Université Claude Bernard Lyon 1
Laboratoire d'Informatique en Image et Système d'information (LIRIS)

Doctorat

présentée en première version en vu d'obtenir le grade de Docteur,
spécialité « Informatique »

par

Madjid KHICHANE

Optimisation sous contraintes par Intelligence Collective Auto-adaptative

Thèse soutenue le 26 octobre 2010 devant le jury composé de :

M.	Jin-Kao HAO	Professeur à l'université d'Angers	(Rapporteur)
M.	Gérard VERFAILLIE	Ingénieur de recherches à l'ONERA à Toulouse (HdR)	(Rapporteur)
M^{me}	Christine SOLNON	Maître de conférences à l'Université Lyon 1 (HdR)	(Directrice)
M.	Patrick ALBERT	Directeur des Recherches à IBM	(Co-Directeur)
M.	Jean-Michel JOLION	Professeur à l'Université de Lyon	(Examinateur)
M.	Thomas STÜTZLE	Chargé de recherches FNRS à L'IRIDIA à Bruxelles	(Examinateur)

À mes parents, à mes frères, à ma soeur et à kahina. . .

Remerciements

Tout d'abord, je tiens à remercier mes encadrants :

– Christine Solnon : et oui, des longs et des longs paragraphes ne seront évidemment pas assez pour te remercier ! Mais quand même, je vais tenter de le faire en essayant d'appliquer ce que tu n'a pas arrêté de m'apprendre à chaque fois que je t'envoyais mes rapports (...ne dire que le nécessaire et le plus important...pour que ça soit compréhensible et agréable à lire...et sauver la planète !). Christine, merci pour tout, en allant de la confiance que tu m'as accordé jusqu'au sens de la rigueur que j'ai appris en travaillant avec toi.
Oui, parfois je me disais : "Que-est ce qu'elle est exigeante !". Mais sans cette exigence et cette rigueur scientifique, nous travaux ne seront sans doute pas aboutis. J'espère que mon aventure scientifique avec toi est renouvelable.

– Patrick Albert : également, histoire de préserver la planète, je vais éviter de longues pages de remerciements. Patrick, merci pour la confiance, l'oreille et tout le temps que tu m'as accordé. La naissance et l'aboutissement de nos publications, et cette thèse même, est en grande partit le fruit de tous ces moments que nous avons passé ensemble à travailler et à échanger nos idées devant ce tableau blanc avec des feutres dans nos mains en train d'écrire des formules, de dessiner des graphes, des vecteurs etc.

Mon profond respect et ma gratitude se dirigent aussi vers Gérard Verfaillie et Jin-Kao Hao d'avoir accepté de rapporter cette thèse. Je remercie également Thomas Stuetzle et Jean-Michel Jolion d'avoir accepté de m'honorer en faisant parti du jury de ma thèse.

Mes remerciements vont également à tous les membres de l'équipe Optim d'ILOG et particulièrement à Jérôme Regerie avec qui la discussion est toujours instructive ; à Renaud Dumeur pour tous ces conseils utiles ; à Philippe Laborie et Philippe Rifalo pour toutes nos discussions fructueuses sur l'optimisation ; à Paul Shaw et tous ceux que je viens de citer pour leurs relectures qui ont permis l'amélioration de nos articles et je les remercie tous pour les discutions fructueuses sur l'optimisation. Je remercie également ces ilogiens avec qui j'ai passé des moments agréables autours d'une table à 64 cases, je cite, Pascal Lenormand, Gilles Benati,

Robert Dupuy, Georges Schumacher, Benoit Nachawati et tous ceux avec qui j'ai partagé des choses autour d'un échiquier et/ou d'un café à ILOG.

Bien sûr, je remercie également tous les membres de l'équipe ILOG-Research Project pour leurs convivialité et leurs bonne humeur et particulièrement je remercie Marcos Didonet del Fabro et Thomas Bodel pour leurs conseils très utiles et leurs soutien morale qu'ils m'ont apporté au moment voulu.

Egalement, je tiens à remercier tous mes amis qui m'ont soutenu de loin ou de près. Et particulièrement, je remercie, Lotfi, Amine, Hakim, Sylvain et bien sûr ma chérie Kahina

Lieu, le 27 août 2010.

Table des matières

Liste des figures

Introduction

Les problèmes d'optimisation combinatoires (COP) sont d'une importance conséquente aussi bien dans le monde scientifique que dans le monde industriel. La plupart des COP sont *NP*-difficiles. Par conséquent, à moins que $P = NP$, ils ne peuvent pas être résolus de façon exacte en un temps polynomial. Parmi les COP qui sont *NP*-difficiles nous pouvons citer par exemple les problèmes du voyageur de commerce (TSP), d'ordonnancement de voitures, du sac-à-dos multidimensionnel (MKP), d'affectation quadratique (QAP) et de clique maximum. Pour résoudre ces COP, deux types d'approches différentes mais complémentaires peuvent être utilisées, à savoir, les approches complètes et les approches par métaheuristiques.

Approches complètes Les approches complètes explorent l'espace des combinaisons de façon exhaustive. Cet espace est structuré en un arbre qui est exploré de façon systématique en utilisant généralement des algorithmes de type *Branch & Bound* (*B&B*). Durant la recherche des solutions, ces algorithmes utilisent des fonctions d'évaluation pour réduire l'espace de recherche à visiter. En effet, à chaque noeud de l'arbre de recherche, une borne sur la fonction objectif est évaluée. Si cette borne s'avère moins bonne que la meilleure solution trouvée jusqu'alors, le noeud en question n'est pas développé. Ces approches permettent de trouver la solution optimale en une limite de temps finie. Cependant, en raison de leur complexité qui est exponentielle dans le pire des cas, leur utilisation est généralement limitée à des problèmes de taille raisonable.

En plus de la fonction objectif à optimiser, les COP qui relèvent du monde réel, notamment du monde industriel, comportent bien souvent des contraintes à satisfaire. Ces contraintes peuvent être fortes de sorte que trouver une solution qui les satisfait est en soit même un problème difficile. Pour résoudre de tels COPs, l'algorithme *B&B* peut être raffiné en ajoutant une phase de propagation des contraintes, ce que l'on peut résumer par *Branch & Propagate & Bound* (B&P&B) : à chaque noeud de l'arbre de recherche, les informations liées aux contraintes, ou du moins une partie, peuvent être exploitées pour réduire l'espace de recherche. La Programmation Par Contraintes (PPC) est l'un des outils les plus populaires qui permettent d'exploiter ces contraintes afin de réduire ainsi l'espace de recherche.

La PPC offre des langages de haut niveau pour la modélisation des COPs en termes de contraintes et elle intègre toute une gamme d'algorithmes dédiés

à la propagation de contraintes et à la recherche de solutions. Par conséquent, résoudre des COPs avec la PPC ne nécessite pas beaucoup de travail de programmation. La PPC est généralement très efficace lorsque les contraintes sont suffisamment nombreuses et fortes pour que leur propagation réduise l'espace de recherche à une taille raisonnable. Toutefois, sur certaines instances, elle peut ne pas trouver de solutions de bonne qualité en un temps acceptable à cause de l'explosion combinatoire.

Approches incomplètes Les métaheuristiques sont des méthodes d'optimisation qui contournent ce problème d'explosion combinatoire en explorant l'espace de recherche de façon incomplète. L'un des points clés des métaheuristiques est leur capacité à équilibrer la diversification et l'intensification de la recherche. D'un côté, la diversification vise à échantillonner l'espace de recherche de façon large de sorte que la recherche ne soit pas concentrée sur une zone de l'espace de recherche particulière. De l'autre côté, l'intensification a pour objectif d'augmenter l'effort de recherche dans les zones les plus prometteuses, aux alentours des bonnes solutions, pour ainsi atteindre la solution optimale.

Les métaheuristiques obtiennent généralement des solutions de très bonne qualité avec des temps de calcul polynomiaux. Cependant, elles présentent des inconvénients que nous pouvons résumer par les trois points suivants (1) elles ne garantissent pas l'optimalité des solutions trouvées, car elles n'assurent pas le parcours entier de l'espace de recherche (2) quand elles ne sont pas intégrées dans un langage de haut niveau, leur utilisation pour résoudre un nouveau COP peut nécessiter un important travail de programmation (3) leurs performances sont étroitement liées aux valeurs de leurs paramètres qui généralement ne sont pas évidentes à trouver.

Approches complètes versus méta-heuristiques En résumé, les approches complètes et les méta-heuristiques sont complémentaires. En effet, les premières offrent la garantie de l'optimalité et sont facilement intégrables dans des langages de PPC au détriment du temps de calcul tandis que les deuxièmes offrent le moyen de contourner l'explosion combinatoire au détriment de la complétude.

Ce dernier constat a donné naissance à l'idée de développer des algorithmes dans lesquels les avantages des deux approches sont combinés. Plusieurs travaux qui proposent de telles combinaisons existent dans l'état de l'art. Cependant, la plupart d'entre eux proposent des algorithmes qui traitent des problèmes spécifiques ou qui proposent des combinaisons entre les deux approches que nous pouvons qualifier de combinaisons faibles, car si, dans ces algorithmes, les deux approches collaborent pour résoudre un problème, leurs composantes élémentaires ne sont pas réellement embarquées les unes dans les autres.

Motivations et contributions Notre ambition dans cette thèse est d'aller un peu plus loin en matière de collaboration entre métaheuristiques et algorithmes com-

plets. En effet, le coeur du sujet de cette thèse est de proposer des algorithmes hybrides génériques où les deux approches sont fortement combinées pour la résolution de problèmes d'optimisation sous contraintes modélisés dans un langage de programmation par contraintes.

Pour le choix de la métaheuristique, nous nous sommes particulièrement intéressés à la métaheuristique d'optimisation par colonies de fourmis (en anglais :"Ant Colony Optimization" (ACO)). Ce choix est justifié par la nature constructive de ACO qui est très similaire aux principes de base de construction des solutions dans les approches complètes. En effet, résoudre un problème avec un algorithme de type *B&P&B* consiste généralement à construire des solutions étape par étape. A chaque étape, à partir d'une solution partielle (éventuellement vide), les composants de solutions sont rajoutés un par un jusqu'à ce qu'une solution soit trouvée ou une incohérence soit détectée. Ce mécanisme de construction incrémentale de solutions constitue le coeur même des algorithmes ACO.

Pour le choix de l'algorithme complet nous avons utilisé IBM ILOG CP Optimizer qui est un produit commercialisé par IBM et qui propose une API de programmation et d'optimisation par contraintes.

Dans le cadre de cette thèse, nous avons développé trois contributions, à savoir : (1) Intégration des algorithmes de type ACO dans un langage de programmation par contraintes pour la résolution de problèmes de satisfaction de contraintes ; (2) Proposition d'un algorithme hybride et générique où ACO est couplé à une approche complète pour résoudre des problèmes d'optimisation combinatoires (3) Proposition d'une stratégie capable d'adapter dynamiquement les paramètres de ACO.

Ci-après, nous donnons un résumé de chacune de ces trois contributions.

Intégration d'ACO à un langage de PPC pour résoudre des CSP :
Nous avons montré comment intégrer ACO dans un langage de programmation par contraintes pour résoudre les problèmes de satisfaction de contraintes : le problème à résoudre est décrit en termes de contraintes dans le langage d'IBM ILOG Solver et il est résolu de façon générique par un algorithme ACO intégré au langage et remplaçant l procédure de recherche "Branch&Propagate". Cet algorithme ACO utilise les procédures de propagation et de vérification des contraintes prédéfinies dans la bibliothèque d'ILOG Solver.

Nous avons validé notre approche sur le problème d'ordonnancement de voitures. Ce dernier, est un problème de référence utilisé au sein de la communauté PPC pour tester les performances et l'efficacité de leurs algorithmes. Cette première contribution à été publiée dans [KAS08a, KAS07, KAS08b, KAS08c].

Couplage d'ACO et de la PPC pour résoudre des COPs :
Nous avons montré comment coupler ACO avec une procédure de recherche complète pour résoudre des COPs : le COP est décrit à l'aide du langage de modélisation par contraintes et est résolu par un algorithme générique intégré au langage.

Cet algorithme générique couple ACO et recherche complète afin de bénéficier des avantages de chacune des deux approches : - ACO est utilisé dans une première phase pour échantillonner l'espace de recherche et identifier les zones prometteuses ; pendant cette première phase, la recherche B&P est utilisée pour fournir à ACO des solutions cohérentes -La recherche complète de type B&P&B est utilisée dans une deuxième phase pour rechercher une solution optimale ; ACO est utilisé pendant cette deuxième phase pour guider la recherche complète, comme heuristique de choix de variables et/ou de valeurs.

Nous avons démontré l'efficacité de cet algorithme hybride sur des problèmes de sac-à-dos multidimensionnels, d'affectation quadratique et cliques maximum. La plupart de ces résultats sont publiés dans [KAS10b, KAS10a].

Adaptation dynamique et automatique des paramètres d'ACO :
Le niveau d'intensification et de diversification de la recherche réalisé par un algorithme ACO est donné par les valeurs de ses nombreux paramètres. Ces paramètres donc ont une forte influence sur le processus de résolution. Nous avons proposé une nouvelle stratégie adaptative qui permet à un algorithme ACO de régler lui-même ses propres paramètres pendant la résolution d'un problème de sorte qu'ils soient adaptés à la structure de l'instance traitée.

Nous avons validé cette stratégie sur un algorithme ACO pour la résolution des problèmes de satisfaction de contraintes. Ces travaux ont été publiés dans [KAS09a, KAS09b]

Organisation du mémoire Ce mémoire de thèse est organisé en deux grandes parties. La première partie est consacrée à l'introduction des notions de base des problèmes d'optimisation combinatoires sous contraintes et des méthodes de résolution les plus connues. La deuxième partie présente nos contributions.

La première partie est divisée en quatre chapitres. Le premier chapitre donne les définitions élémentaires des problèmes d'optimisation combinatoires et des définitions liées à la modélisation en programmation par contraintes. Le deuxième chapitre introduit les méthodes exactes de résolution les plus connues et introduit des notions de cohérence locale utilisées généralement en programmation par contraintes. Le troisième chapitre présente quelques unes des métaheuristiques les plus utilisées et plus particulièrement l'optimisation par colonies de fourmis. Le quatrième chapitre montre comment les contraintes sont gérées dans les métaheuristiques.

La deuxième partie quant à elle, est divisée en trois chapitres. Chacun d'eux développe une contribution parmi les trois contributions décrites brièvement ci-dessus. Ainsi, dans le cinquième chapitre, nous décrivons l'intégration d'ACO à un langage de PPC pour résoudre des CSP. Dans le sixième chapitre, nous introduisons une nouvelle approche couplant ACO et recherche complète pour la résolution des problèmes d'optimisation combinatoires. Dans le septième chapitre, nous introduisons une nouvelle stratégie qui permet d'adapter les paramètres de

ACO de manière dynamique et adaptative. Enfin, nous finissons par une conclusion dans laquelle nous résumons et donnons les perspectives des travaux effectués durant cette thèse.

Première partie

Programmation par contraintes et optimisation combinatoire sous contraintes

Introduction à la première partie

La Programmation Par Contraintes (PPC) est un paradigme de programmation qui vise à offrir : des langages de modélisation déclaratifs de haut niveau et des algorithmes de résolution efficaces et génériques. Résoudre un problème dans un environnement de programmation par contraintes revient à passer par deux étapes distinctes. La première étape consiste à déclarer et définir toutes les variables (les inconnues) du problème à résoudre et définir les contraintes qui gouvernent la manière dont ces variables peuvent être affectées les unes vis-à-vis des autres. Cette première étape est appelée l'étape de modélisation. La deuxième étape consiste à chercher la solution du problème modélisé lors de la première étape. Cette recherche se fait soit en lançant la procédure de recherche prédéfinie dans l'environnement ou en définissant une nouvelle procédure de recherche en s'appuyant sur les différents algorithmes proposés par l'environnement lui même.

Du fait de sa simplicité et son efficacité, la PPC a connu lors de ces vingt dernières années un grand succès que ce soit dans le monde académique ou industriel. D'une part, depuis son émergence, les chercheurs n'ont pas cessé d'améliorer le niveau d'expressivité des langages de modélisation et l'efficacité des algorithmes de résolutions. D'autre part, les industriels ont trouvé dans la PPC un moyen simple pour exprimer et résoudre des problèmes réels et pratiques qui leurs permet d'améliorer leurs services et d'augmenter leur productivité. Comme exemples de ces applications pratiques nous pouvons citer : les emplois du temps des infirmières dans les hôpitaux ; la planification des trains ; l'ordonnancement des tâches de fabrication dans les usines ; la planification du trafic aérien.

Dans cette première partie, nous allons donner les principes de base des différents concepts utilisés lors de la deuxième partie. Dans le premier chapitre, nous allons décrire les problèmes de satisfaction de contraintes (*CSP*) et les problèmes d'optimisation sous contraintes (*COP*). Dans le deuxième chapitre, nous allons parler des méthodes de résolution exactes traditionnellement intégrées dans la PPC. Dans le troisième chapitre, nous donnons quelques méthodes de résolution heuristiques. Enfin, dans le quatrième chapitre, nous allons donner quelques techniques de gestion des contraintes dans les métaheuristiques.

1 Problèmes d'optimisation combinatoire sous contraintes

Sommaire

Dans ce chapitre, nous donnons quelques définitions et notations qui vont constituer le socle de la suite de ce rapport. Nous donnons un bref rappel sur la modélisation en Programmation Par Contraintes des problèmes de satisfaction de contraintes et les problèmes d'optimisation combinatoire sous contraintes. Dans les dernières sections, nous allons donner la définition et la représentation de chacun des problèmes que nous avons utilisés dans les travaux menés dans le cadre de cette thèse. De plus, nous fournirrons la définition et la représentation du problème de voyageur de commerce qui va nous servir d'exemple à maintes reprises lorsque nous introduisons de nouveaux concepts.

Sans perte de généralités, un problème d'optimisation sous contraintes contient deux aspects. L'aspect contraintes et l'aspect optimisation. L'aspect contraintes sert à spécifier la forme que devrait avoir un objet solution du problème. L'aspect optimisation sert à classifier les objets vérifiant cette forme et à sélectionner le(s) meilleure(s).

1.1 Définition d'un problème de satisfaction de contraintes

Un problème de satisfaction de contraintes (Constraint Satisfaction Problem CSP) est un problème où l'on cherche à affecter un ensemble de variables par des valeurs appartenant à leurs domaines respectifs tout en satisfaisant un ensemble de contraintes. Plus formellement, un CSP peut être défini par un triplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- $\mathcal{X}$: est un ensemble fini de n variables $\{x_1, x_2, ..., x_n\}$.
- $\mathcal{D}$: associe à chaque variable $x_i \in \mathcal{X}$ un ensemble fini $\mathcal{D}(x_i)$ de valeurs possibles pour x_i.
- $\mathcal{C}$: est un ensemble fini de contraintes $\{c_1, c_2, ..., c_m\}$. Chaque contrainte c_i définit une relation sur un sous-ensemble de variables dénoté $\mathcal{V}ar(c_i)$. Chaque contrainte c_i définit un sous ensemble du produit cartésien $\prod_{x_i \in \mathcal{V}ar(c_i)} \mathcal{D}(x_i)$ qui donne les valeurs que les variables de $\mathcal{V}ar(c_i)$ peuvent prendre simultanément.

1.1.1 Définition d'une contrainte

Tout comme une relation mathématique, une contrainte peut être définie en extension ou en intention.

Définition d'une contrainte en intention

Definir une contrainte c_i en intention revient à exprimer avec des propriétés mathématiques les tuples autorisés par la contrainte c_i.

Exemple 1 Soit $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP tel que :

- $\mathcal{X} = \{x_1, x_2, x_3\}$.
- $\mathcal{D} = \{\mathcal{D}(x_1) = \{1, 2, 3\}, \mathcal{D}(x_2) = \{1, 2\}, \mathcal{D}(x_3) = \{2, 3\}\}$
- $\mathcal{C} = \{c_1 \equiv (x_1 + x_2 = x_3),\ c_2 \equiv (|x_1 - x_3| \neq 1)\}$

Une illustration graphique de cet exemple est donnée par la figure 1.1.

Définition des contraintes de l'exemple 1 en intention Chacune des contraintes de l'exemple 1 est donnée par une formule mathématique ($c_1 \equiv (x_1 + x_2 = x_3)$ et $c_2 \equiv (|x_1 - x_3| \neq 1)$). Ces deux contraintes c_1 et c_2 sont définit en intention.

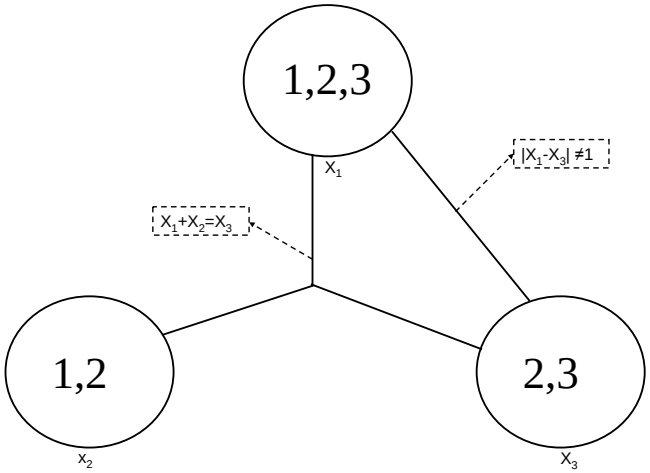

FIGURE 1.1 – *Représentation graphique du CSP décrit dans l'exemple 1*

Définition d'une contrainte en extension

Définir une contrainte c_i en extension revient à donner tous les tuples $\prod_{x_j \in \mathcal{V}ar(c_i)} \mathcal{D}(x_j)$ autorisés par cette contrainte (i.e. les combinaisons qui vérifient la contrainte c_i). L'ensemble de ces tuples peut être nommé par $supports(c_i)$. De façon duale, la contrainte c_i peut être définie par l'ensemble des tuples qui ne sont pas autorisées par la contrainte c_i. Dans ce cas-là, l'ensemble de ces tuples, peut être nommé par $conflicts(c_i)$.

Définition des contraintes de l'exemple 1 en extension Le CSP de l'exemple 1 contient deux contraintes, à savoir $c_1 \equiv (x_1 + x_2 = x_3)$ et $c_2 \equiv (|x_1 - x_3| \neq 1)$. Nous avons donc : $supports(c_1) \equiv \{(1,1,2),(1,2,3),(2,1,3)\}$ et $supports(c_2) \equiv \{(1,3)(2,2),(3,3)\}$.

Notons que pour cet exemple nous avons choisi de donner les supports et non pas les conflits des contraintes.

Remarque : Lorsqu'on définit une contrainte en extension, il faut faire attention à l'espace que peuvent occuper les tuples autorisés par cette contrainte. En effet, la vérification de la satisfiabilité d'une contrainte c_i définie en extension revient à accéder à l'ensemble $supports(c_i)$ (ou $conflicts(c_i)$). En revanche, stocker dans la mémoire tous les tuples présents dans $support(c_i)$ (ou $conflicts(c_i)$) peut demander beaucoup de place mémoire. Par contre, définir une contrainte en intention en utilisant des propriétés mathématiques ne demande pas beaucoup de place mémoire, mais demande l'écriture d'un algorithme qui soit capable d'exploiter la sémantique véhiculée par ces définitions.

1.1.2 Arité d'une contrainte

L'arité d'une contrainte est le nombre de variables sur lesquelles elle porte (c.à-d :. $|\mathcal{V}ar(c)|$). Ainsi, une contrainte binaire est une contrainte qui porte sur deux variables. Une contrainte c_i est dite globale, si elle porte sur un ensemble de variables.

Comme nous allons le montrer dans l'exemple 2, les contraintes globales permettent de décrire les problèmes de manière concise et permettent également de les résoudre plus efficacement.

Remarque Un *CSP* binaire est un *CSP* qui ne contient que des contraintes binaires.

Exemple 2 Considérons le *CSP* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ tel que $\mathcal{X} = \{x_1, x_2, x_3\}$; $\mathcal{D}(x_1) = \mathcal{D}(x_2) = \mathcal{D}(x_3) = \{1, 2\}$ et $\mathcal{C} = \{x_1 \neq x_2,\ x_1 \neq x_3,\ x_2 \neq x_3\}$.

Dans cet exemple, les trois contraintes de différences sont binaires. Cependant, elles peuvent être remplacées par une seule contrainte globale $allDiff$ spécifiant que les trois variables x_1, x_2 *et* x_3 doivent être affectées avec trois valeurs différentes. L'utilisation de cette dernière contrainte nous évite d'écrire trois contraintes binaires.

Contrainte globale vs contraintes binaires Le niveau d'expressivité des contraintes globales n'est pas leur unique avantage sur les contraintes binaires. En effet, une contrainte globale peut aider à résoudre un problème plus efficacement qu'un ensemble de contraintes binaires dans la mesure où elle véhicule des informations sur ce problème. Dans l'exemple 2, le fait d'utiliser la contrainte $allDiff$ permet d'affirmer immédiatement que le problème n'admet pas de solution, car $|\bigcup_{x_i \in \mathcal{X}} \mathcal{D}(x_i)| < |\mathcal{X}|$. Cette propriété est exploitable lorsque nous utilisons la contrainte $allDiff$, mais elle ne peut pas être exploitée en utilisant chacune des trois contraintes binaires séparément $\{\neq (x_1, x_2), \neq (x_1, x_3), \neq (x_2, x_3)\}$.

1.2 Solution d'un problème de satisfaction de contraintes

Pour un *CSP* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ on définit les notions suivantes :

- Une instanciation d'une variable x_i par une valeur $v_j \in \mathcal{D}(x_i)$ est définie par le couple $< x_i, v_j >$.
- Une affectation $\mathcal{A} = \{< x_1, v_1 >, < x_2, v_2 >, ..., < x_k, v_k >\}$ est un ensemble d'instanciations (de couples) correspondant à l'instanciation de la variable x_1 par la valeur v_1, la variable x_2 par la valeur v_2,..., et la variable x_k par la valeur v_k. Dans une affectation $\mathcal{A}$, une variable apparait dans au plus un seul couple. On dénote par $\mathcal{V}ar(\mathcal{A})$ l'ensemble des variables instanciées dans $\mathcal{A}$.
- Une affectation $\mathcal{A}$ est partielle si $|\mathcal{A}| < |\mathcal{X}|$, sinon, elle est totale.

- Une affectation $\mathcal{A}$ satisfait (resp. viole) une contrainte $c_i \in \mathcal{C}$ telle que $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(\mathcal{A})$ si le tuple des valeurs associées aux variables $\mathcal{V}ar(c_i)$ dans $\mathcal{A}$ est autorisé (resp. non autorisé) par la contrainte c.
- Une affectation $\mathcal{A}$ est incohérente si elle viole au moins une des contraintes de $\mathcal{C}$, sinon elle est cohérente.
- Une solution d'un *CSP* est une affectation totale et cohérente.

Exemple 3 Sur le *CSP* de la figure 1.1 :
- $\mathcal{A}_1 = \{< x_1, 1 >, < x_3, 2 >\}$ est une affectation partielle et incohérente (elle viole la contrainte $c_2 \equiv (|x_1 - x_3| \neq 1)$).
- $\mathcal{A}_2 = \{< x_1, 1 >, < x_3, 3 >\}$ est une affectation partielle et cohérente.
- $\mathcal{A}_3 = \{< x_1, 1 >, < x_2, 1 >, < x_3, 3 >\}$ est une affectation totale et incohérente (elle viole la contrainte $c_1 \equiv (x_1 + x_2 = x_3)$).
- $\mathcal{A}_4 = \{< x_1, 1 >, < x_2, 2 >, < x_3, 3 >\}$ est une affectation totale et cohérente. Donc, $\mathcal{A}_4$ est une solution.

L'espace de recherche L'espace de recherche d'un *CSP* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est défini par un ensemble fini Ω qui est constitué de toutes les affectations totales. L'ensemble des contraintes $\mathcal{C}$ divise Ω en une partition de deux ensembles Ω^+ et Ω^-. L'ensemble Ω^+ contient toutes les solutions de $\mathcal{P}$ et l'ensemble Ω^- contient toutes les affectations totales et incohérentes de $\mathcal{P}$. Résoudre $\mathcal{P}$ revient à trouver un élément de Ω^+.

Définition des *MaxCSPs* Dans la pratique, les *CSPs* peuvent être sur-contraints de sorte qu'ils n'admettent pas de solution. Dans ce cas, on peut chercher une affectation totale maximisant le nombre de contraintes satisfaites. Cette catégorie de problème est appelée *MaxCSP*.

1.3 Définition d'un problème d'optimisation combinatoire sous contraintes

Un problème d'optimisation combinatoire (en anglais : Combinatorial Optimization Problem (*COP*)) sous contraintes est un problème de satisfaction de contraintes *CSP* augmenté d'une fonction à optimiser qu'on appelle généralement la fonction objectif ou encore la fonction économique.

Un *COP* donc, est défini par un quadruplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{F})$ tel que
- $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est un problème de satisfaction de contraintes.
- $\mathcal{F} : \mathcal{D}(x_1) \times \ldots \times \mathcal{D}(x_n) \longrightarrow \mathbb{R}$ est la fonction objectif à optimiser.

Résoudre un *COP* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{F})$ à l'optimal, implique de trouver une solution $\mathcal{S}^* \in \Omega^+$ telle que $\forall \mathcal{S}' \in \Omega^+, \mathcal{F}(\mathcal{S}^*) \geq \mathcal{F}(\mathcal{S}')$ si $\mathcal{F}$ est une fonction à maximiser ou une solution $\mathcal{S}^* \in \Omega^+$ telle que $\forall \mathcal{S}' \in \Omega^+,\ \mathcal{F}(\mathcal{S}^*) \leq \mathcal{F}(\mathcal{S}')$ si $\mathcal{F}$ est une fonction à minimiser.

Remarques :
- un CSP peut être vu comme une forme particulière d'un COP dont la fonction objectif est une constante.
- dans la suite nous désignons par COP un COP sous contraintes.
- Egalement, nous désignons par le terme affectation une affectation totale et lorsque nous voulons parler d'une affectation partielle nous le mentionnons de façon explicite. Enfin, le terme solution est réservé pour désigner une affectation complète et cohérente.
- Dans le cas des $MaxCSPs$, $\mathcal{F}$ est la fonction qui évalue le nombre de contraintes satisfaites.

Complexité Dans le cas général, résoudre un COP à l'optimal nécessite l'examen d'un nombre de combinaisons exponentiel en fonction de la taille du problème traité. En effet, la plupart de ces problèmes sont NP-difficiles. Par conséquent, à moins que $P = NP$, ils ne peuvent pas être résolus de façon exacte en un temps polynomial.

Malgré cette observation, aussi dure soit-elle, nous ne devrions pas nous décourager, car la complexité exponentielle de ces problèmes est mesurée dans le pire des cas (i.e. : cette complexité est donnée par rapport à l'instance la plus difficile). Dans la réalité, certaines instances peuvent être résolues beaucoup plus rapidement que d'autres. Donc, nous pouvons tout de même écrire des algorithmes pour résoudre ces problèmes tout en gardant à l'esprit que nos programmes peuvent ne pas résoudre certaines instances en des temps acceptables.

1.4 Exemples de problèmes d'optimisation combinatoire

Dans les sections suivantes, nous allons donner la définition et la modélisation des problèmes que nous allons utiliser dans la suite de cette thèse.

1.4.1 Le problème du voyageur de commerce

Le Problème du Voyageur de Commerce (TSP) consiste à visiter un ensemble de villes en passant une et une seule fois par chacune d'elles et revenir à la ville de départ tout en minimisant la distance totale parcourue.

Une instance d'un TSP est représentée par un graphe $\mathcal{G}(N, E)$ et une fonction de coût $d : E \rightarrow \mathbb{R}$ où N est l'ensemble des noeuds de $\mathcal{G}$ (chaque noeud est associé à une ville), $E \subseteq (N \times N)$ est l'ensemble des arêtes reliant les noeuds de N deux à deux et $d(i, j)$ est la distance entre les deux noeuds i et j. Le coût d'un cycle du graphe $\mathcal{G}$ est la somme des coûts des arêtes qui le composent. Résoudre un TSP consiste à trouver un cycle hamiltonien (passant par chaque sommet une et une seule fois) de coût minimal. Sans perte de généralité, nous supposons dans la suite que la fonction coût est définie pour toute paire de sommets, même s'ils

ne sont pas reliés par une arête (ce coût peut être défini à une valeur très grande, supérieure par exemple à la somme des coûts de toutes les arêtes).

Un modèle COP d'un TSP à n villes est :

- $\mathcal{X}$ un ensemble de variable $x_1, x_2, ..., x_n$ où chaque variable x_i représente une étape du voyageur. Chaque variable donne la ville que le voyageur a visité lors de l'étape qu'elle représente.
- $\mathcal{D}(x_i) = \{1, ..., n\}$, $\forall x_i \in \mathcal{X}$ de sorte que $x_i = j$ si à l'étape i le voyageur visite la ville j.
- $\mathcal{C} = \{AllDiff(x_1, x_2, ..., x_n)\}$ est l'ensemble des contraintes assurant qu'à chaque étape, le voyageur viste une ville différentes.
- la fonction objectif à minimiser est $F = \sum_{i=1}^{n-1} d_{x_i,x_{i+1}} + d_{x_1,x_n}$.

1.4.2 Le problème d'ordonnancement de voitures

Ce problème consiste à ordonner des voitures le long d'une chaîne de montage pour installer des options sur les voitures (par exemple, un toit ouvrant ou l'air conditionné). Chaque option est installée par une station différente conçue pour traiter au plus un certain pourcentage de voitures passant le long de la chaîne. Par conséquent, les voitures demandant une même option doivent être espacées de telle sorte que la capacité de chaque station ne soit jamais dépassée. Plus formellement, une instance de ce problème peut être définie par un tuple (V, O, p, q, r) tel que :

- $V = \{v_1, v_2, ..., v_n\}$: l'ensemble des voitures à produire ;
- $O = \{o_1, o_2, ..., o_m\}$: l'ensemble des différentes options possibles ;
- $p : O \rightarrow N$ et $q : O \rightarrow N$ sont deux fonctions qui définissent la contrainte de capacité associée à chaque option $o_i \in O$. Pour chaque séquence de $q(o_i)$ voitures consécutives sur la ligne d'assemblage, au plus $p(o_i)$ peuvent demander l'option o_i ;
- $r : V \times O \rightarrow \{0, 1\}$ est la fonction qui définit les options requises par chaque voiture. Pour chaque $v_i \in V$ et pour chaque option $o_j \in O$, si o_j doit être installée sur v_i, alors $r(v_i, o_j) = 1$ sinon $r(v_i, o_j) = 0$.

Il s'agit alors de chercher un ordonnancement des voitures qui satisfait les contraintes de capacité. Ce problème a été montré NP-difficile par Kis [Kis04]. Une variante de ce problème, faisant intervenir, en plus des contraintes de capacité liées aux options, des contraintes liées à la couleur des voitures a fait l'objet du challenge ROADEF en 2005 [SCNA08].

Avant de donner le modèle COP du problème d'ordonnancement de voitures, nous allons d'abord introduire la notion de classe de voitures qui permet de réduire la combinatoire. En effet, différentes voitures de V peuvent demander le même ensemble d'options. Ces voitures sont interchangeables dans une séquence et donc symétriques.

Une classe de voiture est définie par l'ensemble des options qu'elle demande. Soit $cl(v_i) = \{o_j \in O | r(v_i, o_j) = 1\}$. Nous définissons l'ensemble des classes de

voitures possibles par $CL = \{cl(v_i)|v_i \in V\}$. Chaque classe $cl_i \in CL$ représente un sous ensemble de voitures de V qui demandent les mêmes options.

Le modèle COP du problème d'ordonnancement de voitures peut être défini comme suit :

Les variables du CSP sont de deux types : $\mathcal{X} = \mathcal{X}_c \cup \mathcal{X}_o$

- $\mathcal{X}_c = \{x_i|i \in 1..n\}$ associe à chaque position i dans la chaîne de montage une variable x_i représentant la classe cl de voiture qui devrait être placée à la position i ; le domaine de ces variables est défini par l'ensemble des classes de voitures CL ($\mathcal{D}(x_i) = CL \ \ \forall x_i \in \mathcal{X}$).
- $\mathcal{X}_o = \{o_i^j|i \in 1..n, \ j \in O\}$ associe à chaque variable $x_i \in \mathcal{X}$ et à chaque option $o_j \in O$ une variable o_i^j indiquant si la voiture à la position i demande l'option j ; le domaine de ces variables est l'ensemble $\{0, 1\}$ de sorte que $o_i^j = 1$ si l'option j doit être installée sur la voiture qui sera placée à la position i, et 0 sinon.

Les contraintes sont de trois types :

- une première série de contraintes fait le lien entre les variables de $\mathcal{X}_c$ et les variables de $\mathcal{X}_o$ spécifiant que $o_i^j = 1$ si et seulement si l'option j doit être installée sur la voiture x_i ; $\forall i \in 1..n, \ \forall j \in O, o_i^j = r(x_i, o_j)$
- une deuxième série de contraintes concerne la capacité des stations et spécifie que, pour chaque option j et chaque sous-séquence de q_j voitures consécutives, la somme des variables o_i^j correspondantes doit être inférieure ou égale à p_j ;
- une troisième série de contraintes spécifie pour chaque classe de voitures le nombre de voitures de cette classe devant être séquencées. $\forall cl_k \in CL, (\sum_{i=1}^{n} x_i)$

1.4.3 Le problème de sac à dos multidimensionnel

Ce problème consiste à sélectionner un sous-ensemble d'objets satisfaisant un ensemble de contraintes linéaires de capacité et maximisant la somme des profits des objets sélectionnés. Ci-dessous, nous donnons le modèle COP de ce problème :

- $\mathcal{X}$ un ensemble de variable $x_1, x_2, ..., x_n$ où chaque variable x_i est associée à un objet o_i ;
- $\mathcal{D}(x_i) = \{0, 1\}, \forall x_i \in X$ de sorte que $x_i = 0$ si l'objet o_i n'est pas sélectionné, et 1 sinon ;
- $\mathcal{C}$ est un ensemble de m contraintes de capacité tel que chaque contrainte $c_j \in C$ est de la forme $\sum_{i=1}^{n} c_{ij} \cdot x_i \leq r_j$ où c_{ij} est la quantité de ressource j consommée par l'objet i et r_j est la quantité disponible de la ressource j ;
- la fonction objectif à maximiser est $F = \sum_{i=1}^{n} p_i \cdot x_i$ où p_i est le profit de l'objet i.

1.4.4 Le problème de l'affectation quadratique

Ce problème consiste à déterminer l'emplacement d'un ensemble d'entrepôts $\{o_1, o_2, ..., o_n\}$ sur un ensemble d'emplacement $\{1, 2, ..., n\}$ de façon à minimiser les distances et les flux entre eux. Le modèle COP de ce problème peut être défini comme suit :

- $\mathcal{X}$ un ensemble de variable $x_1, x_2, ..., x_n$ où chaque variable x_i est associée à un entrepôt o_i ;
- $\mathcal{D}(x_i) = \{1, \ldots, n\}$, $\forall x_i \in X$ tel que $x_i = j$ si l'entrepôt o_i est positionné à l'emplacement j ;
- $\mathcal{C}$ contient uniquement la contrainte all-different sur toutes les variables, assurant ainsi que chaque objet est positionné à un seul emplacement.
- la fonction objectif à optimiser est $F = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} b_{x_i,x_j}$ où $a_{i,j}$ est la distance entre les emplacements i et j, et b_{x_i,x_j} est le flux entre les entrepôts o_i et o_j.

1.4.5 Le problème de clique maximum

Le problème de clique maximum consiste à sélectionner le plus grand sous-ensemble de sommets d'un graphe $\mathcal{G}(N, E)$ de sorte que les sommets sélectionnés sont deux à deux connectés dans le graphe $\mathcal{G}$. Le modèle COP de ce problème peut être défini comme suit :

- $\mathcal{X}$: un ensemble de variable $x_1, x_2, ..., x_n$ où chaque variable x_i est associée à un sommet i du graphe G
- $\mathcal{D}(x_i) = \{0, 1\}$, $\forall x_i \in \mathcal{X}$ de sorte que $x_i = 0$ si le sommet i n'est pas sélectionné, et 1 sinon ;
- $\mathcal{C}$: un ensemble de contraintes binaires qui associe à chaque paire de sommets $(i, j) \in N^2$ non connecté dans le graphe G une contrainte $c_{ij} = (x_i + x_j < 2)$. Cette contrainte assure que les sommets i et j ne seront pas sélectionnés simultanément dans un même ensemble.
- la fonction objectif à maximiser est $F = \sum_{i=1}^{n} x_i$.

Notons dès maintenant que dans la deuxième partie de cette thèse (chapitre 6), nous avons également traité le problème de l'ensemble stable maximal. Nous n'allons pas donner les détails du modèle PPC de ce problème, car il est très similaire à celui du problème de clique maximum. En effet, Le problème de l'ensemble stable maximal consiste à sélectionner le plus grand sous-ensemble de sommets d'un graphe $\mathcal{G}$ de sorte que les sommets sélectionnés sont deux à deux déconnectés dans le graphe $\mathcal{G}$. Le modèle PPC de ce problème peut être défini exactement comme pour le problème de clique maximum en considérant le graphe inverse du graphe $\mathcal{G}$. Le graphe inverse du graphe G est le graphe $\mathcal{G}'$ qui contient exactement les mêmes sommets du graphe $\mathcal{G}$ sauf que les sommets connectés par une arête dans $\mathcal{G}$ ne le sont plus dans $\mathcal{G}'$ et les sommets non connectés dans $\mathcal{G}$ sont connectés dans $\mathcal{G}'$.

1.5 L'API de modélisation d'IBM CP Optimizer

IBM CP Optimizer est une API qui offre (1) un langage de modélisation de haut niveau (2) des algorithmes génériques de résolution de problèmes de satisfaction de contraintes et d'optimisation combinatoires. Pour résoudre un problème dans cette API, l'utilisateur passe par deux étapes distinctes. La première étape consiste à modéliser le problème en déclarant l'ensemble des variables (les inconnues) du problème et un ensemble de contraintes sur ces variables. La deuxième étape consiste à résoudre le problème modélisé lors de la première étape. Dans cette section, nous nous intéressons à la première étape et nous parlerons de la deuxième étape dans le prochain chapitre.

Le langage de modélisation offert par cette API nous permet de définir le problème à résoudre de manière simple et concise. Ce langage est constitué d'un ensemble de classes prédéfinies. Chacune de ces classes nous permet de définir des objets tels que des variables de décisions entières ou booléennes ; des contraintes sur les variables ; des expressions numériques et des fonctions à optimiser etc. Au fur et à mesure que les éléments du problème sont définis (les variables, les contraintes, les fonctions etc.), ils sont assemblés (ajoutés) dans un unique objet instance de la classe IloModel. C'est ce dernier objet qui constitue le modèle du problème à résoudre.

Dans la figure 1.2, nous donnons un exemple concret d'utilisation de l'API *IBM CP Optimizer* pour modéliser un COP. Pour cela, nous avons choisi de donner le programme C++ en entier qui permet de modéliser le problème de sac-à-dos multidimensionnel.

1.6 Conclusion

Dans ce chapitre, nous avons donné les définitions élémentaires d'un problème de satisfaction de contraintes et d'optimisation combinatoire sous contraintes. Nous avons également abordé la notion de modélisation en programmation par contraintes et nous avons donné la représentation en COP des problèmes que nous allons traiter lors de la deuxième partie de cette thèse. Dans les deux prochains chapitres, nous allons aborder la notion de résolution des COP sous contraintes.

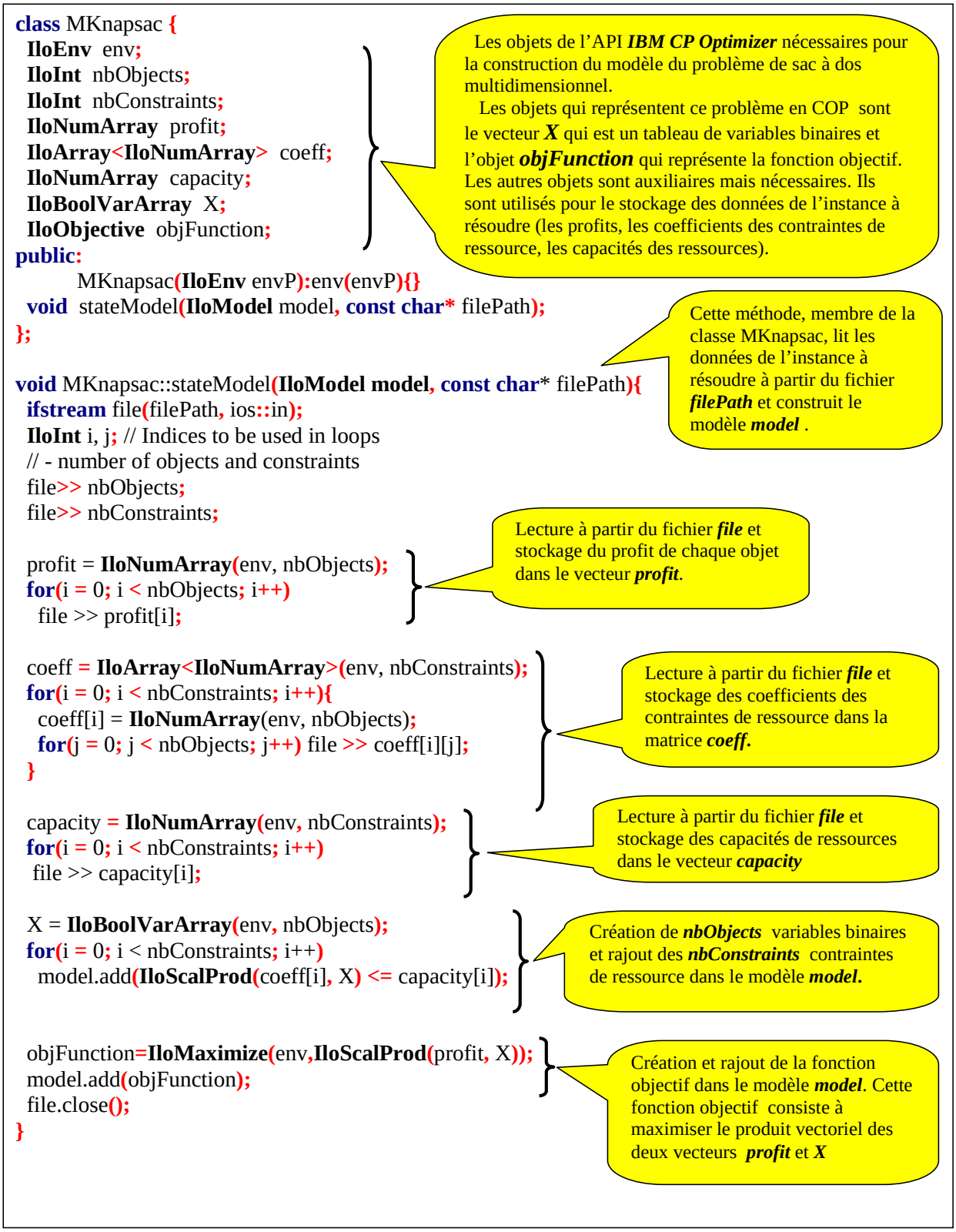

FIGURE 1.2 – *Illustration de l'API d'IBM CP Optimizer sur le MKP*

Méthodes de résolution exactes 2

Sommaire

Ce chapitre introduit les méthodes de résolution exactes. Nous allons donner : les algorithmes de résolution les plus naïfs ; Les techniques de propagation de contraintes qui permettent de réduire l'espace de recherche en maintenant un certain niveau de cohérence locale durant la recherche des solutions ; nous allons présenter quelque uns des algorithmes les plus connus qui utilisent ces notions de propagation de contraintes ; nous parlerons des heuristiques d'ordres de choix de variables et de valeurs ; enfin, nous allons parler brièvement de l'API IBM ILOG CPLEX CP Optimizer.

Comme nous l'avons dit au chapitre 1, la plupart des COP sont *NP*-difficiles [GJ79]. Par conséquent, à moins que $P = NP$, ils ne peuvent pas être résolus de façon exacte en un temps polynomial. De manière générale, pour résoudre un COP, deux types d'approches peuvent être utilisées, à savoir, les approches complètes et les approches heuristiques (ou métaheuristiques). Dans ce chapitre, nous présentons des approches complètes, tandis que dans le chapitre 3, nous parlerons des techniques d'optimisation par métaheuristiques.

Un algorithme basé sur une approche complète explore l'espace de recherche de façon exhaustive et trouve la solution optimale en une limite de temps finie [PS82]. Cependant, la complexité en temps de l'exécution dans le pire des cas est exponentielle en fonction de la taille de l'instance à résoudre.

Généralement, la technique de base des algorithmes basés sur une approche complète est la recherche arborescente : ces algorithmes parcourent un arbre (appelé : " arbre de recherche" ou en anglais "Search Tree" (ST)). Pour un COP, cet arbre peut être organisé comme suit :

1. La racine représente l'affectation vide $\mathcal{A} = \emptyset$;
2. Chaque noeud représente une affectation partielle ou totale.
3. Un noeud fils d'un noeud parent étend l'affectation associée à ce parent avec un couple $< x, v >$ où x est une variable non encore affectée dans l'affectation partielle associée au noeud parent et $v \in \mathcal{D}(x)$. Les différents fils d'un même noeud correspondent aux différentes valeurs pouvant être affectées à x.
4. Les feuilles de cet arbre représentent soit des affectations partielles ne pouvant être étendues sans violer de contraintes, soit des affectations totales cohérentes.

Parcourir entièrement cet arbre garantit la complétude de la recherche. Cependant, un tel parcours demande un temps d'exécution exponentiel.

2.1 L'algorithme retour-arrière chronologique

L'algorithme retour-arrière Chronologique (*BTC*) construit l'arbre de recherche en profondeur d'abord. A chaque fois qu'une affectation partielle $\mathcal{A}_p$ est étendue par un couple $< x, v >$, *BTC* vérifie la cohérence de $\mathcal{A}_p \cup < x, v >$ par rapport à chaque contrainte $c \in \mathcal{C}$ non encore satisfaite par $\mathcal{A}_p$ pour laquelle $\mathcal{V}ar(c) \subseteq (\mathcal{V}ar(\mathcal{A}_p) \cup \{< x, v >\})$. Si toutes les contraintes sont satisfaites, il continue l'exploration en profondeur à partir de ce noeud, sinon, il fait un retour arrière jusqu'au premier noeud ancêtre pour lequel certains fils n'ont pas encore été explorés.

L'un des deux problèmes majeurs de l'algorithme *BTC* est la manière avec laquelle il fait marche arrière dans le cas où la vérification d'une contrainte s'avère négative (pas satisfaite). Ce mode de retour arrière au premier point de choix

peut s'avérer très inefficace, car si la raison de l'incohérence dépend d'une variable affectée bien avant dans l'arbre, alors *BTC* risque de faire des reoutr-arrière ("backtrack") pour rien.

Pour palier à ce problème, une solution est apportée par l'algorithme Conflict-directed BackJumping (*CBJ*) [Pro93]. En effet, lorsque une incohérence est détectée, *CBJ* fait un retour-arrière sur la variable qui est la première variable rencontrée en parcourant les variables affectées dans l'ordre inverse de leurs affectation et qui rentre en conflit avec la dernière variable affectée. Ce mécanisme permet d'éviter de backtraker inutilement sur les variables qui ne sont pas à l'origine de l'incohérence détectée.

L'autre problème majeur de l'algorithme *BTC* est le moment choisi pour vérifier la cohérence d'une affectation. En effet, *BTC* ne vérifie pas la cohérence d'une affectation par rapport à une contrainte que lorsque toutes les variables impliquées dans cette contrainte sont affectées. Ceci peut nettement être amélioré en utilisant les techniques de cohérence locale. Dans la suite de ce chapitre nous allons rappeler certaines de ces techniques.

2.2 Cohérences locales

Pour réduire la taille de l'arbre de recherche développé par *BTC*, des techniques de vérification de cohérences locales d'un *CSP* (ou d'un *COP*) ont été développées. Ces techniques permettent de réduire la taille des domaines des variables en enlevant certaines valeurs qui ne peuvent pas figurer dans une solution. Ainsi, la recherche de solutions est accélérée, car les sous-arbres de recherche qui ont comme racine les valeurs enlevées ne sont pas développés. Le processus de réduction des domaines des variables est appelé le processus de filtrage.

En résumé, l'objectif des techniques de filtrage n'est pas de trouver la solution d'un problème mais d'éliminer les régions de l'espace de recherche qui ne contiennent pas de solutions. Après filtrage, le *CSP* résultant est équivalent au *CSP* du départ (avant le filtrage) dans le sens où ils admettent le même ensemble de solutions. Il est à noter que la cohérence locale d'un *CSP* ne garantit pas que ce *CSP* soit cohérent de façon globale.

La puissance d'un l'algorithme de filtrage dépend du niveau de cohérence locale qu'il vérifie. Plusieurs algorithmes de filtrage ont été proposés dans la littérature. Certains filtrent plus de valeurs que d'autres, mais ils sont également plus couteux en temps de calcul. Un algorithme de filtrage judicieusement choisit doit être celui qui offre un bon compromis entre le nombre de valeurs qu'il filtre et le coût en temps induit par ce filtrage.

Dans les paragraphes suivants, nous allons faire quelques rappels sur quelques techniques de filtrages.

2.2.1 La cohérence d'arc

La cohérence d'arc (en anglais arc consistency (AC)) est basée sur la notion de support et a été initialement définie pour les CSP binaires.

Définition de la relation de support

Soit un CSP binaire $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. La valeur b de la variable $y \in \mathcal{X}$ est un support de la valeur a de la variable $x \in \mathcal{X}$ si l'affectation $\{< x, a >, < y, b >\}$ est cohérente.

Définition de la cohérence d'arc (AC)

Un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ vérifie la propriété de la cohérence d'arc si :

$\forall (x, y) \in \mathcal{X}^2, \forall a \in \mathcal{D}(x), \exists b \in \mathcal{D}(y)$, tel que b est un support de a.

Autrement dit, si un CSP binaire vérifie la propriété AC, alors toute paire de variables peuvent être affectées simultanément sans violer aucune contrainte.

2.2.2 Algorithmes de filtrage assurant la cohérence d'arcs

Un algorithme de filtrage par AC enlève du domaine de chaque variable x chaque valeur v n'ayant pas de support dans le domaine d'une autre variable de sorte qu'après le filtrage, le CSP soit cohérent d'arc. Plusieurs algorithmes de filtrage de la propriété AC ont été développés. $AC3$ [Mac77, McG79] est l'un des algorithmes génériques de filtrage les plus connus et à partir duquel plusieurs variantes ont été développées. Notons que $AC3$ est en $\mathcal{O}(n^2d^3)$ et que $AC3$ est une version simplifiée et plus générale de l'algorithme $AC2$ qui lui même est une version améliorante de $AC1$ ($AC1$ est en $\mathcal{O}(n^3d^3)$).

Deux variantes de $AC3$ existent, la variante basée sur les contraintes du problème et celle basée sur les variables du problème. L'idée est la même dans les deux variantes. La variante orientée-contraintes prend en entrée l'ensemble des contraintes et à chaque fois qu'un domaine d'une variable est changé elle reconsidère toutes les contraintes dans lesquelles cette variable apparait. La variante orientée-variables prend en entrée l'ensemble des variables et elle vérifie pour chaque valeur de chaque variable l'existence d'un support dans les autres variables et si une valeur est supprimée d'un domaine d'une variable, alors toutes les variables qui étaient déjà vérifiées vont être reconsidérées de nouveau, car ces dernières peuvent contenir des valeurs qui n'ont pas de support alors qu'elles en avaient avant.

$AC4$ [MH86] est l'un des successeurs améliorant $AC3$ avec une complexité en temps de $\mathcal{O}(ed^3)$ où e est le nombre de contraintes. Parmi les successeurs de $AC3$, nous avons aussi $AC5$ et $AC7$. Les successeurs de $AC3$, quoiqu'ils améliorent ses performances, ne sont pas aussi génériques que lui. En effet, ces nouveaux algorithmes utilisent des informations liées à la sémantique des contraintes du

problème [BFR95, BFR99]. Une comparaison de complexité entre les différents algorithmes assurant la cohérence d'arc peut être trouvée dans [MF85].

D'autre algorithmes améliorant $AC3$ ont été proposés dans [BR01] à savoir, $AC2000$ et $AC2001$. L'algorithme $AC2000$ ne revérifie que les valeurs qui ont perdu leur support, ce qui nécessite d'enregistrer pour chaque variable l'ensemble des valeurs qui lui ont été supprimées. En revanche, $AC2001$ utilise la propriété d'ordre sur les valeurs des variables de telle sorte que si une valeur a perdu son support dans une autre variable, l'algorithme cherche un nouveau support supérieur au dernier support connu. Ainsi, $AC2001$ ne parcourt pas entièrement les domaines des variables pour chercher un nouveau support d'une valeur.

2.2.3 Autres cohérences locales

D'autres techniques de filtrage basées sur des propriétés plus fortes que l'AC existent. Sans être exhaustif, nous citons la propriété "Path-Consistency" (PC) et la propriété "Restricted Path Consistency" (RPC). Ci-dessous, nous définissons brièvement ces deux propriétés.

La propriété "Path Consistency" (PC) est plus forte que AC et elle est définie comme suit :

$\forall(x,y,z) \in \mathcal{X}^3, \forall(a,b) \in (\mathcal{D}(x) \times \mathcal{D}(y)), \exists c \in \mathcal{D}(z)$, tel que l'affectation $\{< x,a >, < y,b >, < z,c >\}$ est cohérente.

En d'autres termes, la propriété PC, si elle est vérifiée, nous garantit que toute affectation cohérente de deux variables peut être étendue à une affectation cohérente incluant une troisième variable.

Les algorithmes qui vérifient la propriété PC consomme beaucoup de temps de calculs, car pour toute paire (<variable, valeur>, <variable, valeur>), ils devraient s'assurer qu'il existe au moins une possibilité d'extension cohérente vers n'importe quelle variable. Cependant, il serait peut-être intéressant de ne faire ce contrôle de possibilité d'extension que pour les valeurs qui n'ont qu'un seul support sur une variable donné. C'est-à-dire, le contrôle de la PC n'est fait que si : $\exists b \in \mathcal{D}(y)$ tel que b est l'unique support de $a \in \mathcal{D}(x)$. L'idée est de concentrer l'effort de filtrage sur les valeurs qui risquent le plus de devenir incohérentes.

D'autres techniques de filtrage encore plus fortes que la PC et la RPC existent. En exemple, nous citons la $k-$consistency et la $k-$restricted path consistency. Nous renvoyons le lecteur désireux d'approfondir le sujet à [DB01].

Remarque Les techniques de filtrage basées sur la cohérence d'arc présentés précédemment concernent les CSP binaires. Il existe également des algorithmes de filtrage pour les contraintes globales. Par exemple, la contrainte $AllDiff$ admet un algorithme de filtrage qui peut être vu comme une généralisation de la cohérence d'arc [Rég94b].

2.3 Algorithme Branch&Propagate

Pour réduire l'espace de recherche, la construction de l'arbre de recherche peut être combiné a des algorithmes de filtrages. A chaque fois qu'une valeur est affectée à une variable, les contraintes portant sur cette variable sont propagées. Cependant, le choix de l'algorithme de filtrage à utiliser (ou le choix de la propriété de cohérence locale à maintenir) n'est pas évident.

Un utilisateur novice de ces algorithmes peut penser qu'un algorithme qui maintient à chaque point de décision une cohérence locale forte est toujours meilleur que le même algorithme maintenant une propriété de cohérence locale plus faible. En pratique, ceci n'est pas toujours vrai, car en général, plus la propriété de cohérence maintenue est forte plus grande est la complexité de l'algorithme de filtrage.

2.3.1 Forward cheking

L'algorithme Forward Cheking (FC) est l'un des algorithmes de propagation les plus connus et les plus utilisés. Sur un CSP binaire, après chaque affectation d'une variable x_i à une valeur v_i, l'algorithme FC filtre le domaine de chaque variable x_j non encore affectée et liée par une contrainte avec x_i, ne enlevant du domaine de x_j toute valeur v_j telle que l'affectation $\{< x_i, v_i >, < x_j, v_j >\}$ est incohérente .

2.3.2 l'algorithme MAC

L'algorithme MAC (Maintaining Arc Consistency) est également l'un des algorithmes les plus connus. Après chaque affectation d'une variable, il maintient la propriété de cohérence d'arc. Cet algorithme filtre plus de domaines que FC, mais ça ne lui garantit pas d'être le meilleur des deux. En effet, en fonction de l'instance de problème traitée, l'effort de filtrage supplémentaire introduit par le maintien de la cohérence d'arc après chaque affectation peut ne pas compenser la réduction de l'espace de recherche.

2.4 L'algorithme Branch&Propagate&Bound

Dans la section précédente, nous avons parlé des algorithmes qui utilisent la propagation de contraintes au cours de la recherche des solutions. Pour les problèmes où il y a, en plus des contraintes, une fonction objectif F à optimiser, des contraintes sur les bornes de F peuvent être rajoutées à l'ensemble des contraintes du problème au fur et à mesure que des solutions sont trouvées. Ces contraintes sur les bornes de F sont utilisées dans le processus de propagation des contraintes pour filtrer les domaines des variables. Si S est la meilleure solution trouvée jusqu'à présent pour un COP où l'objectif est de minimiser une

fonction F et que $F(S) = f_S$, alors, la contrainte $F(\mathcal{X}) < f_S$ sera rajoutée à l'ensemble des contraintes $\mathcal{C}$ du problème. La propagation de cette contrainte permet de ne pas développer les sous-arbres. On peut raffiner ce principe en ajoutant une étape d'évaluation à chaque noeud de recherche, ce que l'on peut résumer par Branch&Propagate&Bound.

L'idée principale de l'étape d'évaluation est de détecter qu'un ensemble de solutions ne contient pas de solutions meilleures que la meilleure solution trouvée jusqu'à présent sans pour autant calculer séparément chacune des solutions de cet ensemble. En effet, a partir d'une affectation partielle et en se positionnant sur un noeud de l'arbre de recherche, on calcule une borne inférieure (resp. supérieure) de F. Cette borne correspond à la qualité de la meilleure solution que nous pouvons espérer trouver dans ce sous-arbre de recherche. Par conséquent, si cette borne est moins bonne que la qualité de la meilleure solution trouvée jusqu'à présent, l'algorithme peut ne pas développer ce sous-arbre.

La fonction qui calcule les bornes de la fonction F est généralement basée sur une relaxation du problème. Par exemple, pour le TSP, la fonction de calcul de la borne inférieure de la longueur d'un cycle hamiltonien, pour une solution (tour) partielle $\mathcal{T}$, peut être définie comme suit :

$$\mathcal{B}(\mathcal{T}) = \sum_{(i,j)\in\mathcal{T}} d_{ij} + \frac{\sum_{i\notin\mathcal{N}_\mathcal{T}} min^1(i) + min^2(i)}{2} \tag{2.1}$$

où $\mathcal{T}$ est l'ensemble des arêtes déjà traversées ; $\mathcal{N}_\mathcal{T}$ l'ensemble des noeuds déjà visités et $min^1(i)$ et $min^2(i)$ les longueurs des deux plus courtes arêtes incidentes au sommet i. Cette fonction considère deux arêtes au niveau de chaque sommet, car chaque sommet appartenant à un cycle est obligatoirement adjacent à deux autres sommets du même cycle (bien sûr, cela est vrai en supposant que le graphe considéré est simple). Cette fonction donne une borne sur la longueur des cycles hamiltoniens contenant $\mathcal{T}$, i.e., le coût d'un cycle hamiltonien contenant $\mathcal{T}$ est nécessairement supérieur ou égal à $\mathcal{B}(\mathcal{T})$. Par conséquent, tout noeud pour lequel $\mathcal{B}(\mathcal{T})$ est supérieur au coût du meilleur cycle trouvé peut ne pas être développé.

Les performances d'un tel algorithme sont fortement liées à la qualité de la fonction d'estimation des bornes. En effet, plus cette fonction est précise, plus l'espace de recherche est réduit. Cependant, comme pour les algorithmes de filtrage, il s'agit de trouver le bon compromis entre qualité d'estimation et temps de calcul de la borne.

Au début de l'algorithme, la borne inférieure (resp. supérieure) da la fonction objectif F peut être fixée à une valeur suffisamment supérieure (resp. inférieure) pour qu'elle soit améliorée par la première solution trouvée ou elle peut être calculée en utilisant une heuristique donnée.

2.5 Heuristiques d'ordre

A chaque étape de la construction d'un arbre de recherche, l'algorithme doit choisir une variable non encore affectée et lui choisir une valeur de son domaine. L'ordre dans lequel ces variables et ces valeurs sont choisies a un impact crucial sur les performances. C'est pour cette raison que plusieurs heuristiques d'ordre de choix de variables et de valeurs ont été développées.

2.5.1 Heuristiques d'ordre de choix des variables

Une heuristique de choix de variables très populaire est l'heuristique *minDom* qui sélectionne en premier la variable qui a le plus petit domaine [HE80]. Les ex-aequo peuvent être départagés en utilisant le degré dynamique des variables [Bré79] (le nombre de contraintes auxquelles cette variables appartient et qui ne sont pas encore satisfaites). Une autre heuristique est d'utiliser le ratio entre la taille des domaines et le degré des variables [BR96]. D'autres heuristiques considèrent la structure du voisinage d'une variable ou la structure globale de problème [BCS01, MMM01, Smi96].

2.5.2 Heuristiques d'ordre de choix des valeurs

Une fois que la prochaine variable à affecter est sélectionnée, il faut lui choisir une valeur. Si le problème à résoudre consiste à trouver uniquement une solution et que cette solution existe, alors l'ordre de choix des valeurs influence beaucoup le temps de calcul nécessaire pour trouver cette solution. En effet, pour trouver rapidement une solution, il serait intéressant de choisir d'abord les valeurs qui restreignent le moins possible le reste des variables non encore affectées. En revanche, si le problème consiste à trouver toutes les solutions possibles ou si le problème n'admet pas de solution, alors, toutes les valeurs de chaque variable vont être considérées et donc, l'ordre de choix des valeurs n'a plus d'importance.

Parmi les heuristiques génériques d'ordre de choix de valeur les plus utilisées nous citons l'heuristique qui favorise les valeurs qui sont le moins en conflit avec les valeurs des variables non encore affectées. Les heuristiques dédiées d'ordre de choix de valeur dépendent bien sûr du problème à résoudre. Cependant, de manière générale, il serait intéressant de choisir d'abord les valeurs les plus critiques. Par exemple, pour le problème de d'ordonnancement de voitures, il serait intéressant de séquencer d'abord les voitures qui demandent les options qui risquent le plus de dépasser les capacités des stations [1] .

2.5.3 Heuristiques d'ordre basées sur les impacts

Comme nous l'avons déjà dit, en programmation par contraintes, le fait d'affecter une valeur à une variable déclenche la propagation des contraintes dans

1. Nous allons donner plus de détails sur cette heuristique dans le chapitre 5

lesquelles la variable affectée est impliquée. Cette propagation de contraintes se traduit par la réduction de la taille de l'arbre de recherche en filtrant les domaines des variables non encore affectées. La proportion de l'arbre de recherche éliminée par la propagation des contraintes est une information qui peut être récupérée et utilisée pour guider la recherche des solutions.

Refalo [Refo4] a proposé des heuristiques d'ordre basées sur la notion d'impact. Il a défini les impacts d'affectation, de valeur et de variable. L'impact de l'affectation d'une valeur à une variable est défini comme étant la proportion de l'espace de recherche supprimée par la propagation de contraintes causée par cette affectation. L'impact d'une valeur est défini comme étant la moyenne de ses impacts observés et l'impact d'une variable comme étant la moyenne de l'impact des valeurs restantes dans son domaine. Les deux sections suivantes présentent les impacts de manière plus formelle, tels qu'ils ont été définis dans [Refo4].

Impact d'une affectation

Une fois qu'une valeur est affectée à une variable, les domaines des variables non encore affectées vont être diminués par le processus de filtrage. Par conséquent, la taille de l'arbre de recherche restant à explorer est diminuée. L'impact d'une affectation est défini par rapport à la taille de l'arbre de recherche avant et après cette affectation. En effet, pour un $CSP = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, à tout moment, la taille de l'arbre de recherche restant à explorer peut être estimée à :

$$\mathcal{P} = \prod_{x \in \mathcal{X}} |\mathcal{D}(x)| \tag{2.2}$$

si l'on considère $\mathcal{P}_{before}$ la taille de l'arbre de recherche avant l'affectation $< x, v >$ et $\mathcal{P}_{after}$ la taille de l'arbre de recherche restant après cette affectation alors, l'impact de l'affectation $< x, v >$ est :

$$\mathcal{I}(x = v) = 1 - (\frac{\mathcal{P}_{after}}{\mathcal{P}_{before}}) \tag{2.3}$$

Suivant cette définition de l'impact d'une affectation, si un échec survient juste après l'affectation $(x = v)$, alors $\mathcal{I}(x = v) = 1$. A l'inverse, si la propagation de l'affectation n'a réduit aucun domaine, alors $\mathcal{I}(x = v) = 0$.

Impact d'une valeur

Maintenant que l'impact d'une affectation est défini, l'impact d'une valeur v peut être défini comme étant la moyenne de ses impacts (voir l'équation ci-dessus) observés depuis le début de la recherche.

$$\bar{\mathcal{I}}(x = v) = \frac{\sum_{k \in K} \mathcal{I}^k(x = v)}{|K|} \tag{2.4}$$

Avec K l'ensemble d'indexes des impacts de la valeur v observés jusqu'à présent.

Impact d'une variable

L'impact d'une variable est défini comme étant la moyenne des impacts des valeurs restantes dans son domaine. Ainsi, l'impact d'une variable est donné par la formule suivante :

$$\bar{\mathcal{I}}(x) = \frac{\sum_{v \in \mathcal{D}'(x)} \bar{\mathcal{I}}(x = v)}{|\mathcal{D}'(x)|} \tag{2.5}$$

Avec $\mathcal{D}'$, le domaine courant de la variable x.

Utilisation des impacts

Ces impacts sont utilisés pour définir des heuristiques d'ordre : à chaque noeud de l'arbre de recherche, la prochaine variable à affecter est celle ayant le plus fort impact et la valeur affectée est celle ayant le plus petit impact.

Remarque Si nous tenons en compte des définitions des impacts données ci-dessus, à chaque point de décision nous devons tester toutes les valeurs d'une variable pour connaitre leurs impacts ainsi que l'impact de cette variable. Ces calculs peuvent consommer beaucoup de temps. Pour palier à ce problème, Refalo a proposé d'utiliser une estimation de la taille de l'arbre de recherche restant après une affectation. Nous renvoyons le lecteur à [Refo4] pour plus de détails à ce sujet.

2.5.4 Exemple d'heuristique pour le problème d'ordonnancement de voitures

Dans cette section, nous allons donner la définition de l'heuristique DSU (somme dynamique des taux d'utilisation des options) dédiée au problème d'ordonnancement de voitures, car nous allons l'utiliser dans la deuxième partie de cette thèse.

Taux d'utilisation d'une option

En pratique, la difficulté d'une instance du problème d'ordonnancement de voitures est fortement liée aux taux d'utilisation (en anglais "Utilisation Rate" (UR)) des options [Smi96, GPS03]. Ces taux d'utilisation des options est une notion qui traduit le fait qu'une option est d'autant plus critique qu'elle est demandée par beaucoup de voitures par rapport à sa capacité. Avant d'introduire la définition formelle du taux d'utilisation d'une option, nous allons donner quelques notations qui vont être utilisées par la suite.

Notations

- n_{o_i} indique le nombre de voitures qui demandent l'option o_i ;
- p_{o_i} (respectivement q_{o_i}) : indiques la valeur qui est retournée par la fonction p (respectivement la fonction q) définit dans la section 1.4.2. Rappelons que q_{o_i} est égale au nombre maximum de voitures demandant l'option o_i que nous pouvons placer sur une suite de q_{o_i} positions successives sur la chaine de montage ;

Le taux d'utilisation d'une option o_i, noté UR_{o_i}, correspond au ratio entre le nombre de voitures demandant cette option et le nombre de voitures pouvant recevoir cette option sans violer de contraintes de capacité et il est calculé avec la formule ci-dessous.

$$UR_{o_i} = \frac{\text{reqSlots}(o_i, n_{o_i})}{N} \tag{2.6}$$

où N est le nombre de voitures à séquencer et $\text{reqSlots}(o_i, n_i)$ est le nombre minimum de positions permettant de placer n_{o_i} voitures demandant l'option o_i sans violer de contraintes de capacité pour l'option o_i et qui est défini par :

$$\text{reqSlots}(o_i, n_{o_i}) = \begin{cases} q_{o_i} * n_{o_i} / p_{o_i} - (q_{o_i} - p_{o_i}) & n_{o_i} \,\%\, p_{o_i} = 0 \\ q_{o_i} * (n_{o_i} - n_{o_i} \,\%\, p_{o_i}) / p_{o_i} + n_{o_i} \,\%\, p_{o_i} & sinon \end{cases} \tag{2.7}$$

où % est l'opérateur retournant le reste de la division entière.

Signalons que cette dernière formule (introduite dans [BF07]) est plus précise que celle utilisée dans [GPS03] qui ne tient pas compte du fait que n_{o_i} n'est pas forcément un multiple de q_i.

Pour le problème d'ordonnancement de voitures, la fonction heuristique qui s'est montrée la plus performante en moyenne [GPS03] est basée sur la somme dynamique des taux d'utilisation des options demandées par la voiture candidate, c.-à-d.,

$$\eta(\mathcal{A}, x_j, v) = \sum_{o_i \in \text{reqOptions}(v)} UR_{o_i} \tag{2.8}$$

où $\mathcal{A}$ est la séquence (l'affectation partielle) en cours de construction, $\text{reqOptions}(v)$ est l'ensemble des options demandées par la classe de voitures v et UR_{o_i} est le taux d'utilisation défini ci-dessus. Cette heuristique est dynamique, car le taux d'utilisation d'une option est adapté dynamiquement au cours de la construction d'une séquence : le N utilisé dans la formule 2.6 n'est plus le nombre de voitures à séquencer mais le nombre de voitures à séquencer moins le nombre de voitures déjà séquencées dans $\mathcal{A}$.

Pour le problème d'ordonnancement de voitures, l'heuristique DSU est utilisée comme heuristique de choix de valeurs. Pour le choix de la prochaine variable à affecter, on utilise généralement l'ordre naturel des indices des variables en choisissant le variable qui a le plus petit indice et n'est pas encore affectée.

2.5.5 Exemple d'heuristique pour le problème de sac-à-dos

Dans le modèle du problème de sac-à-dos multidimensionnel que nous avons défini en section 1.4.3, chaque variable est associée à un objet et le fait d'affecter la valeur 1 à une variable signifie que l'objet associé à cette variable est sélectionné. Dans cette section, nous allons définir une heuristique de choix de variable (qui

correspond donc à une l'heuristique de choix d'objets) tirée de [ASG04] et que nous allons utiliser dans le chapitre 6.

Soit $\mathcal{A}$ la solution partielle en cours de construction pour un MKP. Les composants de $\mathcal{A}$ sont des couples de la forme $< x_i, 1 >$ si x_i est sélectionné ou $< x_i, 0 >$ si x_i ne peut pas être sélectionné sans violé une contrainte de ressource. Nous définissons $d_{\mathcal{A}}(j)$ la quantité restante de la ressource j comme suit :

$$d_{\mathcal{A}}(j) = r_j - \sum_{<x_i,1>\in\mathcal{A}} x_i \,.\, c_{ij} \tag{2.9}$$

où r_j la quantité de la ressource j disponible au début et c_{ij} la quantité de la ressource j consommé par l'objet associé à la variable x_i. En utilisant $d_{\mathcal{A}}(j)$, nous pouvons calculer le facteur $\mathcal{H}_{\mathcal{A}}(x_i)$ pour toute variable $x_i \in \mathcal{X}$:

$$\mathcal{H}_{\mathcal{A}}(x_i) = \sum_{j=1}^{m} \frac{c_{ij}}{d_{\mathcal{A}}(j)} \tag{2.10}$$

Ce facteur $\mathcal{H}_{\mathcal{A}}(x_i)$ est la somme des taux de consommation de ressources de l'objet associé à la variable x_i. Avec ce facteur, nous définissons le facteur heuristique associé à chaque variable $x_i \in \mathcal{X}$ non encore affectée.

$$\eta(x_i) = \frac{p_i}{\mathcal{H}_{\mathcal{A}}(x_i)} \tag{2.11}$$

où p_i est le profit de l'objet associé à la variable x_i.

Cette heuristique, $\eta(x_i)$, est utilisée au même temps comme heuristique de choix de variables et de valeur. En effet, le sélection d'une variable avec cette heuristique signfie logiquement que cette variable va être affectée à 1.

2.6 Présentation générale d'*IBM CP Optimizer*

Comme nous l'avons indiqué au chapitre 1, *IBM CP Optimizer* est une API qui offre un formalisme de modélisation et des algorithmes génériques de résolution de problèmes de satisfaction de contraintes et d'optimisation combinatoire.

Pour résoudre un problème dans cette *API*, l'utilisateur a le choix entre demander à *IBM CO Optimizer* de résoudre le problème en choisissant un algorithme parmi ceux prédéfinis ou d'écrire son propre algorithme de recherche tout en s'appuyant sur les algorithmes de propagation et de vérification de contraintes prédéfinis.

Une fois le modèle du problème à résoudre est défini, une instance de la classe *IloCP* doit être déclarée. Cette instance va définir l'algorithme de recherche. Un objet IloCP, prend en entrée un modèle (instance de *IloModel*) et cherche la solution optimale de ce modèle. Cet algorithme de recherche admet plusieurs paramètres. Par exemple, l'utilisateur peut fixer le temps de calcul maximal ; le nombre d'échecs autorisés pendant la recherche ; le type de l'algorithme à utiliser etc. En

plus des paramètres, l'utilisateur peut spécifier à l'algorithme de recherche comment faire certains choix. Par exemple, il peut définir la fonction de choix de variables et de valeurs sans se préoccuper du reste ; ou encore, il peut rajouter un niveau supplémentaire de propagation de contraintes etc.

Dans la figure 2.1, nous donnons la suite de l'exemple de la figure 1.2 qui constitue la deuxième partie nécessaire pour la résolution du problème de sac-à-dos multidimensionnel. Pour plus de détails sur *IBM CP Optimizer*, nous renvoyons le lecteur à [ILO98].

2.7 L'algorithme "Restart" d'*IBM CP Optimizer*

Le "Restart" est l'algorithme de recherche par défaut d'*IBM CP Optimizer*. Cet algorithme parcours l'arbre de recherche en profondeur et utilise les "impacts" comme heuristique d'ordre des variables et des valeurs. Il choisit la prochaine variable à affecter celle qui a le plus fort impact et la valeur à affecter à cette variable celle ayant le plus petit impact.

De plus, le Restart reprend la recherche (fait un redémarrage) depuis le début à chaque fois qu'un certain nombre d'échecs est rencontré. Ce nombre d'échecs nécessaire au redémarrage est un paramètre de l'algorithme et il croit d'un redémarrage à un autre de tel sort que chaque redémarrage dure plus longtemps que le précédent.

2.8 Conclusion

Dans ce chapitre, nous avons passé en revue les principaux algorithmes et techniques de résolution de problèmes d'optimisation combinatoire (incluant les problèmes de satisfaction de contraintes) basées sur la recherche complète. Nous avons décrit le principe de l'algorithme *BTC*, construisant un arbre de recherche. Ensuite, nous avons abordé quelques propriétés de cohérence locale qui sont utilisées afin de réduire l'espace de recherche pour accélérer la recherche. Nous avons également parlé des algorithmes plus sophistiqués qui utilisent la propagation de contraintes pour assurer un certain niveau de cohérence locale. Ensuite, nous avons parlé de quelques heuristiques d'ordre sur les variables et les valeurs. Enfin, nous avons introduit l'API IBM ILOG CLPEX CP Optimizer à travers un exemple complet.

Ces algorithmes exacts sont très efficaces sur de nombreux problèmes de taille raisonnable. En revanche, sur les problèmes de plus grande taille, ils peuvent ne pas trouver de bonnes solutions en un temps acceptable. Dans ces derniers cas, les métaheuristiques rentrent en jeu pour prendre le relais. Dans le chapitre suivant, nous allons parler de métaheuristiques qui dans leur ensemble forment une autre famille d'algorithmes de résolution de *COP*.

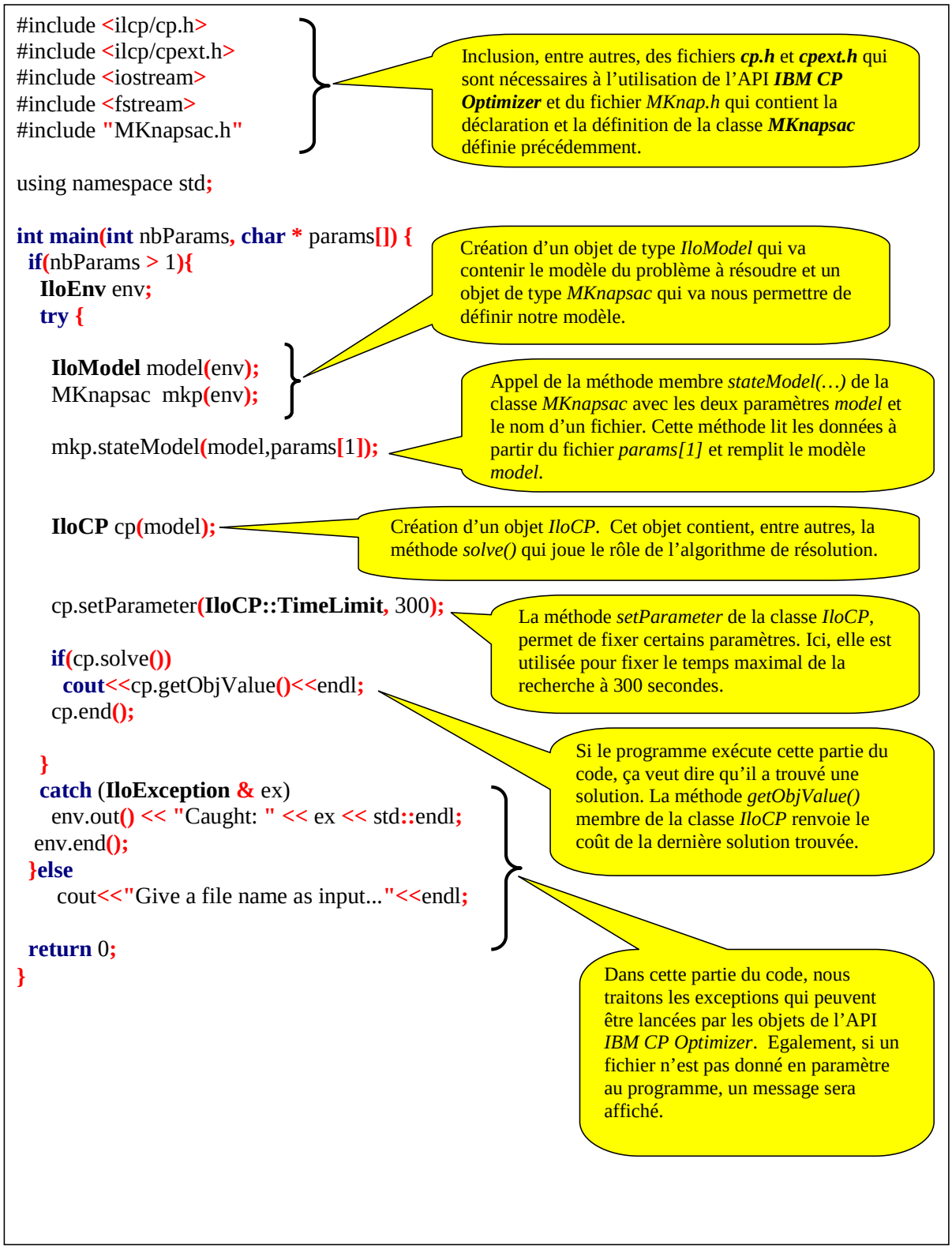

FIGURE 2.1 – *Utilisation de l'API d'IBM CP Optimizer pour résoudre le problème défini dans la figure 1.2*

Méthodes de résolution heuristiques

3

Sommaire

Dans ce chapitre, nous présentons quelques unes des métaheuristiques les plus connues. Nous décrirons des métaheuristiques perturbatives en section 3.1 et des métaheuristiques constructives en section 3.2. En section 3.4, nous décrirons quelques approches hybrides. En section 3.5, nous discuterons des mécanismes d'intensification et de diversification mis en oeuvre par les métaheuristiques et nous discuterons également de l'importance du paramétrage pour ces approches. Enfin, nous présentons des méthodes de paramétrage automatiques.

Comme nous l'avons mentionné au chapitre 2, pour résoudre un COP, deux types d'approches peuvent âtre utilisées à savoir, les approches complètes et les approches métaheuristiques. Les approches complètes garantissent l'optimalité des solutions trouvées. Cependant, leur utilisation est souvent limitée à des problèmes de taille raisonnable, car, trouver les solutions optimales a un coût exponentiel dans le pire des cas. Les métaheuristiques sont une alternative aux algorithmes exacts.

Les métaheuristiques trouvent leur origine dans des phénomènes physiques, biologiques ou encore éthologiques. Indépendamment de leurs origines, elles ont toutes des objectifs et des caractéristiques communes à savoir : - contourner l'explosion combinatoire en s'autorisant à ne pas parcourir tout l'espace de recherche - combine des mécanismes d'intensification (visant à augmenter l'effort de recherche aux alentours des meilleures solutions trouvées) et des mécanismes de diversification (visant à échantillonner le plus largement possible l'espace de recherche).

De manière générale, les métaheuristiques se divisent en deux types. Les métaheuristiques perturbatives et les métaheuristiques constructives. Nous décrivons ces deux approches dans les deux sections suivantes.

3.1 Méthodes perturbatives

3.1.1 Méthodes par Recherche locale

Un algorithme d'optimisation par perturbation évolue dans un espace de recherche constitué d'un ensemble de points où chaque point est une affectation. Son principe est de se déplacer d'un point vers un autre suivant des règles bien déterminées. Il commence toujours par se positionner sur un point de départ (point courant), généralement choisi de manière aléatoire ou en utilisant une heuristique donnée. Ensuite, il applique des règles (appelées règles de perturbation) sur le point de départ pour déterminer ses points voisins. Ces points voisins constituent l'ensemble de points candidats. Cet ensemble peut éventuellement être un singleton ou vide. Une fois cet ensemble constitué, une fonction d'évaluation est utilisée pour évaluer la pertinence de chacun de ses points. Cette évaluation peut se limiter à un sous-ensemble si l'ensemble des candidats est jugé très grand. Après l'évaluation des points candidats, un point parmi eux sera sélectionné suivant une heuristique donnée. A partir du point courant, l'algorithme va se positionner sur le point sélectionné. Cette procédure de déplacement est réitérée jusqu'à ce que les critères d'arrêt soient atteints.

Les ingrédients de base d'un algorithme d'optimisation par perturbation peuvent être résumés par la liste suivante :

1. une fonction qui donne le point de départ.
2. Un ensemble de règles de perturbation applicables sur tout point $\mathcal{A}$ pour déterminer l'ensemble de ses points voisins $\mathcal{N}_{\mathcal{A}}$.

3. Une fonction $Next : (\mathcal{A} \times \mathcal{N}_{\mathcal{A}}) \rightarrow \mathcal{N}_{\mathcal{A}}$ qui fixe le prochain point de l'espace de recherche à visiter.

Algorithme de recherche locale glouton

Un algorithme glouton (en anglais : "greedy algorithm") est principalement caractérisé par la nature de la fonction *Next*, utilisée pour sélectionner le prochain point à visiter : pour un algorithme glouton se trouvant sur un point $\mathcal{A}$, cette fonction sélectionne toujours le prochaine point à visiter parmi ceux améliorant le point $\mathcal{A}$ par rapport à la fonction objectif. Si l'ensemble $\mathcal{N}_{\mathcal{A}}$ ne contient pas d'élément améliorant $\mathcal{A}$, l'algorithme s'arrête et retourne $\mathcal{A}$ comme étant la meilleure solution trouvée.

Les algorithmes gloutons peuvent être très efficaces sur certains problèmes. Cependant, une fois sur un optimum local (un point tel que tous ses voisins sont de moins bonne qualité), les algorithmes gloutons s'arrêtent en retournant cet optimum local comme étant la meilleure solution trouvée. Le problème est que cet optimum local peut être très mauvais par rapport à l'optimum global.

Pour s'extraire du piège de l'optimum local, il est impératif que la fonction *Next* accepte de sélectionner un point même s'il détériore la fonction objectif par rapport à la solution courante. Utilisant ce principe, plusieurs chercheurs ont développé des métaheuristiques qui permettent, chacune à sa façon, de s'échapper du piège de l'optimum local. Ci-après, nous donnons quelques exemples de telles métaheuristiques.

La méthode de recuit simulé

Le recuit simulé est un algorithme itératif proposée par trois chercheur de la société IBM [KGV83] et inspiré par un phénomène physique appelé le recuit. Sans perte de généralités, cet algorithme fonctionne selon les principes suivants : - à chaque itération, l'algorithme choisit aléatoirement un point voisin du point courant - Si le point choisi est meilleur que le point courant, il est accepté (intensification de la recherche) - s'il est moins bon, il est accepté selon une probabilité d'acceptation (voir l'équation 3.1) qui dépend (1) de la qualité du point (moins il est bon , moins il a de chances d'être accepté) (2) d'un paramètre T de température (plus T est élevé, plus il a de chances d'être accepté).

$$\mathcal{P} = e^{-\frac{|f_{\mathcal{A}} - f_{\mathcal{B}}|}{T}} \quad \textit{avec } \mathcal{A} \textit{ le point courant et } \mathcal{B} \textit{ le nouveau point choisit.} \tag{3.1}$$

En regardant de plus près la formule de probabilité d'acceptation 3.1 d'une affectation dégradante, nous pouvons comprendre mieux le rôle du paramètre T et du même comprendre l'analogie avec la méthode de recuit. Dans cette formule, la valeur de $\mathcal{P}$ tend vers l'unité lorsque T tend vers l'infini. Cela signifie que plus le paramètre T est grand plus la probabilité d'accepter une affectation détériorante est grande (diversification de la recherche). En revanche, $\mathcal{P}$ tend vers zéro lorsque

T tend vers zéro. Ce qui signifie que la probabilité d'acceptation d'une affectation détériorante décroit avec la décroissance du paramètre T et par conséquent, la vitesse de convergence vers un optimum local est plus rapide (intensification de la recherche).

Au début du processus de recherche, T est initialisé à une valeur importante afin d'accepter un plus grand nombre de ces mouvements détériorant (diversification de la recherche) ; T est progressivement diminuée afin de progressivement intensifier la recherche.

La recherche taboue

La recherche taboue est une autre technique d'optimisation introduite par F. Glover [Glo86].

La recherche taboue commence par l'évaluation d'un ensemble d'affectations voisines à l'affectation de départ $\mathcal{A}$ et sélectionne la meilleure d'entre-elles. L'affectation sélectionnée prendra la place de l'affectation $\mathcal{A}$ et le processus sera réitéré. Lorsque l'algorithme atteint un optimum local (i.e. : la solution actuelle $\mathcal{A}$ ne peut plus être améliorée), la recherche taboue accepte de passer à une affectation voisine $\mathcal{A}'$ non améliorante. Le risque de passer d'une affectation donnée à une autre affectation non améliorante est de revisiter les mêmes solutions au fil des itérations et donc, rester piéger dans un sous espace de recherche.

Pour éviter de tourner en cycle, la recherche taboue introduit une liste qui s'appelle la liste taboue. Cette liste sert à enregistrer les derniers mouvements effectués pour les considérer comme étant des mouvements tabous (interdits). L'interdiction d'un mouvement n'est valable que pendant un certain temps t (un certain nombre d'itérations de l'algorithme). Pour rendre cette méthode plus efficace, lorsqu'un mouvement est tabou alors qu'il permet d'obtenir une affectation meilleure que celles déjà trouvées, l'algorithme peut remettre en cause le statut de ce mouvement et l'autoriser de nouveau. Dans ce cas, nous dirons que ce mouvement est aspiré.

La valeur du paramètre t influence beaucoup les performances de la recherche taboue. En effet, d'un côté, si la valeur de t est petite, la recherche taboue risque d'être trop intensifiée et aura tendance à rester piégée dans les optima locaux. De l'autre côté, si la valeur de t est très grande, la recherche pourra visiter différentes zones de l'espace de recherche mais elle risque de ne pas être capable d'intensifier la recherche pour atteindre l'optimum global. En résumé, une bonne valeur du paramètre t est celle qui permet un bon équilibre entre l'intensification et la diversification de la recherche.

Notons que La recherche taboue est l'une des meilleures approches heuristiques connues pour la résolution des CSPs [GH97].

3.1.2 Les algorithmes génétiques

Inspirées du processus d'évolution des espèces vivantes, les algorithmes génétiques (en anglais : "Genetic Algorithms" (AGs)) ont été développés par J. Holland [Hol92]. Dans les algorithmes génétiques, un ensemble d'individus [1], appelé la population initiale, est généré suivant un principe donné. A partir de ces individus, une nouvelle population d'individus est reproduite. Cette reproduction est faite en appliquant les opérateurs de sélection, de croisement et de mutation. Chacun de ces opérateurs a un objectif différent des autres mais dans leurs ensemble espèrent faire évoluer l'ensemble d'individus vers d'autres individus plus performants.

Sans perte de généralité, ces opérateurs peuvent être définit comme suit :

- **L'opérateur de sélection** : classe les individus en se basant généralement sur leurs évaluations par rapport à la fonction objectif [2] puis sélectionne les meilleurs d'entre eux.
- **L'opérateur de croisement** : combine, généralement, les gènes [3] de deux individus parents pour former deux autres individus enfants. Concrètement, cet opérateur coupe deux individus en une ou plusieurs positions et construit deux nouveaux individus où chacun aura des fragments issus des deux parents.
- **L'opérateur de mutation** : change un ou plusieurs gènes de l'individu sur lequel il est appliqué pour lui apporter des caractéristiques nouvelles.

Dans les algorithmes génétiques, la balance entre l'intensification et la diversification de la recherche est gérée à travers la manière dont les trois opérateurs définis ci-dessus sont appliqués. Par exemples :

1. avec l'opérateur de sélection, nous pouvons sélectionner les meilleurs individus parmi tous les individus de la population (intensification de la recherche) ou bien faire une sélection en plusieurs itérations et à chaque itération on sélectionne le meilleur individu parmi un sous-ensemble d'individus choisis de manière aléatoire (diversification de la recherche)
2. avec l'opérateur de croisement, nous pouvons croiser les parents en un petit nombre de positions et avoir des enfants plus proches de leurs parents (intensification de la recherche) ou croiser les parents sur un nombre relativement grand de positions (diversification de la recherche)
3. avec l'opérateur de mutation, nous pouvons ne muter qu'un petit nombre de gènes (intensification de la recherche) ou changer un grand nombre de gènes (diversification de la recherche).

1. dans la communauté des algorithmes génétiques, le terme individu désigne une affectation
2. dans la communauté des GAs, la fonction objectif est appelée fonction de fitness
3. un gène représente une composante d'une affectation

3.2 Méthode Constructives

Les approches par perturbations décrite en 3.1 génèrent de nouvelles affectations en appliquant des perturbations sur des affectations de départ. Dans cette section, nous allons parler des méthodes qui génèrent des affectations à partir d'une affectation vide en rajoutant des composants un par un jusqu'à ce que l'affectation soit complète. La brique de base de ce genre d'algorithmes est la fonction qui sélectionne le prochain composant à ajouter à l'affectation partielle. Cette sélection peut être faite soit de manière déterministe soit de manière probabiliste. Nous allons commencer par un bref rappel sur les algorithmes gloutons. Puis nous allons présenter les algorithmes à estimation de distribution et enfin, nous allons parler des algorithmes d'optimisation par colonies de fourmis sur lesquels repose l'essentiel des travaux de cette thèse.

3.2.1 Algorithmes constructifs gloutons

Les algorithmes gloutons constructifs sont basés sur le principe de construction d'affectations de manière incrémentale. A partir d'une affectation vide, ils rajoutent les composants d'affectations un par un jusqu'à l'obtention d'une affectation complète. Le choix du prochain composant à rajouter est fixé par une heuristique donnée. Cette heuristique est généralement dédiée au problème traité, et l'efficacité d'un tel algorithme dépend principalement de l'efficacité de cette heuristique.

Evidemment, les premières décisions prises par un algorithme glouton influencent beaucoup la qualité des solutions trouvées. Ceci est d'autant plus vrai lorsque les variables du problème sont dépendantes. Pour palier à ce problème, nous pouvons naturellement penser à changer ce mécanisme rigide de prise de décision en introduisant un peu d'aléatoire. Ainsi, on peut choisir le prochain composant en fonction de probabilités qui peuvent être définies proportionnellement à l'heuristique de sorte que les "bons" composants ont plus de chances d'être sélectionnés. Ces constructions gloutonnes peuvent être répétées plusieurs fois. On retourne ensuite la meilleure affectation construite. Ce mécanisme de constructions gloutonnes aléatoires itérés peut être amélioré en exploitant l'expérience passée.

3.2.2 Algorithmes à estimation de distribution

La philosophie derrière les algorithmes à estimation de distribution (en anglais : "Estimation of Distribution Algorithms" (EDA)) ressemble à celle des algorithmes génétiques. En effet, les EDA [LL01]proposent de faire évoluer un modèle stochastique utilisé pour générer des affectations. Le schéma de base d'un algorithme de type EDA peut être représenté par les étapes décrites ci-dessous :

1. Génération d'une population initiale d'affectations $\mathcal{P}$ en utilisant une distribution de probabilités $\mathcal{D}_0$ donnée.

2. Evaluation des affectations de la génération $\mathcal{P}$.
3. Sélection des meilleures affectations en utilisant une méthode de sélection donnée.
4. Construction d'une nouvelle distribution de probabilité $\mathcal{D}$ en se basant sur l'ensemble des affectations sélectionnées à l'étape (3).
5. Génération d'une nouvelle population d'affectations $\mathcal{P}'$ en utilisant la nouvelle distribution de probabilité $\mathcal{D}$.
6. Mise-à-jour de la population $\mathcal{P}$ en fonction de $\mathcal{P}'$.
7. Si les conditions d'arrêt ne sont pas atteintes, alors aller à l'étape (2).

La difficulté majeure des EDAs réside dans le choix du modèle probabiliste considéré. Un modèle simple consiste à considérer chaque variable séparément (comme cela est proposé dans [BAL94]). Des modèles plus sophistiqués peuvent considérer certaines dépendances entre les variables en utilisant des réseaux bayésiens [PGCP99].

Bien que les EDAs aient été conçus à l'origine comme une variante d'algorithme évolutionnaires, ils présentent une forte similitude avec les algorithmes d'optimisation par colonies de fourmis. En effet, comme nous allons le voir dans la section suivante, cette façon de construire des solutions en utilisant les probabilités de sélection pour chaque composante de solution est l'un des principes fondamentaux des algorithmes d'optimisation par colonies de fourmis.

3.3 Optimisation par colonies de fourmis (*ACO*)

Les algorithmes de colonies de fourmis sont inspirés du comportement des fourmis réelles. Les fourmis ont des capacités simples et limitées mais peuvent ensemble accomplir des tâches complexes. L'exemple le plus frappant est leur capacité à trouver un chemin le plus court ou proche du plus court.

Dans la nature, les fourmis communiquent entre elles à travers leur environnement en déposant une substance chimique, appelée la phéromone, sur les chemins qu'elles empruntent. Cette phéromone influence beaucoup le comportement global des fourmis. Effectivement, lorsqu'une fourmi se déplace dans un environnement donné elle détecte la présence de la phéromone sur différentes pistes (différentes directions) et elle a tendance à emprunter la piste qui contient le plus de phéromones. L'expérience du double pont [DAGP90, GADP89] a montré comment le dépôt de phéromone mène progressivement les fourmis à emprunter le chemin le plus court. Les auteurs de cette expérience ont connecté une fourmilière à une source de nourriture par deux chemins de longueurs différentes. Après quelques minutes, ils ont constaté que presque toutes les fourmis utilisent le chemin le plus court. L'explication de ce phénomène est basée essentiellement sur le fait que les fourmis laissent des traces de phéromone sur les chemins qu'elles empruntent. En effet, les fourmis qui ont choisi aléatoirement de pendre le plus

court chemin en partant de la fourmilière seront les premières à arriver à la source de nourriture. Lorsque ces fourmis seront de retour vers la fourmilière elles détecteront les traces de phéromones déposées par elles-mêmes et elles préféreront donc prendre le même chemin de retour que celui de l'aller. Ainsi, la phéromone s'accumulera plus rapidement sur le chemin le plus court que sur l'autre chemin. Au bout de quelques minutes, les quantités de phéromones déposées sur les deux chemins seront assez différentes pour qu'une majorité de fourmis préfèrent le plus court chemin.

Inspiré de ce phénomène naturel où des individus simples accomplissent ensemble une tâche complexe, Dorigo et ses collègues ont conçu une heuristique, baptisée Ant System (*AS*) [Dor92, DMC91], pour résoudre le problème de voyageur de commerce. Ainsi, ils ont initié le domaine de l'Optimisation par Colonies de Fourmis (en anglais Ant Colony Optimization (ACO)). Depuis, plusieurs auteurs se sont inspirés de *AS* et ont proposé d'autres variantes. Ci-dessous, nous citons quelques uns de ces algorithmes.

3.3.1 Principe général des algorithmes *ACO*

D'une façon informelle, le principe général de tous les algorithmes ACO est de permettre à des fourmis artificielles de construire des affectations du problème considéré. Dans ces algorithmes, une fourmi construit une affectation de façon gloutonne par une succession de décisions probabilistes. Chacune de ces décisions consiste à étendre une affectation partielle par le rajout d'un composant de solution. Ces composants vont être rajoutés un par un jusqu'à ce qu'une affectation complète soit construite. La séquence des décisions qui ont permis la construction d'une affectation peut être vue comme un chemin dans le graphe de décision sous-jacent au problème traité.

L'objectif principal des fourmis artificielles est de trouver des solutions (des chemins) de bonne qualité à travers le graphe de décisions. Cet objectif est atteint via un processus itératif où les fourmis qui ont trouvé des affectations de bonne qualité guident les fourmis des prochaines itérations. Plus précisément, les fourmis d'une itération qui ont trouvé les meilleures affectations sont autorisées à déposer de la phéromone sur les noeuds (ou les arêtes) correspondant à leurs chemins respectifs dans le graphe de décision. Ces phéromones, guideront progressivement les fourmis des prochaines itérations qui intensifieront la recherche autours des bonnes solutions.

Divers algorithmes *ACO* existent [DS04]. Cependant, ils ont beaucoup de points en commun dont le plus essentiel est leur schéma global. Plus formellement, les algorithmes ACO suivent, généralement, le schéma global donné par l'algorithme 1. Ces algorithmes commencent toujours par initialiser toutes les traces de phéromones (ligne 1) à des valeurs égales et puis, ils rentrent dans la boucle principale (lignes 2-7) qui consiste à construire un ensemble de solutions (lignes 3-4)(une solution par fourmi) et à mettre à jour la phéromone (lignes 5-6).

Dans les algorithmes ACO, chaque itération de la boucle principale est habituellement appelée un cycle.

La mise-à-jour des phéromones ce fait en deux étapes. La première étape (ligne 5) consiste à diminuer les quantités de phéromone sur tous les noeuds (ou arêtes) du graphe de décision afin de diminuer l'influence des anciennes itérations sur les futures itérations. Cette étape introduit donc la notion d'oubli et elle est également appelée l'étape d'évaporation. La deuxième étape (ligne 6) consiste en un dépôt de phéromones sur des affectations précédemment construites (généralement les meilleures). L'objectif de ce dépôt de phéromone est d'intensifier la recherche dans les zones de l'espace de recherche les plus prometteuses.

Dans la suite de cette section, nous allons définir, plus en détails, chacune des étapes principales de l'algorithme 1 à savoir : la définition des composantes phéromonales ; le processus de construction d'affectations ; le processus de mise-à-jour de la structure phéromonale et ensuite, nous allons donner quelques rappels sur quelques-unes des variantes d'algorithmes d'optimisation par colonies de fourmis existants dans l'état de l'art.

Algorithme 1 : Schéma général des algorithmes de fourmis

```
1 Initialisation des traces de phéromones
2 répéter
    /* Étape 1 : Construction de nbAnts solutions        */
3   pour chaque fourmi faire
4     Construire une solution
    /* Étape 2 : Evaporation des phéromones              */
5   Diminuer la phéromone sur toutes les traces phéromonales
    /* Étape 3 : Dépôt de phéromones                     */
6   Augmenter la phéromone sur certaines traces phéromonales
7 jusqu'à Critère d'arrêt atteint ;
```

Structure phéromonale

Dans les algorithmes ACO, le problème à résoudre est modélisé, généralement, sous la forme de la recherche d'un meilleur chemin dans un graphe appelé graphe de construction. Dans ce graphe, les noeuds représentent les composants d'affectations et les arêtes représentent l'enchainement des composants lors de la construction d'affectations. La phéromone est donc déposée sur les noeuds ou sur les arêtes du graphe de construction et on note généralement τ_i la quantité de phéromone déposée sur le noeud i et τ_{ij} la quantité de phéromone déposée sur l'arête $e(i, j)$.

Au début de l'exécution d'un algorithme ACO, toutes les traces de phéromones sont initialisées à une valeur appelée τ_0 qui est un paramètre de cet algorithme.

Exemple de structures de phéromones pour le TSP Le TSP est généralement

représenté par un graphe $G(N, E)$ (voir chapitre 1 section 1.4.1) composé d'un ensemble de noeuds $i \in N$ et d'un ensemble d'arêtes $e \in E$ reliant les noeuds de N deux à deux. La structure de phéromone pour le TSP peut être définie comme suit : pour chaque arête $e(i, j) \in E$, on associe une trace de phéromone τ_{ij}. Intuitivement, la quantité de phéromone déposée sur τ_{ij} représente l'intérêt, suivant l'expérience acquise par les fourmis dans le passé, de visiter la ville j juste après avoir visité la ville i.

Construction d'affectations

A chaque cycle d'un algorithme ACO, chaque fourmi construit une affectation. La construction d'une affectation par une fourmi se fait selon un principe glouton aléatoire : à partir d'une affectation vide, une fourmi ajoute les composants d'affectation un par un jusqu'à ce qu'une affectation complète soit construite. Le choix du prochain composant à ajouter se fait suivant une règle de transition probabiliste. A partir d'une solution partielle $calS$, une fourmi choisi un composant à ajouter à partir d'un ensemble de composants candidats "Cand" selon la probabilité

$$p_{\mathcal{S}}(i) = \frac{[\tau_{\mathcal{S}}(i)]^{\alpha} \cdot [\eta_{\mathcal{S}}(i)]^{\beta}}{\sum_{j \in \mathcal{C}and} [\tau_{\mathcal{S}}(j)]^{\alpha} \cdot [\eta_{\mathcal{S}}(j)]^{\beta}} \tag{3.2}$$

où

- $\tau_{\mathcal{S}}(i)$ est le facteur phéromonal associé au composant i qui, en fonction de l'application considérée, peut être dépendant des composants qui sont déjà sélectionnés dans l'affectation partielle $\mathcal{S}$. Par exemple, il peut être égale à la quantité de phéromone déposée sur le composant i, i.e., τ_i ou il peut être égale à la quantité de phéromone déposée entre le dernier composant j ajouté dans $\mathcal{S}$ et le composant i, i.e., τ_{ij} ou encore, il peut être défini par une fonction qui prend en compte les quantités de phéromones déposées entre le composant i et certains ou tous les composants déjà dans $\mathcal{S}$.
- $\eta_{\mathcal{S}}(i)$ est le facteur heuristique associé au composant i et qui est généralement en fonction des composant déjà sélectionnés dans $\mathcal{S}$. La définition de ce facteur dépend de l'application considérée.
- α et β sont deux paramètres qui déterminent l'influence des deux facteurs dans la probabilité de transition. Notons que si $\alpha = 0$, alors le facteur phéromonal n'intervient pas dans le choix des composants et l'algorithme se comporte comme un algorithme glouton aléatoire pur.

Avec cette règle de transition probabiliste, plus la phéromone associée au composant i est supérieure plus grande est la probabilité que ce composant sera sélectionnée.

Exemple de construction d'affectations pour le TSP Le but d'une fourmi lors de la résolution d'une instance TSP est de parcourir le graphe $G(N, E)$ en vue de

construire un cycle hamiltonien $\mathcal{T}$. En partant d'un tour vide $\mathcal{T} = \emptyset$, une fourmi rajoute les noeuds (ou les arêtes) un par un jusqu'à ce qu'un tour hamiltonien est construit. En supposant que les traces de phéromones sont associées aux arêtes du graphe G, une fourmi k se trouvant sur un noeud i choisit le prochain noeud j à visiter avec une probabilité calculée en utilisant la règle de transition suivante :

$$\begin{cases} p_{ij}^k(t) = \dfrac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{y \in \mathcal{N}_i^k} [\tau_{iy}(t)]^\alpha \cdot [\eta_{iy}]^\beta} & si\ j \in \mathcal{N}_i^k \\ p_{ij}^k(t) = 0 & sinon \end{cases} \tag{3.3}$$

où :

- $\tau_{ij}(t)$ est la quantité de phéromone accumulée sur l'arête $e(i,j)$ jusqu'à l'instant t.
- $\eta_{ij} = \frac{1}{d_{ij}}$ est le facteur heuristique associé à l'arête $e(i,j)$ avec d_{ij} la distance entre le noeud i et le noeud j (la distance entre la ville i et la ville j). Cette valeur heuristique est inversement proportionnelle à la distance, car l'idée est de favoriser la sélection des noeuds les plus proches du noeud i.
- α et β sont deux paramètres qui donnent respectivement le poids du facteur phéromonale et celui du facteur heuristique.
- $\mathcal{N}_i^k$ est l'ensemble des noeuds non encore visités. Cet ensemble ne contient généralement que les noeuds les plus proches du noeud i.

Mise-à-jour de la structure phéromonale

Une fois que chaque fourmi a construit une affectation, les traces de phéromones vont être mises à jour. Le processus de mise-à-jour des phéromones se divise, généralement, en deux étapes :

- La première étape consiste à évaporer (réduire) toutes les traces de phéromone en les multipliant par le paramètre de l'évaporation ρ ($0 \leq \rho \leq 1$). Cette évaporation a comme effet de diminuer l'influence des anciennes itérations et de favoriser les plus récentes.
- La deuxième étape consiste à déposer de la phéromone sur les composants des meilleures affectations construites. La quantité de la phéromone déposée sur un composant dépend généralement de la qualité de l'affectation qui contient ce composant.

Lors de la deuxième étape de la mise-à-jour des phéromones, le choix des affectations à récompenser dépend de la stratégie que l'on souhaite utiliser. Nous pouvons par exemple choisir de récompenser les composants de toutes les affectations du dernier cycle ; on peut également adopter une stratégie élitiste où on ne récompense que les composants des meilleures solutions ; on peut intensifier plus la recherche en récompensant les composants de la meilleure solution trouvée depuis le début de la recherche. Le choix de la stratégie de récompense à utiliser dépend donc du niveau d'équilibre que l'on souhaite avoir entre l'intensification et la diversification de la recherche.

Exemple de mise-à-jour de la structure de phéromone pour le TSP

Reprenons notre exemple de TSP où les traces de phéromones sont associées aux arêtes du graphe $G(N, E)$, i.e., pour chaque arête $e(i, j) \in E$ est associée une trace de phéromone τ_{ij}. La mise-à-jour des trace de phéromone ce fait en deux étapes :

- La première étape consiste à évaporer (réduire) toutes les traces de phéromone en utilisant la formule suivante :

$$\tau_{ij}(t+1) \leftarrow (1-\rho) \cdot \tau_{ij}(t) \quad \forall (i,j) \in N^2 \tag{3.4}$$

- La deuxième étape consiste à accumuler de la phéromone sur les composants des affectations (tours) construites. En effet, chaque fourmi dépose une quantité de phéromone sur toutes les arêtes qui composent le tour qu'elle a construit. Ce dépôt de phéromone est accompli grâce à la formule suivante :

$$\tau_{ij}(t+1) \leftarrow \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \quad \forall (i,j) \in N^2 \tag{3.5}$$

où :

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{1}{\mathcal{L}^k} & si\ e(i,j) \in \mathcal{T}^k \\ 0 & sinon \end{cases} \tag{3.6}$$

avec $\mathcal{L}^k$ la longueur de tour $\mathcal{T}^k$ construit par la k^{ime} fourmi. La longueur d'un tour $\mathcal{T}$ est la somme des longueurs des arêtes qu'il contient.

Ce mécanisme de mise-à-jour des phéromones permet aux fourmis de converger progressivement vers des zones de l'espace de recherche qui contiennent des affectations (des tours) de bonne qualité. En effet, l'équation 3.6 veut dire que les arêtes qui apparaissent dans les meilleures affectations (les tours les plus court) recevront plus de phéromones que les arêtes qui apparaissent dans des tours moins bons.

3.3.2 Principaux schémas ACO

Ant System (AS)

Ant System (AS) est l'ancêtre de tous les algorithmes d'optimisation par colonies de fourmis. Il a été proposé par Dorigo [Dor92]. La première application de (AS) fut sur le problème du voyageur de commerce.

Dans AS, les phéromones sont associées aux différentes arêtes $e(i, j) \in E$ du graphe G et elles sont initialisées avec la valeur $\tau_0 = \frac{m}{C^{nn}}$ où m est le nombre de fourmis (le nombre d'affectations par itération) et C^{nn} la longueur du cycle généré par l'heuristique du plus proche voisin (nearest neighbor). L'idée d'initialiser les phéromones à cette valeur est d'avoir, au début, sur toutes les traces phéromonales τ_{ij} une quantité de phéromone estimée à la quantité de phéromone qu'une fourmi aura à déposer à la fin d'un cycle.

Dans l'algorithme AS appliqué au TSP, les fonctions de construction d'affectation et de mise-à-jour des phéromones sont exactement ceux que nous avons donné dans les exemples ci-dessus.

AS donne des résultats de bonne qualité sur des petites instances de TSP. Cependant, comparé avec d'autres métaheuristiques, ses performance décroient considérablement avec la croissance de la taille des instances traitées. Par conséquent, beaucoup de chercheurs ont proposé de nouvelles variantes de AS qui lui ressemble beaucoup mais améliorent ses performances. Ci-après, nous donnons deux de ces variantes les plus connues.

$\mathcal{M}ax - \mathcal{M}in$ Ant System

L'algorithme $\mathcal{M}ax - \mathcal{M}in$ Ant System $(MMAS)$ est l'un des successeurs de AS les plus performant. Proposé dans [SHoo], il introduit quatre nouveautés par rapport à AS qui sont les suivante :

1. L'exploitation intense des meilleures affectations trouvées. Après chaque itération, les traces de phéromones sont mises-à-jour uniquement en utilisant soit la meilleure affectation trouvée lors de la dernière itération soit la meilleure affectation trouvée depuis le début de l'algorithme.
2. Les valeurs de phéromone possibles sur les arêtes sont restreintes à être comprises dans l'intervalle $[\tau_{min}, \tau_{max}]$ avec τ_{min} et τ_{max} sont deux paramètres de l'algorithme. Le but de restreindre les valeurs des phéromones est d'éviter la stagnation prématurée de la recherche. En effet, avec cette limitation, en supposant que $\beta = 0$ (voir la formule 3.3), la probabilité pour qu'un composant soit sélectionné est limitée à $p_{max} = \frac{\tau_{max}^{\alpha}}{(n-1)(\tau_{min}^{\alpha})+\tau_{max}^{\alpha}}$ et ceci quelque soit le nombre d'itérations autorisées.
3. Les valeurs initiale des phéromones τ_0 est fixée à τ_{max}.
4. A chaque fois que l'algorithme approche un état de stagnation (Les fourmis construisent les mêmes affectations) ou si la meilleure affectation trouvée depuis le début de l'algorithme n'a pas été améliorée depuis un certain nombre d'itérations, l'algorithme réinitialise toutes les traces de phéromones à τ_{max} (c à-d : il fait un "restart").

Les expérimentations de cet algorithme sur le TSP ont montrées que pour les petites instances, il vaudrait mieux utiliser uniquement la meilleure affectation de la dernière itération pour la mise-à-jour des phéromones, tandis que pour de plus grandes instances, il serait intéressant d'altérer entre la meilleure affectation de la dernière itération et la meilleure affectation trouvée depuis le début de l'algorithme.

Ant Colony System

Dans le but d'améliorer les performances de AS sur des problèmes de grandes tailles, Dorigo et Gambardella ont proposé dans [DG97] une variante appelée Ant

Colony System (ACS). Appliquée sur le TSP, cette nouvelle variante est essentiellement différente de AS sur les trois points suivants :

1. Une fourmi k se situant sur le noeud i, choisit le prochain noeud j avec la formule suivante :

$$j = \begin{cases} argmax \sum_{y \in \mathcal{N}_i^k} [\tau_{iy}(t)]^\alpha \cdot [\eta_{iy}]^\beta & si\ q \leq q_0 \\ \mathcal{J} & sinon \end{cases} \tag{3.7}$$

   où q est une variable aléatoire uniformément distribuée dans l'intervalle [0,1], q_0 est un paramètre tel que $0 \leq q_0 \leq 1$, $\mathcal{J}$ est une variable aléatoire sélectionnée en utilisant la formule 3.3.

   La valeur de paramètre q_0 est très importante, car lorsque $q \leq q_0$, l'algorithme exploite au maximum les informations apprises (c à-d. : l'algorithme fait de l'exploitation en cherchant des solutions autour des meilleures solutions déjà trouvées) et lorsque $q \geq q_0$, l'algorithme utilise la règle standard de AS (voir l'équation 3.3) qui lui permet de faire un choix probabiliste et donc il explore l'espace de recherche.

2. Dans ACS, après chaque itération, une seule fourmi est autorisée à déposer de la phéromone. Cette fourmi est celle qui a construit la meilleure affectation depuis le début de l'algorithme $\mathcal{T}^{bs}$. La formule de mise à jour des phéromones devient :

$$\tau_{ij}(t+1) \leftarrow (1-\rho)\tau_{ij}(t) + \rho\Delta\tau_{ij}^{bs} \quad \forall e(i,j) \in \mathcal{T}^{bs} \tag{3.8}$$

   Avec $\Delta\tau_{ij}^{bs} = \frac{1}{L^{bs}}$. Cette formule signifie également que l'évaporation des phéromones, contrairement à AS, n'a lieu que sur les arêtes de $\mathcal{T}^{bs}$.

3. Dès qu'une fourmi se déplace d'un noeud i vers un noeud j, elle modifie la quantité de phéromone sur l'arête $e(i,j)$ en utilisant l'équation suivante :

$$\tau_{ij}(t+1) \leftarrow (1-\xi)\tau_{ij}(t) + \xi\tau_0 \tag{3.9}$$

   où ξ et τ_0 sont deux paramètres tel que $0 < \xi < 1$ et τ_0 la valeur initiale des phéromones sur toutes les arêtes du graphe G. Cette modification de phéromone, qui servient directement après le choit d'une arête, est appelée la mise-à-jour de phéromone locale.

3.4 Méthodes hybrides

Plusieurs auteurs ont proposé des méthodes d'optimisation en combinant différentes approches, complètes et incomplètes. Dans ces méthodes, il s'agit généralement de tirer profi des atouts d'une approche, généralement une métaheuristique, en la combinant avec une procédure de recherche locale. Ci-dessous, nous citons quelqu'une de ces méthodes.

3.4.1 GRASP

De manière générale, GRASP (Greedy Randimized Adaptive Search Procedure) est une procédure itérative, introduite dans [FR95], qui combine un algorithme constructif glouton aléatoire et un algorithme de recherche locale. En effet, cette méthode réalise plusieurs itérations sur deux phases séquentielles. La première phase consiste en la construction d'une affectation de manière itérative en ajoutant à une affectation partielle (au début vide) les composants d'affectation un par un jusqu'à ce qu'une affectation complète soit construite. Pour sélectionner le prochain composant à ajouter à l'affectation partielle, GRASP utilise une fonction gloutonne qui sélectionne un ensemble de composants candidats. A partir de l'ensemble des candidats, GRASP choisi un composant de manière aléatoire et il le rajoute à l'affectation en cours de construction. La deuxième étape consiste en l'amélioration de la dernière affectation complète construite via un algorithme de recherche locale. Ces deux étapes sont répétées jusqu'à ce que les critères d'arrêts soient atteints et la meilleure affectation trouvée est retournée.

3.4.2 Algorithme évolutioniste et la recherche locale

Dans [BDHS04], les auteurs ont proposé un algorithme générique hybride pour la résolution des CSP. L'algorithme proposé consiste principalement en une collaboration entre des techniques issues de la programmation par contraintes (propagation des contraintes, cohérence d'arcs, heuristique de choix de variable MinDom etc.) et des techniques de recherche locale telle que par exemple la recherche taboue. Egalement, l'espace de recherche est géré d'une manière inspirée par les algorithmes évolutionnistes où une population d'individus couvre l'espace des solutions en entier. Notons qu'ici un individu ne représente pas une affectation telle que nous l'avons définit dans le chapitre 1 mais représente une partie de l'espace de recherche.

Dans la forme générale de cette algorithme, les solutions sont recherchées par une succession d'étapes séquentielles et qui sont principalement les suivantes : - la première étape consiste en la sélection d'un individu (un sous-espace de recherche). Différentes méthodes de sélection peuvent être appliquées et notamment, nous pouvons appliquer les méthodes habituellement utilisées dans les algorithmes évolutionnaires. - la deuxième étape consiste en la réduction des domaines des variables de l'individu sélectionné par l'élimination des valeurs qui ne peuvent pas participer dans les solutions contenus dans le sous-espace de recherche représenté par cet individu. Cette réduction est effectuée par une procédure de propagation de contraintes et elle est définie en fonction du niveau de cohérence locale que l'on souhaite obtenir. - la troisième étape de cet algorithme consiste en l'application d'une méthode de recherche locale, telle que la recherche taboue, sur l'espace de recherche représenté par l'individu sélectionné. - la dernière étape de cet algorithme consiste à comptabiliser la solution éventuellement trouvée après la deuxième ou la troisième étape et à l'éclatement de l'individu

sélectionné en plusieurs autres individus. La génération d'individus à partir de l'individu sélectionné se fait en découpant un de ses domaines en plusieurs sous domaines. Egalement, le choix du domaine à découper peut faire recours aux techniques de choix de variables utilisées en programmation par contraintes.

Notons également qu'en fonction des conditions d'arrêts de l'algorithme, la recherche peut être complète ou incomplète. Cet algorithme a été appliqué à plusieurs problèmes différents et la conclusion est que l'efficacité d'une instanciation de cet algorithme dépend du problème traité.

3.4.3 ACO et la recherche locale

Comme nous l'avons vu dans ce chapitre, les algorithmes de la recherche locale manipulent, généralement, des affectations complètes en les améliorants de manière progressive. De tels algorithmes peuvent être utilisés pour améliorer les affectations construites par les fourmis dans les algorithmes ACO. En effet, dans [SB06], C. Solnon et al. ont proposé, en option, d'appliquer une procédure de recherche locale sur les affectations construites par les fourmis afin de les améliorer.

Egalement, dans [DS04], les auteurs ont proposé, pour la résolution de TSP, d'appliquer l'algorithme 2-opt ou 3-opt pour améliorer le coût des chemins trouvés par les fourmis.

Dans beaucoup d'algorithmes ACO, la recherche locale est utilisée pour l'amélioration des affectations construites par les fourmis. Cependant, l'utilisation de la recherche locale peut engendrer un temps de calcul supplémentaire qui peut ne pas être négligeable.

3.4.4 Combinaison de ACO et la PPC

Dans [Mey08], B. Meyer a proposé deux algorithmes hybrides où ACO a été couplée avec la PPC. Il a proposé un premier couplage faible où les deux approches fonctionnent en parallèle en échangeant seulement les solutions et les bornes de la fonction objectif. Il a proposé un deuxième couplage plus fort, où la propagation de contraintes a été incorporée dans ACO pour permettre à une fourmi de revenir en arrière (backtrack) lorsqu'une affectation d'une valeur à une variable donnée échoue. Toutefois, la procédure de retour en arrière a été limitée au niveau de la dernière variable choisie. Cela signifie que, si toutes les valeurs possibles de la dernière variable choisie ont été essayées sans succès, la recherche d'une fourmi se termine par un échec (pas de solution construite). Les résultats de ce travail montrent sur le problème d'ordonnancement de tâches que le couplage plus fort est meilleur. Notons que le couplage fort proposé n'est pas une approche complète.

Egalement, B. Meyer a proposé dans [ME04] une hybridation d'un algorithme ACO avec des techniques de propagation de contraintes pour résoudre le problème d'emploi du temps comportant à la fois des contraintes dures et une fonction objectif à optimiser. l'idée est d'utiliser la propagation de contraintes

pour permettre aux fourmis de construire des combinaisons satisfaisant toutes les contraintes dures, la phéromone étant utilisée pour guider la recherche par rapport à la fonction objectif [4].

3.5 Intensification versus diversification de la recherche

La résolution d'un problème avec une métaheuristique nécessite de trouver un compromis entre deux objectifs contradictoires : d'un coté, il est nécessaire d'intensifier la recherche autour des zones les plus prometteuses alors que de l'autre côté, il est nécessaire de diversifier la recherche pour découvrir de nouvelles zones de l'espace de recherche qui pourraient s'avérer porteuses de meilleures solutions.

En effet, une des idées clé de toutes les métaheuristiques est de maintenir, au cours de la recherche de solutions, un bon équilibre entre la diversification (aussi appelée exploration) et l'intensification (aussi appelée exploitation) de la recherche.

Le problème à résoudre pour arriver à un bon équilibre entre ces deux objectifs contradictoires est de savoir, pendant la recherche de solutions, quel est le moment idéal pour intensifier la recherche et le moment idéal pour diversifier la recherche.

Chacune des métaheuristiques a sa propre façon de gérer l'équilibre entre l'intensification et la diversification de la recherche.

Dans les métaheuristiques basées sur la recherche locale, l'intensification et la diversification de la recherche sont gérées par les règles d'acceptations des nouvelles affectations. Par exemple, dans l'algorithme de recuit simulé, il suffi d'utiliser une grande valeur du paramètre température T pour que la diversification de la recherche soit favorisée, car avec une grande valeur de T, un plus grand nombre d'affectations non améliorantes seront acceptées. A l'inverse, plus la valeur du paramètre T est petite, plus la recherche est intensifiée autour des meilleures affectations.

Dans les métaheuristiques évolutionnaires comme les algorithmes génétiques, l'intensification et la diversification de la recherche sont essentiellement gérées par les opérateurs de sélection ; de croisement et de mutation. Dans les EDA, l'équilibre intensification/diversification est fortement dépendant de la distribution de probabilités utilisée, tandis que, dans les algorithmes ACOs, il est géré par les différents paramètres de ACO comme nous allons le voir dans la sous-section suivante.

3.5.1 Contrôle de l'intensification et la diversification dans ACO

Les deux paramètres principaux de l'algorithme ACO qui lui permettent de contrôler le rôle de la phéromone vis-à-vis de la balance entre l'intensification et la diversification de la recherche sont ρ et α.

4. Les travaux de B. Meyer, à notre connaissance, sont les seuls travaux où les techniques de propagation de contraintes ont été incorporées dans un algorithme ACO

Généralement, dans ACO, la phéromone est mise-à-jour après chaque cycle en déposant une quantité de phéromone sur les composants des meilleures affectations. Cela a pour effet d'attirer les fourmis des prochains cycles vers les zones de l'espace de recherche qui contiennent des affectations de bonne qualité.

Les deux paramètres ρ et α interviennent à des instants différents de l'algorithme ACO. Le paramètre ρ est utilisé lors de la mise-à-jour de la phéromone tandis que le paramètre α est utilisé lors de l'exploitation de la phéromone.

A chaque cycle, les traces de phéromones sont évaporées en les diminuant d'une quantité qui dépend de la valeur de ρ. Cette évaporation couplée avec le processus de dépôt de phéromone, va faire en sorte, au fil de temps, que certains composants d'affectation vont être distingués des autres et vont être souvent sélectionnés. Si cette distribution est trop forte, les fourmis vont se retrouver dans une situation de stagnation où toutes les fourmis construisent les mêmes affectations. Dans ce dernier cas, on dit que l'algorithme a convergé. Par conséquent, le paramètre ρ a une forte influence sur la vitesse de convergence. Plus ρ est faible plus lente est la convergence.

Tandis que le paramètre ρ influence la vitesse d'évaporation de la phéromone, le paramètre α détermine le degré de la sensibilité des fourmis par rapport aux traces de phéromones. Plus la valeur de α est élevée, plus les différences entre les traces de phéromones sont répercutées sur les choix des fourmis.

En résumé, pour favoriser l'intensification de la recherche, il faudrait utiliser une valeur élevée de α et de ρ. En revanche, pour favoriser la diversification de la recherche, il faudrait utiliser une petite valeur de α et de ρ.

En plus des deux paramètres α et ρ, l'intensification et la diversification de la recherche dans les algorithmes ACO sont également dépendant des paramètres suivants :

- τ_0 : la valeur initiale des phéromones influence l'équilibre intensification/diversification d'un algorithme ACO. En effet, si une fourmi dépose, à la fin de la première itération, une quantité de phéromone très élevée par rapport à τ_0 sur certaines traces phéromonales, alors, l'algorithme va rapidement intensifier la recherche autour des affectations contenant les composants associés à ces traces phéromonales. De l'autre côté, si τ_0 est très élevé par rapport aux quantités de phéromones que les fourmis déposent, alors, beaucoup d'étapes d'évaporations sont nécessaires pour qu'il y ait des différences exploitables entre les quantités de phéromones accumulées et donc, pendant toutes ces étapes, la recherche est diversifiée.
- le nombre de fourmis : correspond aux nombre d'affectations construites en une itération de l'algorithme. Plus le nombre de fourmis est grand, plus la recherche est intensifiée (resp. diversifiée) si les quantités de phéromones déposées sur les traces phéromonales sont différentes (resp. équilibrées).
- choix des affectations à récompenser : Cela joue sur la rapidité de la convergence de l'algorithme. Plus les affectations à récompenser se ressemble (resp. différentes) plus la recherche est intensifiée (resp. diversifiée). Par

exemple, si l'on choisit de ne récompenser que la meilleure affectation construite depuis le début de l'algorithme et celle-là n'a pas été améliorée depuis un grand nombre de cycles, alors, la recherche va être intensifiée autour de cette affectation. En revanche, si l'on choisit de récompenser toutes les affectations du dernier cycle, alors la recherche sera plus diversifiée.

En fonction de la variante utilisée, d'autres paramètres peuvent influencer sur l'équilibre intensification/diversification d'un algorithme ACO. Par exemple, dans la variante $\mathcal{M}ax - \mathcal{M}in$ Ant System, la différence entre les bornes τmin et τmax influence l'équilibre intensification/diversification de la recherche. En effet, si cette différence est très petite, alors, quoi qu'il arrive, la recherche ne sera pas suffisamment intensifiée. En revanche, si elle est très grande, alors, dans la plupart du temps, la recherche sera diversifiée.

Il est à noter que les bonnes valeurs des paramètres d'un algorithme ACO qui garantissent un bon équilibre entre l'intensification et la diversification de la recherche sont généralement dépendantes les unes des autres.

3.5.2 Paramétrage des métaheuristiques

Le succès des métaheuristiques est donc dépendant de leurs capacités à maintenir un bon niveau d'équilibre entre l'intensification et la diversification de la recherche. Cet objectif est réalisable à travers le bon réglage de leurs paramètres.

Le réglage de ces paramètres est un problème délicat, car il faut choisir entre deux tendances contradictoires : si nous choisissons des valeurs de paramètres qui favorisent la diversification, la qualité de la solution obtenue est souvent très bonne, mais au prix d'une convergence plus lente, et donc d'un temps de calcul plus long. Si par contre nous choisissons des valeurs qui favorisent l'intensification, l'algorithme sera amené à trouver de bonnes solutions plus rapidement, mais souvent sans converger vers les meilleures solutions ; il convergera vers des valeurs sous-optimales.

Les valeurs optimales des paramètres dépendent à la fois de l'instance du problème à traiter et du temps alloué à sa résolution. Pour améliorer le processus de recherche vis-à-vis de la dualité intensification/diversification, plusieurs chercheurs ont proposé des méthodes de recherche de bonnes valeurs des paramètres.

De manière générale, pour trouver les bonnes valeurs des paramètres nous avons le choix entre deux méthodes différentes. La première méthode consiste à apprendre les bonnes valeurs des paramètres dynamiquement durant la recherche des solutions. Cette méthode est appelé méthode "ON-Line". La deuxième méthode consiste à apprendre les bonnes valeurs des paramètres sur un échantillon d'instances représentatif d'un problème et d'utiliser ces valeurs pour résoudre des nouvelles instances. Cette deuxième méthode est appelé méthode "OFF-Line".

Chacune des deux méthodes, ON-Line et OFF-Line, a ses avantages et ses inconvénients. La méthode ON-Line peut trouver les bonnes valeurs des paramètres pour une instance particulière d'un problème. En revanche, le temps de

calcul utilisé pour apprendre les paramètres durant une exécution peut être important de tel sorte que les performances de l'algorithme sont détériorées. De son coté, la méthode OFF-Line n'introduit pas du temps de calcul supplémentaire car les paramètres sont fixés au début d'une exécution. En revanche, les valeurs des paramètres utilisées pour résoudre une nouvelle instance peuvent s'avérer être les plus mauvaises, car, l'instance en cours de résolution peut être très différente des instances utilisées lors de l'apprentissage des paramètres en mode OFF-Line.

Il existe de nombreux travaux relatifs aux approches qui adaptent dynamiquement des paramètres pendant le processus de recherche.

Battiti et al [BBM08] ont proposé d'exploiter l'historique de la recherche pour adapter automatiquement et dynamiquement les valeurs de paramètres, donnant ainsi naissance aux approches dites "réactives". Par exemple, Battiti et Protasi ont proposé dans [BP01] d'employer l'information de ré-échantillonnage afin d'adapter dynamiquement la longueur de la liste taboue dans une recherche taboue : quand une affectation est recalculée, la longueur de la liste est augmentée ; quand il n y a pas eu de ré-échantillonnage depuis un grand nombre de mouvements elle est diminuée.

Il existe également des approches réactives pour les algorithmes ACO. En particulier, Randall a proposé dans [Ran04] d'utiliser ACO pour adapter dynamiquement des paramètres d'un algorithme ACO. Dans ce travail, les paramètres sont appris au niveau des fourmis de sorte que chaque fourmi fait évoluer ses propres paramètres et considère le même paramétrage pendant une construction de solution. Dans ce modèle, chaque fourmi est considérée comme un agent autonome qui améliore ses performances au fur et à mesure qu'il acquiert de la connaissance sur le problème.

Frace [BSPV02] est l'une des approches OFF-Line les plus connues. Elle propose d'apprendre les bonnes valeurs des paramètres d'un algorithme en exécutant celui-ci sur un échantillon d'instances représentatif d'un problème et en utilisant plusieurs jeux de paramétrages. A fur et à mesure des exécutions, *Frace* élimine les jeux de paramétrages les moins performants en utilisant des mesures statistiques de telle sorte qu'à la fin de toutes les exécutions, les jeux des paramètres restant dans la course soient les meilleurs.

3.6 Conclusion

Dans ce chapitre, nous avons passé en revue les éléments de base de quelques unes des métaheuristiques les plus connues et les plus utilisées. Nous avons vu que les métaheuristiques se divisent généralement en deux types. Les métaheuristiques perturbatives et les métaheuristiques constructives. Nous avons également abordé les notions d'intensification et de diversification de la recherche et enfin, nous avons parlé, de manière générale, des problèmes posés par le paramétrage des métaheuristiques.

Pendant notre exposé sur ces métaheuristiques, nous avons essayé du pointer

de doigt l'idée utilisée par chacune d'elles pour éviter de tomber dans le piège de l'optimum local et nous avons essayé de monter comment elles arrivent à maintenir un équilibre entre l'intensification et la diversification de la recherche.

Maintenir un bon équilibre entre la diversification et l'intensification de la recherche est un vrai problème, car les valeurs des paramètres de ces métaheuristiques qui permettent un tel équilibre sont très fortement dépendantes du problème traité et de plus, elles peuvent varier d'une instance à une autre instance du même problème.

Au delà de leurs capacités de navigation dans l'espace de recherche, les métaheuristiques sont généralement pensées dans l'optique de l'optimisation d'une (ou plusieurs) fonction(s) objectif(s). Plus précisément, la motivation principale derrière leur développement est de pouvoir sélectionner une solution qui soit la meilleure (ou presque) parmi plusieurs autres solutions sans pour autant tester la performance de chacune d'elles. Cet objectif est très difficile à atteindre. Etant donné que le but principal de ces métaheuristiques est d'optimiser l'objectif, il est très difficile de les utiliser pour résoudre un problème qui, en plus de cet objectif, présente des contraintes dures à satisfaire.

Le sujet du prochain chapitre se focalise sur la question de l'application des métaheuristiques sur des problèmes d'optimisation combinatoire contenant de telles contraintes.

Utilisation des contraintes dans les métaheuristiques

4

Sommaire

Dans ce chapitre, nous donnons quelques unes des techniques qui permettent d'inclure les contraintes dans la procédure de résolution. D'abord, nous décrirons des techniques basées sur la pénalisation des affectations qui violent des contraintes ; puis nous décrirons des techniques basées sur la réparation des affectations qui violent des contraintes pour les transformer en solutions ; et nous donnons également des techniques qui, lors de la résolution d'un problème, séparent les contraintes de la fonction objectif et enfin, nous donnons quelques techniques de gestion des contraintes dans les algorithmes de recherche locale.

Les métaheuristiques ont été appliquées avec succès sur différents problèmes. Cependant, la plupart ont été testées sur des problèmes sous contraints. Par exemple, pour le TSP, la difficulté est de trouver le plus court chemin et non pas de trouver un chemin. Dans ce problème, le graphe considéré est généralement complet et la seule contrainte à satisfaire est que chaque ville soit visitée exactement une fois. Dans la pratique, les problèmes d'optimisation sont généralement sujets à des contraintes qui sont, dans beaucoup de cas, difficiles à satisfaire. Pour ce genre de problèmes, la difficulté n'est pas seulement d'optimiser la fonction objectif mais aussi de trouver des solutions satisfaisant toutes les contraintes.

Par exemple, dans la réalité, un TSP peut contenir des contraintes additionnelles qui rendront sa résolution plus compliquée. Par exemple, le voyageur ne devrait jamais visiter une ville qui est dans un ensemble de villes A juste après qu'il ait visité une ville qui est dans un certain ensemble de villes B ; ou le voyageur devrait impérativement avoir visité une certaine ville i avant de visiter la ville j. Pratiquement parlant, ce genre de contraintes peut rendre la résolution du problème encore plus dure.

Résoudre des problèmes d'optimisation contenant des contraintes difficiles nécessite donc de trouver un moyen de les incorporer dans la procédure de résolution elle-même. Ci-dessous, nous donnons quelques-unes de ces approches.

4.1 Gestion des contraintes dans les algorithmes génétiques

4.1.1 Techniques de pénalisation

Les techniques de pénalisation sont généralement utilisées dans les algorithmes évolutionnaires et elles sont basées sur la manière dont la fonction de fitness (fonction objectif) est évaluée. En effet, avec les techniques de pénalisation, la fonction objectif $\mathcal{F}(x)$ à optimiser est généralement transformée de la façon suivante : $\mathcal{F}itness(\mathcal{S}) = \mathcal{F}(\mathcal{S}) + \lambda.\mathcal{G}(\mathcal{S})$ où $\mathcal{G}(\mathcal{S})$ est une fonction positive qui mesure la quantité et/ou le volume des contraintes violées et λ le coefficient de pondération. Plusieurs travaux de recherches ont été menées pour concevoir des techniques permettant d'inclure les contraintes dans la fonction objectif [RPLH89, Coe02]. Loin d'être exhaustifs, ci-dessous, nous citons quelque uns de ces travaux.

La technique "death penalty"

Dans cette approche, les affectations incohérentes vont voir leur fonction de fitness affectée à 0. Par conséquent, du point de vue de la fonction de fitness, une affectation qui viole un très grand nombre de contraintes va être considérée de la même manière qu'une affectation qui ne viole qu'une seule contraintes. En raison de sa simplicité, cette approche est très populaire dans la communauté des algorithmes évolutionnaire. Cependant, elle est efficace lorsque l'espace des

solutions constitue une large part de l'espace de recherche car si ce n'est pas le cas, il serait très difficile de construire des affectations cohérentes.

La technique de pénalisation statique

Dans cette approche, le temps (nombre de générations) écoulé depuis le début de l'exécution des programmes n'est pas pris en compte dans les formules de pénalisation des affectations incohérentes. Une affectation incohérente appartenant à la première génération de solutions sera pénalisée de la même manière que si elle appartenait à la dernière génération de solutions. En d'autres termes, on n'exige pas des affectations d'être de plus en plus performantes au fil du temps. Plusieurs auteurs ont utilisé des fonctions de pénalité de type statiques (ne changeant pas dans le temps). Parmi eux nous citons [Mic96, HS96, MQ98].

La technique de pénalisation dynamique

Dans cette approche, les affectations incohérentes sont pénalisées de plus en plus sévèrement en fonction du temps écoulé depuis le début du programme. En d'autres termes, on exige de construire de meilleures affectations au fil du temps. Parmi les travaux où cette technique est utilisée, nous citons [KP98, JH94, MA94].

La techniques de pénalisation adaptative

Cette technique, essaye d'adapter la fonction de pénalité suivant ce qui ce passe au cours d'un certain ensemble de générations (généralement les k dernières générations). Donc, par définition, cette techniques fait partie des techniques de pénalisation dynamique. Cependant, dans les techniques adaptatives la formule de pénalité peut changer. Par exemple, en fonction des k dernières générations, si les affectation construites sont : - incohérentes, alors la pénalité devient plus sévère ; - cohérente, alors la pénalité devient moins sévère - sinon, la formule de pénalité reste inchangé [HAB97]. Sans être exhaustif, parmi les auteurs qui ont proposé des fonctions de pénalité adaptatives, nous citons : [ST93, CS96, CST96, NS97, YGIT95, GC96].

Avantages et inconvénients

Les méthodes de pénalisation ont principalement deux avantages : elles ont une forme générique applicable à différents problèmes et elles sont relativement faciles à implémenter. Cependant, de manière générale, elles souffrent de la multiplicité des paramètres utilisés dans les formules de fitness. Trouver des bonnes valeurs à ces paramètres est en lui même un problème d'optimisation difficile. En pratique, un mauvais choix de ces paramètres peut soit empêcher de trouver une solution satisfaisant toutes les contraintes (le cas d'une sous pénalisation) soit de converger vers une solution sous-optimale (cas d'une sur-pénalisation).

En effet, d'un côté, si la fonction de fitness distingue avec des écarts très larges les affectations cohérentes de celle incohérentes et si la génération d'individus en cours contient quelques affectations cohérentes et d'autres incohérentes, alors l'algorithme risque de ne considérer que les affectations cohérentes lors de la construction de la prochaine génération. Ce processus peut mener l'algorithme à converger sur des solutions sous optimales très mauvaises, car une fois qu'il a trouvé une région de l'espace de recherche contenant des affectations cohérentes, il a du mal à en sortir pour visiter d'autres. Ceci est d'autant plus vrai que, dans l'espace de recherche, passer d'une région faisable à une autre région faisable peut nécessiter de passer par plusieurs régions infaisables. De l'autre côté, si la fonction de fitness distingue avec des écarts très petits les affectations cohérentes de celles incohérentes, ou encore, si la fonction de fitness ne fait pas la distinction entre les affectations incohérentes du point de vue du nombre et du degré de contraintes violées, alors l'algorithme de recherche peut être piégé dans un sous espace de recherche ne contenant pas de solutions.

4.1.2 Méthode de réparation

Dans beaucoup de problèmes d'optimisation combinatoire, les solutions infaisables peuvent être facilement transformées en solutions faisables en les modifiants. Cette méthode est appelée la méthode de réparation.

Réparation des solutions

La réparation des solutions infaisables peut être utilisée pour leur évaluation. Transformer une solution infaisable s en une autre solution faisable $s^{'}$ peut mener au remplacement (avec une certaine probabilité) de la solution s par $s^{'}$ dans la population d'individus. Dans [LV90, LP91], les auteurs ont montré sur un ensemble de problèmes contraints que les méthodes de réparations peuvent surpasser d'autres méthodes de gestion des contraintes en matière de qualité et rapidité. Notons que dans [LV90, LP91], les auteurs ont choisi d'utiliser la procédure de réparation des solutions juste pour l'évaluation de la fonction objectif et ne jamais faire des remplacements dans la population d'individus après l'étape de réparation. D'autres auteurs ont utilisé la technique de réparation pour gérer les contraintes dans les métaheuristiques [Müh92, OD94, TS95].

Pour procéder à la réparation d'une solution, on utilise généralement l'algorithme "greedy" basé sur une heuristique que l'on sait efficace pour le problème à résoudre. Donc, le succès de cette méthode repose en grande partie sur le savoir faire de l'utilisateur qui devra fournir une heuristique efficace.

Avantages et inconvénients

Cette méthode peut se révéler très efficace sur les problèmes pour lesquels la transformation d'une solution infaisable en une solution faisable est facile et ne

consomme pas beaucoup de temps du calcul. Cependant, elle reste une méthode dépendante du problème à résoudre car l'heuristique utilisée par la procédure de réparation est une heuristique ad-hoc.

4.1.3 Séparation des contraintes de la fonction objectif

Il existe plusieurs approches qui gèrent séparément les contraintes et la fonction objectif. Ci-après, nous allons citer quelques-unes de ces approches.

Méthode de coévolution

Paredis a proposé dans [Par94] un modèle basé sur deux populations. La première population représente les contraintes qui devraient être satisfaites et qui ne changent pas dans le temps. Une deuxième population qui contient les affectations construites. La fitness d'une affectation dans la deuxième population est élevé si elle satisfait beaucoup de contraintes. La fitness d'une contrainte dans la première population est élevé si elle est violée par plusieurs affectations de la deuxième population. De plus, dans cette approche, l'évaluation des individus (contraintes et affectation) est basée sur les k dernières observations de leurs finesses. L'idée derrière cette approche est de détecter les contraintes qui sont difficiles à satisfaire pour que la procédure de recherche se concentre d'avantage sur elles. L'inconvénient de cette approche est que si toutes les contraintes (ou la plupart des contraintes) ont la même difficulté la procédure de recherche peut stagner.

Méthode favorisant les affectations cohérentes

De manière générale, dans cette approche, la comparaison de deux affectations incohérentes est uniquement basée sur la quantité de violation de contraintes et une affectation cohérente est toujours évaluée meilleure qu'une affectation incohérente [Deb01, PS93]. Notons que cette méthode se rapproche des méthodes par pénalisation vue précédemment, cependant, ici nous insistant sur le fait que la quantité de contraintes violées par une affectation n'est pas forcément incluse dans la fonction de fitness.

méthode de gestion séquentielle des contraintes

Dans [SX93], Schoenauer et Xanthakis ont proposé un algorithme qui gère les contraintes de la manière suivante : A chaque itération, l'algorithme choisit une contrainte c_j (cette contrainte est donnée par un compteur de contraintes j égal à un au début de l'algorithme et puis, il est incrémenté d'une unité à chaque itération) et tente de minimiser la violation de cette contrainte. Le comportement de cet algorithme à la génération j est tel que les individus qui violent l'une des contraintes $(c_1, c_2, ..., c_j)$ seront éliminés de la population. Une étape s'arrête lorsque un certain pourcentage d'individus satisfont la contrainte c_j. Le but de

cet algorithme est de satisfaire les contraintes de manière séquentielle (une par une dans l'ordre choisi). Lors de la dernière étape, l'algorithme utilise la méthode de "'death Panalty" pour n'accepter que les solutions faisables. L'inconvénient de cet algorithme est que l'ordre dans lequel les contraintes sont sélectionnées peut influencer non seulement sur les performances en matière de la qualité des solutions mais aussi sur le temps de calcul.

4.2 La recherche locale et les contraintes

Comme nous l'avons vu dans la section 3.1.1, les algorithmes de recherche locale ont été utilisés avec succès pour résoudre des problèmes d'optimisation. Ces algorithmes partent d'une affectation complète sur laquelle ils appliquent des mouvements (des changements), un par un, pour la transformer en une autre affectation complète. Ces mouvements sont effectués en utilisant une heuristique qui choisi, pour une affectation, une ou plusieurs variables puis une ou plusieurs valeurs à affecter à ces variables.

Les algorithmes de recherche locale, lorsqu'ils sont appliqués à des problèmes contenant des contraintes dures n'utilisent pas la propagation de contraintes mais évaluent le nombre de contraintes violées et tentent de minimiser ce nombre. Des travaux hybridant recherche locale et recherche arborescente ainsi que propagation de contraintes ont été menée. La plupart des auteurs ont proposé des couplages faibles où les différents algorithmes tournent séquentiellement ou en parallèle (ex : [KD95, Sch97]). En revanche, dans [JL02], les auteurs ont proposé un couplage fort où la recherche locale ne considère plus uniquement les affectations complètes mais aussi les affectations partielles. Ils ont proposé une approche générique appelée *decision − repair* qui à partir d'une affectation partielle (cette affectation peut être vide) effectue un filtrage en propageant les contraintes du problème et si l'algorithme de propagation se termine avec succès (aucun domaine n'est devenu vide) elle étend l'affectation partielle à la façon des algorithmes de retour-arrière (i.e. choisit une variable non affectée en utilisant une heuristique de choix de variables et choisit une valeur pour cette variable en utilisant une heuristique de choix de valeurs). Par contre, si l'algorithme de propagation détecte un échec, elle choisit une variable parmi celles déjà affectées dans l'affectation en cours en utilisant une heuristique donnée, pour la rendre non affectée. Dans ce dernier cas, les informations concernant les variables et leur valeur qui ont causé l'échec seront enregistrés et utilisées notamment pour prouver l'incohérence d'une affectation partielle. Cette approche a démontré son efficacité sur le problème "Open JobShop" [JL02].

Pour instancier un algorithme de type *decision − repair*, il faut fournir : (1) Deux heuristiques, une pour le choix de variables et une autre pour le choix de valeurs, qui sont utilisées pour étendre une affectation partielle (2) Une procédure de filtrage qui permet de propager et d'enlever les valeurs incohérentes des variables non encore affectées à chaque affectation d'une variable (3) Une autre

heuristique pour le choix de la variable à remettre comme étend non affectée lorsque l'algorithme de filtrage détecte un échec. Dans [PV04], les auteurs ont étudié quelques instanciations de cette approche et ils les ont appliquées sur des problèmes de satisfaction de contraintes générés aléatoirement. Les résultats sont très intéressants puisque dans beaucoup de cas, ils surpassent les algorithmes basés sur le retour-arrière intelligent comme CBJ.

Egalement, dans les algorithmes de recherche locale, certaines contraintes du problème peuvent être prises en compte lors du calcul de voisinage d'une affectation de façon à ne générer que des affectations qui satisfont ces contraintes. Par exemple, pour le TSP, le voisinage d'une solution (un tour hamiltonien) peut être défini en échangeant l'ordre de visite de deux villes, ainsi, les solutions résultantes sont également des tours hamiltoniens.

4.3 Gestion des contraintes dans les algorithmes ACO

Le schéma général de la métaheuristique ACO est essentiellement orienté vers l'optimisation de la fonction objectif. Du fait, la gestion des contraintes avec un algorithme ACO n'est pas une tâche facile. La plupart des algorithmes de type ACO proposés dans l'état de l'art ont été appliqués sur des problèmes d'optimisation qui, généralement, ne contiennent pas de contraintes difficiles à satisfaire : lors de la construction d'une affectation, on suppose généralement qu'il est toujours possible d'étendre l'affectation partielle sans violer de contraintes de sorte que les fourmis ne construisent que des affectations cohérentes. Ce manque dans le schéma de ACO a conduit certains auteurs à proposer des algorithmes génériques permettant de traiter les contraintes. Ci-après, nous allons porter notre attention sur trois de ces algorithmes à savoir : $Ant - Solver$, $Ant - CarSequencing$ et $CPACS$. Le premier [1] est un algorithme qui résout les problèmes de satisfaction de contraintes, le deuxième $Ant - CarSequencing$ est un algorithme ACO qui résout le problème d'ordonnancement de voitures tandis que le troisième est un algorithme où le mécanisme de propagation de contraintes est incorporé au sein de l'algorithme ACO pour la résolution des problèmes d'ordonnancement des tâches.

4.3.1 $Ant - Solver$

L'algorithme $Ant - Solver$ est dédié à la résolution des problèmes de satisfaction de contraintes et il est décrit dans l'algorithme 2. Ci-dessous, nous rappelons ses principaux fondamentaux. Une description plus détaillée peut être trouvée dans [Solo2].

1. $Ant - Solver$ sera présenté plus en détaille, car nous allons le reprendre dans le chapitre 7 pour en proposer une extension et traiter les problèmes $MaxCSP$.

Algorithme 2 : Ant Solver

Entrées : un CSP (X, D, C) et un ensemble de paramètres
$\{\alpha, \beta, \rho, \tau_{min}, \tau_{max}, nbAnts, maxCycles\}$
Sorties : Une affectation complète de (X, D, C)

```
1 Initialisation à τ_max des traces de phéromones associées avec (X, D, C)
2 répéter
3     pour k dans 1..nbAnts faire
4         Construire une affectation complète A_k
5         Optionnellement : Améliorer A_k par la recherche locale
6     Evaporation des traces de phéromones
7     Renforcement des traces de phéromones
8 jusqu'à cost(A_i) = 0 pour un i ∈ {1..nbAnts} ou maxCycles atteint ;
9 return l'affectation qui a le coût le plus petit
```

Traces de phéromone associées à un CSP (X, D, C) (ligne 1).

Une trace de phéromone $\tau(< x_i, v >)$ est associée à chaque couple variable/valeur $\langle x_i, v \rangle$ tel que $x_i \in X$ et $v \in D(x_i)$. Intuitivement, cette trace de phéromone représente l'intérêt appris d'affecter la valeur v à la variable x_i. Ainsi qu'il est proposé dans [SHoo], les traces de phéromone sont bornées entre τ_{min} et τ_{max}, et sont initialisées à τ_{max}.

Construction d'une affectation par une fourmi (ligne 4) :

A chaque cycle (lignes 2-8), chaque fourmi construit une affectation : à partir d'une affectation vide $\mathcal{A} = \emptyset$, elle ajoute itérativement des couples variable/valeur à $\mathcal{A}$ jusqu'à ce que $\mathcal{A}$ forme une affectation complète. A chaque étape, pour choisir un couple variable/valeur, la fourmi choisit d'abord une variable x_j qui n'est pas encore affectée dans $\mathcal{A}$. Ce choix est réalisé avec l'heuristique `min-domain` : la fourmi choisit une variable qui a le plus petit nombre de valeurs cohérentes vis-à-vis de l'affectation partielle $\mathcal{A}$ en cours de construction. Puis, la fourmi choisit une valeur $v \in D(x_j)$ à affecter à x_j selon la probabilité suivante :

$$p_{\mathcal{A}}(<x_j, v>) = \frac{[\tau(<x_j, v>)]^{\alpha} \cdot [\eta_{\mathcal{A}}(<x_j, v>)]^{\beta}}{\sum_{w \in D(x_j)} [\tau(<x_j, w>)]^{\alpha} \cdot [\eta_{\mathcal{A}}(<x_j, w>)]^{\beta}}$$

où

- $\tau(<x_j, v>)$ est la trace de phéromone liée à $(<x_j, v>)$,
- $\eta_{\mathcal{A}}(<x_j, v>)$ est le facteur heuristique et est inversement proportionnel au nombre de nouvelles contraintes violées par l'affectation de la valeur v à la variable x_j, c.-à-d., $\eta_{\mathcal{A}}(<x_j, v>)) = 1/(1 + cost(\mathcal{A} \cup \{<x_j, v>\}) - cost(\mathcal{A}))$, où $cost(\mathcal{A})$ est le nombre de contraintes violées par l'affectation $\mathcal{A}$.
- α et β sont les paramètres qui déterminent les poids relatifs des facteurs.

Amélioration locale des affectations (ligne 5) :

Lorsqu'une affectation complète a été construite par une fourmi, elle peut être améliorée en exécutant une recherche locale, c.-à-d., en changeant itérativement quelques affectations variable/valeur. Différentes heuristiques peuvent être employées pour choisir la variable à modifier et la nouvelle valeur à lui affecter. Pour le choix de valeur, $Ant - Solver$ utilise l'heuristique appelée `min-conflicts` [MJPL92] : cette heuristique choisit aléatoirement une variable impliquée dans une contrainte violée et l'affecte avec la valeur qui minimise le nombre de contraintes violées. Ces améliorations locales sont réitérées jusqu'à ce que la solution ne puisse plus être améliorée.

Mise à jour des traces de phéromone (lignes 6-7) :

Lorsque chacune des fourmis de la colonie a construit une affectation, éventuellement améliorée par la procédure de recherche locale, la quantité de phéromone associée à chaque couple variable/valeur est mise à jour selon la métaheuristique ACO. Dans un premier temps, toutes les traces de phéromone sont uniformément diminuées (ligne 6) en les multipliant par $(1 - \rho)$ afin de simuler le phénomène d'évaporation qui permet aux fourmis d'intensifier la recherche sur les affectations les plus récentes. Puis, la phéromone est ajoutée sur chaque couple variable/valeur appartenant à la meilleure affectation du cycle (ligne 7) afin d'attirer les fourmis vers les zones correspondantes de l'espace de recherche. Plus précisément, soit $\mathcal{A}_{best}$ la meilleure affectation construite durant le dernier cycle, pour chaque couple $< x_i, v_i > \in \mathcal{A}_{best}$, la quantité de phéromone associée, $\tau(< x_i, v_i >)$, est incrémentée de $\frac{1}{cost(\mathcal{A}_{best})}$ de sorte que moins l'affectation viole de contraintes, plus ses composants reçoivent de phéromone.

4.3.2 Les fourmis et le problème d'ordonnancement de voitures

Dans [Solo8], C. Solnon a traité le problème d'ordonnancement de voitures avec ACO. Ce problème, dans sa forme générale, est un CSP et pourrait donc être résolu de façon générique par l'algorithme $Ant - Solver$ décrit précédemment. Dans ce travail, Solnon propose d'utiliser deux structures de phéromones qui sont plus dédiées au problème d'ordonnancement de voitures. Une pour apprendre les bonnes valeurs des variables et une autre pour détecter les voitures les plus dures à ordonnancer (i.e. : les voitures demandant les options les plus dures). La deuxième structure de phéromones joue le rôle de l'heuristique qui dans ce cas-là est une heuristique apprise sur le tas durant l'exécution de l'algorithme. L'algorithme proposé construit des affectations complètes incohérentes. Une solution est trouvée lorsqu'une affectation complète est construite sans violer aucune contrainte. En résumé, les contraintes sont gérées en considérant le CSP comme un problème d'optimisation dans lequel la fonction objectif est de minimiser le nombre de contrainte violées.

4.3.3 *CPACS*

Comme nous l'avons vu dans le chapitre 3 section 3.4.4, B. Meyer a proposé, dans [Meyo8], deux algorithmes hybrides où ACO a été couplé avec la PPC. Nous avons dit que le meilleur algorithme été celui où la propagation de contraintes a été incorporée dans ACO de telle sorte qu'une fourmi revient en arrière (backtrack) lorsqu'une affectation d'une valeur à une variable donnée échoue. Toutefois, la procédure de retour en arrière a été limitée au niveau de la dernière variable choisie. Par conséquent, si toutes les valeurs possibles de la dernière variable choisie ont été essayées sans succès, alors, la recherche d'une fourmi se termine par un échec (pas de solution construite). En résumé, dans ce travail, Meyer propose de ne pas accepter la construction d'affectations incohérentes.

4.4 Conclusion

Dans ce chapitre, nous avons fait un tour d'horizon sur les différentes manières de gérer les contraintes dans les métaheuristiques. Nous avons cité quelques techniques utilisées dans les algorithmes génétiques, la recherche locale et les algorithmes ACO.

Deuxième partie

Combinaison de l'optimisation par colonie de fourmis et la programmation par contraintes pour la résolution de problèmes combinatoires

Introduction à la deuxième partie

Dans la première partie de cette thèse, nous avons fait un tour d'horizon sur les concepts de bases des problèmes d'optimisation combinatoire (COP) ; des algorithmes de résolution exactes et métaheuristiques et également, nous avons parlé de quelques techniques utilisées pour la gestion des contraintes dans les métaheuristiques.

Dans cette deuxième partie, nous allons présenter nos contributions à l'état de l'art sur l'intégration des algorithmes d'optimisation par colonies de fourmis dans un langage de programmation par contraintes.

De manière générale, nos motivations principales derrières les développements décrits dans les chapitres qui suivent ont tous un facteur en commun que nous pouvons résumer par les deux constatations suivantes :

- La programmation par contraintes permet de décrire des problèmes combinatoires dans des langages de haut niveau simples à utiliser. Cependant, les algorithmes de résolution intégrés à la plupart de ces langages ne sont pas capables de résoudre les instances difficiles en un temps acceptable, car ils sont souvent basés sur une approche de type "Branch & Propagate&Bound".
- La métaheuristique d'optimisation par colonies de fourmis a été utilisée avec succès pour résoudre de très nombreux problèmes d'optimisation combinatoires. Cependant, la majorité des algorithmes ACO sont dédiés à la résolution de problèmes spécifiques. Par conséquent, utiliser un algorithme ACO pour résoudre un nouveau problème autre que celui pour lequel il a été conçu, peut nécessiter un important travail de programmation.

Notre objectif principal est de développer des algorithmes hybrides entre les algorithmes de la PPC et les algorithmes de type ACO. Nos résultats sont décrits dans les trois prochains chapitres.

Dans le chapitre 5, nous décrivons l'intégration d'ACO à un langage de PPC pour résoudre des CSP. Dans le chapitre 6, nous introduisons une nouvelle approche couplant ACO et recherche complète pour la résolution des problèmes d'optimisation combinatoires. Dans le chapitre 7, nous introduisons une nouvelle stratégie qui permet d'adapter des paramètres de ACO de manière dynamique et adaptative.

Intégration de ACO dans la PPC pour la résolution des CSPs

5

Sommaire

Dans ce chapitre, nous allons explorer les possibilités d'intégration de la méta-heuristique d'optimisation par colonies de fourmis (ACO) dans le langage de programmation par contraintes IBM ILOG Solver. [1].

1. Les travaux présentés dans ce chapitre ont été partiellement publiés dans CPAIOR08 (à Paris) [KAS08a] et intégralement publiés dans la conférence ANTS08 (à Bruxelles)[KAS08b] et les JFPC08 (à Nantes)[KAS08c]

5.1 Motivations

Dans les chapitres précédents, nous avons vu que les métaheuristiques peuvent être appliquées aux problèmes d'optimisation combinatoires sous contraintes.

Dans ce chapitre, nous allons proposer une première intégration de l'algorithme d'optimisation par colonies de fourmis dans la bibliothèque IBM ILOG Solver. Cette intégration se distingue des travaux existant dans l'état de l'art par le fait qu'elle n'est pas dédiée pour le traitement d'un problème particulier mais pour traiter tout problème modélisé dans le langage de l'API d'IBM ILOG Solver ou plus généralement, tout problème écrit dans le format d'un CSP tel que nous l'avons décrit dans la section 1.1. Pour parvenir à une telle intégration, nous nous sommes appuyés sur les outils fournis dans l'API d'IBM ILOG Solver.

Les motivations principales qui nous ont poussé à proposer cette intégration trouvent leur origine dans le double constat suivant :

- d'une part, la programmation par contraintes (PPC) permet de décrire des problèmes combinatoires, qui peuvent être très complexes, de façon déclarative dans des langages de haut niveau. Ces langages permettent à l'utilisateur de se concentrer uniquement sur la forme du problème qu'il voudrait traiter. En effet, au moment de la modélisation d'un problème en utilisant la PPC, l'utilisateur peut s'abstraire de toute réflexion quant à la manière dont ce problème va être résolu. Cependant, comme nous l'avons dit au chapitre 2, les procédures de résolution intégrées à la plupart de ces langages sont traditionnellement basées sur une approche de type "Branch & Propagate". Par conséquent, elles ne sont pas toujours capables de résoudre les instances difficiles en un temps acceptable.
- d'autre part, la métaheuristique d'optimisation par colonies de fourmis a été utilisée avec succès pour résoudre de très nombreux problèmes d'optimisation combinatoires (voir la section 3.3). Cependant, la majorité des travaux sur ACO sont concentrés sur la conception d'algorithmes efficaces pour résoudre différents problèmes, et non sur l'intégration de ces algorithmes dans des langages déclaratifs. Par conséquent, concevoir un algorithme ACO pour résoudre un nouveau problème autre que celui pour lequel il a été conçu, peut nécessiter un important travail de programmation.

Nous allons explorer les possibilités de combinaison des avantages de la métaheuristique ACO et de la PPC pour proposer un algorithme hybride. Plus précisément, ce que l'on souhaite obtenir à travers cette combinaison est un algorithme capable de recevoir un problème $\mathcal{P}$ modélisé dans le langage de la PPC et de le résoudre de la façon la plus générique possible. Le succès d'une telle intégration dépend des deux points fondamentaux suivants :

- le langage doit permettre à l'utilisateur de décrire son problème facilement, de façon déclarative, sans avoir à entrer dans des détails liés à sa résolution ;

– la procédure de résolution de CSP intégrée au langage doit être capable de résoudre efficacement les problèmes ainsi modélisés.

Il est clair que ces deux points ne sont pas faciles à concilier, car, un algorithme qui résout un problème $\mathcal{P}$ est généralement d'autant plus efficace qu'il exploite les spécificités (les caractéristiques ou encore la structure de l'espace de recherche) de ce problème pour guider la recherche de solutions.

Pour notre intégration, nous utiliserons la bibliothèque de programmation par contraintes IBM ILOG Solver. Cette bibliothèque contient un grand nombre de méthodes permettant de déclarer des variables et des contraintes ; des procédures de vérification et de propagation sont associées aux contraintes et sont activées au fur et à mesure de la construction d'affectations. Par défaut, la stratégie de recherche de solution est basée sur une recherche exhaustive arborescente. Nous proposons de remplacer cette dernière par une recherche de solutions guidée par l'algorithme d'optimisation par colonies de fourmis.

Plusieurs algorithmes ACO ont été appliqués pour la résolution des CSPs. Cependant, la plupart de ces algorithmes accepte de construire des affectations qui violent des contraintes et leurs but est de minimiser le nombre de ces contraintes violées. L'algorithme que nous allons présenter dans ce chapitre fonctionne différemment. En effet, d'une part, c'est un algorithme générique, i.e., il n'est pas dédié à un problème particulier. D'autre part, il n'accepte pas de construire des affectations incohérentes. La suite de ce chapitre est consacrée à la présentation de cet algorithme nommé Ant-CP.

5.1.1 Principe d'une intégration d'ACO dans un langage de PPC

Nous proposons donc d'utiliser le formalisme de la PPC d'ILOG Solver pour décrire le problème à résoudre et de remplacer la procédure de recherche arborescente prédéfinie dans ILOG Solver par une recherche ACO. Ainsi, nous utilisons tout le travail fait par ILOG à la fois au niveau de la modélisation de problèmes et au niveau des algorithmes de vérification et de propagation de contraintes.

L'utilisation d'ILOG Solver pour la modélisation, la vérification et la propagation de contraintes nous impose d'adapter les algorithmes ACO proposés précédemment pour la résolution de CSP :

– dans les algorithmes introduits dans [Solo8, Solo2](voir section 4.3), chaque fourmi construit une affectation complète (telle que chaque variable est affectée à une valeur), cette affectation pouvant éventuellement être incohérente ; dans ce cas, la qualité des affectations construites dépend du nombre de contraintes violées et l'objectif des fourmis est de construire l'affectation complète violant le moins de contraintes ;
– dans notre nouvel algorithme Ant-CP, chaque fourmi construit une affectation cohérente (ne violant aucune contrainte), cette affectation pouvant éventuellement être partielle (certaines variables n'étant pas affectées) ; dans ce cas, la qualité des affectations construites dépend du nombre de variables

affectées et l'objectif des fourmis est de construire l'affectation cohérente affectant le plus grand nombre de variables.

Dans la section suivante, nous allons décrire l'algorithme *Ant-CP* que nous appliquons ensuite sur le problème d'ordonnancement de voitures.

5.2 Description d'*Ant-CP*

Algorithme 3 : *Ant-CP*

Entrées :
- Un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$,
- Une stratégie phéromonale $\Phi = (S, \tau, comp)$,
- Un facteur heuristique η,
- Un ensemble de paramètres $\{\alpha, \beta, \rho, \tau_{min}, \tau_{max}, nbAnts, maxCycles\}$

Sorties : Retourne une affectation cohérente (éventuellement partielle) pour $(\mathcal{X}, \mathcal{D}, \mathcal{C})$

1 Initialiser toutes les traces de S à τ_{max}
2 $\mathcal{A}_{best} \leftarrow \emptyset$
3 **répéter**
4 **pour** *chaque* $k \in 1..nbAnts$ **faire**
   `/* Construction d'une affectation partielle et cohérente` $\mathcal{A}_k$ `*/`
5 $\mathcal{A}_k \leftarrow \emptyset$
6 **répéter**
7 Choisir une variable $x_j \in \mathcal{X}$ telle que $x_j \notin var(\mathcal{A}_k)$
8 Choisir une valeur $v \in D(x_j)$ selon la probabilité $p(x_j, v) = \frac{[\tau(\mathcal{A}_k, x_j, v)]^\alpha [\eta(\mathcal{A}_k, x_j, v)]^\beta}{\sum_{w \in \mathcal{D}(\S_|)} [\tau(\mathcal{A}_k, x_j, w)]^\alpha [\eta(\mathcal{A}_k, x_j, w)]^\beta}$
9 Ajouter $\langle x_j, v \rangle$ à $\mathcal{A}_k$
10 Propager les contraintes
11 **jusqu'à** $var(\mathcal{A}_k) = \mathcal{X}$ *ou Echec* ;
12 **si** $card(\mathcal{A}_k) \geq card(\mathcal{A}_{best})$ **alors** $\mathcal{A}_{best} \leftarrow \mathcal{A}_k$
13 **si** $var(\mathcal{A}_{best}) = X$ **alors retourner** $\mathcal{A}_{best}$
   `/* Mise-à-jour de la phéromone */`
14 Evaporer chaque trace de S en la multipliant par $(1 - \rho)$
15 **pour** *chaque affectation partielle* $\mathcal{A}_k \in \{\mathcal{A}_1, \ldots, \mathcal{A}_{nbAnts}\}$ **faire**
16 **si** $\forall \mathcal{A}_i \in \{\mathcal{A}_1, \ldots, \mathcal{A}_{nbAnts}\}, card(\mathcal{A}_k) \geq card(\mathcal{A}_i)$ **alors**
17 incrémenter chaque trace de $comp(\mathcal{A}_k)$ de $\frac{1}{1 + card(\mathcal{A}_{best}) - card(\mathcal{A}_k)}$
18 **si** *une trace de phéromone est inférieure à* τ_{min} **alors** la mettre à τ_{min}
19 **si** *une trace de phéromone est supérieure à* τ_{max} **alors** la mettre à τ_{max}
20 **jusqu'à** *maxCycles atteint* ;
21 **retourner** $\mathcal{A}_{best}$

L'algorithme 3 décrit la procédure *Ant-CP* utilisée pour chercher une solution, procédure qui remplace la recherche arborescente classiquement utilisée avec IBM ILOG Solver. Notons bien que cette procédure ACO n'est pas exacte, contrairement aux procédures de recherche intégrées à ILOG Solver : *Ant-CP* n'est pas ca-

pable de prouver l'incohérence du problème s'il n'admet pas de solution, et peut ne pas trouver de solution à un problème admettant une solution ; dans ce cas, *Ant-CP* fournira en sortie la meilleure affectation trouvée, c.-à-d. celle affectant le plus grand nombre de variables.

Ant-CP construit itérativement des affectations (lignes 5 à 11) qui sont utilisées pour mettre-à-jour les traces de phéromone (lignes 16 à 20). On décrit dans les paragraphes suivants la stratégie phéromonale Φ utilisée par les fourmis pour guider la recherche, ainsi que les principales étapes de l'algorithme *Ant-CP*, à savoir, le choix de variables et de valeurs, la propagation de contraintes et la mise-à-jour de la phéromone.

5.2.1 Stratégie phéromonale

La stratégie phéromonale Φ est un paramètre de la procédure *Ant-CP*. Elle est définie par un triplet $(S, \tau, comp)$ tel que :

- S est l'ensemble des composants sur lesquels les fourmis peuvent déposer de la phéromone ;
- τ est la fonction qui détermine comment exploiter les traces de phéromone au moment de choisir une valeur pour une variable ; plus précisément, étant données une affectation partielle en cours de construction $\mathcal{A}$, une variable x_j et une valeur $v \in D(x_j)$, $\tau(\mathcal{A}, x_j, v)$ retourne la valeur du facteur phéromonal qui évalue l'expérience passée de la colonie concernant le fait d'ajouter $\langle x_j, v\rangle$ à l'affectation partielle $\mathcal{A}$;
- $comp$ est la fonction qui détermine l'ensemble des composants phéromonaux sur lesquels déposer de la phéromone quand on veut récompenser une affectation $\mathcal{A}$; plus précisément, étant donnée une affectation partielle $\mathcal{A}$, $comp(\mathcal{A})$ retourne l'ensemble des composants phéromonaux de S qui sont associés à l'affectation $\mathcal{A}$.

L'objectif de la stratégie phéromonale est d'apprendre quelles sont les décisions qui ont permis de construire de bonnes affectations afin de pouvoir utiliser cette information pour biaiser les constructions futures. Par défaut, la stratégie phéromonale considérée, notée Φ_{defaut}, est la même que celle utilisé dans l'algorithme $Ant - Solver$ (voir la section 4.3) :

- les fourmis déposent de la phéromone sur les couples variable/valeur, c.-à-d.,
$$S = \{\tau_{\langle x_i, v\rangle} | x_i \in X, v \in D(x_i)\}$$
de sorte que chaque trace phéromonale $\tau_{\langle x_i, v\rangle}$ représente l'expérience passée de la colonie concernant le fait d'affecter la valeur v à la variable x_i ;
- le facteur phéromonal est défini par
$$\tau(\mathcal{A}, x_j, v) = \tau_{\langle x_j, v\rangle}$$
- l'ensemble des composants phéromonaux sur lesquels de la phéromone est déposée quand on récompense une affectation $\mathcal{A}$ est l'ensemble des couples

variable/valeur contenus dans $\mathcal{A}$, c.-à-d.,

$$comp(\mathcal{A}) = \{\tau_{\langle x_i,v\rangle} | \langle x_i, v\rangle \in \mathcal{A}\}$$

Pour certains problèmes, il peut être souhaitable de considérer d'autres stratégies phéromonales de sorte que l'utilisateur peut définir sa propre stratégie phéromonale. Il doit alors définir le triplet $(S, \tau, comp)$ correspondant à sa stratégie. Cet aspect sera illustré en 5.3 sur le problème d'ordonnancement de voitures.

5.2.2 Choix d'une variable et d'une valeur

A chaque itération, la prochaine variable à instancier est choisie selon une heuristique d'ordre donnée (ligne 7). On peut utiliser pour cela les nombreuses heuristiques d'ordre prédéfinies dans ILOG Solver comme, par exemple, l'heuristique `IloChooseMinSizeInt` qui consiste à choisir la variable ayant le plus petit domaine.

Après avoir choisi la prochaine variable à affecter, les fourmis choisissent une valeur pour cette variable (ligne 8). La contribution principale d'ACO pour la résolution de CSP est de fournir une heuristique générique pour ce choix de valeur : la valeur v à affecter à la variable x_j est choisie aléatoirement dans le domaine $D(x_j)$ en fonction d'une probabilité $p(x_j, v)$ dépendant d'un facteur phéromonal $\tau(\mathcal{A}, x_j, v)$ et d'un facteur heuristique $\eta(\mathcal{A}, x_j, v)$ dont les importances relatives sont modulées par deux paramètres α et β. Le facteur phéromonal reflète l'expérience passée de la colonie concernant l'ajout de $\langle x_j, v\rangle$ à l'affectation partielle $\mathcal{A}$; sa définition dépend de la stratégie phéromonale Φ. Le facteur heuristique permet d'introduire des heuristiques d'ordre de choix de valeurs. Il peut s'agir d'heuristiques génériques, comme celles prédéfinies dans IBM ILOG Solver. Il peut également s'agir d'heuristiques spécifiques aux problèmes considérés comme, par exemple, l'heuristique introduite dans [Solo8] et rappelée en section 2.8 du chapitre 1 pour le problème d'ordonnancement de voitures.

5.2.3 Propagation de contraintes

A chaque fois qu'une variable est affectée à une valeur, les contraintes sont propagées afin de réduire les domaines des variables qui ne sont pas encore affectées (ligne 10). Si le domaine d'une variable est réduit à un singleton, l'affectation partielle en cours de construction est complétée avec l'affectation de cette variable et le processus de propagation de contraintes est continué. Si le domaine d'une variable devient vide, alors la propagation se termine sur un échec et la construction de l'affectation partielle $\mathcal{A}_k$ se termine en retirant la dernière valeur affectée à la dernière variable.

Pour la propagation de contraintes, Ant-CP utilise les algorithmes de propagation intégrés dans ILOG Solver.

5.2.4 Mise-à-jour de la phéromone

Une fois que toutes les fourmis ont construit une affectation chacune, les traces de phéromone de S sont mises-à-jour selon la métaheuristique ACO : elles sont tout d'abord diminuées par évaporation (ligne 15) ; elles sont ensuite récompensées en fonction de leur contribution à la construction des meilleures affectations du cycle, c.-à-d., celles ayant affecté le plus grand nombre de variables (lignes 16 à 20). L'ensemble des composants phéromonaux à récompenser dépend de la stratégie phéromonale considérée. La quantité de phéromone déposée sur ces composants est inversement proportionnelle à la différence entre le nombre de variables affectées dans la meilleure affectation construite depuis le début de l'exécution, $\mathcal{A}_{best}$, et le nombre de variables affectées dans l'affectation récompensée $\mathcal{A}_k$.

Enfin, les traces de phéromone sont bornées entre τ_{min} et τ_{max} (lignes 21 et 22) selon le principe du $\mathcal{MAX} - \mathcal{MIN}$ *Ant System*.

5.3 Application de Ant-CP sur le problème d'ordonnancement de voitures

Dans cette section, nous allons illuster Ant-CP sur le problème d'ordonnancement de voitures décrit en section 1.4.2.

5.3.1 Modèle PPC considéré

Le problème d'ordonnancement de voitures à été introduit dans la communauté PPC en 1988, par Dincbas, Simonis et Van Hentenryck [DSH88]. Il est devenu depuis un problème emblématique, souvent utilisé pour évaluer les performances des langages de PPC. Ainsi, ce problème est le premier de la bibliothèque *CSPlib* [GW].

Pour évaluer *Ant-CP*, nous avons considéré le modèle PPC classique, correspondant au premier modèle proposé dans le manuel utilisateur d'ILOG Solver pour le problème d'ordonnancement de voitures. Nous invitons le lecteur intéressé à consulter ce manuel pour plus d'informations sur ce modèle [ILO98].

5.3.2 Heuristique d'ordre de choix de variables

Pour cette application de Ant-CP sur le problème d'ordonnancement de voitures, nous avons utilisé une heuristique d'ordre classique pour ce problème : nous affectons les variables x_i associées aux positions dans l'ordre défini par les positions, c.-à-d., les variables vont être affectées dans l'ordre suivante : $x_1, x_2, x_3, ..., x_n$. Notons que chaque variable o_i^j est affectée par propagation de contraintes lorsque la variable x_i correspondante est affectée ou lorsque les contraintes de capacité permettent d'inférer que l'option o_j ne peut être installée sur la voiture qui devrait être placée sur la position i de la séquence de voiture.

5.3.3 Stratégies phéromonales considérées

La stratégie phéromonale par défaut Φ_{defaut} associe une trace de phéromone à chaque couple $\langle x_i, v\rangle$ tel que x_i est une variable associée à une position et v une valeur du domaine de x_i, c.-à-d., une classe de voiture.

Nous comparons les performances de la stratégie phéromonale par défaut avec deux autres stratégies phéromonales dénotées par $\Phi_{classes}$ et Φ_{cars}.

Stratégie $\Phi_{classes}$

Cette stratégie phéromonale a été proposée dans [GGP06]. La structure phéromonale S associe une trace $\tau_{(v,w)}$ à chaque couple de classes de voitures (v, w). Cette trace représente l'expérience de la colonie concernant le fait d'affecter la valeur w à une variable x_i quand x_{i-1} a été affectée à la valeur v ou, autrement dit, de séquencer une voiture de la classe w juste derrière une voiture de la classe v. Pour cette stratégie phéromonale, le facteur phéromonal est égal à la trace de phéromone entre la classe de la dernière voiture séquencée et la classe candidate (le facteur phéromonal est égal à un pour la première variable) :

$$\begin{array}{ll} \tau(\mathcal{A}, x_k, v) = \tau_{(w,v)} & si\ k > 1\ et\ \langle x_{k-1}, w\rangle \in \mathcal{A} \\ \tau(\mathcal{A}, x_k, v) = 1 & si\ k = 1 \end{array}$$

Lors de la mise-à-jour des traces de phéromone, la phéromone est déposée sur les couples de valeurs affectées consécutivement dans les affectations à récompenser, c.-à-d.,

$$comp(\mathcal{A}) = \{\tau_{(v,w)} | \exists x_l \in X, \{\langle x_l, v\rangle, \langle x_{l+1}, w\rangle\} \subseteq \mathcal{A}\}$$

Stratégie Φ_{cars}

Cette stratégie phéromonale a été proposée dans [Solo8, Soloo]. La structure phéromonale S associe une trace $\tau_{(v,i,w,j)}$ à chaque couple de classes de voitures (v, w) et chaque $i \in [1; \#v]$ et $j \in [1; \#w]$ où $\#v$ et $\#w$ sont les nombres de voitures des classes v et w respectivement. Cette trace représente l'expérience de la colonie concernant le fait de séquencer la j^{eme} voiture de la classe w juste derrière la i^{eme} voiture de la classe v. Dans ce cas, le facteur phéromonal est défini par

$$\begin{array}{ll} \tau(\mathcal{A}, x_k, v) = \tau_{(w,j,v,i+1)} & si\ k > 1\ et\ \langle x_{k-1}, w\rangle \in \mathcal{A} \\ & et\ j = card(\{\langle x_l, w\rangle \in \mathcal{A}\}) \\ & et\ i = card(\{\langle x_l, v\rangle \in \mathcal{A}\}) \\ \tau(\mathcal{A}, x_k, v) = 1 & si\ k = 1 \end{array}$$

Lors de la mise-à-jour des traces de phéromone, la phéromone est déposée sur les composants correspondant à des couples de voitures séquencées consécutivement dans les affectations à récompenser :

$$\begin{array}{ll} comp(\mathcal{A}) = \{\tau_{(v,i,w,j)} | & \exists x_l \in X, \{\langle x_l, v\rangle, \langle x_{l+1}, w\rangle\} \subseteq \mathcal{A}_k \\ & et\ i = card(\{\langle x_m, v\rangle / m \leq l\}) \\ & et\ j = card(\{\langle x_m, w\rangle / m \leq l+1\})\} \end{array}$$

5.3.4 Facteur heuristique

Dans la probabilité de transition utilisée pour choisir la valeur à affecter à une variable, le facteur phéromonal $\tau(\mathcal{A}_k, x_j, v)$, qui représente l'expérience passée de la colonie, est combiné à un facteur heuristique $\eta(\mathcal{A}_k, x_j, v)$ dépendant du problème à résoudre.

Pour le problème d'ordonnancement de voitures, nous avons choisi d'utiliser l'heuristique basée sur la somme des taux d'utilisation (DSU) que nous avons vu dans la section 2.5.4 comme heuristique de choix de valeurs.

Nous avons constaté que la DSU, en plus de son rôle d'heuristique de choix de valeurs, peut être utilisée pour détecter des incohérences et pour filtrer les domaines des variables. Nous avons appelé cette nouvelle utilisation de la DSU par DSU+P ("p" pour propagation). La sous-section suivante donne plus de détails sur DSU+P.

Somme des taux d'utilisation avec propagation (DSU+P)

Si l'en regarde de plus près la formule définissant le taux d'utilisation d'une option o_i (2.6), nous pouvons constater que nous pouvons exploiter ces taux d'utilisation pour détecter des incohérences et pour filtrer les domaines des variables. Pour cela, nous avons défini les deux règles suivantes :

- quand le taux d'utilisation d'une option UR_{o_i} devient supérieur à 1, i.e., quand $reqSlots(o_i, n_i)$ devient supérieur à N, nous pouvons en conclure qu'il n'est pas possible de compléter la séquence sans violer de contraintes, et on peut arrêter la construction en cours sur un échec ;
- quand le taux d'utilisation d'une ou plusieurs options devient égal à 1, nous pouvons supprimer du domaine de la prochaine variable à affecter toutes les classes de voitures ne demandant pas toutes les options dont le taux d'utilisation est égal à 1.

5.4 Résultats expérimentaux

5.4.1 Instances utilisées

Les instances satisfiables de la bibliothèque CSPlib [GW] à 100 voitures (4 instances proposées par Régin et Puget [RP97]) et 200 voitures (70 instances proposées par Lee et al dans [LLW98]) sont toutes facilement résolues par *Ant-CP*, quelles que soient la structure phéromonale et l'heuristique considérées.

Nous considérons ici un jeu d'essai plus difficile généré par Perron et Shaw [PS04]. Les instances de ce jeu d'essai ont toutes 8 options et 20 classes de voitures différentes par instance ; les contraintes de capacité sur les options, définies par les fonctions p et q, sont générées aléatoirement en respectant les contraintes suivantes : $\forall o_i \in O, 1 \leq p(o_i) \leq 3$ et $p(o_i) < q(o_i) \leq p(o_i) + 2$. Le jeu d'essai comporte 32 instances à 100 voitures, 21 instances à 300 voitures et 29 instances

à 500 voitures. Toutes ces instances admettent une solution ne violant aucune contrainte.

5.4.2 Instanciations d'*Ant-CP* considérées

Nous allons comparé les stratégies phéromonales Φ_{defaut}, $\Phi_{classes}$ et Φ_{cars} introduites en 5.3.3. Afin d'évaluer l'apport de la phéromone, nous considérons également une stratégie sans phéromone, notée $\Phi_{\emptyset}$: dans ce cas, l'ensemble S est l'ensemble vide et le facteur phéromonal $\tau(\mathcal{A}_k, x_j, v)$ est fixé à 1 pour qu'il n'ait pas d'impact dans la formule de calcul des probabilités.

Nous comparons également les deux facteurs heuristiques DSU et DSU+P introduits en 5.3.4.

Nous notons *Ant-CP*(Φ, h) l'instanciation d'*Ant-CP* obtenue avec la stratégie phéromonale $\Phi \in \{\Phi_{\emptyset}, \Phi_{classes}, \Phi_{cars}, \Phi_{defaut}\}$ et l'heuristique $h \in \{$DSU,DSU+P$\}$.

5.4.3 Paramétrage

Ant-CP est paramétré par

- le nombre de fourmis *nbAnts*, qui détermine le nombre d'affectations construites à chaque cycle avant chaque étape de mise-à-jour de la phéromone,
- α et β, qui déterminent les poids relatifs des facteurs phéromonal et heuristique dans la probabilité de choix de valeur,
- ρ, qui détermine la vitesse d'évaporation des traces phéromonales,
- τ_{min} et τ_{max} qui fixent les bornes minimale et maximale des traces de phéromone.

La valeur du paramètre β change d'une application à l'autre, en fonction de la fiabilité du facteur heuristique. Nous avons fixé ce paramètre à 6, comme cela a été proposé dans [GPS03] où l'heuristique DSU a été introduite.

Les valeurs des autres paramètres déterminent l'influence de la phéromone sur le processus de résolution, et donc le degré d'intensification de la recherche. Comme nous l'avons vu en section 3.5 du chapitre 3, il s'agit là de trouver un bon compromis entre, d'une part, intensifier la recherche aux alentours des zones les plus prometteuses (contenant les meilleures affectations trouvées) et, d'autre part, diversifier la recherche afin de découvrir de nouvelles zones de l'espace de recherche dont on espère qu'elles contiendront de meilleures affectations.

Les instances que nous considérons ici étant relativement difficiles, nous avons choisi un paramétrage permettant une intensification modérée, à savoir, $\alpha = 1$, $\rho = 0.02$, *nbAnts*=30, $\tau_{min} = 0.01$ et $\tau_{max} = 4$.

5.4.4 Comparaison des différentes instanciations d'*Ant-CP*

La figure 5.1 compare les différentes stratégies phéromonales pour l'heuristique DSU (courbes du haut), puis pour l'heuristique DSU+P (courbes du bas).

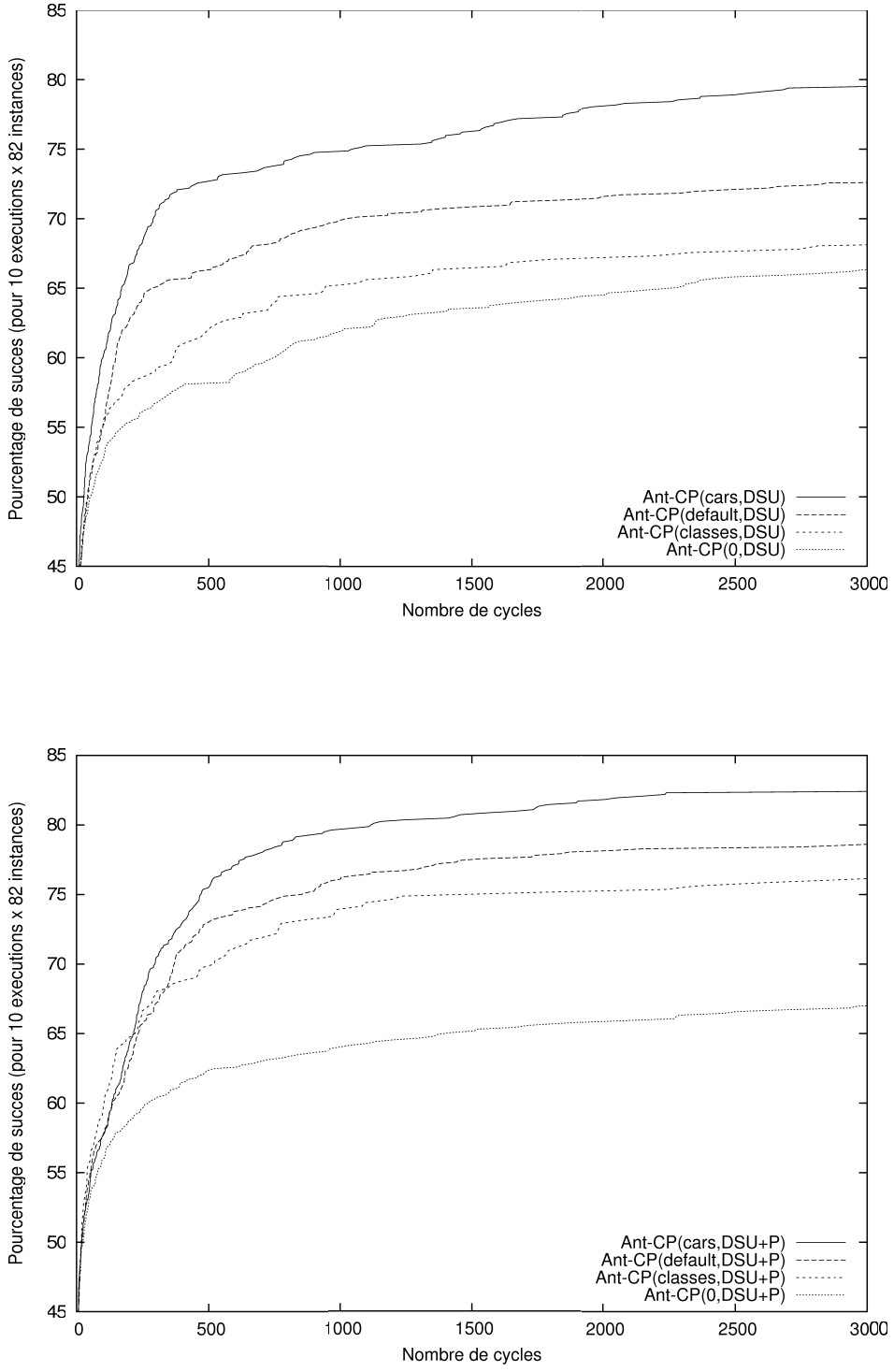

FIGURE 5.1 – *Comparaison de différentes instanciations d'Ant-CP*(Φ, h)*, avec* $\Phi \in \{\Phi_{defaut}, \Phi_{classes}, \Phi_{cars}, \Phi_{\emptyset}\}$ *et* $h \in \{DSU, DSU + P\}$.

Chaque courbe trace l'évolution du pourcentage d'exécutions ayant trouvé une solution en fonction du nombre de cycles (pour 10 exécutions sur chacune des 82 instances). Nous constatons qu'après 3000 cycles (chaque cycle correspondant à la construction de *nbAnts* affectations) la phéromone a permis d'augmenter le pourcentage d'instances résolues de 66.32% pour *Ant-CP*($\Phi_\emptyset$,DSU) à 79.39% pour *Ant-CP*(Φ_{cars},DSU), 72.56% pour *Ant-CP*($\Phi_{default}$,DSU) et 68.29% pour *Ant-CP*($\Phi_{classes}$,DSU). Ainsi, sur ces instances la meilleure stratégie phéromonale est Φ_{cars} ; la stratégie phéromonale par défaut obtient de moins bons résultats, mais est cependant nettement meilleure que $\Phi_{classes}$.

L'heuristique DSU+P améliore généralement les résultats par rapport à l'heuristique DSU. Cette amélioration est différente selon la stratégie phéromonale considérée : elle est quasiment nulle quand la phéromone est ignorée (le taux de succès pour $\Phi_\emptyset$ passe de 66.32% à 66.97%) ; elle est plus importante pour les stratégies phéromonales $\Phi_{classes}$ et Φ_{defaut} (les taux de succès passent de 68.29% à 74.39% pour $\Phi_{classes}$ et de 72.56% à 78.54% pour Φ_{defaut}) ; elle est plus modérée pour Φ_{cars} (le taux de succès passe de 79.39% à 82.32%).

Notons finalement que l'heuristique DSU+P diminue les temps d'exécution d'un cycle par rapport à l'heuristique DSU car elle filtre plus les domaines des variables et détecte plus tôt certaines incohérences. Ainsi, pour les instances à 100 voitures, 3000 cycles prennent près de 5 minutes avec l'heuristique DSU, tandis qu'ils prennent près de 3 minutes avec l'heuristique DSU+P (sur un Intel Core Duo cadencé à 2Ghz).

5.4.5 Comparaison d'*Ant-CP* avec d'autres approches

De très nombreuses approches différentes ont été proposées pour le problème d'ordonnancement de voitures, qui est un problème de référence pour évaluer l'efficacité de nouvelles approches de résolution. Les approches exactes, basées sur la programmation par contraintes, ne permettent toujours pas de résoudre en un temps acceptable les instances à 100 et 200 voitures de CSPlib, même en utilisant des algorithmes de filtrage dédiés à ce problème comme, par exemple, ceux proposés dans [RP97, BNQ+07, HPRS06]. Ces instances sont toutes facilement résolues par *Ant-CP*, quelle que soit la stratégie phéromonale considérée (en moins d'une seconde pour les 70 instances à 200 voitures, et moins d'une minute pour les 4 instances satisfiables à 100 voitures, sur un Intel Core Duo cadencé à 2Ghz). A titre de comparaison, aucun des différents filtrages introduits dans [HPRS06] pour la contrainte de séquence ne permet de résoudre plus de la moitié de ces instances en moins de 100 secondes avec ILOG solver sur un Pentiun 4 cadencé à 3.2 Ghz.

De nombreuses approches heuristiques ont également été proposées, la plupart d'entre elles étant basées sur une exploration par recherche locale, par exemple, pour n'en citer que quelques unes, [LLW98, MH02, GPS03, NTG04, PS04, EGN07]. Plusieurs algorithmes ACO ont été proposés pour ce problème,

et ces algorithmes obtiennent des résultats très compétitifs avec les meilleures approches heuristiques connues pour ce problème, et notamment VFLS [EGN07] qui a remporté le challenge ROADEF 2005 [SCNA08] : la variante ACO la plus performante, appelée ACO(τ_1, τ_2) [Solo8], permet de résoudre un grand nombre d'instances nettement plus rapidement que VFLS ; elle est cependant moins performante pour quelques instances à 500 voitures.

Comme nous l'avons expliqué dans l'introduction, *Ant-CP* diffère des algorithmes ACO introduits dans [Solo8] par le fait qu'il construit des affectations partielles cohérente, et non des affectations totales incohérente, l'objectif étant d'intégrer les procédures de propagation et de vérification de contraintes d'ILOG Solver. On constate cependant que ces algorithmes ont des performances très similaires si l'on compare le nombre de cycles nécessaires pour résoudre les instances. Plus précisément, *Ant-CP*(Φ_{cars},DSU) obtient des résultats très similaires à ceux de la variante ACO de [Solo8] qui utilise la même structure phéromonale et la même heuristique, à savoir ACO(τ_1, η). En revanche, si on compare le temps nécessaire pour résoudre les instances (et non le nombre de cycles), on constate qu'*Ant-CP* est plus lent que l'algorithme ACO de [Solo8]. En effet, l'utilisation des procédures de propagation et de vérification intégrées à ILOG Solver est nettement plus coûteux que l'utilisation de procédures *ad-hoc* définies pour ce problème.

En contrepartie, l'effort de programmation à fournir est différent. Pour la version d'*Ant-CP* qui utilise la stratégie phéromonale par défaut, il s'agit uniquement de décrire le problème à résoudre en déclarant les variables et leurs domaines ainsi que les contraintes. Il est intéressant à ce niveau de noter que nous avons pu utiliser pour cela le modèle décrit dans le manuel utilisateur d'ILOG Solver. Pour les deux autres variantes d'*Ant-CP*, qui utilisent des stratégies phéromonales spécifiques, il s'agit en plus de spécifier l'ensemble S des composants phéromonaux ainsi que les deux fonctions τ et $comp$ définissant le facteur phéromonal et les composants associés à une affectation à récompenser.

5.5 Conclusion

Ces premiers essais sur le problème d'ordonnancement de voitures montrent que l'on peut facilement intégrer une stratégie de recherche inspirée par les colonies de fourmis à un langage de programmation par contraintes. Cette intégration permet de bénéficier des facilités offertes par ces langages pour modéliser des problèmes de façon déclarative, en termes de contraintes. Il est intéressant de noter à ce niveau que le modèle utilisé pour décrire le problème à résoudre est le même quelle que soit l'approche de résolution utilisée. Bien évidemment, ces expérimentations devront être poursuivies sur d'autres problèmes pour pouvoir effectivement valider notre approche.

Il est important de remarquer que ces premiers résultats pourraient être améliorés en utilisant des algorithmes de propagation de contraintes dédiés à une recherche ACO. En effet, les algorithmes de propagation de contraintes intégrés

dans ILOG Solver ont été conçus pour être utilisés dans un contexte de recherche arborescente avec retours-arrières. Ils maintiennent pour cela des structures de données qui permettent de restaurer, à chaque retour-arrière, le contexte du point de choix précédent. Ces structures de données qui sont coûteuses à maintenir ne sont pas nécessaires dans le contexte d'une recherche ACO où les choix faits ne sont jamais remis en cause.

Un constat similaire peut être fait en ce qui concerne l'utilisation d'un langage comme Comet pour permettre une " recherche ACO basée sur les contraintes. En effet, Van Hentenryck et Michel ont montré dans [HM05] que le langage Comet peut être utilisé pour implémenter un algorithme ACO de façon très déclarative. Cependant, Comet est dédié à la recherche locale, c.-à-d., à une exploration de l'espace des affectations par applications successives de transformations élémentaires à une affectation courante. Il maintient pour cela des structures de données permettant une évaluation incrémentale et automatique des différentes propriétés invariantes et fonctions objectifs à chaque transformation élémentaire. Ces structures de données peuvent être utilisées pour une recherche ACO (pour évaluer incrémentalement les facteurs heuristiques à chaque étape de la construction) mais elles maintiennent plus de choses que nécessaire dans la mesure où les choix faits lors de la construction gloutonne des affectations ne sont jamais remis en cause. Notons cependant que ces structures de données facilitent, de fait, une hybridation avec des procédures de recherche locale.

Ainsi, une première piste de recherche pour une réelle intégration d'ACO dans un langage de programmation par contraintes concerne la conception d'algorithmes permettant une évaluation automatique et incrémentale des contraintes dans un contexte d'approche constructive où les choix faits ne sont jamais remis en cause (ni par retour-arrière comme dans les approches arborescentes, ni par application d'une transformation élémentaire comme dans la recherche locale).

Une seconde piste de recherche importante concerne le paramétrage d'ACO. Les nombreux paramètres d'un algorithme ACO et leur forte influence sur la résolution sont un frein évident à une utilisation simple dans un contexte de programmation par contraintes. Il s'agit donc d'étudier les possibilités de paramétrage adaptatif, où les valeurs des paramètres sont adaptées dynamiquement au cours du processus de résolution. Cet aspect sera plus particulièrement étudié au chapitre 7.

Intégration de ACO dans la PPC pour la résolution des COPs

6

Sommaire

Dans le cinquième chapitre, nous avons présenté un algorithme hybride entre la métaheuristique d'optimisation par colonies de fourmis et la programmation par contraintes pour la résolution de problèmes de satisfaction de contraintes. Dans ce chapitre, nous allons présenter une nouvelle approche hybride et générique dans laquelle ACO est combiné avec l'approche $B\&P\&B$ pour résoudre des problèmes d'optimisation sous contraintes [1].

1. Les travaux présentés dans ce chapitre ont été partiellement publiés dans CPAIOR2010 (en Italie) [KAS10b] et aux JFPC2010 (à Caen)[KAS10a]

6.1 Motivations

Comme nous l'avons vu dans le chapitre 1, un COP est un CSP augmenté avec une fonction à optimiser (la fonction objectif). Habituellement, pour résoudre un COP, l'utilisateur a le choix entre l'utilisation d'une approche complète ou l'utilisation d'une approche métaheuristique.

Dans le chapitre 2, nous avons vu que l'approche complète trouve la solution optimale en explorant l'espace de recherche de manière exhaustive en utilisant généralement des algorithmes de type *Branch & Propagate & Bound* (B&P&B). Egalement, nous avons vu que la garantie de l'optimalité que l'approche complète nous offre a un coût en terme du temps de calcul qui peut être exponentiel.

Dans le chapitre 3, nous avons vu que les métaheuristiques peuvent traiter un COP et trouver des solutions de très bonne qualité en un temps acceptable. Cependant, les métaheuristiques présentent généralement deux inconvénients majeurs à savoir (1) elles ne garantissent pas l'optimalité des solutions trouvées, car, elles n'explorent pas l'intégralité de l'espace de recherche (2) elles sont généralement dédiées à la résolution d'un problème spécifique.

L'examen des avantages et des inconvénients des deux approches montre qu'elles sont complémentaires. Cette complémentarité entre ces deux approches nous a inspiré pour proposer un nouvel algorithme dans lequel nous avons essayé de palier aux inconvénients de chacune des deux approches. En effet, dans ce chapitre, nous allons présenter un algorithme hybride complet et générique pour la résolution de problèmes d'optimisation combinatoires. Cet algorithme est hybride, car il combine l'algorithme d'optimisation par colonies de fourmis (ACO) avec IBM ILOG CP Optimizer qui est basé sur une approche complète de type "Branch &Propagate &Bound" (B&P&B). Il est complet, car il explore l'espace de recherche de manière exhaustive. Il est générique, car il n'est pas dédié à la résolution d'un problème particulier mais pour résoudre tout problème modélisé dans le format d'un COP tel que nous l'avons défini dans le premier chapitre.

Dans l'approche que nous proposons dans ce chapitre, le COP à résoudre est modélisé dans le langage de CP Optimizer. Ensuite, il est résolu par notre algorithme générique qui fonctionne en deux phases séquentielles. La première phase sert à échantillonner l'espace des solutions. Pendant cette première phase, CP Optimizer est utilisé pour construire des affectations complètes et cohérentes et le mécanisme d'apprentissage par renforcement d'ACO est utilisé pour identifier les zones prometteuses de l'espace de recherche par rapport à la fonction objectif. Durant la deuxième phase, CP Optimizer effectue une recherche arborescente exhaustive guidée par les traces de phéromone accumulées lors de la première phase. Nous avons testé notre algorithme sur les problèmes de sac à dos ; d'affectation quadratique et de clique maximum. Les premiers résultats sur ces trois problèmes montrent que ce nouvel algorithme améliore les performances de CP Optimizer.

Rappelons dès maintenant que notre objectif principal n'est pas de rivaliser

avec les algorithmes existants dans l'état de l'art qui sont dédiés à la résolution de problèmes spécifiques, mais de montrer qu'ACO peut améliorer le processus de recherche d'un algorithme générique utilisant la technique de B&P&B tout en conservant ses avantages principaux, à savoir la déclarativité et la complétude de la recherche. Pour cela, nous avons choisi CP Optimizer comme référence, et nous l'avons utilisé comme une "boite noire" avec sa configuration par défaut correspondant à la stratégie de recherche "Restart" proposée par Refalo dans [Refo4] et décrite en section 2.7.

La suite de ce chapitre est organisée comme suit : dans la section suivante, nous allons donner la description de l'algorithme que nous proposons et qui est nommé CPO-ACO (CP Optimizer-Ant Colony Optimization). Dans la troisième section, nous allons donner les résultats expérimentaux de la comparaison de CPO-ACO et CP Optimizer. Dans la quatrième section, nous allons introduire et évaluer une structure de phéromone adaptée aux COP qui ne contiennent que des variables binaires. Dans la cinquième section, nous allons évaluer CPO-ACO en utilisant une heuristique dédiée. Dans la dernière section, nous donnons la synthèse et la conclusion de ce chapitre.

6.2 Description de $CPO - ACO$

Comme dans le chapitre 5, nous proposons de combiner ACO avec la PPC afin de pouvoir réutiliser les nombreuses procédures disponibles en PPC pour la gestion des contraintes. En revanche, cette fois-ci, l'algorithme que nous proposons effectue une recherche complète et il n'est pas dédié à la résolution d'un problème de satisfaction de contraintes mais à un problème d'optimisation combinatoire sous contraintes.

L'algorithme que nous proposons $CPO - ACO$ fonctionne en deux phases :

- Pendant la première phase, ACO utilise CP Optimizer pour construire des solutions et le mécanisme d'apprentissage phéromonal est utilisé pour intensifier progressivement la recherche autour des meilleures solutions trouvées.
- Pendant la deuxième phase, CP Optimizer est utilisé pour rechercher la solution optimale et les traces de phéromone recueillies au cours de la première phase sont utilisées pour guider CP Optimizer dans sa recherche.

La structure de phéromone utilisée dans $CPO - ACO$ est telle que pour chaque couple variable/valeur ($< x_i, v_j >$) est associé une trace de phéromone $\tau(x_i, v_j)$, exactement comme dans la stratégie phéromonale Φ_{defaut} que nous avons utilisé dans le chapitre 5 comme structure de phéromone par défaut. La variante de ACO utilisée dans $CPO - ACO$ est $\mathcal{MAX} - \mathcal{MIN}$ *Ant System*. Ci-dessous, nous donnons plus de détails sur chacune des deux phases de $CPO - ACO$.

6.2.1 La première phase de $CPO - ACO$

L'algorithme 4 décrit la première phase de $CPO - ACO$, les grandes lignes de cet algorithme sont décrites ci-après.

Algorithme 4 : Phase 1 de $CPO - ACO$

Entrées : $P = (X, D, C, F)$ et les paramètres
$\{t_{max1}, d_{min}, it_{max}, \alpha, \beta, \rho, \tau_{min}, \tau_{max}, nbAnts\}$
Sorties : une solution $\mathcal{A}_{best}$ et une matrice de phéromone
$\tau : X \times D \rightarrow [\tau_{min}; \tau_{max}]$

1 **début**
2 **pour chaque** $x_i \in X$ **et chaque** $v_j \in D(x_i)$ **faire** $\tau(x_i, v_j) \leftarrow \tau_{max}$
3 **répéter**
`/* Construction des solutions */`
4 **pour chaque** $k \in \{1, \ldots, nbAnts\}$ **faire**
5 Construire une solution $\mathcal{A}_k$ en utilisant CP Optimizer
`/* Evaporation de la phéromone */`
6 **pour chaque** $x_i \in X$ **et chaque** $v_i \in D(x_i)$ **faire**
7 $\tau(x_i, v_i) \leftarrow max(\tau_{min}, (1 - \rho) \cdot \tau(x_i, v_i))$
`/* Dépôt de phéromone */`
8 Soit $\mathcal{A}_{best}$ la meilleure solution construite jusqu'à présent (y compris le dernier cycle)
9 **pour chaque** $k \in \{1, \ldots, nbAnts\}$ **faire**
10 **si** $\forall l \in \{1, \ldots, nbAnts\}$, $\mathcal{A}_k$ *est au moins aussi bon que* $\mathcal{A}_l$ **alors**
11 **pour chaque** $< x_i, v_i > \in \mathcal{A}_k$ **faire**
$\tau(x_i, v_i) \leftarrow min(\tau_{max}, \tau(x_i, v_i) + \frac{1}{1+|F(\mathcal{A}_k) - F(\mathcal{A}_{best})|})$
12 **si** $\mathcal{A}_{best}$ *est strictement meilleur que toutes les solutions* $\{\mathcal{A}_1, ..., \mathcal{A}_{nbAnts}\}$ **alors**
13 **pour chaque** $< x_i, v_i > \in \mathcal{A}_{best}$ **faire**
$\tau(x_i, v_i) \leftarrow min(\tau_{max}, \tau(x_i, v_i) + 1)$
14 **jusqu'à** *temps utilisé* $\geq t_{max1}$ **ou** *nombre de cycle sans amélioration de* $\mathcal{A}_{best} \geq it_{max}$ **ou** *distance moyenne de* $\{\mathcal{A}_1, ..., \mathcal{A}_{nbAnts}\} \leq d_{min}$;
15 **retourne** $\mathcal{A}_{best}$ **et** τ
16 **fin**

Construction d'une affectation

Lors de chaque cycle (lignes 3-14), chaque fourmi demande à CP Optimizer de construire une solution (ligne 5). Notez que pendant cette première phase, nous ne demandons pas à CP Optimizer d'optimiser la fonction objectif, mais simplement de trouver des solutions satisfaisant toutes les contraintes. Chaque nouvel appel à CP Optimizer correspond à une nouvelle recherche (restart) et CP Optimizer construit une solution selon le principe B&P : il propage les contraintes après chaque affectation et si un échec est détecté, il fait marche arrière pour prendre une autre décision jusqu'à ce qu'il trouve une solution. Lors de cette première

phase, CP Optimizer est utilisé avec ses paramètres par défaut à l'exception de l'heuristique de choix de valeur qui lui est transmise en paramètre. Cette heuristique est basée sur ACO et définit la probabilité de choisir la valeur v_j pour une variable x_i par

$$p(v_j) = \frac{[\tau(x_i, v_j)]^\alpha \cdot [1/impact(v_j)]^\beta}{\sum_{v_k \in D(x_i)} [\tau(x_i, v_k)]^\alpha \cdot [1/impact(v_k)]^\beta} \tag{6.1}$$

où $impact(v_j)$ est l'impact observé de la valeur v_j [Refo4] (voir la section 2.5.3). Comme toutes les traces de phéromone sont initialisées à la même valeur (i.e., τ_{max}), au cours des premiers cycles, les impacts sont plus déterminants que les traces de phéromone dans le choix des valeurs. Toutefois, à la fin de chaque cycle les traces de phéromone sont mises à jour. Ainsi, ces dernières, influencent de plus en plus le choix des valeurs.

Notons que CPO-ACO est plutôt destiné à résoudre des COP pour lesquels la difficulté n'est pas de construire des solutions, mais de trouver la solution qui optimise la fonction objectif. Pour ces problèmes, CP Optimizer est capable de construire très rapidement une solution (en faisant très peu de retours arrière) de sorte que l'algorithme peut rapidement collecter un nombre significatif de solutions qui peuvent alors être utilisées par ACO pour biaiser la recherche. CPO-ACO pourrait être utilisé pour résoudre des COP plus contraints, mais dans ce cas, CP Optimizer peut avoir besoin de plus de temps pour calculer une solution satisfaisant toutes les contraintes ce qui fait que l'apprentissage phéromonal sera basé sur trop peu de solutions pour être intéressant.

Mise-à-jour des traces de phéromones

Une fois que chaque fourmi a construit une affectation, les traces de phéromone sont évaporées en les multipliant par $(1 - \rho)$ où $\rho \in [0; 1]$ est le taux d'évaporation des phéromones (lignes 6-7).

À la fin de chaque cycle, les meilleures solutions (par rapport à la fonction objectif) sont récompensées dans le but d'intensifier la recherche autour d'elles. Les meilleures solutions du cycle sont systématiquement récompensées (lignes 9-11). En revanche, la meilleure solution construite depuis le début de la recherche est récompensée seulement si elle est meilleure que les meilleures solutions du dernier cycle (lignes 12-13).

Une solution $\mathcal{A}$ est récompensée en augmentant la quantité de phéromone sur chaque couple $< x_i, v_j >$ de $\mathcal{A}$. Ainsi, la probabilité d'affecter x_i à v_j lors des futures affectations est augmentée. La quantité de phéromone ajoutée est telle que plus la qualité d'une solution est proche de la qualité de la meilleure solution trouvée, plus grande est la quantité de phéromone déposée sur ses composants.

Conditions d'arrêt

La première phase s'arrête dans l'un des cas suivants : si le temps CPU t_{max1} a été atteint ; si la meilleure solution $\mathcal{A}_{best}$ n'a pas été améliorée depuis it_{max} ité-

rations ; ou bien si la distance moyenne entre les affectations calculées pendant le dernier cycle est plus petite que d_{min}, ce qui indique que les traces de phéromone ont permis à la recherche de converger. Nous définissons la distance entre deux affectations comme étant le taux de couples variable-valeur sur lesquels les deux affectations sont différentes : la distance entre $\mathcal{A}_1$ et $\mathcal{A}_2$ est $\frac{|X|-|\mathcal{A}_1 \cap \mathcal{A}_2|}{|X|}$.

6.2.2 Deuxième phase de $CPO - ACO$

À la fin de la première phase, la meilleure solution construite $\mathcal{A}_{best}$ et la structure phéromonale τ sont transmises à la deuxième phase. Le coût de $\mathcal{A}_{best}$ est utilisé pour borner la fonction objectif. Ensuite, CP Optimizer est lancé pour chercher la solution optimale : dans cette deuxième phase, chaque fois que CP Optimizer trouve une meilleure solution, il borne la fonction objectif avec son coût et il fait marche arrière pour trouver de meilleures solutions ou prouver l'optimalité de la dernière solution trouvée.

Comme dans la première phase, CP Optimizer est utilisé comme une boîte noire avec ses paramètres de recherche par défaut et utilise les impacts comme heuristique d'ordre de variables. Cependant, l'heuristique d'ordre de valeurs est une combinaison entre la structure phéromonale τ et les "impacts" : pour une variable x_i à affecter, la valeur $v_i \in D(x_i)$ qui lui sera choisie est celle qui maximise la formule

$$\tau(x_i, v_i)]^{\alpha} \cdot [1/impact(v_i)]^{\beta}. \tag{6.2}$$

Notons que nous avons comparé expérimentalement les différentes variantes de CPO-ACO suivantes :

- Dans le but de montrer l'intérêt d'utiliser le mécanisme d'apprentissage phéromonal lors de la première phase, nous avons testé la variante où, pendant la première phase, la recherche est effectuée sans utiliser de phéromone de sorte que l'heuristique de choix de valeurs est définie par les impacts uniquement. Cette variante donne des résultats significativement moins bons.
- Pour évaluer l'intérêt de l'utilisation des traces de phéromone lors de la deuxième phase, nous avons testé la variante où, à la fin de la première phase, nous ne retenons que $\mathcal{A}_{best}$ qui est utilisé pour borner la fonction objectif au début de la deuxième phase. La deuxième phase est alors constituée par la recherche par défaut de CP Optimizer (qui utilise uniquement les impacts comme heuristique de choix de valeurs). Cette variante obtient également des résultats beaucoup moins bons que lorsque la structure de phéromone est utilisée lors de cette deuxième phase.
- Enfin, nous avons testé la variante où, au cours de la deuxième phase, le choix de valeur pour une variable donnée est fait en utilisant la règle de transition probabiliste de ACO (voir la section 6.2.1) comme dans la première phase au lieu de sélectionner la valeur qui maximise la formule donnée en section 6.2.2. Cette variante obtient, sur la plupart des tests effectués,

des résultats qui ne sont pas significativement différents de ceux que nous allons donner dans la section suivante.

6.3 Evaluation expérimentale de CPO-ACO

6.3.1 Les problèmes et benchmarck considérés

Dans un premier temps, nous avons évalué l'algorithme CPO-ACO sur trois COP bien connus, à savoir : le problème de sac-à-dos multidimensionnel (MKP) ; le problème d'affectation quadratique (QAP) ; le problème de clique maximum (voir respectivement les sections 1.4.3, 1.4.4, 1.4.5)

Les instances considérées pour (MKP) Nous avons considéré quelqu'une des instances académique avec 100 objets disponibles sur http ://people.brunel.ac.uk/~ mastjjb/jeb/orlib/mknapinfo.html. Nous avons considéré les 20 premières instances avec 5 contraintes de ressources (de 5-100-00 à 5-100-19), les 20 premières instances avec 10 contraintes de ressources (de 10-100-00 à 10-100-19) et les 20 premières instances avec 30 contraintes de ressources (de 30-100-00 à 30-100-19).

Les instances considérées pour (QAP) Nous avons utilisé les instances académiques de la QAPLIB qui sont disponibles sur www.opt.math.tu-graz.ac.at/qaplib/inst.html.

Les instances du problème de cliques maximum considérées Les instances du problème de clique maximum que nous avons utilisé sont : (1) 40 instances FRBs disponibles sur http ://www.nlsde.buaa.edu.cn/∼ kexu/benchmarks/graph-benchmarks.htm (2) et une dizaine d'instances DIMACS.

Remarque : pour chaque problème, le modèle PPC considéré est celui décrit en section 1.4 du chapitre 1 et il a été implémenté en utilisant le langage de modélisation de CP Optimizer en C++.

6.3.2 Comparaison entre CPO et CPO-ACO

Conditions expérimentales

Dans cette section, nous allons comparer CPO-ACO avec CP Optimizer (noté dans la suite par CPO). Dans les deux cas, nous avons utilisé la version V2.3 de CPO avec ses paramètres de recherche par défaut. Toutefois, pour CPO-ACO, les fonctions de choix de valeurs sont transmises en paramètres à CPO comme décrit dans la section précédente. Rappelant que pour CPO, les impacts sont utilisés comme heuristique de choix de variable et de valeur [Ref04].

Pour toutes les expériences présentées dans cette section, le temps CPU total a été limité à 300 secondes sur une machine Pentium-4 2.2 Gz. Pour CPO-ACO,

cette durée totale est partagée entre les deux phases : la durée de la phase 1 est au plus égale à $t_{max1} = 25\%$ du temps total. Nous avons par ailleurs fixé d_{min} à 0.05 (de sorte que la phase 1 est arrêtée dès que la distance moyenne entre les solutions d'un même cycle est inférieure à 5%) et it_{max} à 500 (de sorte que la phase 1 est arrêtée si $\mathcal{A}_{best}$ n'a pas été améliorée depuis 500 cycles). Le nombre de fourmis est $nbAnts = 20$; le poids du facteur phéromonal est $\alpha = 1$ et le poids du facteur heuristique est $\beta = 2$. Les traces de phéromone sont bornées entre $\tau_{min} = 0.01$ et $\tau_{max} = 1$.

Notons que nous n'avons pas utilisé le même taux d'évaporation de la phéromone pour toutes les expérimentations. En effet, pour le MKP et le problème de clique maximum les variables sont binaires, ce qui fait qu'à chaque cycle une des deux valeurs possibles (0 ou 1) est récompensée, et ACO converge assez rapidement. Pour le QAP, tous les domaines sont de taille n (où n est égal au nombre de variables), donc à chaque cycle, seulement une valeur sur les n valeurs possibles est récompensée. Donc pour le QAP, nous avons délibérément augmenté le taux d'évaporation afin d'accélérer la convergence de ACO lors de la première phase. Par conséquent, $\rho = 0.01$ pour le MKP et le problème de clique maximum et $\rho = 0.1$ pour le QAP.

Pour les deux algorithmes, CPO et CPO-ACO, nous avons effectué 30 exécutions par instance de chaque problème.

Analyse des résultats

le tableau 6.1, donne les résultats obtenus par CPO et CPO-ACO sur les trois problèmes MKP, QAP et clique maximum. Pour chaque classe d'instances et pour chaque algorithme, ce tableau donne l'écart (en pourcentage) par rapport à la meilleure solution connue. Notons d'abord que CPO et CPO-ACO n'atteignent (presque) jamais la solution optimale connue : les meilleures solutions connues sont en effet, obtenues avec des approches dédiées. CPO et CPO-ACO sont des approches complètement génériques qui ne visent pas à concurrencer ces approches dédiées. Aussi, nous avons choisi une limite raisonnable de temps CPU (300 secondes) afin de nous permettre d'effectuer un nombre suffisant de tests, pour pouvoir utiliser des tests statistiques. Avec cette limite de temps, CPO-ACO obtient des résultats compétitifs avec des approches dédiées sur le MKP (moins de 1% d'écart par rapport aux meilleures solutions connues), mais il est assez loin des meilleures solutions connues sur de nombreuses instances du QAP et du problème de clique maximum.

Comparons maintenant CPO avec CPO-ACO. Le tableau 6.1 nous montre que l'utilisation d'ACO pour guider CPO améliore le processus de recherche sur toutes les classes d'instances sauf deux. Toutefois, cette amélioration est plus importante pour le MKP que pour les deux autres problèmes. Comme les deux approches ont obtenu des résultats assez proches sur certaines classes d'instances, nous avons fait des tests statistiques (test de Student avec un niveau de confiance

Résultats pour le MKP

			CPO				*CPO − ACO*			
Name	# I	# X	avg	(sd)	$>_{avg}$	$>_{t-test}$	avg	(sd)	$>_{avg}$	$>_{t-test}$
5.100-*	20	100	1.20	(0.30)	0%	0%	**0.46**	(0.23)	**100%**	**100%**
10.100-*	20	100	1.53	(0.31)	0%	0%	**0.83**	(0.34)	**100%**	**100%**
30.100-*	20	100	1.24	(0.06)	5%	0%	**0.86**	(0.08)	**95%**	**85%**

Résultats pour le QAP

			CPO				*CPO − ACO*			
Name	# I	# X	avg	(sd)	$>_{avg}$	$>_{t-test}$	avg	(sd)	$>_{avg}$	$>_{t-test}$
bur*	7	26	1.17	(0.43)	0%	0%	**0.88**	(0.43)	**100%**	**57 %**
chr*	11	19	12.11	(6.81)	45%	9%	**10.99**	(6.01)	**55 %**	**45 %**
had*	5	16	1.07	(0.89)	0%	0%	**0.54**	(1.14)	**100%**	**60 %**
kra*	2	30	17.46	(3.00)	0%	0%	**14.99**	(2.79)	**100%**	**100%**
lipa*	6	37	22.11	(0.82)	0%	0%	**20.87**	(0.75)	**100%**	**100%**
nug*	15	20	8.03	(1.59)	7%	0%	**5.95**	(1.44)	**93 %**	**80 %**
rou*	3	16	5.33	(1.15)	33%	0%	**3.98**	(1.00)	**67 %**	**67 %**
scr*	3	16	**4.60**	(2.4)	33%	0%	5.12	(2.60)	**67 %**	0 %
tai*	4	16	6.06	(1.35)	25%	25%	**4.84**	(1.25)	**75 %**	**50 %**

Résultats pour clique maximum

			CPO				*CPO − ACO*			
Name	# I	# X	avg	(sd)	$>_{avg}$	$>_{t-test}$	avg	(sd)	$>_{avg}$	$>_{t-test}$
frb-30-15-*	5	450	9.83	(1.86)	0%	0%	**9.46**	(2.00)	**80%**	**20%**
frb-35-17-*	5	595	**11.62**	(2.05)	**60%**	0%	11.82	(2.31)	40%	0%
frb-40-19-*	5	760	13.47	(1.92)	20%	0%	**12.85**	(2.22)	**80%**	**20%**
frb-45-21-*	5	945	15.40	(2.43)	0%	0%	**14.35**	(1.82)	**100%**	**80%**
frb-50-23-*	5	1150	16.24	(2.32)	20%	0%	**15.84**	(2.00)	**80%**	**20%**
frb-53-24-*	5	1272	18.15	(2.55)	0%	0%	**16.86**	(1.84)	**100%**	**80%**
frb-56-25-*	5	1400	17.85	(2.37)	20%	0%	**16.89**	(1.08)	**80%**	**40%**
frb-59-26-*	5	1534	18.40	(2.44)	40%	0%	**18.37**	(2.16)	**60%**	**20%**
C*.*	5	775	7.15	(1.2)	0%	0%	**5.6**	(0.3)	60%	40%
gen*_p0.9_*	5	320	1.7	(0.26)	0%	0%	**1.44**	(0.1)	20%	0%

TABLE 6.1 – *Comparaison de CPO et CPO-ACO sur le MKP, le QAP et le problème de clique maximum. Chaque ligne donne successivement : le nom de la classe d'instances, le nombre d'instances dans la classe (#I), le nombre moyen de variables dans ces instances (#X), les résultats obtenus par CPO (resp. CPO-ACO), à savoir, le pourcentage d'écart par rapport à la meilleure solution connue (moyenne (avg) et écart type (sd)), le pourcentage d'instances pour lesquelles CPO (resp. CPO-ACO) a obtenu de meilleurs résultats en moyenne ($>_{avg}$), et le pourcentage d'instances pour lesquelles CPO (resp. CPO-ACO) est donné meilleur par le test statistique t-test.*

de 95%) en utilisant le langage $\mathcal{R}$ [2] pour déterminer si les résultats sont significativement différents ou non. Pour chaque classe, nous avons indiqué le pourcentage d'instances pour lesquelles une approche a obtenu des résultats significativement meilleurs que l'autre (colonne $>_{t-test}$ du tableau 6.1). Pour le MKP, CPO-ACO est significativement meilleur que CPO sur 57 instances, alors qu'il n'est pas significativement différent sur 3 instances. Pour le QAP, CPO-ACO est significativement meilleur que CPO sur un grand nombre d'instances. Toutefois, CPO est meilleur que CPO-ACO sur une instance de la classe tai* du QAP. Pour le problème de clique maximum, CPO-ACO est significativement meilleur que CPO sur 32% d'instances, mais il n'est pas significativement différent sur toutes les autres.

6.4 Stratégie phéromonale (Var/Dom vs Vertex)

Dans le chapitre 5, nous avons vu que l'utilisation d'une structure de phéromone dédiée au problème considéré peut améliorer, de manière significative, les résultats d'un algorithme ACO. Cependant, comme ces structures de phéromones développées dans le chapitre 5 sont dédiées au problème d'ordonnancement de voitures, elles ne peuvent pas être utilisées par CPO-ACO. En effet, le but recherché cette fois-ci, est d'avoir des structures génériques applicables sur tous les COPs.

Cependant, il existe un cas où nous pouvons adapter la structure de phéromone de CPO-ACO pour le rendre plus performant. En effet, lorsque toutes les variables d'un problème sont binaires, nous pouvons considérer ce problème comme étant un problème de sélection d'un sous ensemble d'objets optimisant une fonction objectif. Dans ce cas là, le fait d'affecter une certaine valeur, parmi les deux valeurs possibles, à une variable peut être assimilé au fait que l'objet associé à cette variable est sélectionné. En d'autres termes, la décision qui consiste à choisir quelle valeur à affecter à une variable revient à décider si l'objet associé à cette variable doit être sélectionné ou non.

Il existe principalement deux stratégies phéromonales pour résoudre des problèmes de sélection de sous-ensembles [LM99, ASG04] que nous décrivons brièvement ci-après.

Soit un problème de sélection de sous-ensemble à partir d'un ensemble à n objets $O = \{o_1, o_2, ..., o_n\}$.

1. Vertex : avec cette stratégie, la phéromone est déposée sur les objets et une trace $\tau(o_i)$ est associée à chaque objet $o_i \in O$. Lors de la récompense d'une solution $\mathcal{S} \subseteq O$, la phéromone est déposée sur les objets $o_i \in \mathcal{S}$. Lors de la sélection d'un objet, nous utilisons la phéromone déposée sur chacun de ces objets pour calculer la probabilité de sélection de chacun d'eux, i.e., le facteur phéromonale associé à un objet $\in O$ est défini par $\tau(o_i)$

2. $\mathcal{R}$ est un environnement de calcul statistique et graphique. Il est gratuitement téléchargeable sur le site officiel du projet $\mathcal{R}$: "http ://www.r-project.org/"

2. Edge : avec cette stratégie, la phéromone est déposée sur les paires d'objets et une trace $\tau(o_i, o_j)$ est associée à chaque paire d'objets $\{o_i, o_j\} \subseteq O$. Lors de la récompense d'une solution $\mathcal{S} \subseteq O$, la phéromone est déposée sur tous les couples (o_i, o_j) avec o_i et o_j deux objets différents qui apparaissent dans la solution $\mathcal{S}$. Lors de la sélection d'un objet, le facteur phéromonale d'un objet candidat o_i est défini par l'équation 6.3 qui correspond à la somme des quantités de phéromone déposée sur les couples (o_i, o_s) où o_s est un objet déjà sélectionné. L'intuition derrière cette stratégie de phéromone est de mesurer l'interêt de sélectionner un objet donné sachant que l'on a déjà sélectionné certains objets et cela, sans considérer l'ordre dans lequel ces objets ont été sélectionnés.

$$\tau_{\mathcal{S}}(o_i) = \sum_{o_s \in \mathcal{S}} \tau(o_i, o_s) \tag{6.3}$$

Les deux structures phéromonales définies ci-dessus sont génériques et sont applicables sur n'importe quel problème de sélection de sous-ensembles comme le MKP ou le problème de clique maximum.

Ces deux stratégies phéromonales ont été comparées sur différents problèmes de sélection de sous-ensembles [LM99, ASG04]. Dans de nombreux cas, la stratégie Edge donne des résultats légèrement meilleurs que la stratégie Vertex, mais elle est également beaucoup plus longue (demande plus de temps calcul). Ainsi, nous proposons dans la suite d'utiliser la stratégie Vertex.

6.4.1 CPO-ACO-Vertex

Nous proposons ici une variante de CPO-ACO dédiée aux problèmes de sélection de sous-ensembles et utilisant la stratégie Vertex. Donc, la phéromone est déposée sur les variables (correspondant aux objets à sélectionner) et non pas sur les couples <variable, valeur>. Cette modification induit naturellement des changements dans l'algorithme CPO-ACO. Dans ce cas là, le problème se résume à trouver quelles sont les variables à sélectionner et non plus quelles sont les valeurs à affectées aux variables : quand une variable est sélectionnée, elle est systématiquement affectée à 1, signifiant ainsi que l'objet correspondant à cette variable est sélectionné. Les fonctions qui sont le plus affectées par ces changements sont les fonctions de choix de variables et de valeurs lors de la première et la deuxième phase. Ces changements peuvent être résumés comme suit :

Structure phéromonales

Nous associant à chaque variable $x \in \mathcal{X}$ une trace phéromonale τ_x. Cette quantité de phéromone représente l'intérêt de sélectionner l'objet associé à la variable x, ou autrement dit, d'affecter la variable x à 1.

La première phase CPO-ACO-Vertex

Les modifications apportées à CPO-ACO dans la première phase sont :

– Choix de variables : la prochaine variable x_i à affecter est choisie de manière aléatoire selon la probabilité :

$$\mathcal{P}_{x_i} = \frac{\tau_{x_i}^{\alpha} \,.\, impact(x_i)^{\beta}}{\sum_{x_j \in \mathcal{C}and} (\tau_{x_j}^{\alpha} \,.\, impact(x_j)^{\beta})} \tag{6.4}$$

avec $\mathcal{C}$and l'ensemble des variables non encore affectées.

– Choix de valeurs : quand une variable est sélectionnée, l'algorithme l'affecte systématiquement à 1.

La deuxième phase de CPO-ACO-Vertex

Les modifications apportées à CPO-ACO dans la deuxième phase sont :

– Choix de variables : la prochaine variable à affecter est choisie de manière déterministe et elle est la variable x_i qui maximise la formule suivante :

$$\tau_{x_i}^{\alpha} \,.\, impact(x_i)^{\beta} \tag{6.5}$$

– Choix de valeurs : Comme lors de la première phase, quand une variable est sélectionnée, l'algorithme l'affecte systématiquement à 1.

6.4.2 Evaluation expérimentales de CPO-ACO-Vertex

Pour vérifier l'intérêt d'utiliser la structure de phéromone Vertex pour des problèmes de sélection de sous-ensembles, nous avons mené des expérimentations sur des instances du MKP et du problème de clique maximum.

Pour le MKP, nous avons choisi les dix premières instances parmi les instances que nous avons utilisées lors des tests précédents. Pour le problème de clique maximum, nous avons choisi les cinq instances $frb-*-*$ et nous avons choisi dix instances $DIMACS$.

Pour le paramétrage, nous avons gardé exactement les mêmes paramètres que lors des tests précédents à l'exception de la durée des exécutions qui est fixée à une demi-heure par exécution.

Ci-dessous, nous donnons les définitions des algorithmes utilisés dans la suite de cette section :

– CPO : qui est CP Optimizer avec les paramètres par défaut.
– CPO-ACO : est l'algorithme à deux phases présenté dans la section précédente.
– CPO-ACO-Vertex-ph1 : est l'algorithme CPO-ACO-Vertex sauf qu'il n'utilise pas la deuxième phase. Nous avons utilisé cette variante pour montrer l'intérêt de la deuxième phase avec la nouvelle structure de phéromone.
– CPO-ACO-Vertex : est la variante décrite dans cette section.

	CPO		*CPO* − *ACO*		*CPO* − *ACO* − *Vertex* − *ph*1		*CPO* − *ACO* − *Vertex*	
Instance	min	avg	min	avg	min	avg	min	avg
5-100-0	0.16	1.33	0.15	0.67	0.21	0.77	0.00	**0.30**
5-100-1	0.00	0.9	0.00	**0.38**	0.44	1.1	0.00	0.67
5-100-2	0.00	0.8	0.00	**0.30**	0.00	0.67	0.05	0.38
5-100-3	0.33	1.2	0.25	0.67	0.02	0.77	0.00	**0.30**
5-100-4	0.18	2.1	0.10	0.46	0.18	0.63	0.10	**0.26**
5-100-5	0.33	1.33	0.00	0.72	0.07	0.9	0.00	**0.33**
5-100-6	0.00	0.9	0.00	0.55	0.00	0.76	0.00	**0.44**
5-100-7	0.00	1.1	0.00	0.78	0.00	0.62	0.00	**0.30**
5-100-8	0.24	1.77	0.04	**0.32**	0.16	0.85	0.16	0.52
5-100-9	0.00	1.9	0.00	0.67	0.18	0.9	0.00	**0.47**

TABLE 6.2 – *Ce tableau donne le nom de l'instance MKP, puis pour les quatre algorithmes il donne, la déviation minimale de l'optimum connu suivi de la déviation de l'optimum en moyenne sur les 30 exécutions.*

	CPO	*CPO* − *ACO*	*CPO* − *ACO* − *Vertex* − *ph*1	*CPO* − *ACO* − *Vertex*
CPO	-	$>_{0\%}$	$>_{0\%}$	$>_{0\%}$
CPO − *ACO*	$>_{100\%}$	-	$>_{50\%}$	$>_{20\%}$
CPO − *ACO* − *Vertex* − *ph*1	$>_{90\%}$	$>_{0\%}$	-	$>_{0\%}$
CPO − *ACO* − *Vertex*	$>_{100\%}$	$>_{70\%}$	$>_{70\%}$	-

TABLE 6.3 – *Résultats des tests de pertinence statistique : chaque ligne/colonne donne le pourcentage d'instances pour lesquelles l'algorithme sur la ligne est meilleur que l'algorithme sur la colonne en ce basant sur les résultats en moyenne donnés dans le tableau 6.2.*

		CPO		*CPO* − *ACO*		*CPO* − *ACO* − *Vertex* − *ph*1		*CPO* − *ACO* − *Vertex*	
Instance	Opt	max	avg	max	avg	max	avg	max	avg
frb59-26-1	59	52	51	54	52	54	52	54	**52.6**
frb59-26-2	59	53	52	54	52.3	54	**53**	54	**53**
frb59-26-3	59	53	52	53	52.2	54	52.6	54	**53**
frb59-26-4	59	52	52	54	**53**	54	**53**	54	**53**
frb59-26-5	59	53	51.2	53	51.4	59	54.2	59	**55**
C125.9	34	34	**34**	34	**34**	34	**34**	34	**34**
C250.9	44	44	**44**	44	**44**	44	**44**	44	**44**
C500.9	≥ 57	55	54.6	56	54.9	56	54.9	57	**56.1**
C1000.9	≥68	62	61	62	61.3	67	65.6	67	**66.2**
C2000.9	≥78	70	67	70	68.8	76	**73.3**	74	71.6
gen200_p0.9_44	44	44	43.4	44	**44**	44	**44**	44	**44**
gen200_p0.9_55	55	55	**55**	55	**55**	55	**55**	55	**55**
gen400_p0.9_55	55	52	51	52	51	52	51	53	**52**
gen400_p0.9_65	65	65	**65**	65	**65**	65	**65**	65	**65**
gen400_p0.9_75	75	75	**75**	75	**75**	75	**75**	75	**75**

TABLE 6.4 – *Ce tableau donne le nom de l'instance suivi de la valeur optimale connue pour cette instance, puis donne pour les quatre algorithmes, la meilleure valeur trouvée par les 30 exécutions et la moyenne.*

	CPO	*CPO* − *ACO*	*CPO* − *ACO* − *Vertex* − *ph*1	*CPO* − *ACO* − *Vertex*
CPO	-	$>_{0\%}$	$>_{0\%}$	$>_{0\%}$
CPO − *ACO*	$>_{46\%}$	-	$>_{0\%}$	$>_{0\%}$
CPO − *ACO* − *Vertex* − *ph*1	$>_{53\%}$	$>_{26\%}$	-	$>_{6\%}$
CPO − *ACO* − *Vertex*	$>_{53\%}$	$>_{33\%}$	$>_{26\%}$	-

TABLE 6.5 – *Résultats des tests de pertinence statistique : chaque ligne/colonne donne le pourcentage d'instances pour lesquelles l'algorithme sur la ligne est meilleur que l'algorithme sur la colonne en ce basant sur les résultats en moyenne donnés dans le tableau 6.4.*

6.4.3 Analyse des résultats

Le tableau 6.2 montre sur le MKP que CPO-ACO et CPO-ACO-Vertex sont meilleure en moyenne que CPO-ACO-Vertex-ph1 sur toutes les instances utilisées. Cela est du au fait que CPO-ACO-Vertex-ph1 n'effectue pas de phase d'intensification de la recherche (il n'utilise pas la deuxième phase). Ce tableau montre également que CPO-ACO-Vertex est meilleure en moyenne que CPO-ACO sur sept instances et il est moins bon que lui sur trois instances. Ces résultats montrent, en comparant CPO-ACO avec CPO-ACO-Vertex, l'intérêt de la nouvelle structure de phéromone Vertex. La comparaison de CPO-ACO-Vertex-ph1 et CPO-ACO-Vertex montre l'intérêt de la deuxième phase (la phase d'intensification) en utilisant *CP Optimizer*.

Remarquons que sur les pour le MKP, CPO-ACO et CPO-ACO-Vertex trouvent les solutions optimales (voir les colonnes *min* du tableau 6.2) dans la moitié des cas. Ce résultats sont intéressant, car ils sont obtenus sans l'utilisation d'heuristiques dédiées au problème de sac-à-dos.

Le tableau 6.3, donne les résultats des tests statistiques (test de Student avec un niveau de confiance de 95%) de comparaison des quatre algorithmes en se basant sur les résultats donnés dans le tableau 6.2. Pour chaque paire d'algorithmes, nous donnons le pourcentage d'instances sur lesquelles l'algorithme qui est sur la ligne du tableau est meilleur que l'algorithme sur la colonne. Nous remarquons que $CPO - ACO - Vertex$ est majoritairement le meilleur.

Le tableau 6.4, montre que CPO-ACO-Vertex est meilleur que CPO-ACO-Vertex-ph1 sur six instances ; il est moins bon que lui sur une instance et ils obtiennent les mêmes résultats sur le reste des instances. Ces résultats montrent que l'utilisation de la deuxième phase améliore légèrement les résultats de CPO-ACO-Vertex. La comparaison de CPO-ACO-Vertex avec CPO montre que CPO-ACO-Vertex obtient en moyenne des résultats au moins aussi bon que CPO et il est meilleur que lui en moyenne sur huit instances.

Le tableau 6.5, donne les résultats des tests statistiques (test de Student avec un niveau de confiance de 95%) de comparaison des quatre algorithmes en se basant sur les résultats donnés dans le tableau 6.4. Pour chaque paire d'algorithmes, nous donnons le pourcentage d'instances sur lesquelles l'algorithme qui est sur la ligne du tableau est meilleur que l'algorithme sur la colonne. Egalement, Nous remarquons que $CPO - ACO - Vertex$ est majoritairement le meilleur.

6.5 Utilisation d'heuristique dédiée dans CPO-ACO

Dans cette section, nous allons étudier le comportement de CPO-ACO-Vertex appliqué au MKP en utilisant une heuristique dédiée.

	$CPO-ACO-Vertex-ph1-h$		$CPO-ACO-Vertex-h$	
Instance	min	avg	min	avg
5-100-0	0.03	0.29	0.03	**0.22**
5-100-1	0	0.07	0	**0.06**
5-100-2	0.05	0.09	0.05	**0.08**
5-100-3	0	0.27	0	**0.23**
5-100-4	0	0.13	0	**0.12**
5-100-5	0	**0.06**	0	**0.06**
5-100-6	0	0.38	0	**0.33**
5-100-7	0	0.16	0	**0.13**
5-100-8	0	**0.16**	0	**0.16**
5-100-9	0	0.2	0	**0.08**

TABLE 6.6 – *Ce tableau donne le nom de l'instance MKP, puis pour les deux algorithmes il donne, la déviation minimale de l'optimum connu suivi de la déviation de l'optimum en moyenne sur les 30 exécutions.*

6.5.1 Heuristique de choix de variable pour le MKP

Pour tester CPO-ACO-Vertex avec une heuristique dédiée au problème de sac-à-dos, nous avons choisit d'utiliser l'heuristique définie dans la section 2.5.5. Par conséquence, dans les nouvelles expérimentations données dans la sous-section suivante, $\eta(x_i)$ (voir la formul 2.11) serra utilisé comme heuristique de choix de variables dans l'algorithme CPO-ACO-Vertex. En d'autres termes, $\eta(x_i)$ va remplacer le facteur $impact(x_i)$ dans les formules 6.4 et 6.5.

6.5.2 Evaluation expérimentales

Dans le tableau 6.6, nous avons comparé les deux algorithmes $CPO-ACO-Vertex-ph1-h$, $CPO-ACO-Vertex-h$ sur dix instances de MKP. $CPO-ACO-Vertex-ph1-h$ est l'algorithme CPO-ACO-Vertex qui est lancé sans l'utilisation de la deuxième phase et en utilisant l'heuristique de MKP définie ci-dessus. Cette heuristique est également utilisé dans $CPO-ACO-Vertex-h$ qui est l'algorithme à deux phases défini dans 6.4.1.

Les paramètres utilisés dans ces tests sont principalement ceux utilisés dans [ASG04] à savoir $\alpha = 1$, $\beta = 5$, $\rho = 0.01$, $nbAnts = 30$, $\tau_{min} = 0.01$, $\tau_{max} = 6$ et $nbCycle = 2000$. Pour $CPO-ACO-Vertex-ph1-h$, le paramètre $nbCycle = 2000$ correspond au critère d'arrêt de l'algorithme tandis que pour $CPO-ACO-Vertex-h$, ce paramètre est l'unique critère d'arrêt de la première phase.

6.5.3 Analyse des résultats

Le tableau 6.6, montre que les performances de CPO-ACO-Vertex sont améliorées en utilisant la deuxième phase et ceci même en utilisant une heuristique dédiée au problème.

6.6 Conclusion

Nous avons proposé une approche générique et exacte pour résoudre les COP définis par un ensemble de contraintes et une fonction objectif. Cette approche générique combine une approche B&P&B avec ACO. L'idée principale de cette combinaison est de bénéficier de l'efficacité (i) de ACO pour explorer l'espace de recherche et identifier les zones prometteuses (ii) de CP Optimizer pour exploiter fortement le voisinage des meilleures solutions trouvées par ACO. Cette combinaison nous permet d'atteindre un bon équilibre entre la diversification et l'intensification de la recherche : la diversification est principalement assurée au cours de la première phase par ACO et l'intensification est assurée par CP Optimizer au cours de la deuxième phase. Nous avons montré par des expériences sur trois COP différents que cette approche hybride améliore les performances de CP Optimizer.

Il est à noter que grâce à la nature modulaire de CP Optimizer qui sépare clairement la partie modélisation du problème de la partie résolution, la combinaison de ACO et CP Optimizer a été faite de manière naturelle. Par conséquent, le programme CPO-ACO utilisé était exactement le même pour les expériences sur les différents problèmes utilisés.

Nous avons proposé d'utiliser une structure de phéromone particulière pour les problèmes de sélection de sous-ensemble. Egalement, nous avons montré sur le problème de sac-à-dos multidimensionnel que le fait de rajouter une heuristique rend CPO-ACO (notamment CPO-ACO-Vertex) plus performant.

Le réglage des paramètres reste un point délicat. Pour l'instant les paramètres de CPO-ACO sont réglés en utilisant notre expérience. Cependant, on pense qu'une approche adaptative qui change dynamiquement les valeurs des paramètres pendant l'exécution peut augmenter l'efficacité et la robustesse de l'algorithme. Une telle approche est décrite dans le chapitre suivant.

Adaptation dynamique des paramètres de ACO

7

Sommaire

Dans ce chapitre, nous introduisons deux méthodes pour adapter dynamiquement deux paramètres de ACO (α et β). Dans un cas l'apprentissage est global à l'ensemble des variables du problème, dans l'autre, il est spécifique à chaque variable [1].

1. Les travaux présentés dans ce chapitre ont été intégralement publiés dans la conférence Learning In OptimizatioN (LION) en janvier 2009 [KAS09a] en Italie ainsi que dans la conférence JFPC 2009 à Orléans [KAS09b]

7.1 Motivations

Comme nous l'avons vu dans les chapitres précédents, l'optimisation par colonies de fourmis a démontré sont intérêt pour résoudre de nombreux problèmes combinatoires [DS04]. Par contre, la résolution d'un problème particulier avec ACO nécessite de trouver un compromis entre deux objectifs contradictoires : d'un coté, il est nécessaire d'intensifier la recherche de solutions autour des zones les plus prometteuses alors que de l'autre côté, il est nécessaire de diversifier la recherche pour découvrir de nouvelles zones de l'espace de recherche qui pourraient s'avérer porteuses de meilleures solutions. De manière générale, le contrôle du comportement de l'algorithme ACO vis-à-vis de ce double objectif (intensification/diversification) se fait à travers le réglage de l'ensemble de ses paramètres.

Le réglage de ces paramètres est un problème délicat, car il faut choisir entre deux tendances contradictoires : si l'on choisit des valeurs de paramètres qui privilégient la diversification, la qualité des solutions obtenues est souvent très bonne, mais au prix d'une convergence plus lente, et donc d'un temps de calcul plus long. Si par contre on choisit des valeurs qui favorisent l'intensification, l'algorithme sera amené à trouver de bonnes solutions plus rapidement, mais souvent sans converger vers la ou les meilleures solutions (il convergera vers des valeurs sous-optimales).

Les valeurs optimales des paramètres de l'algorithme ACO dépendent à la fois de l'instance du problème à traiter et du temps alloué à sa résolution. De plus, il peut s'avérer utile de changer les valeurs des paramètres durant la résolution même du problème afin de s'adapter au mieux aux caractéristiques locales de l'espace de recherche en cours d'exploration et surtout, afin de pouvoir alterner entre des étapes de diversification et des étapes d'intensification.

Pour améliorer le processus de recherche vis-à-vis de la dualité intensification/diversification, Battiti et al [BBM08] ont proposé d'exploiter l'historique de la recherche pour adapter automatiquement et dynamiquement les valeurs des paramètres, donnant ainsi naissance aux approches dites "réactives".

Dans ce chapitre, nous introduisons deux méthodes réactives permettant à ACO d'adapter dynamiquement certains de ses paramètres pendant la recherche de solution. Dans les deux méthodes, cette adaptation dynamique est réalisée avec ACO lui-même. Plus concrètement, nous allons introduire deux méthodes réactives pour l'adaptation dynamique des paramètres de ACO : dans la première méthode, les valeurs des paramètres sont fixées à chaque début de construction d'une solution ou d'un ensemble de solutions et elles ne sont pas modifiées durant la construction d'une même solution. Dans la deuxième méthode, les paramètres sont associés à chacune des variables du problème et avant d'affecter une de ces variables par une valeur, l'algorithme choisit d'abord une valeur pour chacun des paramètres impliqués dans le processus de sélection d'une valeur à cette variable. Notre avons évalué les deux méthodes proposées pour résoudre des problèmes de satisfaction de contraintes (MaxCSP).

Ce chapitre est organisé comme suit. Nous allons commencer par la description des deux méthodes proposées pour l'adaptation dynamique des paramètres de ACO. Ensuite, nous allons donner les résultats et la comparaison de ces deux méthodes ainsi que leur comparaison avec la version de ACO classique dans laquelle les paramètres ne sont pas adaptés (ou modifiés) durant l'exécution du programme. Egalement, nous allons comparer les résultats de ces deux méthodes avec les résultats des algorithmes existant dans l'état de l'art. Dans la dernière section, nous concluons en présentant des travaux proches ainsi que les évolutions que nous prévoyons d'explorer.

7.2 Utilisation d'ACO pour adapter dynamiquement α et β

7.2.1 Choix des paramètres à adapter dynamiquement

Les deux facteurs principaux de l'algorithme ACO qui lui permettent de balancer entre l'intensification et la diversification de la recherche sont : le poids du facteur phéromonale α et le poids du facteur heuristique β. La phéromone représente les informations apprises dynamiquement sur le problème en cours de résolution et l'heuristique représente les informations liées également au problème que l'on souhaite résoudre mais qui sont disponibles avant même le début de la résolution de ce problème.

Le cadre réactif proposé dans ce chapitre, vise à adapter uniquement les deux paramètres α et β qui ont une très forte influence sur le processus de construction de solution et notamment sur le processus de l'intensification et la diversification du processus de recherche de manière générale.

La valeur du facteur phéromonal α est fondamentale pour articuler l'intensification et la diversification. En effet, plus α est grand, plus la recherche est intensifiée autour des composants de solution portant d'importantes traces de phéromone, i.e., autour des composants qui sont intervenus dans la construction des meilleures solutions.

Une bonne valeur du poids de facteur heuristique β dépendent également de l'instance à résoudre. En effet on constate que l'importance du facteur heuristique varie souvent d'une instance à l'autre. De plus, pour une instance donnée son importance peut évoluer pendant la recherche de la solution.

Enfin, nous remarquons que les valeurs relatives de α et β sont bien sûr importantes, mais que leurs valeurs absolues le sont également —dans le cas contraire un seul paramètre aurait suffit.

Nous proposons d'employer ACO pour adapter dynamiquement les valeurs de α et de β. En particulier, nous proposons et comparons dans ce cadre deux méthodes différentes. La première méthode, appelé AS($\mathcal{GPL}$) et décrite dans 7.2.2, où les valeurs de α et de β sont fixées au début de la construction d'une solution et sont adaptées après chaque cycle, lorsque toutes les fourmis ont construit une solution. La deuxième méthode, appelé AS($\mathcal{DPL}$) et décrite dans 7.2.3, où les va-

leurs de α et de β sont définies pour chaque variable de sorte qu'elles changent pendant la construction d'une solution. Ces deux méthodes sont expérimentalement évaluées dans 7.3.

7.2.2 Description de AS($\mathcal{GPL}$)

AS($\mathcal{GPL}$) (Ant Solver with Global Parameter Learning) suit essentiellement l'algorithme 2 décrit dans la section 4.3.1 du chapitre 4, mais intègre de nouveaux dispositifs pour adapter dynamiquement α et β. Par conséquent, la valeur de α et la valeur de β ne sont plus à fournir comme paramètres en entrée de l'algorithme, mais leurs valeurs sont choisies avec la métaheuristique ACO à chaque cycle [2]. En revanche, l'utilisateur doit fournir un ensemble de valeurs possibles pour α et un autre ensemble de valeurs possible pour β. A partir de ces deux ensemble, le but est de sélectionner les meilleurs valeurs de α et β qui sont le mieux adaptées à l'instance de problème en cours de résolution.

Paramètres de AS($\mathcal{GPL}$) En plus des paramètres de Ant-Solver, c.-à-d., le nombre de cycles *nbCycles*, le nombre de fourmis *nbAnts*, le taux d'évaporation ρ, et les bornes minimales et maximales des traces de phéromone τ_{min} et τ_{max}, AS($\mathcal{GPL}$) est donc paramétré par un ensemble de nouveaux paramètres permettant d'adapter dynamiquement α et β. Ces nouveaux paramètres sont donnés ci-dessous :

- deux ensembles de valeurs $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ qui contiennent les ensembles de valeurs qui peuvent être respectivement considérées pour l'affectation de α et β ;
- les bornes minimale et maximale de la quantité de phéromone, $\tau_{min_{\alpha\beta}}$ et $\tau_{max_{\alpha\beta}}$;
- le taux d'évaporation $\rho_{\alpha\beta}$ qui sera utilisé dans le processus d'évaporation des phéromones déposées sur les valeurs possibles de α et β. Ce taux d'évaporation joue le même rôle que le taux d'évaporation de ACO.

Il est à noter que notre cadre réactif suppose que α et β prennent leurs valeurs dans deux ensembles discrets $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ qui doivent être connus a priori. Ces deux ensembles doivent contenir de bonnes valeurs, *c.-à-d.*, celles qui permettent à Ant-Solver de trouver les meilleurs résultats pour chaque instance de problème à résoudre. Comme discuté dans la section 7.3, nous proposons de choisir les valeurs de $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ en lançant Ant-Solver avec différentes affectations pour α et β sur un ensemble représentatif d'instances, puis d'initialiser $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ avec les ensembles des valeurs qui ont permis à Ant Solver de trouver les meilleurs résultats sur ces instances.

2. Nous avons comparé expérimentalement deux variantes de ce cadre réactif : une première variante où les valeurs sont choisies au début de chaque cycle (entre les lignes 2 et 3) de sorte que chaque fourmi considère les mêmes valeurs pendant le cycle, et une deuxième variante où les valeurs sont choisies par des fourmis avant de construire une affectation (entre les lignes 3 et 4). Les deux variantes obtiennent des résultats qui ne sont pas sensiblement différents. Par conséquent, nous considérons seulement la première variante qui est décrite dans cette section.

Structure de phéromone. Nous associons une trace de phéromone $\tau_\alpha(i)$ à chaque valeur $i \in \mathcal{I}_\alpha$ et une trace de phéromone $\tau_\beta(j)$ à chaque valeur $j \in \mathcal{I}_\beta$. Intuitivement, ces traces de phéromone représentent l'intérêt appris d'affecter respectivement α et β à i et à j. Pendant le processus de recherche, ces traces de phéromone sont maintenues entre les deux limites $\tau_{min_{\alpha\beta}}$ et $\tau_{max_{\alpha\beta}}$. Au début du processus de recherche, elles sont initialisées à $\tau_{max_{\alpha\beta}}$, car, encore une fois, nous utilisons le schéma de l'algorithme $\mathcal{M}ax\mathcal{M}in\mathcal{A}CO$ pour l'apprentissage effectué sur les paramètres.

Choix des valeurs pour α et β. A chaque cycle, (i.e., entre les lignes 2 et 3 de l'algorithme 2), α (resp. β) est affecté en choisissant $i \in \mathcal{I}_\alpha$ (resp. $i \in \mathcal{I}_\beta$) relativement à une probabilité $p_\alpha(i)$ (resp. $p_\beta(i)$) qui est proportionnelle à la quantité de phéromone déposée sur i, i.e.,

$$p_\alpha(i) = \frac{\tau_\alpha(i)}{\sum_{j \in \mathcal{I}_\alpha} \tau_\alpha(j)} \quad (\text{resp. } p_\beta(i) = \frac{\tau_\beta(i)}{\sum_{j \in \mathcal{I}_\beta} \tau_\beta(j)})$$

Mise à jour des traces de phéromone. Les traces de phéromone associées à α et β sont mises à jour à chaque cycle, entre les lignes 7 et 8 de l'algorithme 2. D'abord, chaque trace de phéromone $\tau_\alpha(i)$ (resp. $\tau_\beta(i)$) est évaporée en la multipliant par $(1 - \rho_{\alpha\beta})$. Ensuite la trace de phéromone associée à α (resp. β) est renforcée. La quantité de phéromone déposée sur $\tau_\alpha(\alpha)$ (resp. $\tau_\beta(\beta)$) est inversement proportionnelle au nombre de contraintes violées $\mathcal{A}_{best}$ par la meilleure affectation construite pendant le cycle. Ainsi, les valeurs de α et β qui ont permis aux fourmis de construire les meilleures affectations recevront les plus grandes quantités de phéromone.

7.2.3 Description de AS($\mathcal{DPL}$)

Le cadre réactif décrit à la section précédente adapte dynamiquement α et β à chaque cycle, mais il considère la même affectation pour toutes les décisions élémentaires d'un même cycle. Nous décrivons maintenant une autre variante réactive nommée AS($\mathcal{DPL}$) (Ant Solver with Distributed Parameter Learning). Le principe est de choisir de nouvelles valeurs pour α et β à chacune des étapes de la construction d'une solution, c.-à-d., chaque fois qu'une fourmi doit choisir une valeur pour une variable. Le but est d'ajuster l'affectation de α et de β pour chaque variable du CSP.

Paramètres de AS($\mathcal{DPL}$). Les paramètres de AS($\mathcal{DPL}$) sont les mêmes que ceux de AS($\mathcal{GPL}$). Les différences se situes au niveau des traces de phéromones utilisées et les formules de sélection des valeurs pour les deux paramètres α et β.

Structure phéromonale Nous associons une trace de phéromone $\tau_\alpha(x_k, i)$ (resp. $\tau_\beta(x_k, i)$) à chaque variable $x_k \in \mathcal{X}$ et chaque valeur $i \in \mathcal{I}_\alpha$ (resp. $i \in \mathcal{I}_\beta$). Intuiti-

vement ces traces de phéromone représentent l'intérêt appris d'affecter la valeur i à α (resp. β) lorsqu'on choisit une valeur pour la variable x_k. Pendant le processus de recherche de solution ces traces de phéromone sont maintenues entre les deux bornes $\tau_{min_{\alpha\beta}}$ et $\tau_{max_{\alpha\beta}}$; elles sont initialisées à $\tau_{max_{\alpha\beta}}$.

Choix des valeurs pour α et β. A chaque étape de la construction d'une affectation, avant de choisir une valeur v pour une variable x_k, α (resp. β) est positionné en choisissant une valeur $i \in \mathcal{I}_\alpha$ (resp. $i \in \mathcal{I}_\beta$) en fonction de la probabilité $p_\alpha(x_k, i)$ (resp. $p_\beta(x_k, i)$) qui est proportionnelle à la quantité de phéromone associée à i pour x_k, i.e.,

$$\begin{cases} p_\alpha(x_k, i) = \dfrac{\tau_\alpha(x_k, i)}{\sum_{j \in \mathcal{I}_\alpha} \tau_\alpha(x_k, j)} \\ \\ p_\beta(x_k, i) = \dfrac{\tau_\beta(x_k, i)}{\sum_{j \in \mathcal{I}_\beta} \tau_\beta(x_k, j)}) \end{cases} \tag{7.1}$$

Mise à jour des traces de phéromone. Les traces de phéromone associées à α et β sont mises à jour à chaque cycle : entre les lignes 7 et 8 de l'algorithme 2. D'abord chaque trace de phéromone $\tau_\alpha(x_k, i)$ (resp. $\tau_\beta(x_k, i)$) est évaporée en la multipliant par $(1 - \rho_{\alpha\beta})$. Ensuite, une quantité de phéromone est déposée sur les traces associées aux valeurs α et β qui ont été utilisées pour construire la meilleure affectation du cycle ($\mathcal{A}_{best}$) : pour chaque variable $x_k \in X$, si α (resp. β) a été affecté à i pour choisir la valeur affectée à x_k lors de la construction de $\mathcal{A}_{best}$, alors, $\tau_\alpha(x_k, i)$ (resp. $\tau_\beta(x_k, i)$) est incrémenté de $1/cost(\mathcal{A}_{best})$.

7.3 Résultats expérimentaux

7.3.1 Instances considérées

Dans cette section, nous allons donner les résultats des deux algorithmes AS($\mathcal{GPL}$) et AS($\mathcal{DPL}$) sur le benchmark de maxCSP qui a été utilisé pour la compétition [DLR07] CSP 2006. Nous avons considéré les 686 instances binaires de ce benchmark. Ces instances sont toutes définies en extension. Parmi ces 686 instances, 641 sont résolues à l'optimal, à la fois par la version statique de Ant Solver (la version originale de Ant-Solver) et par les deux versions réactives que nous avons introduit, les temps de réponses des trois variantes de Ant-Solver ne différant pas significativement. Nous avons donc concentré notre expérimentation de la section 7.3.3 sur les 25 instances les plus difficiles. Parmi ces 25 instances, nous avons sélectionné 10 instances représentatives dont les caractéristiques sont décrites dans la table 7.1.

Pour les 641 instances résolues à l'optimal, la meilleure solution est connue. Cependant pour les autres instances, la solution optimale n'est pas connue. Pour chacune de ces dernières instances, nous avons considéré la meilleure solution

Nb	Nom	X	D	C	B
1	brock-400-1	401	2	20477	378
2	brock-400-2	401	2	20414	378
3	mann-a27	379	2	1080	252
4	san-400-0.5-1	401	2	40300	392
5	rand-2-40-16-250-350-30	40	16	250	1
6	rand-2-40-25-180-500-0	40	25	180	1
7	rand-2-40-40-135-650-10	40	40	135	1
8	rand-2-40-40-135-650-22	40	40	135	1

TABLE 7.1 – *Pour chaque instance, Nom, X, D, C, et B indiquent respectivement le nom, le nombre de variables, la taille des domaines des variables, le nombre de contraintes, et le nombre de contraintes violées par la meilleure solution trouvée lors de la compétition de 2006.*

connue et qui est la solution trouvée par le meilleur solveur de la compétition par rapport à cette instance.

Nous avons également comparé nos deux algorithmes réactifs avec les meilleurs solveurs de la compétition et cette fois-ci, nous avons considéré les résultats expérimentaux pour toutes les instances du benchmark de la compétition.

7.3.2 Contexte d'expérimentation

Pour régler les paramètres de la version originale de Ant Solver nous l'avons fait tourner en faisant varier ses paramètres sur un sous-ensemble représentatif de 100 instances choisies parmi les 686 instances de la compétition. Ces 100 instances contiennent les 25 instances les plus difficiles. Puis nous avons sélectionné le paramétrage avec lequel Ant-Solver trouve en moyenne les meilleurs résultats, i.e., $\alpha = 2$, $\beta = 8$, $\rho = 0.01$, $\tau_{min} = 0.1$, $\tau_{max} = 10$, and $nbAnts = 15$. Nous avons limité le nombre de cycles à 10000, mais on constate que le nombre de cycles réellement nécessaire à l'obtention de la meilleure solution est souvent très inférieur. Dans cette section, AS(Static) se réfère à Ant Solver utilisé avec ce paramétrage.

Nous avons également réglé α et β séparément pour chaque instance (tout en conservant inchangées les valeurs des autres paramètres). Dans cette section, AS$(Tuned)$ se réfère à Ant Solver dont les paramètres statiques sont affectés avec la meilleure valeur pour l'instance considérée.

Pour les deux variantes réactives de Ant Solver (AS($\mathcal{GPL}$) et AS($\mathcal{DPL}$)), les paramètres non appris gardent les mêmes valeurs que pour $AS(Tuned)$ et AS $(Static)$, i.e., $\rho = 0.01$, $\tau_{min} = 0.1$, $\tau_{max} = 10$, et $nbAnts = 15$.

Pour les nouveaux paramètres, qui ont été introduits pour adapter dynamiquement α et β, nous avons positionné $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ aux valeurs qui donnent les meilleurs résultats en moyenne pour AS(Static), i.e., $\mathcal{I}_\alpha = \{0, 1, 2\}$ et $\mathcal{I}_\beta = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. Quant au taux d'évaporation, et aux bornes supérieures et inférieures des traces de phéromone, nous avons utilisé les même valeurs que pour AS(Static), i.e., $\rho_{\alpha\beta} = 0.01$, $\tau_{min_{\alpha\beta}} = 0.1$, $\tau_{max_{\alpha\beta}} = 10$.

	1	2	3	4	5	6	7	8
AS($\mathcal{DPL}$)/AS($\mathcal{GPL}$)	=	=	=	=	>	>	=	>
AS($\mathcal{DPL}$)/AS(Static)	=	>	>	>	=	>	=	=
AS($\mathcal{DPL}$)/AS(Tuned)	=	>	=	=	=	>	<	<
AS($\mathcal{GPL}$)/AS(Static)	=	>	>	>	<	=	<	=
AS($\mathcal{GPL}$)/AS(Tuned)	=	>	=	=	<	=	<	<
AS(Static)/AS(Tuned)	=	=	<	<	=	=	<	<

TABLE 7.2 – *Résultats des tests de pertinence statistique : chaque ligne compare deux variantes X/Y et donne pour chaque instance les résultats des tests sur 50 exécutions,* i.e., *= (resp. < et >) si X n'est pas significativement différent de Y (resp. est significativement moins bon ou meilleur que Y).*

	AS(Tuned)				AS(Static)		AS($\mathcal{GPL}$)		AS($\mathcal{DPL}$)	
Nb	#const	(sdv)	α	β	#const	(sdv)	#const	(sdv)	#const	(sdv)
1	374.84	(0.7)	1	6	374.92	(0.39)	**374.**	(1.01)	**374.**	(1.01)
2	373.12	(0.26)	1	5	374.68	(1.09)	**371.32**	(1.09)	371.48	(1.31)
3	253.88	(0.26)	1	6	254.62	(0.49)	**253.74**	(0.44)	253.96	(0.28)
4	387.2	(0.11)	1	8	388.04	(1.77)	**387.**	(0)	**387.**	(0)
5	**1.**	(0)	2	8	**1.**	(0)	1.02	(0.14)	**1.**	(0)
6	1.02	(0.02)	2	6	1.02	(0.14)	1.04	(0.19)	**1.**	(0)
7	**1.**	(0)	1	6	1.12	(0.32)	1.66	(0.47)	1.48	(0.5)
8	**1.**	(0)	1	5	1.08	(0.27)	1.12	(0.32)	1.08	(0.27)

TABLE 7.3 – *Comparaison expérimentale des meilleures solutions trouvées. Pour chaque variante de Ant Solver considérée, chaque ligne indique le nombre de contraintes violées dans la meilleure solution (moyenne et écart-type sur 50 exécutions). Pour AS(Tuned), nous donnons également les valeurs de α et β qui ont été utilisées.*

7.3.3 Comparaison expérimentale de AS(Tuned), AS(Static), AS($\mathcal{GPL}$) et AS($\mathcal{DPL}$)

La table 7.3 indique la meilleure affectation pour α et β qui a été utilisée pour exécuter AS(Tuned). Elle montre que les meilleures valeurs de ces deux paramètres sont clairement différentes d'une instance à l'autre. Nous avons également observé qu'à la fin du processus de recherche de AS($\mathcal{GPL}$), les traces de phéromone utilisées pour les affecter portent des valeurs différentes d'une instance à l'autre. Ceci est particulièrement marqué pour le paramètre β, ce qui indique que la pertinence du facteur heuristique dépend de l'instance considérée.

La table 7.3 compare également le nombre de contraintes violées dans la meilleure solution trouvée après 10000 cycles pour les quatre variantes de Ant-Solver. Comme les différences entre ces variantes sont plutôt faibles pour certaines instances, nous avons réalisé des tests statistiques pour vérifier la pertinence de ces différences. Les résultats de ces tests sont donnés dans la table 7.2. Elle met en évidence que les variantes réactives sont toujours au moins aussi performantes que AS(Static), excepté pour les instances 5 et 7 qui sont mieux résolues par AS(Static) que par AS($\mathcal{GPL}$). Elle nous indique également que les deux variantes réactives sont au moins au même niveau de performance que AS(Tuned), et qu'elles le surpassent même pour de nombreuses instances – en fait, elles le surpassent sur toutes les instances sauf sur deux instances pour AS($\mathcal{DPL}$) et trois instances pour

	AS(Tuned)			AS(Static)			AS($\mathcal{GPL}$)			AS($\mathcal{DPL}$)		
	cycles		time	cycles		time	cycles		time	cycles		time
Nb	avg	(sdv)	avg	avg	(sdv)	avg	avg	(sdv)	avg	avg	(sdv)	avg
1	44	(7)	4	30	(4)	2	2717	(516)	98	2501	(491)	93
2	2247	(477)	140	323	(194)	12	2668	(463)	96	3322	(415)	125
3	2193	(309)	542	1146	(335)	160	2714	(432)	399	2204	(295)	328
4	710	(213)	123	347	(174)	39	316	(38)	34	112	(14)	12
5	394	(32)	5	394	(32)	4	379	(44)	5	412	(37)	5
6	476	(19)	26	606	(23)	13	507	(49)	18	579	(23)	15
7	2436	(166)	160	1092	(152)	31	736	(126)	39	1557	(266)	52
8	1944	(120)	140	884	(65)	29	1302	(286)	66	1977	(252)	87

TABLE 7.4 – *comparaison expérimentale du nombre de cycles (moyenne et écart-type sur 50 exécutions) et du temps CPU exprimé en secondes (moyenne sur 50 exécutions) nécessaires pour atteindre la meilleure solution.*

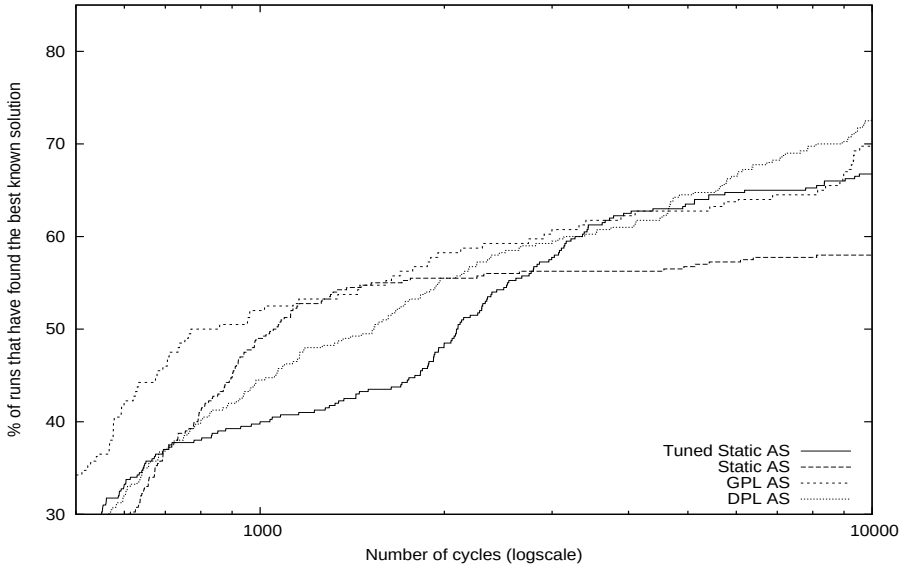

FIGURE 7.1 – *Pourcentage d'exécutions qui ont trouvé la solution optimale.*

AS($\mathcal{GPL}$)). Enfin la table indique également que AS($\mathcal{DPL}$) n'est pas significativement différent de AS($\mathcal{GPL}$) pour cinq instances, et le dépasse sur 3 instances.

La table 7.4 compare le nombre de cycles et le temps CPU nécessaire à l'obtention de la meilleure solution pour les différentes variantes de Ant-Solver. Nous notons d'abord que la surcharge en temps de calcul induite par la mise en oeuvre de nos méthodes réactives n'est pas significative ; les variantes de Ant-Solver mettent un temps comparable pour réaliser un cycle de calcul d'une instance à l'autre. Nous remarquons aussi que la variante AS(Static) converge souvent plus rapidement que AS(Tuned).

Pour comparer les quatre variantes de Ant Solver pendant l'ensemble de la recherche de solution, et non seulement à la fin des 10000 cycles, la figure 7.1 représente l'évolution du pourcentage d'exécutions qui ont trouvé la solution op-

				VB-Solver		AS($\mathcal{DPL}$)	
Bench	#I	#C	$\sum$ best known	$\sum cost$	#I^{best}	$\sum cost$	#I^{best}
brock	4	56381	1111	1123	2	**1111**	**4**
hamming	4	14944	460	463	1	**460**	**4**
mann	2	1197	281	**281**	**2**	283	1
p-hat	3	312249	1472	1475	1	**1472**	**3**
san	3	48660	687	692	2	**687**	**3**
sanr	1	6232	182	183	0	**182**	**1**
dsjc	1	736	19	20	0	**19**	**1**
le	2	11428	2869	2925	1	**2869**	**2**
graphw	6	16993	416	420	4	**416**	**6**
scenw	27	29707	809	904	25	**809**	**27**
tightnesso.5	15	2700	15	**15**	**15**	16	14
tightnesso.65	15	2025	15	**15**	**15**	18	12
tightnesso.8	15	1545	21	**22**	**13**	25	10
tightnesso.9	15	1260	26	**30**	**11**	31	10

TABLE 7.5 – *Comparaison expérimentale de AS($\mathcal{DPL}$) avec les solveurs de la compétition 2006. Chaque ligne indique le nom du benchmark, le nombre d'instances dans ce benchmark (#I), le nombre total de contraintes dans ces instances (#C), et le nombre de contraintes violées en considérant, pour chaque instance, le meilleur résultat trouvé ($\sum_{best\ known}$). Puis, nous donnons les meilleurs résultats obtenus pendant la compétition : pour chaque instance, nous avons considéré le meilleur résultat des 9 solveurs de la compétition et nous donnons le nombre de contraintes qui sont violées ($\sum cost$) suivi du nombre d'instances pour lesquelles la meilleure solution a été trouvée (#I^{best}). Enfin, nous donnons les résultats obtenus avec AS($\mathcal{DPL}$) : le nombre de contraintes violées ($\sum cost$) suivi du nombre d'instances pour lesquelles la meilleure solution connue a été trouvée (#I^{bests}).*

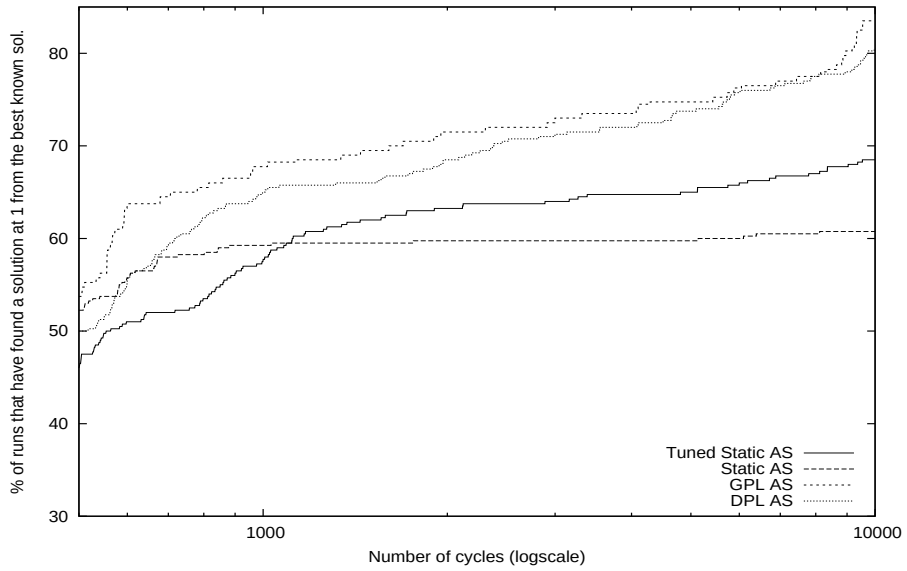

FIGURE 7.2 – *Pourcentage d'exécutions qui ont trouvé une solution optimale (à une contrainte près).*

timale, en fonction du nombre de cycles. Notons que dans les courbes que nous donnons dans cette section, nous avons considéré les meilleures solutions connues des instances pour lesquelles la solution optimale n'a pas été prouvée. La figure indique que AS(Static) peut trouver la solution optimale pour plus de la moitié des exécutions avant le 2000^{ieme} cycle. Cependant, après 2000 cycles, le pourcentage des exécutions résolues à l'optimalité ne croit que faiblement. AS(Tuned) affiche un comportement assez diffèrent : il nécessite plus de cycles pour converger de sorte qu'il trouve moins souvent la solution optimale au début du processus de recherche ; cependant, il présente des performances nettement supérieures à AS(Static) à la fin des 10000 cycles. Considérons par exemple les instances 2, 3 et 8 : la table 7.3 nous montre que AS(Tuned) à besoin de plus de cycles que AS(Static) pour les résoudre.

Egalement, la figure 7.1 nous indique que AS($\mathcal{GPL}$) présente de meilleures performances que les trois autres variantes pendant les 2000 premier cycles, alors qu'après 2000 cycles AS($\mathcal{GPL}$), AS($\mathcal{DPL}$) et AS(Tuned) sont assez proches et ont toutes de meilleures performances que AS(Static). Enfin, à la fin du processus de recherche, AS($\mathcal{DPL}$) est légèrement meilleur que AS($\mathcal{GPL}$) qui lui-même est légèrement meilleur que AS(Tuned).

La figure 7.2 montre l'évolution du pourcentage d'exécutions qui sont proches de l'optimalité, *i.e.*, les solutions optimales ou les solutions qui violent une contrainte de plus que la solution optimale. Elle montre que AS($\mathcal{GPL}$) trouve les solutions quasi-optimales plus rapidement que AS($\mathcal{DPL}$), qui lui-même affiche une meilleure performance que les variantes statiques de Ant Solver. De

plus, AS(Static) est meilleur que AS(Tuned) au début de la recherche de solution, mais il est généralement dépassé par AS(Tuned) au-delà des 1000 cycles.

7.3.4 Comparaison expérimentale de AS($\mathcal{DPL}$) avec les solvers de la compétition 2006

Nous comparons maintenant AS($\mathcal{DPL}$) avec les solveurs de maxCSP de la compétition 2006. Il y avait neuf solveurs, parmi lesquels huit sont basés sur des approches complètes, et un est basé sur une méthode de recherche locale incomplète. Pour la compétition, chaque solveur a bénéficié d'une durée allant jusqu'à 40 minutes sur un 3GHz Intel Xeon (voir [DLR07] pour plus de détails). Pour chaque exemple, nous avons comparé AS($\mathcal{DPL}$) avec le meilleur résultat trouvé pendant la compétition (par l'un des 9 solvers). Plus précisément, nous avons comparé AS($\mathcal{DPL}$) avec un algorithme meilleur que le meilleur de cette compétition, car nous l'avons comparé avec un algorithme à qui nous avons attribué le meilleur résultat connu pour chacune des instances. Nous appelons cet algorithme par le VB-Solver ("Virtual Best Solver"). Nous avons également limité AS($\mathcal{DPL}$) à 40 minutes, mais il a été exécuté sur un ordinateur moins puissant (Intel P4 Dual Core à 1.7 GHz). Nous ne rapportons pas les temps CPU car ils ont été obtenus sur différents ordinateurs. Le but ici est d'évaluer la qualité des solutions trouvées par AS($\mathcal{DPL}$).

Cette comparaison a été faite sur les 686 instances binaires définis en extension. Ces instances ont été regroupées dans 45 benchmarks. Parmi ces 45 benchmarks, il y avait 31 benchmarks pour lesquels AS($\mathcal{DPL}$) et les meilleurs solveurs de la compétition ont trouvé les mêmes valeurs pour chaque instance. Par conséquent, nous ne montrons les résultats que pour les 14 benchmarks pour lesquels AS($\mathcal{DPL}$) et VB-Solver ont obtenu des résultats différents (pour au moins une instance du benchmarks).

La table 7.5 donne les résultats pour ces 14 benchmarks. Elle montre que AS($\mathcal{DPL}$) a de meilleures performances que les VB-Solver pour 9 benchmarks. Plus précisément, AS($\mathcal{DPL}$) a amélioré les meilleures solutions trouvées par un solveur de la compétition pour 19 instances. Par contre, il n'a pas trouvé la meilleure solution pour 15 instances ; parmi ces 15 instances, 14 appartiennent à la série `tightness*` qui apparait clairement plus difficile pour AS($\mathcal{DPL}$) (notons cependant qu'aucun solveur de la compétition n'a été capable de trouver la solution optimale pour 6 de ces instances).

7.4 Conclusion

Nous avons présenté deux algorithmes réactifs pour adapter automatiquement et dynamiquement les valeurs du poids α du facteur phéromonal et du poids β du facteur heuristique qui ont une forte influence sur l'articulation intensification/diversification de la recherche dans ACO. Le but est double : nous visons

en premier lieu à libérer l'utilisateur du problème délicat du réglage de ces paramètres ; nous visons également à améliorer les résultats sur des instances difficiles.

Les premiers résultats expérimentaux sont très encourageants. En effet, dans la plupart des cas les variantes réactives obtiennent les performances d'une variante statique dont le paramétrage a été spécialement réglé pour chaque instance ; elles les surpassent même sur quelques instances.

Perspectives Il serait intéressant d'évaluer notre cadre réactif sur d'autres mises en oeuvre de ACO afin de qualifier sont niveau de généricité. En particulier, il sera intéressant de comparer les variantes réactives sur d'autres problèmes : pour certains problèmes tels que le Voyageur de Commerce, il est plus probable que l'association de paramètres à chaque composant de solution (à chaque ville) ne soit pas intéressante, tandis que sur d'autres problèmes, tels que le sac à dos multidimensionnel ou les problèmes d'ordonnancement, nous pensons que ceci devrait améliorer le processus de recherche.

Une limite de notre cadre réactif se situe dans le fait que l'espace de recherche pour les valeurs de paramètre doit être connu à l'avance et discrétisé.

Conclusion générale

Dans cette thèse, nous avons étudié les capacités de la métaheuristique d'optimisation par colonies de fourmis à s'intégrer dans un langage de programmation par contraintes pour la résolution des problèmes d'optimisation combinatoire. Nous avons commencé dans la première partie par un tour d'horizon de l'état de l'art sur les différents concepts traités et/ou utilisés dans cette thèse. Nous avons donné les définitions de bases des (1) problèmes de satisfaction de contraintes d'optimisation combinatoire (2) des approches de résolution exactes et métaheuristiques tout en insistant sur les avantages et les inconvénients de chacune d'elles (3) des techniques de gestion des contraintes dans les métaheuristiques. Dans la deuxième partie nous avons exposé nos principales contributions à savoir : (1) Intégration des algorithmes de type ACO dans un langage de programmation par contraintes notamment *IBM ILOG Solver* pour la résolution des problèmes de satisfaction de contraintes (2) proposition d'un algorithme hybride et générique où ACO est intégré dans *IBM CP Optimizer* pour résoudre des problèmes d'optimisation combinatoires (3) Réflexion et proposition d'une stratégie capable d'adapter dynamiquement les paramètres de ACO.

7.5 Nos contributions

Intégration d'ACO à un langage de PPC pour résoudre des CSPs :

Dans ces travaux, nous avons intégré ACO dans un langage de PPC pour résoudre les problèmes de satisfaction de contraintes. Nous avons développé une approche dans laquelle ACO à été intégré à la bibliothèque de programmation par contraintes développée par ILOG (ILOG Solver).

Dans cette nouvelle approche, le problème à résoudre est décrit en termes de contraintes dans le langage d'IBM ILOG Solver et il est résolu par notre algorithme hybride. Cet algorithme utilise les procédures de propagation et de vérification des contraintes prédéfinies dans la bibliothèque d'ILOG Solver. Cependant, la procédure de prise de décisions est remplacée par l'algorithme ACO.

Pour valider nos idées, nous avons choisi de tester notre approche sur le problème d'ordonnancement de voitures. Ce dernier, est un problème de référence utilisé au sein de la communauté PPC pour tester les performances et l'efficacité de leurs algorithmes.

L'algorithme ACO que nous avons développé prend plusieurs paramètres en

entrée. Nous avons étudié l'influence de ces paramètres sur le processus de résolution. Egalement, nous avons étudié et comparé l'influence des différentes formes de structure d'apprentissages (phéromonales) utilisées par notre algorithme hybride.

Dans le souci d'être compétitif, nous avons augmenté notre ACO avec une heuristique dédiée au problème d'ordonnancement de voitures. Car comme tout algorithme de type ACO, les performances de notre système sont très dépendantes de l'utilisation d'une heuristique. Nous avons pris l'une des meilleures heuristiques proposées dans l'état de l'art et nous l'avons améliorée puis appliquée.

Les résultats que nous avons obtenus sont encourageants, car en dépit de sa généricité, l'algorithme que nous avons proposé, à savoir Ant-CP, donne des résultats comparables aux algorithmes ACO développés pour résoudre le problème d'ordonnancement de voitures. Cependant, Ant-CP est un algorithme basé sur la recherche incomplet.

Les travaux que nous avons développés sur l'intégration de ACO dans la PPC pour résoudre les CSP ont été publiés dans [KAS08a, KAS07, KAS08b, KAS08c].

Intégration d'ACO à un langage de PPC pour résoudre des COPs :

Les études menées dans ces travaux nous ont permis de proposer une extension de l'API de *IBM ILOG CP Optimizer*. Par cette extension, nous avons introduit un algorithme hybride complet et générique qui fonctionne en deux étapes. Lors de la première étape, l'algorithme commence par échantillonner l'espace des solutions en lançant les fourmis qui utilisent CP Optimizer pour trouver des solutions respectant toutes les contraintes du problème. De plus, durant cette première étape, les fourmis collectent des informations liées au problème traité. Ces informations reflètent l'expérience des fourmis acquise durant leurs recherches des solutions. Lors de la deuxième étape, IBM CP Optimizer est lancé pour faire une recherche complète sur l'espace des solutions en utilisant les traces de phéromones récoltées lors de la première étape comme heuristique d'ordre de variables et/ou de valeurs.

Nous avons démontré l'efficacité de cet algorithme hybride à travers des tests sur différents problèmes. La plupart de ces résultats sont publiés dans [KAS10b, KAS10a].

Adaptation dynamique et automatique des paramètres d'ACO :

Au cours de cette étude, nous avons testé plusieurs versions d'ACO avec différents réglages des paramètres sur des problèmes *MaxCSP* (des problèmes de satisfaction de contraintes sur contraints). Ainsi, nous avons pu proposer une nouvelle stratégie dynamique et indépendante du problème traité pour le réglage des paramètres d'ACO.

L'idée derrière cette nouvelle stratégie vient de l'observation suivante : généralement les heuristiques sont pensées pour qu'elles soient utilisées de manière

cohérente sur l'ensemble du problème à résoudre. Par exemple, si nous voulons résoudre un problème en utilisant une heuristique donnée, alors cette heuristique sera utilisée de la même manière au niveau de tous les points de décisions de la procédure de recherche. Sachant que la structure de l'espace de recherche d'un problème peut considérablement varier d'une instance à l'autre, nous pensons qu'une bonne stratégie d'utilisation d'une heuristique doit être capable d'adapter dynamiquement la manière dont cette heuristique est utilisée en fonction de la forme ou de la structure de l'espace de recherche de l'instance à résoudre. Cette stratégie devrait être dynamique, car en pratique il est impossible d'écrire une nouvelle heuristique ou de chercher les bonnes valeurs des paramètres à chaque nouveau lancement de l'algorithme.

Dans [KAS09a, KAS09b], nous avons proposé une nouvelle stratégie adaptative qui permet à l'algorithme ACO de régler lui-même certains de ses paramètres pendant la résolution d'un problème.

7.6 Perspectives

Plusieurs perspectives sont à envisager pour chacune de nos contributions présenté dans cette thèse. Cependant, toutes ces perspectives tournent autours de l'étude de la généricité des concepts nouveaux introduits et également autours d'améliorations possibles de nos algorithmes. Ci-dessus, nous citons les principales perspectives que nous envisageons :

- Dans tous les algorithmes ACO que nous avons introduit dans cette thèse, à savoir Ant-CP, CPO-ACO, AS($\mathcal{GPL}$) et AS($\mathcal{DPL}$), le temps allouer à recherche des solutions est généralement pas utilisé dans sa totalité. En effet, une fois que ces algorithmes ont convergé, i.e., toutes les fourmis construisent les mêmes affectations (ou solutions), le temps restant est perdu car aucune amélioration des solutions ne peut être apportée. Il serait donc intéressant d'introduire un mécanisme efficace capable de détecter qu'un algorithme ACO est en état de stagnation et donc, relancer la recherche depuis le début (faire un Restart) ou en biaisant les phéromones de manière à ce que l'état de stagnation soit levé. nous pouvons envisager pour cela d'utiliser des indicateurs tel que le taux de ré-échantillonnage et/ou le taux de similarité des solutions construites ou encore, utiliser le $\lambda - branching$ pour détecter que la structure de phéromone a convergée [DS04].
- Dans le chapitre 5, nous avons proposé l'algorithme Ant-CP pour la résolution de CSPs et dans le chapitre 6, nous avons proposé l'algorithme CPO-ACO pour la résolution des COPs pour lesquels construire une solution est facile. Il serait intéressant de trouver un moyen de traiter avec CPO-ACO les COPs qui contiennent des contraintes dures pour ainsi avoir un seul algorithme qui sera une unification des deux algorithmes Ant-CP et CPO-ACO. Par exemple, nous pouvons envisager de traiter les COPs contenant des contraintes dures comme étant des problèmes à deux fonctions objec-

tif (problèmes multi-objectifs), une fonction objectif à optimiser (celle déjà dans le CPO) et une autre fonction objectif associée aux nombre de variables affectées.

- Dans le dernier chapitre de cette thèse, nous avons introduit un mécanisme générique et dynamique qui permet à un algorithme d'apprendre les bonnes valeurs de ses paramètres. Il serait intéressant d'appliquer ce mécanisme d'apprentissage des paramètres sur d'autres métaheuristiques ou de manière générale, sur d'autres algorithmes dont les performances dépendent de leurs paramètres.
- Egalement, il serait intéressant d'étudier le comportement, les limites et les extensions possibles de nos algorithmes, notamment CPO-ACO, sur d'autres problèmes pour vérifier leurs robustesses.
- Un autre point important à citer en perspective est de tester CPO-ACO sur différents problèmes en utilisant des heuristiques dédiés à ces problèmes pour voir si CPO-ACO peut être compétitif avec les algorithmes dédiés de l'état de l'art.

Bibliographie

[ASG04] I. Alaya, C. Solnon, and K. Ghédira. Ant algorithm for multi-dimensional knapsack problem. In *International Conference on Bioinspired Optimization methods and teir Applications, (BIOMA2001)*, pages 63–72, 2004.

[ASG07] I. Alaya, C. Solnon, and K. Ghedira. Ant Colony Optimization for Multi-objective Optimization Problems. In *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 450–457. IEEE Computer Society, 2007.

[BAL94] S. BALUJA. *Population-Based Incremental Learning : A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning*. Rapport, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[BBM08] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*. Operations research/Computer Science Interfaces. Springer Verlag, 2008.

[BCS01] C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP 2001*, pages 565–569, 2001.

[BDHS04] V. Barichard, H. Deleau, JK. Hao, and F. Saubion. A hybrid evolutionary algorithm for constraint satisfaction problems. In *Lecture Notes in Computer Science*, volume 2936, pages 79–90. Springer-Verlag, 2004.

[BF07] N. Boysen and M. Fliedner. Comments on solving real car sequencing problems with ant colony optimization. *European Journal of Operational Research (*EJOR*)*, 182(1) :466–468, 2007.

[BFR95] C. Bessière, E.C. Freuder, and J.-C. Regin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598. Montréal Canada, 1995.

[BFR99] C. Bessière, E.C. Freuder, and J-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107 :125–148, 1999.

[BHS99a] B. Bullnheimer, F.R. Hartl, and C. Strauss. A new rank-based version of the ant system : A computational study. *Central European Journal for Operations Research and Economics*, 7(1) :25–38, (1999).

[BHS99b] B. Bullnheimer, R.F. Hartl, and C. Strauss. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89 :319–328, 1999.

[Bir04] M. Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, 2004.

[BNQ+07] S. Brand, N. Narodytska, C.-G. Quimper, Peter J. Stuckey, and T. Walsh. Encodings of the sequence constraint. In *CP2007*, volume 4741 of *LNCS*, pages 210–224. Springer, 2007.

[BP01] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4) :610–637, 2001.

[Bré79] D. Brélaz. New methods to color the vertices of a graph. *Communication of the ACM*, 22 :251–256, 1979.

[BR96] C. Bessière and J-C. Régin. MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. In *CP96, Second International Conference on Principles and Practice of Constraint Programming*, volume 61-75. Cambridge, MA, USA, 1996.

[BR01] C. Bessière and J-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01), San Francisco, CA, août 4-10*, pages 309–315. Morgan Kaufmann Publishers, 2001.

[BSPV02] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In *GECCO*, pages 11–18, 2002.

[Coe02] C.A. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms : a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11–12) :1245–1287, 2002.

[CS96] David W. Coit and Alice E. Smith. Penalty guided genetic search for reliability design optimization. *Computer and Industrial Engeneering*, 30(4) :895–904, 1996.

[CST96] David W. Coit, Alice E. Smith, and David M. Tate. Adaptive penalty methods for genetic optimization of constrained combinatorial problems. *INFORMS Journal on Computing*, 8(12) :173–182, 1996.

[DAGP90] J.L Deneubourg, S. Aron, S. Goss, and J.M. Pasteels. The self-organizing exploratory pattern of the argentine ant. volume 32, pages 159–168, 1990.

[DB01] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[DCG99] M. Dorigo, G. Di Caro, and L.M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2) :137–172, 1999.

[DD99] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw Hill, UK, 1999.

[Deb01] K. Deb. *An effecient Constraint Handling Method for Genetic Algorithms*. Computer Methods in Applied Mechanics and Engineering, 2001.

[DG97] M. Dorigo and L.M. Gambardella. Ant Colony System : A cooperative learning approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1) :53–66, 1997.

[DLR07] M. Van Dongen, C. Lecoutre, and O. Roussel. Results of the second CSP solver competition. In *Second International CSP Solver Competition*, pages 1–10, 2007.

[DMC91] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy, tech. *Rep. 91-016, Politecnico di Milano, Italy*, (1991).

[DMC96] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System : Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B : Cybernetics*, 26(1) :29–41, 1996.

[Dor92] M. Dorigo. *Optimization, learning and natural Algorithms (in Italian)*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, (1992).

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[DSH88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car sequencing problem in constraint logic programming. In *Y. Kodratoff, edotor, Proceedings of ECAI-88*, pages 290–295, 1988.

[DT99] A. Davenport and E.P.K. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. pages 345–357. In Proceedings of the First International Conference on the Practical Applications of Constraint Technologies and Logic Programming, 1999.

[EGN07] B. Estellon, F. Gardi, and K. Nouioua. Real-life car sequencing : very large neighborhood search vs very fast local search. *European Journal of Operational Research (EJOR)*, 191(3) :928–944, 2007.

[FGS06] S. Favuzza, G. Graditi, and E. Riva Sanseverino. Adaptive and dynamic ant colony search algorithm for optimal distribution systems reinforcement strategy. *Applied Intelligence*, 24(1) :31–42, 2006.

[FR89] T.A. FEO and M.G.C. RESENDE. A probabilistic heuristic for a computationally difficult set covering problem. In *Operations Research Letters*, pages 67–71, 1989.

[FR95] T.A. FEO and M.G.C. RESENDE. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6 :109–133, 1995.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[GADP89] S. Goss, S. Aron, J.L Deneubourg, and J.M. Pasteels. Self-organized shortcuts in the argentine ant. volume 76, pages 579–581, 1989.

[GC96] M. Gen and R. Cheng. A survey of penalty techniques genetic algorithms. In *Toshio Fukuda Takeshi Furuhashi, Proceedings of the 1996 International Conference on Evolutionary Computation*, pages 804–809. IEEE, 1996.

[GGP05] M. Gravel, C. Gagné, and W.L. Price. Review and comparison of three methodes for the solution of the car sequencing problem. *Journal of the Operational Research Society*, 56(4) :1287–1295, 2005.

[GGP06] C. Gagné, M. Gravel, and W.L. Price. Solving real car sequencing problems with ant colony optimization. *European Journal of Operational Research (EJOR)*, 174 :1427–1448, 2006.

[GH97] P. Galinier and J-K Hao. Tabu search for maximal constraint satisfaction problems. In *Lecture Notes in Computer Science, CP97, Third International Conference on Principles and Practice of Constraint Programming*, volume 1330, pages 196–208. Springer-Verlag, 1997.

[GJ79] MR. Garey and DS. Johnson. *Computer and intractability ; A guide to the theory of NP Completeness*. New York : WH Freeman, 1979.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[Glo86] F. Glover. Future paths for integer programming and links to artifitial intelligence. *Computers and Operations Research*, 13(5) :533–549, 1986.

[GMPW96] I. Gent, E. Macintyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of AAAI-96*. AAAI Press Menlo Park, California, 1996.

[GPS03] J. Gottlieb, M. Puchta, and C. Solnon. A study of greedy, local search and ant colony optimization approaches for car sequencing problems. *In Applications of evolutionary computing*, LNCS(2611) :246–257, 2003.

[GTD99] L.M. Gambardella, E. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50 :167–176, 1999.

[GW] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. In *Technical report, APES-09-1999, 1999. available from http ://cs-plib.cs.strath.ac.uk/. A shorter version appears in CP99*.

[HAB97] A. Ben Hadj-Alouane and James C. Bean. A genetic algorithm for the multiple-choice integer program. *Operational Research*, 45 :92–101, 1997.

[HE80] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[HGH99] J-K. Hao, P. Galinier, and M. Habib. Metaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Revue d'Intelligence Artificielle*, 13(2) :283–324, 1999.

[HM99] P. Hensen and N. Mladenović. *An introduction to Variable Neighborhood Search*. In S. Voβ, S. Martello, I. Osman, and C. Roucairol, editors, Meta-heuristics : advances and trends in local search paradigms for optimization, pages 433–438. Kluwer Academic Publisher, 1999.

[HM05] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, 2005.

[Hol92] J.H. Holland. *Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, 1992.

[HPRS06] Willem Jan V. Hoeve, G. Pesant, Louis M. Rousseau, and A. Sabharwal. Revisiting the sequence constraint. In *12th International Conference on Principles and Practice of Constraint Programming - CP 2006*, volume volume 4204 of Lecture Notes in Computer Science of *LNCS*, pages 620–634. Springer, 2006.

[HS] H.H. Hoos and T. Stützle. Towards a characterisation of the behaviour of stochastic local search algorithms for sat. *Artificial Intelligence*, 112.

[HS96] F. Hoffmeister and J. Sprave. Problem-independent handling of constraints by use of metric penalty functions. In *Lawrence J. Fogel, Peter J. Angeline and Thomas Bäck, editors, Proceedings of the Fifth Annual Conference on Evolutionary Programming (EP'96)*, pages 289–294. MIT Press, 1996.

[ILO98] ILOG. *Ilog Solver user's manual, Technical report* ILOG. 1998.

[JH94] J. Joines and C. Houk. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with gas. In *David Fogel, editors, Proceedings of the first IEEE Conference on Evolutionary Computation*, pages 579–584. IEEE Press, 1994.

[JL02] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1) :21–45, 2002.

[KAS07] M. Khichane, P. Albert, and C. Solnon. Using ACO to guide a CP search. In *First Workshop on Autonomous Search, in conjunction with CP'07, Providence, Etats-Unis*, 2007.

[KAS08a] M. Khichane, P. Albert, and C. Solnon. CP with ACO. In *5th International Conference on Integration of AI and OR Techniques in CP for Combinatorial Optimization Problem (CPAIOR2008)*, volume 5015 of *LNCS*, pages 328–332. short paper, Springer, 2008.

[KAS08b] M. Khichane, P. Albert, and C. Solnon. Integration of ACO in a constraint programming language. In *6th International Conference on*

Ant Colony Optimization and Swarm Intelligence (ANTS'08), volume 5217 of *LNCS*, pages 84–95. Springer, 2008.

[KAS08c] M. Khichane, P. Albert, and C. Solnon. Programmation par contraintes avec des fourmis. In *Journées Francophones de Programmation par Contraintes (JFPC), Nantes*, pages 337–348, 2008.

[KAS09a] M. Khichane, P. Albert, and C. Solnon. An ACO-based reactive framework for Ant Colony Optimization : First experiments on constraint satisfaction problems. In *LION3 Learning and Intelligent OptimisatioN*, volume 5851 of *LNCS*, pages 119–133. Springer, 2009.

[KAS09b] M. Khichane, P. Albert, and C. Solnon. Un modèle réactif pour l'optimisation par colonies de fourmis : application à la satisfaction de contraintes. In *Journées Francophones de Programmation par Contraintes (JFPC), Orléans*, 2009.

[KAS10a] M. Khichane, P. Albert, and C. Solnon. Intégration de l'optimisation par colonies de fourmis dans cp optimizer. In *Journées Francophones de Programmation par Contraintes (JFPC), Caen*, 2010.

[KAS10b] M. Khichane, P. Albert, and C. Solnon. Strong Combination of Ant Colony Optimization with Contraint Programming Optimization. In *CPAIOR2010 7^{th} International Conference on Integration of AI and OR Techniques in CP for Combinatorial Optimisation Problems*, LNCS. Springer, 2010.

[KD95] K. Kask and R. Dechter. GSAT and local consistency. In *Proc. of IJCAI*, 1995.

[KGV83] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *science*, 220(4598) :671–680, 1983.

[Kis04] T. Kis. On the complexity of the car sequencing problem. *Operations Research Letters*, 32 :331–335, 2004.

[KP98] S. Kazarlis and V. Petridis. Varying fitness functions in genetic algorithms : Studying the rate of increase of the dynamic penalty terms. In *A. E. Eiben, T. Bäck, M Schoenauer, H.-P. Schwefel, editors, Parallel Problem Solving from Natural V — PPSN V, Amsterdam, The Netherlands*. Springer-Verlag, 1998.

[LL01] P. LARRANAGA and J. LOZANO. *Estimation of Distribution Algorithms. A new tool for Evolutionary Computation*. Kluwer Academic Publishers, 2001.

[LLW98] J.H.M. Lee, H.F. Leung, and H.W. Won. Performance of a comprehensive and efficient constraint library using local search. *In 11th Australian JCAI, LNAI*, 1998.

[LM99] G. Leguizamon and Z. Michalewicz. A new version of Ant System for Subset Problem. In *Congress on Evolutionary* Computation, pages 1459–1464, 1999.

[LMS01] H.R. Lourenço, O. Martin, and T. Stützle. *Iterated local search*. In F. Golver and G. Kochenberger, editors, Handbook of Metaheuristics, volume 57 of International Series in Operations Research Management Science. Kluwer Academic Publisher, 2001. pages 321–353.

[LP91] Gunar E. Liepins and W.D Potter. A genetic algorithm approach to multiple-fault diagnosis. In *Lawrence Davis, editor, Handbook of Genetic Algorithms, chapter 17*, pages 237–250, 1991.

[LV90] G.E. Liepins and Michael D. Vose. Representational issues in genetic optimization. *Journal of Experimental and Theortical Computer Science*, 2(2) :4–30, 1990.

[MA94] Z. Michalewicz and Naguib F. Attia. Evolutionary optimization of constrained problems. In *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, pages 98–108. World Scientific, 1994.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 :99–118, 1977.

[McG79] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19 :229–250, 1979.

[ME04] B. Meyer and A. Ernst. Integrating aco and constraint propagation. In *4th International Conference on Ant Colony Optimization and Swarm Intelligence (ANTS'04)*, volume 3172 of *LNCS*, pages 166–177. Springer, 2004.

[Mey08] B. Meyer. *Hybrids of constructive meta-heuristics and constraint programming : A case study with ACO*. In Chr. Blum, M.J. Blesa, A. Roli, and M. Sampels, editors, Hybrid Metaheuristics-An emergent approach for optimization. Springer Verlag, New York, 2008.

[MF85] A. K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25 :65–74, 1985.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[MH02] L. Michel and P. Van Hentenryck. A constraintbased architecture for local search. In *OOPSLA 02 : Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA*, pages 83–100. ACM Press, 2002.

[Mic96] Z. Michalewicz. Genetic algorithms + data structures = evolution programs. *Springer-Verlag, third edition*, 1996.

[MJPL92] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satistaction and scheduling problems. *Artificial Intelligence*, 58 :161–205, 1992.

[MMM01] M.W. Moskewicz, C.F. Madigan, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.

[MQ98] Angel K. Morales and Carlos V. Quezada. A universal electic genetic algorithm for constrained optimization. In *Proceedings* 6^{th} *European Congress on Intelligent Techniques Soft Computing, EUFIT'*98, pages 518–522. Verlag Mainz, 1998.

[MRR+53] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys*, 21 :1087–1092, 1953.

[MS98] K. Marriott and Peter J. Stuckey. *Programming with Constraints : An Introduction*. MIT Press, 1998.

[Müh92] H. Mühlenbein. Parallel genetic algorithms in combinatorial optimization. In *O. Balci, R. Sharda, and S. Zenios, editors, Computer Science and Operations Research*, pages 441–456. Pergamon Press, 1992.

[NS97] Bryan A. Norman and Alice E. Smith. Random keys genetic algorithm with adaptive penalty function for optimization of constrained facility layout problem. In *Thomas Bäck, Zbigniew Michalewicz and Xin Yao, editors, Proceedings of the 1997 International Conference on Evolutionary Computation*, pages 407–411. IEEE, 1997.

[NTG04] B. Neveu, G. Trombettoni, and F. Glover. Id walk : A candidate list strategy with a simple diversification device. In *Proceedings of CP2004*, volume 3258 of *LNCS*, pages 423–437. Springer-Verlag, 2004.

[NW88] GL. Nemhauser and AL. Wolsey. *Integer and combinatorial optimization*. New York :Jhon Wiley and & Sons, 1988.

[OD94] D. Orvosh and L. Davis. Using a genetic algorithm to optimize problems with feasibility constraints. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 548–553. IEEE Press, 1994.

[Par94] J. Paredis. Co-evolutionary constraint satisfaction. In *Proceedings of the* 3^{rd} *Conference on Parallel Problem Solving from Nature*, pages 46–55. Springer Verlag, 1994.

[PFM06] P. Pellegrini, D. Favaretto, and E. Moretti. On Max-Min ant system's parameters. In *ANTS'2006 -, Fifth International Workshop on Ant Colony Optimization and Swarm Intelligence*, volume 4150 of *Lecture Notes in Computer Science*, pages 203–214. Springer Berlin / Heidelberg, 2006.

[PGCP99] M. PELIKAN, D. GOLDBERG, and E. CANTÚ-PAZ. BOA : The bayesian optimization algorithm. pages 525–532. Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99, vol. I, Morgan Kaufmann Publishers, San Fransisco, CA, 1999.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problems. In *Computational Intelligence*, volume 9, 1993.

[PS82] CH. Papadimitriou and K. Steiglitz. *Combinatorial optimization–Algorithms and complexity*. New York : Dover, 1982.

[PS93] D. Powell and Michael M. Skolnick. Using genetic algorithms in engineering design optimization with nonlinear constraints. In *Stephanie Forrest, editors, Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 424–431. Morgan Kaufmann Publishers, 1993.

[PS04] L. Perron and P. Shaw. Combining forces to solve the car sequencing problem. In *Proceedings of CPAI-OR2004*, volume 3011, pages 225–239. Springer, 2004.

[PV04] C. Pralet and G. Verfaillie. Travelling in the world of local searches in the space of partial assignments. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, (3011) :240–255, 2004.

[Ran04] M. Randall. Near parameter free Ant Colony Optimisation. In *ANTS Workshop*, volume 3172 of *Lecture Notes in Computer Science*, pages 374–381. Springer, 2004.

[Ref04] P. Refalo. Impact-based search strategies for constraint programming. *In CP04, 10th International Conference on Principals and Practice of Constraint Programming,*, (10) :557–571, 2004.

[Reg04] J.-C. Regin. Combination of amoung and cardinality. pages 255–239. In proceeding of CPAIOR'2004, Springer, 2004.

[Rég94a] J.-C. Régin. A filtering algorithm for constraints of difference in csps. *European Journal of Operational Research (EJOR)*, 1994.

[Rég94b] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94, Seattle, WA, USA*, pages 362–367, 1994.

[RP97] J.-C. Regin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *CP97*, volume 1330, pages 32–46. Springer-Verlag, 1997.

[RPLH89] Jon T. Richardson, Mark R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In *J. David Schaffer, editor, Proceedings of the third International Conference on Genetic Algorithms*, pages 191–197. Morgan Kaufmann Publishers, 1989.

[SB] C. Solnon and D. Bridge. An Ant Colony Optimization Meta-Heuristic for Subset Selection Problems, publisher = Nova Science, pages = 7-29, year = 2006. In *System Engineering using Particle Swarm Optimization*.

[SB06] C. Solnon and D. Bridge. *An Ant Colony Optimization Meta-Heuristic for Subset Selection Problems - Chapter of the book « System Engineering using Particle Swarm Optimization », Nadia Nedjah and Luiza Mourelle editors, Pages 7-29*. Nova Science publisher, 2006.

[Sch97] A. Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proc. of IJCAI*, 1997.

[SCNA08] C. Solnon, V.D. Cung, A. Nguyen, and C. Artigues. The car sequencing problem : overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *European Journal of Operational Research (EJOR)*, 191(3) :912–927, 2008.

[SF06] C. Solnon and S. Fenet. A study of ACO capabilities for solving the maximum clique problem. *Journal of Heuristics*, 12(3) :155–180, 2006.

[SH00] T. Stützle and H.H. Hoos. $\mathcal{MAX} - \mathcal{MIN}$ Ant System. *Journal of Future Generation Computer Systems*, 16 :889–914, 2000.

[Smi96] B. Smith. Succeed-first or fail-first : A case study in variable and value ordering heuristics. pages 321–330. Third Conference on the Practical Applications of Constraint Technology, 1996.

[Sol00] C. Solnon. Solving Permutation Constraint Satisfaction Problems with Artificial Ants. In 14^{th} *European Conference on Artificial Intelligence (ECAI)*, pages 118–122. IOS Press, 2000.

[Sol02] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.

[Sol08] C. Solnon. Combining two pheromone structures for solving the car sequencing problem with Ant Colony Optimization. *European Journal of Operational Research (EJOR)*, 191 :1043–1055, 2008.

[ST93] Alice E. Smith and David M. Tate. Genetic optimization using a penalty function. In *Stephanie Forrest, editors, Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 499–503. Morgan Kaufmann Publishers, 1993.

[SX93] M. Schoenauer and S. Xanthakis. Constrained GA optimization. In *Stephanie Forrest, editors, Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 573–580. Morgan Kaufmann Publishers, 1993.

[TS95] D.M. Tate and Alice E. Smith. A genetic approach to the quadratic assignment problem. *Computers and Operational Research*, 22(1) :73–78, 1995.

[Tsa93] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[YE04] Sherin M. Youssef and Dave G. Elliman. Reactive Prohibition-based Ant Colony Optimization (RPACO) : A new parallel architecture for constrained clique sub-graphs. In *ICTAI '04 : Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 63–71, Washington, DC, USA, 2004. IEEE Computer Society.

[YGIT95] T. Yokota, M. Gen, K. Ida, and T. Taguchi. Optimal design of system reliability by an improved genetic algorithm. *Transactions of Institute of Electronics, Information and Computer Engineering*, J78-A(6) :702–709, 1995.