HABILITATION
À DIRIGER DES RECHERCHES

présentée à

L'UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS
UFR SCIENCES

ECOLE DOCTORALE STIC

Spécialité : INFORMATIQUE

par

Bertrand NEVEU

Techniques de résolution de problèmes de satisfaction de contraintes

Soutenue publiquement le 1er février 2005

Après avis des rapporteurs :

M.	**Christian**	**BESSIÈRE**	Chargé de recherches CNRS
M.	**Boi**	**FALTINGS**	Professeur
M.	**Gérard**	**VERFAILLIE**	Ingénieur de recherches CERT

Devant la commission d'examen composée de :

M.	**Christian**	**BESSIÈRE**	Chargé de recherches CNRS
M.	**Patrice**	**BOIZUMAULT**	Professeur
M.	**Pedro**	**MESEGUER**	Chercheur IIIA-CSIC
M.	**Dominique**	**MICHELUCCI**	Professeur
M.	**Michel**	**RUEHER**	Professeur
M.	**Gérard**	**VERFAILLIE**	Ingénieur CERT

Résumé

Ce mémoire présente un résumé des recherches que j'ai effectuées ou encadrées dans le domaine de la résolution de problèmes de satisfaction de contraintes depuis 1992.

Une première partie présente les principales contributions effectuées dans les problèmes à variables à domaines finis d'une part et à domaines continus d'autre part. Chacune des contributions est replacée dans son contexte particulier en donnant les références de base et quelques travaux proches, sans vouloir être exhaustif.

Le chapitre 2 présente les méthodes et algorithmes généraux pour résoudre des problèmes de satisfaction de contraintes (CSP) en domaines finis, en partant des méthodes complètes et en allant jusqu'aux méthodes incomplètes en passant par les méthodes arborescentes tronquées.

Le chapitre 3 traite des méthodes de résolution des problèmes de contraintes sur domaines continus, en deux sous-domaines : la partie 3.1 présente la résolution de contraintes fonctionnelles et la partie 3.2 présente la résolution de contraintes géométriques sous forme de CSP numériques, avec en particulier les méthodes de décomposition, une étude de la rigidité et de l'assemblage de sous-parties rigides.

Une conclusion présente les voies de recherche que je souhaite poursuivre ces prochaines années (chapitre 4).

Dans une deuxième partie, les contributions les plus récentes sont présentées en détail dans les chapitres qui reprennent des publications. Il s'agit de :

- la recherche à focalisation progressive `PFS` (chapitre 5),
- la bibliothèque de recherche locale INCOP (chapitre 6),
- l'algorithme hybride `GWW-idw` (chapitre 7),
- la détection de parties sur-rigides dans des CSP géométriques (chapitre 8),
- l'algorithme de retour-arrière inter-blocs `IBB` (chapitre 9).

Table des matières

Chapitre 1

Vue d'ensemble des contributions

Ce chapitre présente les principales contributions réalisées dans le domaine de la résolution de problèmes de satisfaction de contraintes auxquelles j'ai participé directement ou que j'ai encadrées.

Une première partie (cf 1.1) rappelle comment, venant des systèmes experts, je suis venu à m'intéresser aux contraintes. Puis un plan présente les différents chapitres présentant les diverses contributions.

1.1 Des systèmes experts aux contraintes

Mon intérêt pour le domaine de recherche des contraintes provient de l'expérience que j'ai eue avec les systèmes experts, mon précédent domaine d'intérêt entre 1984 et 1992. Durant ces années, j'ai tout d'abord participé, dans le projet dirigé par Pierre Haren, entre 1984 et 1987 au développement du logiciel Smeci [Neveu et Haren, 1986; Neveu *et al.*, 1990] un générateur de systèmes experts comme on en concevait alors. Un tel logiciel, programmé par objets [Neveu, 1990] comprenait divers formalismes :

- de la représentation par objets avec une description statique des classes, des attributs et des méthodes,
- des règles de production,
- des tâches indiquant quelles règles utiliser pour les effectuer,
- **des contraintes**.

J'ai également participé au premier prototype de ce logiciel, un système expert de conception de digues portuaires [Neveu, 1987].

Le thème central du projet Secoia que j'ai dirigé, une fois le logiciel Smeci terminé, était alors la représentation des connaissances par objets. Plusieurs thèses ont été menées dans ce thème :

- Claude Fornarino, ObjectiveAda : une extension objet du langage Ada. Application à un environnement de conception de systèmes experts [Fornarino, 1991],
- Mireille Blay-Fornarino et Anne-Marie Pinna-Déry, Un modèle objet logique et relationnel : Le langage Othelo [Blay-Fornarino et Pinna-Déry, 1990]

- Franck Lebastard, Correspondance entre SGBD relationnels et SGBD objets [Lebastard, 1993b]

D'autres thèses ont été menées dans le domaine des systèmes à base de connaissances :

- Coco Djossou, Approche multi-processus pour la coopération entre systèmes à base de connaissances [Djossou, 1993]
- Wided Lejouad, Etude et application des techniques de distribution pour un générateur de systèmes à base de connaissances [Lejouad, 1994]
- Stéphane LeMénec, Théorie des jeux dynamiques et techniques de programmation avancée appliquées au duel aérien à moyenne distance [Le Ménec, 1994]

Les contraintes dans Smeci

Dans Smeci, on pouvait poser des contraintes entre les objets : un prédicat permettait de les vérifier et des méthodes permettaient de satisfaire les contraintes fonctionnelles en appliquant un algorithme de propagation locale. Il y avait aussi des domaines de valeurs pour certains attributs d'objets et des restrictions de ces domaines avaient lieu quand, lors d'un raisonnement, on précisait un objet en le sous-classant. On peut voir cela comme une propagation de contraintes élémentaire. Les points de choix étaient faits par le moteur de Smeci qui gérait une recherche arborescente avec retour-arrière. Les contraintes pouvaient provoquer un retour-arrière en cas d'échec de leur vérification. Dans certains cas particuliers (contraintes fonctionnelles), on pouvait établir leur cohérence par propagation locale, mais il n'y avait pas d'algorithme dédié à la satisfaction d'un problème de contraintes en domaines finis.

On voit donc qu'il existait beaucoup d'ingrédients pour traiter des contraintes, mais qu'ils étaient dispersés dans le logiciel et que les problèmes faisant intervenir des contraintes n'étaient pas formalisés.

Les contraintes dans le projet SECOIA

Pierre Berlandier a alors étudié dans sa thèse comment intégrer des contraintes dans un systèmes à base de connaissance [Berlandier, 1992b]. A partir de cette première expérience, nous avons pris conscience dans le projet Secoia de l'importance du domaine des contraintes en tant que tel et que de nombreux problèmes (emplois du temps, ordonnancement, affectation de ressources, aménagement spatial ...) pouvaient être modélisés dans le cadre des problèmes de satisfaction de contraintes.

A cette époque là, un premier logiciel de satisfaction de contraintes PROSE [Berlandier, 1992a] a été développé dans l'équipe. Nous avons intégré le projet Contraintes flexibles [Bel *et al.*, 1992; Bellicha *et al.*, 1995] du PRC-IA, et les contraintes sont devenues un des principaux axes de recherche de Secoia (1992-1995), puis le thème unique de l'équipe "Contraintes" du CERMICS (1995-2001). En plus des traditionnels problèmes de satisfaction de contraintes en domaines finis, nous nous sommes intéressés aux contraintes fonctionnelles et aux contraintes géométriques. Avec la création du projet COPRIN à l'INRIA Sophia Antipolis, nous avons étendu nos sujets de recherche aux contraintes portant sur des variables à domaines continus.

Thèses soutenues

Les thèses suivantes que j'ai encadrées ou co-encadrées ont été soutenues dans l'équipe dans le domaine des contraintes :

- Pierre Berlandier, Etude de mécanismes d'interprétation de contraintes et de leur intégration dans un langage à base de connaissances [Berlandier, 1992b]
- Philippe Ballesta, Contraintes et objets : clefs de voûte d'un outil d'aide à la composition musicale ? [Ballesta, 1994]
- Nicolas Chleq, Contribution à l'étude du raisonnement temporel : usage de la résolution avec contraintes et application à l'abduction en raisonnement temporel [Chleq, 1995a]
- Philippe Charman, Gestion des contraintes géométriques pour l'aide à l'aménagement spatial [Charman, 1995]
- Mouhssine Bouzoubaa, Hiérarchies de contraintes : quelques approches de résolution [Bouzoubaa, 1996]
- Gilles Trombettoni, Algorithmes de maintien de solution par propagation locale pour les systèmes de contraintes [Trombettoni, 1997]
- Maria-Cristina Riff Rojas, Résolution d'un problème de satisfaction de contraintes avec des algorithmes évolutionnistes [Riff, 1997b]
- Nicolas Prcovic, Recherche arborescente parallèle et séquentielle pour les problèmes de satisfaction de contraintes [Prcovic, 1998]
- Blaise Madeline, Algorithmes évolutionnaires et résolution de problèmes de satisfaction de contraintes en domaines finis, [Madeline, 2002]
- Christophe Jermann, Résolution de contraintes géométriques par rigidification récursive et propagation d'intervalles, [Jermann, 2002]

1.2 Plan du mémoire

Le thème principal de mes recherches se situe dans le domaine des algorithmes de résolution, en laissant de côté l'aspect langage de programmation par contraintes, largement étudié dans la communauté PLC (Programmation logique avec contraintes [Jaffar et Maher, 1994]).

Ce domaine des algorithmes de résolution étant lui-même très vaste, chacune des contributions est replacée dans son contexte particulier en donnant les références de base et quelques travaux proches, sans vouloir être exhaustif.

Le chapitre 2 présente les méthodes et algorithmes généraux pour résoudre des problèmes de satisfaction de contraintes (CSP) en domaines finis, en partant des méthodes complètes et en allant jusqu'aux méthodes incomplètes en passant par les méthodes arborescentes tronquées. Le chapitre 3 traite des méthodes de résolution des problèmes de contraintes sur domaines continus, en deux sous-domaines : la partie 3.1 présente la résolution de contraintes fonctionnelles et la partie 3.2 présente la résolution de contraintes géométriques sous forme de CSP numériques, avec en particulier les méthodes de décomposition, une étude de la rigidité et de l'assemblage de sous-parties rigides. Une conclusion présente les voies de recherche que je souhaite poursuivre ces prochaines années (chapitre 4).

Les contributions les plus récentes sont présentées en détail dans les chapitres qui reprennent des publications. Il s'agit de :
- la recherche à focalisation progressive `PFS` (chapitre 5),
- la bibliothèque de recherche locale INCOP (chapitre 6),
- l'algorithme hybride `GWW-idw` (chapitre 7),
- la détection de parties sur-rigides dans des CSP géométriques (chapitre 8),

- l'algorithme de retour-arrière inter-blocs `IBB` (chapitre 9).

Chapitre 2

Résolution de problèmes de satisfaction de contraintes en domaines finis

Ce chapitre présente les recherches effectuées pour la satisfaction de contraintes en domaines finis. Après un rappel sur le cadre des CSP, nous présenterons les recherches effectuées dans le domaine des méthodes complètes arborescentes, puis des méthodes incomplètes stochastiques de recherche locale, à population et hybrides.

2.1 Le cadre des CSP

Rappelons le cadre des problèmes de satisfaction de contraintes (CSP) en domaines finis défini dans [Montanari, 1974].

Définition d'un CSP en domaines finis
Un CSP est constitué d'un couple $< V, C >$ où V est un ensemble donné de n variables, chaque variable v_i prenant ses valeurs dans un domaine fini D_i, et où C est un ensemble de m contraintes portant chacune sur un sous ensemble de k variables $v_{c_1}, ..., v_{c_k}$ et indiquant quels sont les k-uplets de valeurs satisfaisant la contrainte. Une solution du CSP est alors une affectation de toutes les variables satisfaisant toutes les contraintes.

Les contraintes peuvent être binaires ou d'arité quelconque : on parle alors de contraintes n-aires. Dans le cas de contraintes binaires, on définit le graphe de contraintes comme le graphe ayant pour nœuds les variables et pour arêtes les contraintes liant deux variables. Dans le cas de contraintes n-aires, on peut définir un hypergraphe ou un graphe biparti variables-contraintes.

Le cadre des CSP en domaines finis permet de modéliser de nombreux problèmes combinatoires tant académiques qu'industriels comme le coloriage de graphes, l'ordonnancement de tâches, l'affectation de ressources, les emplois du temps, l'affectation de fréquences radio ...

Méthodes de résolution complètes et incomplètes
Dans le cas général, l'existence d'une solution d'un CSP donné est un problème NP-complet. Bien qu'il n'existe pas, sauf si l'on démontre P=NP, d'algorithme polynomial pour résoudre un CSP dans le pire des cas, de nombreuses méthodes ont cependant été mises au point ces 30 dernières années pour essayer de résoudre efficacement des problèmes de taille industrielle.

On distingue deux grands types de méthodes : les méthodes complètes d'une part, qui explorent totalement l'espace de recherche et les méthodes incomplètes qui ne l'explorent qu'en partie et ne peuvent prouver l'inexistence de solution d'un problème sur-contraint.

Les méthodes complètes effectuent un parcours systématique de l'espace de recherche. La manière la plus simple est d'effectuer une recherche arborescente en profondeur d'abord (cf 2.2). Quand l'espace de recherche devient trop grand pour être exploré entièrement, une telle recherche arborescente a le principal défaut de rester longtemps dans les sous-arbres déterminés par les premiers choix qui ne sont pas toujours les bons.

Quand le problème a des solutions, on peut effectuer une recherche arborescente tronquée (méthodes de type `LDS`) en ne parcourant que les parties les plus prometteuses en étant guidé par une heuristique. Enfin, en l'absence de connaissance heuristique pour guider la recherche, pour éviter de stagner très longtemps dans un sous-arbre sans solution, on peut parcourir plusieurs sous-arbres de manière alternée (cf paragraphe 2.2.4 Recherche alternée) ou en parallèle (cf paragraphe 2.2.5 Recherches arborescentes parallèles).

Les méthodes incomplètes stochastiques (cf 2.3) effectuent un parcours non systématique de l'espace de recherche : récemment, des méthodes de recherche arborescente incomplètes comprenant une part d'aléatoire dans les points de retour-arrière [Prestwich, 2000] ou dans les choix des heuristiques et avec des stratégies de redémarrage ont été proposées [Gomes *et al.*, 1998]. Mais les méthodes incomplètes les plus employées sont celles effectuant de la recherche locale à partir d'une configuration (cf 2.3.1) ou celles manipulant un ensemble de configurations comme les algorithmes génétiques ou évolutionnistes (cf 2.3.2.1) et l'algorithme "Go with the winners" `GWW` (cf 2.3.2.2).

2.2 Méthodes arborescentes

2.2.1 Schéma général des méthodes arborescentes complètes

Ces méthodes construisent un arbre de recherche : un nœud de l'arbre correspond à une instanciation[1] partielle cohérente : certaines variables ont leur valeur affectée, d'autres leurs domaines réduits et un niveau de cohérence est garanti : au minimum, toutes les contraintes portant sur les variables affectées sont satisfaites. Le long d'une branche, on suit donc un raisonnement monotone, les domaines des variables se réduisent jusqu'à ce que toutes les variables soient affectées ou que l'on ait détecté un échec : une feuille est donc soit une solution quand on a réussi à construire une branche qui a instancié toutes les variables en satisfaisant les contraintes, soit un échec.

Une des principales caractéristiques des méthodes de recherche arborescente pour la résolution de CSP est leur aspect constructif : elles manipulent des instanciations partielles cohérentes qui le long d'une branche de l'arbre de recherche sont de plus en plus précisées.

[1]on parle d'instancier une variable quand on lui affecte une valeur de son domaine

Une deuxième caractéristique est leur complétude : chaque point de choix est tel qu'il réalise une partition de l'espace de recherche correspondant au nœud courant. De part leur caractère systématique, elles assurent qu'en terminant, elles auront trouvé toutes les solutions ou prouvé que le problème n'en a aucune.

Ces méthodes de recherche arborescente sont au coeur des systèmes de programmation par contraintes, décrits ci-dessous.

On distingue généralement dans ces méthodes l'algorithme lui-même qui indique comment construire cet arbre de recherche et quelle réduction des domaines (*filtrage*) effectuer à chaque nœud, des heuristiques qui indiquent comment sont effectués les points de choix.

Programmation par contraintes

La programmation par contraintes a démarré, pour les domaines finis, avec les recherches menées à l'ECRC à Munich dans les années 1980 [Van Hentenryck, 1989]. Il s'agit d'un mode de programmation déclaratif, qui sépare la description du problème (variables, contraintes) de sa résolution (algorithme de recherche arborescente). Il en est sorti deux grands types de logiciels : ceux, construits au dessus d'un moteur Prolog avec un retour-arrière intégré dans ce langage, et ceux écrits dans un langage de programmation plus classique, dans lesquels on a dû réécrire une gestion de l'arbre de recherche.

Parmi les systèmes en Prolog gérant les contraintes en domaines finis, citons Prolog IV, Sicstus Prolog, CLP(FD), Eclipse ...

Dans la deuxième catégorie, de premiers résolveurs de contraintes commerciaux en domaines finis sont apparus comme CHARME de Bull, CHIP de Cosytec, PECOS d'Ilog (ancêtre de IlogSolver). C'est dans cette catégorie qu'on peut situer la bibliothèque de contraintes en Lelisp nommée Prose [Berlandier, 1992a] écrite dans le projet par P. Berlandier. Ce dernier a d'autre part proposé dans sa thèse une intégration des contraintes dans une représentation de connaissances à objets [Berlandier, 1992b].

Algorithmes

L'algorithme de base est celui du Backtrack chronologique [Golumb et Baumert, 1965] : c'est une recherche en profondeur d'abord avec retour-arrière chronologique et test des contraintes quand leurs variables sont instanciées. De très nombreuses recherches ont été menées pour améliorer cet algorithme.

Le premier axe de recherche consiste à propager les contraintes le plus tôt possible pour rapidement détecter des incohérences (voir paragraphe 2.2.2 Méthodes de filtrage). C'est un des points clés des succès de la programmation par contraintes.

Un deuxième axe concerne le retour-arrière lui-même. Le mécanisme de base est un retour-arrière chronologique : en cas d'échec, on revient sur le dernier choix effectué, sans se soucier de savoir si ce choix a une quelconque responsabilité dans l'échec courant. Divers mécanismes permettant un retour arrière plus performant ont été conçus : GBJ (Graph Based Backjumping) [Dechter, 1990] utilise le graphe de contraintes, CBJ (Conflict Based Backjumping) [Prosser, 1993] utilise les conflits détectés pour déterminer un point de retour, DBT (Dynamic Backtracking) gère une base de "no-goods" qui lui permet à la fois de trouver un point de retour et de garder si possible les valeurs intermédiaires [Ginsberg, 1993]. Certains algorithmes vont même plus loin dans la mémorisation des no-goods découverts pendant la recherche et effectue de l'apprentissage : citons les algorithmes de Learning [Dechter, 1990] et le Nogood Recording [Schiex et Verfaillie, 1994]. On peut trouver une description de ces algorithmes dans le mémoire d'HDR de Thomas Schiex [Schiex, 2000] et dans le livre récent de Rina Dechter [Dechter, 2003]. On verra dans la partie 3.2.2 comment nous avons adapté ces méthodes pour la résolution de CSP en domaines continus.

Un troisième axe concerne la stratégie de construction de l'arbre de recherche. Toutes les stratégies étudiées sont basées sur une stratégie de recherche principalement en profondeur d'abord, ce qui permet de limiter la mémoire nécessaire au stockage de la branche courante et des points de choix en suspens dans cette branche. Des variantes ont été établies (`LDS`, `DDS`, `IDFS`,...) pour pallier le principal défaut de la recherche en profondeur, à savoir la trop grande dépendance envers les premiers choix effectués (voir paragraphe 2.2.3 Recherches arborescentes tronquées et paragraphe 2.2.4 sur les recherches alternées).

On peut aussi construire des arbres de recherche binaires comme MAC [Sabin et Freuder, 1994] avec des points de choix ($X = i$ ou $X \neq i$), des arbres N-aires (X=1,..., X=n) avec tous les choix possibles pour chaque variable ou des arbres où les points de choix correspondent à des bissections de domaines ($X \leq i$ ou $X \geq i+1$), quand les domaines sont des ensembles ou des intervalles d'entiers.

Toutes ces méthodes améliorant l'algorithme du backtrack sont complémentaires et ont été hybridées entre elles : citons par exemple les algorithmes FC-CBJ [Prosser, 1993], MAC-CBJ [Prosser, 1995], MAC-DBT [Jussien *et al.*, 2000], qui combinent filtrage et retour arrière informé. Cependant, il a été remarqué que le choix d'une bonne heuristique, allié à une filtrage puissant, peut rendre presqu'inutile un retour en arrière sophistiqué [Bessière et Régin, 1996].

Heuristiques

Pour construire un arbre de recherche assez petit, les points de choix effectués lors de la recherche arborescente doivent généralement être guidés par des heuristiques de choix de variable et de valeur.

Les heuristiques de choix de la prochaine variable à instancier sont très importantes pour la forme et donc la taille de l'arbre de recherche construit, les heuristiques de choix de valeur régissent l'ordre dans lequel les branches sont visitées et sont particulièrement intéressantes quand on recherche une seule solution ou une solution optimisant un critère.

Il est bien sûr toujours intéressant d'utiliser la connaissance liée au problème pour faire en premier les choix les plus importants et se diriger sur les branches les plus pertinentes.

En l'absence de connaissance spécifique, il est alors utile d'utiliser les heuristiques générales. Pour réduire la taille de l'arbre de recherche, il faut à la fois limiter la largeur de l'arbre construit et la profondeur des branches menant à des échecs. Les heuristiques de choix de variable générales comme le choix de la variable ayant le plus petit domaine (*dom*) satisfont le premier critère et le choix de la variable la plus contrainte le second (on peut alors choisir de minimiser le degré de la variable dans le graphe de contraintes (*deg*). Il est apparu que souvent une combinaison de ces 2 heuristiques comme (*dom+deg*, qui utilise le degré comme second critère servant à départager les variables ayant même taille de domaines et *dom/deg*) qui choisit la variable ayant le plus petit rapport *dom/deg* permettait d'obtenir les meilleurs résultats.

Les heuristiques de choix de valeur servent surtout à guider la recherche vers les zones les plus prometteuses. Parmi les heuristiques générales, signalons par exemple *max-promises* [Geelen, 1992] et *LVO* [Frost et Dechter, 1995] qui sont basées respectivement sur le produit et la somme des domaines des futures variables résultant de la propagation des contraintes due au choix effectué pour la variable courante. L'idée est que les valeurs les moins contraignantes ont plus de chance de faire partie d'une solution.

2.2.2 Méthodes de filtrage

Le principal défi des méthodes arborescentes est de limiter la taille de l'arbre de recherche exploré en filtrant les domaines au fur et à mesure, c-à-d en éliminant les valeurs qui ne peuvent être ajoutées à l'instanciation partielle courante pour mener à une solution. C'est le principe des techniques qui entrelacent points de choix et filtrage des domaines par propagation des contraintes.

Un certain nombre de niveaux de cohérence ont été définis dans la littérature : citons par exemple parmi les plus utilisés dans l'ordre de cohérence croissant : la cohérence de borne, la cohérence d'arc, de chemin ... Etablir ces niveaux de cohérence consiste à réduire les domaines des variables et à rendre certaines explicites. Des algorithmes polynomiaux sont alors appliqués à chaque nœud de l'espace de recherche pour obtenir la cohérence souhaitée.

Des méthodes générales, pouvant traiter des contraintes quelconques données en extension ont été définies selon le degré de cohérence établi complètement ou partiellement à chaque nœud de l'arbre de recherche. Pour les contraintes binaires, divers algorithmes effectuent ce filtrage suivant le degré de propagation réalisé : Forward Checking (FC) et Real Full Lookahead (RFL) [Haralick et Elliott, 1980], Maintaining Arc Consistency (MAC) [Sabin et Freuder, 1994], Quick [Debruyne, 1998]. Différentes variantes ont été établies pour leur adaptation aux contraintes n-aires : Pour FC, elles sont proposées dans [Bessière *et al.*, 1999], pour l'obtention de la cohérence d'arc généralisée aux contraintes n-aires `GAC` un algorithme est proposé dans [Bessière et Régin, 1997].

Mais les contraintes dans un problème réel ont souvent une sémantique particulière (égalité, inégalité, différence ...), qu'il est utile d'utiliser pour réduire la complexité du filtrage. Il existe également de nombreuses contraintes globales (faisant intervenir plusieurs, voire toutes les variables du problème), pour lesquelles un algorithme général établissant la GAC est impraticable. Des algorithmes spécialisés ont alors été développés.

Les principaux résolveurs commerciaux actuels (IlogSolver, CHIP, ...) utilisent un tel principe. Dans ces résolveurs, on utilise un filtrage spécialisé pour chaque type de contrainte, comme ceux définis dans le schéma AC5 pour obtenir la cohérence d'arc [Van Hentenryck *et al.*, 1992], et des contraintes globales sont proposées avec leur propre algorithme de filtrage. Citons, par exemple :

- la contrainte cumulative de Chip [Aggoun et Beldiceanu, 1993] pour les problèmes d'ordonnancement comprenant l'utilisation d'une ressource limitée,
- la contrainte "Alldiff" qui spécifie que les variables de cette contrainte doivent avoir des valeurs différentes. Un algorithme de couplage maximum réalise la cohérence d'arc généralisée (GAC) [Régin, 1994].
- la contrainte de cardinalité globale, qui spécifie le nombre d'occurrences de chaque valeur pour chacune des variables d'un ensemble de variables. Un algorithme de flot maximum établit alors la GAC [Régin, 1996].

Un nouveau niveau de cohérence locale

Dans cet axe des algorithmes établissant des niveaux de cohérence, P. Berlandier a proposé dans l'équipe une cohérence de chemin restreinte nommée RPC (Restricted Path Consistency) [Berlandier, 1995], cohérence intermédiaire entre la cohérence d'arc et la cohérence de chemin pour les CSP binaires. Cette cohérence a le principal avantage de ne pas créer de nouvelles contraintes, ni de modifier les relations des contraintes existantes, elle ne fait que vérifier les valeurs fragiles (qui n'ont qu'un seul support) pour voir si ce support est viable avec une troisième variable. Un algorithme basé sur AC4 [Mohr et Henderson, 1986] a été conçu et implanté pour réaliser cette nouvelle cohérence. Par la suite, une étude complète a été réalisée par R. Debruyne à Montpellier sur les différentes cohérences intermédiaires entre la cohérence d'arc et la cohérence de chemin[Debruyne, 1998] et leur maintien pendant la recherche de solutions.

Un nouvel algorithme de maintien de cohérence d'arc dans les CSP dynamiques
Nous avons proposé par ailleurs un maintien "simple" [Neveu et Berlandier, 1994] de la cohérence d'arc (sans maintien de structure de données de justifications) dans le cadre des CSP dynamiques. Les algorithmes qui établissent la cohérence d'arc comme AC3 [Mackworth, 1977] sont naturellement incrémentaux quand on ajoute une contrainte. Le problème de leur incrémentalité se pose lors du retrait de contraintes. Il faut en effet remettre des valeurs dans les domaines. Nous avons proposé un algorithme, nommé AC-DC, qui quand on retire une contrainte, ne remet que les valeurs enlevées du domaine des variables liées par cette contrainte et lance l'algorithme de cohérence d'arc (de type AC3) sur ces valeurs. (Y. Georget et P. Codognet ont aussi proposé une telle approche dans le cadre CLP [Georget et Codognet, 1999]). N. Jussien a ensuite proposé à Nantes une autre approche plus fine utilisant des justifications qui stocke lors d'un retrait de valeurs les causes de ce retrait [Jussien, 1997].

2.2.3 Recherches arborescentes tronquées

Quand l'espace de recherche devient trop grand pour être exploré entièrement par des méthodes complètes et que le problème a des solutions, on peut adopter un autre moyen de parcourir l'espace de recherche pour essayer de se focaliser sur les zones prometteuses en remettant à plus tard (si on en a le temps) l'exploration du reste. C'est le principe des recherches guidées par une heuristique de choix de valeur qui produisent un parcours limitant les divergences par rapport aux choix proposés par l'heuristique. Citons `LDS` [Harvey et Ginsberg, 1995], amélioré en `ILDS` [Korf, 1996], qui limitent le nombre de divergences et `DDS` [Walsh, 1997] qui limite la profondeur dans l'arbre de recherche à laquelle ces divergences peuvent se produire. Ces méthodes peuvent rester incomplètes ou devenir complètes, au prix d'une certaine redondance, si on les relance en augmentant la limite qui les définit jusqu'à obtenir l'arbre de recherche entier.

Nicolas Prcovic a étudié dans sa thèse effectuée dans l'équipe [Prcovic, 1998] les algorithmes à divergences limitées de type `LDS`. Nous avons proposé une variante de `LDS`, en restreignant la recherche à un sous espace prometteur [Prcovic et Neveu, 1998]. En chaque nœud de l'arbre de recherche, on compare la probabilité que la branche de gauche mène à une solution à une borne fixée B et le sous-arbre entier est élagué si cette probabilité est inférieure à B. Cette probabilité est estimée en utilisant le concept des solutions potentielles [Geelen, 1992].

Une étude théorique [Prcovic et Neveu, 1999a; 1999b] a d'autre part été menée sur l'arbre de recherche construit par les différentes méthodes de recherche en profondeur d'abord entrelacée (IDFS), à divergence limitée (LDS,ILDS), à divergence limitée en profondeur(DDS). Nous avons défini un modèle de recherche heuristique, qui fait l'hypothèse que la qualité de l'heuristique qui ordonne les successeurs d'un nœud croît avec la profondeur des nœuds dans l'arbre. Nous avons vérifié cette hypothèse avec l'heuristique des solutions potentielles [Geelen, 1992] utilisée avec un filtrage de type FC. Notre modèle définit un ordre partiel sur les feuilles de l'arbre de recherche suivant leur probabilité d'être une solution. Nous avons montré quelles stratégies de recherche visitent les feuilles en respectant cet ordre partiel. Notre étude a conclu que, parmi les méthodes existantes, seules IDFS pur et une modification de ILDS qui inverse l'ordre de parcours des branches le respectent.

2.2.4 Recherche alternée

En étudiant le comportement des heuristiques de choix de valeur, il a été remarqué que de mauvais choix en haut de l'arbre de recherche pouvaient avoir des conséquences catastrophiques.

Nous avons vérifié ce phénomène en réalisant des expérimentations sur des CSP aléatoires, avec l'heuristique de choix de valeur *max-promises*. Ces expérimentations sont présentées dans le chapitre 5. L'heuristique est efficace dans la zone des problèmes ayant beaucoup de solutions. Par contre, près du pic de complexité, l'efficacité de l'heuristique diminue. Ceci peut être expliqué par le fait que, quand l'heuristique se trompe, les effets peuvent être désastreux, car le sous-arbre sans solution sélectionné est celui qui a potentiellement la plus grande taille.

Pour remédier à ce phénomène, des algorithmes de recherche alternée dans différents sous-arbres (comme `IDFS`) [Meseguer, 1997] ont été conçus. Les premiers nœuds du haut de l'arbre de recherche sont construits en largeur. Quand on a atteint le nombre maximum de sous-arbres que l'on souhaite traiter de manière alternée (limite due à la mémoire utilisée qui est proportionnelle au nombre de sous-arbres), on lance une recherche qui alterne entre ces sous-arbres. On change de sous-arbre chaque fois qu'on a atteint une feuille (un échec).

Nous avons conçu un nouvel algorithme appelé recherche à focalisation progressive (`PFS`) [Prcovic et Neveu, 2000; 2002] qui au départ alterne des recherches entre les différents sous-arbres issus des premiers choix de valeur et accumule de l'information statistique sur ces sous-arbres. Cette information accumulée permet à l'algorithme de se focaliser ensuite vers le sous-arbre le plus prometteur. Nous avons proposé comme mesure actualisée au cours de la recherche la longueur moyenne des branches explorées dans un sous-arbre[2].

Au lieu de fonctionner en deux phases (apprentissage de la mesure sur un échantillon de la recherche, puis utilisation dans une recherche complète), nous avons intégré la phase d'apprentissage dans la recherche elle-même. Pour pouvoir utiliser la mesure dès le début de l'algorithme, nous avons proposé d'utiliser un critère biaisé. On initialise à une valeur égale au nombre de variables le critère de la longueur des branches de chaque sous-arbre en faisant comme si on en avait déjà exploré K branches. La moyenne de la longueur de branche du sous-arbre courant ne peut donc que baisser au début de l'algorithme et

[2]On suppose que l'on est dans le cas où chaque branchement correspond à l'affectation d'une variable avec ses différentes valeurs.

cela force l'alternance entre les différents sous-arbres. Quand suffisamment de branches ont été explorées dans chaque sous-arbre, le critère converge vers la mesure souhaitée, la longueur moyenne des branches et l'algorithme se focalise sur le sous-arbre le plus prometteur.

Cet algorithme est présenté en détail dans le chapitre 5 et quelques essais sur des CSP aléatoires en ont montré l'intérêt. Des études restent à mener pour régler automatiquement ce paramètre K.

La recherche alternée peut aussi être vue comme une simulation sur un seul processeur d'une recherche arborescente parallèle. Le paragraphe suivant étudie plusieurs manières d'utiliser du parallélisme pour résoudre des CSP.

2.2.5 Recherches arborescentes parallèles

Les méthodes de recherche arborescentes peuvent aussi être parallélisées et plusieurs études ont été menées dans ce sens. Nous avons étudié 3 principaux types de parallélisme : une parallélisation de l'arbre de recherche lui-même, une parallélisation basée sur les coûts pour un problème d'optimisation et une parallélisation basée sur le graphe de contraintes.

2.2.5.1 Parallélisation de l'arbre de recherche

Nous avons proposé un algorithme distribué [Prcovic *et al.*, 1996] qui trouve toutes les solutions d'un problème de satisfaction de contraintes et qui repose sur la répartition sur plusieurs processus de différents sous-arbres qui composent l'arbre de recherche du problème. Des nœuds sont alloués dynamiquement aux processus qui ont terminé un sous-arbre. Nous avons montré que les performances de l'algorithme sont optimales lorsque le problème à résoudre est de grande taille puisque l'accélération de la résolution devient asymptotiquement linéaire (par rapport au nombre de processus). L'algorithme a été implanté en utilisant Prose et Chooe [Lebastard, 1993a], un outil de communication entre processus LeLisp. Des tests ont ainsi pu être effectués pour vérifier son efficacité. Nous avons intégré des techniques connues d'élagage de l'arbre de recherche comme le *forward checking* et nous avons étudié comme intégrer le *nogood recording* [Schiex et Verfaillie, 1994].

D'autres algorithmes de répartition de la charge de travail ont été proposés comme [Kumar *et al.*, 1994], et il serait intéressant de les comparer en pratique.

Par la suite, nous avons étudié comment définir la profondeur à partir de laquelle il est intéressant de paralléliser la recherche quand on recherche une seule solution. Nous développons de manière séquentielle le haut de l'arbre et déterminons dynamiquement en fonction d'une évaluation heuristique du nombre de nœuds de chaque sous-arbre la profondeur de parallélisation [Prcovic et Neveu, 1997; Prcovic, 1997].

2.2.5.2 Parallélisation basée sur les coûts

Nous avons cherché à paralléliser des problèmes d'optimisation combinatoire sous contraintes.

Nous avons tout d'abord proposé un algorithme séquentiel, nommé `NDO` (Non Diffident Combinatorial Optimization Algorithm), qui est un intermédiaire entre l'algorithme classique d'optimisation par *séparation et évaluation (branch and bound)* (`B&B`) en profondeur d'abord qui effectue une seule recherche arborescente en mettant à jour la borne supérieure chaque fois qu'une meilleure solution est trouvée et l'algorithme "Iterative deepening" (`ID`) [Korf, 1985], qui effectue des recherches successives en augmentant les bornes inférieure et supérieure jusqu'à trouver une solution. L'originalité de `NDO` consiste simplement à effectuer une série de recherches en imposant à chaque recherche un intervalle de coûts (des bornes inférieure et supérieure) donné. En fonction des résultats de cette recherche, on détermine la valeur des bornes pour l'itération suivante.

L'intérêt de cet algorithme est d'essayer de mieux élaguer l'arbre de recherche que `B&B`, au risque de manquer la meilleure solution et d'être obligé de le reparcourir et d'être moins redondant que `ID`. Notre attention s'est portée sur le choix de la borne supérieure à chaque itération en tenant compte du critère à minimiser et de son influence sur le problème donné. Un article présente des heuristiques de choix et des résultats expérimentaux sur des problèmes aléatoires [Trombettoni *et al.*, 1995].

Une version *parallèle* de cet algorithme destinée à l'optimisation d'un critère de maintien de solutions a été implantée [Astier et Saada, 1994] : chaque processeur reçoit le même problème de contraintes mais impose une borne supérieure différente pour le critère. Les résultats de la parallélisation ont montré qu'une grande partie de l'effort était situé dans la preuve d'optimalité, c-à-d pour des coûts entiers entre la valeur optimale et l'entier suivant et qu'une telle parallélisation basée uniquement sur les coûts était insuffisante. En effet, la recherche se concentrait assez rapidement dans cette tranche de coûts et il devenait alors intéressant de passer à une parallélisation de l'arbre de recherche comme celle proposée au paragraphe précédent.

Signalons que des travaux sur ce sujet ont été menés de manière indépendante à l'ECRC et ont été implantés dans leur outil de programmation par contraintes (Eclipse) [Prestwich et Mudambi, 1995].

2.2.5.3 Parallélisation basée sur le graphe de contraintes

Nous avons étudié comment résoudre certains problèmes de contraintes qui se présentent comme une réunion de sous problèmes faiblement connectés entre eux. Notre motivation principale est de prendre en compte des problèmes de conception ou de configuration qui satisfont cette propriété (ils ont souvent une nature hiérarchique où un problème de niveau supérieur réalise cette connexion faible entre les problèmes de niveau inférieur).

Nous avons conçu un algorithme [Berlandier et Neveu, 1997] exploitant cette structure et implantant un retour arrière particulier : les variables de haut niveau sont instanciées en premier et les sous problèmes sont traités indépendamment les uns des autres. Des gains importants ont été réalisés par cet algorithme qui utilise le fait que les problèmes sont faiblement couplés par rapport à un algorithme de recherche arborescente standard. Cet algorithme est de plus naturellement parallélisable : là, l'accélération due à la parallélisation est plus importante pour les problèmes avec beaucoup de solutions que pour les problèmes sur-contraints.

Une généralisation de ce type d'approche a été proposée par Makoto Yokoo [Yokoo, 2001] qui a proposé différents algorithmes pour résoudre des CSP distribués, où les agents regroupant des variables sont hiérarchisés. Y. Hamadi a également proposé dans sa thèse à Montpellier d'autres algorithmes sur ce thème [Hamadi, 1999].

2.3 Algorithmes incomplets

Quand on a du mal à guider une recherche arborescente et que l'espace de recherche est trop grand pour être exploré entièrement, il devient intéressant d'utiliser des méthodes incomplètes, que ce soient des méthodes de recherche locale ou des méthodes à population comme les algorithmes génétiques ou mémétiques (hybrides d'algorithmes génétiques et de recherche locale). On trouvera dans [Hao *et al.*, 1999] et plus récemment dans [Dréo *et al.*, 2003] une présentation détaillée des diverses méthodes de recherche locale ou à population. Nous n'indiquons dans les paragraphes suivants qu'un résumé de ces méthodes en rapport avec les recherches que nous avons effectuées.

2.3.1 Recherche locale

Les méthodes de recherche locale ont été définies dans le cadre de l'optimisation combinatoire : elles essaient d'améliorer une instanciation complète des variables, appelée dans la suite configuration, dans l'espoir d'obtenir une solution optimale en lui faisant subir une suite de modifications. Dans le cas d'un problème de minimisation, elles ne peuvent garantir l'optimalité des solutions obtenues, sauf si cette solution a la même valeur qu'une borne inférieure obtenue par ailleurs. Le caractère local de ces méthodes réside dans la notion de voisinage : on cherche à remplacer la configuration courante par une configuration proche.

Quand il s'agit de résoudre un CSP, il faut se donner une fonction à optimiser : on prend souvent le nombre de contraintes non satisfaites que l'on cherche à minimiser (cadre MAX-CSP). Ce critère peut être affiné en donnant des poids aux contraintes : on minimise alors une somme pondérée des violations de contraintes.

Il faut aussi définir un voisinage, c.à.d l'ensemble des configurations atteignables par un mouvement à partir d'une configuration. Par exemple, le voisinage le plus simple pour un CSP est de changer la valeur d'une variable quelconque. Souvent, on réduit ce voisinage aux variables participant à un conflit dans la configuration courante.

Le problème avec son voisinage définit un paysage de recherche, qui peut être représenté par un graphe orienté dont les nœuds sont les configurations et les arcs relient deux voisins.

Contrairement aux méthodes arborescentes qui construisent une solution, les méthodes de recherche locale essaient de réparer la configuration courante qui ne vérifie pas toutes les contraintes. Tant que certaines contraintes ne sont pas satisfaites, on cherche dans le voisinage de la configuration courante une meilleure configuration.

Le caractère incomplet de ces algorithmes provient du fait qu'ils ne parcourent pas l'espace de recherche de manière systématique et ne peuvent garder en mémoire toutes les configurations déjà vues.

Aspects stochastiques

Pour éviter des biais dus à des choix déterministes qui feraient explorer toujours les mêmes parties de l'espace de recherche, une part d'aléatoire est introduit dans ces algorithmes, les rendant stochastiques.

Le choix de la configuration initiale peut être le résultat d'un algorithme glouton ou au contraire complètement aléatoire. Même dans le cas d'un algorithme glouton qui choisit pour chaque variable la meilleure valeur possible avec l'instanciation en construction, les valeurs ex-aequo sont départagées au hasard comme par exemple dans l'algorithme `GRASP` [Feo et Resende, 1995].

Le choix d'un voisin fait souvent intervenir une part de hasard. L'heuristique *min-conflict* [Minton *et al.*, 1992] par exemple choisit une des meilleures valeurs d'une variable en conflit : le fait que les choix ex-aequo soient départagés de manière aléatoire permet de diversifier et d'éviter le biais que produirait une heuristique déterministe. Nous avons remarqué en utilisant la recherche locale d'IlogSolver, qu'il vaut toujours mieux parcourir le voisinage de manière aléatoire. C'est pourquoi IlogSolver propose un mécanisme "IloRandomize" qui réordonne aléatoirement le voisinage.

Minimums locaux

L'algorithme de base de la recherche locale est l'algorithme de descente (ou HillClimbing pour les Anglo-Saxons qui préfèrent maximiser), qui n'accepte que des voisins améliorant (strictement ou non) l'évaluation de la configuration courante. Cet algorithme ne peut sortir des minimums locaux ou des plateaux du paysage de recherche. De nombreux mécanismes ont alors été définis pour essayer de s'échapper des minimums locaux. On les appelle généralement des métaheuristiques, car ce sont des méthodes heuristiques indépendantes du problème traité. Les plus connues sont la méthode avec liste taboue [Glover, 1986] qui garde une mémoire à court terme pour éviter de cycler et le recuit simulé [Kirkpatrick *et al.*, 1983] qui autorise des mouvements détériorants avec une probabilité qui décroît au cours du déroulement de l'algorithme. Une autre idée simple est de lancer plusieurs recherches de longueur limitée en repartant d'une configuration initiale choisie aléatoirement. Une telle méthode, appelée GSAT pour les problèmes SAT de satisfaction d'une formule booléenne, a donné des résultats intéressants [Selman *et al.*, 1992]. Cette méthode pour les problèmes SAT a par la suite été améliorée en donnant naissance à l'algorithme Walksat [Selman *et al.*, 1996]. On y a ajouté la possibilité d'effectuer avec une certaine probabilité quelques mouvements aléatoires qui permettent d'échapper aux minima locaux.

2.3.1.1 La bibliothèque logicielle INCOP

Nous avons constaté en 2001 l'absence de bibliothèque libre disponible permettant de développer et tester rapidement de nouveaux algorithmes incomplets. Seule, la bibliothèque Easylocal++ [DiGaspero et Schaerf, 2000] était proposée pour la recherche locale. De nombreux autres systèmes avaient été présentés dans des publications [Voß et Woodruff, 2002a], mais n'étaient pas disponibles.

Nous avons développé une nouvelle bibliothèque de méthodes incomplètes, dénommée INCOP [Neveu et Trombettoni, 2003c; 2003b] pour résoudre les problèmes d'optimisation combinatoire. Les problèmes de satisfaction de contraintes en domaines finis peuvent être résolus dans le cadre Max-CSP, c-à-d en minimisant le nombre de contraintes non satisfaites. Cette bibliothèque, implantée en C++, permet de définir facilement différents parcours de voisinage et différentes méta-heuristiques classiques (Metropolis, recuit simulé, acceptation à seuil, liste taboue) ainsi que d'implanter de nouvelles méthodes. Un effort particulier a été fait pour l'efficacité des algorithmes en effectuant une évaluation incrémentale des mouvements. Nous avons aussi proposé une gestion générale et originale du voisinage, ressemblant aux listes de mouvements candidats (candidate lists) proposées par Glover [Glover et Laguna, 1997]. Une nouvelle méthode simple `idw` pour marche avec intensification et diversification (Intensification Diversification Walk) à un seul paramètre a alors été définie dans ce cadre : il s'agit du nombre N de voisins à explorer pour choisir un mouvement. Le caractère aléatoire de cette méthode provient du fait qu'une portion du voisinage (au maximum N voisins) est parcourue aléatoirement, son caractère glouton du fait que le premier voisin parmi les N qui a une évaluation meilleure ou égale à la courante est pris. Deux variantes permettent de sortir des minimums locaux en remontant plus ou moins haut [Neveu *et al.*, 2004]. Si tous les N voisins visités détériorent la fonction d'évaluation, on choisit soit un voisin quelconque, soit le voisin détériorant le moins la fonction d'évaluation. Suivant les types de problèmes et le paysage de recherche, une de ces deux variantes est la plus efficace : par exemple, pour les problèmes d'affectation de fréquences du CELAR (Centre d'Electronique de l'Armement) au paysage très chahuté, il vaut mieux remonter assez haut, alors que pour les problèmes d'ordonnancement de véhicules de la CSPLIB, il vaut mieux au contraire remonter le moins haut possible. Le réglage du paramètre N du nombre de voisins examinés permet d'équilibrer l'approfondissement de la recherche dans le voisinage (N assez long) et le fait de s'échapper des minimums locaux (N assez court).

Un algorithme hybride gérant plusieurs configurations à la fois comme `GWW` et intégrant cette recherche locale `idw` (cf ci après `GWW-idw`) a aussi été réalisé et de bons résultats ont été obtenus sur des problèmes de coloriage de graphes et sur des instances d'affectation de fréquences du CELAR [Neveu et Trombettoni, 2004]. Nous avons par ailleurs réussi à colorier pour la première fois en 30 couleurs le graphe `flat_300_28` du challenge Dimacs[Johnson et Trick, 1996] avec l'algorithme de Metropolis (température fixe) et un voisinage donné par l'heuristique "Min-conflict" de Minton [Minton *et al.*, 1992](choisir une variable en conflit et une valeur pour cette variable minimisant les conflits). Cette bibliothèque peut également traiter des problèmes de clique maximum dans un graphe, des CSP binaires ou n-aires avec des contraintes données en extension, le problème des n-reines, les carrés latins, l'ordonnancement de chaînes de montage de véhicules défini dans la CSPLIB ...

Cette bibliothèque est présentée en détail dans le chapitre 6 et une première version est disponible sur le site :
`http ://www-sop.inria.fr/coprin/neveu/incop/presentation-incop.html`

2.3.2 Algorithmes à population

Au lieu de faire évoluer une seule configuration comme le font les algorithmes de recherche locale, des algorithmes ont été conçus pour explorer l'espace de recherche en faisant évoluer plusieurs configurations et en échangeant de l'information entre elles. On parle alors d'algorithmes à population : les plus célèbres d'entre eux sont les algorithmes génétiques dont le cadre a ensuite été élargi pour donner les algorithmes évolutionnistes.

2.3.2.1 Les algorithmes génétiques

Les algorithmes génétiques, introduits par Holland [Holland, 1975], sont des algorithmes très généraux d'optimisation combinatoire qui peuvent s'avérer utiles quand on manque d'information pour guider la recherche dans un problème d'optimisation ou quand on cherche plusieurs solutions différentes. Ils font évoluer une population de configurations en lui appliquant divers opérateurs inspirés par la théorie de l'évolution (sélection, mutation, croisement). Il s'agit donc d'une exploration de l'espace de recherche par plusieurs individus interagissant. Par la suite, on a étendu le cadre de ces algorithmes en proposant d'autres mécanismes de sélection ou de recombinaison : on parle alors d'algorithmes évolutionnistes.

Nous avons cherché à utiliser ces algorithmes génétiques pour résoudre des problèmes de satisfaction de contraintes. Mais les opérateurs génétiques généraux, conçus pour une modélisation par chaîne de bits, ne prenaient pas en compte les spécificités des problèmes de satisfaction de contraintes. Or, dans les CSP, on a souvent une structuration du problème et des corrélations entre variables (les contraintes) que les opérateurs classiques de croisement ne voient pas.

Quelques études pour résoudre des CSP avec des algorithmes génétiques sont présentées dans [Eiben et Ruttkay, 1997] et dans [Eiben, 2001].

Opérateurs spécialisés pour les CSP
Maria-Cristina Riff [Riff, 1997b] a mené une thèse sur le sujet. Notre idée a été de prendre en compte les caractéristiques du problème pour définir de nouveaux opérateurs génétiques et modifier la fonction d'évaluation. M-C Riff a tout d'abord proposé de biaiser la fonction d'évaluation en prenant en compte le graphe de contraintes lui-même. Au lieu de minimiser la somme des violations de contraintes, on pondère ces violations en fonction de la "difficulté probable" de satisfaire une contrainte en regardant localement la densité du graphe [Riff, 1996]. Elle a aussi proposé de nouveaux opérateurs de recombinaison et de mutation qui utilisent l'état de satisfaction des contraintes. Ces opérateurs nommés *Arc-mutation* et *Arc-crossover* ne fonctionnent plus en aveugle [Riff, 1997a]. Arc-crossover est un opérateur de recombinaison qui à partir de deux parents constitue un enfant. Celui-ci est construit par un processus glouton qui cherche à satisfaire en premier les contraintes portant sur le plus grand nombre de variables quand il choisit les valeurs des variables parmi celles des parents. Une variante nommée *Arc-crossover Dynamique Adaptatif* recalcule l'ordre de priorité des contraintes selon leur satisfaction par les parents [Riff, 1998], traitant en premier les contraintes non satisfaites dans les deux parents. L'opérateur de mutation Arc-mutation choisit une variable au hasard et pour cette variable, choisit la valeur (différente de la valeur courante) qui satisfait au mieux les contraintes.

Opérateurs généraux dynamiques ou adaptatifs
Une deuxième thèse effectuée par Blaise Madeline [Madeline, 2002] s'est plus intéressée au coloriage de graphes et aux méthodes évolutionnistes générales (sans spécialisation par rapport au problème). Nous avons proposé de nouveaux opérateurs généraux à paramétrage dynamique (taux de mutation adaptatif en fonction du degré de convergence de la population, taux de mutation suivant une fonction sinus). Nous avons conçu d'autre part un nouvel opérateur de croisement multi-points, qui offre plus de possibilités de croisements que celui à N-points classique et un opérateur de diversification pour le coloriage de graphe. Tous ces opérateurs ont été testés sur des problèmes de coloriage de graphe, dont certains sont issus de problèmes de dimensionnement de réseaux tout optique (technologie WDM) étudiés par le projet MASCOTTE de l'INRIA [Madeline et Neveu, 2001]. Nous avons utilisé pour ces tests la bibliothèque AgCSP, bibliothèque C++ pour résoudre des problèmes de satisfaction de contraintes avec des algorithmes génétiques, développée dans l'équipe par Fabrice Didierjean.

2.3.2.2 GWW

Les algorithmes génétiques, avec leurs très nombreuses variantes, nous semblant difficiles à régler et surtout les opérateurs globaux comme le croisement ne permettant pas des évaluations incrémentales des individus, nous avons expérimenté une autre méthode à population plus simple : Go with the winners (`GWW`).

Principes de l'algorithme
L'algorithme `GWW` a été introduit dans [Aldous et Vazinari, 1994] pour trouver une feuille de profondeur maximum dans un arbre. Au début, plusieurs particules sont placées à la racine de l'arbre. L'algorithme alterne ensuite phases d'exploration et de regroupement. Dans la phase d'exploration, chaque particule est déplacée sur un nœud fils choisi aléatoirement. Dans la phase de regroupement, les particules qui ont atteint une feuille sont redistribuées : elles sont remplacées par une copie d'une autre particule non feuille choisie aléatoirement. Le processus s'arrête quand toutes les particules sont sur une feuille. Les auteurs ont déterminé le nombre de particules nécessaire pour trouver le nœud le plus profond avec une forte probabilité dépendant d'une mesure d'équilibrage de l'arbre.

L'algorithme `GWW` que nous avons étudié est présenté dans [Dimitriou et Impagliazzo, 1996]. C'est une modification de la première version pour l'optimisation combinatoire. Un arbre apparaît en effet dans tout problème d'optimisation quand on gère un seuil et utilise une méthode de recherche locale qui interdit tout mouvement franchissant le seuil. Le graphe de recherche est formé par la notion de voisinage : les nœuds du graphe correspondent aux points de l'espace de recherche et deux nœuds sont reliés par un arc s'il existe un mouvement élémentaire allant du premier au second. Un tel graphe peut alors être divisé hiérarchiquement en plusieurs sous-graphes selon leurs coûts. Plus précisément, un nœud de l'arbre est constitué d'un sous-graphe de recherche dont les sommets ont un coût inférieur ou égal au seuil. Le sommet de l'arbre contient le graphe de recherche entier (quand le seuil est au plus haut). Baisser le seuil divise le graphe de recherche en composantes connexes qui forment une hiérarchie entre un nœud de l'arbre et ses fils. Le fait que deux régions deviennent déconnectées quand le seuil décroît signifie qu'aucune marche aléatoire (restant au niveau du seuil ou sous le seuil) ne peut déplacer une particule d'une région à l'autre.

```
algorithme GWW (B : nombre de particules ; S : longueur de la marche, T : paramètre seuil ;
WP : paramètres de la marche) : configuration
    Initialisation : chaque particule est aléatoirement placée sur une configuration et rangée
    dans le tableau Particules
    seuil ← coût plus mauvaise particule
    Boucle
        seuil ← Baisserseuil(seuil, T, Particules)
        si Meilleur(Particules) > seuil alors retourner(Meilleur-trouvé) /* la
        meilleure configuration trouvée durant la recherche */
        Redistribuer(Particules) /* Place chaque particule ayant un coût supérieur au
        seuil sur la configuration d'une particule sous le seuil choisie aléatoirement */
        pour tout particule p dans Particules faire
            pour i de 1 à S faire
                voisin ← Cherchervoisin(p, WP)
                Déplacer(p,voisin) /* Mouvement de p sur une nouvelle configuration */
            fin.
        fin.
    fin.
fin.
```

Algorithm 1: L'algorithme `GWW pour l'optimisation combinatoire` : l'hybridation avec la recherche locale réside dans l'implantation de la fonction Cherchervoisin

Hybridation avec de la recherche locale

Pour accélérer la recherche, nous avons adopté une stratégie d'hybridation avec de la recherche locale, idée qui s'est avérée fructueuse pour les algorithmes évolutionnistes en aboutissant à la définition du cadre des algorithmes mémétiques [Moscato, 1999], qui appliquent systématiquement une méthode de descente après chaque individu créé pour n'avoir dans la population que des optimums locaux.

Nous avons remplacé l'étape de marche aléatoire de `GWW` par de la recherche locale pour favoriser les meilleures solutions. Nous avons proposé une instance de cet algorithme hybride, appelée `GWW-idw`), où la marche de chaque particule est effectuée avec l'algorithme `idw` pour "intensification diversification walk" défini en 2.3.1.1)[Neveu et Trombettoni, 2003d; 2003a]. Une attention particulière a été portée sur l'efficacité pour des problèmes de taille réaliste. Les évaluations des mouvements testés et effectués pendant la marche utilisent les structures de données incrémentales de la bibliothèque INCOP. Nous avons comparé différentes manières de baisser le seuil. Dans l'algorithme de base, on optimisait sur des petits entiers et on pouvait baisser le seuil de 1 à chaque fois, ce qui n'est plus le cas avec des valeurs d'évaluation plus grandes. Nous avons aussi adopté l'élitisme défini dans les algorithmes génétiques : nous remettons à chaque changement de seuil dans la population le meilleur individu trouvé jusqu'alors.

Un seul paramètre a été ajouté pour intégrer la recherche locale dans le schéma existant : il s'agit du paramètre de la méthode `idw`, le nombre de voisins explorés à chaque pas de la marche. L'algorithme `GWW-idw` a finalement 4 paramètres : le nombre de particules, la vitesse de baisse du seuil, la longueur de la marche de chaque particule et le nombre de voisins explorés à chaque pas de la recherche locale. Une méthode simple pour régler en séquence ces paramètres a été proposée. Cet algorithme a été implanté dans la bibliothèque INCOP.

Une étude expérimentale a été menée sur des instances difficiles de coloriage de graphe issues du challenge DIMACS et sur des problèmes d'affectation de fréquences du CELAR. Les meilleures bornes connues ont été trouvées sur toutes les instances.

Une présentation détaillée de cet algorithme se trouve dans le chapitre 7.

2.4 Autres recherches

Citons ici deux autres recherches effectuées dans le projet Secoia dans le domaine des contraintes.

Philippe Ballesta a conçu dans sa thèse un système d'aide à la composition musicale en utilisant la programmation par contraintes [Ballesta, 1994]. Un prototype a été réalisé,dans le cadre particulier, mais non restrictif, de la musique tonale en utilisant l'outil de programmation par contraintes `PECOS` d'ILOG. Un certain nombre de structures musicales omniprésentes dans ce type de musique (notes, intervalles, accords ou tonalité) ont été représentées

Le réemploi et l'extension de telles structures ont conduit au développement d'un outil d'aide dédié à la résolution d'exercices d'harmonie à quatre voix. Ces structures de données ont formé les variables du CSP, les contraintes étant formées par les règles de l'harmonie (distance entre les voix, mouvements mélodiques autorisés ...)

Nicolas Chleq a étudié dans sa thèse le raisonnement temporel abductif (génération d'hypothèses de persistance). Il a placé un tel raisonnement dans le cadre de la programmation logique avec contraintes, en gérant ces hypothèses de persistances comme des contraintes temporelles quantitatives [Chleq, 1995b].

Chapitre 3

Résolution de problèmes de contraintes sur domaines continus

Ce chapitre présente les recherches effectuées dans les domaines continus. Alors que les contraintes en domaines finis se retrouvent essentiellement dans les problèmes de recherche opérationnelle comme l'ordonnancement, l'affectation de ressources, les emplois du temps, les tournées de véhicule, les problèmes en domaines continus se trouvent eux plutôt dans des domaines faisant intervenir des grandeurs physiques ou géométriques comme en CAO, dans le graphique. Nous nous sommes intéressés à la résolution de deux types de contraintes, les contraintes fonctionnelles (3.1) qui apparaissent souvent dans la modélisation des interfaces graphiques et les contraintes géométriques (3.2) pour lesquelles nous avons plus particulièrement étudié les méthodes de décomposition du système de contraintes.

3.1 Contraintes fonctionnelles

Nous quittons les domaines finis pour aborder les domaines continus. Dans une cette partie, nous présentons les recherches menées dans le domaine des contraintes fonctionnelles, c'est-à-dire des contraintes contenant des formules explicites pour calculer une variable de la contrainte en fonction des autres. Ces recherches ont fait l'objet de la thèse de Gilles Trombettoni [Trombettoni, 1997].

Le modèle le plus simple est celui des tableurs [Hudson, 1994], où les contraintes sont unidirectionnelles (les formules sont écrites dans un seul sens).

Nous nous sommes intéressés aux contraintes multi-directionnelles, utiles pour modéliser des interfaces graphiques, en spécifiant la relation à vérifier et différentes méthodes pour recalculer une variable en fonction des autres. Ce modèle, qui remonte à Sketchpad [Sutherland, 1963], est pourtant encore peu utilisé à cause de doutes concernant l'efficacité des techniques de résolution par propagation locale d'une part, et du manque de prédictibilité de la solution obtenue [Myers *et al.*, 2000]. Chaque contrainte qui porte sur un ensemble de k variables possède différentes méthodes qui (r)établissent la satisfaction de la contrainte en recalculant une ou plusieurs variables, appelées variables de sortie de la méthode en fonction d'autres variables, les variables d'entrée.

Les algorithmes de propagation locale, contrairement aux algorithmes réactifs procèdent en 2 phases : une première phase, dite de planification, construit un graphe orienté de méthodes, si possible sans circuit et une deuxième phase applique ces méthodes dans un ordre topologique compatible avec ce graphe. Les circuits peuvent être traités par une résolution globale des contraintes y participant. On atteint alors les limites de la propagation locale.

3.1.1 Nouveaux algorithmes de propagation locale

Gilles Trombettoni a proposé dans sa thèse plusieurs nouveaux algorithmes de propagation locale pour résoudre des systèmes de contraintes fonctionnelles. Citons en particulier OpenPlan et GPDOF qui traitent la phase de planification dans des cadres plus étendus que PDOF.

Ces algorithmes se situent dans le cadre de l'algorithme de propagation de degrés de liberté PDOF [Sutherland, 1963]. Cet algorithme recherche une méthode libre, c'est-à-dire une méthode recalculant une variable liée à aucune autre contrainte. Il détecte donc la contrainte à résoudre en dernier, l'enlève du graphe et continue sa recherche de méthode libre, jusqu'à que le graphe soit vide ou qu'il soit bloqué. Un échec de cet algorithme signifie qu'il n'existe pas de graphe de méthodes sans circuit.

GPDOF traite le cas où des méthodes sont définies pour résoudre non seulement une, mais plusieurs contraintes à la fois. Cela permet de traiter certains cycles simples dans le graphe de contraintes pour lesquels on connaît une méthode de résolution. Une application de cet algorithme a été effectuée pour un problème de construction de modèle géométrique 3D à partir d'images 2D et de contraintes géométriques sur les objets [Trombettoni et Wilczkowiak, 2003]. Les méthodes correspondent à des résolutions géométriques simples, par exemple la résolution de deux contraintes de distance entre 3 points revient à calculer l'intersection de deux cercles.

L'algorithme OpenPlan fait appel à des résolveurs annexes quand PDOF est bloqué et ne trouve plus de méthode libre à appliquer. OpenPlan construit alors des "petits" ensembles de contraintes appelés blocs libres qui pourront être résolus en fin de séquence par un résolveur annexe. Le problème de minimiser la taille du plus grand de ces sous-ensembles de contraintes semble être NP-difficile (mais la démonstration n'a, à notre connaissance, pour le moment pas été effectuée). Nous avons donc proposé deux versions de cet algorithme [Bliek *et al.*, 1998] une version OpenPlanSB, de complexité exponentielle qui minimise le critère de taille des blocs et une version heuristique OpenPlanHM qui, en appliquant un algorithme de couplage maximum, trouve un bloc libre et essaie de réduire sa taille de manière heuristique par un algorithme de HillClimbing.

3.1.2 Analyse de complexité

Certains algorithmes de résolution par propagation locale ont une phase de planification qui a une complexité en pire cas polynomiale (comme QuickPlan [Vander Zanden, 1996], DeltaBlue [Freeman-Benson *et al.*, 1990]), d'autres (comme SkyBlue [Sanella, 1994]) exponentielle. En fait, chaque système existant traite un problème spécifique et les conséquences des diverses restrictions dans le formalisme des contraintes traitées sur la complexité théorique n'étaient pas déterminées précisément. Nous avons présenté des résultats de complexité théorique qui ont permis de classifier les différents problèmes traités lors de la phase de planification par les systèmes existants. Les principales caractéristiques discriminantes sont la recherche de graphes de méthodes solutions interdisant ou non les circuits, la présence de méthodes avec plusieurs variables de sortie, le fait que toutes les variables d'une contrainte apparaissent en entrée ou en sortie dans une méthode, la présence de hiérarchies de contraintes. Nous avons démontré en particulier que le problème traité par SkyBlue (recherche de solution avec ou sans circuit pour des méthodes à plusieurs variables de sortie) est NP-difficile [Trombettoni et Neveu, 1997] même en l'absence de hiérarchie de contraintes.

3.1.3 Liens sémantiques : algorithme Azurelink

Un autre handicap des méthodes de propagation locale est le manque de prédictibilité, ce qui peut être très gênant dans les systèmes interactifs. En effet, après une interaction de l'utilisateur comme l'ajout d'une contrainte, une nouvelle solution est recalculée par le système et peut être très différente de celle attendue par l'utilisateur. Nous avons proposé le formalisme [Trombettoni et Neveu, 2001] des liens sémantiques pour justement spécifier localement quelle méthode doit être lancée pour rétablir chaque contrainte. Ce modèle indique, pour chaque variable perturbée dans une contrainte, quelle est l'unique méthode qui doit être déclenchée pour rétablir la satisfaction de la contrainte. Nous avons montré que ce nouveau modèle permet d'obtenir un algorithme polynomial nommé Azurelink pour recalculer une solution satisfaisant les contraintes mais qu'une extension du modèle permettant un choix de plusieurs méthodes par interaction rend le problème correspondant NP-complet. Ce modèle semble être un bon compromis entre le très utilisé modèle des tableurs et celui des contraintes multi-directionnelles.

3.1.4 Hiérarchie de contraintes

Une thèse a été menée dans l'équipe par Mouhssine Bouzoubaa [Bouzoubaa, 1996] sur les algorithmes de résolution de hiérarchies de contraintes fonctionnelles dans le cadre du formalisme défini par A. Borning [Borning *et al.*, 1992]. M. Bouzoubaa a conçu un nouvel algorithme, Houria, qui agrège les erreurs dans chaque niveau et entre niveaux de manière globale, alors que les principaux algorithmes de propagation locale ne prenaient en compte que des critères locaux (ils comparaient deux solutions en comparant les erreurs contrainte par contrainte sans agrégation : une solution localement meilleure au niveau k qu'une autre doit satisfaire de la même manière un sous ensemble de contraintes du niveau k et mieux les contraintes restantes de ce niveau).

D'autres méthodes réalisent une agrégation des critères, mais abandonnent la propagation locale pour une résolution globale : les contraintes doivent alors être linéaires. On peut citer QOCA quand le critère global est quadratique (somme de carrés d'erreurs) ou Cassowary [Borning *et al.*, 1997] quand le critère global est est une somme de valeurs absolues d'erreurs. Dans le système Hirise, le critère reste local, la résolution globale permet de résoudre des systèmes linéaires assez gros [Hosobe, 2000].

3.2 Contraintes géométriques et méthodes de décomposition

Les contraintes géométriques sont apparues dans des applications de conception mécanique traitées avec Smeci [Trousse, 1989]. Quand le projet Secoia s'est intéressé aux contraintes, nous avons donc naturellement étudié la résolution de systèmes de contraintes géométriques. Il s'agit de manière générale de systèmes de contraintes où l'on cherche à placer un ensemble d'objets dans un espace. Il peut s'agir de problèmes d'architecture, de CAO mécanique, de chimie moléculaire ...

3.2.1 Aménagement spatial : approche par discrétisation

Dans les problèmes d'aménagement spatial, une discrétisation préexiste souvent (éléments préfabriqués de dimensions données). Philippe Charman a mené dans l'équipe une thèse sur ce sujet [Charman, 1995]. Nous avons particulièrement étudié les problèmes d'aménagement d'appartement en 2D avec des systèmes géométriques composés de rectangles parallèles aux axes. Nous avons proposé une représentation particulière des objets avec des variables bidimensionnelles pour ces rectangles et un filtrage particulier pour la contrainte de non recouvrement entre 2 rectangles, qui a mené à définir la notion d'arc-cohérence semi-géométrique.

3.2.2 Les méthodes de décomposition

Les problèmes de contraintes géométriques rencontrés en CAO, chimie, sont généralement formés de systèmes d'équations ayant un nombre fini de solutions, les inégalités ne servant qu'à sélectionner certaines solutions. Le cadre étudié ci-après est donc celui de la résolution de systèmes d'équations sans optimisation.

Chaque équation ne fait souvent intervenir qu'un petit nombre d'objets généralement proches. Les systèmes sont donc peu denses et se prêtent bien aux méthodes de décomposition. Nous avons étudié deux décompositions, une générale et une spécifique aux systèmes comprenant des sous-parties rigides.

3.2.2.1 Décomposition équationnelle

En étudiant les problèmes continus généraux mais restant peu denses, nous avons proposé l'utilisation d'une méthode de décomposition de systèmes d'équations. Une première décomposition canonique, de Dulmage et Mendelsohn [Dulmage et Mendelsohn, 1958], basée sur un algorithme de couplage maximum du graphe bi-parti contraintes variables, isole les parties structurellement bien contraintes, sur-contraintes et sous-contraintes. Une deuxième décomposition fine [König, 1916] de la partie bien contrainte produit un graphe sans circuit de blocs carrés. Ces blocs peuvent être résolus en suivant un ordre topologique sur ce graphe. Les algorithmes réalisant ces décompositions sont explicités dans [Pothen et Fan, 1990] et une telle décomposition a été mise en œuvre dans [Ait-Aoudia *et al.*, 1993] pour la résolution de contraintes géométriques.

Nous avons repris cette décomposition et nous avons utilisé pour la résolution de chaque bloc une méthode de résolution par intervalles, celle de IlogSolver. Plusieurs ordres de grandeur peuvent ainsi être gagnés en temps de résolution par rapport à une résolution par intervalles sur le système initial non décomposé.

Nous avons étudié les mécanismes de retour-arrière entre blocs en adaptant des méthodes existantes pour la résolution de CSP en domaines finis. Nous avons proposé une méthode originale de retour-arrière entre blocs [Bliek *et al.*, 1998], inspirée du Partial Order Backtracking [McAllester, 1993], utilisant un stockage de “nogoods” comme l'algorithme Dynamic Backtracking [Ginsberg, 1993]. Nous avons par la suite proposé une deuxième méthode de retour-arrière inter-blocs [Jermann *et al.*, 2003b], nommée Inter-Block Backtracking (IBB), plus simple, inspirée par le Graph Based Backjumping [Dechter, 1990]. Nous avons également étudié des mécanismes de propagation des contraintes inter-blocs, mais ceux-ci se sont souvent révélés être moins efficaces qu'une propagation effectuée uniquement à l'intérieur des blocs. Le chapitre 9 détaille cette recherche.

3.2.2.2 Décomposition géométrique

En étudiant plus particulièrement la résolution des systèmes de contraintes géométriques, nous nous sommes intéressés aux méthodes utilisant la rigidité. Ces études ont fait l'objet de la thèse de Christophe Jermann [Jermann, 2002].

Nous nous sommes limités alors aux systèmes d'objets géométriques simples (points, droites, plans), en 2D et en 3D, reliés par des contraintes géométriques topologiques (incidence, parallélisme) ou métriques (distance, angle).

Pour résoudre un système rigide (n'ayant que des déplacements comme mouvements) ayant des sous-parties elles-mêmes rigides, il est intéressant de résoudre indépendamment chaque sous-partie rigide et d'assembler ensuite le tout. Plutôt que d'avoir à résoudre un gros système d'équations, on est ramené à la résolution successive de systèmes de plus petites tailles : on peut ainsi gagner des ordres de grandeur dans le temps de résolution des méthodes par intervalles.

On peut trouver dans le mémoire d'HDR de Pascal Schreck [Schreck, 2002] une présentation détaillée de méthodes de résolution de contraintes géométriques et en particulier celles qui utilisent la rigidité qui correspond à l'invariance par déplacement des solutions.

Nous avons plus particulièrement étudié les méthodes de détection et d'assemblage de sous-parties rigides proposées par l'équipe de C. Hoffmann. La technique est appelée rigidification récursive : on détecte de petits sous-systèmes rigides qui vont ensuite être assemblés dans des systèmes rigides plus gros, jusqu'à si possible le système entier si ce dernier est rigide. Ces algorithmes fonctionnent généralement en deux phases : une phase de planification qui détermine les différents sous-systèmes et une phase de résolution numérique qui résout de manière effective les systèmes d'équations des sous-systèmes.

Des premiers systèmes à base de règles reconnaissaient à partir d'un répertoire de formes rigides connues, certaines sous-parties rigides [Bouma *et al.*, 1995]. Un schéma général d'algorithme basé sur un algorithme de flot maximum avait ensuite été présenté dans [Hoffmann *et al.*, 1997], pour détecter des parties rigides et les assembler, mais les heuristiques pour déterminer la rigidité étaient mises en défaut dans de nombreux cas. Elles étaient en effet basées sur la rigidité structurelle. De même, la version de base de l'algorithme de détection de systèmes rigides ou sur-rigides devait être modifiée pour prendre en compte aussi de nombreux cas particuliers.

La rigidité structurelle généralement utilisée est une extension aux objets géométriques du théorème de Laman [Laman, 1970], qui caractérise la rigidité générique de système de barres (points et contraintes de distance) en 2D. Elle se fonde sur un calcul simple des degrés de liberté du système géométrique à partir des degrés de liberté des objets eux-mêmes et des contraintes.

Rigidité structurelle étendue
Nous avons défini une nouvelle heuristique pour la rigidité, appelée rigidité structurelle étendue, [Jermann *et al.*, 2000] qui essaie de tenir compte des caractéristiques géométriques du système et du nombre de déplacements indépendants que ces caractéristiques permettent. Nous avons appelé cette notion **degré de rigidité**. Ainsi, en 3D, un système rigide formé de points alignés sur une droite n'a que 5 degrés de rigidité, alors qu'un triangle a 6 degrés de rigidité.

Détermination de parties bien ou sur rigides
Nous avons aussi proposé de nouveaux algorithmes, toujours à base de calculs de flots maximums comme celui de [Hoffmann *et al.*, 1997] pour détecter des systèmes bien ou sur-rigides [Jermann *et al.*, 2003a]. La principale différence réside dans la manière de poser la contrainte de placement de repère sur des sous-ensembles d'objets, en prenant en compte notre nouvelle heuristique de rigidité structurelle étendue. Ces algorithmes sont détaillés dans le chapitre 8. Ils peuvent être utilisés pour déterminer d'une part des parties structurellement sur-contraintes et pour d'autre part effectuer la phase de planification d'un algorithme de rigidification récursive.

Limites
Dans la cas de la détection d'une sur-rigidité structurelle, il reste à déterminer si cela correspond à un système réellement surcontraint sans solution ou à une contrainte supplémentaire permettant de discriminer la solution voulue parmi l'ensemble des solutions du système bien-contraint, comme dans le cas de la conformation de molécules. On a en effet besoin de plus de mesures de distances que celles garantissant la rigidité pour obtenir une solution unique [Hendrickson, 1992].

Il existe des cas où le degré de rigidité d'un système dépend de la solution elle-même : des singularités peuvent être présentes dans certaines solutions et pas dans d'autres. Il faudrait dans ce cas au lieu de faire un plan d'assemblage a priori le réaliser au fur et à mesure de la résolution qui va alors entrelacer phases de planification et de résolution.

La rigidité structurelle étendue n'est par ailleurs qu'une heuristique qui ne prend pas en compte les singularités dépendant des valeurs numériques des équations (comme par exemple des valeurs de distance rendant 3 points alignés). Elle ne peut non plus détecter certaines sur-rigidités (comme le contre-exemple de la "double banane" qui invalide l'extension du théorème de Laman en 3D), qui peuvent être trouvées par d'autres méthodes comme la méthode numérique probabiliste [Lamure et Michelucci, 1998] ou des modélisations non cartésiennes [Ortuzar et Serré, 2003].

Chapitre 4

Voies de recherche

Deux principales voies de recherche m'intéressent actuellement, une première concerne les méthodes incomplètes pour les CSP à domaines finis, une deuxième concerne les contraintes sur domaines continus, avec en particulier les résolveurs par intervalles, thème principal du projet COPRIN.

4.1 Voies de recherche en recherche locale

4.1.0.3 Réglage automatique des paramètres

Les méthodes de recherche locale ont souvent des paramètres dont le réglage est primordial pour l'efficacité de la méthode (la température pour l'algorithme de Metropolis, le schéma de baisse de température pour le recuit simulé, la longueur de la liste taboue).

Souvent, le temps de réglage des paramètres n'est pas pris en compte dans les résultats expérimentaux, qui ne présentent que les résultats obtenus avec les meilleurs paramétrages. Il semble donc important, quand une nouvelle méthode est définie, soit de fournir un paramétrage par défaut robuste, soit de fournir un mode si possible automatique de réglage des paramètres.

Nous avons défini une nouvelle méthode, assez simple, `idw` [Neveu *et al.*, 2004], qui a pour principal paramètre la taille du voisinage exploré autour de la configuration courante. Nous avons mis en œuvre une méthode de réglage automatique de ce paramètre qui alterne phases de réglage sur des marches courtes et phase d'éxécution avec le paramètre réglé.

Nous avons aussi proposé dans [Neveu et Trombettoni, 2003d] une méthode de réglage des 4 paramètres de l'algorithme `GWW-idw`. Nous souhaiterions également automatiser cette méthode.

4.1.0.4 Adaptation des paramètres

Au lieu d'essayer de déterminer a priori un réglage des paramètres avant de lancer la recherche, des études ont été menées pour changer dynamiquement les valeurs des paramètres pendant la recherche : on parle alors de méthodes dynamiques, si ce changement ne dépend pas de l'état de la recherche (paramètre suivant une fonction sinusoïdale [Madeline, 2002], paramètre variant aléatoirement) ou de méthodes adaptatives quand le changement dépend de l'état de la recherche (difficulté à sortir d'un minimum local ou au contraire difficulté à se focaliser dans des zones susceptibles d'améliorer l'évaluation).

On peut citer les méthodes à voisinage variable (`VNS`) [Mladenovic et Hansen, 1997], qui pour sortir d'un minimum local, augmentent la taille du voisinage exploré. D'autres méthodes règlent de manière adaptative la longueur de la liste taboue comme dans [Hao *et al.*, 1998] ou le schéma de baisse de température du recuit simulé [Anagnostopoulos *et al.*, 2003]. Les algorithmes évolutionnistes ont poussé cette idée jusqu'à intégrer les paramètres dans la configuration et optimiser à la fois la solution et la manière d'explorer l'espace de recherche. On parle alors d'algorithmes auto-adaptatifs [Bäck, 1997].

Il serait intéressant d'étudier une variante adaptative de `idw`, qui modifierait la taille du voisinage exploré en fonction de la recherche. Une longue stagnation pourrait par exemple faire augmenter cette taille.

D'autres méthodes enfin adaptent la fonction d'évaluation elle-même en ajoutant des pénalités aux contraintes qui restent non satisfaites (`SAW` [Eiben et van der Hauw, 1997]).

4.2 Voies de recherche dans le domaine continu

4.2.1 Résolveurs de contraintes dans le domaine continu

Dans l'équipe Coprin, nous étudions plus particulièrement les résolveurs de contraintes dans le domaine continu. Il s'agit alors de résoudre des systèmes d'équations et inégalités avec des variables dont les domaines sont des intervalles de réels. On est dans le cadre des CSP numériques [Lhomme, 1993].

Résolveurs existants
Les méthodes arborescentes complètes par intervalles sont utiles pour trouver toutes les solutions d'un système d'équations ayant un nombre fini de solutions isolées.

En effet, les méthodes numériques itératives classiques trouvent une solution, celle-ci dépendant du point de départ de la méthode. Pour trouver toutes les solutions, il existe trois grands types de méthodes :

- les méthodes de calcul formel qui sont, de part leur complexité, limitées à des systèmes d'équations algébriques de petite taille. Citons les bases de Gröbner [Buchberger, 1985] et les ensembles caractéristiques [Ritt, 1950; Wu, 1986].
- les méthodes homotopiques qui sont plus ou moins faciles à mettre en œuvre en fonction de la proximité entre le système à résoudre et le système initial [Lamure et Michelucci, 1998] (voir la thèse de C. Durand [Durand, 1998] pour leur utilisation pour résoudre des systèmes de contraintes géométriques).
- les méthodes par intervalles, très générales, qui nécessitent seulement un encadrement a priori des solutions.

Le projet se focalise sur les méthodes par intervalles, qui ont besoin de bornes données a priori pour les domaines des variables, cas qui se produit toujours dans les applications de robotique étudiées par COPRIN. Cette restriction devient alors un avantage car les solutions obtenues sont garanties de respecter ces bornes. Ces méthodes ont de plus l'avantage de fournir un encadrement sûr de chacune des solutions. Aucune solution n'est perdue. Dans certains cas, des théorèmes d'unicité garantissent également qu'il existe une solution et une seule dans les boîtes rendues. Elles réalisent, comme les méthodes de recherche arborescente définies pour les CSP en domaines finis, une recherche arborescente en alternant bissection des intervalles et filtrage des domaines par propagation des contraintes. Plusieurs degrés de filtrage ont été définis ces dernières années. Citons des consistances locales comme la 2B-consistance (appelée aussi Hull-consistance) et la Box-consistance [Benhamou *et al.*, 1994], des consistances plus globales comme la 3B-consistance [Lhomme, 1993] ou la Bound-consistance.

Deux résolveurs par intervalles ont déjà été développés dans l'équipe, ALIAS [Merlet, 2001] et ICOS [Lebbah, 2002]. ALIAS est un résolveur en C++, interfacé avec Maple, qui permet de résoudre des systèmes d'équations et d'inégalités en mixant des techniques d'analyse numérique et quelques techniques de propagation de contraintes (2B, 3B). ICOS est un résolveur qui contient diverses techniques de filtrage (2B, 3B, Box, Quad [Lebbah *et al.*, 2003], Newton par intervalles) et permet un pilotage assez fin de ces techniques (notions de points fixes, de précision, ...).

D'autres résolveurs ont aussi été développés comme Numerica [Van Hentenryck *et al.*, 1997], en partie réimplanté dans IlogSolver[ILO, 1997], comme Declic [Goualard, 2000] intégré dans GNU-Prolog ou RealPaver [Granvilliers, 2003], mais sont des boîtes noires, difficiles à utiliser par une équipe de recherche pour développer de nouvelles méthodes.

Recherches à mener

Nous sommes en train de définir un nouveau résolveur, facilement paramétrable, où l'on puisse combiner diverses méthodes de filtrage par propagation de contraintes et méthodes provenant de l'analyse numérique (application du théorème de Kantorovitch pour l'existence d'une solution unique dans une boîte, méthode de Newton par intervalles).

Nous avons montré que la consistance d'arc n'était pas toujours réalisable pour les contraintes continues [Chabert *et al.*, 2004] et proposé une méthode réalisant des "bissections naturelles" en utilisant les parties monotones des contraintes pour trouver un ensemble de boîtes arc-consistantes. Il reste à valider cette méthode expérimentalement.

Dans les cas où l'obtention de toutes les solutions est trop coûteux, il peut être judicieux d'utiliser les algorithmes de recherche arborescente tronquée de type LDS. De telles techniques doivent aussi pouvoir être utilisées dans le cas de problèmes d'optimisation.

Il est aussi, je pense, intéressant pour les problèmes d'optimisation de pouvoir utiliser de la recherche locale pour améliorer une borne quand une solution est trouvée et ainsi permettre d'avoir un meilleur élagage de l'arbre de recherche.

Enfin, quand l'espace de recherche est hors d'atteinte des méthodes arborescentes complètes, les méthodes de recherche locale ou à population peuvent s'avérer intéressantes pour l'optimisation globale. Bien que les métaheuristiques comme la recherche tabou ou le recuit simulé aient été conçues pour résoudre des problèmes discrets d'optimisation combinatoire, quelques essais ont montré leur utilité dans les domaines continus [Chelouah, 1999]. Il faudrait étudier comment adapter la recherche dans le voisinage de notre méthode de recherche locale `idw`.

De même, il serait intéressant d'étudier comment adapter la méthode `GWW`. D'autres méthodes à population, comme les stratégies d'évolution (ES) [Rechenberg, 1973], ou la programmation évolutionniste (EP) [Fogel *et al.*, 1966] ont été définies pour l'optimisation de fonctions sur les réels. Il serait intéressant de s'inspirer de ces méthodes pour l'adaptation de `GWW` et de `GWW-idw` (en particulier pour leurs opérateurs de mutation qui correspondent à un pas de recherche dans un voisinage).

Enfin, plus généralement, il faudrait étudier l'adaptation au continu des méthodes hybrides mêlant recherche locale et recherche arborescente, définies pour l'optimisation combinatoire, comme par exemple Large Neighborhood Search (LNS) [Shaw, 1998], où de grands voisinages sont explorés par des méthodes arborescentes.

4.2.2 Décomposition

Notre méthode de résolution de systèmes décomposés présentée en 3.2.2 et décrite en détail dans le chapitre 9 [Jermann *et al.*, 2004b] comporte un "point faible" dans sa phase de résolution : l'heuristique du point milieu. En effet, chaque intervalle autour d'une valeur solution d'une variable d'un bloc est remplacé pour les résolutions des blocs en aval par le point milieu de l'intervalle. D'autre part, il arrive que dans certains cas, la précision du résolveur soit insuffisante pour isoler des solutions. Dans ce cas, la résolution de chaque bloc peut renvoyer un grand nombre de boîtes alors que quelques-unes seulement contiennent une solution. Il peut alors se produire une explosion combinatoire de sous-systèmes à résoudre. Pour pallier ce problème, nous ne gardons pas les boîtes solutions à petite distance d'une solution déjà trouvée. Cette heuristique, combinée à la précédente peut faire perdre des solutions. Il faudrait laisser les intervalles solutions et dans le cas de solutions proches, les remplacer par un intervalle englobant. Cela suppose un résolveur sachant manipuler des intervalles constants, ce qui n'est pas le cas d'IlogSolver, le résolveur actuellement utilisé. Du coup, les blocs ne seraient plus carrés et on perdrait le test d'unicité de solution et les tests d'arrêt pour isoler les solutions tels qu'ils existent actuellement. Il faudrait définir un nouveau test d'arrêt pour les équations paramétrées ou permettre de réduire les intervalles des variables déjà résolues dans les blocs en amont.

Nous pensons donc revoir en profondeur cet algorithme et en réaliser ensuite une nouvelle implantation dans le nouveau résolveur de contraintes par intervalles en cours de spécification dans le projet COPRIN.

Il faudrait aussi pouvoir tester cette méthode sur des problèmes de contraintes décomposables réalistes, qui semblent exister en CAO ou en chimie moléculaire.

4.2.3 Traitement des incertitudes

Nous avons étudié comment on pouvait gérer les incertitudes dans les paramètres d'un système d'équations de distance. Quand le système sans incertitudes a un nombre fini de solutions ponctuelles, le fait de prendre en compte des incertitudes dans les équations implique que les solutions deviennent des régions continues.

En partant des solutions ponctuelles du système sans les incertitudes, nous avons cherché à isoler les ensembles continus de solutions se trouvant autour de chacune des solutions ponctuelles. Nous avons pour cela réalisé [Grandon, 2003] un algorithme dynamique de bissection des domaines des variables de manière à ce que chaque solution appartienne à un seul sous-domaine et avons appliqué des techniques de consistance sur domaine continu (intervalles) pour obtenir des boîtes intérieures (i-consistance [Collavizza *et al.*, 1999]) et extérieures des solutions du système avec incertitudes, sous l'hypothèse que ces domaines-solutions soient sans intersection. Il s'agit maintenant d'étudier d'autres types d'équations plus générales, de détecter les limites de cette approche, de la comparer avec d'autres approches plus générales qui essaient d'englober des sous-ensembles de solutions continus [Vu *et al.*, 2002; 2004].

Cette problématique de traitement des incertitudes qui se pose dès que l'on prend en considération les erreurs de mesure, a aussi de fortes implications dans les applications de robotique comme l'étalonnage de robot qui sont étudiées dans le projet COPRIN.

4.3 Conclusion

Le domaine des contraintes s'est révélé ces dernières années être un domaine de recherche très actif, avec des liaisons importantes qui se sont développées avec d'autres domaines comme la recherche opérationnelle (RO) pour ce qui concerne l'optimisation combinatoire et l'analyse par intervalles pour la résolution de problèmes de satisfaction de contraintes numériques. Il s'appuie à la fois sur des équipes de recherche et des sociétés commerciales (Ilog, Cosytec) qui vendent des logiciels utilisant les résultats des recherches et permettent ainsi à de nombreuses applications industrielles l'usage de ces méthodes.

Depuis 1995, le congrès annuel CP (Constraint Programming) regroupe la communauté de recherche dans le domaine des contraintes. En plus de ce congrès spécialisé, des sessions dédiées aux contraintes ont lieu dans les principaux congrès d'intelligence artificielle (IA) : IJCAI, AAAI et ECAI. Le nouveau congrès CP-AI-OR sur l'intégration de techniques d'IA et de RO dans la programmation par contraintes pour des problèmes d'optimisation combinatoire, faisant suite en 2004 au colloque du même nom, témoigne de l'interaction croissante entre la propagation par contraintes et les méthodes de la recherche opérationnelle [Milano, 2003].

Le projet européen COCONUT et son colloque COCOS montrent le développement des méthodes utilisant la propagation de contraintes pour l'optimisation globale dans les domaines continus [Neumaier, 2004].

De nombreuses voies de recherche restent donc à explorer pour les prochaines années, en particulier pour la résolution de systèmes de contraintes dans les domaines continus.

Chapitre 5

La recherche à focalisation progressive

Version française de l'article paru dans les actes de ECAI 2002 sous le titre Progressive Focusing Search

Auteurs : Nicolas Prcovic, Bertrand Neveu

Résumé

Cet article traite des heuristiques de choix de valeur utilisées dans un algorithme complet de recherche arborescente pour la résolution de problèmes de satisfaction de contraintes binaires. Leur but est de guider la recherche vers une solution. Tout d'abord, nous montrons les limites des approches prospectives traditionnelles, qui utilisent la taille des domaines des variables pas encore affectées. Dans un contexte avantageux, quand la cohérence d'arc est maintenue et que le temps de calcul pour cette heuristique est négligeable, l'accélération est faible quand les problèmes sont difficiles. Nous présentons alors une nouvelle heuristique basée sur un schéma d'apprentissage par l'échec. Au lieu de faire un choix a priori, une recherche alternée est menée dans chaque sous-arbre pour accumuler de l'information. Après cette phase d'apprentissage, l'algorithme se focalise sur le sous-arbre le plus prometteur. Nous avons enfin intégré cette phase d'apprentissage dans la recherche elle-même et conçu un nouvel algorithme, appelé Recherche à focalisation progressive (PFS). Cet algorithme est comparé avec la recherche entrelacée IDFS et est apparu efficace pour des problèmes au pic de complexité de la transition de phase.

5.1 Introduction

Pour résoudre un problème de satisfaction de contraintes (CSP) en domaines finis avec une méthode complète, on utilise généralement un algorithme de recherche arborescente en profondeur d'abord (DFS). Plusieurs améliorations du retour arrière chronologique ont été proposées pour limiter l'explosion combinatoire. Citons par exemple le filtrage des domaines par propagation des contraintes, les retours arrière avec saut (backjumping) et l'utilisation d'heuristiques de choix de variables.

Pour les plus grands espaces de recherche explorables par une méthode complète, les résultats suivants sont généralement admis pour les CSP binaires. Le meilleur niveau de filtrage est le maintien de cohérence d'arc(MAC)[1] [Sabin et Freuder, 1994], les retours-arrière avec sauts ne sont pas utiles [Bessière et Régin, 1996] et, en l'absence de connaissances spécifiques sur le problème à résoudre, les heuristiques de choix de variable générales comme *min-domain+degree* ou *min-domain/degree* sont aujourd'hui reconnues être les plus efficaces. Pour l'obtention d'une seule solution, les heuristiques de choix de valeur peuvent aussi être utilisées, mais elles sont considérées moins importantes pour réduire l'arbre de recherche.

Nous présentons tout d'abord dans la partie 5.2 quelques heuristiques de choix de valeur existantes. Jusqu'à présent, l'idée principale de ces heuristiques est d'étudier les effets du choix d'une valeur dans une approche prospective, comme le fait l'heuristique des solutions potentielles (MP) [Geelen, 1992] Les effets de l'instanciation de la valeur courante avec chaque valeur sont partiellement propagés, avant de choisir la valeur la plus prometteuse. Nous proposons une autre approche basée sur un schéma d'apprentissage par l'échec, utilisant de l'information accumulée pendant la recherche. Dans la partie 5.3, nous montrons les limites de l'heuristique de choix de valeur MP. Les gains dus à cette heuristique sont importants quand le problème a beaucoup de solutions, mais deviennent négligeables au pic de complexité. La raison en est la suivante : l'heuristique n'a qu'une connaissance très partielle au début de la recherche et peut faire des choix désastreux.

Nous proposons dans la partie 5.4 une nouvelle sorte d'heuristique pour les premiers choix. Nous explorons une sous-partie de chaque sous-arbre issu de l'instanciation des premières variables avec chacune de leurs valeurs. L'information ainsi accumulée pendant le début de la recherche dans chaque sous-arbre est alors utilisée pour choisir le plus prometteur à explorer entièrement. Nous définissons un nouvel algorithme, nommé Recherche à focalisation progressive, qui implante cette idée. Nous montrons comment intégrer l'accumulation d'information dans la recherche elle-même et comment l'algorithme alterne des recherches dans tous les sous-arbres au début et finit par se focaliser sur le sous-arbre le plus prometteur. Dans la partie 5.5, nous le comparons avec l'algorithme de recherche en profondeur entrelacée (IDFS) [Meseguer, 1997] sur des CSP aléatoires.

5.2 Travaux antérieurs sur les heuristiques de choix de valeur

Dans une recherche arborescente standard, deux choix doivent être faits à chaque étape. Tout d'abord, on choisit la prochaine variable à instancier. Le problème est alors décomposé en k sous-problèmes, un pour chaque valeur du domaine de cette variable. Le choix de valeur détermine quel sous arbre sera exploré en premier par DFS.

Les effets de ces deux choix sur la recherche sont complémentaires. Le choix de variable a un effet sur l'arbre de recherche lui-même et le choix de valeur sur les branches de l'arbre qui seront explorées. Le but principal de l'utilisation d'une heuristique de choix de variable est de construire un arbre le plus petit possible, c'est-à-dire de limiter le

[1] Dans cet article, quand nous nous référons à MAC, il s'agit seulement du niveau de filtrage (cohérence d'arc) effectué à chaque affectation de variable et non du caractère binaire de l'arbre de recherche développé par cet algorithme.

nombre de branches et d'avoir les branches conduisant à des échecs les plus courtes possibles [Smith et Grant, 1998]. Les heuristiques hybrides *min-domain+degree* et *min-domain/degree* se basant sur la taille des domaines et le degré des variables jouent sur ces deux effets.

Le but d'une heuristique de choix de valeur est d'éviter de faire un mauvais choix, c'est-à-dire de sélectionner un sous-arbre sans solution. Elles utilisent pour cela une information partielle qui différencie les valeurs. Cette information peut être calculée une fois pour toutes avant la recherche ou dynamiquement au cours de la recherche.

Quand l'heuristique est dynamique, la principale information utilisée est la taille des domaines des variables futures, c'est-à-dire non encore instanciées. Ces tailles varient dynamiquement quand l'algorithme de recherche comme MAC [2] [Sabin et Freuder, 1994]. Cette information peut être utilisée de diverses manières.

– LVO-MC de Frost et Dechter [Frost et Dechter, 1995] : la valeur sélectionnée est celle qui a la plus grande somme des nombres de valeurs compatibles dans tous les domaines des variables futures.
– plus grand support de Larrosa et Meseguer (highest support) [Larrosa et Meseguer, 1995] : somme des tailles normalisées des domaines des variables futures en prenant en compte les incohérences.
– plus grande taille de domaine (max-domain-size) [Frost, 1997] : les valeurs sont ordonnées dans l'ordre décroissant de la taille du plus petit domaine des futures variables.
– ES estimation du nombre de solutions [Dechter et Pearl, 1988] : le nombre de solutions potentielles est estimé en résolvant une relaxation arborescente du graphe de contraintes restant.
– MP Solutions potentielles (max-promises) de Geelen [Geelen, 1992]. C'est un cas simple de l'heuristique précédente : la relaxation du problème restant est le problème formé des contraintes reliant la variable courante aux variables futures. La valeur choisie est celle pour laquelle le produit des tailles de domaines compatibles est maximum.

Pour calculer cette information sur la taille des domaines des variables futures, il faut propager l'effet de l'instanciation de la variable courante avec toutes les valeurs de son domaine pour pouvoir choisir la meilleure valeur selon le critère de l'heuristique. En d'autres termes, il faut effectuer à l'avance la propagation du Forward Checking dans des branches qui ne seront peut être pas explorées par la suite et cela peut être coûteux.

Pour pallier cette difficulté, quelques heuristiques statiques ont été définies.

– static least-conflicts (SLC) [Frost, 1997] : les valeurs sont ordonnées selon le nombre de conflits auxquels elles prennent part. Ces calculs sont effectués sur les domaines initiaux et cet ordre statique est utilisé pendant toute la recherche.
– static max-promises (SMP) : de la même manière, pour chaque variable v, on calcule avant la recherche pour chaque valeur dans le domaine de v, le produit des nombres de valeurs compatibles dans les domaines des autres variables et on ordonne le domaine de v selon ces produits.
– champ moyen (mean field) : un prétraitement basé sur la théorie du champ moyen ordonne les domaines avant la recherche [Cabon *et al.*, 1996].

La plupart des résultats publiés utilisent pour la résolution l'algorithme du Forward

[2]Dans cet article, quand nous nous référons à l'algorithme MAC, il s'agit seulement du degré de filtrage réalisé (cohérence d'arc) et non de la caractéristique d'arbre binaire de l'algorithme.

Checking [Haralick et Elliott, 1980]. Ils montrent qu'une heuristique dynamique a un surcoût important et ne devient bénéfique que quand le problème est suffisamment difficile (nécessitant plus d'un million de tests de contraintes dans [Frost et Dechter, 1995]).

Mais, aujourd'hui, les problèmes résolus le sont avec un algorithme maintenant un niveau de cohérence comme la cohérence d'arc et l'on peut espérer que le surcoût effectué à chaque nœud pour calculer une heuristique dynamique devienne négligeable.

5.2.1 Justification de l'heuristique des solutions potentielles

Les diverses expérimentations effectuées jusqu'à présent n'ont pas mis en évidence la supériorité d'une heuristique par rapport aux autres. L'ensemble de ces heuristiques donnent des résultats comparables. Parmi ces heuristiques, nous avons décidé d'étudier plus en détail le comportement de l'heuristique des solutions potentielles MP, qui a un fondement théorique.

Cette heuristique donne le nombre de solutions dans un sous-arbre, s'il n'y avait pas de contraintes entre les variables futures. En absence d'information sur ces contraintes, on peut supposer que ce nombre est proportionnel à une estimation du nombre de solutions dans un sous-arbre. En effet, dans le modèle de CSP aléatoires que nous avons utilisé dans nos tests (voir ci-après), les paires de valeurs interdites par une contrainte sont indépendantes des paires interdites par une autre contrainte. Ainsi, si nous considérons un algorithme comme le Forward Checking, où il n'y a pas de propagation due aux contraintes entre les variables futures, chaque paire de valeurs restant dans les domaines des variables futures a la même probabilité de satisfaire toutes les contraintes entre ces variables.

Nous avons montré dans une étude précédente [Prcovic et Neveu, 1999a] que le comportement de cette heuristique dynamique s'améliorait quand la profondeur de recherche augmentait. Elle est plus informée et se trompe moins souvent. Dans la partie suivante, nous allons étudier plus en détail son efficacité.

Nous avons aussi décidé d'utiliser MAC : l'information sur les tailles des domaines des variables futures est alors plus pertinente qu'avec FC et la qualité de l'heuristique MP devrait être meilleure.

5.3 Effets de l'heuristique des solutions potentielles

Pour étudier les effets de l'heuristique de choix de valeur des solutions potentielles, nous avons effectué quelques expérimentations sur des CSP aléatoires binaires. Nous nous sommes restreints à des tests dans des zones où les CSP ont des solutions et où les problèmes ne sont pas trop faciles (un nombre significatif de retours-arrière se produit). Quand MAC résout sans heuristique un problème sans aucun retour-arrière, il n'y a évidemment aucun besoin d'heuristique de choix de valeur. Une heuristique est aussi inutile pour les problèmes sans solution. Frost avait remarqué dans [Frost et Dechter, 1995] que l'utilisation d'une heuristique de choix de valeur pouvait avoir un effet quand on effectue des sauts avec CBJ, mais comme il a déjà été vu dans [Bessière et Régin, 1996], de tels sauts n'arrivent en pratique jamais avec MAC.

5.3.1 Résultats expérimentaux sur des CSP aléatoires

Nous avons réalisé des tests en utilisant la CSP-LIB de Hulubei [Hulubei, 1999], une bibliothèque C++ pour la résolution de CSP binaires en domaines finis. Nous avons implanté les heuristiques de choix de valeur des solutions potentielles statique (SMP) et dynamique (MP). Pour étudier l'importance du premier choix, nous avons aussi réalisé des tests où l'heuristique n'était utilisée que pour ce premier choix, tous les autres choix étant effectués selon l'ordre naturel des domaines. Nous avons appelé H1 cette heuristique du premier choix dans le tableau présentant les résultats et HO l'algorithme sans heuristique de choix de valeur.

Les tests ont été effectués sur des problèmes aléatoires, en utilisant le générateur de problèmes aléatoires de la CSP-LIB. Une classe de problèmes est définie par 4 paramètres :

- le nombre de variables,
- la taille des domaines,
- le nombre de contraintes, déterminé par le paramètre densité D,
- le nombre de paires de valeurs interdites par contrainte, déterminé par le paramètre dureté T.

Les tests ont été effectués avec des problèmes ayant 50 variables, des domaines de taille 15. Nous avons fait varier les paramètres de densité et de dureté pour obtenir des problèmes difficiles, mais qui pouvaient être résolus en un temps raisonnable (au maximum quelques minutes par instance de problème). Pour chaque ensemble de paramètres, nous avons réalisé des tests avec les heuristiques H1, SMP et MP, et les avons comparés avec des tests sans heuristique(H0). Quant à l'heuristique de choix de variable, nous avons utilisé *min-domain/degree* qui sélectionne la variable ayant le plus petit rapport taille des domaines sur degré (nombre de contraintes entre cette variable et les variables futures). Le niveau de filtrage, la cohérence d'arc, a été réalisé avec l'algorithme AC3, fourni par la bibliothèque.

La partie gauche du tableau 5.1 présente la profondeur du plus haut retour-arrière apparu sur un ensemble de 100 problèmes sans heuristique(H0), avec les heuristiques de premier niveau H1, statique SMP et dynamique MP. Un résultat $i(k)$ signifie que les i-1 premiers choix ont été de bons choix pour les 100 instances de problème, il n'a pas été nécessaire de revenir sur eux pour trouver une solution et que k problèmes sur 100 ont dû revenir à ce niveau i. Cela peut être vu comme une sorte de mesure de la distance de la classe de problèmes considérée au pic de complexité correspondant à la zone de transition de phase entre problèmes avec et sans solutions [Smith et Dyer, 1996]. Quand la profondeur i pour l'algorithme H0 est assez grande, cela signifie que pour toutes les instances, bien qu'il ne fussent pas guidés par une heuristique de choix de valeur, ces $i-1$ premiers choix n'ont jamais causé d'erreur. On est donc alors dans une classe de problèmes avec de nombreuses solutions. Quand, au contraire, des retours-arrière remontent jusqu'au premier niveau pour certaines instances, le premier choix a conduit à un sous-arbre sans solution et on se trouve dans une classe de problèmes proche du pic de complexité.

La partie droite du tableau 5.1 présente le temps cpu utilisé sur un Pentium III 500 pour la résolution des 100 instances de la classe traitée.

Nous pouvons voir que l'heuristique MP est très efficace pour des problèmes assez éloignés du pic de complexité. Le gain en temps obtenu est un facteur entre 4 et 8. Le faible surcoût dû au calcul de l'heuristique est largement compensé par la diminution du nombre de nœuds générés : la version dynamique de l'heuristique est presque toujours

TAB. 5.1 – Profondeur du plus haut retour-arrière (partie gauche), avec le nombre de problèmes à cette profondeur, et temps cpu en mn utilisé (partie droite) pour résoudre 100 instances de problèmes ayant 50 variables, des domaines de taille 15, une densité D et une dureté T

D,T	H0	H1	SMP	MP	H0	H1	SMP	MP
.89,.10	4(3)	4(1)	7(7)	7(3)	1143	996	147	149
.75,.12	4(9)	4(4)	6(9)	6(1)	1140	980	196	156
.52,.16	4(2)	4(3)	6(7)	7(9)	133	130	39	21
.41,.20	2(2)	3(5)	4(1)	5(11)	323	206	61	57
.40,.20	4(16)	4(3)	5(2)	5(1)	72	39	13	11
.32,.25	2(20)	2(2)	2(1)	3(6)	337	187	98	83
.31,.25	2(2)	3(5)	4(4)	4(3)	66	42	17	10
.30,.25	4(5)	4(3)	5(1)	6(2)	10	6	2	2

meilleure que la statique. Nous pouvons aussi remarquer en analysant les résultats de H1 qu'une part significative des gains est due au premier choix, même si ce choix n'a pas été remis en cause durant la recherche.

5.3.2 Inefficacité au pic de complexité

Une seconde série de tests a été réalisée plus près du pic de complexité avec des domaines de taille 8 et 15. Alors, le facteur de gain dû à l'utilisation de l'heuristique MP est tombé entre 1.4 et 1.1 (voir par exemple les classes de problèmes S=15, D=0.20, T=0.36 et S=8, D=0.44, T=0.15). Comme nous l'avons déjà remarqué, le sous-arbre ayant le plus grand nombre de solutions potentielles a la plus forte probabilité de contenir des solutions. Mais, ce sous-arbre a aussi la probabilité d'avoir la plus grande taille et quand l'heuristique se trompe, le dommage est plus important.

En restant dans la zone des problèmes avec solutions et en s'approchant du pic de complexité, une telle erreur arrive plus haut dans l'arbre et peut avoir des effets désastreux. Ceci peut expliquer l'inefficacité de l'heuristique près du pic, bien que les cas d'erreurs soient moins nombreux que sans heuristique. En analysant en détail les résultats près du pic, par exemple pour une taille des domaines de 15, une densité de 0.20 et une dureté de 0.36, l'heuristique se trompe 18 fois pour l'affectation de la première variable, alors que sans heuristique, une telle erreur au premier niveau de la recherche intervient 50 fois sur l'ensemble des 100 instances de problèmes de cette classe. Mais le temps le plus long pour résoudre une instance est de 14mn avec heuristique et de 2 mn sans heuristique. On peut aussi remarquer que ce premier choix est critique pour le temps global de résolution des 100 instances. Même si l'heuristique guide la recherche vers une solution dans un sous-arbre qui en contient, elle n'a aucun effet sur les sous-arbres sans solutions. Comme ces sous-arbres, une fois choisis, doivent être explorés entièrement, le temps passé dans leur exploration est la majeure partie du temps total.

Dans les problèmes suffisamment éloignés du pic de complexité dans la zone des problèmes avec solution, il y a toujours eu dans l'ensemble des 100 problèmes traités une différence de niveau pour le plus haut retour-arrière intervenu avec et sans heuristique.

TAB. 5.2 – Résultats au pic de complexité pour des problèmes avec 50 variables, une taille des domaines S, une densité D et une dureté T.

S,D,T	H0	H1,SMP,MP	H0	H1	SMP	MP
15,.25,.31	1(38)	1(16)	592	512	407	502
15,.20,.36	1(50)	1(18)	140	124	126	121
15,.14,.44	1(44)	1(12)	18	12	10	8
8,.50,.13	1(51)	1(26)	149	109	107	108
8,.44,.15	1(49)	1(27)	97	88	82	87
8,.435,.15	1(30)	1(9)	46	35	31	31
8,.315,.20	1(39)	1(18)	16	15	14	14

Ceci peut expliquer l'efficacité de l'heuristique dans ce cas. L'heuristique dynamique, qui est plus informée par les choix précédents dans la branche courante est alors plus efficace.

Nous avons vu au contraire que l'heuristique dynamique des solutions potentielles n'était pas suffisante pour améliorer substantiellement les résultats dans la zone de complexité : elle fait trop souvent un choix désastreux au premier niveau. Comme les choix cruciaux sont les premiers et comme l'heuristique ne peut pas bénéficier d'une information descendante par la branche courante, une idée pour l'améliorer est de remonter de l'information des sous-arbres, avant de choisir celui à explorer en premier. Dans la partie qui suit, nous allons présenter l'algorithme de recherche à focalisation progressive, qui exploite cette idée.

5.4 Ordonner les valeurs par apprentissage

Notre but est de trouver une meilleure heuristique de choix de valeur que MP pour les premières variables en haut de l'arbre de recherche. Pour ces variables, nous avons substitué l'approche prospective par une approche d'apprentissage par l'échec.

Nous avons pour cela défini une mesure statistique, calculée sur un échantillon de chaque sous arbre à profondeur d, et basée sur la longueur moyenne des branches (ABL), c.à.d la longueur entre le sommet du sous-arbre et un échec.

Nous allons tout d'abord présenter une méthode où cette statistique est précalculée, puis l'algorithme de recherche à focalisation progressive (PFS) qui met à jour cette mesure pendant la recherche et change de sous-arbre si la mesure ABL du sous-arbre courant devient mauvaise.

5.4.1 La mesure ABL

Nous sommes partis d'une idée simple :

1. effectuer une recherche DFS sur un sous-arbre à partir de la profondeur d et s'arrêter quand un nombre donné b de feuilles a été exploré, puis effectuer cette recherche sur un autre sous-arbre et ainsi de suite jusqu'à ce que tous les sous-arbres à profondeur d aient été ainsi partiellement explorés.
2. extraire une information empirique de l'échantillon de b branches de chaque sous-arbre et ordonner ces sous-arbres.

3. terminer la recherche DFS sur le reste du meilleur sous-arbre, puis sur le deuxième et ainsi de suite jusqu'à ce qu'une solution soit trouvée ou que l'arbre soit entièrement exploré

Il semble raisonnable de penser que le sous-arbre qui a la plus grande moyenne des branches contiendra le plus grand nombre de solutions. Appelons $D_{i,j}$, la longueur de la $j^{\text{ème}}$ branche du $i^{\text{ème}}$ sous-arbre. Nous pouvons calculer la longueur moyenne $D_i(b) = \frac{\sum_{j=1}^{j=b} D_{i,j}}{b}$ sur l'échantillon des b branches explorées dans le $i^{\text{ème}}$ sous-arbre et ordonner les sous-arbres selon ce critère $D_i(b)$, appelé ABL.

Une question importante demeure : comment fixer la valeur b ? Le critère ABL peut nous permettre d'obtenir un meilleur ordre si l'échantillon est assez grand. Si l'échantillon est trop petit, le résultat peut être moins bon que l'ordre donné par MP ; si l'échantillon est trop grand, le temps passé à trouver cet ordre peut être supérieur au temps gagné par l'heuristique. Pour éviter d'avoir à choisir une valeur b, nous proposons d'intégrer la phase d'échantillonnage dans la recherche elle-même, ce qui donne l'algorithme PFS décrit ci-dessous.

5.4.2 L'algorithme PFS

L'idée est de mettre à jour la mesure ABL pendant la recherche. Le nouveau critère est alors $D_i(b_i)$, la longueur moyenne des b_i branches explorées dans le sous-arbre i avant et pendant la recherche. Le sous-arbre i tel que $D_i(b_i) = max_j(\{D_j(b_j)\})$ est exploré jusqu'à ce que $D_i(b_i)$ devienne inférieur à un autre $D_j(b_j)$ ou que le sous-arbre i soit complètement exploré. Puis, la recherche continue dans le sous-arbre contenant le plus grand $D_i(b_i)$.

L'avantage de continuer à mettre à jour $D_i(b_i)$ pendant la recherche est de permettre à l'algorithme de “changer d'avis” s'il s'aperçoit que le sous-arbre qu'il a choisi s'avère moins prometteur qu'il le semblait auparavant. Quand b_i croît, chaque $D_i(b_i)$ devient de plus en plus stable et la recherche finit par rester coincée dans un sous-arbre jusqu'à son exploration complète.

Ce mécanisme permet d'éliminer la phase d'échantillonnage, laquelle n'a plus besoin d'atteindre un ordre des valeurs de qualité suffisante avant de commencer la recherche. En effet, l'échantillonnage est maintenant intégré dans la recherche. Une branche est explorée dans chaque sous-arbre et la recherche débute dans le sous-arbre qui a la première branche la plus longue. Ensuite, la recherche peut passer à un autre sous-arbre si la longueur moyenne des branches du sous-arbre courant devient plus petite que celle d'un autre sous-arbre. Cependant, il peut se passer le phénomène suivant : la première branche d'un sous-arbre peut être très courte et pas représentative. Dans ce cas, ce sous-arbre risque d'être définitivement mis de côté. Pour résoudre cette difficulté, nous proposons une mesure biaisée qui va forcer l'algorithme à visiter plusieurs branches dans chaque sous-arbre.

La version finale de l'algorithme est la même que celle que nous venons de présenter sauf que, au lieu de prendre comme mesure $D_i(b_i)$, nous prenons une nouvelle mesure pour changer de sous-arbre : $E_i(b_i) = \frac{K.n+\sum_{j=1}^{j=b_i} D_{i,j}}{K+b_i}$, où K est un paramètre constant et n est le nombre de variables dans le CSP. $E_i(b_i)$ est une moyenne biaisée qui calcule la moyenne comme si une phase de prétaitement avait exploré K branches de longueur n. Cet algorithme, appelé Recherche à focalisation progressive (PFS), est présenté en 5.1.

PFS commence la recherche en alternant entre tous les sous-arbres commençant à profondeur d, ordonnés dans une file de prorité Q selon leur valeur E_i. Durant la recherche, comme la mesure baisée converge vers la mesure ABL, l'algorithme va progressivement se focaliser sur le sous-arbre le plus prometteur : les sous-arbres avec la meilleure moyenne $D_i(b_i)$ (ABL) seront explorés de plus en plus souvent, jusqu'à que l'algorithme rejoigne le mécanisme précédent, où la recherche est maintenue dans un sous-arbre tant que celui-ci a la plus grande valeur $D_i(b_i)$.

```
PFS(d) :
Insérer les noeuds de profondeur d dans la file de priorité Q
s ← enlève_max(Q)
TANTQUE sommet(pile[s]) n'est pas une solution
   noeud ← dépile(pile[s])
   SI noeud = Echec
      mettre à jour E_s
      insérer(s, Q)
      s ← enlève_max(Q)
   SINON
      empile(les_fils(noeud), pile[s])
   SI pile[s] = ∅
      SI Q = ∅
         retourne Echec
      SINON
         s ← enlève_max(Q)
retourne sommet(pile[s])
```

FIG. 5.1 – Recherche à focalisation progressive. Les sous-arbres à la profondeur d sont ordonnés dans la file de priorité Q en fonction de leur valeur E_s. *les_fils(noeud)* retourne la liste des fils de *noeud*.

Voici les propriétés de la mesure $E_i(b_i)$ et leurs conséquences sur la recherche arborescente :

- Comme $\forall D_{i,b_i} : D_{i,b_i} \leq n$, nous avons $\forall b_i : E_i(b_i) > D_i(b_i)$.
  $E_i(0) = n$ et $E_i(b_i)$ converge vers $D_i(b_i)$ quand b_i croît et dépasse largement K. Cela signifie que l'exploration du sous-arbre i fait augmenter b_i et diminuer $E_i(b_i) - D_i(b_i)$. Donc, $E_i(b_i)$ a tendance à diminuer même si $D_i(b_i)$ reste constant. Ceci fait alterner la recherche entre les sous-arbres de manière artificielle.
- $E_i(b_i+1) = \frac{K.n+\sum_{j=1}^{j=b_i+1} D_{i,j}}{K+b_i+1} = \frac{K+b_i}{K+b_i+1} E_i(b_i) + \frac{D_{i,b_i+1}}{K+b_i+1}$.

  Ainsi, $E_i(b_i+1) < E_i(b_i)$ si et seulement si $E_i(b_i) > D_{i,b_i+1}$.
  En début de recherche, si $K \gg b_i$ alors $K.n \gg \sum_{j=1}^{j=b_i} D_j$ et $E_i(b_i) \simeq n > D_{i,b_i+1}$. Donc, $E_i(b_i)$ va diminuer presque toujours et on changera souvent de sous-arbre. Après un certain temps, quand $K \ll b_i$, nous aurons $K.n \ll \sum_{j=1}^{j=b_i} D_j$ et $E_i(b_i) \simeq D_i(b_i)$. Donc, $E_i(b_i)$ décroîtra si et seulement si $D_{i,b_i+1} < D_i(b_i)$. Cela signifie que $E_i(b_i)$ est plus susceptible de diminuer en début de recherche que plus tard. Ainsi, l'entrelacement sera progressivement atténué tout au long de la recherche.
- Si $K = 0$ alors $E_i(b_i) = D_i(b_i)$. Si $K \gg 1$ alors $E_i(b_i) \simeq n$. Le réglage de K permet

donc de changer la vitesse de convergence de $E_i(b_i)$ vers $D_i(b_i)$, c'est-à-dire, la vitesse de focalisation vers le sous-arbre présentant la plus grande valeur de $D_i(b_i)$.

5.4.3 Comparaison avec IDFS

PFS peut être comparé avec la recherche en profondeur d'abord entrelacée (en anglais :Interleaved Depth First Search, IDFS)[Meseguer, 1997]. IDFS a été défini afin de prendre en compte la qualité croissante de l'heuristique de choix de valeur en fonction de la profondeur. Son comportement peut être opposé à celui de DFS. Quand il rencontre un échec, DFS revient d'abord sur le *dernier* choix alors que IDFS revient sur le *premier* choix. La justification de ce comportement est simple : les choix effectués à de petites profondeur sont moins sûrs donc ce sont les plus profonds qui doivent en priorité être conservés. Il y a deux versions de IDFS. IDFS pur applique exactement le comportement que nous venons de présenter. L'arbre de recherche est exploré de telle manière que, quelle que soit la profondeur, aucun sous-arbre ne sera revisité avant que tous les autres de même profondeur aient été visités aussi. IDFS limité restreint l'entrelacement à une profondeur fixée et à un nombre limité de sous-arbres. Il simule exactement l'ordre d'examen des branches qui serait effectué par une recherche arborescente parallèle, la profondeur d'entrelacement et le nombre de sous-arbres actifs correspondant respectivement à la profondeur de distribution et au nombre de processus engagés. Dans le reste de l'article, lorsque nous nous référerons à IDFS, il s'agira implicitement de sa version limitée.

IDFS ne fait aucun choix jusqu'à ce que la profondeur d'entrelacement d soit atteinte. Chaque n-uplet de valeurs pour les d premières variables est ensuite essayé de façon entrelacée.

On notera que IDFS est plus efficace que DFS dans le contexte que nous étudions. Il a été montré, en théorie et en pratique [Meseguer, 1997; Meseguer et Walsh, 1998; Prcovic et Neveu, 1999a] que IDFS pouvait avoir de meilleures performances que DFS lorsque la qualité de l'heuristique de choix de valeur augmentait avec la profondeur, ce qui est le cas de l'heuristique dynamique des solutions potentielles (MP). Les résultats expérimentaux de la partie suivante le confirmeront.

Au début de la recherche, PFS se comporte en quelque sorte comme IDFS, dans le sens où il génère une branche dans chaque sous-arbre et n'en générera pas une autre dans le même avant qu'il ait visité tous les autres sous-arbres. Quand la recherche arborescente commence, chaque $E_i(0) = n$. Ainsi, le premier sous-arbre est sélectionné, une de ses branches est générée jusqu'à la profondeur $D_{1,1}$ (où un échec survient) et $E_1(1)$ prend une valeur inférieure à n. En conséquence, la recherche doit être continuée dans le second sous-arbre et le processus se répète jusqu'à ce qu'une branche ait été générée dans chacun des sous-arbres. Ensuite, le sous-arbre j ayant la plus grande valeur de $E_j(1)$ est sélectionné et $E_j(2)$ est d'autant plus susceptible d'être la plus petite valeur que K est grand.

Quand $K \gg b_i$, si $b_i > b_j$, $E_i(b_i)$ est souvent inférieur à $E_j(b_j)$.
Si $b_i = b_j$, $(E_i(b_i) > E_j(b_j)) \equiv (D_i(b_i) > D_j(b_j))$. Donc, PFS avec un grand K entrelacera autant la recherche que IDFS mais pas exactement dans le même ordre : un sous-arbre qui aura généré moins de branches que les autres aura plus tendance à être choisi avant ces derniers mais si lui et les autres ont généré le même nombre de branches alors c'est celui qui aura la plus grande valeur de $D_i(b_i)$ qui sera choisi. Une recherche PFS avec un K très grand peut être vue comme une recherche IDFS qui réordonne

les sous-arbres selon leur valeur de $D_i(b_i)$ après avoir généré une branche dans chacun d'eux. Cela entraîne que PFS avec un très grand K devrait générer un petit peu moins de branches et de nœuds que IDFS. Maintenant, nous allons voir pourquoi une valeur de K plus faible et une focalisation progressive peuvent conduire à de meilleures performances.

PFS peut être vu comme une recherche arborescente dont le comportement imite IDFS au début mais qui se focalise progressivement sur le sous-arbre le plus prometteur, imitant ainsi DFS après un certain temps. Si nous décidons d'utiliser l'heuristique de longueur moyenne des branches (ABL) pour les d premières variables et MP pour les $n - d$ variables restantes, alors :

- Au début de la recherche, la mesure ABL est pauvrement informée et incertaine donc la qualité de MP la dépasse. A ce stade de la recherche, il est donc fondé d'utiliser IDFS plutôt que DFS.
- Les estimations de ABL convergeant vers les vraies valeurs, il y aura un moment dans la recherche où la qualité de ABL dépassera celle de MP. Alors, la qualité de l'heuristique de choix de valeur MP utilisée à la profondeur $d + 1$ ne sera plus meilleure que celle, ABL, utilisée entre les profondeurs 1 et d. Il deviendra néfaste d'entrelacer la recherche à la profondeur d. Ainsi, l'utilisation de DFS donnera de meilleurs résultats à partir de ce moment là.

Au lieu d'un changement progressif de IDFS vers DFS, nous pourrions envisager un changement brusque dès que ABL deviendrait meilleur que MP. Malheureusement, nous n'avons pas d'élément nous permettant de savoir quand aura lieu ce basculement. C'est pourquoi un changement progressif est un moyen d'obtenir un algorithme stable. L'efficacité de PFS repose sur la vitesse de focalisation. En conséquence, il faudra prendre soin du réglage de K.

5.5 Résultats expérimentaux

Les expérimentations ont été faites en utilisant le même générateur de CSP aléatoires. Les problèmes ont été choisis de façon à rester à gauche du pic de complexité, où la plupart des instances ont des solutions. Pour chaque paramétrage, nous avons exécuté différents algorithmes sur les mêmes 100 instances de problèmes dont chacune avait au moins une solution. Tous les algorithmes maintenaient la cohérence d'arc, utilisaient l'heuristique de choix de variable dynamique *min-domain/degree* et l'heuristique de choix de valeur MP. Les algorithmes étaient : DFS, IDFS(d) et PFS$_K(d)$, où d est la profondeur d'entrelacement.

Notre but n'était pas de trouver les paramètres optimaux pour maximiser les gains de PFS sur DFS ou IDFS, mais de commencer à étudier comment ce gain varie quand on s'approche du pic de complexité ou quand d et K varient.

Nous avons commencé par tester si PFS pouvait être plus efficace sur des problèmes de même taille (50 variables, domaines de taille 8) comme dans la partie 5.3 à gauche du pic de complexité (voir tableau 5.3).

On peut remarquer que quand les problèmes deviennent difficiles, PFS est plus efficace et que l'entrelacement au niveau 2 est meilleur qu'au niveau 1. Pour voir ce qui se passerait si on augmentait encore l'entrelacement, nous avons continué les tests sur des problèmes avec $S = 4$ pour limiter le nombre de sous-arbres présents en mémoire. (voir tableau 5.4).

Nous avons aussi fait varier K. Quand K était grand ($K = 1000$), PFS$_d$ était toujours légèrement meilleur(de moins de 1%) que IDFS$_d$. Avec K petit ($K = 5$), nous avons pu

TAB. 5.3 – Expérimentations avec $N = 50$, $S = 8$, $D = 5/64$ et T variant. Nous avons toujours indiqué le nombre *total* de tests de contraintes (en millions) sur les 100 instances.

T	1000/1275	1020/1275	1040/1275	1060/1275
DFS	2060	2290	7872	21070
$IDFS_1$	**541**	894	3093	8720
$IDFS_2$	1233	1555	3808	8176
$PFS_1(5)$	707	**680**	**2246**	8721
$PFS_2(5)$	1216	1452	3487	**8029**

TAB. 5.4 – Variation de la dureté des problèmes et de la profondeur d'entrelacement. Expérimentations avec $N = 100$, $S = 4$ et différents (D, T). Pour le paramétrage (0.166, 0.125), deux problèmes étaient sans solution.

D,T	.158,.125	.162,.125	.166,.125	.327,.0625	.336,.0625
DFS	109	598	1306	384	2005
$IDFS_1$	102	509	1302	198	1230
$IDFS_2$	60.1	298	979	142	605
$IDFS_3$	73.9	308	718	197	597
$PFS_1(5)$	91.9	520	1483	**121**	1223
$PFS_2(5)$	**49.9**	**195**	973	137	505
$PFS_3(5)$	69.7	204	**597**	175	**444**

obtenir obtenir de meilleurs résultats que IDFS, mais le gain n'était pas très important. Par exemple, pour les problèmes dans la troisième colonne du tableau 5.4, les résultats varient entre 563 et 659 quand $1 \leq K \leq 10$.

On notera que :
- IDFS surpasse DFS sur des problèmes difficiles.
- Une profondeur d'entrelacement plus importante est meilleure pour les problèmes difficiles.
- PFS surpasse DFS (gains jusqu'à 78%) et IDFS (jusqu'à 35%) quand les problèmes sont difficiles.
- L'efficacité de PFS sur IDFS semble augmenter quand les problèmes deviennent plus difficiles.

Nous avons noté que les résultats de PFS avaient un plus petit écart type que ceux de DFS ou IDFS. Quand le problème est facilement résolu par DFS, IDFS et PFS produisent plus de nœuds. Quand le problème a pris beaucoup de temps pour être résolu par (de grands sous-arbres sans solutions étant explorés), PFS produit généralement beaucoup moins de nœuds.

Le fait que IDFS surpasse DFS sur les problèmes difficiles confirme l'intérêt d'éviter de choisir entre les valeurs possibles pour les premières variables. L'entrelacement est une bonne option quand la qualité de l'heuristique de choix de valeur augmente avec la profondeur. Les performances de PFS sur IDFS montrent qu'il y a un moment où se focaliser sur un sous-arbre devient meilleur que de continuer l'entrelacement.

Les gains de PFS sur IDFS ne sont pas très grands. Une raison peut être que les CSP aléatoires traités étaient réguliers et qu'il n'y avait de sous-arbre réellement meilleur sur lequel se focaliser pour accélérer la recherche.

Nous pensons que PFS pourrait montrer de meilleures performances sur des CSP moins équilibrés. Le critère de moyenne de longueur des branches était très simple : un autre critère plus pertinent pourrait peut être produire de meilleurs résultats.

5.6 Conclusion et perspectives

Nous avons montré l'inefficacité de l'approche prospective pour ordonner les valeurs dans les problèmes difficiles avec des solutions. Nous avons commencé à explorer les possibilités d'un schéma d'apprentissage par l'échec. Une simple mesure de la longueur moyenne des branches des sous-arbres a permis à PFS d'obtenir de meilleurs résultats que DFS ou IDFS. Ce critère ABL étant très simple, il y a sûrement d'autres critères à trouver pour améliorer les performances de PFS. Il serait aussi utile de trouver un moyen automatique de fixer le paramètre K à sa meilleure valeur. Deux approches semblent possibles :

1. précalculer K à partir des caractéristiques du CSP
2. ajuster K durant la recherche selon un processus à déterminer.

Nous pensons qu'une approche empirique pour (ré)ordonner les valeurs est prometteuse pour accélérer la recherche des problèmes difficiles et peut avantageusement remplacer les approches prospectives traditionnelles.

Chapitre 6

Bibliothèque INCOP

Version longue en français de l'article de présentation de la bibliothèque INCOP publié dans les actes de CP 2003.

Auteurs : Bertrand Neveu, Gilles Trombettoni

Résumé

Nous présentons une nouvelle bibliothèque, `INCOP`, d'algorithmes incomplets pour des problèmes d'optimisation combinatoire. Cette bibliothèque offre des méthodes de recherche locale classiques comme le recuit simulé, la recherche avec liste taboue, ainsi qu'une méthode à population, "Go with the winners". Différents problèmes ont été codés, incluant les problèmes de satisfaction de contraintes, le coloriage de graphes, l'affectation de fréquences.

`INCOP` est une bibliothèque C++ extensible. L'utilisateur peut facilement ajouter de nouveaux algorithmes et coder de nouveaux problèmes. La gestion du voisinage a été étudiée avec soin. D'abord, une sélection de mouvement originale et paramétrée permet d'implanter facilement la plupart des métaheuristiques existantes. Ensuite, différents niveaux d'incrémentalité peuvent être spécifiés pour le calcul du coût d'un mouvement, ce qui augmente grandement l'efficacité en terme de temps de calcul.

`INCOP` s'est montré très performant sur quelques bancs d'essais connus. Il surpasse la plupart des outils concurrents. L'instance difficile de coloriage de graphe `flat300_28` a été coloriée en 30 couleurs pour la première fois par un algorithme de Metropolis standard.

6.1 Introduction

Les problèmes d'optimisation discrets peuvent être résolus par deux principales classes de méthodes : les méthodes complètes et les méthodes incomplètes. Les algorithmes complets sont basés sur une recherche arborescente avec un schéma de "séparation - évaluation" (Branch and Bound). Plusieurs outils logiciels commerciaux proposent de telles méthodes. Quand un problème d'optimisation peut être modélisé par des contraintes et un critère linéaires, des logiciels de programmation linéaire en nombres entiers (PLNE) peuvent être utilisés. Autrement, des outils de programmation par contraintes, comme `IlogSolver` or `Chip`, peuvent traiter des contraintes quelconques.

Quand l'espace de recherche devient trop grand, ces techniques de recherche systématique sont souvent dépassées par les méthodes incomplètes stochastiques. Ces algorithmes incomplets ne peuvent prouver l'optimalité d'une solution, mais peuvent fournir rapidement une bonne solution. Les méthodes incomplètes les plus courantes sont à base de recherche locale : elles essaient de modifier localement *une* configuration pour améliorer son coût. D'autres méthodes incomplètes explorent l'espace de recherche en manipulant une population de configurations, c'est-à-dire *plusieurs* configuration à la fois.

Implanter efficacement des algorithmes complets requiert un grand effort. Il faut en effet propager les contraintes le plus possible pour réduire la combinatoire et chaque type de contraintes peut nécessiter le développement d'algorithmes spécifiques. Des logiciels commerciaux ont été réalisés et sont utilisés avec succès. Au contraire, il est facile d'implanter une méthode incomplète et aucun effort important n'a été réalisé pour construire un outil commercial. `IlogSolver` a récemment ajouté un module de recherche locale, mais ce dernier est inclus dans la bibliothèque entière et ne peut être utilisé séparément.

Ce manque d'outil a conduit récemment de nombreux chercheurs à construire leurs propres bibliothèques de méthodes incomplètes de recherche. Beaucoup de projets ont été présentés ces dernières années [Voß et Woodruff, 2002c]. Citons `iOpt` [Voudouris et Dorne, 2002] par British Telecom, `SCOOP` [Nielsen, 1998] par SINTEF, `Localizer++` [Michel et Van Hentenryck, 2001] à Brown University, `Hotframe` [Voß et Woodruff, 2002b] à l'université de Braunschweig, `Discropt` [Phan et Skiena, 2002] à State University of New York. Philippe Galinier et Jin-Kao Hao [Galinier et Hao, 1999] ont aussi proposé un cadre pour la recherche locale.

Cependant, pour le moment, toutes ces bibliothèques ne sont pas libres ou pas disponibles. Nous n'avons trouvé qu'une seule bibliothèque libre sur Internet : `EasyLocal++`, à l'Université d'Udine [DiGaspero et Schaerf, 2000], qui implante des méthodes de recherche locale.

Au départ, nous voulions tester une méthode à population et la comparer avec des méthodes de recherche locale dans la même implantation. En effet, une comparaison équitable ne peut être réalisé que sur une même plate-forme logicielle. Nous avons remarqué que l'implantation elle-même, les structures de données utilisées et le degré d'incrémentalité étaient très importants pour les performances. C'est pourquoi nous avons décidé de réaliser une bibliothèque libre, implantant les principales métaheuristiques de recherche locale et des méthodes à population efficaces.

6.1.1 Principales caractéristiques de la bibliothèque `INCOP`

Sans perte de généralité, notre bibliothèque ne traite que des problèmes de minimisation.

Pour le moment, seul un type de configuration est implanté : une configuration est représentée par un ensemble fixé de variables entières avec des domaines de valeurs connus a priori. De nombreux problèmes d'optimisation combinatoire peuvent être codés dans ce cadre. Il est immédiat de représenter les problèmes de voyageur de commerce (TSP) et de satisfaction de contraintes (CSP). Le coloriage de graphe et certains problèmes d'affectation de fréquences peuvent aussi être vus comme des CSP spécifiques. La bibliothèque étant extensible, d'autres types de configurations peuvent être ajoutés.

Contributions.

Puisque la recherche locale effectue des déplacements dans l'espace des configurations, nous devons proposer un composant pour les mouvements. Pour des raisons d'efficacité, `INCOP` permet aux algorithmes de calculer, lorsque la fonction d'évaluation est décomposable, de manière incrémentale le coût d'une configuration quand un mouvement est réalisé. De plus, `INCOP` possède les caractéristiques suivantes :

- Une façon originale et paramétrée pour sélectionner un voisin est proposé : c'est une sorte de stratégie de liste de candidats [Glover et Laguna, 1997].
- Une méthode à population (`GWW`) est implantée, avec une version hybridée avec de la recherche locale, qui la rend plus efficace (`GWW-idw`).
- Il est possible d'ajouter une nouvelle méthode et d'implanter un nouveau problème d'optimisation combinatoire.
- Les métaheuristiques de recherche locale les plus courantes sont implantées comme la descente, `GSAT` [Selman *et al.*, 1992], le recuit simulé [Kirkpatrick *et al.*, 1983], Metropolis [Connolly, 1990], la recherche taboue [Glover, 1986].

6.1.2 Plan

La partie 6.2 présente l'architecture de la bibliothèque programmée par objets. Cette partie montre aussi comment ajouter une nouvelle métaheuristique, un nouveau problème ou comment définir un nouveau voisinage. La partie 6.3 détaille les contributions de `INCOP` : les calculs incrémentaux des coûts des configurations, la sélection paramétrée des mouvements et les algorithmes à population implantés. Les résultats expérimentaux obtenus sur des instances de coloriage de graphe et d'affectation de fréquences sont présentés dans la partie 6.4 et mettent en valeur les bonnes performances de `INCOP`. Les travaux en rapport sont présentés dans la partie 6.5.

6.2 Architecture

Nous avons choisi une conception par objets et implanté la bibliothèque en C++, en utilisant les méthodes virtuelles et les structures de données fournies par la STL. Bien qu'aucun langage spécifique pour les algorithmes incomplets n'ait été développé (comme `OPL` [Michel et Van Hentenryck, 2001] ou `SALSA` [Laburthe et Caseau, 1998]), nous allons montrer qu'il n'est pas difficile de définir de nouvelles métaheuristiques ou de nouveaux problèmes.

`INCOP` comprend six classes principales :

- La classe `Configuration` spécifie le codage d'une solution.
- Spécialiser la classe `Move` permet de définir un nouveau voisinage pour les configurations.
- La classe `NeighborhoodSearch` permet à une configuration de chercher un voisin de façon paramétrée (cf partie 6.3.3).
- La spécialisation de la classe `IncompleteAlgorithm` définit un nouvel algorithme incomplet.
- La spécialisation de la classe `Metaheuristic` définit une nouvelle métaheuristique.
- La spécialisation de la classe `OpProblem` ajoute un nouveau problème.

On notera que les algorithmes et les problèmes sont séparés en deux hiérarchies de classe. Cela rend possible l'ajout d'un nouveau problème et sa résolution avec toutes les métaheuristiques déjà implantées, ou inversement, l'ajout d'un nouvel algorithme et son test avec tous les problèmes déjà implantés.

Signalons aussi qu'une gestion originale du voisinage est proposée dans `INCOP`. L'attribut `tabconflicts` des sous-classes de `Configuration`, et les méthodes de la classe `OpProblem` offrent un haut degré d'incrémentalité pour l'évaluation du coût d'un mouvement par les métaheuristiques. Les attributs de la classe `NeighborhoodSearch` permettent une sélection paramétrée d'un mouvement (cf partie 6.3.3).

6.2.1 Définir une configuration

Dans la version courante, la classe `Configuration` est seulement spécialisée pour représenter la solution d'un problème `MAX-CSP`, où l'on minimise le nombre (ou une somme pondérée) des violations de contraintes. La hiérarchie de classes courantes est la suivante :

```
Configuration
    CSPconfiguration
         IncrCSPconfiguration
         FullincrCSPconfiguration
```

Un objet de la classe `CSPconfiguration` a trois attributs principaux. La solution est codée dans l'attribut `config` comme un vecteur de valeurs, une pour chaque variable de l'instance de CSP. Un autre attribut `valuation` contient le coût de la configuration.

Un troisième attribut, appelé `tabconflicts`, est défini dans les sous-classes de `CSPconfiguration` pour stocker le nombre de conflits de la valeur courante ou de chaque valeur selon le degré d'incrémentalité choisi. La partie 6.3.1 détaille la manière dont l'attribut `conflicts` est mis à jour quand un mouvement est effectué.

6.2.2 Définir un nouveau voisinage

Le voisinage est défini en deux classes principales : `NeighborhoodSearch` et `Move`. Les objets de la classe `Move` contiennent l'information associée à la modification de la configuration (le mouvement) : un calcul du coût de ce mouvement permet le calcul incrémental du coût de la configuration modifiée par ce mouvement.

Pour ajouter un nouveau voisinage, il faut définir une sous-classe de cette classe. Deux sous-classes sont actuellement implantées. La sous-classe `SwapMove` définit un mouvement d'échange dans une configuration de type permutation : les valeurs de deux variables sont échangées. Un tel mouvement est défini par les 2 variables échangées `variable1` et `variable2`.

```
class SwapMove : public Move {
  public :
     int variable1;
     int variable2;
     SwapMove();
     ...
};
```

La sous-classe `CSPMove` réalise la manière la plus courante de définir un voisinage pour les CSP : deux voisins diffèrent par la valeur d'une variable.

Un objet de la classe `NeighborhoodSearch` peut aussi être utilisé pour affiner le voisinage d'un CSP. En se basant sur les travaux de Minton et al [Minton *et al.*, 1992], `INCOP` propose des variantes pour la définition du voisinage. Deux attributs booléens sont définis à cet effet.

- `var_conflict` : indicateur de la restriction aux variables participant à un conflit.
- `min_conflict` : indicateur de la restriction aux valeurs de la variable qui minimisent les conflits pour cette variable (heuristique Min-conflicts [Minton *et al.*, 1992]).

On pourrait aussi être plus sélectif dans la détermination des voisins possibles en choisissant des variables ayant une participation forte aux violations de contraintes ou au contraire moins sélectif et implanter des voisinages larges (comme dans la méthode LNS [Shaw, 1998]).

6.2.3 Algorithmes incomplets

`IncompleteAlgorithm` est la classe racine. Elle est subdivisée en `LSAlgorithm` et `GWWAlgorithm`.

```
IncompleteAlgorithm
   LSAlgorithm
   GWWAlgorithm
        ThresholdGWWAlgorithm
        NoThresholdGWWAlgorithm
```

La classe `LSAlgorithm` implante les algorithmes de recherche locale qui effectuent une marche pour une seule configuration. Cette classe a les trois attributs principaux suivants :

- la nombre de mouvements à effectuer,
- un pointeur `nbsearch` vers un objet de la classe `NeighborhoodSearch`, qui permet à l'algorithme de passer d'une configuration à une autre de façon paramétrée,
- un pointeur `mheur` vers un objet `Metaheuristic`.

Les métaheuristiques courantes sont implantées comme des sous-classes de la classe abstraite `Metaheuristic`, citons `TabuSearch`, `GreedySearch`, `SimulatedAnnealing`. D'autres peuvent être facilement ajoutées. Les sous-classes contiennent une méthode spécifique `acceptance` qui décide si un mouvement essayé est accepté et des structures de données spécifiques à la métaheuristique. Par exemple, la classe `TabuSearch` a un attribut pour stocker sa liste taboue.

La classe `GWWAlgorithm` est une spécificité importante de `INCOP` détaillée dans la partie 6.3.2. Elle implante des algorithmes qui traitent plusieurs configurations à la fois.

Ajouter une nouvelle métaheuristique

Pour implanter une nouvelle métaheuristique, il faut définir une sous-classe de `Metaheuristic`, avec ses données, et trois méthodes :

- `reinit` : initialisation des données de la métaheuristique,
- `acceptance` : condition d'acceptation d'un mouvement candidat,
- `executebeforemove` : mise à jour des données de la métaheuristique (comme la température du recuit simulé ou la liste taboue) avant d'exécuter un mouvement.

Pour illustrer cette partie, nous présentons le code de la métaheuristique d'acceptation à seuil (Threshold Accepting) [Dueck et Scheuer, 1990] : un mouvement est accepté si la détérioration de l'évaluation de la configuration courante est inférieure à un seuil ; ce seuil baisse linéairement de `thresholdinit` à 0 pendant la recherche (une marche de longueur `walklength`). A chaque mouvement, le seuil est décrémenté de `delta`.

```
class ThresholdAccepting: public Metaheuristic {
  public :
    double thresholdinit;
    double delta;
    double thresholdaccept;
    ThresholdAccepting(double maxthreshold, int walklength);
    int acceptance (Move * move, Configuration* config);
    void executebeforemove(Move*  move, Configuration * configuration,
                           OpProblem * problem);
    void reinit();
};

ThresholdAccepting::ThresholdAccepting
                 (double maxthreshhold,int walklength) {
  thresholdinit=maxthreshhold; delta=thresholdinit/walklength;}

int ThresholdAccepting::acceptance (Move * move, Configuration*  config) {
  return ((move->valuation - configvaluebefore) <  thresholdaccept);}

void ThresholdAccepting::reinit() {
   thresholdaccept = thresholdinit;}

void ThresholdAccepting::executebeforemove (Move * move,
 Configuration * configuration, OpProblem * problem) {
   if (thresholdaccept >= delta)  thresholdaccept -= delta; }
```

La métaheuristique peut alors être directement utilisée en remplissant l'attribut `mheur` de l'objet de classe LSAlgorithm.

```
algo->mheur = new ThresholdAccepting (100., algo->walklength);
```

Quelques ajouts sont aussi nécessaires dans l'interface pour intégrer cette métaheuristique dans la liste des méthodes connues au niveau de la ligne de commande.

6.2.4 La hiérarchie des problèmes

Les problèmes qui peuvent être implantés dans la bibliothèque sont des problèmes d'optimisation discrète comme, par exemple, le problème du voyageur de commerce. Les problèmes de satisfaction de contraintes (CSP) sont transformés en problèmes MAX-CSP d'optimisation pour lesquels on cherche à minimiser le nombre de contraintes non satisfaites(ou plus généralement un critère calculé sur ces violations de contraintes). Une solution est obtenue quand on a trouvé une configuration avec un coût égal à zéro. Nous avons implanté plusieurs CSP, incluant le coloriage de graphe et les affectations de fréquences.

```
OpProblem : problème d'optimisation
   CSProblem : problème de satisfaction de contraintes
      BinaryCSProblem : CSP binaire
         ColorCSProblem : Coloriage de graphe
         CelarCSProblem : Affectation de fréquences du CELAR (critère additif)
         ExtensionBinaryCSProblem : CSP en extension (traités comme MAX-CSP)
         CliqueProblem : Trouver une clique de taille donnée dans un graphe
         NQueenProblem   : problème des N reines
   TSProblem : problème du voyageur de commerce
```

Ajouter un nouveau problème

Si le problème a des contraintes binaires, il peut hériter de la structure `constraints` de la classe `BinaryCSProblem`. Si nécessaire, un voisinage spécifique peut être créé en spécialisant la classe `Move`. Dans ce cas, la méthode `Next_move` doit être redéfinie et remplir un tel mouvement.

Le critère à optimiser est spécifique à un problème donné. Trois méthodes de la classe `OpProblem` calculent le coût d'une configuration. La méthode `config_evaluation` évalue le coût d'une configuration initiale ; `move_evaluation` réalise l'évaluation incrémentale d'un mouvement ; les méthodes `incr_update_conflicts` ou `fullincr_update_conflicts` mettent à jour la structure de données des conflits d'une configuration quand un mouvement est exécuté, selon le degré d'incrémentalité choisi.

Par exemple, les instances de coloriage de graphe sont implantées par la classe `ColorCSProblem`. Nous utilisons une incrémentalité totale pour les évaluations des mouvements.[1].

```
class ColorCSProblem : public BinaryCSProblem {
  public :
    ColorCSProblem (int nvar, int nconst, int nbcol);
    int config_evaluation (CSPconfiguration * config);
    void fullincr_update_conflicts (FullincrCSPConfiguration *config,
                                   Move * move);
    void init_domains (int nbvar, int nbcol);
    Configuration* create_configuration();
};
```

Les instances de coloriage de graphe peuvent être codées comme des instances de MAX-CSP : les variables sont les sommets du graphe à colorier ; le nombre `nbcol` de couleurs avec lesquelles le graphe doit être colorié donne les domaines allant de 1 à `nbcol` ; les sommets liés par une arête doivent être coloriés avec des couleurs différentes : les variables correspondantes doivent prendre des valeurs différentes. Colorier un graphe en `nbcol` couleurs revient à minimiser le nombre de contraintes non satisfaites et à trouver une solution de coût 0. Cette modélisation nous permet de réutiliser une grande part du code défini dans la classe `BinaryCSProblem`.

[1] En plus, il faut définir une fonction qui lit les données du graphe à partir d'un fichier au format Dimacs, appelle le constructeur `ColorCSProblem`, et remplit la structure de données des contraintes.

Les mouvements sont aussi les mêmes que pour un CSP standard : changer la couleur d'un sommet revient à modifier la valeur d'une variable du CSP correspondant, La structure de données des conflits et la méthode `move_evaluation` sont aussi hérités de `BinaryCSProblem`.

Seules les méthodes `config_evaluation` et `fullincr_update_conflicts` ne sont pas immédiates à écrire. Elles sont détaillées ci-après.

La méthode `create_configuration` spécifie quel degré d'incrémentalité est utilisé (ici le degré d'incrémentalité totale).

```
Configuration* ColorCSProblem::create_configuration()
{return (new FullincrCSPConfiguration (nbvar,domainsize ));}
```

La méthode `config_evaluation` calcule le coût d'une configuration et remplit la structure de données des conflits.

```
int ColorCSProblem::config_evaluation (Configuration* configuration)
{int value=0;
 configuration->init_conflicts();
 for(int i=0; i<nbvar; i++)
   for (int j=i+1; j< nbvar; j++)
     if (constraints[i][j])
       {if (configuration->config[i]==configuration->config[j])
          value++;
       configuration->incr_conflicts(i,configuration->config[i],
                                     configuration->config[j],1);
       configuration->incr_conflicts(j,configuration->config[i],
                                     configuration->config[i],1);
       }
 return value;
}
```

La méthode qui met à jour la structure de données des conflits `tabconflicts` est implantée comme suit.

```
void ColorCSProblem::fullincr_update_conflicts
                     (FullincrCSPConfiguration* configuration,Move* move)
{int changed_variable = ((CSPMove*)move)-> variable;
  int changed_value = ((CSPMove*)move)-> value;
 for (vector<int>::iterator vi= connections[changed_variable].begin();
                       vi!= connections[changed_variable].end(); vi++)
   {configuration->tabconflicts[*vi][configuration->config[changed_variable]]--;
    configuration->tabconflicts[*vi][changed_value]++;
   }
}
```

Ce code illustre les principales structures de données utilisées. Le tableau `config` contient la valeur courante de chaque variable d'une configuration. Les conflits sont stockés dans le tableau bidimensionnel `tabconflicts` indexé par une variable et un indice de valeur. Les contraintes binaires,c.à.d les liens entre les sommets, sont stockées de manière redondante pour des raisons d'efficacité dans deux structures. Un tableau bidimensionnel `constraints` indique si deux variables i et j sont liées par une contrainte.

6.3 Contributions

Cette partie détaille les caractéristiques originales de `INCOP`. D'abord, les calculs de coûts incrémentaux offerts par notre bibliothèque assurent son efficacité. Ensuite, des algorithmes à population efficaces peuvent être utilisés pour traiter les instances les plus difficiles quand la recherche locale ne suffit plus. Enfin, une sélection originale et paramétrée des mouvements, gérant une sorte de "candidate list strategy" [Glover et Laguna, 1997] peut mener à créer facilement de nouvelles variantes d'algorithmes existants.

6.3.1 Incrémentalité

Il est important de mettre à jour de manière incrémentale la structure `tabconflicts` des objets `Configuration` (voir partie 6.2.1). On suppose qu'on a à traiter un critère additif comme c'est le cas dans les MAX-CSP. En effet, la contribution de chaque valeur d'une variable à l'évaluation du coût d'une configuration est le nombre de contraintes non satisfaites par cette valeur, les autres variables ayant leur valeur courante. Puisque cette opération est réalisée très souvent (à chaque mouvement testé), il est utile d'évaluer rapidement l'impact d'un mouvement sur le coût de la configuration entière :

1. Dans `CSPconfiguration` : les conflits ne sont pas stockés : il faut calculer le nombre de contraintes non satisfaites par les anciennes et nouvelles valeurs.
2. Dans `IncrCSPconfiguration` : la contribution de la valeur courante est stockée dans `tabconflicts`; il ne faut calculer que la contribution de la nouvelle valeur.
3. Dans `FullincrCSPconfiguration` : toutes les contributions de toutes les valeurs possibles sont stockées dans `tabconflicts` ; l'évaluation d'un mouvement est immédiate.

Cela est implanté par deux méthodes de la classe du problème à résoudre (sous-classe de `OpProblem`).

- La méthode `move_evaluation` est appelée à la fin de la méthode `next_move`, c.à.d quand un mouvement est essayé.
- Les méthodes `incr_update_conflicts` ou `fullincr_update_conflicts` sont appelées chaque fois qu'un mouvement est effectué.

L'incrémentalité totale implantée dans les instances `FullincrCSPConfiguration` implique que l'effort de calcul est réalisé principalement dans la méthode `fullincr_update_conflicts`. C'est intéressant quand beaucoup de mouvements doivent être testés avant d'en accepter un. Quand le problème est peu dense, ce qui est le cas dans la plupart des instances de coloriage de graphe, la mise à jour est peu coûteuse : elle ne concerne que les valeurs des quelques variables liées par une contrainte à la variable courante. En pratique, ce mode tout incrémental peut faire gagner un ordre de grandeur dans le temps de calcul. La mémoire requise est aussi raisonnable pour les problèmes de coloriage : la taille de la structure de données des conflits est $N \times D$, où N est le nombre de sommets et D est le nombre de couleurs. Les bons résultats expérimentaux décrits dans la partie 6.4 montrent l'intérêt de cette incrémentalité.

6.3.2 Algorithmes Go With the Winners

Les algorithmes à population implantés dans `INCOP` sont des variantes de l'algorithme "Go With the Winners" [Dimitriou et Impagliazzo, 1996]. Plusieurs configurations sont traitées simultanément. Chaque configuration, appelée particule, réalise une marche aléatoire et, périodiquement, les plus mauvaises particules sont redistribuées sur les meilleures. Pour assurer une amélioration de la population, un seuil est baissé durant la recherche et aucun mouvement ne peut passer au dessus du seuil.

L'hybridation avec de la recherche locale est immédiate : au lieu de réaliser une marche aléatoire, chaque particule effectue de la recherche locale. Dans la bibliothèque, l'objet `GWWAlgorithm` possède un attribut, nommé `walkalgorithm`, qui contient un objet `LSAlgorithm`. Ainsi, aucun code supplémentaire n'est nécessaire et toute méthode de recherche locale peut être utilisée. Une de ces hybridations, `GWW-idw` [Neveu et Trombettoni, 2003d], qui hybride `GWW` avec l'algorithme de recherche locale `idw` a donné de très bons résultats (cf la partie 6.4 pour les résultats expérimentaux et le chapitre 7 pour une présentation complète de cet algorithme).

Il serait possible d'implanter dans la bibliothèque des algorithmes génétiques classiques ou des algorithmes mémétiques (hybrides d'algorithmes génétiques et de recherche locale). On notera cependant que `GWW` et `GWW-idw` présentent plusieurs avantages en comparaison avec des algorithmes génétiques. D'abord, ils sont plus simples. Ensuite, l'opérateur de croisement des algorithmes génétiques n'est généralement pas très pertinent dans des problèmes très structurés comme les CSP. Enfin, des évaluations incrémentales des mouvements ne peuvent pas être réalisées par l'opérateur de croisement, ce qui rend les algorithmes génétiques difficilement compétitifs pour des problèmes à critère additif.

6.3.3 Parcours du voisinage

Le point crucial dans les algorithmes de recherche locale est la manière dont les voisins de la configuration courante sont examinés et comment la prochaine configuration est choisie. Un parcours original du voisinage a été implanté dans `INCOP`.

Plusieurs paramètres spécifient ce parcours. Ils sont définis dans les objets de la classe `NeighborhoodSearch`. La classe `LSAlgorithm` possède un attribut qui contient un objet de la classe `NeighborhoodSearch`.

D'abord, il faut spécifier la méthode `is_feasible` qui donne la condition de faisabilité d'un mouvement. Par exemple, dans les algorithmes `ThresholdGWWAlgorithm`, le coût de la configuration doit rester sous le seuil courant. Le voisin est alors dit *atteignable*.

Pour régler finement l'effort d'intensification de la recherche, trois paramètres sont à la disposition de l'utilisateur pour le parcours du voisinage de la configuration courante : `Min_neighbors`, `Max_neighbors` et `No_acceptation`.

1. On teste d'abord `Min_neighbors` voisins pour sélectionner le *meilleur*.
2. Si aucun voisin n'a été accepté par la métaheuristique, on teste d'autres voisins jusqu'à ce qu'un soit accepté ou que `Max_neighbors` voisins aient été examinés.

3. Finalement, si aucun voisin parmi ces `Max_neighbors` n'a été accepté, le paramètre `No_acceptation` avec sa valeur `no_move`, `one_neighbor` ou `bestneighbor` indique quelle prochaine configuration choisir :
   - `no_move` : la configuration courante n'est pas modifiée,
   - `one_neighbor` : on choisit un voisin quelconque atteignable parmi les `Max_neighbors` voisins (sans tenir compte du critère d'acceptation de la métaheuristique),
   - `best_neighbor` : on choisit le meilleur voisin atteignable.

Dans les algorithmes existants, `Min_neighbors` vaut généralement 0 (par exemple, dans Metropolis), ou `Max_neighbors` (par exemple, dans une recherche avec liste taboue). Ce paramétrage permet de réaliser un compromis entre intensification de la recherche et une exploration qui tente de s'échapper des minimums locaux. Il permet d'implanter facilement des parcours de voisinage classiques. Par exemple :
- Rechercher le meilleur voisin dans le voisinage entier :
  `Min_neighbors` = `Max_neighbors` = taille du voisinage
- Rechercher le premier voisin acceptable dans un échantillon de K voisins :
  `Min_neighbors` = 0, `Max_neighbors` = K.

Remarque

`Min_neighbors` et `No_Acceptation` sont deux paramètres qui sont rarement explicités dans les algorithmes existants. Des études plus approfondies seront nécessaires pour évaluer leur importance. Nous avons déjà observé des comportements très intéressants avec des variantes d'algorithmes existants obtenus avec ce parcours du voisinage. Par exemple, nos expérimentations semblent indiquer que fixer `No_acceptation` à `one_neighbor` est un bon moyen pour s'échapper des minimums locaux et nous avons ainsi défini un algorithme simple de recherche locale nommé `idw` pour "intensification diversification walk" avec `Min_neighbors` = `Max_neighbors`. Cet algorithme n'a qu'un seul paramètre à régler, le nombre de voisins examinés au plus à chaque pas. Dès qu'un voisin ne détériorant pas l'évaluation est trouvé, il est choisi. Si tous les `Max_neighbors` voisins détériorent l'évaluation du courant, un voisin quelconque atteignable parmi eux est choisi et l'on peut ainsi sortir des minimums locaux.

L'algorithme `Configuration_move` (cf Algorithme 2) est implanté comme une méthode de la classe `LSAlgorithm`.

6.4 Expérimentations

Nous avons réalisé des expérimentations sur des instances difficiles issues principalement de deux catégories de problèmes codés comme des problèmes MAX-CSP pondérés : des instances de coloriage de graphe proposées dans le challenge Dimacs il y a 10 ans et des problèmes d'allocation de fréquence du CELAR [2]. Tous les tests ont été réalisés sur un `PentiumIII 935 Mhz` avec un système d'exploitation Linux.

6.4.1 Coloriage de graphe

Nous avons choisi des instances difficiles dans deux catégories du catalogue : les graphes de Leighton avec 450 sommets `le450_15c` (16680 arêtes), `le450_15d` (16750 arêtes), `le450_25c` (17425 arêtes) et `le450_25d` (17343 arêtes), et l'instance plus dense

[2] "Centre d'ELectronique de l'ARmement".

```
algorithme Configuration_move(x : a configuration) Retourne : a configuration
    i ← 0
    best ? ← (Min_neighbors > 1) or (No_acceptation=best-neighbor)
    best-cost ← +∞ ; one-x ← ⊥
    accepted ? ← false
    while (i < Min-Neighbors) or (i < Max-Neighbors and not(accepted ?)) do
        x' ← next_move(x)
        if is_feasible(x') then
            if acceptance ?(x, x') then accepted ? ← true
            if best ? and (cost(x') < best-cost) then
                x-best ← x'
                best-cost ← cost(x')
            end
            if (No_acceptation=one-neighbor) then one-x ← x'
        end
        i ← i + 1
    end
    if accepted ? then
        if best ? then
            return x-best
        else
            return x'
        end
    end
    if No-Acceptation=no-move or one-x=⊥ then
        return x
    end
    if No-Acceptation=best-neighbor then return x-best
    if No-Acceptation=one-neighbor then return one-x
fin.
```

Algorithm 2: La méthode `Configuration_move`

`flat300_28` avec 300 sommets et 21695 arêtes. Pour ces instances, une solution est cachée coloriable en respectivement 15, 25 et 28 couleurs.

L'instance `flat300_28` n'a encore jamais été coloriée en 28 couleurs (ni 29 ou 30 couleurs). A notre connaissance, trois algorithmes incomplets ont réussi à la colorier en 31 couleurs : la version distribuée de l'algorithme Impasse [Morgenstern, 1996], une recherche avec liste taboue [Dorne et Hao, 1998], et un algorithme hybride à base d'algorithme génétique et de recherche locale [Galinier et Hao, 1999]. Nous avons réussi à le colorier en 31 couleurs en quelques minutes et en 30 colors en 1.6 heures en moyenne en utilisant un algorithme de Metropolis (c.a.d., un recuit simulé à température constante comme proposé par [Connolly, 1990]) et le voisinage `min-conflict` implantant l'heuristique Min-conflicts.

Nous présentons ces résultats dans le tableau 6.1 : pour chaque instance, nous avons réalisé 10 essais pour un nombre donné de couleurs en changeant la graine du générateur de nombres aléatoires. Nous donnons le temps de calcul moyen, le nombre de conflits moyen restant à la fin de chaque essai et le taux de succès.

	nb-coul	temps	nb-coul	conflits	temps	succès	algo	voisin.
`le450_15c`	15	min	15	0	1.1 min	10/10	`GWW-idw`	`var-conflict`
`le450_15d`	15	min	15	1.4	1 min	5/10	`GWW-idw`	`var-conflict`
`le450_25c`	25	min	25	1.5	55 min	1/10	Metropolis	`var-conflict`
`le450_25d`	25	min	25	1.3	58 min	1/10	Metropolis	`var-conflict`
`flat300_28`	31	h	31	0.3	4 min	9/10	Metropolis	`min-conflict`
`flat300_28`	31	h	**30**	1.6	1.6 h	5/10	Metropolis	`min-conflict`

TAB. 6.1 – Résultats de coloriage de graphes. Les meilleurs résultats connus sont indiqués sur la patrie gauche du tableau(nombre de couleurs, temps) ; les résultats avec `INCOP` sur la partie droite (nombre de couleurs, nombre de conflits (moyenne), temps de calcul (moyenne), taux de succès, algorithme et voisinage utilisés.

6.4.2 Comparaison avec d'autres bibliothèques

Nous avons aussi comparé `INCOP` avec d'autres bibliothèques de recherche locale sur les mêmes instances de coloriage de graphe. Nous présentons les résultats publiés dans trois articles : `iOpt` [Voudouris et Dorne, 2002] implanté en Java, `Easylocal++` [DiGaspero et Schaerf, 2000], Galinier et Hao dans [Galinier et Hao, 1999]. Ces comparaisons ont été faites sur les graphes aléatoires DSJC du challenge DIMACS avec un nombre de couleurs supérieur au nombre chromatique pour rester conforme avec les résultats déjà publiés. Récemment, quelques résultats sur ces graphes ont été publiés avec `Discropt` [Phan et Skiena, 2002]. Comme le temps de calcul était limité à 100 s, ces graphes ont été coloriés avec un nombre de couleurs plus grand que celui des expérimentations décrites ci-après et cette approche n'est pas apparue compétitive. Les résultats présentés pour `INCOP` dans le tableau 6.2 sont les temps de calcul moyens sur 10 essais pour la résolution de chaque instance (la méthode ayant réussi tous ses essais). Nous avons utilisé un algorithme de Metropolis avec température constante et un voisinage `var-conflict`. Nous pensons que les bons résultats que nous avons obtenus sont en grande partie due à l'incrémentalité totale du calcul des coûts réalisée par notre bibliothèque lors de l'exploration du voisinage. Le seul autre système qui mentionne le même type d'incrémentalité est le système de Galinier et Hao [Galinier et Hao, 1999] qui obtient des performances comparables.

6.4.3 Problème des N-reines

Sur la page Web de `Easylocal++`, il y a deux exemples de problèmes qui sont fournis : les n-reines et un problème d'emploi du temps. Nous avons téléchargé la bibliothèque `Easylocal++` avec son implantation des n-reines et l'avons fait tourner sur notre machine. Nous avons alors implanté dans `INCOP` ce problème avec le même voisinage (échange dans un codage par permutation). Nous avons obtenu des temps de calcul comparables en utilisant le même algorithme de descente : par exemple, 10 essais de taille 3000, initialisés aléatoirement, sont résolus en 20.5 s en moyenne avec `Easylocal++` et en 21.6 s avec `INCOP`.

6.4.4 Affectation de fréquences

Nous avons aussi traité les trois instances les plus difficiles du problème d'affectation de fréquences du CELAR. : les instances `celar6`, `celar7` et `celar8`. Ces problèmes sont réalistes ayant été construits à partir de sous-parties d'un problème réel. `celar6` a 200

Problème	Nb coul	`INCOP`	`Easylocal++`	Galinier	`IOPT`
Machine		`P III 935`	`P III 750`	`Sparc II 333`	`P III 866`
Method		Metropolis	Tabou	Tabou	Tabou ou RS
125.1	6	0.005	0.031		166 (RS)
125.1	5	0.36			44 (T)
125.5	18	0.18	2.65		557 (T)
125.9	44	0.97	13.5		2904 (T)
250.1	9	0.04	0.61		
250.5	30	1.42	52.2	15	
250.5	29	7.81		85	
250.9	75	3.07	118.		
500.1	14	0.18	7.05		
500.5	54	2.43	457		
500.5	52	14.7		14	
500.5	51	96		47	
500.9	140	8.75	1856		

TAB. 6.2 – Résultats sur les graphes aléatoires `DSJC` (temps de calcul en secondes)

variables et 1322 contraintes; `celar7` a 400 variables et 2865 contraintes; `celar8` a 916 variables et 5744 contraintes.

Les variables sont les fréquences auxquelles ont doit affecter une valeur appartenant à un ensemble de fréquences autorisées. (les domaines ont environ 40 valeurs). Les contraintes sont de la forme $|x_i - x_j| = \delta$ ou $|x_i - x_j| > \delta$ (pour éviter les interférences). Notre codage est standard et ne crée que les variables paires dans le CSP avec seulement les contraintes d'inégalités [3].

La fonction objectif à minimiser est une somme pondérée des violations de contraintes. Les contraintes appartiennent à quatre catégories avec différents coûts de violation :

- les contraintes de `celar8` ont pour coût 1, 2, 3 ou 4.
- les contraintes de `celar6` ont pour coût 1, 10, 100 ou 1000.
- les contraintes de `celar7` ont pour coût 1, 100, 10000 ou 10^6.

[de Givry *et al.*, 1997] et [Koster *et al.*, 1999] décrivent des algorithmes complets basés sur une décomposition du graphe de contraintes et de la programmation dynamique pour résoudre l'instance `celar6`. Les deux autres instances sont réellement difficiles et aucun algorithme complet n'a fourni de preuve d'optimalité [4]. L'algorithme décrit dans [Kolen, 1999] est basé sur un algorithme génétique spécialisé avec un opérateur de mutation codant de la PNLE. L'algorithme décrit dans [Voudouris et Tsang, 1998] est de la recherche locale prenant en compte les conflits pour sélectionner les voisins. Les bons résultats obtenus par l'algorithme `GWW-idw` implanté dans `INCOP` sont présentés sur le tableau 6.3. Nous avons par ailleurs obtenu la meilleure borne connue 343592 avec une variante de `GWW-idw` utilisant un autre paramétrage dans la gestion du voisinage.

[3] Une bijection existe entre les variables paires et impaires. Une simple propagation des égalités permet de déduire les valeurs des variables impaires.

[4] Cela peut être expliqué par la taille de `celar8` et par le critère de `celar7`.

	borne	ref	meilleur (moyen)	temps	succès	algo
celar6	3389	mn (GVS97,KHK99)	3389 (3405.7)	9 mn	4/10	GWW-idw
celar7	343592	mn (KHK99,VT98)	343596 (343657)	4.5 h	1/10	GWW-idw
celar8	262	mn (KHK99,VT98)	262 (267.4)	33 mn	2/10	GWW-idw

TAB. 6.3 – Résultats sur les bancs d'essai du CELAR. Les résultats des meilleurs algorithmes sont présentés sur la gauche du tableau ; les résultats avec INCOP sur le côté droit.

6.5 Autres bibliothèques

Comme il a été mentionné dans l'introduction, de nombreuses bibliothèques ont été développées ces dernières années. Il n'est pas possible de les présenter en détails en quelques lignes. Certaines sont limitées à des méthodes de recherche locale, d'autres fournissent des algorithmes de recherche locale pure et des algorithmes à population comme des algorithmes génétiques hybridés avec de la recherche locale. `Templar` permet d'écrire des algorithmes distribués et d'utiliser du parallélisme. Tous ces outils sont présentés dans [Voß et Woodruff, 2002c]. Nous pouvons résumer leurs principales caractéristiques dans le tableau suivant.

Bibliothèque	langage	métaheuristique	divers
Templar	C++	tabou, RS, algos gén.	parallélisme
Neighbor Searcher	C++ Templates	rec. locale + VNS	
Hotframe	C++ extension	rec. locale	dispo. annoncéee
Esylocal++	C++	rec. locake	ouvert ; disponible
Iopt	Java	rec. locale + algo mémétique	
Discropt	C++	rec. locale	dispo. annoncée
Localizer++	C++	rec.locale	

TAB. 6.4 – Bibliothèques de méthodes incomplètes pour l'optimisation combinatoire

Le caractère principal de la plupart de ces bibliothèques est souvent plus la simplicité et l'extensibilité que l'efficacité. Il est difficile de comparer leurs performances car les résultats ont été publiés sur des problèmes différents. Nous avons analysé dans la partie 6.4.1 les résultats publiés en coloriage de graphe qui montrent que `INCOP` est souvent compétitif.

6.6 Conclusion

Cet article a présenté la nouvelle bibliothèque `INCOP` écrite en C++ pour l'optimisation combinatoire avec des algorithmes incomplets. Nous avons implanté plusieurs algorithmes de recherche locale et à population originaux et efficaces. Un grand effort a porté sur l'incrémentalité des évaluations des mouvements que nous avons obtenue en maintenant une structure de données spécifique pour les conflits. Une autre caractéristique est notre processus paramétré de gestion du voisinage qui améliore des métaheuristiques existantes et fournit un nouvel algorithme simple de recherche locale `idw`.

Nous pensons qu'il n'existe pas d'algorithme incomplet pouvant résoudre efficacement tous les problèmes. Ainsi il est important de pouvoir tester rapidement différents algorithmes, différents voisinages. Une telle bibliothèque le permet et nous avons obtenu de bons résultats pour les problèmes d'affectation de fréquences du `CELAR` avec `GWW-idw` et pour les problèmes de coloriage de graphe avec `idw` ou `Metropolis`, avec un voisinage de type `min-conflict` ou `var-conflict` selon l'instance résolue. Nous avons, pour la première fois, colorié l'instance `flat300_28` avec 30 couleurs.

La première version de `INCOP` est téléchargeable.

Chapitre 7

Algorithme hybride GWW-idw

Article paru dans la revue électronique JEDAI (volume 3, 2004)
Auteurs : Bertrand Neveu, Gilles Trombettoni

Résumé

Une nouvelle métaheuristique pour l'optimisation combinatoire, appelée *"Go With the Winners"* (`GWW`) a été proposée par Dimitriou et Impagliazzo en 1996. `GWW` gère plusieurs configurations en même temps, utilise un seuil pour éliminer les pires configurations, et inclut une étape de marche aléatoire qui permet une distribution uniforme des configurations visitées dans chaque sous-problème.

Cet article propose de remplacer l'étape de marche aléatoire de `GWW` par de la recherche locale pour favoriser les meilleures solutions.

Nous proposons une instance de cet algorithme hybride, appelée `GWW-idw`. Nous comparons différentes manières de baisser le seuil. Nous ajoutons un seul paramètre pour intégrer la recherche locale dans le schéma existant. Nous montrons comment et dans quel ordre, régler les paramètres.

Une étude expérimentale a été menée sur des instances difficiles de coloriage de graphe issues du challenge DIMACS et sur des problèmes d'affectation de fréquences du CELAR. Les meilleures bornes connues ont été trouvées sur toutes les instances.

7.1 Introduction

Notre travail est basé sur l'algorithme *"Go With the Winners"* (`GWW`) présenté dans [Dimitriou et Impagliazzo, 1996]. `GWW` explore différentes configurations de l'espace de recherche en même temps en gérant une population de solutions appelées *particules*. Au début, les particules sont distribuées aléatoirement et un seuil est placé au niveau du coût de la plus mauvaise particule. Les phases suivantes sont effectuées itérativement jusqu'à qu'il n'y ait plus de particule en dessous du seuil (nous considérons un problème de minimisation) :

1. **Redistribution :** Les particules au dessus du seuil sont "redistribuées" sur les autres (d'où le nom de l'algorithme) : une particule redistribuée est placée sur la même configuration qu'une autre particule sous le seuil, choisie aléatoirement.

2. **Exploration aléatoire :** Cette phase est réalisée pour chaque particule et tend à séparer les particules qui ont été regroupées par la phase de redistribution. Elle est basée sur une marche aléatoire de longueur spécifiée. Chaque étape de cette marche fait passer une particule sur un état voisin qui reste sous le seuil.

3. La valeur du seuil est décrémentée de 1.

Quand le problème a de bonnes propriétés (voir partie 7.2), `GWW` est théoriquement capable de trouver une solution optimale en temps polynomial avec une forte probabilité. Cependant, en pratique, de premières expérimentations réalisées sur les instances du CELAR et de coloriage ont donné de plus mauvaises performances qu'un algorithme de Metropolis standard [Connolly, 1990] (recuit simulé à température constante).

Dans la phase d'exploration aléatoire de l'algorithme `GWW`, les meilleures configurations ne sont pas favorisées et une particule peut indifféremment monter ou descendre. Ainsi, la progression dans `GWW` est due uniquement à la gestion du seuil. Dans cet article, nous montrons de manière expérimentale que `GWW` peut être rendu plus efficace en remplaçant la phase de marche aléatoire par de la recherche locale. Cela donne l'algorithme `GWW-LS` qui est décrit par la suite. Les principales contributions sont les suivantes :

- Précédemment, `GWW` a été appliqué à des problèmes de théorie des graphes : la clique maximum et la bissection [1]. Toutes les versions correspondantes de `GWW` supposent que les coûts possibles sont des entiers petits, justifiant la baisse du seuil de 1 à chaque itération. Ce n'est pas le cas pour de nombreux problèmes comme les MAX-CSP pondérés présentés dans cet article. Par exemple, l'instance `celar7` comprend des configurations dont le coût varie entre 343000 et 2.10^7. Nous proposons donc différentes manières de baisser le seuil et les comparons entre elles.
- `GWW-LS` gère plusieurs paramètres : le nombre B de particules, un paramètre T utilisé pour baisser le seuil, la longueur S de la marche, et les paramètres additionnels de la recherche locale sélectionnée (la longueur de la liste taboue, ou une température permettant de s'échapper des minimums locaux). Nous proposons une instance de ce schéma `GWW-LS`, appelé `GWW-idw` (`GWW` avec *marche avec intensification et diversification*), qui limite le nombre de paramètres additionnels à 1. Pour minimiser l'effort requis pour régler les paramètres, les expérimentations ont validé une manière heuristique d'effectuer ce réglage.
- Nous avons mené des expérimentations sur des instances difficiles de problèmes d'affectation de fréquences et de coloriage de graphe. `GWW-idw` trouve les meilleures bornes (connues) pour tous ces problèmes et nous le comparons avec `GWW` et d'autres algorithmes de recherche locale.

La partie 7.2 donne un bref historique de `GWW`. La partie 7.3 détaille le schéma `GWW-LS`. L'instance `GWW-idw` de `GWW-LS` est décrite dans la partie 7.4. Les expérimentations sont exposées dans la partie 7.5. La partie 7.5.2 présente divers moyens de gérer le seuil dans des problèmes réels. La partie 7.5.3 donne une première manière heuristique de régler les quatre paramètres de `GWW-idw`. Les parties 7.5.4 et 7.5.5 montrent les performances de `GWW-idw` et le comparent à `GWW` et à des méthodes de recherche locale.

[1] Couper un graphe en deux sous graphes de même nombre de nœuds avec un minimum d'arêtes entre les deux sous-graphes.

7.2 De GWW à GWW-LS

Avant de présenter l'algorithme GWW, nous définissons un *graphe de recherche* qui est une abstraction d'un problème d'optimisation combinatoire. Les nœuds du graphe de recherche sont les différentes configurations (c.à.d. les solutions potentielles) et les arêtes relient deux configurations qui sont "voisines", c.à.d , pour lesquelles un changement élémentaire permet de passer d'une configuration à l'autre. Nous supposons que la valeur, ou coût, de chaque nœud est un entier. Le but de l'optimisation combinatoire est de trouver un nœud de coût minimum.

L'algorithme "Go with the winners" a été introduit dans [Aldous et Vazinari, 1994]. Cet algorithme a été conçu pour trouver une feuille de profondeur maximum dans un arbre. Au début, plusieurs particules sont placées à la racine de l'arbre. Dans la phase d'exploration, chaque particule est déplacée sur un nœud fils choisi aléatoirement. Dans la phase de redistribution, les particules qui ont atteint une feuille sont redistribuées : elles sont remplacées par une copie d'une autre particule non feuille choisie aléatoirement. Le processus s'arrête quand toutes les particules sont sur une feuille. Les auteurs ont déterminé le nombre de particules nécessaire pour trouver le nœud le plus profond avec une forte probabilité dépendant d'une mesure d'équilibrage de l'arbre.

L'algorithme GWW dont il est question dans cet article est présenté dans [Dimitriou et Impagliazzo, 1996]. C'est une modification de la première version pour l'optimisation combinatoire. L'arbre de la première version apparaît dans tout problème d'optimisation en gérant un seuil. En effet, le graphe de recherche peut être divisé hiérarchiquement en plusieurs sous-graphes selon leurs coûts. Plus précisément, un nœud de l'arbre est constitué d'un sous-graphe de recherche dont les sommets ont un coût inférieur ou égal au seuil. Le sommet de l'arbre contient le graphe de recherche entier (quand le seuil est au plus haut). Baisser le seuil divise le graphe de recherche en composantes connexes qui forment une hiérarchie entre un nœud de l'arbre et ses fils. Le fait que deux régions deviennent déconnectées quand le seuil décroît signifie qu'aucune marche aléatoire (restant au niveau du seuil ou sous le seuil) ne peut déplacer une particule d'une région à l'autre.

Dimitriou et Impagliazzo analysent dans [Dimitriou et Impagliazzo, 1998] les performances de GWW par rapport aux propriétés combinatoires du problème. Ils démontrent en particulier que deux conditions suffisantes rendent un problème soluble en temps polynomial avec une forte probabilité :

- Le nombre de fils d'un nœud de l'arbre doit être borné polynomialement. Cela signifie que baisser le seuil (de 1) ne doit pas faire apparaître un nombre exponentiel de régions déconnectées. Cela garantit une convergence en temps polynomial si chaque région déconnectée possède une particule.
- L'*expansion locale* est vérifiée, c.à.d chaque nœud d'un sous-graphe de recherche a un nombre suffisant de voisins. Ainsi, une marche suffisamment longue dans une région donnée assure une distribution uniforme des particules à l'intérieur et ne les confine pas dans une petite portion de la région. Les phases de redistribution et de marche aléatoire assurent que chaque composante connexe reçoit un nombre de particules proportionnel à sa taille.

La première condition souligne l'importance du nombre B de particules. La seconde est liée à la longueur S d'une marche aléatoire.

Différentes marches aléatoires pour GWW sont décrites dans la littérature[2] :

[2] Dans la version initiale [Aldous et Vazinari, 1994], une particule est simplement déplacée sur un fils.

- Dans [Dimitriou et Impagliazzo, 1996], une marche aléatoire de longueur 1 est effectuée à chaque étape : pour chaque particule de coût i, tous les voisins sont visités et l'un d'entre eux est choisi parmi ceux ayant un coût $i-1$.
- Dans [Dimitriou et Impagliazzo, 1998], une marche aléatoire de longueur S est proposée. Cette marche alterne des nœuds de coût i et de coût $i-1$.
- [Carson et Impagliazzo, 1999] et [Dimitriou, 2002] présentent une version simplifiée où la marche aléatoire de longueur S doit rester au seuil ou sous le seuil.

Cet article introduit l'algorithme `GWW-LS` où la marche aléatoire de `GWW` est remplacée par de la recherche locale, chaque étape de cette marche restant sous ou au niveau du seuil. La différence entre les deux approches est qu'une recherche locale tend à faire progresser la particule, ce qu'une marche aléatoire ne fait pas. La population descend donc plus vite mais la distribution uniforme des particules dans une région connexe n'est plus respectée. Cependant, les bons résultats expérimentaux obtenus avec `GWW-LS` nous laissent penser que d'autres conditions de convergence en temps polynomial avec forte probabilité existent. Une telle hypothèse est évoquée dans la conclusion.

7.3 Description de `GWW-LS`

Les différences entre `GWW` et `GWW-LS` résident dans les deux points suivants :

- *La descente du seuil est adaptative.*
  La partie 7.5.2 compare différentes façons de définir la fonction `Baisserseuil` utilisée pour baisser le seuil. Malheureusement, une gestion réaliste implique l'ajout d'un nouveau paramètre T dans le schéma `GWW-LS`.
- *La marche n'est plus simplement aléatoire.*
  Les deux boucles `for` de l'algorithme 1 décrivent la marche : chaque particule effectue une marche de longueur S. La différence principale avec les marches utilisées dans les versions précédentes [Carson et Impagliazzo, 1999; Dimitriou, 2002] réside dans la fonction `Cherchervoisin`.

Dans `GWW`, cette fonction renvoie un voisin de la configuration de la particule p. Plus précisément, tous les voisins (ou un nombre limité) sont visités jusqu'à ce qu'un soit trouvé sous ou au niveau du seuil. Si aucun n'est trouvé, la configuration courante est renvoyée, c.à.d. la particule ne bouge pas. Dans `GWW-LS`, cette fonction essaye de faire progresser la particule, tout en cherchant à s'échapper des minimums locaux.

```
algorithme GWW-LS (B : nombre de particules ; S : longueur de la marche, T : paramètre
seuil ; WP : paramètres de la marche) : configuration
  Initialisation : chaque particule est aléatoirement placée sur une configuration et rangée
  dans le tableau Particules
  seuil ← coût de Pire(Particules)
  Boucle
    seuil ← Baisserseuil(seuil, T, Particules)
    si Meilleur(Particules) > seuil alors retourner(Meilleur-trouvé) /* la
    meilleure configuration trouvée durant la recherche */
    Redistribuer(Particules) /* Place chaque particule ayant un coût supérieur au
    seuil sur la configuration d'une particule vivante choisie aléatoirement */
    pour tout particule p dans Particules faire
      pour i de 1 à S faire
        voisin ← Cherchervoisin(p, WP)
        Déplacer(p,voisin) /* Mouvement de p sur une nouvelle configuration */
      fin.
    fin.
  fin.
fin.
```

Algorithm 3: L'algorithme GWW-LS

7.4 De GWW-LS à GWW-idw

Afin de valider le schéma GWW-LS, nous devons concevoir au moins un algorithme déduit de GWW-LS et montrer son efficacité. Nous avons essayé divers algorithmes classiques de recherche locale pour améliorer GWW. Nous avons obtenu de bons résultats avec un algorithme de Metropolis (acceptant toujours un voisin d'un coût inférieur ou égal, et acceptant un voisin plus mauvais avec une probabilité dépendant d'une "température"). Cependant, un important travail nous a conduits à le simplifier en diminuant le nombre de paramètres et en trouvant comment les régler plus rapidement.

L'algorithme GWW gère deux paramètres : le nombre B de particules et la longueur S de la marche aléatoire. En pratique, un troisième paramètre T doit être ajouté pour que le seuil ne baisse pas trop lentement. Notre nouvel algorithme GWW-idw (GWW avec *marche avec intensification et diversification*) ajoute seulement un paramètre N pour la recherche locale. Cet unique paramètre est utilisé par une particule pour progresser vers les meilleures solutions tout en étant capable de s'échapper des minimums locaux. Le paramètre N est le nombre maximum de voisins qui sont explorés par une particule pendant l'exécution de la fonction Cherchervoisin, en vue de bouger d'une configuration à une autre. A partir d'une configuration courante x, la détermination de la nouvelle configuration retournée par la fonction Cherchervoisin de GWW-idw s'effectue ainsi :

1. un voisin x' parmi les N explorés a un coût inférieur ou égal au coût de x $\Rightarrow$ **Retourner** x'
2. aucun voisin parmi les N explorés n'a un coût inférieur ou égal au coût de x et un voisin x'' parmi les N est au niveau du seuil ou en dessous $\Rightarrow$ **Retourner** x''
3. Les N voisins explorés sont au dessus du seuil $\Rightarrow$ **Retourner** x

Le premier cas diffère radicalement de `GWW` standard. Il reproduit le comportement des algorithmes de descente comme GSAT [Selman *et al.*, 1992] : un voisin est accepté seulement si son coût est meilleur ou le même que celui de la configuration courante. Le deuxième cas indique comment choisir une configuration quand aucun voisin visité n'a été accepté. Il propose une sélection aléatoire d'un voisin qui permet de sortir des minimums locaux. Certaines particules peuvent ainsi remonter assez haut, mais cette remontée est limitée par le seuil.

Ainsi le paramètre N évite de visiter tous les voisins et réalise un compromis entre progresser et sortir des minimums locaux. Cette recherche locale, appelée `idw` [Neveu et Trombettoni, 2003d], peut se voir comme une généralisation de la *candidate list strategy Acceptation+* [Glover et Laguna, 1997]. Pour résumer, `GWW-idw` gère les paramètres suivants :

- Un nombre B de particules utilisé pour explorer l'espace de recherche en parallèle : les particules doivent être uniformément distribuées entre les composantes connexes apparaissant dans le graphe de recherche quand le seuil est baissé.
- Une longueur S pour les marches utilisée pour séparer les particules regroupées, et qui permet aux particules de progresser vers de meilleures configurations.
- Un paramètre T utilisé pour gérer le seuil dans les problèmes réels (voir partie 7.5.2).
- Un paramètre additionnel N qui a un rôle crucial. D'abord, il est nécessaire de limiter le nombre de voisins visités quand le voisinage est grand. Par ailleurs, N doit être assez grand pour permettre aux particules de progresser. Enfin, N doit être suffisamment petit pour permettre aux particules de sortir rapidement des minimums locaux.

Complexité de `GWW-idw`

La complexité en pire cas de `GWW-LS` ou `GWW-idw` reste la même que celle de `GWW`, c.à.d, $O(B \times S \times N \times nT)$, où nT est le nombre de fois que le seuil est baissé, B le nombre de particules, S la longueur d'une marche et N le nombre maximum de voisins explorés par pas d'une marche.

7.5 Expérimentations

Cette partie présente quelques résultats expérimentaux. Nous sélectionnons un moyen de baisser le seuil. Nous proposons une méthode pour régler les quatre paramètres. Enfin, nous comparons `GWW-idw` avec `GWW` standard d'une part, et des algorithmes de recherche locale : Metropolis, recuit simulé, liste taboue et `idw` d'autre part.

7.5.1 Bancs d'essais, codage, implantation

Nous avons réalisé des expérimentations sur des instances difficiles issues de deux catégories de problèmes modélisés comme des problèmes MAX-CSP pondérés : des instances de coloriage de graphe proposées dans le challenge DIMACS [Morgenstern, 1996], et des problèmes d'affectation de fréquences du CELAR (Centre d'électronique de l'Armement).

Coloriage de graphe

Nous avons sélectionné trois instances difficiles de coloriage de graphes du catalogue DIMACS : `le450_15c` avec 450 sommets et 16680 arêtes, `le450_25c` avec 450 sommets et 17425 arêtes, et l'instance plus dense `flat300_28` avec 300 sommets et 21695 arêtes. Ces instances se trouvent sur le site de DIMACS :
`ftp://dimacs.rutgers.edu/pub/challenge/graph`

Dans toutes ces instances, une solution coloriable en respectivement 15, 25 et 28 couleurs est cachée. Dans cet article, les instances de coloriage sont modélisées en MAX-CSP : les variables sont les sommets du graphe à colorier ; le nombre d de couleurs donne des domaines allant de 1 à d ; les sommets reliés par une arête doivent prendre des couleurs différentes. Colorier un graphe en d couleurs revient à minimiser le nombre de contraintes non satisfaites et à trouver une solution de coût 0.

Affectation de fréquences du CELAR

Nous avons aussi sélectionné les trois instances les plus difficiles des problèmes d'affectation de fréquences du CELAR : `celar6`, `celar7` et `celar8`. Ces instances sont réalistes car elles ont été construites à partir de différentes parties d'un problème réel. `celar6` a 200 variables et 1322 contraintes ; `celar7` a 400 variables et 2865 contraintes ; `celar8` a 916 variables et 5744 contraintes. Ces instances se trouvent sur le site :
`http://fap.zib.de/problems/CALMA/`

Les variables sont les fréquences à affecter avec une valeur appartenant à un ensemble discret prédéfini (domaines de taille d'environ 40). Les contraintes sont de la forme : $|x_i - x_j| = \delta$ ou $|x_i - x_j| > \delta$ (pour éviter les interférences). Notre modélisation est standard et crée seulement les variables paires dans le CSP avec les inégalités uniquement[3].

La fonction objectif à minimiser est une somme pondérée des violations de contraintes. Les contraintes appartiennent à quatre catégories avec différents coûts de violation :

- Les contraintes de `celar8` ont un poids 1, 2, 3 ou 4.
- Les contraintes de `celar6` ont un poids 1, 10, 100 ou 1000.
- Les contraintes de `celar7` ont un poids 1, 10^2, 10^4 ou 10^6.

Voisinage

Nous avons choisi la définition habituelle de voisinage pour les CSP : une nouvelle configuration x' est un voisin de la configuration courante x si les deux ont les mêmes valeurs, sauf pour une variable. Ce voisinage est réduit en utilisant en partie l'heuristique *Min-conflicts* de Minton [Minton *et al.*, 1992] : on ne considère que les variables dont la valeur courante participe à un conflit dans la configuration x. De plus, pour le problème `celar7`, on ne considère que les conflits d'un coût significatif par rapport au coût de configuration courante : contraintes de poids 10000, 100 et 1 pour des configurations de coûts respectivement inférieurs à 20000000, 800000 et 400000.

Implantation

Tous les algorithmes ont été développés dans la même bibliothèque logicielle INCOP [Neveu et Trombettoni, 2003d]. Notre plate-forme est implantée en `C++` et toutes

[3] Une bijection existe entre variables paires et impaires. Une simple propagation des égalités permet de déduire les valeurs des variables impaires.

les expérimentations ont été effectuées sur `PentiumIII 935 Mhz` sous Linux. Tous les algorithmes appartiennent à une hiérarchie de classes partageant du code, ce qui permet des comparaisons équitables.

7.5.2 Gestion du seuil

Nos premières expérimentations ont montré que la baisse du seuil de 1 à chaque itération ne convenait pas pour les problèmes du CELAR, pour lesquels l'intervalle des coûts possibles est très grand. Même pour le coloriage où le paysage est plus "plat", nous avons obtenu de meilleurs résultats en gérant avec attention la baisse du seuil. Plusieurs gestions du seuil ont été conçues. Elles diffèrent l'une de l'autre par le delta (en terme de coût) à soustraire au seuil courant à chaque itération :

1. $\Delta_{aveugle} = 1 + \text{seuil} \times T_{aveugle}$
2. $\Delta_{pire} = 1 + \text{dist}(B) + \text{seuil} \times T_{pire}$
3. $\Delta_{adaptatif} = 1 + \text{dist}(i) \quad (= 1 + \text{dist}(B) + 100\%(\text{coût}(B) - \text{coût}(i)))$
4. $\Delta_{meilleur} = 1 + \text{dist}(B) + T_{meilleur}(\text{coût}(B) - \text{coût}(1))$
5. $\Delta_{borneinf} = 1 + \text{dist}(B) + (\text{seuil} - B_{borneinf}) \times T_{borneinf}$

$T_{aveugle}$, T_{pire}, $T_{meilleur}$, $T_{borneinf}$ ou i est le paramètre à régler pour rendre la baisse du seuil plus ou moins rapide. Les quatre premiers paramètres sont des taux ; `dist(i)` est la distance (c.à.d., la différence de coût) entre le seuil et la $i^{\text{ème}}$ meilleure particule ; ainsi, B est la particule la plus mauvaise et `dist`(B) est la distance (en coût) entre le seuil et cette particule B. `coût`(j) est le coût de la $j^{\text{ème}}$ meilleure particule.

Le premier delta essayé a été $\Delta_{aveugle}$. Les autres deltas sont adaptatifs et tiennent compte de la manière dont les particules descendent. Ils essayent de remédier à deux mauvaises caractéristiques de $\Delta_{aveugle}$: $\Delta_{aveugle}$ est souvent trop petit au début et trop grand à la fin. En effet :

- Lors des premières marches, les particules progressent facilement et même la plus mauvaise particule est très en dessous du seuil. Le composant `dist`(B) des 4 derniers deltas est surtout utile dans ce cas.
- A la fin, quand le seuil se rapproche de la meilleure configuration trouvée, le delta reste grand si le coût de la meilleure solution est supérieur à 0 (c'est le cas pour les problèmes du CELAR).

  $B_{borneinf}$ est un second paramètre nécessaire pour régler $\Delta_{borneinf}$. Il correspond à une borne inférieure dépendant du problème (par exemple 3389 pour `celar6` puisque cette borne a été trouvée par des méthodes complètes). $\Delta_{borneinf}$ a été testé pour vérifier l'intérêt d'avoir un seuil baissant lentement à la fin.

Les conclusions des tests décrits plus bas sont les suivantes. Premièrement, il est généralement intéressant de forcer le seuil à descendre sous la plus mauvaise particule (pour effectuer au moins un regroupement). La seconde conclusion contredit nos premières intuitions : prêter attention à la façon dont le seuil décroît à la fin n'est pas utile. C'est pourquoi nous avons adopté la simple gestion du seuil Δ_{pire}.

$\Delta_{adaptatif}$ tend à converger rapidement vers 1, même quand on est encore loin des meilleures solutions. En effet, quand progresser devient difficile, les particules ne sont pas uniformément distribuées (en terme de coût) entre le seuil et la meilleure particule, de plus en plus de particules restent au seuil et `dist`(i) devient nul.

Pour pallier cette difficulté, $\Delta_{meilleur}$ utilise la plus grande distance adaptative possible puisque `coût`(B)−`coût`(1) ne devient nul que quand `GWW-LS` termine.

Finalement, les relativement mauvais résultats obtenus par une gestion avec $\Delta_{meilleur}$ par rapport au simple $\Delta_{aveugle}$ sur des instances avec des paysages de recherche chahutés nous ont conduits à simplement corriger ce dernier en prenant en compte la particule la pire pour accélérer le début de l'algorithme et nous avons conçu et adopté Δ_{pire}. Les séries de tests conduisant à ce choix sont rassemblés dans le tableau 7.1.

	temps	**aveugle**	**adaptatif**	**meilleur**	**pire**	**borneinf**
`le450_15c`	103	61 (44)	4.2 (42)	0.9 (42)	**0.2 (41)**	–
`le450_25c`	70	11.4 (29)	10.2 (32)	10.1 (31)	10.1 (31)	–
`flat300_28`	112	3.7 (48)	3.4 (48)	3.6 (48)	3.7 (49)	–
`celar8`	140	303 (103)	335 (123)	306 (99)	**289 (106)**	303 (104)
`celar6`	102	3504 (49)	3515 (64)	3557 (43)	**3451 (49)**	3537 (41)
`celar7`	135	372125 (138)	627774 (152)	422616 (135)	377214 (141)	**370784 (138)**

TAB. 7.1 – Gestion du seuil. Les cases donnent le coût moyen d'une solution obtenu par `GWW-idw` pour un temps d'exécution donné en secondes (deuxième colonne) ; entre parenthèses, le nombre moyen de modifications de seuil. Les meilleurs résultats sont en gras.

Pour chaque instance, 10 essais ont été réalisés et les moyennes sont inscrites sur le tableau. Les instances du CELAR ont été traitées avec $B = 50$ ($B = 200$ pour `celar6`), $S = 200$, $N = 40$. Les instances de coloriage ont été résolues avec $B = 20$, $S = 2000$ et différentes valeurs pour N.

Les tests sur `le450_25c` et `flat300_28` ne sont pas significatifs (n'importe quel schéma de baisse du seuil convient). $\Delta_{adaptatif}$ est souvent le pire et $\Delta_{borneinf}$ donne généralement de plus mauvais résultats que Δ_{pire}. Enfin, Δ_{pire} n'est jamais mauvais et est souvent le meilleur.

7.5.3 Réglage des paramètres

Un avantage de `GWW-idw` est que chacun de ses quatre paramètres semble avoir un rôle spécifique. Cependant, trois de ces quatre paramètres ne sont pas faciles à régler a priori. En effet, réduire T ou augmenter B ou S donne toujours des résultats de meilleure qualité en allongeant le temps de calcul. Il n'est pas évident de sélectionner quelle augmentation de paramètre améliorera beaucoup la solution sans coûter trop de temps. Cependant, un réglage très fin semble inutile et nous avons obtenu des procédures de réglage prometteuses. Nos expérimentations et la littérature nous ont conduits à régler les paramètres de la manière heuristique suivante :

1. **T :** Le comportement asymptotique de T est le plus simple à observer. Dans nos expérimentations, les taux T_{pire} et $T_{aveugle}$ sont toujours choisis entre 0.5% et 2% ; le taux $T_{meilleur}$ entre 0.5% et 10%. Des taux plus forts conduisent à de très mauvais résultats et des taux plus faibles n'améliorent pas les résultats tout en perdant beaucoup de temps. Ainsi, en commençant avec un taux de 1%, quelques essais (typiquement 3 ou 4) sont suffisants pour sélectionner une valeur acceptable[4].

[4] De petites valeurs pour B, S et N peuvent être choisies à cette étape, par exemple $B = 20$, $S = 200$, $N = 50$.

2. **N :** Les expérimentations décrites ci-après montrent que la valeur de N a un impact significatif sur la qualité des résultats. De nouveau, quelques essais avec de petites valeurs pour B et S suffisent à déterminer une valeur de N acceptable pour une instance donnée.
3. **S :** Suivant l'étude menée dans [Dimitriou, 2002], les tests décrits plus bas nous permettent de déterminer une longueur de marche suffisamment longue pour explorer les sous-composantes connexes du graphe de recherche
4. **B :** Une fois les paramètres précédents réglés, B peut être augmenté jusqu'à ce qu'une bonne solution soit obtenue.

Bien sûr, cette étude doit être considérée comme une première approche. Nos expérimentations nous laissent à penser que T et N sont faciles à régler. Une meilleure compréhension du comportement de `GWW-idw` nous permettrait de conforter ou de modifier les moyens de régler S et B.

Réglage de N

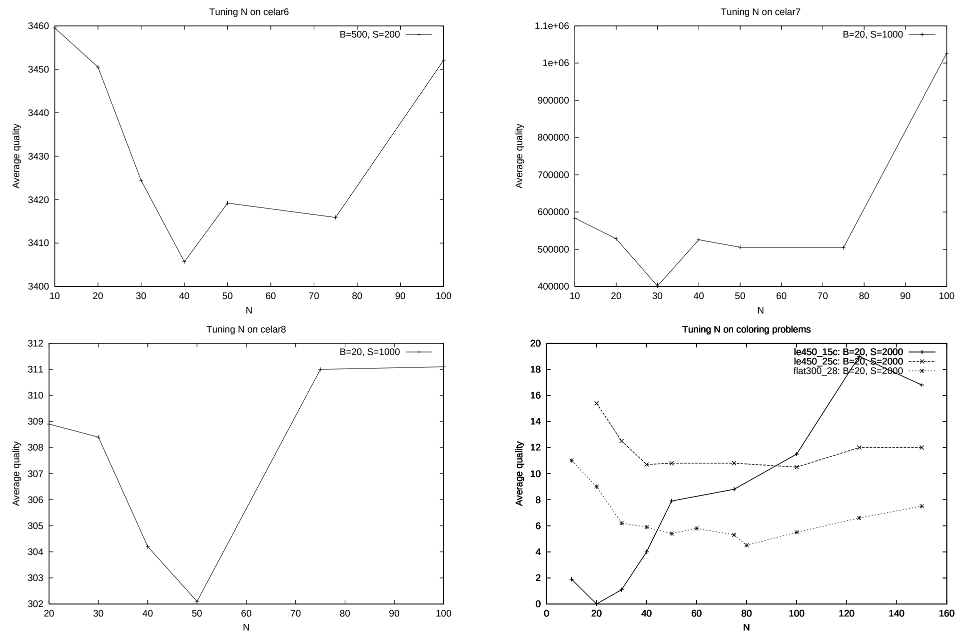

FIG. 7.1 – Réglage du paramètre N. Les valeurs choisies pour les paramètres B et S sont indiquées sur les figures.

La figure 7.1 montre les courbes obtenues pour le réglage du paramètre N sur les différentes instances. La valeur de N est donnée en abscisse et les ordonnées donnent le coût de la meilleure solution (moyenne sur 10 essais). Les conclusions sont les suivantes :

– Il n'est pas difficile de régler N puisque les courbes ont un minimum. En effet, si N est trop petit, les particules ne peuvent pas progresser ; si N est trop grand, les particules ne peuvent pas sortir des minimums locaux.

– Régler N a un impact très significatif sur la performance. En particulier, il permet à `GWW-idw` de colorier toujours en 15 couleurs `le_450_15c` en 2 minutes. Seule l'instance `celar7` ne semble pas très sensible à ce paramètre.

Nous devons mentionner que, pour toutes les instances testées, les courbes obtenues avec différents paramètres B et S ont à peu près la même forme et donnent la même meilleure valeur pour N. Cela semble signifier que pour l'algorithme `GWW-idw`, la valeur de N est intrinsèque à un paysage de recherche donné (instance, voisinage) et ne dépend pas des autres paramètres.

Régler S

Nous avons appliqué la technique décrite dans [Dimitriou, 2002] pour `GWW`. Dans cet article, des tests avaient été effectués avec différentes valeurs de S mais avec une valeur constante de $B \times S$: la valeur de S donnant la meilleure solution (en moyenne) est choisie. Cette approche est justifiée par l'intuition suivante. Dans `GWW`, S doit être suffisamment long pour permettre aux particules d'être uniformément distribuées sur les sous-composantes du graphe de recherche. Ainsi, quand cette longueur “minimum” de marche a été atteinte, il n'y a plus d'intérêt à continuer la marche.

Cette situation est plus compliquée pour `GWW-idw` où S est utilisé pour diffuser les particules dans les composantes connexes mais aussi pour les faire progresser vers de meilleures configurations. De plus, la technique décrite ci-dessus a été appliquée pour `GWW` qui a seulement deux paramètres B et S, alors que notre gestion du seuil introduit un troisième paramètre (monotone) T (si on suppose que N a déjà été réglé et fixé).

Nous avons malgré tout appliqué cette approche de réglage à `GWW-idw`, et obtenu les résultats suivants :

1. Pour les instances du CELAR, le meilleur coût est obtenu quand S est compris entre 200 et 400, et la même valeur est obtenue pour différentes valeurs de $B \times S$. Ceci conduit à adopter une grande valeur de B (plusieurs milliers) et une petite valeur de S (300) dans les tests suivants.
2. Pour les instances de coloriage de graphe, le comportement opposé a été observé. Les meilleurs résultats sont toujours obtenus avec un petit nombre de particules, par exemple 20, et une longue marche.

Ces tests sont une bonne indication de la pertinence de l'utilisation de `GWW-idw`. L'intérêt de gérer plusieurs particules en parallèle est clair dans le premier cas, alors qu'une recherche locale peut obtenir de meilleurs résultats dans le second.

7.5.4 Performance de `GWW-idw`

Le tableau 7.2 rend compte des meilleurs résultats obtenus par `GWW-idw` sur les instances étudiées.

Le graphe `flat300_28` n'a jamais été colorié en 28 couleurs. A notre connaissance, seulement trois algorithmes incomplets ont réussi à le colorier en 31 couleurs : la version distribuée de l'algorithme `Impasse` [Morgenstern, 1996], une recherche avec liste taboue [Dorne et Hao, 1998], et une heuristique hybridant un algorithme génétique avec de la recherche locale [Galinier et Hao, 1999].

[de Givry *et al.*, 1997] et [Koster *et al.*, 1999] décrivent des algorithmes complets basés une décomposition du graphe de contraintes et de la programmation dynamique

	Meilleur algo	borne	temps	GWW-idw	temps	succès	B, S, N
le450_15c	Mo93,CO99	15 coul	mn	15 coul (0.2)	1.1 mn	8/10	20,2000,20
le450_25c	Mo93	25 coul	mn	25 coul (1.4)	2.3 h	1/10	20,800000,100
flat300_28	Mo93,Do98,Ga99	31 coul	heures	31 coul (1.3)	5.9 mn	2/10	20,20000,80
celar6	Gi97,Vo98,Ko99	3389	mn/h	3389 (3405.7)	9.2 mn	4/10	500,200,40
celar7	Vo98,Ko99	343592	mn	343596 (343657)	4.5 h	1/10	5000,300,50
celar8	Vo98,Ko99	262	mn	262 (267.4)	33.7 mn	2/10	1000,200,30

TAB. 7.2 – Meilleurs résultats. Les résultats des meilleurs algorithmes connus sont inscrits sur la partie gauche du tableau ; les résultats de GWW-idw sur la partie droite, les coûts moyens étant entre parenthèses.

qui ont pu résoudre celar6. Les deux autres instances sont réellement difficiles et n'ont jamais été résolues par un algorithme complet [5]. L'algorithme décrit dans [Kolen, 1999] est basé sur un algorithme génétique spécialisé avec un opérateur de mutation codant de la programmation linéaire en nombres entiers. L'algorithme décrit dans [Voudouris et Tsang, 1998] est une recherche locale prenant en compte les conflits pour sélectionner les voisins.

Le tableau 7.2 montre les très bons résultats obtenus par GWW-idw. Les meilleures bornes connues sont obtenues pour toutes les instances testées, sauf pour celar7 où elle est approchée de près[6]. De plus, le temps requis par GWW-idw est souvent très satisfaisant. A notre connaissance, il est parmi les meilleurs algorithmes pour résoudre flat300_28, le450_15c et celar6. Pour les trois autres instances, le temps requis par GWW-idw est plus grand (des heures contre des minutes pour les meilleurs algorithmes). Cependant, on peut noter que GWW-idw n'utilise aucune technique ou modélisation spécifiques au type de problème, contrairement aux algorithmes décrits dans [Morgenstern, 1996; Kolen, 1999].

7.5.5 Comparaisons entre GWW-idw, GWW et la recherche locale

Le tableau 7.3 rassemble les comparaisons entre GWW-idw, GWW et différents algorithmes de recherche locale (Metropolis, le recuit simulé RS, la méthode de la liste taboue tabou [Glover et Laguna, 1997]) sur les instances choisies. La ligne GWW' rend compte des résultats de GWW avec la même gestion du seuil que GWW-idw. La ligne idw donne les résultats obtenus avec l'algorithme de recherche locale idw utilisé seul (en dehors du schéma GWW et donc sans notion de seuil).

Rappelons que tous les algorithmes de recherche locale ont été implantés dans le même logiciel [Neveu et Trombettoni, 2003d] et partagent la plupart du code avec GWW-idw et GWW.

Notre Metropolis est standard. Il commence avec une configuration aléatoire et une marche de longueur S est effectuée comme suit :

– Un nouveau voisin est accepté si son coût est inférieur ou égal au courant.
– Un nouveau voisin dont le coût est plus mauvais que le courant est accepté avec une probabilité dépendant d'une température donnée, constante pour Metropolis [Connolly, 1990].
– Quand aucun voisin n'est accepté, la configuration n'est pas changée dans l'itération courante.

[5]Ceci peut être expliqué par la taille de celar8 et par le critère de celar7.

[6]Une variante de GWW-idw comprenant une recherche locale qui intensifie un peu plus la recherche permet de trouver 343592 2 fois sur 10 essais avec une moyenne de 343676 en 8h.

Le recuit simulé RS suit le même schéma, avec une température diminuant au cours de la recherche. Il a été implanté avec une baisse linéaire de température à partir d'une température initiale donnée en paramètre.

La méthode taboue a été implantée de la manière classique suivante : une liste taboue des derniers mouvements est constituée et on s'interdit de réaffecter une variable avec une valeur qu'on vient d'enlever. Le critère d'aspiration est appliqué quand on a trouvé une configuration meilleure que la meilleure trouvée jusqu'à présent. On choisit comme mouvement le meilleur non tabou dans la portion du voisinage exploré. Les deux paramètres de cet algorithme sont donc la longueur de la liste taboue (fixe) et la taille du voisinage exploré.

L'algorithme de recherche locale `idw` est le même que celui utilisé par chaque particule dans l'hybridation `GWW-idw`. La seule différence est qu'il n'y a plus de seuil et que tous les mouvements sont donc "faisables". On choisit le premier voisin d'un coût inférieur ou égal à la configuration courante si on en trouve un dans une portion du voisinage de taille N, un voisin quelconque sinon.

	le15c	**le25c**	**flat28**	**celar6**	**celar7**	**celar8**
Nb coul	15	25	31			
Temps	2	14	9	14	6	50
Metrop.	5.9 (2)	**3.1** (2)	0.9 **(0)**	5048 (3906)	$6\,10^6$ ($2.9\,10^6$)	410 (300)
RS	9.6 **(0)**	5.8 (4)	1.8 **(0)**	4167 (3539)	$1.2\,10^6$ (456893)	281 (264)
tabou	1.5 **(0)**	3.7 (3)	2.5 (1)	3778 (3616)	$1.2\,10^6$ (620159)	373 (315)
`idw`	**0 (0)**	**3.1 (1)**	**0.8 (0)**	3447 **(3389)**	373334 (343998)	291 (273)
`GWW'`	430 **(0)**	11.8 (9)	3.9 (1)	3648 (3427)	583278 (456968)	272 **(262)**
`GWW-idw`	**0 (0)**	4 (3)	1.3 **(0)**	**3405 (3389)**	**368452 (343600)**	**267 (262)**

TAB. 7.3 – Comparaisons entre algorithmes. La première ligne indique le nombre de couleurs essayées pour les problèmes de coloriage, la deuxième ligne donne le temps (en minutes) alloué par essai. Chaque case contient le coût moyen des solutions (sur 10 ou 20 essais) ; le meilleur coût sur ces essais apparaît entre parenthèses.

Toutes les instances ont été résolues avec le même voisinage. Bien sûr, changer aussi le voisinage peut changer les performances relatives des algorithmes sur une instance donnée, par exemple suivre l'heuristique de Minton de minimisation des conflits pour le choix de valeur améliore le comportement de Metropolis sur `flat300_28`.

Les résultats de nos tests peuvent être interprétés comme suit. D'abord, les résultats de `GWW` standard (avec un seuil baissé de 1 à chaque itération) sont très mauvais et n'ont pas été inscrits dans le tableau. Ensuite, `GWW'` donne de mauvais résultats sur les instances de coloriage de graphe et est toujours plus mauvais que `GWW-idw`. En particulier, pour l'instance `le15c`, la limite des 2mn de temps alloué explique qu'on a dû mettre pour `GWW'` une marche trop courte, empêchant de trouver la solution dans 9 cas sur 10, car toutes les particules sont alors atteintes assez rapidement par le seuil. Ses résultats sont moins mauvais sur les problèmes du CELAR, spécialement pour `celar8`, où la meilleure borne connue (262) est atteinte avec un petit nombre de particules (10).

Les algorithmes de recherche locale ont été utilisés avec leurs propres paramètres (la température pour Metropolis, la température initiale pour le recuit simulé et la taille du voisinage pour `idw`, la taille de la liste taboue et la taille du voisinage pour la méthode taboue) qui ont été réglés pour obtenir les meilleurs résultats possibles.

Metropolis a obtenu de mauvais résultats sur les problèmes du CELAR. Pour `celar6` et `celar7`, ceci peut être expliqué par les poids différents associés aux contraintes et pour lesquels une température unique constante ne serait pas adéquate. Le recuit simulé avec une baisse linéaire de température donne de meilleurs résultats sur ces problèmes. Metropolis semble meilleur sur le coloriage de graphes. Il n'est pas mauvais sur `le450_15c` et bon sur `le450_25c` et `flat300_28`, où il surpasse légèrement `GWW-idw`.

L'algorithme de recherche locale `idw` donne de bons résultats sur l'ensemble des problèmes étudiés. Ceci confirme le fait que pour les problèmes de coloriage de graphe traités, un tel algorithme suffit et la mécanique de `GWW-idw` est alors inutile. Par contre, pour les problèmes du CELAR, `GWW-idw` améliore de manière significative les résultats de `idw`.

Sur ces problèmes, nous avons également vérifié que plusieurs marches `idw` assez courtes menées sans aucune interaction (comme pour GSAT) ne donnaient pas de bons résultats. Ces essais sont dénommés `idw-restart` dans le tableau 7.4. Nous avons pour cela effectué K essais de longueur L avec $K = 20*B$ et $L = S*Nb\_changement\_seuil$ pour donner la même longueur de marche totale par particule et le même temps global que les 20 essais de `GWW-idw` (10 pour celar8). La seule comparaison valide est alors la meilleure solution obtenue durant la recherche. Le paramètre de voisinage de `idw-restart` est celui optimal pour `idw`.

	temps total (mn)	`idw-restart`	`GWW-idw`
`celar6`	280	3420	**3389**
`celar7`	120	364522	**343600**
`celar8`	500	294	**262**

TAB. 7.4 – Comparaisons entre `idw-restart` et `GWW-idw`. La première colonne indique le temps total alloué. Chaque case contient le meilleur coût trouvé.

Pour conclure, `GWW-idw` est le meilleur pour 4 des 6 instances et semble toujours donner de bons résultats (sur les 6 instances étudiées). L'écart type sur les coûts obtenus par `GWW-idw` est généralement très faible, ce qui indique que l'algorithme est robuste et justifie le fait de n'avoir effectué que 10 essais par jeu de paramètres. Finalement, les résultats semblent confirmer qu'utiliser `GWW-idw` est pertinent quand l'utilisateur peut facilement trouver une longueur de marche S minimisant le coût en maintenant constant $B \times S$ (voir fin de 7.5.3).

7.6 Travaux connexes

`GWW` peut être vu comme une version simplifiée (sans opérateur de croisement) d'un algorithme génétique. Cet opérateur est souvent très coûteux et ne permet pas une évaluation incrémentale des coûts dans le cas des problèmes MAX-CSP. Au contraire, les mouvements élémentaires dans `GWW-LS` sont de la recherche locale si bien qu'une évaluation incrémentale des coûts peut facilement être implantée. Certaines études comme [Baluja et Caruana, 1995] avaient déjà essayé d'enlever l'opérateur de croisement dans les algorithmes génétiques. L'hybridation avec de la recherche locale a aussi été essayée avec les algorithmes génétiques et a souvent grandement amélioré leurs résultats : on parle alors d'algorithmes mémétiques [Moscato, 1999].

L'algorithme d'acceptation à seuil [Dueck et Scheuer, 1990] et l'algorithme σ [Carson et Impagliazzo, 1999] sont des métaheuristiques de recherche locale qui utilisent un seuil. Dans ces deux algorithmes, une marche aléatoire est effectuée et un mouvement est accepté si la nouvelle configuration ou le delta ne dépassent pas le seuil courant, qui est réduit tout au long de l'algorithme.

La technique de regroupement *clustering* est principalement utilisée pour de l'optimisation globale sur les réels [Törn, 1973], où elle obtient de très bons résultats. Elle gère une population et des mécanismes existent pour éviter le regroupement de "particules".

7.7 Conclusion

Nous avons conçu un algorithme hybride nommé `GWW-LS` introduisant de la recherche locale dans l'algorithme "Go With the Winners". Une instance de ce schéma, nommé `GWW-idw`, a été défini. `GWW-idw` gère 4 paramètres et une première tentative de réglage a été décrite. La gestion du seuil a été étudiée sur des problèmes réalistes. Le nombre maximum N de voisins visités permet à `GWW-idw` de progresser vers les meilleures solutions tout en sortant des minimums locaux. Ce paramètre est crucial en pratique et semble pouvoir être réglé de manière indépendante.

`GWW-idw` a obtenu de très bons résultats sur le coloriage de graphe et sur les problèmes du CELAR. Seules quelques autres heuristiques d'optimisation peuvent atteindre de telles bornes sur les instances testées. `GWW-idw` n'utilise par ailleurs aucune technique spécifique au type de problème.

Bien sûr, `GWW-idw` devrait être essayé sur d'autres types de problèmes, notamment des instances contenant des contraintes dures.

Les bons résultats expérimentaux obtenus par `GWW-idw` confirment que l'idée utilisée par les algorithmes génétiques consistant à explorer l'espace de recherche en parallèle peut être intéressante. Cependant, le fait que `GWW-idw` ne possède pas d'opérateur de croisement apporte plusieurs avantages :

- L'algorithme est plus facile à régler.
- Puisque le concept de voisinage de la recherche locale est préservé, des structures de données incrémentales peuvent être utilisées quand on passe d'une configuration à son voisin.
- L'algorithme peut être développé dans la même plate-forme logicielle que les algorithmes de recherche locale.

Comme mentionné à la fin de la partie 7.2, la descente gloutonne de `GWW-idw` viole la distribution uniforme des particules dans une région connexe (distribution sur laquelle est bâtie la propriété d'expansion). Ainsi, un point intéressant serait d'établir de manière théorique ou expérimentale une autre condition suffisante de succès de `GWW-idw`, par exemple, la distribution uniforme des particules *à chaque niveau de coût* des composantes connexes. Ceci indiquerait l'intérêt d'utiliser une recherche locale dans `GWW-LS` où les particules ne seraient pas piégées dans les minimums locaux.

Chapitre 8

Détection de parties sur-rigides

Article paru dans la revue électronique JEDAI (volume 2, 2004)
Auteurs : Christophe Jermann, Bertrand Neveu, Gilles Trombettoni

Résumé

Le théorème de Laman permet de caractériser la rigidité des systèmes à barres en 2D. La rigidité structurelle est basée sur une généralisation de ce théorème. Elle est généralement considérée comme une bonne heuristique pour identifier des sous-parties rigides dans les CSP géométriques (GCSP), mais peut en réalité se tromper sur des sous-systèmes très simples car elle ne tient pas compte des propriétés géométriques vérifiées par les objets. Hoffmann *et al.* ont proposé en 1997 des algorithmes à base de flots s'appuyant sur la caractérisation par rigidité structurelle pour répondre aux principales questions liées au concept de rigidité : déterminer si un GCSP est rigide, identifier ses composantes bien-, sur- et sous-rigides, en minimiser la taille, *etc.*.

La rigidité structurelle étendue, une nouvelle caractérisation de la rigidité, a été proposée par Jermann *et al.* en 2002. Elle permet de prendre en compte les propriétés géométriques du GCSP étudié et s'avère ainsi plus fiable. Dans le présent article, nous présentons des algorithmes qui répondent aux principales questions liées à la notion de rigidité en utilisant cette nouvelle caractérisation. Plus précisément, nous montrons que deux modifications de la fonction de distribution de flot utilisée dans les algorithmes de Hoffmann *et al.* permettent l'obtention d'une famille d'algorithmes basés sur la rigidité structurelle étendue. Nous démontrons la correction et la complétude des nouveaux algorithmes et étudions leur complexité en pire cas.

8.1 Introduction

Les problèmes géométriques apparaissent dans de nombreuses applications pratiques : la CAO, la robotique et la biologie moléculaire en sont trois exemples. Le paradigme de la programmation par contraintes (PPC) consiste à modéliser les problèmes de façon déclarative sous forme de problèmes de satisfaction de contraintes (CSP) et à utiliser des outils génériques pour résoudre ces CSP. L'application de ce paradigme aux problèmes géométriques permet de les considérer comme des problèmes de satisfaction de contraintes géométriques (GCSP).

Dans un GCSP, les contraintes (distances, angles, incidences, ...) visent à restreindre les positions, orientations et dimensions que peuvent adopter des objets géométriques (points, droites, plans, ...). Un GCSP a généralement pour vocation d'être résolu, afin de déterminer des positions, orientations et dimensions de tous les objets qui satisfont les contraintes. Cependant, des questions d'ordre qualitatif peuvent se poser avant de résoudre un GCSP : le système modélisé est-il indéformable ? Sinon, quelles sont les déformations qu'il admet ? Admet-il des solutions, et sinon pourquoi ? Ces questions apparaissent souvent dans les domaines que nous avons cités, où un concepteur agissant plus au niveau géométrique qu'au niveau CSP, souhaite connaître a priori les propriétés du système qu'il a modélisé.

On a alors recours au concept géométrique de rigidité et à différentes caractérisations de ce concept pour essayer de répondre à ces questions. Intuitivement, les sous-parties rigide d'un GCSP sont indéformables, alors que ses sous-parties sous-rigides (sous-déterminées) présentent des déformations et ses sous-parties sur-rigides (sur-déterminées) fournissent des explications à l'absence de solution globale.

Ces informations qualitatives sont également souvent utilisées de façon implicite ou explicite dans les résolveurs dédiés aux GCSP [Kramer, 1992; Bouma *et al.*, 1995; Dufourd *et al.*, 1998; Lamure et Michelucci, 1998; Hoffmann *et al.*, 2001; Jermann *et al.*, 2000]. En particulier, les méthodes de décomposition géométriques ont pour but de produire des séquences de sous-GCSP rigides pouvant être résolus séparément puis assemblés. Elles nécessitent des algorithmes efficaces permettant d'identifier de petits sous-GCSP rigides dont les solutions partielles peuvent être assemblées pour reconstituer une solution globale du GCSP initial.

Le concept de rigidité dispose donc d'un statut central dans l'analyse et la résolution efficace de GCSP. La principale problématique est alors de concevoir des caractérisations de ce concept qui soient suffisamment fiables et puissent donner naissance à des algorithmes efficaces répondant aux questions : un GCSP est-il rigide ? Quelles sont ses sous-parties bien-, sur- et sous-rigides ? Peut-on identifier de telles sous-parties de taille minimale ?

Les méthodes de caractérisation de rigidité peuvent être classées en deux catégories : les *approches à base de règles* [Bouma *et al.*, 1995; Kramer, 1992] utilisent un répertoire de formes rigides connues qui ne peut couvrir tous les cas de figure, alors que les *approches structurelles* [Hoffmann *et al.*, 1997; Lamure et Michelucci, 1998] utilisent des algorithmes de flots (ou de couplage maximum) pour vérifier une propriété appelée *rigidité structurelle*, basée sur un compte des degrés de liberté dans le GCSP.

Les approches structurelles sont plus générales puisqu'elles peuvent être appliquées à toute classe de GCSP (tous types d'objets, tous types de contraintes). Cependant, la rigidité structurelle n'est qu'une approximation de la rigidité dans le cas général, c-a-d. qu'il existe des GCSP mal caractérisés par cette propriété. Plusieurs heuristiques sont alors employées pour assister cette propriété, mais aucune ne permet d'éliminer tous les cas d'erreur.

Dans [Jermann *et al.*, 2004a], nous avons proposé une nouvelle caractérisation de la rigidité, appelée *rigidité structurelle étendue*, qui subsume la rigidité structurelle, et correspond même exactement à la rigidité pour des GCSP exempts de contraintes redondantes.

Dans le présent article, nous proposons de nouveaux algorithmes basés sur notre nouvelle caractérisation de la rigidité pour répondre aux principales questions liées au concept de rigidité. Ces nouveaux algorithmes découlent naturellement de deux modifications principales dans la fonction de distribution de flots utilisée par les algorithmes de Hoffmann *et al.* [Hoffmann *et al.*, 1997].

Après les définitions données en section 8.2, nous introduisons le principe de la caractérisation de la rigidité à base de flots et présentons brièvement les algorithmes proposés par Hoffmann *et al.* (section 8.3). Finalement, nous présentons les deux modifications que nous proposons dans la fonction `Distribute` et la famille d'algorithmes qui en découle (section 8.4). Pour chaque algorithme, nous analysons ses propriétés : correction, complétude et complexité.

8.2 Définitions

Dans cette section, nous donnons les définitions nécessaires à la compréhension de cet article.

8.2.1 Problème de satisfaction de contraintes géométriques

Définition 1 GCSP
Un **problème de satisfaction de contraintes géométriques** *(GCSP)* $S = (O, C)$ *est composé d'un ensemble* O *d'objets géométriques et d'un ensemble* C *de contraintes géométriques.*
$S' = (O', C')$ *est un* **sous-GCSP** *de* $S = (O, C)$*, noté* $S' \subset S$*, ssi* $O' \subset O$ *et* $C' = \{c \in C | c$ *ne porte que sur des objets dans* $O'\}$*, c-à-d. que* S' *est induit par* O'*.*

Les objets géométriques usuels sont les points, les droites et, en 3D, les plans. On peut aussi considérer des objets géométriques plus complexes, tels que des cercles, des coniques, des parallélépipèdes, *etc.*. Les paramètres d'un objet géométrique de type donné définisent sa position, son orientation et ses dimensions.

Les contraintes géométriques sont, par exemple, des distances, des angles, des incidences, des parallélismes, des symétries, des alignements, *etc.*. Les contraintes ont pour effet de restreindre l'ensemble des positions, orientations et dimensions que peuvent prendre les objets géométriques du GCSP.

Une solution d'un GCSP est la donnée d'une position, d'une orientation et d'un jeu de dimensions pour chaque objet géométrique de telle sorte que toutes les contraintes géométriques soient satisfaites.

cm

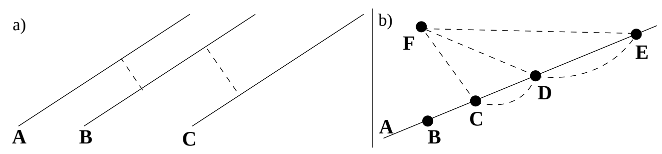

FIG. 8.1 – Deux exemples de GCSP

La figure 8.1-a présente un GCSP en 2D constitué de 3 droites liées par 2 parallélismes et deux distances droite-droite. La figure 8.1-b présente un GCSP en 3D constitué d'une droite et 5 points, liés par 4 incidences point-droite et 5 distances point-point.

Pour être résolu, un GCSP est généralement transformé en un système d'équations, chaque objet étant représenté par des variables définissant sa position et son orientation, et chaque contrainte devenant un sous-système d'équations sur les variables des objets qu'elle contraint.

Hypothèses : Nous supposerons que les objets géométriques sont indéformables (pas de cercle à rayon variable par exemple) et que les contraintes ne peuvent porter que sur les positions et orientations relatives des objets (pas de fixation par rapport au repère global par exemple). Ces limitations simplifient la définition des caractérisations structurelles de la rigidité et sont nécessaires aux méthodes de résolution de GCSP basées sur la rigidité.

Sous ces hypothèses, une solution d'un GCSP est composée d'une position et d'une orientation pour chacun de ses objets qui satisfont toutes ses contraintes.

8.2.2 Rigidité

La rigidité d'un GCSP se définit à partir des mouvements que celui-ci admet[1]. On distingue deux types de mouvements : les déformations, qui ne préservent pas les positions et orientations relatives des objets, et les déplacements (rotations et translations) qui les préservent. Intuitivement, un GCSP est **bien-rigide** s'il n'admet aucune déformation et admet tous les déplacements de l'espace géométrique considéré[2]. Un GCSP qui admet des déformations est dit **sous-rigide**, alors qu'un GCSP n'admettant pas certains déplacements, ou aucune solution, est dit **sur-rigide**. Des définitions plus formelles peuvent être trouvées dans [Jermann, 2002].

Dans l'exemple présenté en figure 8.1-b, le sous-GCSP CDF est rigide : un triangle est indéformable et admet tous les déplacements de l'espace ; AF est sous-rigide puisque la droite A et le point F ne sont liés par aucune contrainte ; le sous-GCSP $ACDEF$ est quant à lui sur-rigide : il est génériquement impossible de placer le point F à l'intersection des 3 sphères (contraintes de distance) dont les centres C, D et E sont alignés.

8.2.3 Rigidité structurelle

La **rigidité structurelle** correspond à une analyse des **degrés de liberté** (DDL) dans le GCSP. Intuitivement, un DDL représente un mouvement indépendant dans le GCSP. Plus formellement :

Définition 2 Degré de liberté (DDL)
- Objet o : DDL(o) est le nombre de variables indépendantes définissant la position et l'orientation de o.

[1]En réalité, la rigidité et les mouvements se définissent au niveau de chaque solution d'un GCSP. Cependant, comme c'est généralement le cas en CAO, on considère la rigidité au niveau du GCSP comme représentant celle de toutes ses solutions car on souhaite généralement étudier des systèmes non résolus afin d'en déduire des propriétés générales.

[2]Cette seconde condition est toujours vérifiée sous l'hypothèse que nous avons posée sur les contraintes géométriques.

- *Contrainte c : DDL(c) est le nombre d'équations indépendantes représentant la contrainte c.*
- *GCSP* $S = (O, C)$ *:* $DDL(S)=\sum_O DDL(o)$ - $\sum_C DDL(c)$.

En 3D, un point possède 3 DDL et une droite 4 ; l'incidence point-droite retire 2 DDL, et une distance point-point 1. Ainsi, les sous-GCSP ACD, CDF et AF issus de la figure 8.1-b ont respectivement 5, 6 et 7 DDL.

La rigidité structurelle est une généralisation du théorème de Laman [Laman, 1970] qui caractérise la rigidité générique des systèmes à barres en 2D. Elle est basée sur l'intuition suivante : si un GCSP admet moins (resp. plus) de mouvements que le nombre de déplacements (égal à $\frac{d(d+1)}{2}$ en dimension d) admis par l'espace géométrique qui le contient, alors il est sur-(resp. sous-)rigide. Plus formellement :

Définition 3 Rigidité structurelle (s_rigidité)
Un GCSP $S = (O, C)$ *en dimension* d *est* **s_rigide** *ssi* $DDL(S) = \frac{d(d+1)}{2}$ *et* S *ne contient pas de sous-GCSP sur-s_rigide.*
S *est* sous-s_rigide *ssi* $DDL(S) > \frac{d(d+1)}{2}$ *et* S *ne contient pas de sous-GCSP sur-s_rigide.*
S *est* sur-s_rigide *ssi* $\exists S' \subseteq S$ *tel que* $DDL(S') < \frac{d(d+1)}{2}$.

En pratique, la rigidité structurelle est considérée comme une bonne approximation de la rigidité [Lamure et Michelucci, 1998; Hoffmann *et al.*, 1997]. L'écart entre s_rigidité et rigidité est en réalité important (cf. [Jermann *et al.*, 2004a; Jermann, 2002]). Nous illustrons ici cet écart sur 2 sous-GCSP issus de la figure 8.1-b : $ABCD$ est s_rigide en 3D puisque DDL($ABCD$)=6, alors que ce sous-GCSP est en réalité sous-rigide : le point B peut bouger indépendamment des points C et D sur la droite A. $ACDE$ est quant à lui sur-s_rigide car DDL($ACDE$)=5, mais il s'agit en réalité d'un sous-GCSP bien-rigide.

8.2.4 Rigidité structurelle étendue

La rigidité structurelle étendue (notée es_rigidité) est basée sur le concept de *degré de rigidité* (DDR). Le degré de rigidité d'un GCSP représente le nombre de déplacements admis par celui-ci. Ce nombre dépend des propriétés géométriques vérifiées par les objets du GCSP. Par exemple, le DDR d'une paire de droites en 2D est 3 si ces droites sont quelconques, alors qu'il vaut 2 si elles sont parallèles ; le parallélisme de ces droites peut être explicite (contrainte entre les droites) mais peut aussi être induit par l'ensemble des contraintes du GCSP contenant ces droites. Dans ce dernier cas, inférer le parallélisme (et donc déterminer le DDR) est équivalent à la preuve de théorème géométrique.

Le principe de la rigidité structurelle étendue est de comparer le nombre de DDL d'un GCSP au DDR de ce même GCSP, c-a-d. le nombre de mouvements au nombre de déplacements. On peut ainsi déterminer si un GCSP admet ou non des déformations.

Définition 4 Rigidité structurelle étendue (es_rigidité)
Un GCSP $S = (O, C)$ *est* **es_rigide** *ssi DDL(S)=DDR(S) et* S *n'est pas sur-es_rigide.*
S *est* sous-es_rigide *ssi* $DDL(S) > DDR(S)$ *et* S *n'est pas sur-es_rigide.*
S *est* sur-es_rigide *ssi* $\exists S' \subseteq S$, $DDL(S') < DDR(S')$.

La es_rigidité subsume la s_rigidité ; par exemple, es_rigidité et rigidité correspondent exactement sur tout sous-GCSP de la figure 8.1. Cependant, elle demeure une approximation de la rigidité dans le cas général : elle peut être trompée par la présence de redondances dans un GCSP, et il existe des GCSP dont les solutions n'ont pas toutes la même rigidité, et ne peuvent être caractérisés par aucune caractérisation *a priori*. Enfin, elle pose le problème du calcul du degré de rigidité, qui requiert la détermination des propriétés géométriques (incidences, parallélismes, ...) induites par les contraintes du GCSP et équivaut généralement à la preuve de théorème en géométrie. Nous renvoyons le lecteur à [Jermann *et al.*, 2004a] pour plus de détails sur cette nouvelle caractérisation et une comparaison plus complète avec la rigidité structurelle.

8.3 Caractérisation par rigidité structurelle

La caractérisation structurelle de la rigidité s'appuie sur une représentation du GCSP considéré sous la forme d'un réseau dans lequel une distribution de flots correspond à une distribution des DDL des contraintes sur les DDL des objets.

8.3.1 Réseau Objet-Contrainte

Un GCSP $S = (O, C)$ peut être représenté par un réseau $G = (s, V, t, E, w)$ appelé *réseau objets-contraintes* (introduit dans [Hoffmann *et al.*, 1997]). La figure 8.2-a représente le réseau objets-contraintes correspondant au GCSP de la figure 8.1-a.

cm

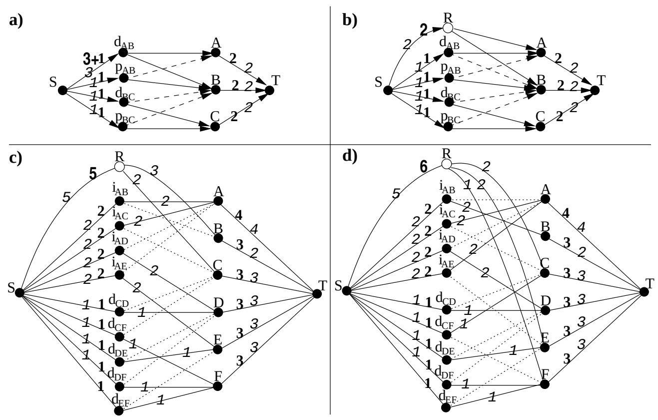

FIG. 8.2 – Réseaux objets-contraintes et distributions de flots par les algorithmes `Dense` et `Over-Rigid`

Définition 5 Réseau objets-contraintes (s, V, t, E, w)

- s est la source et t est le puits.
- Chaque objet $o \in O$ est représenté par un nœud-objet $v_o \in V$.
- Chaque contrainte $c \in C$ est représentée par un nœud-contrainte $v_c \in V$.
- Pour chaque objet $o \in O$, on ajoute un arc $(v_o \to t) \in E$ de capacité $w(v_o \to t)$ =DDL(o).
- Pour chaque contrainte $c \in C$, on ajoute un arc $(s \to v_c) \in E$ de capacité $w(s \to$

v_c) =DDL(c).
- Pour chaque objet $o \in O$ sur lequel porte une contrainte $c \in C$, il existe un arc $v_c \rightarrow v_o$ de capacité $w(v_c \rightarrow v_o) = \infty$ dans E.

Par définition, un flot dans ce réseau représente bien une distribution des DDL des contraintes sur les DDL des objets ; c'est ce principe qui est utilisé pour la caractérisation structurelle de la rigidité.

8.3.2 Principe de caractérisation par un flot

Du point de vue géométrique, la caractérisation de la rigidité s'effectue en vérifiant qu'un GCSP n'admet que des déplacements. Sous cet angle, la détection de rigidité à base de flots peut être expliquée comme suit :

1. retirer K déplacements du GCSP en introduisant K DDL supplémentaires du côté des contraintes ;
2. vérifier si un sous-GCSP sur-contraint S' existe en calculant un flot maximum dans le réseau objets-contraintes surchargé ;
3. si tel est le cas, alors S' vérifie DDL(S')$< K$.

En effet, nous avons déjà expliqué qu'un flot dans le réseau objets-contraintes représente une distribution des DDL des contraintes sur les DDL des objets, ce qui revient intuitivement à choisir quelles contraintes résolvent quels objets. Un flot maximum représente donc une distribution *optimale* des DDL dans le GCSP. Si un flot maximum ne peut saturer tous les arcs issus de la source du réseau, cela signifie que certains DDL des contraintes ne peuvent être distribués sur les objets (cf. définition 5). Il existe alors obligatoirement un sous-GCSP sur-contraint dans le GCSP.

La détection structurelle de rigidité exploite cette propriété pour identifier un sous-GCSP S' qui vérifie DDL(S')$< K$ pour un K donné. Pour ce faire, K DDL additionnels sont artificiellement introduits dans le réseau en provenance de la source. Selon le principe expliqué ci-dessus, si le calcul d'un flot maximum ne permet pas de saturer tous les arcs issus de la source cela signifie que les objets ne peuvent pas *absorber* les DDL des contraintes plus ces K DDL additionnels. S' vérifie alors nécessairement DDL(S')$< K$. [Hoffmann *et al.*, 1997] ont montré que S' est alors induit par l'ensemble des nœuds-objets traversés durant la dernière recherche d'un chemin augmentant dans le réseau : S' est induit par l'ensemble de tous les nœuds-objets atteignables par un parcours du graphe résiduel partant de la source.

En choisissant la valeur adéquate pour K, [Hoffmann *et al.*, 1997] ont dérivé des algorithmes pour identifier des sous-GCSP s_rigides ($K = \frac{d(d+1)}{2} + 1$) ou sur-s_rigides ($K = \frac{d(d+1)}{2}$).

8.3.3 Fonction `Distribute`

La fonction `Distribute` est la mise en application algorithmique du principe de caractérisation structurelle de la rigidité à base de flots. Cette fonction, proposée dans [Hoffmann *et al.*, 1997], applique la surcharge K en augmentant simplement la capacité de l'arc liant la source à une contrainte c dans le réseau, c-a-d. en ajoutant K au nombre de DDL de la contrainte c. Ceci a pour but de retirer K déplacements au sous-GCSP induit par les objets liés à c.

Requiert: $K > 0$, $S = (O, C)$, $c \in C$
Assure: $S' \subset S$ vérifie DDL(S')$< K$, ou S' est vide
$G \leftarrow$ `Réseau-Surchargé`(S, K, c)
$V \leftarrow$ `Ford-Fulkerson`(G)
$S' \leftarrow$ `Sous-GCSP-Induit`(V, S)
Retourne S'

Algorithm 4: Distribute (S : un GCSP ; K : un entier ; c : une contrainte) **retourne** S' : un GCSP

La fonction `Réseau-Surchargé` retourne le réseau objets-contraintes correspondant au GCSP S dans lequel la contrainte c a été surchargée, c-a-d. que l'arc $s \to c$ liant la source s à la contrainte c est de capacité DDL(c)+K. Le calcul du flot maximum dans ce réseau est effectué par la fonction `Ford-Fulkerson` qui applique l'algorithme classique de [Ford et Fulkerson, 1962]. Cette fonction retourne l'ensemble V des nœuds-objets traversés lors de la dernière recherche d'un chemin augmentant, cet ensemble étant vide si le flot maximum sature tous les arcs issus de la source s. Enfin, la fonction `Sous-GCSP-Induit` retourne le sous-GCSP S' induit par V. Selon la preuve de Hoffmann *et al.*, S' vérifie DDL(S')$< K$ ou S' est vide.

Exemple d'application de `Distribute` : La figure 8.2-a illustre l'appel `Distribute(`$S, 3, dAB$`)` sur le GCSP S de la figure 8.1-a. Un flot maximum est représenté sur le réseau par les chiffres en gras. Comme l'arc $s \to dAB$ n'est pas saturé dans ce flot maximum, la fonction retourne le sous-GCSP induit par le sous-ensemble d'objets atteignables dans le graphe résiduel en partant de dAB ; il s'agit du sous-GCSP AB, qui dispose effectivement de moins de $K = 3$ DDL.

Complexité de `Distribute` : La complexité de la fonction `Distribute` est dominée par celle de la fonction `FordFulkerson` qui effectue le calcul du flot maximum. En effet, la production du réseau objets-contraintes surchargé et l'extraction du sous-GCSP induit par V peuvent être effectués en temps linéaire, alors que le calcul d'un flot maximum est en $O(N^2(N + M))$, N étant le nombre de nœuds et M le nombre d'arcs dans le réseau. Notons n le nombre d'objets géométriques et m le nombre de contraintes géométriques ; en supposant que les contraintes géométriques considérées sont d'arité bornée[3], on obtient $N = n+m$ et $M \sim n$. Il en résulte que la complexité de la fonction `Distribute` exprimée en nombre d'objets et de contraintes est $O(n * (n + m)^2)$.

8.3.4 Algorithmes de Hoffmann *et al.*

`Dense` et `Minimal-Dense` sont deux algorithmes proposés par [Hoffmann *et al.*, 1997] reposent entièrement sur la fonction `Distribute`.

[3]En pratique, les contraintes géométriques sont toujours d'arité bornée ; même une contrainte d'égalité de distance sera d'arité au plus 4.

8.3.4.1 Algorithme `Dense`

L'algorithme `Dense` a pour but d'identifier un sous-GCSP bien ou sur-s_rigide s'il en existe un dans le GCSP passé en paramètre. Pour ce faire, il consiste simplement à appliquer séquentiellement la fonction Distribute à chaque contrainte du GCSP considéré, avec une surcharge valant à chaque fois $K = \frac{d(d+1)}{2} + 1$. La séquence se termine lorsque l'appel courant retourne un sous-GCSP S' non vide. Par définition de la fonction `Distribute`, S' vérifie alors DDL(S')$< \frac{d(d+1)}{2} + 1$, c-a-d. que S' est bien- ou sur-s_rigide (cf. définition 3). Notons que cet algorithme peut être utilisé pour identifier des sous-GCSP sur-s_rigides uniquement en changeant simplement la valeur de la surcharge en $K = \frac{d(d+1)}{2}$.

Requiert: $d > 0$ représente la dimension de l'espace géométrique contenant S
Assure: $S' \subset S$ vérifie DDL(S')$< \frac{d(d+1)}{2}$, ou S' est vide
  $Contraintes \leftarrow C$
  $S' \leftarrow EmptyGCSP$
  Tant que $Contraintes \neq \emptyset \wedge S' = EmptyGCSP$ **Faire**
    $c \leftarrow$ `Pop(`$Contraintes$`)`
    $S' \leftarrow$ `Distribute(`$S, \frac{d(d+1)}{2}, c$`)`
  Fin Tant que
  Retourne S'

Algorithm 5: Dense ($S = (O, C)$: un GCSP ; d : un entier) **retourne** S' : un GCSP)

Exemple d'application de `Dense` : La figure 8.2-a illustre aussi l'application de l'algorithme `Dense` sur le GCSP de la figure 8.1-a : `Dense(`$S, 2$`)`. En effet, cet algorithme va immédiatement effectuer l'appel `Distribute(`$S, 4, dAB$`)`. Comme nous l'avons vu précédemment, `Distribute(`$S, 3, dAB$`)` retourne le sous-GCSP AB, et il en va de même pour l'algorithme `Dense` : la surcharge $K = 4$ étant supérieure à $K = 3$, le flot maximum ne pourra pas non plus saturer l'arc $s \rightarrow dAB$ liant la source à la contrainte considérée.

Le sous-GCSP AB est donc identifié bien- ou sur-s_rigide. Il est en réalité sur-s_rigide, mais il s'agit de l'un des cas où la s_rigidité caractérise mal la rigidité, comme nous l'avons vu à la section 8.2.3.

Complexité de `Dense` : La complexité de l'algorithme `Dense` dépend directement de celle de la fonction `Distribute` puisque cet algorithme consiste à appeler cette fonction au plus une fois par contrainte du GCSP. Il est donc en $O(m * n * (n + m)^2)$. Il est à noter que l'algorithme présenté dans [Hoffmann *et al.*, 1997] s'applique sur un réseau construit incrémentalement, ce qui conduit à une complexité pratique moindre puisque l'algorithme termine toujours au plus tôt. Par ailleurs, d'autres optimisations peuvent encore abaisser le coût de cet algorithme. Par exemple, pré-calculer un flot maximum du réseau non-surchargé une seule fois avant le premier appel à l'algorithme `Dense` permet d'avoir seulement à mettre à jour ce réseau pour chaque surcharge introduite, ce qui abaisse la complexité de la fonction FordFulkerson d'un facteur linéaire.

8.3.4.2 Algorithme `Minimal-Dense`

L'algorithme `Minimal-Dense` a pour but de retourner un sous-GCSP bien- ou sur-s_rigide minimal au sens de l'inclusion, c-a-d ne contenant aucun sous-GCSP propre bien- ou sur-s_rigide. Pour ce faire, l'algorithme commence par appliquer l'algorithme `Dense` afin d'identifier un sous-GCSP S' bien- ou sur-s_rigide. Il applique ensuite une étape de minimisation qui consiste à essayer de retirer les objets de S' un à un. Après le retrait d'un objet, un appel à l'algorithme `Dense` permet de tester s'il existe toujours un sous-GCSP bien- ou sur-s_rigide, auquel cas le processus se poursuit sur ce nouveau sous-GCSP strictement plus petit ; autrement, l'objet retiré était nécessaire à l'existence d'un sous-GCSP bien- ou sur-s_rigide et doit donc être conservé.

Requiert: $d > 0$ représente la dimension de l'espace géométrique contenant S
Assure: $S' \subset S$ vérifie DDL(S')$< \frac{d(d+1)}{2}$ et $\forall S'' \subsetneq S'$, DDL($S''$)$> \frac{d(d+1)}{2}$, ou S' est vide
  $S' \leftarrow$ `Dense`(S, d)
  $Necessaires \leftarrow \emptyset$
  $O' \leftarrow$ `Objets`(S')
  Pour chaque $o \in O' \setminus Necessaires$ **Faire**
    $S'' \leftarrow$ `Sous-GCSP-Induit`$(O' \setminus \{o\}, S')$ $\{$*S'' est le sous-GCSP de S' ne contenant pas l'objet o*$\}$
    $S'' \leftarrow$ `Dense`(S'', d)
    Si $S'' \neq EmptyGCSP$ **Alors**
      $S' \leftarrow S''$ $\{$*S'' est dense, donc o n'est pas nécessaire ; le sous-GCSP S' devient le nouveau sous-GCSP S'' identifié*$\}$
    Sinon
      $Necessaires \leftarrow Necessaires \cup \{o\}$ $\{$*o est nécessaire ; il est conservé afin de ne pas être testé à nouveau*$\}$
    Fin Si
  Fin Pour
  Retourne S'

Algorithm 6: Minimal-Dense ($S = (O, C)$: un GCSP ; d : un entier) **retourne** S' : un GCSP)

Exemple d'application de `Minimal-Dense` : appliqué au GCSP de la figure 8.1-a, cet algorithme retournera également le sous-GCSP AB. En effet, le premier appel à l'algorithme `Dense` retournera AB. Le retrait de A donne alors un sous-GCSP vide de contraintes, qui n'est donc pas bien- ou sur-s_rigide ; le retrait de B également. A et B étant nécessaires, le sous-GCSP minimisé est bien AB.

Complexité de `Minimal-Dense` : La complexité de l'algorithme dépend directement de celle de l'algorithme `Dense` puisqu'il consiste à appliquer au plus une fois cet algorithme pour chaque objet du sous-GCSP identifié par le premier appel à `Dense`, qui est au pire le GCSP initial. Elle est en $O(n*m*n*(n+m)^2)$. Là encore, les optimisations de l'algorithme `Dense` permettent d'obtenir une complexité pratique inférieure à cette complexité en pire cas.

8.4 Nouveaux Algorithmes

Nos nouveaux algorithmes, tout comme les algorithmes proposés par [Hoffmann *et al.*, 1997], sont basés sur un calcul de flot maximum dans le réseau objets-contraintes. Cependant, ils présentent deux différences principales :

- ils utilisent la es_rigidité au lieu de la s_rigidité ;
- ils effectuent la distribution du flot de façon *géométriquement correcte*.

Ces deux différences sont obtenues au moyen de deux modifications majeures de la fonction `Distribute` dans la façon d'appliquer une surcharge dans le réseau objets-contraintes :

1. La valeur de la surcharge appliquée dans le réseau dépend des propriétés géométriques vérifiées par les objets au lieu d'être constante.
2. La surcharge est appliquée via un nœud R dédié à cet effet, au lieu d'être appliquée sur chaque contrainte.

8.4.1 Nouvelle fonction `Distribute`

Comme nous l'avons vu dans la section précédente, le principe de la caractérisation structurelle de la rigidité est de fixer arbitrairement K déplacements et de vérifier l'existence d'un sous-GCSP bien-contraint. La fonction `Distribute` proposée par Hoffmann *et al.* applique ce principe en fixant les K déplacements via une contrainte, c-a-d. en fixant un repère local sur les objets géométriques liés à une seule contrainte. Cependant, retirer K déplacements à un sous-ensemble O' d'objets n'est géométriquement correct que si le sous-GCSP S' qu'il induit possède au moins K déplacements, c-a-d. qu'il vérifie DDR(S')$\geq K$.[4]

Considérons par exemple un segment en 3D, c-a-d. un sous-GCSP constitué de 2 points liés par une contrainte de distance. Ce sous-GCSP n'admet que 5 des 6 (3 rotations + 3 translations) déplacements autorisés par l'espace 3D puisque la rotation selon l'axe défini par le segment est sans effet sur celui-ci. Ainsi, retirer 6 déplacements à un segment 3D pour vérifier s'il est rigide est géométriquement incorrect et produit assurément un résultat erroné. C'est pourtant ce que fait la fonction `Distribute` proposée par Hoffmann *et al.* lorsqu'elle applique une surcharge $K = 6 + 1$ sur une contrainte de distance liant deux points dans un GCSP en 3D.

De façon à appliquer le principe de détection structurelle de rigidité de façon géométriquement correcte, nous proposons d'introduire une contrainte virtuelle R possédant K DDL. Cette contrainte, qui représente le fait de fixer K déplacements sur un ensemble d'objets, ne sera liée qu'aux sous-GCSP S' vérifiant DDR(S')$\geq K$. K et S' sont les deux paramètres d'entrée de notre version de la fonction `Distribute`.

La fonction `Réseau-Surchargé` est un peu différente de celle utilisée par Hoffmann *et al.*. Elle retourne le réseau objets-contraintes correspondant au GCSP S dans lequel la contrainte R a été introduite, ainsi que les arcs $s \rightarrow R$ et $R \rightarrow o$, pour chaque $o \in S'$, de capacités respectives K et $+\infty$. Le calcul du flot maximum dans ce réseau est toujours effectué par la fonction `Ford-Fulkerson`, qui retourne toujours l'ensemble V des nœuds-objets traversés lors de la dernière recherche d'un chemin augmentant, cet ensemble étant vide si le flot maximum sature tous les arcs issus de la source s. Selon la preuve de Hoffmann *et al.*, S'' vérifie DDL(S'')$< K$ ou S'' est vide.

[4]Rappelons que le DDR représente le nombre de déplacements admis par un sous-GCSP.

Requiert: $K > 0$, $S' \subset S$ vérifie DDR(S')$\geq K$
Assure: $S'' \subset S$ vérifie DDL(S'')$< K$, ou S'' est vide
$G \leftarrow$ `Réseau-Surchargé`(S, K, S')
$V \leftarrow$ `Ford-Fulkerson`(G)
$S'' \leftarrow$ `Sous-GCSP-Induit`(V, S)
Retourne S''

Algorithm 7: Distribute (S : GCSP ; K : entier ; S' : GCSP) **retourne** S'' : GCSP)

Nos deux modifications de la fonction `Distribute` permettent d'une part d'appliquer la surcharge sur n'importe quel sous-GCSP, et d'autre part de ne distribuer la surcharge que vers les sous-GCSP pour lesquels cette opération est géométriquement correcte.

Exemple d'application de `Distribute` : Toujours sur le GCSP de la figure 8.1-a, notre fonction `Distribute` procède différemment de celle de Hoffmann *et al.* : puisque DDR(AB)=2, la surcharge maximale applicable sur ce sous-GCSP est $K = 2$. La figure 8.2-b présente l'appel `Distribute(`$S, 2, AB$`)` pour la nouvelle version de la fonction. Le flot maximum parvient cette fois à saturer tous les arcs issus de la source et aucun sous-GCSP n'est donc retourné. Ceci est à la fois correct du point de vue structurel puisque DDL(AB) n'est pas strictement inférieur à $K = 2$, et du point de vue géométrique puisque ce sous-GCSP n'est pas sur-rigide.

Complexité de `Distribute` : La complexité de la nouvelle fonction `Distribute` est exactement la même que celle de la fonction initialement proposée par Hoffmann *et al.*. En effet, la seule différence réside dans la construction du réseau, et nos modifications se font en temps linéaire dans la taille du sous-GCSP à lier à la contrainte virtuelle R. Cette étape de construction est toujours dominée par l'étape de calcul du flot maximum, et la fonction demeure en $O(n * (n + m)^2)$, complexité en pire cas qui peut toujours être améliorée par les heuristiques disponibles pour la version initiale de cette fonction.

8.4.2 Algorithmes pour la détection de rigidité

A partir de la fonction `Distribute`, Hoffmann *et al.* dérivaient les algorithmes `Dense` et `Minimal-Dense` qui permettent d'identifier des sous-GCSP bien- ou sur-rigides et de les minimiser (en nombre d'objets).

Ces algorithmes sont entièrement reproductibles avec la nouvelle fonction `Distribute`. Ceci nous permet de répondre aux mêmes problèmes mais de façon géométriquement correcte et avec une meilleure caractérisation de la rigidité : la rigidité structurelle étendue

8.4.2.1 Détection de sous-GCSP sur-es_rigides

Schématiquement, l'algorithme `Dense` effectue des appels à la fonction `Distribute` pour chaque contrainte présente dans le GCSP considéré, et ce jusqu'à ce que cette fonction retourne un GCSP non-vide ou jusqu'à ce que toutes les contraintes aient été traitées. Dans cet algorithme, la surcharge est basée uniquement sur la dimension de l'espace géométrique considéré ; elle représente le nombre maximum de déplacements indépendants : 6 en 3D et 3 en 2D. L'algorithme `Dense` retourne donc des sous-GCSP sur-s_rigides[5] puisque les GCSP retournés n'admettent pas certains déplacements de l'espace géométrique considéré. Comme nous l'avons expliqué, cet algorithme est en réalité incorrect puisqu'il peut retirer parfois plus de déplacements que n'en admet le sous-GCSP lié à la contrainte surchargée.

Pour obtenir un algorithme correct, nous proposons d'utiliser la définition de la es_rigidité et notre fonction `Distribute` : la surcharge passée en paramètre à la fonction Distribute sera le DDR du sous-GCSP sur lequel on applique la surcharge.

Ceci nous permet de définir le nouvel algorithme `Over-Rigid` qui effectue un appel de la forme `Distribute(`S`,DDR(`S'`),`S'`)` pour chaque $S' \subset S$ jusqu'à ce qu'un sous-GCSP sur-es_rigide soit retourné ou que tous les S' aient été traités. En effet, un sous-GCSP S'' retourné par un appel de ce type vérifie nécessairement DDL(S'') $<$ DDR(S'), une condition suffisante pour que S'' soit sur-rigide (cf. définition 4 et lemme 1).

Bien sûr, un tel algorithme serait exponentiel en pire cas puisque le nombre de sous-GCSP dans un GCSP est égal au nombre de sous-ensembles d'objets dans l'ensemble d'objets de ce GCSP. Fort heureusement, nous montrerons (cf. section Propriétés de `Over-Rigid`) qu'il est suffisant d'appliquer la fonction `Distribute` aux sous-GCSP *DDR-minimaux* uniquement (cf. définition 6 ci-après), ce qui produit l'algorithme suivant :

Assure: $S'' \subset S$ est bien- ou sur-es_rigide ou vide
  $S'' \leftarrow emptyGCSP$
  $M \leftarrow$ `DDR-Minimaux(`S`)` $\{$*construit l'ensemble des DDR-minimaux de S*$\}$
  Tant que $S'' = emptyGCSP$ et $M \neq \emptyset$ **Faire**
    $S' \leftarrow$ `Pop(`M`)`
    $S'' \leftarrow$ `Distribute(`S`,DDR(`S'`),`S'`)`
  Fin Tant que
  Retourne S''

Algorithm 8: Over-Rigid (S : GCSP) retourne S'' : GCSP

Définition 6 GCSP DDR-minimaux
Un GCSP S est **DDR-minimal** *ssi il ne contient aucun sous-GCSP possédant le même DDR, c-a-d. que* $\forall S' \subsetneq S$, *DDR(*S'*)<DDR(S).*

L'importance de cette modification vient du fait que la taille d'un sous-GCSP DDR-minimal est bornée en dimension donnée, et le nombre de tels sous-GCSP n'est donc pas exponentiel (cf. paragraphe ci-dessous sur la complexité de l'algorithme).

Exemple d'application de `Over-Rigid`

Considérons le GCSP S de la figure 8.1-b. Soit $M = \{BC, CEF, ...\}$ l'ensemble de ses sous-GCSP DDR-minimaux produits par la fonction `DDR-Minimaux(`S`)`. L'algorithme `Over-Rigid` procède alors comme suit :

1. A la première itération, $S' = BC$ et K =DDR(BC)=5. La figure 8.2-c représente l'appel `Distribute(`$S, 5, BC$`)`. Les arcs issus de la source s sont saturés par le calcul du flot maximum et aucun sous-GCSP sur-rigide n'est identifié.

[5]ou bien- ou sur-s_rigide si on utilise une surcharge augmentée de 1.

2. A la seconde itération, $S' = CEF$ et K =DDR(CEF)=6. La figure 8.2-d présente l'appel `Distribute(`$S, 6, CEF$`)` correspondant à cette itération. Cette fois-ci, on peut constater que l'arc $s \rightarrow R$ n'est pas saturé par le flot maximum. L'ensemble des objets atteignables à partir de la source dans le graphe résiduel étant $\{A, C, D, E, F\}$, l'algorithme termine à cette itération en retournant le sous-GCSP $S'' = ACDEF$ qui est effectivement sur-es_rigide.

Sur ce même GCSP, l'algorithme `Dense`, qui a le même objectif que notre algorithme `Over-Rigid` retournerait soit un segment (par exemple CD), soit un sous-GCSP composé de la droite et d'un point incident (par exemple AB), comme étant un sous-GCSP sur-rigide, ce qui est faux[6].

Propriétés de `Over-Rigid`

Afin de démontrer la correction et la complétude de l'algorithme, nous utiliserons les lemmes suivants qui établissent des propriétés du concept de DDR et de la distribution de flots dans des réseaux objets-contraintes :

Lemme 1 *Soit S un GCSP et $S' \subset S'' \subset S$ deux sous-GCSP. Alors DDR(S') $\leq$ DDR(S'').*

Démonstration : Chaque unité du DDR dans un GCSP représente un déplacement indépendant (translation or rotation) admis par ce GCSP. L'ajout d'un nouvel objet dans ce sous-GCSP ne peut en aucun cas retirer un déplacement de ce GCSP puisque les contraintes sont indépendantes du repère global. Ainsi, DDR(S') $\leq$ DDR($S' \cup \{o\}$).□

Lemme 2 *Soit $S' \subset S'' \subset S$ deux sous-GCSP du GCSP S. Si l'appel* `Distribute(`S, K, S''`)` *retourne un sous-GCSP S_0* non vide, *alors l'appel* `Distribute(`S, K, S'`)` *retourne nécessairement un sous-GCSP S_1* non vide.

Démonstration : Soit $G_{S'}$ le réseau surchargé utilisé par `Distribute(`S, K, S'`)` et $G_{S''}$ le réseau objets-contraintes surchargé utilisé par `Distribute(`S, K, S''`)`. La seule différence entre ces deux réseaux est la présence d'arcs supplémentaires du type $R \rightarrow o$ dans $G_{S''}$ puisque $S' \subset S''$. Ainsi, la distribution de flot est nécessairement plus *difficile* dans $G_{S'}$ que dans $G_{S''}$, les capacités totales issues de la source et entrant dans le puits étant identiques dans ces deux réseaux. Ainsi, si un calcul de flot maximum ne peut saturer les arcs issus de la source dans $G_{S''}$, il est impossible, a fortiori, qu'un flot maximum les sature dans $G_{S'}$.□

Correction de `Over-Rigid` :

Soit S'' un sous-GCSP *non vide* résultant de l'appel à `Over-Rigid`(S). Supposons que S'' résulte de l'appel `Distribute`($S, DDR(S'), S'$) pour un sous-GCSP S donné ; S'' vérifie alors nécessairement DDL(S'') < DDR(S') puisqu'il est *non vide*. De plus, par définition de la fonction `Distribute`, $S' \subset S''$. Le lemme 1 assure alors que DDR(S')$\leq$DDR(S''), et on peut donc affirmer que DDL(S'')<DDR(S''), c-a-d. que S'' est sur-es_rigide.□

[6]En pratique, l'algorithme `Dense` est assisté par des heuristiques qui lui permettent d'éviter des erreurs aussi triviales, mais il demeure géométriquement incorrect et peut toujours être trompé par des configurations non répertoriées [Jermann, 2002].

Complétude de `Over-Rigid` :

L'algorithme `Over-Rigid` applique la surcharge sur chaque sous-GCSP dans l'ensemble M des sous-GCSP *DDR-minimaux* (généré par la fonction `DDR-Minimaux(`S`)`). Remarquons que tout sous-GCSP non DDR-minimal contient par définition un sous-GCSP DDR-minimal ayant même DDR. Le lemme 2 assure qu'aucun sous-GCSP sur-es_rigide ne peut être manqué par l'algorithme.□

Complexité de `Over-Rigid` :

La complexité en temps de l'algorithme `Over-Rigid` dépend directement du nombre de sous-GCSP DDR-minimaux. Nous avons prouvé par énumération que la taille (en nombre d'objets) d'un sous-GCSP DDR-minimal est 2 et 2D et 3 en 3D pour des GCSP constitués de points, droites et plans sous des contraintes de distances, incidences, angles et parallélismes[7]. Ainsi, le nombre de sous-GCSP DDR-minimaux dans un GCSP de ce type constitué de n objets en dimension d est en $O(n^d)$.

Appelons C_1 la complexité de la fonction `DDR-Minimaux`, et C_2 celle de la fonction `Distribute` que nous avons étudiée à la section précédente. Alors, la complexité en pire cas de l'algorithme `Over-Rigid` est $O(C_1 + n^d * C_2)$.

C_1 est généralement la complexité de la démonstration automatique en géométrie puisque le calcul du DDR nécessite la détermination des propriétés géométriques vérifiées par les objets du GCSP. C_1 est alors exponentielle en pire cas. Cependant, cette complexité peut être polynomiale voire constante pour des classes de GCSP particulières, comme les systèmes à barres ou les mécanismes génériques. Qui plus est, on peut employer plusieurs heuristiques dans le calcul du DDR qui permettent de rendre la complexité moyenne plus abordable en pratique. Dans ces cas là, C_1 est généralement négligeable en comparaison de C_2, et on obtient une complexité globale de l'algorithme `Over-Rigid` en $O(n^d * n * (n + m)^2)$. En comparaison, la complexité de l'algorithme `Dense` est en $O(m * n * (n + m)^2)$. Notre algorithme induit donc un surcoût approximativement linéaire en 2D et quadratique en 3D, puisque le nombre de contraintes m dans un GCSP est généralement $O(n)$ (moins d'équations que de variables dans un GCSP rigide).

8.4.2.2 Détection de sous-GCSP bien- ou sur-es_rigides

Afin d'identifier des sous-GCSP bien- ou sur-es_rigides, il suffit d'apporter une très légère modification à l'algorithme `Over-Rigid` présenté ci-dessus. Cette modification consiste simplement à ajouter 1 à la valeur de la surcharge appliquée à chaque appel de la nouvelle fonction `Distribute`. Celle-ci devient $K = DDR(S') + 1$ au lieu de $K = DDR(S')$. Il en résulte que les GCSP S'' non vides retournés par l'algorithme `Well-Or-Over-Rigid` vérifieront nécessairement $DDL(S'') \leq DDR(S')$, ce qui constitue une condition suffisante pour que S'' soit bien- ou sur-es_rigide (cf. définition 4 et lemme 1).

Toutes les propriétés de l'algorithme `Over-Rigid` sont préservées : l'algorithme `Well-Or-Over-Rigid` est correct, complet et de complexité $O(n^d * n * (n + m)^2))$.

Remarque : Même si le sous-GCSP S'' vérifie $DDL(S'') = DDR(S'')$, cela ne signifie pas qu'il est bien-es_rigide. En effet, il faut encore vérifier la seconde condition de la

[7] La plupart des GCSP peuvent se ramener à des systèmes utilisant uniquement ces objets et contraintes primitives.

es_rigidité, c-a-d. que S'' ne contient aucun sous-GCSP sur_es-rigide.

8.4.2.3 Décision de rigidité d'un GCSP

Pour décider si un GCSP S donné est bien-es_rigide ou pas, il faut vérifier les deux conditions imposées par la définition de la es_rigidité (cf. définition 4) :
- $DDL(S) = DDR(S)$
- S n'est pas sur-es_rigide

Pour vérifier la première condition, il suffit de calculer le nombre de DDL de S (par un simple compte, cf. définition 2) et son degré de rigidité DDR (voir [Jermann *et al.*, 2004a]).

Afin de vérifier la seconde condition, un simple appel à l'algorithme `Over-Rigid` suffit : si cet algorithme retourne un sous-GCSP vide, alors S n'est pas sur-es_rigide.

L'algorithme est complet, correct et de complexité égale soit à celle du calcul du DDR (si celle-ci est exponentielle ou chère), soit égale à celle de l'algorithme `Over-Rigid` si le calcul du DDR est négligeable.

8.4.2.4 Minimisation d'un sous-GCSP bien- ou sur-es_rigide

Cet algorithme reprend le principe de minimisation proposé par Hoffmann *et al.* dans l'algorithme `Minimal-Dense`, qui visait à minimiser un sous-GCSP bien- ou sur-s_rigide. Il suffit de remplacer l'appel à `Dense` par un appel à `Well-Or-Over-Rigid`[8] dans l'algorithme `Minimal-Dense` présenté précédemment afin d'obtenir l'algorithme `Minimal-Rigid` qui retourne effectivement un sous-GCSP bien- ou sur-es_rigide minimal pour l'inclusion.

La correction et la complétude de cet algorithme sont encore une fois assurées par les propriétés des algorithmes `Minimal-Dense` et `Well-Or-Over-Rigid`. La complexité, quant à elle, est en pire cas $O(n*n^d*n*(n+m)^2))$ si le sous-GCSP initialement identifié est le GCSP entier et que tout objet doit effectivement être testé.

8.5 Conclusion

Dans cet article, nous avons proposé une nouvelle famille d'algorithmes permettant de répondre aux principales questions associées au concept géométrique de rigidité en nous appuyant sur une nouvelle caractérisation : la rigidité structurelle étendue. Ces algorithmes découlent des algorithmes à base de flots proposés par Hoffmann *et al.* pour la caractérisation par rigidité structurelle.

D'une part, nous avons montré dans [Jermann *et al.*, 2004a] que la caractérisation par rigidité structurelle étendue correspond strictement mieux à la rigidité que celle par rigidité structurelle, et d'autre part, nous venons de montrer que les algorithmes existants pour l'ancienne caractérisation pouvaient tous être reproduits pour la nouvelle caractérisation, moyennant un surcoût approximativement linéaire en 2D et quadratique en 3D.

Nous pensons donc que toutes les méthodes d'analyse qualitative ou de résolution de GCSP actuellement basées sur la caractérisation par rigidité structurelle peuvent à présent considérer l'utilisation de notre nouvelle caractérisation. Cette alternative apportera vraisemblablement un gain significatif en fiabilité, mais également en généricité

[8] ou `Over-Rigid` si on souhaite seulement minimiser des sous-GCSP sur-es_rigides.

puisque les méthodes actuelles ont pour la plupart recours à des règles *ad-hoc* pour gérer les nombreuses exceptions connues à la rigidité structurelle classique.

Le concept de degré de rigidité (DDR), qui s'avère central dans la définition de la nouvelle caractérisation, pose cependant encore quelques questions : en pire cas, calculer le DDR d'un GCSP s'avère de l'ordre de la preuve formelle de théorème géométrique. Nous avons déjà identifié des classes de GCSP pour lesquelles ce problème est polynomial, mais un effort reste à faire sur ce sujet afin d'identifier d'autres classes d'une part, et de proposer des heuristiques de calcul permettant d'éviter le plus possible le recours à des algorithmes exponentiels d'autre part.

Par ailleurs, une comparaison expérimentale entre les deux types de caractérisation, tant du point de vue qualitatif que du point de vue des performances des algorithmes correspondants, reste à mener. Elle permettra de valider la démonstration formelle de la supériorité de la caractérisation par rigidité structurelle étendue, et d'assurer la faisabilité pratique des algorithmes correspondants.

Chapitre 9

Retour-arrière inter-blocs et résolution par intervalles

Article paru dans les actes de JNPC'04
Auteurs : Bertrand Neveu, Gilles Trombettoni, Christophe Jermann

Résumé

Cet article présente une technique, appelée Retour-arrière inter-blocs (en anglais InterBlock Bactracking `IBB`), qui améliore la résolution par intervalles de systèmes d'équations non linéaires sur les réels quand ces systèmes sont décomposables.

Cette technique, introduite en 1998 par Bliek et al, traite un système d'équations décomposé au préalable en un ensemble de (petits) sous-systèmes $k \times k$, appelés blocs. Une solution est obtenue en combinant les solutions partielles calculées dans les différents blocs. Cette approche semble particulièrement intéressante pour accélérer les techniques de résolution par intervalles.

Dans cet article, nous analysons en détail différentes variantes de `IBB`, qui diffèrent dans leurs stratégies de retour-arrière et de filtrage. Nous introduisons aussi `IBB-GBJ`, une nouvelle variante basée sur le *graph-based backjumping* de Dechter.

Une comparaison complète sur huit problèmes nous a permis de mieux comprendre le comportement de `IBB`. Il montre que les variantes `IBB-BT+` et `IBB-GBJ` sont de bons compromis entre simplicité et performance. De plus, cela montre que limiter le filtrage à l'intérieur des blocs est intéressant. Pour toutes les instances testées, `IBB` gagne plusieurs ordres de grandeur par rapport à une résolution globale sans décomposition.
Mots-clés : intervalles, décomposition, retour-arrière, systèmes peu denses

9.1 Introduction

Il existe seulement quelques techniques pour calculer toutes les solutions d'un système d'équations non linéaires sur les réels. Les techniques de calcul formel, comme les bases de Gröbner [Buchberger, 1985] et les méthodes de Ritt-Wu [Wu, 1986] sont souvent très coûteuses en temps et en mémoire et sont limitées à des systèmes d'équations algébriques de petite taille. La méthode par continuation, appelée aussi homotopie [Lahaye, 1934;

Durand, 1998], peut donner de très bons résultats. Cependant, éliminer les solutions complexes et prendre en compte les inégalités n'est pas simple. De plus, la méthode doit partir d'un système initial "proche" du système à résoudre. Cela rend difficile une automatisation, particulièrement pour des systèmes non algébriques.

Les techniques par intervalles sont des alternatives prometteuses, quand les domaines des variables sont bornés a priori, ce qui est souvent le cas. Elles prennent aussi facilement en compte les inégalités. Elles ont obtenu de bons résultats dans plusieurs domaines, comme la commande robuste [Jaulin *et al.*, 2001] et la robotique [Merlet, 2002]. Cependant, ces méthodes sont assez lentes et leur efficacité décroît avec la taille du système à résoudre : il est reconnu que des systèmes avec des centaines (parfois des dizaines) de contraintes non linéaires ne peuvent être résolus en pratique.

Dans certaines applications faites de contraintes non linéaires, les systèmes sont peu denses et peuvent être décomposés en sous-systèmes par des techniques équationnelles ou géométriques. La CAO, la reconstruction de scène 3D avec des contraintes géométriques [Wilczkowiak *et al.*, 2003], la chimie moléculaire [Hendrickson, 1992] et la robotique représentent de tels champs d'applications prometteurs. Différentes techniques peuvent être utilisées pour décomposer de tels systèmes en *blocs* $k \times k$. Les décompositions équationnelles travaillent sur le graphe biparti formé par les variables et les équations [Ait-Aoudia *et al.*, 1993; Bliek *et al.*, 1998]. Quand les équations représentent des contraintes géométriques, les décompositions géométriques produisent généralement des blocs plus petits [Hoffmann *et al.*, 1997; Jermann *et al.*, 2000].

L'approche originale, introduite en 1998 [Bliek *et al.*, 1998], appelée dans cet article *Retour-arrière inter-blocs* (en anglais InterBlock Backtracking `IBB`), peut être utilisée après cette phase de décomposition. En suivant l'ordre partiel entre les blocs donné par la décomposition, on peut appliquer un processus de résolution classique pour chaque bloc, traitant ainsi des systèmes de taille réduite. `IBB` combine les solutions partielles obtenues pour construire les solutions globales du problème.

Bien que `IBB` puisse être utilisé avec d'autres types de résolveurs, nous avons intégré des techniques d'intervalles qui sont très générales et efficaces sur des systèmes réduits. Le premier article [Bliek *et al.*, 1998] a présenté les premières versions de `IBB` en incluant plusieurs schémas de retour-arrière et une technique de décomposition équationnelle. Depuis, plusieurs variantes de `IBB` ont été développées sans être présentées en détails et dans [Jermann *et al.*, 2000] nous nous sommes par ailleurs intéressés à des techniques de décomposition géométrique utilisant des algorithmes de flots pour détecter des parties rigides.

Contributions.

Cet article détaille les phases de résolution réalisées par `IBB` avec des techniques par intervalles. Il apporte plusieurs contributions :

- De nombreuses expérimentations ont été réalisées sur un banc d'essais de plus grande taille (entre 30 et 178 équations). Elles ont conduit à une comparaison plus équitable entre les variantes. Ainsi, cela a confirmé que `IBB` pouvait gagner plusieurs ordres de grandeur en temps de calcul par rapport aux techniques par intervalles appliquées sur le système entier non décomposé. Finalement, cela nous a permis de mieux comprendre les subtilités de l'intégration des techniques d'intervalles dans `IBB`.

- Une nouvelle version de IBB est présentée, basée sur l'algorithme bien connu en domaines finis GBJ [Dechter, 1990]. Les expérimentations ont montré que IBB-GBJ est un bon compromis entre les versions antérieures.
- Un filtrage inter-blocs peut être ajouté à IBB. Son impact sur les performances est analysé dans les expérimentations.

Contenu

La partie 9.2 pose quelques hypothèses sur les problèmes qui peuvent être traités. La partie 9.3 rappelle les principes de IBB et de la résolution par intervalles. La partie 9.4 détaille l'algorithme IBB-GBJ et la stratégie de propagation inter-blocs. La partie 9.5 présente les expérimentations réalisées sur un échantillon de huit problèmes. Les résultats sont analysés dans la partie 9.6.

9.2 Hypothèses

IBB résout un système décomposé d'équations sur les réels. Tout type d'équation peut être traité, algébrique ou non. Notre banc d'essais comprend des équations linéaires et quadratiques. IBB est utilisé pour trouver **toutes** les solutions d'un système de contraintes. Il pourrait être modifié pour l'optimisation globale (choisir la solution minimisant un critère donné) en remplaçant le retour-arrière inter-blocs par un algorithme classique de séparation-évaluation (*Branch and Bound*). Rien n'a été fait dans cette direction pour le moment.

Nous faisons l'hypothèse que les systèmes à résoudre ont un nombre fini de solutions ponctuelles. Ainsi, cela permet à IBB de combiner un ensemble fini de solutions partielles. Cette condition vaut donc ainsi pour chaque sous-système (bloc), qui doit donc être un sous-système carré, c.-à-d qui contient autant d'équations que de variables.

Aucune autre hypothèse ne doit être posée sur la technique de décomposition. Cependant, comme nous utilisons une décomposition structurelle, il ne doit pas y avoir d'équations redondantes. Les inégalités ou les équations supplémentaires utilisées pour réduire le nombre de solutions peuvent ensuite être ajoutées facilement pendant la phase de résolution dans le bloc correspondant à leurs variables (comme par exemple, en tant que contraintes "soft" dans Numerica [Van Hentenryck *et al.*, 1997]), mais cette intégration est en dehors du champ de cet article.

Pour traiter certaines redondances dans la phase de décomposition, des décompositions "symboliques" pourraient aussi être envisagées [Bondyfalat *et al.*, 1999]. Ces algorithmes basés sur des techniques de calcul formel peuvent tenir compte des *coefficients* présents dans les contraintes, et pas seulement des dépendances structurelles.

Remarque

En pratique, les problèmes qui peuvent être décomposés sont souvent sous-contraints et ont plus de variables que d'équations. Cependant, dans les applications existantes, le problème peut être rendu carré (autant de variables que d'équations) en donnant une valeur à un sous-ensemble de variables appelées *paramètres d'entrée*. Les valeurs de ces paramètres peuvent être fournies par l'utilisateur, lues sur une esquisse ou données par un processus préliminaire (par exemple, en reconstruction de scène 3D [Wilczkowiak *et al.*, 2003]).

9.3 Fondements

Cette partie présente d'abord brièvement la résolution par intervalles. La version la plus simple de IBB est ensuite introduite sur un exemple.

9.3.1 Techniques de résolution par intervalles

CSP numériques

Un problème de satisfaction de contraintes numériques (NCSP) $P = (V, C, I)$ contient un ensemble de contraintes C et un ensemble de n variables V. Chaque variable $v_i \in V$ peut prendre une valeur réelle dans l'intervalle $d_i \in I$; les bornes de d_i sont des nombres flottants. Résoudre P consiste à affecter des valeurs aux variables de V de telle sorte que les contraintes de C soient satisfaites.

Un produit cartésien d'intervalles en dimension n peut être représenté par un parallélépipède rectangle n-dimensionnel appelé **boîte**. Les algorithmes de résolution vont donc manipuler des boîtes, en appliquant des bissections et des propagations de contraintes qui servent à éliminer des boîtes sans solution. Ils peuvent ainsi isoler des boîtes pouvant contenir des solutions.

Les nombres réels ne peuvent pas être représentés de manière exacte dans les ordinateurs, le processus de résolution s'arrête quand une très petite boîte a été obtenue. Une telle boîte est appelée boîte atomique dans cet article. En théorie, une boîte atomique peut avoir la largeur existant entre deux flottants consécutifs, En pratique, le processus est interrompu quand tous les intervalles contiennent w_1 flottants[1]. Il est important de remarquer qu'une boîte atomique ne contient pas nécessairement une solution. En effet, le processus est semi-déterministe : évaluer une égalité avec l'arithmétique des intervalles peut prouver que la relation n'a pas de solution (quand les boîtes de gauche et de droite ont une intersection vide), mais ne permet pas d'affirmer qu'il existe une solution dans l'intersection.

Résolveur utilisé dans `IBB`

Nous utilisons `IlogSolver` et sa bibliothèque `IlcNum`. `IlcNum` implante la plupart des caractéristiques du système `Numerica` [Van Hentenryck *et al.*, 1997]. Cette bibliothèque utilise plusieurs principes développés dans l'analyse par intervalles et en programmation par contraintes. Le processus de résolution suivi par `IBB` peut être résumé de la manière suivante :

1. *Bissection :* on choisit une variable et on coupe son domaine en deux intervalles (la boîte est coupée selon l'une de ses dimensions). Cela donne deux sous-CSP qui sont traités en séquence et rend le processus de résolution combinatoire.
2. *Filtrage/propagation :* De l'information locale (sur une contrainte) ou plus globale (3B) est utilisée pour réduire la boîte courante ou arrêter une branche détectée sans solution.

[1] w_1 est un paramètre défini par l'utilisateur. Dans la plupart des implantations, w_1 est une largeur et non un nombre de flottants.

3. *Test d'unicité :* un test utilisant des méthodes d'analyse numérique est réalisé sur le système d'équations en prenant en compte les dérivées premières et secondes des équations. Quand ce test réussit sur la boîte courante, il assure qu'il existe une solution unique et qu'un algorithme numérique classique (la méthode de Newton) peut converger vers cette solution.

Ces trois étapes sont réalisées en boucle. Le processus s'arrête dans une branche quand une boîte atomique d'une taille inférieure à w_1 a été obtenue, ou quand le test d'unicité est vérifié sur la boîte courante.

La propagation est réalisée par un algorithme de point fixe de type AC3. Quatre types de filtrage affinent les bornes des intervalles (sans créer de trou : on ne manipule pas d'unions d'intervalles). La `box-consistance` [Van Hentenryck *et al.*, 1997] provient de `IlcNum`, la `2B-consistance` provient de l'implantation initiale des CSP numériques dans `IlogSolver`. Bien qu'algorithmiquement différentes, ce sont des consistances locales qui ne considèrent qu'une contrainte à la fois en réduisant les bornes des variables impliquées. La `3B-consistance` [Lhomme, 1993] utilise la `2B-consistance` comme sous-procédure et un principe de réfutation (rognage) pour réduire les bornes de chaque variable. La `bound-consistance` suit le même principe, mais utilise la `box-consistance` comme sous-procédure. Un paramètre w_2 doit être spécifié pour la `bound` ou la `3B` : la borne d'une variable n'est pas mise à jour si la réduction de l'intervalle est inférieure à w_2. Le paramètre w_1 est aussi utilisé pour éviter un grand nombre de propagations dans le cas de convergence lente de la `2B` ou `Box` : une réduction n'est effectuée que si la portion d'intervalle à enlever est supérieure à w_1.

Le test d'unicité a été implanté dans `IlcNum`. Malheureusement, il ne peut être découplé de la `Box` ou de la `Bound` et il ne peut pas être appelé quand le filtrage est celui de `2B` ou de la `3B` seul. Cela nous empêche d'analyser finement son comportement.

9.3.2 `IBB-BT`

`IBB` traite un **Graphe sans circuit** de blocs (**DAG**) produit par une technique de décomposition. Un **bloc** i est un sous-système comprenant équations et variables. Certaines variables de i, appelées **variables d'entrée**, seront remplacées par leurs valeurs lors de la résolution du bloc. Les autres variables sont appelées **variables de sortie**. Un bloc **carré** a autant d'équations que de variables de sortie. Il existe un arc d'un bloc i vers un bloc j si et seulement si une équation de j contient au moins une variable de sortie de i. Le bloc i est appelé parent de j. Le `DAG` induit un ordre partiel pour la résolution effectuée par `IBB`.

Exemple :

Nous allons illustrer le principe de `IBB` en prenant l'exemple de CAO mécanique 2D introduit dans [Bliek *et al.*, 1998] (cf Fig. 9.1).

Différents points (cercles blancs) sont reliés par des barres rigides (segments). Les barres imposent une contrainte de distance entre deux points. Le point h (cercle noir) diffère des autres par le fait qu'il est sur la barre $\langle g, i\rangle$. Enfin, le point d est contraint à coulisser sur la droite spécifiée. Le problème est de trouver une configuration des points qui satisfait toutes les contraintes. Une technique de décomposition équationnelle produit le `DAG` montré sur la figure 9.1-droite.

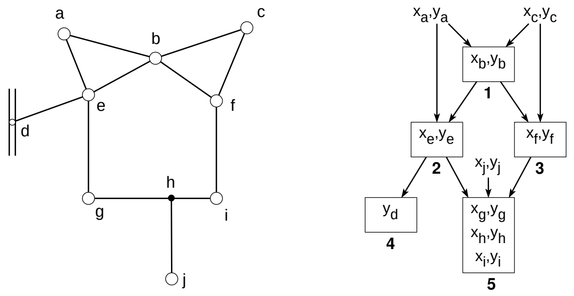

FIG. 9.1 – Graphe de blocs : exemple.

Illustration de `IBB` :

Devant respecter l'ordre partiel du DAG, `IBB` suit donc un des ordres totaux induits, par exemple le bloc 1, puis 2, 3, 4 et 5. Il appelle d'abord le résolveur par intervalles sur le bloc 1 et obtient une première solution pour x_b et y_b(le bloc a deux solutions). Une fois cette solution obtenue, on peut substituer x_b et y_b par leurs valeurs dans les équations des blocs suivants : 2 et 3. Puis, on traite les blocs 2, 3, 4 et 5 de façon similaire.

Quand un bloc n'a pas de solution, il faut revenir en arrière. Un retour-arrière chronologique revient sur le bloc précédent. `IBB` calcule la solution suivante pour ce bloc et réessaye de résoudre les blocs en aval. Cependant, on peut remarquer que ce retour-arrière chronologique de la version `IBB-BT` ne prend pas en compte l'ordre partiel du `DAG`. En effet, supposons que, dans l'exemple ci-dessus, le bloc 5 n'ait pas de solution. Le retour arrière chronologique reviendra sur le bloc 4, en trouvera une solution différente et tentera à nouveau de résoudre le bloc 5. Il est clair que le même échec se reproduira, les équations du bloc 5 ne comprenant pas de variable du bloc 4.

Nous avons expliqué dans [Bliek *et al.*, 1998] que les schémas `CBJ` et `Dynamic Backtracking` provenant des CSP en domaines finis ne peuvent pas être directement utilisés. En effet, un échec dans un bloc ne permet pas de discriminer finement le bloc parent qui est la cause de l'échec. Les contraintes sont généralement n-aires et font intervenir plusieurs parents et un échec est donc dû aux valeurs de tous les parents. Un retour arrière "intelligent" `IBB-GPB`, basé sur le "partial order backtracking" [McAllester, 1993; Bliek, 1998] avait été présenté. La principale difficulté dans l'implantation de `IBB-GPB` est de maintenir un ensemble de conflits appelés "nogoods". De plus, toute modification de `IBB-GPB`, pour ajouter une caractéristique ou une heuristique, comme le filtrage inter-blocs, demande une grande attention.

Nous présentons dans cet article une variante plus simple basée sur le "graph-based backjumping" (`GBJ`) de Dechter [Dechter, 1990], et nous le comparons à `IBB-GPB` et à `IBB-BT`.

Remarque

De part l'utilisation de la résolution par intervalles, une solution d'un bloc ne correspond pas à un ensemble de valeurs réelles, mais à une boîte atomique. Ainsi, le remplacement des variables de sortie des blocs parents reviendrait à introduire de petits intervalles constants de largeur w_1 dans le bloc courant. Mais, le résolveur utilisé ne traite pas les intervalles constants et nous avons dû recourir à une heuristique appelée **heuristique du point milieu** qui remplace un intervalle constant par le nombre flottant situé au milieu de l'intervalle. Cette heuristique a plusieurs conséquences qui sont analysées dans la partie 9.6.1.

9.4 Utilisation de la structure du DAG et filtrage inter-blocs

La structure du DAG peut être prise en compte de deux manières :

- *en descendant :* une *condition de recalcul* peut éviter de recalculer inutilement les solutions d'un bloc ;
- *en remontant :* quand un bloc n'a pas de solution, on peut revenir sur un bloc parent, et non pas nécessairement sur le bloc précédent.

Les deux paragraphes suivants présentent ces améliorations. Le troisième détaille le filtrage inter-blocs qui peut être ajouté à tous les schémas de retour-arrière. Ceci conduit à plusieurs variantes de IBB qui ont été testées sur notre banc d'essais.

9.4.1 La condition de recalcul

Cette condition peut être testée dans toutes les variantes de IBB, même IBB-BT. Vérifier cette condition de recalcul n'est pas coûteux et peut apporter des gains importants.

La **condition de recalcul** indique qu'il est inutile de calculer les solutions d'un bloc si les variables des parents n'ont pas changé de valeur. Dans ce cas, IBB peut réutiliser les solutions calculées la dernière fois que le bloc a été traité. Illustrons ce point sur l'exemple didactique résolu par IBB-BT.

Supposons qu'une première solution ait été calculée pour le bloc 3, et que toutes les solutions calculées dans le bloc 4 aient conduit à un échec. IBB-BT revient alors sur le bloc 3 et la seconde position du point f est calculée. Quand IBB redescend sur le bloc 4, ce bloc devrait normalement être recalculé suite aux modifications de la solution courante de f. Mais ni x_f ni y_f ne sont présents dans les équations du bloc 4, ainsi les deux solutions du bloc 4 calculées auparavant peuvent être réutilisées.

9.4.2 IBB-GBJ

Six tableaux sont utilisés dans l'algorithme IBB-GBJ présenté sur la figure ci dessous :

- `solutions[i,j]` contient la $j^{\text{ème}}$ solution du bloc i.
- `blocks_back[i]` contient l'ensemble de blocs ancêtres du bloc i. Le bloc le plus récent parmi eux (i.e., celui avec le plus grand numéro) est choisi en cas de retour-arrière.
- `parents[i]` contient l'ensemble des blocs parents du bloc i.
- `assignment[v]` contient la valeur courante de la variable v.
- `save_parents[i]` contient les valeurs des variables des blocs parents de i la dernière fois que i a été résolu. Ce tableau est seulement utilisé quand la condition de recalcul est testée.
- `#sols[i]` contient le nombre de solutions du bloc i.

```
Algorithme IBB_GBJ (#blocks, solutions, parents, save_parents, assignment)

  for i = 1 to #blocks  do
     blocks_back[i] = parents[i]
     sol_index[i] = 0
     #sols[i] = 0
  end_for
  i = 1
  while (i >= 1) do
      if (Parents_changed? (i, parents, save_parents, assignment)) then
           update_save_parents (i, parents, save_parents, assignment)
           sol_index[i] = 0
           #sols[i] = 0
      end_if

      if (sol_index[i] >= #sols[i])    and
           not (next_solution(i, solutions, #sols)))
      then
          i = backjumping (i, blocks_back, sol_index)
      else  /* les solutions [i, sol_index[i] ] sont affectées au bloc i */
          assign_block (i, solutions, sol_index, assignment)
          sol_index[i] = sol_index[i] + 1
          if (i == #blocks)  then  /* solution globale trouvée*/
             store_total_solution (solutions, sol_index, i)
             blocks_back[#blocks] = {1...#blocks-1}
          else
             i = i + 1
          end_if
       end_if
   end_while
```

IBB-GBJ peut trouver toutes les solutions d'un CSP numérique. A partir du DAG, les blocs sont tout d'abord ordonnés dans un ordre total et numérotés de 1 à #blocks. Après une phase d'initialisation, la boucle while correspond à la recherche de solutions, i étant le bloc courant. Le processus se termine quand $i = 0$, ce qui signifie que toutes les solutions ont été trouvées.

La fonction next_solution appelle le résolveur pour calculer la solution suivante du bloc i. Si une solution est trouvée, la fonction retourne *vrai*, et les tableaux solutions et #sols sont mis à jour. Sinon, la fonction retourne *faux*.

Le code correspondant au premier else contient les actions à réaliser quand une solution d'un bloc est choisie. La procédure assign_block modifie le tableau assignment de telle sorte que les valeurs de la solution trouvée sont affectées aux variables du bloc i. Quand une solution totale a été trouvée, la mise à jour de blocks_back est standard et assure la complétude [Dechter, 1990]. Parents_changed? vérifie la condition de recalcul.

Quand un bloc n'a pas de solution, une fonction standard backjumping retourne un niveau j où il est possible de revenir sans perdre de solution. Il est important d'ajouter dans les causes d'échec du bloc j (i.e., blocks_back[j]) celles du bloc i. En effet, ces blocs sont une cause d'erreur possible pour la valeur courante du bloc j (comme dans GBJ).

```
Fonction backjumping (i, in-out blocks_back, in-out sol_index)

 if blocks-back[i] then
     j = more_recent (blocks_back[i])
     blocks_back[j] = blocks_back[j] U blocks_back[i] \ {j}
 else
     j = -1
 end_if

 for k = j+1 to i do
    blocks_back[k] = parents[k]
    sol_index[k] = 0
 end_for

 return j
```

Favoriser la valeur courante

Le principal inconvénient des algorithmes basés sur le “backjumping” est que le travail réalisé dans les blocs entre i et j est perdu. Quand ces blocs sont traités de nouveau, on choisit d'abord la valeur courante d'une variable, au lieu de retraverser le domaine depuis le début. Puisque les domaines sont dynamiques avec IBB (les solutions d'un bloc changent quand de nouvelles valeurs d'entrée lui sont données), cette amélioration ne peut être réalisée que si la condition de recalcul permet de réutiliser les solutions antérieures.

Cette heuristique a été ajoutée à IBB-GBJ[2]. Cependant, probablement à cause de la remarque ci-dessus, les gains en performance obtenus par cette heuristique sont faibles et ne sont pas détaillés dans les expérimentations (cf partie 9.5).

9.4.3 Filtrage inter-blocs

Contrairement aux caractéristiques du retour-arrière, le filtrage inter-blocs (*ibf*) est spécifique aux techniques de résolution par intervalles. *ibf* peut être incorporé à toute variante de IBB.

Pour les CSP en domaines finis, on a généralement observé que, pendant la résolution, un filtrage puissant sur le problème restant est intéressant. C'est pourquoi nous avons décidé d'installer un filtrage inter-blocs dans IBB : au lieu de limiter le processus de filtrage (basé sur les consistances 2B, 3B, Box ou Bound dans notre outil) au bloc courant, nous avons étendu le champ du filtrage à toutes les variables.

Plus précisément, avant de résoudre un bloc i, on forme un sous-système de variables et équations extrait des blocs suivants :

[2] L' algorithme doit gérer un autre indice en plus de sol_index.

1. soit $B = \{i ... \#blocks - 1\}$ l'ensemble des blocs non encore résolus,
2. on réduit B aux blocs connexes à i dans le graphe réduit aux blocs de B [3].

Ensuite, la bissection est appliquée seulement sur le bloc i alors que le processus de filtrage est appliqué sur toutes les variables des blocs dans B.

Pour illustrer *ibf*, considérons le `DAG` de l'exemple didactique. Quand le bloc 1 est résolu, tous les blocs sont considérés par *ibf* puisqu'ils sont tous connexes au bloc 1. Alors, toute réduction d'intervalle dans le bloc 1 peut impliquer une réduction pour toute variable du système. Quand le bloc 2 est résolu, une réduction peut avoir une influence sur les blocs 3, 4 ou 5 pour les mêmes raisons. (On notera que le bloc 3 n'est pas en aval du bloc 2.) Quand le bloc 3 est résolu, une réduction ne peut avoir une influence que sur le bloc 5. En effet, après avoir enlevé du graphe les blocs 1 et 2, les blocs 3 et 4 n'appartiennent plus à la même composante connexe. En fait, aucune propagation ne peut atteindre le bloc 4 puisque les variables parents du bloc 4 qui sont dans le bloc 2 ont un intervalle déjà réduit à une largeur d'au plus w_1 et ne peuvent plus être réduits.

Remarque

Il faut faire attention à la façon dont *ibf* est incorporé dans `IBB-GBJ`. En effet, les réductions induites par les blocs précédents doivent être regardées comme des causes d'échec possibles. Cette modification de l'algorithme n'est pas détaillée et nous illustrons juste ce point sur l'exemple didactique. Si aucune solution n'est trouvée dans le bloc 3, `IBB` avec *ibf* doit revenir sur le bloc 2 et non sur le bloc 1. En effet, quand le bloc 2 a été résolu, une réduction pourrait avoir été propagée sur le bloc 3 (à travers 5).

9.5 Expérimentations

Des expérimentations exhaustives ont été menées sur 8 problèmes composés de contraintes géométriques. Nous avons comparé différentes variantes de `IBB` avec une résolution par intervalles appliquée sur le système entier (appelée *résolution globale* ci-après).

9.5.1 Banc d'essais

Certains d'entre eux sont des problèmes artificiels, à base surtout de contraintes quadratiques de distance. `Mechanism` et `Tangent` sont issus de [Latham et Middleditch, 1996] et [Bouma *et al.*, 1995]. `Chair` est un assemblage réaliste d'une chaise fait de 178 équations représentant une grande variété de contraintes géométriques : distance, angle, incidence, parallélisme, orthogonalité, etc.

[3] L'orientation du `DAG` est oubliée à cette étape, les arcs du `DAG` sont transformées en arêtes, et le filtrage peut s'appliquer sur des blocs frères.

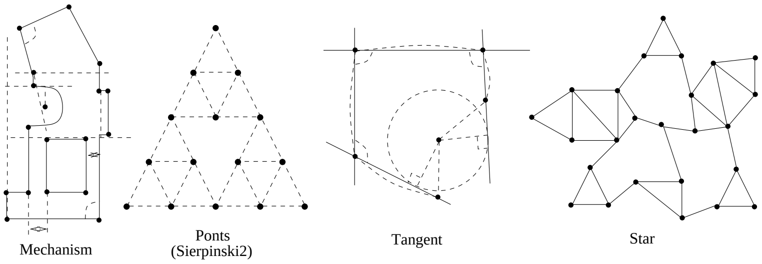

FIG. 9.2 – Banc d'essais 2D

Dim	GCSP	Dec.	Taille	Taille Dec.	t.p	pet.	moy.	gr.
2D	Mechanism	equ.	98	98 = 1x10, 2x4, 27x2, 26x1	1	8	48	448
	Ponts	equ.	30	30 = 1x14, 6x2, 4x1	1	15	96	128
	Sierpinski3	geo.	84	124 = 44x2, 36x1	1	8	96	138
	Tangent	geo.	28	42 = 2x4, 11x2, 12x1	4	16	32	64
	Star	equ.	46	46 = 3x6, 3x4, 8x2	1	4	8	8
3D	Chair	equ.	178	178 = 1x15,1x13,1x9,5x8,3x6, 2x4,14x3,1x2,31x1	6	6	18	36
	Hourglass	geo.	39	39 = 2x4, 3x3, 2x2, 18x1	1	1	2	8
	Tetra	equ.	30	30 = 1x9, 4x3, 1x2, 7x1	1	16	68	256

TAB. 9.1 – Détails du banc d'essais : méthode de décomposition (Dec.) ; nombre d'équations (Taille) ; Taille des blocs (Taille Dec.)- $N \times K$ signifie N blocs de taille K ; nombre de solutions avec les quatre types de domaines sélectionnés : très petits (largeur = 0.1), petits (1), moyens (10), grands (100).

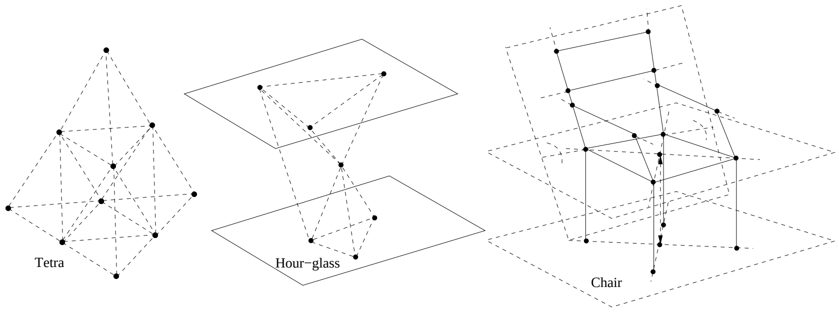

FIG. 9.3 – Banc d'essais 3D

Les domaines ont été choisis autour d'une solution donnée et conduisent à des espaces de recherche radicalement différents. On notera qu'un problème défini avec de grands domaines revient généralement à mettre $]-\infty,+\infty[$ comme domaine de chaque variable.

`Sierpinski3` est la fractale de `Sierpinski` au niveau 3, c-à-d. 3 `Sierpinski2` mis ensemble. Le système d'équations correspondant aurait environ 2^{40} solutions, on a donc limité les domaines initiaux à des largeurs de 0.1 (très petit), 0.8 (petit), 0.9 (moyen), 1 (grand).

9.5.2 Choix du filtrage

Nous avons sélectionné les meilleurs algorithmes de filtrage en réalisant des tests sur deux problèmes de taille moyenne. Plusieurs précisions ont été essayées pour les paramètres w_1 et w_2.

Il est apparu clairement (cf TAB. 9.2) que `2B+Box` et `3B` surpassent les autres filtrages. Tous les tests suivants ont été réalisés avec ces deux techniques.

	w_2	w_1	2B/3B	Box/Bound	2B+Box/3B+Bound
Ponts	0	1e-6	sing	264	**29**
		1e-8	sing	292	**32**
		1e-10	sing	278	**32**
	1e-2	1e-6	116	2078	309
		1e-8	2712	2642	1303
		1e-10	13565	2652	5570
	1e-4	1e-6	84	>54000	523
		1e-8	4413	>54000	5274
Tangent	0	1e-6	sing	547	81
		1e-8	sing	553	82
		1e-10	sing	562	86
	1e-2	1e-6	**26**	265	91
		1e-8	**35**	270	94
		1e-10	**60**	266	93
	1e-4	1e-6	51	2516	369
		1e-8	68	2535	393

TAB. 9.2 – Comparaison de différentes consistances partielles. Les meilleurs résultats sont en gras. Un 0 dans la colonne w_2 signifie que `2B`, `Box`, ou `2B+Box` sont utilisées. Les cases `sing` correspondent à des solutions multiples qui conduisent à une explosion combinatoire (cf 9.6.2).

9.5.3 Tests principaux

Les principales conclusions des tests, dont les résultats sont présentés sur le tableau 9.3, sont les suivantes :

- `IBB` surpasse toujours la résolution globale, ce qui montre l'intérêt d'exploiter la structure du problème. Un, deux ou trois ordres de grandeur peuvent être gagnés en performance. Même avec les très petits domaines, les gains peuvent être significatifs (voir `Sierpinski3`)[4].
- Le filtrage inter-blocs est toujours contre-productif et même parfois très mauvais (cf `Tangent`). Les essais avec la consistance `3B` montrent que la perte de temps due au filtrage inter-blocs est alors réduite.
- L'exploitation de la structure du `DAG` par la condition de recalcul est très bénéfique.

[4] La résolution globale se compare avantageusement avec `IBB` sur le problème `Star` avec de très petits domaines. Ceci est dû à la plus grande précision requise pour rendre `IBB` complet (voir partie 9.6). Avec la même précision, la résolution globale met 75 s à trouver les solutions.

		Très petit		Petit		Moyen		Grand	
		¬IBF	IBF	¬IBF	IBF	¬IBF	IBF	¬IBF	IBF
Chair	Global	XXS		XXS		XXS		XXS	
	BT	3.3	XXS	3.2	XXS	9.4	XXS	12.4	XXS
	BT+	2.4	XXS	2.3	XXS	4.5	XXS	4.7	XXS
	GBJ	2.4	XXS	2.3	XXS	4.5	XXS	4.7	XXS
	GPB	XXI	XXS	XXI	XXS	XXI	XXS	XXI	XXS
Mechanism	Global	XXS		XXS		XXS		XXS	
	BT	0.17	14.1	0.6	15.0	2.8	18.7	13.3	32.8
	BT+	0.11	14.1	0.4	13.6	2.6	17.2	13.1	30.6
	GBJ	0.10	14.1	0.4	13.5	2.6	17.3	13.1	30.4
	GPB	0.10	14.2	0.4	13.3	2.7	17.4	13.1	30.5
	3B(GBJ)	0.23	0.68	1.7	2.3	9.7	11	83	88
Ponts	Global	0.73		32		82		110	
	BT	0.16	0.63	2.38	4.2	6.5	10.6	9.1	14.6
	BT+	0.16	0.63	2.36	4.2	6.1	10.2	8.8	14.7
	GBJ	0.17	0.58	2.35	4.1	6.0	10.4	8.7	14.4
	GPB	0.22	0.61	2.37	4.1	6.3	10.4	8.7	14.4
	3B(GBJ)	0.3	0.6	12	15	25	31	49	59
Hour-glass	Global	0.12		1.89		1.47		22.77	
	BT+	0.03	0.88	0.03	1.64	0.06	1.00	0.06	1.21
	GBJ	0.04	0.75	0.03	1.60	0.02	0.83	0.06	1.19
	GPB	0.05	0.73	0.03	1.61	0.05	0.88	0.05	1.15
	3B(GBJ)	0.03	0.3	0.05	0.6	0.05	0.2	0.1	0.4
Sierpinski3	Global	3.1		>54000		>54000		>54000	
	3B(BT)	0.1	1.32	12.3	160	96	788	136	1094
	3B(BT+)	0.1	1.32	12.7	160	67	703	93	928
	3B(GBJ)	0.1	1.32	12	166	61	682	85	916
	3B(GPB)	XXI	XXI	XXI	XXI	XXI	XXI	XXI	XXI
Tangent	Global	0.5		35		39		46	
	BT+	0.05	1.26	0.11	1.89	0.13	7.63	0.20	8.15
	GBJ	0.07	1.17	0.11	1.89	0.14	7.69	0.19	8.00
	GPB	0.07	1.19	0.10	1.93	0.11	7.69	0.22	8.04
	3B(GBJ)	0.2	0.7	0.2	1.3	0.2	1.3	0.3	1.7
Tetra	Global	2.15		92		197		406	
	BT+	0.14	0.74	1.08	4.00	2.37	7.01	4.73	13.56
	GBJ	0.14	0.67	1.10	3.87	2.30	6.80	4.74	13.20
	GPB	0.16	0.65	1.11	3.90	2.29	6.71	4.72	13.19
Star	Global	8.7		2908		2068		1987	
	BT	9.96	70	40.2	137	81.4	241	80.3	240
	BT+	9.96	70	29.5	99.6	78.1	102	78	102
	GBJ	9.96	70	29.1	99.6	77.9	102	77.9	102
	GPB	9.96	70	29.4	99.6	49.3	102	49	102

TAB. 9.3 – Résultats expérimentaux. `BT+` est `IBB-BT` avec la condition de recalcul. Pour chaque algorithme et chaque taille de domaine, les temps sont donnés avec (IBF) et sans (¬IBF) le filtrage inter-blocs *ibf*. Tous les temps sont en secondes et ont été obtenus sur un `PentiumIII 935 Mhz` sous Linux. Les temps présentés ont été obtenus avec `2B+Box`, souvent meilleurs qu'avec `3B` sauf pour Sierpinski3. Les lignes `3B(GBJ)` montrent les temps avec `IBB-GBJ` et `3B` quand ils sont compétitifs.

Les cases dans le tableau 9.3 contenant XXI ou XXS correspondent à un échec dans la résolution. XXS correspond à un échec dû à `IlogSolver` où une taille maximum est dépassée. XXI vient d'un bogue dans la gestion mémoire des nogoods dans `IBB-GPB`.

Pour affiner nos conclusions, le tableau 9.4 donne les statistiques obtenues sur le nombre de sauts (backjumps) réalisés par `IBB-GBJ`. On notera qu'aucun saut n'a été observé avec les quatre autres problèmes.

Ces expérimentations montrent un résultat significatif. La plupart des sauts disparaissent quand on utilise un filtrage inter-blocs, ce qui rappelle des résultats similaires obtenus en domaines finis avec MAC-CBJ [Bessière et Régin, 1996]. Cependant, le prix à payer pour ce filtrage inter-blocs est dans ces essais trop important pour être rentable.

Sierpinski3-Grand montre cette tendance : 2114 des 2118 sauts sont éliminés par le filtrage inter-blocs, mais l'algorithme correspondant est 10 fois plus lent que IBB-GBJ!

	Filtrage IB	T.P.	Petit	Moyen	Grand
Ponts	non	0	0	1	0
	oui	0	0	0	0
Mechanism	non	3	4	0	0
	oui	0	0	0	0
Star	non	0	2	6	6
	oui	0	0	0	0
Sierpinski3	non	0	12	829	2118
	oui	0	0	5	4

TAB. 9.4 – Nombre de sauts avec et sans filtrage inter-blocs

9.6 Discussion

Deux difficultés sont apparues dans l'utilisation des techniques de résolution par intervalles avec IBB. Elles sont détaillées ci-dessous.

9.6.1 Heuristique du point milieu

Cette heuristique (cf partie 9.3.2) n'est pas satisfaisante puisque des solutions peuvent être perdues, rendant le processus de recherche incomplet. Quand nous l'avons introduite [Bliek *et al.*, 1998], les exemples étaient petits et ce cas ne s'est pas produit. Depuis, il est apparu avec les exemples Star, Sierpinski3 et Chair. Le problème a été résolu avec Star et Sierpinski3 en augmentant la précision (c.-à-d, en diminuant w_1). Des modifications ad hoc du système d'équations ont dû être effectuées pour résoudre le problème sur l'exemple Chair.

Une solution robuste consisterait à introduire des intervalles constants dans les équations au lieu des points milieux (ce qui n'est actuellement pas possible avec IlogSolver). Nous pensons que le surcoût en temps devrait être faible dans le schéma de résolution par filtrage et bissection. Par contre, les tests d'unicité actuels ne pourraient plus être réalisés car ils ont besoin d'un système carré.

9.6.2 Gérer les solutions multiples

Une autre limite des techniques d'intervalles est aggravée par IBB. Les solutions multiples arrivent quand plusieurs boîtes atomiques proches sont trouvées : une seule contient une solution et le filtrage n'a pu écarter les autres. Même quand le nombre de solutions multiples est petit, l'effet multiplicatif dû à la combinaison des solutions partielles par IBB peut rendre le problème impraticable.

Une solution consiste à augmenter la précision (i.e., réduire w_1), ce qui résout certains cas. Mixer plusieurs techniques de filtrage, comme 2B+Box, réduit aussi le phénomène. (Les cases *sing* dans le tableau 9.2 avec 2B proviennent de ce phénomène.) Nous avons implanté une méthode pour détecter les solutions multiples en ne gardant que la première des solutions trouvées à une précision epsilon près. Ceci a résolu le problème dans la plupart des cas. Mais quelques cas pathologiques restent, à cause d'une interaction avec l'heuristique du point milieu. Le point milieu du premier intervalle atomique trouvé peut ne pas être une solution et empêcher la résolution des blocs en aval. Il serait sans doute plus robuste de prendre l'union des solutions multiples.

9.7 Conclusion

Cet article a détaillé le cadre générique du retour arrière inter-blocs `IBB` pour résoudre les CSP numériques. Nous avons implanté trois schémas de retour arrière (chronologique `BT`, basé sur le graphe de blocs `GBJ` et de type *partial order backtracking* `GPB`). Chaque schéma peut incorporer une condition de recalcul qui évite certains appels inutiles au résolveur. Chaque schéma peut aussi utiliser un filtrage inter-blocs.

Des séries de tests expérimentaux ont été réalisées sur un banc d'essais de taille raisonnable et contenant des équations non linéaires. En voici les principaux résultats. D'abord, toutes les variantes de `IBB` peuvent gagner plusieurs ordres de grandeur par rapport à une résolution globale. Ensuite, le choix du schéma de retour-arrière n'est pas apparu crucial. Enfin, exploiter la structure du graphe de blocs pendant le retour-arrière peut améliorer la performance sans créer de surcoût. Ceci nous conduit à proposer l'utilisation de la version `IBB-GBJ` présentée dans cet article.

Un autre résultat de cet article est que le filtrage inter-blocs est apparu contre-productif. Cela tend à montrer qu'un filtrage global qui ne tient pas compte de la structure accomplit beaucoup de travail inutile. La prochaine étape de notre travail est la mise en œuvre d'une résolution avec des intervalles constants pour enlever l'heuristique du point milieu et rendre notre implantation plus robuste.

Bibliographie

[Aggoun et Beldiceanu, 1993] A. Aggoun et N. Beldiceanu. Extending CHIP to solve complex scheduling and packing problems. *Mathl. Comput. Modelling*, 17(7) :57–73, 1993.

[Ait-Aoudia *et al.*, 1993] S. Ait-Aoudia, R. Jegou, et D. Michelucci. Reduction of constraint systems. Dans *Compugraphic*, 1993.

[Aldous et Vazinari, 1994] D. Aldous et U. Vazinari. Go with the winners algorithms. Dans *Proc. STOC*, 1994.

[Anagnostopoulos *et al.*, 2003] A. Anagnostopoulos, L. Michel, P. Van Hentenryck, et Y. Vergados. A simulated annealing approach to the traveling tournament problem. Dans *Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems, CPAIOR'2003*, pages 80–91, Montréal, Canada, 2003.

[Astier et Saada, 1994] M. Astier et J. Saada. Maintien de solution par dichotomie parallèle. Rapport de stage, Ecole Nationale des Ponts et Chaussées, 1994.

[Bäck, 1997] Thomas Bäck. Self-adaptation. Dans T. Bäck, D. B. Fogel, et Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C7.1 :1–14. IOP Publishing and Oxford University Press, 1997.

[Ballesta, 1994] Philippe Ballesta. *Contraintes et objets : clefs de voûte d'un outl d'aide à la composition musicale ?* Thèse de doctorat, Université du Maine (Le Mans), décembre 1994.

[Baluja et Caruana, 1995] Shumeet Baluja et Rich Caruana. Removing the genetics from the standard genetic algorithm. Dans A. Prieditis et S. Russel, editors, *The Int. Conf. on Machine Learning 1995*, pages 38–46, San Mateo, CA, 1995. Morgan Kaufmann Publishers.

[Bel *et al.*, 1992] G. Bel, E. Bensana, K. Ghedira, D. Lesaint, T. Schiex, G. Verfaillie, C.Gaspin, R. Martin-Clouaire, J.P. Rellier, P. Berlandier, B. Neveu, B. Trousse, H. Fargier, J. Lang, P. David, P. Janssen, T. Kokeny, M-C. Vilarem, et P. Jégou. Représentation et traitement pratique de la flexibilité dans les problèmes sous contraintes. Dans *Actes des Journées nationales du PRC-IA*, Marseille, octobre 1992.

[Bellicha *et al.*, 1995] A. Bellicha, B. Neveu, B. Trousse, C. Bessière, C. Gaspin, D. Lesaint, F. Bouquet, F Dupin de Saint-Cyr, G. Trombettoni, G. Verfaillie, H. Fargier, J-C. Régin, J-P. Rellier, J. Amilhastre, J. Lang, K. Ghedira, M-C. Vilarem, M. Cooper, O. Lhomme, P. Charman, P. David, P. Janssen, P. Jégou, P. Berlandier, R. Martin-Clouaire, T. Schiex, et T. Kokeny. Autour du problème de satisfaction de contraintes. Dans *Acte des Cinquièmes journées nationales PRC-GDR I.A.*, Nancy, février 1995.

[Benhamou *et al.*, 1994] F. Benhamou, D. McAllester, et P. Van Hentenryck. CLP(intervals) Revisited. Dans *International Symposium on Logic Programming, ILPS'94*, pages 124–138, 1994.

[Berlandier et Neveu, 1997] Pierre Berlandier et Bertrand Neveu. Problem partition and solver coordination in distributed constraint satisfaction. Dans J. Geller, H. Kitano, et C.B. Suttner, editors, *Parallel Processing for Artificial Intelligence 3*. Elsevier, 1997.

[Berlandier, 1992a] P. Berlandier. PROSE : Une boîte à outils fonctionnelle pour l'interprétation de contraintes. Rapport technique 145, INRIA-Sophia Antipolis, 1992.

[Berlandier, 1992b] Pierre Berlandier. *Etude de mécanismes d'interprétation de contraintes et de leur intégration dans un langage à base de connaissances.* Thèse de doctorat, Université de Nice-Sophia Antipolis, 1992.

[Berlandier, 1995] P. Berlandier. Filtrage de problèmes par consistance de chemin restreinte. *Revue d'intelligence artificielle*, 9(3) :225–238, 1995.

[Bessière *et al.*, 1999] C. Bessière, P. Meseguer, E.C. Freuder, et J. Larrosa. Forward checking pour les contraintes non binaires. Dans *actes JNPC'99*, pages 27–34, Lyon, France, 1999.

[Bessière et Régin, 1996] C. Bessière et J.C. Régin. Mac and combined heuristics : two reasons to forsake fc (and cbj ?) on hard problems. Dans *Proceedings CP'96*, volume 1118 of *LNCS*, pages 61–75, Cambridge MA,USA, 1996. Springer.

[Bessière et Régin, 1997] Christian Bessière et Jean-Charles Régin. Arc consistency for general constraint networks : preliminary results. Dans *Proc of International Joint Conference on Artificial Ingelligence IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[Blay-Fornarino et Pinna-Déry, 1990] M. Blay-Fornarino et A-M. Pinna-Déry. *Un modèle objet logique et relationnel : Le langage Othelo.* Thèse de doctorat, Université de Nice-Sophia Antipolis, avril 1990.

[Bliek *et al.*, 1998] C. Bliek, B. Neveu, et G. Trombettoni. Using graph decomposition for solving continuous csps. Dans *Principles and Practice of Constraint Programming, CP'98*, volume 1520 of *LNCS*, pages 102–116, Pise, Italie, 1998. Springer.

[Bliek, 1998] Christian Bliek. Generalizing partial order and dynamic backtracking. Dans *Fifth National Conference on Artificial Intelligence – AAAI'98*, 1998.

[Bondyfalat *et al.*, 1999] D. Bondyfalat, B. Mourrain, et T. Papadopoulo. An application of automatic theorem proving in computer vision. Dans *2nd International Workshop on Automated Deduction in Geometry, Springer-Verlag*, 1999.

[Borning *et al.*, 1992] Alan Borning, Bjorn Freeman-Benson, et Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3) :223–270, september 1992.

[Borning *et al.*, 1997] Alan Borning, Kim Marriott, Peter Stuckey, et Yi Xiao. Solving linear arithmetic constraints for user interface applications. Dans *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, 1997.

[Bouma *et al.*, 1995] W. Bouma, I. Fudos, C.M. Hoffmann, J. Cai, et R. Paige. Geometric constraint solver. *Computer Aided Design*, 27(6) :487–501, 1995.

[Bouzoubaa, 1996] Mouhssine Bouzoubaa. *Hiérarchies de contraintes : quelques approches de résolution.* Thèse de doctorat, Ecole nationale des Ponts et Chaussées, octobre 1996.

[Buchberger, 1985] B. Buchberger. An algebraic method in ideal theory. Dans *Multidimensional System Theory*, pages 184–232. Reidel Publishing Co., 1985.

[Cabon *et al.*, 1996] B. Cabon, G. Verfaillie, D. Martinez, et P. Bourret. Using Mean Field Methods for Boosting Backtrack Search in Constraint Satisfaction Problems. Dans *Proceedings of the* 12^{th} *European Conference on Artificial Intelligence*, pages 165–169, 1996.

[Carson et Impagliazzo, 1999] Ted Carson et Russell Impagliazzo. Experimentally determining regions of related solutions for graph bisection problems. Dans *IJCAI, workshop ML1 : machine learning for large scale optimization*, 1999.

[Chabert *et al.*, 2004] Gilles Chabert, Gilles Trombettoni, et Bertrand Neveu. New light on arc-consistency over continous domains. Dans *Proc. CP'04 First International Workshop on Constraint Propagation and Implementation*, 2004.

[Charman, 1995] Philippe Charman. *Gestion des contraintes géométriques pour l'aide à l'aménagement spatial.* Thèse de doctorat, Ecole Nationale des Ponts et Chaussées, novembre 1995.

[Chelouah, 1999] Rachid Chelouah. *Adaptation aux problèmes à variables continues de plusieurs métaheuristiques d'optimisation combinatoire : la méthode tabou, les algorithmes génétiques et les méthodes hybrides. Application en contrôle non destructif.* Thèse de doctorat, Université de Cergy-Pontoise, 1999.

[Chleq, 1995a] Nicolas Chleq. *Contribution à l'étude du raisonnement temporel : Usage de la résolution avec contraintes et application à l'abduction en raisonnement temporel.* Thèse de doctorat, Ecole Nationale des Ponts et Chaussées, janvier 1995.

[Chleq, 1995b] Nicolas Chleq. *Contribution à l'étude du raisonnement temporel : Usage de la résolution avec contraintes et application à l'abduction en raisonnement temporel.* Thèse de doctorat, Ecole Nationale des Ponts et Chaussées, janvier 1995.

[Collavizza *et al.*, 1999] H. Collavizza, F. Delobel, et M. Rueher. Extending consistent domains of numeric csp. Dans *proc of IJCAI-99*, Stockholm, Suède, 1999.

[Connolly, 1990] D. T. Connolly. An improved annealing scheme for the qap. *European Journal of Operational Research*, 46 :93–100, 1990.

[de Givry *et al.*, 1997] S. de Givry, G. Verfaillie, et T. Schiex. Bounding the optimum of constraint optimization problems. Dans *Proc. CP97*, number 1330 in LNCS, 1997.

[Debruyne, 1998] Romuald Debruyne. *Etude des consistances locales pour les problèmes de satisfaction de contraintes de grande taille.* Thèse de doctorat, LIRMM-Université de Montpellier II, December 1998.

[Dechter et Pearl, 1988] R. Dechter et J. Pearl. Network–based heuristics for constraint–satisfaction problems. *Artificial Intelligence*, 34 :1–38, 1988.

[Dechter, 1990] Rina Dechter. Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3) :273–312, January 1990.

[Dechter, 2003] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[DiGaspero et Schaerf, 2000] L. DiGaspero et A. Schaerf. Easylocal++ : An object oriented framework for flexible design of local search algorithms. Technical Report UDMI/13, Universita degli Studie di Udine, 2000.

[Dimitriou et Impagliazzo, 1996] Tassos Dimitriou et Russell Impagliazzo. Towards an analysis of local optimization algorithms. Dans *Proc. STOC*, 1996.

[Dimitriou et Impagliazzo, 1998] Tassos Dimitriou et Russell Impagliazzo. Go with the winners for graph bisection. Dans *Proc. SODA*, pages 510–520, 1998.

[Dimitriou, 2002] Tassos Dimitriou. Characterizing the space of cliques in random graphs using "go with the winners". Dans *Proc. AMAI*, 2002.

[Djossou, 1993] C. Djossou. *Approche multi-processus pour la coopération entre systèmes à base de connaissances.* Thèse de doctorat, Université de Nice Sophia-Antipolis, février 1993.

[Dorne et Hao, 1998] Raphaël Dorne et Jin-Kao Hao. Tabu search for graph coloring, t-colorings and set t-colorings. Dans I.H. Osman S. Voss, S.Martello et C. Roucairol, editors, *Meta-heuristics : Advances and Trends in Local Search Paradigms for Optimization*, pages 77–92. Kluwer Academic Publishers, 1998.

[Dréo *et al.*, 2003] J. Dréo, A. Pétrowski, P. Siarry, et E. Taillard. *Métaheuristiques pour l'optimisation difficile.* Eyrolles, 2003.

[Dueck et Scheuer, 1990] G. Dueck et T. Scheuer. Threshold accepting : a general purpose algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90 :161–175, 1990.

[Dufourd *et al.*, 1998] J.-F. Dufourd, P. Mathis, et P. Schreck. Geometric construction by assembling solved subfigures. *Artificial Intelligence*, 99(1) :73–119, 1998.

[Dulmage et Mendelsohn, 1958] A.L. Dulmage et N.S. Mendelsohn. Covering of bipartite graphs. *Canad. J. Math.*, 10 :517–534, 1958.

[Durand, 1998] C. Durand. *Symbolic and Numerical Techniques For Constraint Solving.* PhD thesis, Purdue University, 1998.

[Eiben et Ruttkay, 1997] A.E. Eiben et Zs. Ruttkay. Constraint satisfaction problems. Dans T. Bäck, D. Fogel, et M. Michalewicz, editors, *Handbook of Evolutionary Algorithms*, pages C5.7 :1–C5.7 :8. IOP Publishing Ltd, 1997.

[Eiben et van der Hauw, 1997] A. E. Eiben et J. K. van der Hauw. Adaptive penalties for evolutionary graph coloring. Dans J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, et D. Snyers, editors, *Artificial Evolution – Third European Conference*, pages 95–106. Springer, 1997. LNCS 1363.

[Eiben, 2001] A. E. Eiben. Evolutionary algorithms and constraint satisfaction : Definitions, survey, methodology, and research directions. Dans L. Kallel, B. Naudts, et A. Rogers, editors, *Theoretical Aspects of Evolutionary Computing*, Natural Computing series, pages 13–58. Springer, 2001.

[Feo et Resende, 1995] T. Feo et M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6 :109–135, 1995.

[Fogel *et al.*, 1966] L. J. Fogel, A. J. Owens, et M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley, New York, 1966.

[Ford et Fulkerson, 1962] L.R. Ford et D.R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[Fornarino, 1991] Claude Fornarino. *ObjectiveAda : une extension objet du langage Ada. Application à un environnement de conception de systèmes experts.* Thèse de doctorat, Université de Nice-Sophia Antipolis, 1991.

[Freeman-Benson *et al.*, 1990] Bjorn Freeman-Benson, John Maloney, et Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1) :54–63, January 1990.

[Frost et Dechter, 1995] D. Frost et R. Dechter. Look-ahead value ordering for constraint satisfaction problems. Dans *Proceedings of the* 14^{th} *International Joint Conference on Artificial Intelligence*, 1995.

[Frost, 1997] Daniel H. Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems.* PhD thesis, University of California - Irvine, 1997.

[Galinier et Hao, 1999] Philippe Galinier et Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4) :379–397, 1999.

[Geelen, 1992] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. Dans *Proceedings of the* 10^{th} *European Conference on Artificial Intelligence*, 1992.

[Georget et Codognet, 1999] Yan Georget et Philippe Codognet. Constraint retraction in clp(fd) : Formal framework and performance results. *Constraints*, 4(1) :5–42, 1999.

[Ginsberg, 1993] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

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

[Glover, 1986] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operation Research*, 13 :533–549, 1986.

[Golumb et Baumert, 1965] S. Golumb et L. Baumert. Backtrack programming. Dans *Journal of the ACM*, pages 516–524, 1965.

[Gomes *et al.*, 1998] Carla Gomes, Bart Selman, et Henry Kautz. Boosting combinatorial search through randomization. Dans *Proc of AAAI 98*, 1998.

[Goualard, 2000] F. Goualard. *Langages et environnements en proprammation par contraintes d'intervalles.* Thèse de doctorat, Université de Nantes, juillet 2000.

[Grandon, 2003] C. Grandon. Using constraint programming for tackling uncertainty in numeric constraint satisfaction problems. Rapport de stage, Projet COPRIN - INRIA Sophia Antipolis, 2003.

[Granvilliers, 2003] L. Granvilliers. *RealPaver User's Manual.* IRIN, Nantes, juillet 2003.

[Hamadi, 1999] Youssef Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués.* Thèse de doctorat, Université de Montpellier II, 1999.

[Hao *et al.*, 1998] J.K. Hao, R. Dorne, et P. Galinier. Tabu search for frequency assignment in mobile radio networks. *Journal of Heuristics*, 4(1) :47–62, 1998.

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

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

[Harvey et Ginsberg, 1995] W. D. Harvey et M. L. Ginsberg. Limited discrepancy search. Dans *Proceedings of* 14^{th} *International Joint Conference on Artificial Intelligence*, 1995.

[Hendrickson, 1992] Bruce Hendrickson. Conditions for unique realizations. *SIAM j Computing*, 21(1) :65–84, 1992.

[Hoffmann *et al.*, 1997] C.M. Hoffmann, A. Lomonosov, et M. Sitharam. Finding solvable subsets of constraint graphs. Dans *Principles and Practice of Constraint Programming CP'97*, pages 463–477, 1997.

[Hoffmann *et al.*, 2001] C.M. Hoffmann, A. Lomonosov, et M. Sitharam. Decomposition plans for geometric constraint systems. *Proc. J. Symbolic Computation*, 31(4) :367–427, 2001.

[Holland, 1975] J. H. Holland. *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* The University of Michigan, 1st edition, 1975.

[Hosobe, 2000] Hiroshi Hosobe. A scalable linear constraint solver for user interface construction. Dans *Proceedings of CP'2000,Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *LNCS*, pages 218–232, Singapour, 2000. Springer.

[Hudson, 1994] Scott Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13(3) :209–239, July 1994.

[Hulubei, 1999] T. Hulubei. The CSP Library. http ://www.cs.unh.edu/~tudor/csp/, University of New Hampshire, 1999.

[ILO, 1997] ILOG, 2 Av. Galliéni, F-94253 Gentilly. ILOG SOLVER *Version 4.0, Users' Reference Manual*, 1997.

[Jaffar et Maher, 1994] Joxan Jaffar et Michael J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20 :503–581, 1994.

[Jaulin *et al.*, 2001] L. Jaulin, M. Kieffer, O. Didrit, et E. Walter. *Applied Interval Analysis.* Springer-Verlag, 2001.

[Jermann *et al.*, 2000] C. Jermann, G. Trombettoni, B. Neveu, et M. Rueher. A constraint programming approach for solving rigid geometric systems. Dans *Proceedings of CP'2000,Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *LNCS*, pages 119–134, Singapour, 2000. Springer.

[Jermann *et al.*, 2003a] C. Jermann, B. Neveu, et G. Trombettoni. Algorithms for Identifying Rigid Subsystems in Geometric Constraint Systems. Dans *Proc.of IJCAI'03, International Joint Conference on Artificial Intelligence*, pages 233–238, Acapulco, Mexico, 2003.

[Jermann *et al.*, 2003b] Christophe Jermann, Bertrand Neveu, et Gilles Trombettoni. Inter-block backtracking : Exploiting the structure in continuous csps. Dans *Proc of 2nd International Workshop on Global Constrained Optimization and Constraint Satisfaction, COCOS'03*, Lausanne, Switzerland, 2003.

[Jermann *et al.*, 2004a] C. Jermann, B. Neveu, et G. Trombettoni. A new structural rigidity for geometric constraints systems. Dans *Automated Deduction in Geometry, Revised papers of 4th International Workshop ,ADG'02*, volume 2930 of *LNAI*, pages 87–105, 2004.

[Jermann *et al.*, 2004b] Christophe Jermann, Bertrand Neveu, et Gilles Trombettoni. Algorithmes pour la détection de rigidités dans les csp géométriques. *Journal Electronique d'Intelligence Artificielle JEDAI*, 2, 2004.

[Jermann, 2002] Christophe Jermann. *Résolution de contraintes géométriques par rigidification récursive et propagation d'intervalles.* Thèse de doctorat, Université de Nice-Sophia Antipolis, décembre 2002.

[Johnson et Trick, 1996] David S. Johnson et Michael A. Trick. *Cliques, Coloring, and Satisfiability : Second DIMACS Implementation Challenge, 1993*, volume 26 of *Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1996.

[Jussien *et al.*, 2000] Narendra Jussien, Romuald Debruyne, et Patrice Boizumault. Maintaining Arc-Consistency within Dynamic Backtracking. Dans *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in LNCS, pages 249–261. Springer, 2000.

[Jussien, 1997] Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques.* Thèse de doctorat, Université de Rennes I, Rennes, France, octobre 1997.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. Gellat, et M. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–680, 1983.

[König, 1916] D. König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. Dans *Math Ann 77*, pages 453–465, 1916.

[Kolen, 1999] A. Kolen. A genetic algorithm for frequency assignment. Technical report, Universiteit Maastricht, 1999.

[Korf, 1985] R. E. Korf. Depth-first Iterative-Deepening : an optimal admissible tree search. *Artificial Intelligence*, 27(1) :97–109, 1985.

[Korf, 1996] R. Korf. Improved limited discrepancy search. Dans *Proceedings of the* 13^{th} *AAAI Conference*, 1996.

[Koster *et al.*, 1999] A. Koster, C. Van Hoesel, et A. Kolen. Solving frequency assignment problems via tree-decomposition. Technical Report 99-011, Universiteit Maastricht, 1999.

[Kramer, 1992] Glenn Kramer. *Solving Geometric Constraint Systems.* MIT Press, 1992.

[Kumar *et al.*, 1994] V. Kumar, A. Grama, et V. Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1) :60–79, 1994.

[Laburthe et Caseau, 1998] François Laburthe et Yves Caseau. SALSA : A language for search algorithms. Dans *Proc. CP'98*, number 1520 in LNCS, pages 310–324, 1998.

[Lahaye, 1934] E. Lahaye. Une méthode de résolution d'une catégorie d'équations transcendantes. *Compte-rendu des Séances de L'Académie des Sciences*, 198 :1840–1842, 1934.

[Laman, 1970] G. Laman. On graphs and rigidity of plane skeletal structures. *J. Eng. Math.*, 4 :331–340, 1970.

[Lamure et Michelucci, 1998] H. Lamure et D. Michelucci. Qualitative study of geometric constraints. Dans B. Brüderlin et D. Roller, editors, *Geometric Constraint Solving and Applications*, pages 234–258. Springer, 1998.

[Larrosa et Meseguer, 1995] J. Larrosa et P. Meseguer. Optimization-based Heuristics for Maximal Constraint Satisfaction. Dans *Constraint Programming CP'95*, volume LNCS 976, pages 103–120, 1995.

[Latham et Middleditch, 1996] R.S. Latham et A.E. Middleditch. Connectivity analysis : A tool for processing geometric constraints. *Computer Aided Design*, 28(11) :917–928, 1996.

[Le Ménec, 1994] Stéphane Le Ménec. *Théorie des jeux dynamiques et techniques de programmation avancée appliquées au duel aérien à moyenne distance.* Thèse de doctorat, Université de Nice Sophia-Antipolis, novembre 1994.

[Lebastard, 1993a] F. Lebastard. CHOOE : Un gestionnaire d'environnement distribué. Rapport Technique 93-22, CERMICS-INRIA Sophia Antipolis, 1993.

[Lebastard, 1993b] F. Lebastard. *DRIVER : Une couche objet virtuelle persistante pour le raisonnement sur les bases de données relationnelles.* Thèse de doctorat, INSA de Lyon, mars 1993.

[Lebbah *et al.*, 2003] Yahia Lebbah, Claude Michel, et Michel Rueher. Une combinaison de consistances locales avec un filtrage global sur des relaxations linéaires. Dans *Actes JNPC'03 (9èmes Journées Nationales pour la résolution de Problèmes NP-Complets)*, pages 217–232, Amiens, juin 2003.

[Lebbah, 2002] Y. Lebbah. *Le système ICOS.* projet COPRIN, INRIA Sophia-Antipolis, 2002.

[Lejouad, 1994] W. Lejouad. *Etude et application des techniques de distribution pour un générateur de systèmes à base de connaissances.* Thèse de doctorat, Université de Nice Sophia-Antipolis, octobre 1994.

[Lhomme, 1993] O. Lhomme. Consistency techniques for numeric CSPs. Dans *IJCAI'93 : Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.

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

[Madeline et Neveu, 2001] B. Madeline et B. Neveu. Méthodes de recherche arborescente comparées aux algorithmes génétiques sur des problèmes de coloriage de graphes surcontraints. Dans *actes de JNPC'01*, pages 185–195, Toulouse, France, juin 2001.

[Madeline, 2002] Blaise Madeline. *Algorithmes évolutionnaires et résolution de problèmes de satisfaction de contraintes en domaines finis.* Thèse de doctorat, Université de Nice-Sophia Antipolis, décembre 2002.

[McAllester, 1993] D.A. McAllester. Partial order backtracking. Research Note, Artificial Intelligence Laboratory, MIT, 1993. ftp ://ftp.ai.mit.edu/people/dam/dynamic.ps.

[Merlet, 2001] J-P. Merlet. *ALIAS : Algorithm Library of Interval Analysis for equation Systems.* projet COPRIN, INRIA Sophia-Antipolis, 2001.

[Merlet, 2002] Jean-Pierre Merlet. Optimal design for the micro robot. Dans *in IEEE Int. Conf. on Robotics and Automation*, 2002.

[Meseguer et Walsh, 1998] P. Meseguer et T. Walsh. Interleaved and Discrepancy Based Search. Dans *Proc. of the* 13^{th} *European Conf. on Artificial Intelligence*, 1998.

[Meseguer, 1997] P. Meseguer. Interleaved Depth-First Search. Dans *Proceedings of the* 15^{th} *International Joint Conference on Artificial Intelligence*, pages 1382–1387, 1997.

[Michel et Van Hentenryck, 2001] L. Michel et P. Van Hentenryck. Localizer++ : An open library for local search. Technical Report CS-01-02, Brown University, 2001.

[Milano, 2003] Michela Milano, editor. *Constraint and Integer Programming, Toward a Unified Methodology.* Kluwer, 2003.

[Minton *et al.*, 1992] S. Minton, M. Johnston, A. Philips, et P. Laird. Minimizing conflict : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58 :161–205, 1992.

[Mladenovic et Hansen, 1997] N. Mladenovic et P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11) :1097–1100, 1997.

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

[Montanari, 1974] U. Montanari. Networks of constraints : Fundamental properties and application to picture processing. *Information Science*, 7(3) :95–132, 1974.

[Morgenstern, 1996] Craig Morgenstern. Distributed coloration neighborhood search. Dans David S. Johnson et Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability : Second DIMACS Implementation Challenge, 1993*, volume 26, pages 335–357. American Mathematical Society, 1996.

[Moscato, 1999] Pablo Moscato. Memetic algorithms : a short introduction. Dans David Corne, Marco Dorigo, et Fred Glover, editors, *New Ideas in Optimization*, pages 219–234. McGraw Hill, 1999.

[Myers *et al.*, 2000] Brad Myers, Scott E. Hudson, et Randy Pausch. Past, present and future of user interface software tools. *ACM Transactions on Computer-Human-Interaction*, 7(1) :3–28, 2000.

[Neumaier, 2004] Arnold Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 13 :271–369, 2004.

[Neveu *et al.*, 1990] B. Neveu, B. Trousse, et O. Corby. SMECI : An expert system shell that fits engineering design. Dans *3^er Simposium Internacional de Inteligencia Artificial*, Monterrey, Mexique, 1990.

[Neveu *et al.*, 2004] Bertrand Neveu, Gilles Trombettoni, et Fred Glover. Id-walk : A candidate list strategy with a simple diversification device. Dans *Proc. 10th International Conference on Principles and Practice of Constraint Programming - CP 2004*, volume LNCS 3258, pages 423–437. Springer, 2004.

[Neveu et Berlandier, 1994] B. Neveu et P. Berlandier. Maintaining arc consistency through constraint retraction. Dans *Proc. IEEE International Conference on Tools for Artificial Intelligence (TAI)*, pages 426–431, New Orleans, LA, 1994.

[Neveu et Haren, 1986] Bertrand Neveu et Pierre Haren. Smeci : An expert system for civil engineering design. Dans *Proc of 1st conference on Applications of Artificial Intelligence to Engineering Problems*, Southampton, UK, 1986.

[Neveu et Trombettoni, 2003a] Bertrand Neveu et Gilles Trombettoni. Hybridation de gww avec de la recherche locale. Dans *9èmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets, JNPC'03*, pages 277–292, Amiens, France, 2003.

[Neveu et Trombettoni, 2003b] Bertrand Neveu et Gilles Trombettoni. Incop : An Open Library for INcomplete Combinatorial OPtimization. Dans *Proc. International Conference on Principles Constraint Programming, CP'03*, volume 2833 of *LNCS*, pages 909–913, Kinsale,Ireland, 2003. Springer.

[Neveu et Trombettoni, 2003c] Bertrand Neveu et Gilles Trombettoni. Une bibliothèque de recherche locale pour l'optimisation combinatoire. Dans *Congrès de l'association française de recherche opérationnelle et d'aide à la décision, ROADEF'2003*, pages 68–69, Avignon, France, 2003.

[Neveu et Trombettoni, 2003d] Bertrand Neveu et Gilles Trombettoni. When Local Search Goes with the Winners. Dans *Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems, CPAIOR'2003*, pages 180–194, Montréal, Canada, 2003.

[Neveu et Trombettoni, 2004] Bertrand Neveu et Gilles Trombettoni. Hybridation de gww avec de la recherche locale. *Journal Electronique d'Intelligence Artificielle JEDAI*, 3, 2004.

[Neveu, 1987] Bertrand Neveu. Export : an expert system in breakwater design. Dans *Proc of ORIA'97 : Arificial Intelligence and Sea*, pages I1–I7, Marseille,France, 1987.

[Neveu, 1990] Bertrand Neveu. Object oriented programming for an expert system shell implementation. Dans *Proc of AAAI'90 workshop on Object-Oriented Programming in AI*, 1990.

[Nielsen, 1998] P. K. Nielsen. *SCOOP 2.0 Reference Manual*. SINTEF Report 42A98001, 1998.

[Ortuzar et Serré, 2003] A. Ortuzar et P. Serré. Approches non cartésiennes pour la résolution de problèmes géométriques modélisés par contraintes. Dans *Journés GTMG du GDR ALP et de l'AFIG*, Aix en Provence (France), 2003.

[Phan et Skiena, 2002] V. Phan et S. Skiena. Coloring graphs with a general heuristic search engine. Dans *Computational Symposium of Graph Coloring and Generalizations*, 2002.

[Pothen et Fan, 1990] A. Pothen et C.J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. on Math. Soft.*, 16(4) :303–324, 1990.

[Prcovic *et al.*, 1996] Nicolas Prcovic, Bertrand Neveu, et Pierre Berlandier. Distribution de l'arbre de recherche des problèmes de satisfaction de contraintes en domaines finis. Dans *Actes de CNPC'96 Résolution pratique de problèmes NP-complets*, pages 275–289, Dijon, France, mars 1996.

[Prcovic et Neveu, 1997] Nicolas Prcovic et Bertrand Neveu. Parallélisation de la recherche arborescente des csp avec distribution à profondeur variable. Dans *Journées nationales sur la résolution de problèmes NP-complets*, pages 81–86, Rennes, avril 1997.

[Prcovic et Neveu, 1998] Nicolas Prcovic et Bertrand Neveu. Recherche arborescente restreinte à un sous-espace prometteur. Dans *Journées nationales sur la résolution de problèmes NP-complets*, pages 41–46, Nantes, mai 1998.

[Prcovic et Neveu, 1999a] Nicolas Prcovic et Bertrand Neveu. Ensuring a relevant visiting order of the leaf nodes during a tree search. Dans *Principles and Practice of Constraint Programming, CP'99*, volume 1713 of *LNCS*, pages 361–374, Alexandria,VA,Etats Unis, octobre 1999. Springer.

[Prcovic et Neveu, 1999b] Nicolas Prcovic et Bertrand Neveu. Etude théorique sur les recherches entrelacée et à divergence limitée. Dans *5èmes Journées nationales sur la résolution de problèmes NP-complets*, pages 85–94, Lyon, juin 1999.

[Prcovic et Neveu, 2000] Nicolas Prcovic et Bertrand Neveu. Recherche à focalisation progressive. Dans *6ièmes Journées Nationales de la Résolution Pratique de Problèmes NP-Complets (JNPC 2000)*, pages 191–204, Marseille, France, juin 2000.

[Prcovic et Neveu, 2002] Nicolas Prcovic et Bertrand Neveu. Progressive focusing search. Dans *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 126–130, Lyon, France, juillet 2002.

[Prcovic, 1997] Nicolas Prcovic. Changing the distribution depth during a parallel tree search. Dans *Euro-Par'97*, number 1300 in LNCS, pages 1217–1220, Passau, Germany, august 1997.

[Prcovic, 1998] Nicolas Prcovic. *Recherche arborescente parallèle et séquentielle pour les problèmes de satisfaction de contraintes.* Thèse de doctorat, Ecole Nationale des Ponts et Chaussées, novembre 1998.

[Prestwich et Mudambi, 1995] Steven Prestwich et Shyam Mudambi. Improved branch and bound in constraint logic programming. Dans *Proc of CP 1995*, LNCS, pages 533–548, Cassis, France, 1995.

[Prestwich, 2000] Steven Prestwich. A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. Dans *Proceedings of CP'2000,Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *LNCS*, pages 337–352, 2000.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, 1993.

[Prosser, 1995] P. Prosser. Mac-cbj : maintaining arc consistency with conflict directed backjumping. Technical Report 95/177, Department of Computer Science, University of Strathclyde, 1995.

[Rechenberg, 1973] I. Rechenberg. Werkstatt bionik und evolutionstechnik band, 1973.

[Régin, 1994] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. Dans *Proc. AAAI'94*, pages 362–367, 1994.

[Régin, 1996] Jean-Charles Régin. Generalized arc-consistency for global cardinality constraint. Dans *Proc of AAAI'96*, pages 362–367, Seattle, WA, USA, 1996.

[Riff, 1996] María-Cristina Riff. From quasi-solutions to solution : An evolutionary algorithm to solve csp. Dans *Principles and Practice of Constraint Programming (CP96)*, volume 1118 of *LNCS*, pages 367–381, Cambridge, Massachussets, Etats-Unis, août 1996. Springer.

[Riff, 1997a] Maria-Cristina Riff. Evolutionary search guided by the constraint network to solve csp. Dans *Int. Conf. on Evolutionary Computation ICEC'97*, pages 337–342, Indianapolis, USA, april 1997.

[Riff, 1997b] Maria-Cristina Riff. *Résolution d'un problème de satisfaction de contraintes avec des algorithmes évolutionnistes.* Thèse de doctorat, Ecole Nationale des Ponts et Chaussées, décembre 1997.

[Riff, 1998] Maria-Cristina Riff. A network-based adaptive evolutionary algorithm for constraint satisfaction problems. Dans S. Voss, S. Martello, I. Osman, et C. Roucairol, editors, *Meta-heuristics : Advances and Trends in Local Search Paradigms for Optimization*, pages 269–283. Kluwer Academic Publishers, 1998.

[Ritt, 1950] J.F. Ritt. *Differential Algebra.* American Mathematical Society, 1950.

[Sabin et Freuder, 1994] D. Sabin et E. C. Freuder. Contradicting Cconventional Wisdom in Constraint Satisfaction. Dans *Proceedings of the* 10^{th} *European Conference on Artificial Intelligence*, pages 125–129, 1994.

[Sanella, 1994] M. Sanella. *Constraint Satisfaction and Debugging for Interactive User Interfaces.* PhD thesis, University of Washington, Seattle, 1994.

[Schiex et Verfaillie, 1994] T. Schiex et G. Verfaillie. Nogood Recording for static and dynamic CSP. Dans *5th IEEE International Conference on Tools with Artificial Intelligence*, pages 48–55, 1994.

[Schiex, 2000] Thomas Schiex. *Réseaux de contraintes.* PhD thesis, Université de Toulouse, 2000. Mémoire d'HDR.

[Schreck, 2002] Pascal Schreck. *Résolution symbolique de contraintes géométriques.* PhD thesis, Université Louis Pasteur, Strasbourg, 2002. mémoire de HDR.

[Selman *et al.*, 1992] B. Selman, H. Levesque, et D. Mitchell. A new method for solving hard satisfiability problems. Dans *Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[Selman *et al.*, 1996] Bart Selman, Henry Kautz, et Bram Cohen. Local search strategies for satisfiability testing. Dans David S. Johnson et Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability : Second DIMACS Implementation Challenge, 1993*, volume 26, pages 521–531, 1996.

[Shaw, 1998] Paul Shaw. Using constraint programming and local search methods to solve vehicule routing problems. Dans *Principles and Practice of Constraint Programming, CP'98*, volume 1520 of *LNCS*, pages 417–431, Pise, Italie, 1998. Springer.

[Smith et Dyer, 1996] B. M. Smith et M. E. Dyer. Locating the Phase Transition in Binary Constraint Satisfaction Problem. *Artificial Intelligence*, 81 :155–181, 1996.

[Smith et Grant, 1998] B. M. Smith et S. A. Grant. Trying Harder to Fail First. Dans *Proceeedings of the* 13^{th} *European Conference on Artificial Intelligence*, 1998.

[Sutherland, 1963] I. Sutherland. *Sketchpad : A Man-Machine Graphical Communication System.* PhD thesis, Department of Electrical Engineering, MIT, 1963.

[Törn, 1973] A.A. Törn. Global optimization as a combination of global and local search. Dans *Proc. of Computer Simulation vs Analytical Solutions for Business and Economic Models*, pages 191–206, 1973.

[Trombettoni *et al.*, 1995] G. Trombettoni, B. Neveu, P. Berlandier, M.-C. Riff, et M. Bouzoubaa. A non-diffident combinatorial optimization algorithm. Dans *CP'95 workshop on Studying and Solving Really Hard Problems*, pages 183–187, Cassis, France, September 1995.

[Trombettoni et Neveu, 1997] Gilles Trombettoni et Bertrand Neveu. Computational complexity of multi-way,dataflow constraint problems. Dans *Int. Joint Conf. on Artificial Intelligence IJCAI'97*, pages 358–363, Nagoya, Japan, august 1997.

[Trombettoni et Neveu, 2001] G. Trombettoni et B. Neveu. Links for boosting predictable interactive constraint systems. Dans *UICS'01, International Workshop on User Interaction in Constraint Satisfaction, in the CP'2001 conference*, Paphos, Chypre, décembre 2001.

[Trombettoni et Wilczkowiak, 2003] Gilles Trombettoni et Marta Wilczkowiak. Scene Reconstruction based on Constraints : Details on the Equation System Decomposition. Dans *Proc. International Conference on Constraint Programming, CP'03*, volume LNCS 2833, pages 956–961, 2003.

[Trombettoni, 1997] Gilles Trombettoni. *Algorithmes de maintien de solution par propagation locale pour les systèmes de contraintes.* Thèse de doctorat, Université de Nice Sophia-Antipolis, juin 1997.

[Trousse, 1989] Brigitte Trousse. *Coopération entre systèmes à base de connaissances et outils de CAO : l'environnement multi-agent.* Thèse de doctorat, Université de Nice - Sophia Antipolis, décembre 1989.

[Van Hentenryck *et al.*, 1992] Pascal Van Hentenryck, Yves Deville, et Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3) :291–321, 1992.

[Van Hentenryck *et al.*, 1997] P. Van Hentenryck, L. Michel, et Y. Deville. *Numerica : A Modeling Language for Global Optimization.* MIT Press, 1997.

[Van Hentenryck, 1989] Pascal Van Hentenryck. *Constraint satisfaction in logic programming.* series in logic programming. MIT press, 1989.

[Vander Zanden, 1996] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1) :30–72, 1996.

[Voß et Woodruff, 2002a] S. Voß et D. Woodruff, editors. *Optimization Software Class Libraries.* Kluwer, 2002.

[Voß et Woodruff, 2002b] S. Voß et D.L. Woodruff. Hotframe : A heuristic optimization framework. Dans Voß et Woodruff [2002a], pages 81–154.

[Voß et Woodruff, 2002c] Stefan Voß et David L. Woodruff. Optimization software class libraries. Dans Voß et Woodruff [2002a], pages 1–24.

[Voudouris et Dorne, 2002] C. Voudouris et R. Dorne. Integrating heuristic search and one-way constraints in the iOpt toolkit. Dans Voß et Woodruff [2002a], pages 177–192.

[Voudouris et Tsang, 1998] Christos Voudouris et Edward Tsang. Solving the radio link frequency assignment problem using guided local search. Dans *Nato Symposium on Frequency Assignment, Sharing and Conservation in Systems(AEROSPACE)*, 1998.

[Vu *et al.*, 2002] Xuan-Ha Vu, Djamila Sam-Haroud, et Marius-Calin Silaghi. Numerical constraint satisfaction problems with non-isolated solutions. Dans *1st International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS'2002)*, France, October 2002. COCONUT Consortium.

[Vu *et al.*, 2004] Xuan-Ha Vu, Djamila Sam-Haroud, et Boi Faltings. Clustering for disconnected solution sets of numerical csps. Dans *Joint Annual ERCIM/CoLogNet Workshop on Constraint and Logic Programming*, number 3010 in LNAI, pages 25–43. Springer, 2004.

[Walsh, 1997] T. Walsh. Depth-bounded discrepancy search. Dans *Proceedings of the* 15^{th} *International Joint Conference on Artificial Intelligence*, 1997.

[Wilczkowiak *et al.*, 2003] Marta Wilczkowiak, Gilles Trombettoni, Christophe Jermann, Peter Sturm, et Edmond Boyer. Scene Modeling Based on Constraint System Decomposition Techniques. Dans *Proc. International Conference on Computer Vision, ICCV'03*, pages 1004–1010, 2003.

[Wu, 1986] W. Wu. Basic principles of mechanical theorem proving in elementary geometries. *J. Automated Reasoning*, 2 :221–254, 1986.

[Yokoo, 2001] Makoto Yokoo. *Distributed Constraint Satisfaction*. Springer, 2001.