,00	0	0	78,4	3,8
	OCF + SA	0,0	0,00	0	0	460,4	22,6
10_93	*OCF*	4,2	0,47	3	5	298,93	10,10
	OCF + RL	3,8	0,45	3	5	507,22	13,79
	HIFE + SA	3,4	0,55	3	4	389,8	1757,6
	OCF + SA	3,0	0,00	3	3	1000	2165,6
19_71	*OCF*	2,1	0,27	2	3	309,48	9,26
	OCF + RL	2,0	0,00	2	2	125,61	13,04
	HIFE + SA	2,0	0,00	2	2	111,2	2796,4
	OCF + SA	2,0	0,00	2	2	866	2718,4
6_76	*OCF*	6,0	0,00	6	6	1,01	7,94
	OCF + RL	6,0	0,00	6	6	1,00	11,76
	HIFE + SA	6,0	0,0	6	6	1,00	4180,2
	OCF + SA	6,0	0,00	6	6	1	4178,8
36_92	*OCF*	2,3	0,46	2	3	346,02	8,50
	OCF + RL	2,0	0,00	2	2	145,27	12,66
	HIFE + SA	2,0	0,0	2	2	108	2589,4
	OCF + SA	2,0	0,00	2	2	847,8	2506,8
21_90	*OCF*	2,6	0,49	2	3	324,39	8,73
	OCF + RL	2,0	0,00	2	2	146,81	12,60
	HIFE + SA	2,0	0,0	2	2	118,4	2758,6
	OCF + SA	2,0	0,00	2	2	814,8	2734,2

Tableau 3.19 : Résultats de l'hybridation à relais sur les problèmes de l'*ET2*

Problème	Méthode	Nombre de conflits				Cycle moyen	Temps d'exécution (secondes)
		Moyenne	Écart-type	Minimum	Maximum		
200_01	*OCF*	3,8	0,62	2	5	293,44	19,46
	OCF + RL	1,0	0,67	0	3	985,79	33,42
	HIFE + SA	0,0	0	0	0	327,8	788,8
	OCF + SA	0,0	0,00	0	0	1000	276,4
200_02	*OCF*	4,14	0,51	3	5	422,62	19,42
	OCF + RL	2,41	0,49	2	3	962,63	33,44
	HIFE + SA	2,8	0,45	2	3	232,4	2942,2
	OCF + SA	2,4	0,55	2	3	1000	2930,6
200_03	*OCF*	8,90	0,59	8	10	350,44	19,70
	OCF + RL	6,04	0,82	4	8	998,58	34,45
	HIFE + SA	6,0	1	5	7	497	3086,2
	OCF + SA	6,2	1,10	5	8	1000	3126,6
200_04	*OCF*	9,86	0,51	8	11	274,59	20,49
	OCF + RL	7,57	0,56	7	9	968,62	34,66
	HIFE + SA	7,0	0	7	7	166	2853,2
	OCF + SA	7,0	0,00	7	7	1000	2924,8
200_05	*OCF*	8,81	0,42	8	10	201,67	18,28
	OCF + RL	6,40	0,51	6	8	986,70	32,01
	HIFE + SA	6,0	0	6	6	130,2	2785,2
	OCF + SA	6,0	0,00	6	6	1000	2906
200_06	*OCF*	6,87	0,39	6	8	232,09	19,03
	OCF + RL	6,0	0,00	6	6	314,36	32,52
	HIFE + SA	6,0	0	6	6	142,6	3360
	OCF + SA	6,0	0,00	6	6	1000	3283,4
200_07	*OCF*	2,99	0,36	2	4	344,48	18,95
	OCF + RL	0,0	0,00	0	0	986,47	28,65
	HIFE + SA	0,0	0	0	0	157,4	199,8
	OCF + SA	0,0	0,00	0	0	1000	181,6
200_08	*OCF*	8,0	0,00	8	8	51,06	17,31
	OCF + RL	8,0	0,00	8	8	45,98	31,10
	HIFE + SA	8,0	0	8	8	93	3609,2
	OCF + SA	8,0	0,00	8	8	462,8	3519,2
200_09	*OCF*	11,85	0,48	11	13	284,31	19,93
	OCF + RL	10,0	0,00	10	10	988,06	34,00
	HIFE + SA	10,0	0	10	10	124,6	3167,6
	OCF + SA	10,0	0,00	10	10	1000	3187
200_10	*OCF*	21,44	0,57	20	23	313,96	18,65
	OCF + RL	19,09	0,29	19	20	998,96	32,68
	HIFE + SA	19,6	0,55	19	20	310,4	3205

	OCF + SA	19,0	0,00	19	19	1000	3090,6
300_01	*OCF*	5,33	0,71	4	7	386,80	30,23
	OCF + RL	2,15	0,77	0	4	990,38	65,29
	HIFE + SA	1,0	0	1	1	423,0	3125
	OCF + SA	0,8	0,84	0	2	1000	2522
300_02	*OCF*	13,15	0,39	13	15	307,19	30,68
	OCF + RL	12,02	0,14	12	13	749,79	66,38
	HIFE + SA	12,4	0,55	12	13	139,0	3153
	OCF + SA	12,2	0,45	12	13	1000	3099,6
300_03	*OCF*	14,54	0,54	13	16	370,63	32,26
	OCF + RL	13,06	0,24	13	14	858,31	67,32
	HIFE + SA	13,2	0,45	13	14	170,4	3398,2
	OCF + SA	13,0	0,00	13	13	1000	3350,6
300_04	*OCF*	10,33	0,68	9	12	390,43	30,25
	OCF + RL	8,16	0,81	7	10	966,18	65,99
	HIFE + SA	8,2	0,84	7	9	332,2	3459,6
	OCF + SA	8,6	0,55	8	9	1000	3328,4
300_05	*OCF*	40,55	1,06	36	43	371,53	29,99
	OCF + RL	32,28	1,10	29	35	1000	66,09
	HIFE + SA	33,0	1,22	32	35	390,6	3151,8
	OCF + SA	31,6	0,55	31	32	1000	3057,6
300_06	*OCF*	7,59	0,74	6	9	255,48	30,21
	OCF + RL	4,38	0,99	2	7	983,85	65,92
	HIFE + SA	3,6	1,34	3	6	404,6	3187,4
	OCF + SA	3,8	0,84	3	5	1000	3198
300_07	*OCF*	2,89	0,63	1	4	173,35	30,07
	OCF + RL	0,59	0,64	0	2	973,01	62,64
	HIFE + SA	0,0	0,00	0	0	196,0	434,2
	OCF + SA	0,0	0,00	0	0	1000	444,2
300_08	*OCF*	9,17	0,38	9	10	229,22	29,00
	OCF + RL	8,0	0,00	8	8	869,05	63,21
	HIFE + SA	8,0	0,00	8	8	150,0	3497,2
	OCF + SA	8,0	0,00	8	8	1000	3334,8
300_09	*OCF*	9,05	0,56	8	10	274,34	28,76
	OCF + RL	7,46	0,67	7	9	885,19	63,47
	HIFE + SA	7,6	0,55	7	8	278,2	3310,4
	OCF + SA	7,8	0,45	7	8	1000	3224,6
300_10	*OCF*	34,63	1,04	32	37	237,32	28,57
	OCF + RL	22,6	0,85	21	25	1000	63,91
	HIFE + SA	23,4	1,14	22	25	797,8	3265
	OCF + SA	22,4	0,55	22	23	1000	3108
400_01	*OCF*	3,01	0,56	2	4	201,01	41,77

	OCF + RL	2,52	0,63	1	4	695,43	96,84
	HIFE + SA	1,6	0,5	1	2	201,8	3383,6
	OCF + SA	1,4	0,55	1	2	1000	3293,2
400_02	*OCF*	23,28	0,82	21	25	280,48	38,95
	OCF + RL	17,37	0,90	16	20	1000	96,70
	HIFE + SA	17,4	0,9	16	18	353,8	3531,4
	OCF + SA	17,2	0,84	16	18	1000	3639,8
400_03	*OCF*	11,65	0,5	10	12	212,39	38,40
	OCF + RL	9,91	0,43	9	11	954,46	94,87
	HIFE + SA	9,6	0,5	9	10	396,6	3716,2
	OCF + SA	9,8	0,45	9	10	1000	3725,6
400_04	*OCF*	21,96	0,75	20	24	454,46	43,57
	OCF + RL	19,01	0,1	19	20	994,68	98,98
	HIFE + SA	19,0	0,00	19	19	181,4	3790,4
	OCF + SA	19,0	0	19	19	1000	3761,8
400_05	*OCF*	3,48	0,96	1	6	346,67	36,47
	OCF + RL	0,01	0,10	0	1	979,21	76,85
	HIFE + SA	0,0	0,00	0	0	164,4	149,6
	OCF + SA	0,0	0	0	0	1000	225
400_06	*OCF*	4,20	0,97	1	6	263,12	39,85
	OCF + RL	0,33	0,53	0	2	997,54	87,56
	HIFE + SA	0,0	0,00	0	0	179,2	416,8
	OCF + SA	0,0	0	0	0	884,8	308,4
400_07	*OCF*	7,65	0,83	5	9	260,60	36,93
	OCF + RL	5,44	0,80	4	7	989,49	93,29
	HIFE + SA	4,2	0,4	4	5	232,4	3405,4
	OCF + SA	4,0	0	4	4	1000	3373,6
400_08	*OCF*	11,54	1,62	6	14	179,93	35,51
	OCF + RL	5,30	0,81	4	7	1000	93,39
	HIFE + SA	4,0	0,00	4	4	341,6	3164
	OCF + SA	4,0	0	4	4	1000	3250,4
400_09	*OCF*	17,98	1,10	15	20	209,19	41,38
	OCF + RL	7,63	0,95	5	10	1000	97,75
	HIFE + SA	7,2	1,1	6	9	471,8	3421
	OCF + SA	7,0	0,71	6	8	1000	3458,2
400_10	*OCF*	4,24	0,99	2	7	77,19	40,94
	OCF + RL	0,95	0,72	0	3	991,09	93,89
	HIFE + SA	0,0	0,00	0	0	124,2	98
	OCF + SA	0,0	0	0	0	1000	121,8

Tableau 3.20 : Résultats de l'hybridation à relais sur les problèmes de l'*ET3*

3.5 Quelques constats intéressants

3.5.1 Influence du PLNE sur l'OCF

Dans cette section, nous examinons de quelle façon les hybridations intégratives influencent le travail de l'*OCF*. Pour bien illustrer ce propos, nous utilisons les résultats trouvés pour le problème 300_05 de l'*ET3*. Tout d'abord, la Figure 3.9 montre la progression de l'*OCF* sans recherche locale. Les points bleus indiquent la valeur de la meilleure solution trouvée à chaque cycle. Le trait jaune indique la valeur de la meilleure solution trouvée depuis le démarrage de l'algorithme. On remarque que la qualité des solutions trouvées à chaque cycle n'évolue pas beaucoup avec le temps. La Figure 3.10 montre la progression de la qualité des solutions trouvées par la méthode *HIFE*. Les points bleus représentent la meilleure solution de chaque cycle de l'*OCF*. On observe que la collaboration survient au tout début de l'algorithme. Vers le 125^{e} cycle, le *PLNE* fait passer la meilleure solution trouvée de 44 conflits à une solution de 37 conflits. Ensuite, 100 cycles plus tard, le *PLNE* améliore à nouveau cette solution et la fait passer à 34 conflits. À en juger par ce graphique, on peut conclure que l'*OCF* n'est pas très influencée par les solutions trouvées par le *PLNE*. Cette métaheuristique ne s'avère donc pas très appropriée dans la réalisation d'hybridations intégratives. Nous estimons que l'algorithme génétique, quant à lui, serait peut-être plus approprié pour ce genre d'hybridation, car il pourrait intégrer directement dans sa population les solutions trouvées par le *PLNE*.

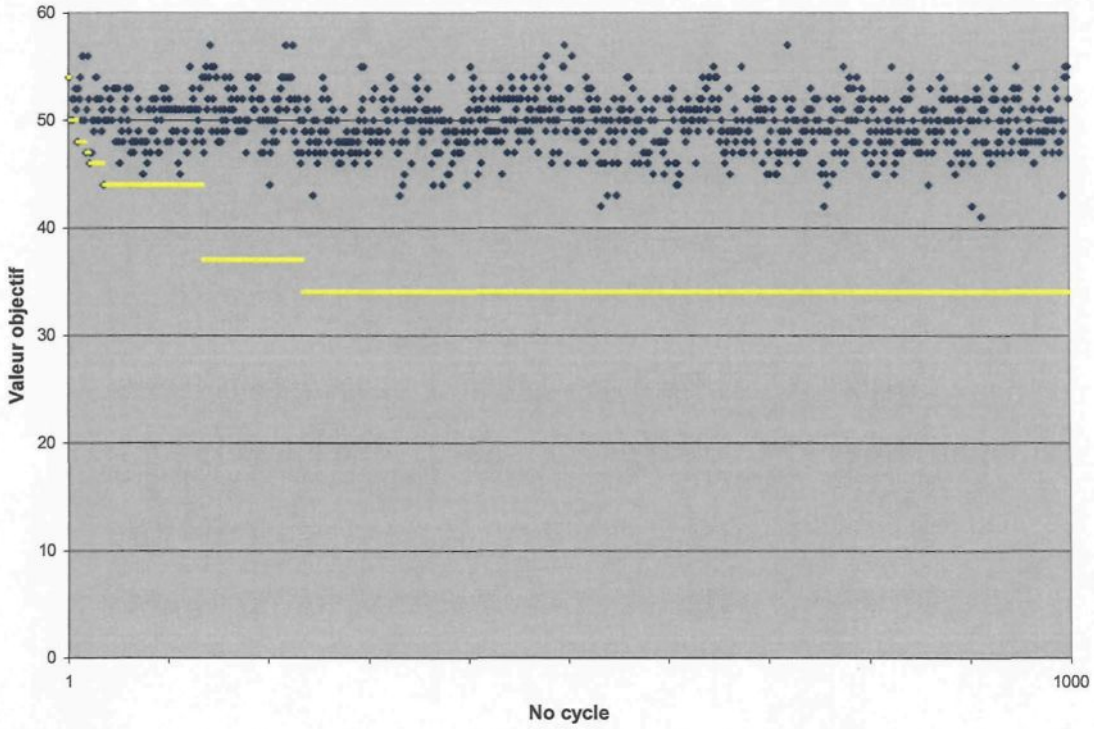

Figure 3.10 : Évolution de la qualité des solutions trouvées par l'*HIFE*

3.5.2 Temps accordé au PLNE

Le grand désavantage de l'utilisation de la *PLNE* pour résoudre des sous-problèmes est que les temps d'exécution sont élevés. Tous les essais effectués dans les sections précédentes ont accordé 5 secondes à chaque appel du *PLNE*. Dans le but de vérifier si le temps accordé a une influence sur la qualité des résultats, les expérimentations suivantes sont effectuées avec la méthode *HIFE* sur les problèmes de l'*ET3*, avec des limites de temps allant de 2 à 5 secondes et en effectuant cinq essais pour chaque cas. Le Tableau 3.21 présente les résultats de ces essais. On remarque que le temps accordé au *PLNE* n'a

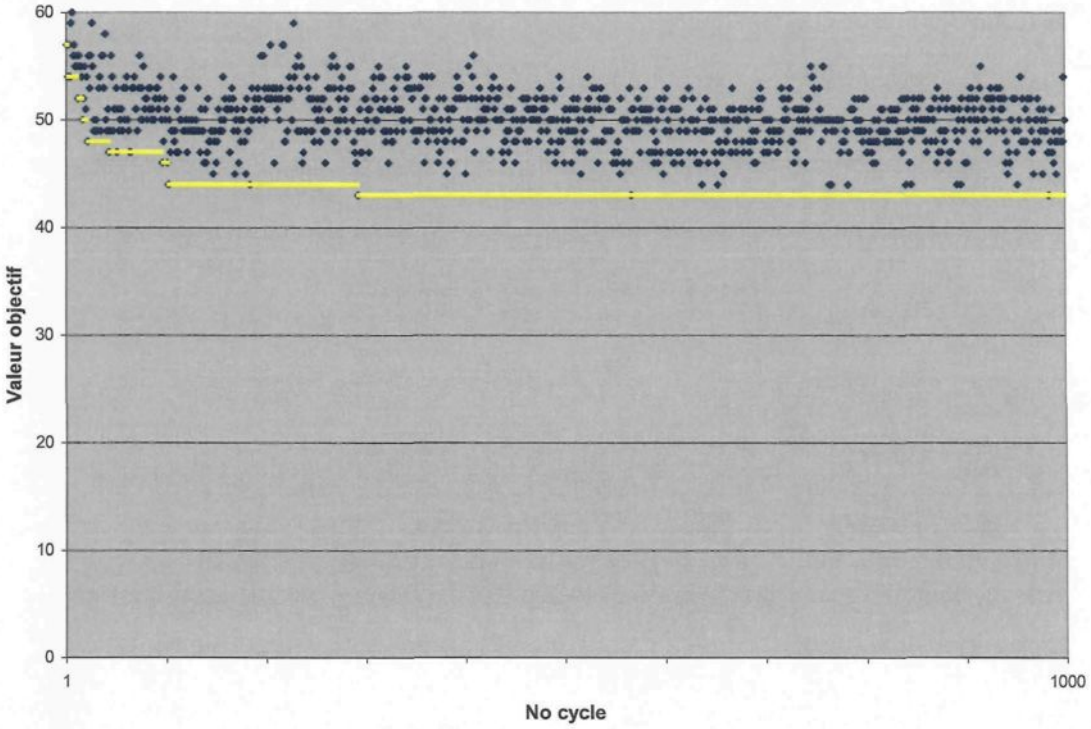

Figure 3.9 : Évolution de la qualité des solutions trouvées par l'*OCF*

pas une énorme influence sur les résultats obtenus. Il est alors possible d'accélérer le temps de calcul d'une hybridation avec un *PLNE*, sans détériorer la qualité de ses résultats, en diminuant le temps qu'on lui accorde pour solutionner un sous-problème. Le pourcentage de positions libres dynamique permet alors de s'ajuster à la capacité du *PLNE* de solutionner un sous-problème.

Problème	Moyenne de conflits selon le temps accordé au *PLNE*			
	5 sec	4 sec	3 sec	2 sec
200_01	0,0	0,2	0,2	0,0
200_02	2,4	2,8	2,6	2,8
200_03	6,2	6,0	6,6	5,6
200_04	7,0	7,0	7,2	7,4
200_05	6,0	6,0	6,0	6,0
200_06	6,0	6,0	6,0	6,0
200_07	0,0	0,0	0,0	0,0
200_08	8,0	8,0	8,0	8,0
200_09	10,0	10,0	10,0	10,0
200_10	19,0	19,6	19,2	19,2
300_01	0,8	0,8	0,8	1,4
300_02	12,2	12,0	12,2	12,2
300_03	13,0	13,0	13,0	13,4
300_04	8,6	8,0	8,4	8,0
300_05	31,6	33,6	32,6	33,8
300_06	3,8	3,2	3,4	4,0
300_07	0,0	0,0	0,0	0,0
300_08	8,0	8,0	8,0	8,0
300_09	7,8	7,2	8,2	7,6
300_10	22,4	23	23,2	23,8
400_01	1,4	1,8	2,0	1,4
400_02	17,2	17,8	18,6	19,6
400_03	9,8	9,8	10,0	10,4
400_04	19,0	19,0	19,2	19,0
400_05	0,0	0,0	0,0	0,0
400_06	0,0	0,0	0,0	0,0
400_07	4,0	4,0	4,0	4,0
400_08	4,0	4,2	4,2	4,4
400_09	7,0	7,8	7,4	6,8
400_10	0,0	0,0	0,0	0,0

Tableau 3.21 : Relation entre le nombre de conflits moyens et le temps accordé au *PLNE* avec la méthode *HIFE* sur les problèmes de l'ET3

3.6 Conclusion

On constate que les trois formes d'hybridation proposées dans les sections précédentes ont permis d'améliorer les résultats de l'*OCF* sans recherche locale. On peut également conclure, à la lumière des résultats présentés, que ce sont les hybridations effectuant des intensifications sur une solution, telles que l'*HIFE* et l'hybridation à relais, qui produisent les meilleurs résultats. Celles-ci peuvent rivaliser avec les techniques de recherche locale utilisées par l'*OCF* en ce qui concerne la qualité des solutions produites, mais le temps requis se révèle nettement plus important. Parmi les hybridations présentées dans cette section, c'est l'hybridation à relais qui s'avère être la plus intéressante par sa performance et sa simplicité. Elle pourrait facilement être jumelée à l'*OCF* avec recherche locale pour des résultats potentiellement excellents. L'*HIFE*, quant à elle, pourrait peut-être donner de meilleurs résultats en calibrant différemment les paramètres α, β et δ de l'*OCF*. Ces derniers ont été calibrés par Gravel, Gagné *et al.* [42] pour le fonctionnement de l'OCF avec et sans recherche locale. L'ajustement de ces paramètres est toutefois un travail de précision demandant beaucoup de temps et qui dépasse le cadre de ce travail. Il demeure que le désavantage majeur des hybridations présentées est leur temps d'exécution élevé. Ceci constitue un sérieux obstacle pour leur utilisation quotidienne dans une industrie.

CHAPITRE 4 :

CONCLUSION

La présente recherche visait à proposer une approche efficace de solution au problème d'ordonnancement de voitures (*POV*). Dans cette optique, un premier objectif spécifique visait à concevoir des méthodes de résolution hybrides adaptées au *POV* en combinant une métaheuristique et une méthode exacte. À cet effet, nous avons proposé trois différentes formes d'hybridation entre l'*OCF* et le *B&B*. Deux d'entre elles sont des hybridations de type intégrative, où un sous-problème est résolu par le *PLNE* pendant le fonctionnement de l'*OCF*. La troisième hybridation proposée est de type à relais, où l'*OCF* fournit la meilleure solution obtenue, par son processus complet, à la méthode *RLE* qui démarre, à son tour, un processus itératif de recherche.

Avant même de créer ces hybridations, différentes stratégies de création de sous-problèmes plus faciles à résoudre ont été proposées. Ces stratégies consistaient à choisir les voitures de la chaîne d'assemblage à fixer, pour permettre au *PLNE* de résoudre le problème en ordonnant les voitures laissées libres. Le choix des voitures fixées ou libérées est fait de façon aléatoire et en tenant compte de l'emplacement des conflits et de certaines propriétés des problèmes d'ordonnancement de voitures, telle que le taux d'utilisation d'une classe de voiture. Après avoir identifié les stratégies les plus prometteuses, on a procédé à l'élaboration des méthodes *HICC*, *HIFE* et de l'hybridation à relais. Ces hybridations consistent à combiner la rapidité de l'*OCF* sans recherche locale proposée par Gravel, Gagné *et al.* [42] et la force brute d'une méthode exacte.

Le deuxième objectif spécifique visait à établir la performance des hybridations proposées par rapport à d'autres méthodes existantes pour la résolution du problème d'ordonnancement de voitures. Chacune des hybridations proposées dans cette recherche a

été comparée avec les *OCF*, avec et sans recherche locale, de Gravel, Gagné *et al.* [42]. Dans un premier temps, les résultats des expériences menées ont montré que les trois hybridations proposées obtiennent de meilleurs résultats que l'OCF sans recherche locale. La situation est différente lorsque les hybridations sont comparées à l'*OCF* avec recherche locale. Les résultats ont montré que l'*HICC* donne de moins bons résultats que l'*OCF* avec recherche locale, car elle n'effectue pas d'intensification de solution. Les deux autres hybridations, axées sur l'intensification, ont justement montré qu'elles peuvent être tout aussi performantes que l'*OCF* avec recherche locale. L'inconvénient majeur des hybridations proposées est que le *PLNE* demande un énorme temps d'exécution. En fait, son exécution est trop longue pour considérer l'utilisation de cette méthode dans une industrie.

De plus, on a étudié plus en détails le comportement de l'hybridation entre le *PLNE* et l'*OCF*. On a remarqué que l'*OCF* n'intègre pas suffisamment la qualité des solutions trouvées par le *PLNE* dans sa trace de phéromone. La collaboration entre ces deux méthodes se fait dans les 100 à 150 premiers cycles. Dans les cycles suivants, seul le *PLNE* effectue tout le travail. Une meilleure collaboration aurait pu être obtenue en ajustant les différents paramètres de l'*OCF*, mais il s'agit d'un travail de précision qui sort du cadre de cette recherche.

Malgré le fait que le temps soit un obstacle à l'utilisation industrielle des méthodes hybrides développées dans cette recherche, il serait intéressant d'observer leurs performances sur de vrais problèmes industriels. En adaptant à la résolution de cas réels les méthodes hybrides ainsi que l'*OCF* avec recherche locale proposée par Gravel, Gagné *et al.*

[42], on pourrait comparer leurs résultats respectifs pour savoir laquelle de ces méthodes produit les meilleurs résultats dans ce nouveau contexte.

Il serait aussi intéressant de concevoir une autre méthode hybride en utilisant une métaheuristique qui intégrerait, de façon plus directe, les solutions fournies par le *PLNE*. Par exemple, l'algorithme génétique pourrait ajouter dans sa population une solution fournie par le PLNE à chaque génération. De cette façon, les individus créés à chaque génération pourraient posséder des gènes des solutions optimisées par le *PLNE*.

BIBLIOGRAPHIE

1. Abbattista, F., N. Abbattista et L. Caponetti. *An Evolutionary and Cooperative Agent Model for Optimization*, in *IEEE International Conference on Evolutionary Computation ICEC'95*, 1995, Perth, Australia, pp. 668-671.

2. Ahuja, R.K., Ö. Ergun, J.B. Orlin et A.P. Punnen, *A Survey of Very Large-scale Neighborhood Search Technique.* Discrete Applied Mathematics, 2002. 123(1-3): pp. 75-102.

3. Angel, E., E. Bampis, M. Rahoual et H. Bouchema. *Ants Metaheuristic and Constraint Programming Cooperation for the Protein Folding Problem*, in *MIC2005, The 6th Metaheuristics International Conference*, 2005, Vienna, Austria, pp. 57-62.

4. Applegate, D., R. Bixby, V. Chvátal et W. Cook, *On the Solution of Travelling Salesman Problems.* Documenta Mathematica, Extra Volume ICM III, 1998: pp. 645-656.

5. Applegate, D. et W. Cook, *A Computational Study of the Job-Shop Scheduling Problem.* ORSA Journal on Computing, 1991. 3(2): pp. 149-156.

6. Azimi, Z.N., *Hybrid Heuristics for Examination Timetabling Problem.* Applied Mathematics and Computation, 2005. 163(2): pp. 705-733.

7. Bachelet, V., Z. Hafidi, P. Preux et E.-G. Talbi. *Diversitying Tabu Search by Genetic Algorithms*, in *INFORMS'98 on Operations Research and Management Sciences meeting*, 1998, Montréal, Canada

8. Barnhart, C., E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh et P.H. Vance, *Branch-and-Price: Column Generation for Solving Huge Integer Programs.* Operations Research, 1998. 46(3): pp. 316-329.

9. Belding, T.C. *The Distributed Genetic Algorithm Revisited*, in *Sixth International Conference on Genetic Algorithms*, 1995, D. Eshelmann (Ed), San Mateo, CA

10. Bentley, J.J., *Fast Algorithms for Geometric Traveling Salesman Problems.* ORSA Journal on Computing, 1992. 4(4): pp. 387-411.

11. Boivin, S., *Résolution d'un problème de satisfaction de contraintes pour l'ordonnancement d'une chaîne d'assemblage automobile, Mémoire de maitrise*, in

Département d'Informatique et de Mathématique. 2005, Université du Québec à Chicoutimi: Chicoutimi. p. 118.

12. Boivin, S., M. Gravel, M. Krajeckti et C. Gagné, *Résolution du problème de car-sequencing à l'aide d'une approche de type FC.* Actes des Premières Journées Francophones de Programmation par Contraintes, Université d'Artois, 2005: pp. 11-20.

13. Bouzgarrou, M.E., *Parallélisation de la méthode du "Branch and Cut" pour résoudre le problème du voyageur de commerce, Thèse de doctorat*, in *Institut d'Informatique et de Mathématiques Appliquées de Grenoble*. 1998, Institut National Polytechnique de Grenoble: Grenoble, France.

14. Büdenbender, K., T. Grünert et H.-J. Sebastian, *A Hybrid Tabu Search/Branch-and-Bound Algorithm for the Direct Flight Network Design Problem.* Transportation Science, 2000. 34(4): pp. 364-380.

15. Burke, E.K., P.I. Cowling et R. Keuthen. *Effective Local And Guided Variable Neighborhood Search Methods for the Asymmetric Travelling Salesman Problem.*, in *Applications of Evolutionary Computing: EvoWorkshops 2001, volume 2037 of LNCS*, 2001, S.B. Heidelberg (Ed), Springer, pp. 203-212.

16. Cerny, V., *Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm.* Journal of Optimization Theory and Application, 1985. 45(1): pp. 41-51.

17. Chen, S.Y., S.N. Talukdar et N.M. Sadeh. *Job Shop Scheduling by a Team of Asynchronous Agents*, in *Proceedings of the IJCAI '93 Workshop on Knowledge-Based Production, Scheduling and Control*, 1993, Chambery, France

18. Chew, T.-L., J.-M. David, A. Nguyen et Y. Tourbier. *Solving Constraint Satisfaction Problem with Simulated Annealing: The Car Sequencing Problem Revisited*, in *Proceeding of the 12th International Conference on Artificial Intelligence, Expert Systems and Natural Language*, 1991, pp. 405-416.

19. Chu, P.C. et J.E. Beasley, *A Genetic Algorithm for the Multidimensional Knapsack Problem.* Journal of Heuristics, 1998. 4(1): pp. 63-86.

20. Congram, R.K., C.N. Potts et S.L.v.d. Velde, *An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem.* INFORMS Journal on Computing, 2002. 14(1): pp. 52-67.

21. Cormen, T.H., C.E. Leiserson, R.L. Rivest et C. Stein, *Introduction à l'algorithmique*, Dunod (Ed.) 2002: Paris. pp 745-794.

22. Cotta, C., E.-G. Talbi et E. Alba, *Parallel Hybrid Metaheuristics*, in *Parallel Metaheuristics: A New Class of Algorithms*, E. Alba (Ed.) 2005, Wiley-Interscience. pp 347-370.

23. Cotta, C. et J.M. Troya, *Embedding Branch and Bound Within Evolutionary Algorithms.* Applied Intelligence, 2003. 18: pp. 137-153.

24. Davenport, A. et E. Tsang. *Solving Constraint Satisfaction Sequencing Problems by Iterative Repair*, in *Proceedings of the First International Conference on the Practical Applications of Constraint Technologies and Logic Programming, Practical Applications Company*, 1999, London, England, pp. 345-357.

25. Davenport, A., E. Tsang, C.J. Wang et K. Zhu. *GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement*, in *Proceedings of AAAI'94*, 1994, Seattle, Washington, Menlo Park, California, AAAI Press, pp. 325-330.

26. Denzinger, J. et T. Offermann. *On Cooperation Between Evolutionary Algorithms and Other Search Paradigms*, in *Proceedings of the 1999 Congress on Evolutionary Computation*, 1999, Washington DC, USA, IEEE Press, pp. 2317-2324.

27. Dincbas, M., H. Simonis et P.V. Hentenryck. *Solving the Car-Sequencing Problem in Constraint Logic Programming*, in *Proceedings of the European Conference on Artificial Intelligence (ECAI-88)*, 1988, Munich, Germany, Pittman Publishing, pp. 290-295.

28. Dorigo, M., V. Maniezzo et A. Colorni, *The Ant System: Optimization by a Colony of Cooperating Agents.* IEEE Transactions on Systems, Man, and Cybernetics-Part B, 1996. 26(1): pp. 1-13.

29. Dorigo, M. et T. Stützle, *Ant Colony Optimization.* 2004, Cambridge, Massachusetts et London, England: The MIT Press.

30. Dueck, G. et T. Scheuer, *Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing.* Journal of Computational Physics, 1990. 90: pp. 161-175.

31. Dumitrescu, I. et T. Stützle, *A Survey of Methods that Combines Local Search and Exact Algorithms.* Rapport technique AIDA-03-07, FG Intellektik, FB Informatik, TU Darmstadt, Allemagne, 2003.

32. Feltl, H. et G.R. Raidl. *An Improved Hybrid Genetic Algorithm for the Generalized Assignment Problem*, in *Proceedings of the 2004 ACM Symposium on Applied Computing*, 2004, Nicosia, Cyprus, ACM Press, pp. 990-995.

33. Filho, G.R. et L.A.N. Lorena. *Constructive Genetic Algorithm and Column Generation: an Application to Graph Coloring*, in *APORS 2000 - The Fifth Conference of the Association of Asian-pacific Operations Research Societies within INFORS*, 2000

34. Fischetti, M. et A. Lodi, *Local Branching.* Mathematical Programming Series B, 2003. 98: pp. 23-47.

35. Fleurent, C. et J.A. Ferland, *Genetic Hybrids for the Quadratic Assignment Problem.* DIMACS Series in Mathematics and Theoretical Computer Science, 1994. 16: pp. 190-206.

36. French, A.P., A.C. Robinson et J.M. Wilson, *Using a Hybrid Genetic-Algorithm/Branch and Bound Approach to Solve Feasibility and Optimization Integer Programming Problems.* Journal of Heuristics, 2001. 7: pp. 551-564.

37. Gagné, C., M. Gravel et W.L. Price, *Solving Real Car Sequencing Problems with Ant Colony Optimization.* European Journal of Operational Research, 2006. 174(3): pp. 1427-1448.

38. Gambardella, L., É. Taillard et M. Dorigo, *Ant Colonies for the Quadratic Assignment Problem.* Journal of the Operational Research Society, 1999. 50: pp. 167-176.

39. Gent, I.P. et T. Walsh, *CSPLib: A Benchmark Library for Constraints.* Research Reports of the APES Groups, APES-09-1999, 1999.

40. Glover, F., *Future Paths for Integer Programming and Links to Artificial Intelligence.* Computers and Operations Research, 1986. 5: pp. 533-549.

41. Gottlieb, J., M. Puchta et C. Solnon. *A Study of Greedy, Local Search, and Ant Colony Optimization Approaches for Car Sequencing Problems.*, in *Applications of Evolutionary Computing, Lecture Notes in Computer Science*, 2003, Heidelberg, Springer-Verlag, pp. 246-257.

42. Gravel, M., C. Gagné et W.L. Price, *Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem.* Journal of the Operational Research Society, 2005. 56: pp. 1287-1295.

43. Hammami, M. et K. Ghédira. *Cooperation between Simulated Annealing, Tabu Search, X-over operator and Kernighan and Lin Heuristic for the K-Graph Partitioning Problem*, in *MIC2005, The 6th Metaheuristics International Conference*, 2005, Vienna, Austria, pp. 480-485.

44. Hammami, M. et K. Ghédira. *COSATS: A new Cooperation Model between Simulated Annealing and Tabu Search for the K-Graph Partitioning Problem*, in *The Fourth IEEE International Workshop on Soft Computing as Transdisciplinary Science and Technology*, 2005, Mororan, Japan, pp. 863-873.

45. Hansen, P. et N. Mladenovic, *Variable Neighborhood Decomposition Search.* Journal of Heuristics, 2001. 7(4): pp. 335-350.

46. Hao, J.-K., P. Galinier et M. Habib, *Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes.* Revue d'Intelligence Artificielle, 1999. 13(2): pp. 283-324.

47. Holland, J.H., *Adaptation in Natural and Artificial Systems*, 1975, MIT Press.

48. Kernighan, B. et S. Lin, *An Efficient Heuristic Procedure for Partitioning Graphs.* Bell Systems Technical Journal, 1970. 49(2): pp. 291-308.

49. Kirkpatrick, S., C.D.G. Jr. et M.P. Vecchi, *Optimization by Simulated Annealing.* Science, 1983. 220(4598): pp. 671-680.

50. Kis, T., *On the Complexity of the Car Sequencing Problem.* Operations Research Letters, 2004. 32(4): pp. 331-335.

51. Klau, G.W., I. Ljubic, A. Moser, P. Mutzel, P. Neuner, U. Pferschy, G. Raidl et R. Weiskircher. *Combining a Memetic Algorithms with Integer Programming to Solve the Prize-Collecting Steiner Tree Problem.*, in *Genetic and Evolutionary Computation - GECCO*, 2004, pp. 1304-1315.

52. Krueger, M., *Méthodes d'analyse d'algorithmes d'optimisation stochastiques à l'aide d'algorithmes génétiques.* 1993, École Nationale Supérieure des Télécommunications: Paris, France.

53. Lee, J., H. Leung et H. Won. *Performance of a Comprehensive and Efficient Constraint Library Based on Local Search*, in *11th Australian Joint Conference on Artificial Intelligence*, 1998, J.K.S. G. Antoniou (Ed), Heidelberg, Brisbane, Australia, Springer-Verlag, pp. 191-202.

54. Lin, F.T., C.Y. Kao et C.C. Hsu. *Incorporating Genetic Algorithms into Simulated Annealing*, in *Fourth International Symposium On Artificial Intelligence*, 1991, pp. 290-297.

55. Martin, O.C. et S.W. Otto, *Combining Simulated Annealing with Local Search Heuristics.* Annals of Operations Research, 1996. 63: pp. 57-75.

56. Mautor, T. et P. Michelon, *MIMAUSA: an Application of Referent Domain Optimization.* Technical Report 260, Laboratoire d'Informatique, Université d'Avignon et des Pays de Vaucluse, Avignon, France, 2001.

57. McKendall, A.R. et J. Shang, *Hybrid And Systems for the Dynamic Facility Layout Problem.* Computers and Operations Research, 2006. 33: pp. 790-803.

58. Monfroglio, A., *Hybrid Heuristic Algorithms for Set Covering.* Computers and Operations Research, 1998. 25(6): pp. 441-455.

59. Morin, S., *Algorithme de fourmis avec apprentissage et comportement spécialisés pour l'ordonnancement de voitures, Mémoire de maîtrise*, in *Département d'Informatique et de Mathématique*. 2005, Université du Québec à Chicoutimi: Chicoutimi. p. 162.

60. Nemhauser, G.L. et L.A. Wolsey, *Integer and Combinatorial Optimization.* 1999, New-York, Chichester, Brisbane, Toronto, Singapore: Wiley-Interscience.

61. Nicholson, T., *Optimization Techniques*, in *Optimization in Industry*1971, Longmann Press: London.

62. Osman, I.H. et G. Laporte, *Metaheuristics: A bibliography.* Annals of Operations Research, 1996. 63: pp. 513-623.

63. Palpant, M., C. Artigues et P. Michelon, *LSSPER: Solving the Resource-Constrained Project Scheduling Problem with Large Neighbourhood Search.* Annals of Operations Research, 2004. 131: pp. 237-257.

64. Parello, B.D., *CAR WARS: The (Almost) Birth of an Expert System.* AI Expert, 1988. 3: pp. 60-64.

65. Parello, B.D., W.B. Kabat et L. Wos, *Job-Shop Scheduling Using Automated Reasoning: A Case Study of the Car-Sequencing Problem.* Journal of Automated Reasoning, 1986. 2: pp. 1-42.

66. Puchinger, J. et G.R. Raidl. *Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification*, in *Proceedings of the*

First International Work-Conference on the Interplay Between Natural and Artificial Computation, 2005, Springer (Ed), Las Palmas, Spain, LNCS, pp. 41-53.

67. Puchta, M. et J. Gottlieb, *Solving Car Sequencing Problems by Local Optimization*, in *S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, G.R. Raidl (Eds), Proceedings EvoWorkshops, Lecture Notes in Computer Science*, S.B. Heidelberg (Ed.) 2002: Kinsale, Ireland. pp 181-188.

68. Raidl, G.R. *An Improved Genetic Algorithm for the Multiconstrained 0-1 Knapsack Problem*, in *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, 1998, D.B. Fogel (Ed), IEEE Press, pp. 207-211.

69. Ribeiro, M.H., V. Trindade, A. Plastino et S. Martins, *Hybridization of GRASP Metaheuristic with Data Mining Techniques.* Special Issue on Hybrid Metaheuristic of the Journal of Mathematical Modelling and Algorithms, 2005.

70. Salcedo-Sanz, S., Y. Xu et X. Yao, *Hybrid Meta-Heuristics Algorithms for Task Assignment in Heterogeneous Computing Systems.* Computers and Operations Research, 2006. 33: pp. 820-835.

71. Shahookar, K. et P. Mazumder, *A Genetic Approach to Standard Cell Placement Using Meta-Genetic Parameter Optimization.* IEEE Transactions on Computer-Aided Design, 1990. 9(5): pp. 500-511.

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

73. Solnon, C. *Solving Permutation Constraint Satisfaction Problems with Artificial Ants.*, in *14th European Conference on Artificial Intelligence (ECAI)*, 2000, H. Werner (Ed), IOS Press, pp. 118-122.

74. Souza, P.d., *Asynchronous Organizations for Multi-Algorithm Problems*, in *Dept. of Electrical and Computer Engineering.* 1993, Carnegie Mellon University: Pittsburgh, PA.

75. Staggemeier, A.T., A.R. Clark, U. Aickelin et J. Smith. *A Hybrid Genetic Algorithm to Solve a Lot-Sizing and Scheduling Problem*, in *16th Triannual Conference of the International Federation of Operational Research Societies*, 2002, Edinburgh, U.K.

76. Stützle, T. et H. Hoos. *MAX-MIN Ant System and Local Search for Combinatorial Optimization Problems*, in *2nd International Conference on Metaheuristics*, 1997, Sopia-Antipolis, France, pp. 191-193.

77. Taillard, É.D., *Parallel Iterative Search Methods for Vehicle Routing Problems.* Networks, 1993. 23: pp. 661-673.

78. Taillard, É.D. et G. Burri, *POPMUSIC pour le placement de légendes sur des plans.* dans É. D. Taillard, Ph. Waelti, M. Widmer (eds), Actes de FRANCORO 4, Fribourg, Suisse, 2004: pp. 95-97.

79. Taillard, É.D. et S. Voss, *POPMUSIC: Partial Optimization Metaheuristic Under Special Intensification Conditions.* Dans C.C. Ribeiro, P.Hansen (eds), Essays and Surveys in Metaheuristics, Kluwer academic publishers, 2001: pp. 613-629.

80. Talbi, E.-G., *A Taxonomy of Hybrid Metaheuristics.* Journal of Heuristics, 2002. 8: pp. 541-564.

81. Talukdar, S., L. Baerentzen, A. Gove et P.d. Souza, *Asynchronous Teams: Cooperation Schemes for Autonomous Agents.* Journal of Heuristics, 1998. 4: pp. 295-321.

82. Tanese, R. *Parallel genetic Algrithms for a Hypercube*, in *Proceedings of the Second International Conference on Genetic Algorithms*, 1987, MIT, Cambridge, MA, USA, pp. 177-183.

83. Tsen, C.K., *Solving Train Scheduling Problems Using A-Teams, Thèse de doctorat,* in *Dept. of Electrical and Computer Engineering.* 1995, Carnegie Mellon University: Pittsburgh, PA.

84. Warwick, T. et E.P.K. Tsang, *Tackling Car Sequencing Problems Using a Generic Genetic Algorithm.* Evolutionary Computation, 1995. 3(3): pp. 267-298.

85. Widmer, M. *Les métaheuristiques: des outils performants pour les problèmes industriels.*, in *3ième conférence francophone de modélisation et de simulation "Conception, Analyse et Gestion des Systèmes Industriels"*, 2001, Troyes, pp. 13-21.

86. Woodruff, D.L. *A Chunking Based Selection Strategy for Integrating Meta-heuristics with Branch and Bound*, in *Metaheuristics: Advances and Trends in Local Search Paradigms for Optimization*, 1999, Kluwer Academic Publishers, pp. 499-511.

87. Zhu, K., *Unpublished Results in the GENET Project, EPSERC funded (UK), reference GR/H75275.* 1993.