Application de la programmation par contraintes à des problèmes de gestion du trafic aérien

Nicolas Barnier

DOCTORAT DE L'I.N.P.T.

Spécialité : **INFORMATIQUE ET TÉLÉCOMMUNICATIONS**

Nicolas BARNIER

Laboratoires :

Laboratoire d'Optimisation Globale (CENA/ENAC)
Laboratoire d'Informatique et de Mathématiques Appliquées (ENSEEIHT)

Titre de la thèse :

Application de la programmation par contraintes à des problèmes de gestion du trafic aérien

Soutenue le :

6 décembre 2002
Salle des thèses de l'ENSEEIHT

Directeur de recherche : **Joseph NOAILLES**

JURY :	Mr.	Jean-Marc Alliot	Examinateur
	Mr.	Patrice Boizumault	Rapporteur
	Mr.	Pascal Brisset	Examinateur
	Mr.	François Laburthe	Examinateur
	Mr.	Joseph Noailles	Examinateur
	Mr.	Michel Van Caneghem	Rapporteur

MOTS CLÉS :

- Optimisation combinatoire
- Programmation par contraintes
- Implémentation
- Gestion du trafic aérien
- Modélisation

Remerciements

Nous sommes arrivés à Toulouse en 1994 tous les deux. Je commençais à étudier et lui à enseigner, mais j'avais déjà un fort penchant pour la programmation quand il m'a donné mon premier cours de λ-calcul. Évidemment, tout était plus beau et FaCiLe vu d'en haut, et je me suis retrouvé un jour dans son bureau à demander :

« *Monsieur, comment on fait pour devenir comme vous ?* »

C'est à Pascal Brisset que je dois à peu près tout. Assez patient pour supporter toutes ces années mon Caml fumeux et mes Chester délétères, assez brillant pour s'attaquer aux massifs les plus escarpés, toujours premier de cordée. Abstraction et démystification, élégance et pertinence... mais si ! l'informatique peut n'être que jubilation et transcendance. Il suffit d'avoir le bon prof'. Merci Pascal.

Je remercie chaleureusement Joseph Noailles d'avoir accepté d'être mon directeur de thèse. Sa gentillesse et son omniscience de l'optimisation m'avaient déjà conquis en cours de DEA. J'ai pu me reposer avec confiance sur sa compétence et son efficacité dans cet exercice de longue haleine.

Si Pascal a pu expérimenter sur moi diverses formes d'encadrement plus ou moins orthodoxes au sein du LOG, c'est d'abord grâce à notre chef bien-aimé, Jean-Marc Alliot, et à son adjoint, Nicolas Durand, qui nous ont défriché un espace de recherche assez libre pour nous exprimer et assez concret pour éviter de nous y noyer. Alors que je dois à Pascal d'avoir pu pénétrer les arcanes de la programmation par contraintes, je tiens d'eux à peu près tout ce que je sais du dantesque embouteillage aérien qui se trame quotidiennement au-dessus de nos têtes — et incidemment nous finance. Je les remercie tous deux vivement pour leur bienveillance[1] et leur disponibilité, ainsi que leur aisance peu commune au maniement de la chose administrative, ce qui laisse toujours un peu d'espoir au thésard fauché... Pas si fauché que ça d'ailleurs, grâce à notre mécène, le CENA, que je remercie pour m'avoir permis de travailler dans d'excellentes conditions.

Je tiens également à remercier sincèrement les autres membres du jury qui a bien voulu accepter de juger mon travail. Patrice Boizumault et Michel Van Caneghem ont eu la tâche ingrate et exigeante d'être les rapporteurs de cette thèse. J'ai été réellement touché par la rigueur et la sagacité de leur travail, au moment critique où j'avais banni le vocable « thèse » de mon entourage pour lui préférer le terme moins connoté de « lapin ». Leurs encouragements ont donc été d'une importance capitale pour ma santé mentale... Enfin, François Laburthe, dont les préoccupations coïncident plus que largement avec les nôtres et dont les travaux ont été une source d'inspiration vivifiante durant cette thèse, nous a comblé par la précision et l'opportunité de ses commentaires. J'espère avoir encore en

[1] Sauf Jean-Marc qui prétend être plutôt méchant, mais uniquement parce que la vie est injuste ☺

sa compagnie, après une épuisante journée de conférencier, de longues et passionnantes discussions sur l'état de la recherche et de l'âme.

Ceux qui ont eu l'occasion d'y séjourner n'ont pas pu l'oublier, il règne dans notre petit labo et celui des économ(ètr)es mitoyens une atmosphère toute particulière de convivialité, chaleureuse et stimulante. Je ne sais pas comment font les thésards des autres labos pour aller jusqu'au terme. J'exprime ma gratitude aux membres du LOG et du LEEA, collègues et amis, qui n'ont pas encore été remerciés : Géraud Granger, mon jumeau de thèse comme dirait Kévin ; Thomas Rivière, le nouveau grouillot qui en prend pour trois ans ; Jean-Baptiste Gotteland, attention ! s'il vous invite à boire du champagne à deux heures du matin, n'y allez pas, c'est un guet-apens ! ; David Gianazza, avec qui de fructueuses collusions de trajectoires s'annoncent ; Nathalie Lenoir, ma responsabilité dans sa reconversion en bikeuse reste à démontrer ; Christian Bontemps, j'ai des tas de points communs avec lui, mais c'est un sportif ; Karim Zbidi, nous avons discuté des heures de questions existentielles fondamentales, pourtant il a un peu le même problème que Kiki ; Marianne Raffarin, au début elle faisait du Word mais on a réussit à l'attirer dans notre bureau en lui offrant du thé, maintenant elle vient me poser des questions de LaTeX ; Kévin Guittet, lui c'est pire, il est sportif *et* poète, mais j'ai toujours eu un faible pour les Normaliens ; Nicolas Gruyer, on partage un peu le même sort, et du coup on trinque aux mêmes luttes. Tout ce petit monde m'a conforté durant ces quelques années et donne du sens au quotidien des jours ouvrables quand il lui arrive d'en perdre.

Aussi bien à l'ENAC qu'à l'N7, je dois dire que j'ai pu compter sur la bienveillance, l'efficacité et l'humour des secrétaires qui m'ont materné pendant cette thèse. Je remercie Nicole Vaissière, Annie Adelantado et Violette Anton-Roigt pour m'avoir rendu la vie plus simple et agréable.

J'ai eu la chance de pouvoir participer régulièrement à des conférences, congrès et séminaires durant ces quelques années. Ces rencontres et les échanges qui s'y produisent stimulent, nourrissent, et ensemble construisent une communauté scientifique chaleureuse et prodigue. Je salue ainsi celles et ceux de la « communauté contraintes » que j'ai eu plaisir à rencontrer et à retrouver lors de ces réunions le plus souvent idylliques : Louis-Martin Rousseau, Carmen Gervet, Michela Milano, Michel Vasquez, Charlotte Truchet, Fabien Le Huédé, Filippo Foccaci, Mireille Ducassé, Olivier Ridoux, Michel Lemaître, Gérard Verfaillie, Abderrahmane Aggoun, Warwick Harvey, Steven Prestwich, Simon de Givry, Thomas Schiex, Gilles Pesant, Lucas Bordeaux, Martin Henz, Pascal Van Hentenryck, Benoît Rottembourg, Mark Wallace. La liste pourrait s'allonger longtemps encore ; que ceux que j'ai oubliés me pardonnent et soient remerciés également.

Certains l'appréhendent comme une tribu, mais c'est surtout mon milieu naturel, robuste et mouvant à la fois, fondamental et complexe, corpusculaire et ondulatoire. Je n'oublie rien de la chaleur que j'ai reçu, ni des déceptions forcément plus profondes mais si rares. Je remercie du fond du cœur ma famille et mes amis, ceux et celles qui ont partagés ces quelques années de ma vie. Ceux qui m'ont soutenu jusqu'au bout savent bien ce que je leur dois.

Ah oui, j'allais oublier. Merci Pascal.

Résumé

L'efficacité de la gestion des flux de trafic aérien (ATFM) est une nécessité du développement de l'aviation civile européenne au sein d'un réseau déjà saturé. Actuellement, certaines phases de l'ATFM qui correspondent à des problèmes d'optimisation combinatoire sont résolues empiriquement par des experts et leur modélisation standard est parfois mal adaptée aux contraintes opérationnelles. Nous utilisons la Programmation Par Contraintes pour modéliser de manière plus rigoureuse et réaliste certains de ces problèmes et calculer des solutions optimales. Le premier concerne la régulation par l'attribution de retards au décollage et dépend du second, la configuration des centres de contrôle. Le troisième est l'optimisation de l'utilisation de l'espace aérien dans l'hypothèse où les flux d'avions qui se croisent évoluent à des altitudes différentes. Pour modéliser et résoudre ces problèmes, nous avons défini et réalisé un outil d'optimisation adapté, FaCiLe, qui prend la forme d'une librairie pour le langage fonctionnel Objective Caml. Ses performances sont comparables aux meilleurs produits commerciaux tout en offrant une grande souplesse d'utilisation grâce à de puissantes abstractions. Nous illustrons la flexibilité de l'utilisation de FaCiLe et rapportons les détails de son implémentation.

Abstract

The efficiency of Air Traffic Flow Management (ATFM) is a key issue of the development of the European civil aviation within an already overloaded network. Currently, some phases of ATFM corresponding to combinatorial optimization problems are empirically solved by experts and their standard modelings are often not well suited to the operational constraints. We use Constraint Programming to model in a more rigorous and realistic way some of these problems and to compute optimal solutions. The first relates to the regulation by attribution of delays at take-off and depends on the second one, the configuration of the control centers. The third one is the optimization of the airspace use, assuming that crossing flows of aircrafts are given different altitudes. To model and solve these problems, we have defined and developed a convenient optimization tool, FaCiLe , which takes the form of a library for the functional language Objective Caml. Its performances are comparable with the best commercial products while being very versatile thanks to powerful abstractions. We illustrate the flexibility of its use and report the details of its implementation.

Table des matières

Table des figures

Liste des tableaux

Introduction

Contexte

Gestion des flux de trafic aérien Malgré un ralentissement conjoncturel de la croissance du trafic aérien en Europe, les prévisions de son évolution et l'observation des performances actuelles du contrôle indiquent que l'ensemble du système chargé d'assurer la régulation du trafic est proche de la saturation. Au niveau européen, la gestion de cette régulation est centralisée par Eurocontrol et se divise en plusieurs phases que leurs dynamiques hiérarchisent :

1. Stratégique/Structurelle : des négociations entre diverses instances (aviation civile, militaire, compagnies aériennes) et des plans d'orientation à long terme définissent les grandes lignes de l'évolution de l'espace aérien français et européen.
2. Pré-tactique : c'est l'organisation d'une journée de trafic un jour ou deux à l'avance. Lors de cette phase, la majeure partie des plans de vols sont connus ainsi que l'effectif des contrôleurs et les capacités des différentes configurations possibles des centres de contrôle. Les Flow Management Positions (FMP) sont alors chargées de planifier l'ouverture des secteurs de leur centre respectif pour l'adapter au trafic. La Central Flow Management Unit centralise ensuite toutes ces informations pour identifier les surcharges de trafic et les réguler (retards et re-routage).
3. Tactique : les allocations de créneaux de la CFMU sont mises à jour quelques heures avant le vol en tenant compte des modifications non connues lors de la phase précédente.
4. Temps réel : les contrôleurs sont chargés d'organiser l'écoulement du trafic au sein de leur secteur et de la coordination avec les secteurs voisins. Ils peuvent donner des ordres de manœuvres aux pilotes pour faire varier leur trajectoire et leur vitesse afin de respecter les normes de séparation entre aéronefs.

Le Laboratoire d'Optimisation Globale (LOG) du Centre d'Études de la Navigation Aérienne (CENA) mène divers projets de recherche destinés à analyser et améliorer l'efficacité de la gestion du contrôle. Nous nous intéresserons ici aux trois premières étapes qui constituent l'Air Traffic Flow Management (ATFM) et auxquelles correspondent des problèmes de grande taille qui doivent être résolus dans des intervalles de temps de l'ordre de quelques minutes à quelques heures — contrairement à la dernière phase de contrôle beaucoup plus dynamique (quelques secondes). Lors de chacune de ces phases, les experts

ont un certain degré de liberté pour choisir une solution qui permette de satisfaire au mieux la demande de trafic tout en assurant la sécurité des utilisateurs. De nombreuses procédures et contraintes limitent fortement les différents choix possibles et confèrent aux problèmes à résoudre un aspect très combinatoire. Or les outils dont disposent les opérateurs humains, en dehors de leur propre expérience et des données des années ou des semaines précédentes, se limitent souvent à la visualisation de la qualité d'une solution et à la reproduction des techniques manuelles devenues trop fastidieuses. Ce procédé empirique ne permet évidemment pas d'obtenir de solutions optimales à de tels problèmes dont certains ont été identifiés comme *NP-difficiles*[2]. Néanmoins, les diverses technologies qui concourent à la régulation du trafic ont le mérite d'être bien intégrées au sein d'un système très sûr et opérationnel.

Cependant, l'efficacité de la régulation doit être améliorée pour limiter les retards fréquents des appareils et s'adapter à la croissance du trafic. Les problèmes de l'ATFM tels que l'allocation de créneaux de décollage ou les schémas d'ouverture des centres de contrôle peuvent être modélisés comme des problèmes d'optimisation combinatoire (respectivement *scheduling* et *partitionnement*). Ce type de recherche opérationnelle est un domaine assez jeune et les techniques de résolution efficaces de ces problèmes commencent à peine à se populariser dans l'industrie.

Programmation Par Contraintes et Recherche Locale Les paradigmes les mieux adaptés à la résolution de ces problèmes sont la Programmation Par Contraintes et la Recherche Locale (prise dans son sens le plus large, c'est-à-dire en y incluant des méthodes telles que les algorithmes génétiques). La PPC effectue en général une recherche énumérative complète en inférant des réductions de l'espace de recherche à partir des contraintes du problème et permet donc de prouver l'optimalité d'une solution ; mais les temps de calcul peuvent devenir rédhibitoires quand la taille du problème est trop grande. Au contraire, la RL parcourt l'espace de recherche en se dirigeant vers les régions « prometteuses » grâce à une heuristique mais en n'exploitant les contraintes que passivement — pénalité dans le coût pour les solutions qui violent les contraintes — et sans pouvoir garantir l'optimalité du résultat ; cependant, la RL permet d'obtenir des solutions pour les problèmes de grandes tailles ou très difficiles.

Mais les différences entre ces deux techniques ne s'arrêtent pas à la méthode de recherche de solution. La PPC constitue en fait un langage de modélisation déclaratif avec une sémantique bien fondée qui permet d'écrire des prototypes rapidement et de les modifier très facilement. Comme les problèmes de l'ATFM posent souvent des difficultés importantes d'interprétation et d'adéquation entre les spécifications et l'utilisation opérationnelle des résultats (définition et pertinence des capacités, différences extrêmes entre schémas d'ouverture déposés et réalisés...), la PPC est particulièrement appropriée lors du travail de conception et de raffinement d'un modèle. De plus, la PPC permet de concevoir simple-

[2]Un problème est dit *NP-difficile* s'il n'existe pas d'algorithme *efficace* pour le résoudre. Un algorithme est *efficace* si son temps de calcul est proportionnel à un polynôme (de degré borné) en fonction de la taille du problème.

ment des stratégies de recherche grâce à un langage spécifique, ce qui facilite également l'expérimentation et la modélisation de procédures complexes (par exemple pour simuler les décisions d'un opérateur humain).

La RL ne jouit pas du même cadre formel et de l'élégance de la PPC. Elle rassemble une collection de techniques heuristiques fondées sur différentes métaphores (par exemple l'évolution darwinienne pour les algorithmes évolutionnistes ou un procédé métallurgique pour le recuit simulé). L'utilisation de la RL demande beaucoup d'expertise (définition des voisinages ou opérateurs, nombreux paramètres qui influencent la convergence, prise en compte des contraintes...) et les tentatives d'unification de ces différents algorithmes sont très récentes. Néanmoins, sur certains problèmes de grandes tailles ou peu structurés, les algorithmes de RL parviennent aux meilleurs résultats connus.

Pour résoudre des problèmes industriels de plus en plus complexes, la tendance actuelle est l'hybridation de la PPC, de techniques de Recherche Opérationnelle et de RL. La RO, qui fédère les paradigmes d'optimisation des problèmes industriels, recense des algorithmes efficaces et éprouvés dans de nombreux domaines, y compris l'optimisation combinatoire, mais limités par un cadre très strict (linéaire, quadratique, théorie des graphes...). La PPC utilise depuis plus d'une dizaine d'années des algorithmes de RO qui ont été adaptés (incrémentalité, gestion des *backtracks*...) pour être utilisés au sein de *contraintes globales* (c'est-à-dire des contraintes associées à des sous-problèmes spécifiques) afin d'assurer la consistance ou les propagations pour des sous-problèmes récurrents. La PPC apparaît ainsi comme une plate-forme d'intégration de ces techniques.

Quant à la RL, elle est souvent utilisée quand les autres méthodes échouent à cause de la trop grande taille ou du manque de structure d'un problème. Des travaux récents proposent diverses formes d'hybridation *ad hoc* entre RL et PPC (en général en sacrifiant la complétude de la recherche) sur des problèmes particuliers, mais l'intégration de ces deux paradigmes avec une sémantique claire comporte d'importantes difficultés conceptuelles et constitue un sujet de recherche actif.

Solveurs La PPC est un outil de premier choix pour modéliser des problèmes d'optimisation combinatoire comme ceux rencontrés dans l'ATFM, et un bon candidat pour l'intégration d'autres techniques de recherche lorsque le problème est trop difficile et que la recherche énumérative n'est pas assez efficace. Reste à choisir un solveur de contraintes souple et efficace doté de constructions de haut niveau dignes d'un langage de modélisation, et assez sûr pour détecter les erreurs (grossières) de l'expérimentateur qui préfère se concentrer sur la modélisation et la stratégie de recherche plutôt que sur la gestion de la mémoire.

La première contrainte sur le choix d'un solveur n'est pas technique (du moins en apparence) mais plutôt commerciale : sa disponibilité. Deux grandes classes de logiciels s'opposent, les logiciels *libres* ou *Open Source* dont le code source est accessible, et les logiciels *commerciaux* dont seuls les exécutables sont distribués. Les solveurs Open Source permettent à leurs utilisateurs d'en comprendre et corriger les erreurs ainsi que de modifier ses caractéristiques. Le logiciel profite ainsi des améliorations apportées par des utilisateurs

avertis très divers, ce qui tend à le rendre robuste et complet. Seuls un petit nombre de solveurs de contraintes sont Open Source.

Le paradigme de la PPC est né de l'extension des systèmes Prolog à la résolution d'équations arithmétiques : les contraintes sont des prédicats et la recherche non-déterministe de solution est inhérente à Prolog. La plupart des solveurs de contraintes sont donc naturellement fondés sur des compilateurs de programmes logiques. Pourtant, si cette élégante adéquation est praticable pour l'exhibition de petits exemples, elle peut être trop coûteuse dans un cadre industriel pour des problèmes de taille réelle : le traitement des données, le codage d'algorithmes de filtrage efficace pour des contraintes globales (d'ailleurs souvent écrits en C dans ses systèmes), l'affichage des résultats etc. sont des tâches délicates dans le cadre de la programmation logique. Le manque de typage et de vérifications à la compilation dans les systèmes Prolog classiques peuvent également dissuader de les utiliser dans le développement de gros projets pour des raisons de sécurité et de robustesse car les erreurs sont difficiles à détecter et à corriger.

Cependant, [Puget 92] a montré avec le système PECOS qu'un solveur de contraintes pouvait être réalisé avec un langage généraliste non-logique (Lisp dans le cas de PECOS). Ainsi, un solveur PPC peut profiter de la puissance, de l'efficacité et de la portabilité de compilateurs d'autres langages (fruits de beaucoup d'expertise et de plusieurs années de développement) sans avoir à réinventer la roue. Le solveur est alors une *librairie* pour un langage hôte.

Mais les projets de systèmes libres fondés sur des langages de programmation de haut niveau sûrs et efficaces ne sont que des tentatives très récentes Le champ est donc largement ouvert pour la conception et la réalisation d'une librairie de PPC Open Source écrite avec un langage applicatif.

Contributions

Nous présentons trois études sur des problèmes d'ATFM, sujets de recherches en cours ou récentes du CENA et d'Eurocontrol : l'allocation des créneaux de décollage [Barnier 00b, Barnier 00a, Barnier 01d], la conception des schémas d'ouverture des centres de contrôle et l'attribution des niveaux de vols d'un réseau de routes directes [Barnier 02a]. Les contributions originales des deux premières études concernent essentiellement les différentes modélisations de ces problèmes en PPC destinées à rendre les solutions plus réalistes d'un point de vue opérationnel. La troisième utilise un algorithme hybride, qui n'a jamais été publié à notre connaissance, pour accélérer la recherche de solutions. Les résultats obtenus correspondent à des données réelles de journées de trafic aérien.

L'implémentation des divers modèles a systématiquement été réalisée avec FaCiLe (Functional Constraint Library) [Barnier 01a, Barnier 01b, Barnier 01c], notre librairie Open Source de Programmation Par Contraintes sur les domaines finis, entièrement écrite avec Objective Caml [Leroy 00]. FaCiLe est le premier solveur de contraintes écrit en ML et elle bénéficie de l'expressivité et de la sûreté de son langage hôte ainsi que de l'efficacité du compilateur OCaml. Ces qualités nous ont permis de concentrer nos efforts sur la

modélisation des problèmes étudiés plutôt que sur des détails de codage ou des corrections d'erreurs. D'autre part, l'ambition de FaCiLe est de faciliter l'élaboration d'algorithmes hybrides. Une première étape a été réalisée en ce sens avec l'intégration d'*invariants* « à la Localizer » qui offrent un support pour la recherche locale. Nous présentons les fonctionnalités de FaCiLe et les caractéristiques de son implémentation.

Modèles et algorithmes pour l'ATFM

Allocation de créneaux Les secteurs de contrôle ont une capacité maximale exprimée en débit d'avions. Lorsque le trafic est trop important, cette capacité risque d'être dépassée et la sécurité du contrôle compromise. Pour anticiper et éviter des surcharges, la CFMU affecte des retards aux avions incriminés à l'aide d'un algorithme glouton n'ayant qu'une vue locale de chaque secteur.

Nous présentons plusieurs modèles du problème d'allocation de créneaux qui permettent de comparer les mérites de différentes interprétations de la capacité des secteurs de contrôle. Nous proposons notamment un modèle « continu » original utilisant des contraintes de tri. Alors que les formulations classiques de ce problème tendent à générer des pics de trafic périodiques, ce modèle permet de lisser le débit des flux d'avions entrant dans les secteurs de contrôle et de respecter les contraintes de capacité sur tout intervalle de temps d'une durée donnée. Nous tentons ainsi de rendre la charge de travail des contrôleurs plus régulière.

Notons également que la stratégie d'allocation implantée dans la plate-forme SHAMAN développée au CENA avec ILOG Solver a été reproduite avec FaCiLe [Rivière 01a] et le comportement des différents modèles vis-à-vis des incertitudes des horaires des plans de vol a été étudié dans [Rivière 01b].

Schémas d'ouverture Le schéma d'ouverture d'un centre de contrôle est le planning, élaboré manuellement par le FMP, des positions ouvertes pendant une journée de trafic. Les configurations successives du centre doivent permettre de contrôler le trafic en évitant de surcharger les secteurs et être compatible avec le nombre de contrôleurs disponibles.

Nous transformons le modèle de [Gianazza 02], implémenté avec un algorithme de Branch & Bound *ad hoc*, qui optimise l'utilisation des positions de contrôle ouvertes en un modèle en PPC. Mais les solutions obtenues présentent des changements de configurations qui sont trop complexes et fréquentes. Nous proposons un raffinement original de ce modèle pour rendre plus réalistes les transitions entre configurations successives.

Les performances de l'implémentation de ces modèles avec FaCiLe sont très supérieures (un ordre de magnitude pour les instances les plus grandes) à celles de l'algorithme *ad hoc* utilisé par [Gianazza 02].

Réseau de routes directes Pour étudier la faisabilité d'un réseau de routes rectilignes dans l'espace aérien français, tous les vols de même origine et destination sont regroupés en flux et la séparation de deux flux qui s'intersectent est assurée en leur affectant des niveaux de vol différents.

Le nombre minimal de niveaux de vol d'un réseau de routes directes est calculé en modélisant en PPC les flux d'avions et leur séparation comme un problème de coloriage de graphe. Pour améliorer l'efficacité du coloriage, les symétries entre les couleurs sont supprimées dynamiquement et une technique originale est présentée : des cliques sont découvertes au préalable par un algorithme glouton et sont utilisées pour poser des contraintes globales (qui détectent des inconsistances plus précocement) et guider la recherche.

L'implémentation en FaCiLe de cet algorithme permet d'obtenir une solution optimale pour tous les flux de plus de deux avions. Elle donne également de bons résultats sur les instances réelles de *benchmarks* standards (de coloriage de graphe), ce qui montre la robustesse de la technique.

FaCiLe

Malgré la popularité croissante de la PPC dans le monde académique aussi bien que dans celui de l'industrie, l'implémentation des solveurs de contraintes reste un domaine qui recense un faible nombre de publications et de rapports techniques — bien moins nombreux que ceux consacrés à l'implémentation de langage de programmation logique par exemple. De surcroît, très peu de solveurs permettent à leurs utilisateurs d'accéder au code source et de l'utiliser gratuitement, y compris à des fins commerciales (Open Source). Des efforts sont entrepris pour remédier à cette situation comme le projet CHOCO [Laburthe 00] mené en parallèle avec notre travail et qui en partage les intentions, c'est-à-dire publier une librairie qui soit *petite*, *simple*, *extensible*, *efficace* et *gratuite*.

Nous présentons les fonctionnalités, la conception et les techniques d'implémentation de FaCiLe, une librairie Open Source de Programmation Par Contraintes sur les domaines finis, entièrement écrite avec Objective Caml [Leroy 00]. La distribution standard de FaCiLe comprend un manuel d'utilisation et une documentation exhaustive de ses fonctionnalités [Barnier 01c].

Une librairie pour Objective Caml Lors de la conception de FaCiLe, nous avons tout d'abord délibérément écarté l'approche « encore un autre (plus ou moins) nouveau langage de plus » : notre propos n'est pas de réinventer la roue et devoir refaire le travail et les compromis des concepteurs de langages de programmation, mais bien de réaliser un système de PPC simple et expressif. Le choix d'une *librairie* s'est donc naturellement imposé pour satisfaire nos préoccupations d'efficacité, portabilité et disponibilité.

Pour le choix du langage hôte de notre librairie, nous avons exclu d'utiliser un langage procédural de bas niveau tel que C++ sans gestion automatique de la mémoire, ni typage sûr ou réelle capacité d'abstractions. D'autre part, les langages de programmation logique, candidats naturels pour un système de PPC, souffrent des handicaps évoqués dans la section précédente et les rend fastidieux à utiliser dans le cadre de projets industriels.

Bien que la programmation fonctionnelle n'offre pas la sémantique logique adéquate pour spécifier des contraintes ou des buts de recherche, les langages de la famille de ML évitent les écueils des langages de bas niveau en proposant une sémantique forte, la manipulation de termes d'ordre supérieur et de clôtures, la gestion de mémoire automatique.

Ils échappent également aux difficultés partagées par les systèmes Prolog standards car ils offrent des modules génériques (*foncteurs*), un typage fort et polymorphe, des structures de données riches etc. En outre, l'implémentation Objective Caml nous paraît être celle qui offre le meilleur compromis entre fonctionnalités et efficacité. Elle est d'ailleurs utilisée avec beaucoup de bonheur au LOG depuis 1995.

Fonctionnalités de FaCiLe FaCiLe offre la plupart des contraintes usuelles disponibles dans les systèmes de PPC classiques pour les domaines finis et les ensembles finis : contraintes arithmétiques linéaires (somme globale) et non-linéaires, contraintes globales (« tous différents », cardinalité généralisée, tri avec des algorithmes efficaces et paramétrables), contraintes sur des tableaux de variables (indexation, minimum), contraintes ensemblistes, réification. FaCiLe définit également un langage de buts clair et expressif, destiné à la recherche de solutions et à l'optimisation (Branch & Bound, Limited Discrepancy Search). Notre librairie permet aussi de construire des contraintes et des buts définis par l'utilisateur à l'aide d'une interface simple et souple qui fait largement appel au niveau d'abstraction élevé permis par le langage (fonctions d'ordre supérieur). Par ailleurs, des *références invariantes* « à la Localizer » [Michel 97] automatisent la mise à jour de structures de données d'un algorithme de RL ou peuvent servir à maintenir des critères de choix durant la recherche de solutions.

PPC fonctionnelle typée FaCiLe étant écrite pour ML, cette librairie jouit des puissantes abstractions inhérentes aux fonctions d'ordre supérieur et de la sûreté du typage polymorphe strict. Cette souplesse est complétée par les structures de données très riches et les constructions du langage adaptées au traitement symbolique (*pattern-matching*, ordre supérieur, foncteurs etc.) — l'un des domaines de prédilection de la programmation fonctionnelle.

Les itérateurs (fonctionnels), l'application partielle et les clôtures confèrent à OCaml certaines caractéristiques des langages de modélisation mathématique comme les opérateurs d'agrégats ou les quantificateurs universels. Le langage de buts définit par FaCiLe profite de ces capacités d'abstraction pour offrir une expressivité proche des buts de la programmation logique, avec une sémantique claire et rigoureuse ; ce n'est en général pas le cas des solveurs PPC qui ne sont pas fondés sur un système Prolog.

D'autre part, le typage inféré et vérifié à la compilation élimine la majeure partie des erreurs typiques commises lors du prototypage d'un modèle — sans parler des erreurs de gestion de la mémoire qui sont une plaie pour les langages sans récupérateur automatique, mais sans objet dans le cas de ML. FaCiLe se sert ainsi du typage pour contraindre l'utilisateur à écrire des expressions arithmétiques (sur les variables logiques) correctes et sans ambiguïté (pas de coercition implicite ni de surcharge).

Plan

Ce document tente de rendre plus réaliste la modélisation des problèmes d'ATFM grâce à l'expressivité de FaCiLe, notre librairie de PPC fonctionnelle. La première partie constitue un état de l'art des techniques de résolution de problèmes d'optimisation combinatoire sur les domaines finis. Les fonctionnalités et l'implémentation de FaCiLe sont présentées dans la deuxième partie. La troisième et dernière partie propose des modèles et des techniques originales pour résoudre des problèmes d'ATFM.

La première partie rappelle tout d'abord au chapitre 2 le formalisme des problèmes de satisfaction de contraintes (CSP) sur des domaines finis et sa puissance de modélisation des problèmes d'optimisation combinatoire. Cette présentation est biaisée pour mettre en évidence les aspects de la modélisation liés à la résolution des CSP par des programmes en contraintes (« langage de contraintes »). Puis l'état de l'art des deux grands paradigmes de recherche de solutions, la recherche énumérative utilisée généralement avec la PPC et la Recherche Locale, est exposé au chapitre 3, ainsi que la tendance actuelle à l'hybridation des différents type d'algorithmes. Enfin, les systèmes existants destinés à l'optimisation combinatoire sont passés en revue en insistant sur l'abstraction qu'ils apportent lors de la modélisation.

La seconde partie traite de FaCiLe en exposant d'abord l'éventail de ses fonctionnalités au chapitre 4 puis les principales caractéristiques de son implémentation au chapitre 5. L'expressivité du langage de buts de recherche y est illustrée ainsi que l'utilisation des références invariantes. L'architecture générale de FaCiLe est ensuite exposée, puis l'exécution des buts de recherche, la gestion des contraintes, la « normalisation » des contraintes arithmétiques, l'utilisation des mécanismes existants pour l'intégration des invariantes etc. sont détaillées.

La troisième partie est consacrée aux applications issues de l'ATFM : l'allocation des créneaux de décollage (chapitre 6), l'optimisation des schémas d'ouverture des centres de contrôle (chapitre 7) et l'attribution des niveaux de vols d'un réseau de routes directes (chapitre 8). La multiplicité des modélisations facilitée par notre librairie y est mise en exergue ainsi que les contributions apportées par les nouveaux modèles et par l'utilisation de FaCiLe en lieu et place d'autres solveurs de contraintes ou d'algorithmes *ad hoc*.

Enfin, nous présentons dans la conclusion les perspectives de développement de FaCiLe et de raffinement des modèles pour les problèmes d'ATFM étudiés. Nous y mentionnons également quelques autres applications de FaCiLe de moindre importance.

Première partie

Optimisation combinatoire

Chapitre 1

Problématique

1.1 Motivations

Les paramètres variables qui déterminent une instance particulière d'un procédé industriel, par exemple le planning d'un chantier de construction ou encore les fréquences attribuées aux émetteurs d'un réseau hertzien sans générer d'interférence, sont (étaient) généralement déterminés de manière empirique par des experts. La Recherche Opérationnelle (RO), discipline récente dont la naissance se situe dans les années 50, a pour but de formaliser et d'étudier ces problèmes ainsi que les techniques adaptées à leur résolution, en particulier à l'aide de calculateurs. En effet, quand la taille du problème est importante (nombre de variables, taille des ensembles de valeurs possibles pour ces variables, nombre d'équations ou « relations » entre les variables), un opérateur humain devient trop lent pour calculer une solution exacte au problème.

Par exemple dans le domaine du trafic aérien, les vols doivent être retardés si l'heure initiale de leur départ entraîne une surcharge dans un des secteurs de contrôle en route qu'ils traversent. La procédure européenne actuelle [CFMU 00], implémentée à la Central Flow Management Unit (CFMU) à Bruxelles, commence avec l'analyse par un expert du trafic prévu, à l'aide d'un simulateur mettant en évidence les secteurs en surcharge ; cet expert décide ensuite de placer certaines « régulations » (i.e. la limitation du débit d'un flux d'avions) en simulant leurs effets ; un autre algorithme (glouton en l'occurrence) calculera finalement les heures de décollage allouées, mais sans garantie de respecter les contraintes imposées, et en induisant éventuellement des surcharges dans d'autres secteurs. En se limitant à l'espace aérien français, ce problème doit typiquement prendre en compte plusieurs milliers de vols et des centaines de contraintes de capacité. Ce processus de résolution conduit à des solutions sous-optimales et ne permet pas non plus de respecter les contraintes de charge des contrôleurs. Or ces retards ont un coût élevé pour les compagnies aériennes (plus de 5 millions d'euros pour 1999 [IP 99]) et sont préjudiciables pour les passagers, si bien que le CENA et Eurocontrol mènent des études pour pouvoir obtenir de meilleures solutions (par exemple en respectant strictement les contraintes et en minimisant le retard maximal). Ce problème est étudié au chapitre 6.

1.2 Solution(s) d'un problème d'optimisation

L'exemple précédent propose de minimiser les retards attribués à des vols, c'est-à-dire de trouver une valeur numérique pour chaque variable du problème (une heure de décollage par vol) telle que le maximum de ces valeurs soit la plus petite possible parmi toutes les solutions qui respectent les contraintes de capacité. Cependant, certains problèmes requièrent d'autres types de résultats :

- trouver *une* solution qui satisfait les contraintes ;
- trouver *toutes* les solutions qui satisfont les contraintes ;
- parmi toutes les solutions qui satisfont les contraintes, trouver l'ensemble de celles qui minimisent ou maximisent un critère ;
- prouver que le problème n'a pas de solution ;
- trouver la solution qui minimise le nombre de contraintes non satisfaites ;
- etc.

Suivant le type d'application, le problème peut donc être formulé de manière assez diverse. Dans la suite, nous considérerons essentiellement des problèmes analogues au problème d'allocation de créneaux, c'est-à-dire trouver *une* solution qui respecte les contraintes du problème *et* qui minimise un certain critère ou prouver l'absence de telle solution le cas échéant. En pratique, des solutions sous-optimales sont souvent obtenues avant de trouver l'optimum, et si le temps de calcul de ce dernier est trop important, on peut arrêter la recherche et se contenter de *bonnes* solutions approchées.

1.3 Des problèmes intrinsèquement difficiles

Parmi les problèmes d'optimisation, nous ne considérerons pas la fraction qui s'exprime avec des variables « continues » et des équations simples (e.g. linéaires) telles que des techniques d'analyse mathématique classique (calcul de dérivées, résolution de système d'équations) permettent de les résoudre. De nombreux problèmes issus de l'industrie ne répondent pas à ces critères et nécessitent des variables discrètes ainsi que des relations non-linéaires complexes.

Ainsi, une partie importante de ces problèmes appartient à la classe des problèmes *NP-difficiles* [Garey 79], c'est-à-dire qu'il n'existe pas d'algorithmes efficaces[1] pour les résoudre ; on peut noter qu'une contrainte d'intégralité portant sur une variable d'un problème continu linéaire peut suffire à le rendre NP-difficile. Plus généralement, les premiers problèmes intrinsèquement identifiés comme tels sont issus de la logique propositionnelle, de l'étude des graphes et de partitionnement d'ensembles, c'est-à-dire des problèmes discrets et peu structurés, en d'autres termes des problèmes *combinatoires* dont on ne connaît pas de propriétés assez fortes pour guider efficacement la construction d'une solution. Pour caractériser de manière plus intuitive ces problèmes difficiles, on peut dire que les meilleures méthodes connues ne peuvent se dispenser de « chercher » parmi toutes les combinaisons

[1] Un algorithme est considéré efficace si son temps de calcul s'exprime avec un polynôme de degré borné en fonction de la taille du problème.

possibles, c'est-à-dire l'ensemble des tuples de valeurs que peuvent prendre les variables. Or cet espace de recherche croît exponentiellement en fonction du nombre de variables. Ainsi, pour un problème à 20 variables pouvant prendre chacune 10 valeurs, et en utilisant un calculateur vérifiant en 10^{-6} s l'admissibilité d'une combinaison, il ne faut pas moins de 3 millions d'années pour parcourir exhaustivement l'espace de recherche.

De tels problèmes sont omniprésents dans le monde industriel, si bien qu'il serait vain d'en dresser une liste exhaustive. Parmi les exemples les plus courants, citons néanmoins les problèmes de planning, emploi du temps, ordonnancement, affectation de ressources, configuration, flots dans un réseau, tournées de véhicules, optimisation de micro-code, cryptographie, séquençage d'ADN, conformation de molécules, et bien sûr de nombreux problèmes liés au trafic aérien comme l'affectation de créneaux, les schémas d'ouverture de secteurs de contrôle ou la conception d'un réseau de routes. Beaucoup d'exemples sont également issus de disciplines plus académiques comme la théorie des graphes : coloration, clique maximale, couverture etc., la partition d'ensembles, la satisfiabilité de formules logiques (SAT), les « puzzles » logiques (zèbre), l'arithmétique cryptée (SEND+MORE=MONEY)...

1.4 Améliorer l'efficacité des algorithmes

L'approche la plus naïve pour résoudre un problème combinatoire est de générer tous les éléments de l'espace de recherche et d'en tester l'admissibilité, i.e. vérifier que l'élément est bien une solution au problème ou la solution de meilleure coût sur l'ensemble des solutions admissibles. Comme l'illustre l'exemple de la section précédente, cette méthode n'est plus envisageable, car trop coûteuse en temps de calcul, dès que l'on s'attaque à des problèmes de taille « réelle ». Pour remédier à cette inefficacité liée à une recherche « aveugle » de solutions, de nombreuses techniques ont été développées dans le cadre de la Recherche Opérationnelle, puis plus tardivement dans celui de (ce que l'on appelait) l'Intelligence Artificielle.

Ces techniques tentent d'exploiter la *structure* particulière d'un problème pour réduire la fraction de l'espace de recherche explorée et se diriger vers des régions *a priori* prometteuses de cet espace. Nous nous intéresserons à deux grandes classes de méthodes :

1. la *recherche énumérative* qui fait appel à des algorithmes d'exploration arborescente systématique de l'espace de recherche couplé à des techniques de *consistance*[2] chargées d'inférer *a priori* des réductions de cet espace en exploitant les contraintes du problème ;
2. la *recherche locale* qui parcourt un « chemin » dans l'espace de recherche, sans l'explorer systématiquement, en modifiant à chaque étape un élément initial de cet espace

[2] *Consistance* est un anglicisme, dérivé de *consistency*, que nous utiliserons car il est bien plus employé dans le jargon technique que son équivalent français *cohérence*. Notons néanmoins à notre décharge l'une des définitions du dictionnaire TLF (`zeus.inalf.fr`) :

> *Emplois spéc. a) LOG. Consistance d'un système d'axiomes. Caractère de ce système lorsque ses termes ne sont pas contradictoires. Synon. cohérence, non-contradiction.*

pour tenter de l'améliorer.

Les caractéristiques de ces deux techniques sont très différentes, à commencer par leur *complétude* : un algorithme de recherche est *complet* (ou *exact*) s'il garantit de trouver la solution du problème en un temps fini, c'est-à-dire, suivant la formulation la plus courante pour les problèmes d'optimisation combinatoire (cf. section 1.2), de trouver une solution optimale ou de prouver qu'il n'existe pas de solution. La recherche énumérative est une méthode complète alors que la recherche locale est *incomplète* et ne peut fournir à elle seule aucune preuve d'optimalité ou d'absence de solution.

Les techniques de parcours d'arbre de recherche et de consistance, en exploitant la structure du problème *a priori* et en supprimant des régions de l'espace de recherche qui ne contiennent pas de solution, ont une bien meilleure complexité en moyenne que l'approche naïve « générer et tester ». De plus, les progrès considérables des matériels en termes de puissance de calcul et de mémoire disponible permettent d'envisager l'utilisation de ces techniques sur des problèmes de plus en plus ambitieux.

Cependant, de telles méthodes de recherche complètes gardent une complexité exponentielle en pire cas, c'est pourquoi tout un éventail d'algorithmes *heuristiques* incomplets, que nous amalgamons allègrement sous l'appellation générique « recherche locale », ont été développés depuis le début des années 70 pour fournir de meilleures solutions que celles obtenues avec des algorithmes complets sur des problèmes de grandes tailles, peu structurés ou particulièrement difficiles. Ils privilégient l'efficacité en temps de calcul en renonçant à pouvoir fournir des garanties qualitatives sur les solutions obtenues. Ces techniques sont également appelées *méta-heuristiques* car elles guident la recherche de solution en s'inspirant de métaphores génériques (qui ne sont pas dédiées à un problème particulier) : biologique pour les algorithmes évolutionnistes, métallurgique pour le recuit simulé...

1.5 Abstraire les techniques de résolution

Historiquement, l'approche la plus courante pour résoudre un problème d'optimisation combinatoire (à l'aide d'un calculateur) était d'écrire un programme *ad hoc* dans un langage procédural. Cette méthode demande un effort et une expertise très importants pour le développement du programme, les algorithmes de la Recherche Opérationnelle devenant de plus en plus complexes et nécessitant de maintenir des structures de données sophistiquées. En outre, de tels programmes dédiés à la résolution de problèmes particuliers seront difficiles à maintenir, à modifier ou à réutiliser, bien qu'ils exploitent des techniques similaires.

Pour y remédier beaucoup de progrès ont été faits depuis une vingtaine d'années, tant dans l'élaboration de cadres théoriques qui permettent de fédérer les multitudes de techniques particulières à chaque grande classe d'algorithmes et d'améliorer leur compréhension, que dans la conception d'outils informatiques (langages, librairies, interfaces...) destinés à réduire les temps de développement et à proposer des couches d'abstraction pour favoriser l'expérimentation de techniques complexes.

Si les librairies de solveurs et langages de modélisation les plus populaires ont été développés essentiellement pour la programmation linéaire continue et en nombres entiers

(ou mixte) fondée sur l'algorithme du *Simplex* [Dantzig 49] (XMP [Marsten 81], CPLEX de Robert Bixby (1988), AMPL [Fourer 93]), le paradigme le plus élégant et polyvalent est sans doute la Programmation par Contraintes (PPC), issue de la généralisation théorique *et* technique des systèmes de programmation logique d'une part et de la formalisation des Problèmes de Satisfaction de Contraintes (voir chapitre 2) d'autre part. Ce cadre a permis d'intégrer les techniques antérieures spécifiques à l'optimisation combinatoire non-linéaire (recherche énumérative et consistance), mais aussi celles des autres domaines de l'optimisation (en réels, rationnels...), tout en offrant des langages de programmation très proches d'un langage de modélisation/spécification. De nombreux systèmes de programmation logique permettent de résoudre des programmes en contraintes, voire leur sont dédiés : Prolog-II/III/IV [Colmerauer 82], CHIP [Dincbas 88], ECL^{i}PS^e [ECL^{i}PS^e 01], SICStus [Carlsson 88]... et depuis une dizaine d'années, des solveurs ont été implémentés en tant que librairies pour des langages hôtes non-logiques (procéduraux/objets et fonctionnels), notamment pour des raisons d'efficacité et de génie logiciel : PECOS [Puget 92] en Lisp, ILOG Solver [Solver 99] en C++, JCL [Torrens 97] en Java...

En revanche, les travaux menés en ce sens pour la recherche locale sont très récents et disparates, alors que son efficacité est manifeste sur certaines classes de problèmes d'optimisation combinatoire, notamment des problèmes de grande taille et peu structurés (comme certains problèmes de *benchmark* aléatoires par exemple, cf. chapitre 8). Quelques articles présentent des formalismes pour unifier les différents types de méta-heuristiques [Taillard 98, Yagiura 99] tels que la recherche taboue, les algorithmes génétiques ou encore le recuit simulé, mais sans véritable consensus ni notion forte de sémantique (contrairement à la PPC [Jaffar 98]). D'autre part, des langages et librairies de recherche locale ont été développés récemment tels que Localizer [Michel 97] conçu par Pascal Van Hentenryck et Laurent Michel, un langage de modélisation précurseur dans ce domaine, ou encore Local++ [Schaerf 00], HotSpot [Fink 98] et SCOOP [SCOOP 98], des librairies de classes génériques, avec l'ambition de réduire le temps de développement des applications et de permettre des expérimentations plus nombreuses et audacieuses[3] sans sacrifier la rapidité d'exécution du code généré.

1.6 Vers l'hybridation

Certains problèmes particulièrement difficiles et/ou de grande taille (*Large Scale Combinatorial Optimization* [Gervet 01]) résistent encore et à la PPC, et à la RL utilisées seules. En effet, les outils « standards » de la PPC telle que la recherche arborescente systématique sont trop coûteux en temps et en mémoire quand le nombre de variables et la taille des domaines augmentent, et quand la structure du problème ne permet pas de guider efficacement la stratégie de recherche. Les techniques de RL, quant à elles, n'exploitent généralement qu'*a posteriori* les contraintes d'un problème, par exemple en pénalisant les

[3]Peut-être encore plus que pour la PPC, l'efficacité des méta-heuristiques est sensible à la conception et au choix des opérateurs ainsi qu'au paramétrage, et nécessite donc de nombreuses expérimentations et beaucoup d'expertise.

éléments non-admissibles, et peuvent ainsi stagner dans des régions de l'espace de recherche qui violent les contraintes.

Une des approches les plus fructueuses pour résoudre de tels problèmes est l'hybridation des deux paradigmes pour en combiner les avantages. Parmi les formes de couplage apparues dans la littérature depuis une dizaine d'années, on peut recenser :

- Le retour arrière flexible : la RL est utilisée pour parcourir l'arbre de recherche d'un programme en contraintes de manière moins systématique et « aveugle » que le parcours en profondeur d'abord standard [Milano 02].
- Les « voisinages » (cf. section 3.2.1) d'un algorithme de RL peuvent être explorés avec un programme en contraintes pour éviter de générer des éléments non-admissibles [Pesant 96] et parcourir efficacement de « grands » voisinages.
- La RL peut être utilisée pour optimiser après coup (*post-optimisation*) une solution non-optimale d'un programme en contraintes.
- Des sous-arbres sans solution ou sous-optimaux d'une recherche arborescente peuvent être élagués par une RL [Sellmann 02, Focacci 02].
- Etc.

Une synthèse détaillée de ces divers types d'hybridation est présentée section 3.3.3.

Cette tendance semble prendre de plus en plus d'ampleur au sein des communautés scientifiques concernées par l'optimisation combinatoire, comme en témoigne le succès croissant rencontré par le séminaire CP-AI-OR[4] [CPAIOR 99] créé en 1999. En outre, d'excellents résultats obtenus sur des benchmarks classiques à l'aide d'algorithmes hybrides sont rapportés de plus en plus fréquemment [Bent 01, Sellmann 02].

Cependant, de tels programmes restent difficiles à écrire et peu réutilisables, car les outils d'optimisation actuels sont fortement dédiés à l'une ou l'autre de ces techniques. De plus, nous verrons section 3.3.1 que PPC et RL sont fondées sur des notions parfois antagonistes, comme la manipulation de solutions partielles consistantes pour la PPC alors que la RL fait évoluer des éléments entièrement déterminés (toutes les variables sont instanciées) mais pas forcément admissibles. La conception de programmes hybrides est donc confrontée à des problèmes de modélisation et de « génie logiciel » pour pouvoir faire cohabiter les deux paradigmes.

Toutefois, quelques travaux de recherches très récents tentent d'élaborer un cadre théorique commun et des systèmes assez polyvalents pour concevoir et écrire de manière élégante ce type d'algorithme hybride. L'un des efforts les plus remarquables en ce sens est sans doute le langage SaLSA (Specification Language for Search Algorithms) de François Laburthe et Yves Caseau [Laburthe 98a] qui utilisent la notion de « point de choix » pour unifier le *branchement* d'une recherche arborescente typique de la PPC et le *déplacement* d'un élément de l'espace de recherche à un autre dans un algorithme de RL ; ces points de choix sont des objets du langage qui peuvent être combinés à l'aide d'opérateurs pour spécifier un algorithme de recherche.

De manière plus anecdotique, la librairie commerciale ILOG Dispatcher destinée à ré-

[4] International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems

soudre des problèmes de tournées de véhicules intègrent à un système de PPC des techniques de RL.

Chapitre 2

Problèmes de Satisfaction de Contraintes

Les Problèmes de Satisfaction de Contraintes (CSP) sont un formalisme générique destiné à modéliser les problèmes typiques de la Recherche Opérationnelle, notamment ceux dont la nature est combinatoire tels que les problèmes d'ordonnancement, d'allocation, de configuration... Les CSP sont caractérisés par la difficulté de trouver une solution admissible, c'est-à-dire qui respecte toutes les spécifications qui le structurent, ou par la difficulté d'obtenir une solution optimale pour un critère donné.

Ces problèmes sont décrits en termes de *variables* (« inconnues ») et de *contraintes* (relations entre les variables) ; leur résolution consiste à affecter une valeur à chacune des variables de telle manière que les contraintes soient respectées. Nous définissons d'abord dans la section 2.1 les principales notions utiles pour comprendre les algorithmes de résolution présentées au chapitre 3. Quelques aspects de la phase de modélisation sont ensuite examinés section 2.2, puis nous présentons des extensions du cadre classique dans la section 2.3.

2.1 Définitions

Le cadre des CSP, contrairement à ceux de la programmation mathématique (linéaire, mixte, quadratique...), est très générique et concerne des domaines quelconques — nous nous intéresserons plus particulièrement aux domaines *finis*, i.e. discrets — et des relations quelconques entre les variables. Nous définissons et discutons les concepts principaux de la spécification d'un CSP et des propriétés des affectations partielles de valeurs aux variables, utiles aux méthodes constructives de résolution (cf. section 3.1).

2.1.1 Contraintes

Une contrainte est une relation entre des inconnues ou variables d'un problème qui restreint les combinaisons de valeurs qu'elles peuvent prendre simultanément. Par exemple,

une contrainte géométrique de non-recouvrement entre deux rectangles spécifie simplement que les positions et dimensions de chacun sont reliés par les équations suivantes :

Exemple 2.1 (Non-recouvrement de rectangles)

$$\begin{cases} x_1 + l_1 \leq x_2 \quad ou \quad x_2 + l_2 \leq x_1 \\ y_1 + h_1 \leq y_2 \quad ou \quad y_2 + h_2 \leq y_1 \end{cases}$$

(x_1, y_1) et (x_2, y_2) étant les variables qui représentent les coordonnées du sommet inférieur gauche des deux rectangles, l_1 et l_2 leur largeur, et h_1 et h_2 leur hauteur.

Ces contraintes, comme les équations d'un problème classique en programmation mathématique, sont :

- *déclaratives* : les contraintes ne décrivent pas de procédure chargée de les faire respecter ;
- *non-directionnelles* : contrairement aux équations fonctionnelles d'un tableur (standard) qui distinguent les variables d'entrées et la variable unique de sortie[1], les contraintes ne supposent rien sur la nature de leurs variables ; une contrainte reste bien définie quelque soit l'ordre des modifications apportées à ses variables ;
- *additives* : toutes les contraintes d'un CSP doivent être satisfaites simultanément, i.e. la propriété à vérifier est implicitement la conjonction de l'ensemble des contraintes posées.

Ainsi, la résolution d'un CSP consistera à affecter des valeurs à toutes les variables du problème (qu'on appellera *variables de décision*) telles que l'ensemble des contraintes du problème soient respectées.

Expressivité des contraintes

La puissance de modélisation des CSP est liée à la généralité de la notion de contraintes qui a été exploitée dans la plupart des solveurs disponibles pour fournir un langage très riche. En effet, l'utilisateur de ces systèmes dispose non seulement d'équations (inéquations, diséquations) linéaires ou non, mais aussi de contraintes sophistiquées fournies sous forme de primitives du système. Ces contraintes permettent par exemple de spécifier le nombre d'occurrences d'une valeur dans un ensemble de variables, d'indexer un tableau de variables par un indice variable etc. On appelle ces dernières des contraintes *symboliques*. Ces deux types de contraintes, arithmétiques et symboliques, dotés de sémantique particulière sont exprimées en *intention*.

Outre les contraintes en intention, une contrainte sur des variables discrètes peut être exprimée en donnant la liste exhaustive des tuples de valeurs qu'elle autorise ou interdit. La relation exprimée peut donc être quelconque, et cette forme de contraintes, appelée

[1] Par exemple, l'équation fonctionnelle $z = 2x + y$ sous-entend que des informations sur z ne peuvent être déduites qu'après l'obtention d'informations sur x et y ; réciproquement, une information sur z ne permet pas d'en déduire sur x ni y. Ce type d'équations *directionnelles* est parfois appelé *one-way constraints*.

contraintes en extension (cf. section 2.1.5), subsume la précédente pour des variables discrètes. La définition des CSP discrets (section 2.1.4) est d'ailleurs fondée sur ce type de contraintes.

D'autre part, la plupart des solveurs offrent des connecteurs logiques ($\vee, \wedge, \neg, \Rightarrow ...$) pour combiner les contraintes et en construire de nouvelles, ce qui permet d'exprimer des relations non-linéaires, par exemple de modéliser simplement les disjonctions de tâches typiques des problèmes de *scheduling* ou de l'exemple 2.1 sur un problème géométrique.

Ainsi, de nombreux problèmes de la RO peuvent être modélisés naturellement à l'aide d'un programme en contraintes très concis : peu d'efforts sont nécessaires pour transformer la structure d'un problème exprimé en langage « naturel » vers un ensemble de contraintes et de variables. Cette adéquation confère à la PPC un statut de « boîte à outils » généraliste pour les problèmes d'optimisation combinatoire ; la variété et le nombre des domaines d'application évoqués dans la section 1.3 en témoignent.

Contraintes binaires et n-aires

La plupart des résultats théoriques sur les CSP et des algorithmes qui les résolvent portent sur des contraintes *binaires*, c'est-à-dire que la relation exprimée par la contrainte n'implique que deux variables. Un tel CSP peut être représenté par un graphe dont les nœuds sont les variables est les arcs les contraintes : des algorithmes et des propriétés topologiques sur ce graphe peuvent alors être utilisés pour l'étudier et le résoudre. Par exemple, le *degré* d'une variable correspond au degré de son nœud dans le graphe, c'est-à-dire le nombre de contraintes qui concernent cette variable, et ce critère est parfois utilisé dans des heuristiques de recherche (pour résoudre d'abord les parties les plus contraintes).

Cependant, certains problèmes, notamment issus d'applications réelles, s'expriment naturellement avec des contraintes d'arité plus importante. Avec l'application de la PPC à des problèmes industriels, une partie des résultats et des algorithmes sur les CSP binaires ont été généralisés pour prendre en compte ces contraintes plus générales. Les solveurs actuels permettent ainsi d'utiliser des contraintes sur un nombre fixé de variables supérieur à deux ainsi que sur un nombre quelconque de variables.

Les CSP discrets d'arité quelconque peuvent néanmoins être systématiquement transformés en CSP binaires et les techniques dédiées à ces derniers peuvent donc être utilisées [Rossi 90, Bacchus 98]. Mais les transformations impliquées ne sont définies que sur les contraintes en extension : elles les traduisent en variables dont les valeurs possibles correspondent aux tuples autorisés par la contrainte[2]. Si la contrainte est exprimée en intention, le coût de la transformation peut être prohibitif car l'ensemble des valeurs autorisées (ou interdites) peut avoir une taille exponentielle avec l'arité de la contrainte.

[2]Des contraintes auxiliaires doivent assurer que les variables du problème original qui sont partagées entre plusieurs contraintes aient bien la même valeur.

2.1.2 Domaines

Jusqu'à présent, les quelques exemples de contraintes cités ne précisaient pas explicitement dans quel type de domaines les variables concernées pouvaient prendre leur valeur. En effet, l'élégance du concept des CSP permet d'appliquer (en partie) le même formalisme sur une grande variété de « structures mathématiques » dotées d'une théorie équationnelle décidable : les arbres finis (termes de Prolog), les booléens, les entiers, les rationnels, les réels (ou plutôt les intervalles de nombres flottants), les ensembles, les listes finies, voire même les λ-termes [Ridoux 95]... La complexité de la résolution d'un problème dépend de son domaine : quasi-linéaire pour les termes de l'univers de HERBRAND (Prolog), polynomiale pour les équations linéaires sur les réels ou encore exponentielle sur les entiers... Des problèmes mixtes peuvent également être exprimés en CSP mais seules les contraintes sur des variables à domaine fini peuvent être formulées en extension.

Cependant, les avancées les plus importantes ont été réalisés dans le cadre de l'optimisation combinatoire qui est devenu le domaine de prédilection de la PPC. C'est pourquoi nous nous intéresserons essentiellement aux CSP sur des domaines finis, en général des sous-ensembles des entiers de cardinal fini, et parfois des ensembles (finis) d'ensembles finis. Nous préciserons donc pour chaque variable son *domaine* initial (p.ex. $x \in [1,4] \cup [6,10]$ pour une variable entière ou $s \in [\emptyset, \{1,2,3\}]$ pour une variable d'ensemble), et les modèles utilisés peuvent éventuellement contenir des variables de types différents intervenant dans des contraintes communes (e.g. contrainte de cardinalité entre une variable d'ensemble et une variable entière, cf. section 4.3.2).

Par ailleurs, notons que l'on peut convenir d'attribuer à toutes les variables le même domaine — par exemple le plus grand domaine représentable dans un système de PPC donné ou l'union des domaines de toutes les variables — et de représenter les restrictions initiales sur les valeurs que peuvent prendre les variables par des contraintes *unaires*, i.e. l'appartenance de la variable à un ensemble de valeurs ou simplement des inéquations ($5 \leq x \wedge x \leq 10$).

2.1.3 Le problème de satisfiabilité

Le problème de *satisfiabilité* [3] concerne les formules logiques du *calcul propositionnel*, c'est-à-dire des termes construits à l'aide de *variables* et des connecteurs logiques $\neg$, $\vee$ et $\wedge$ (éventuellement enrichis des raccourcis $\rightarrow$ pour l'implication $p \rightarrow q \equiv \neg p \vee q$, et $\leftrightarrow$ pour l'équivalence $p \leftrightarrow q \equiv (\neg p \wedge \neg q) \vee (p \wedge q)$). Une *interprétation* $\mathcal{I}$ pour une telle formule est une substitution de l'ensemble de ses variables, noté $\mathcal{V}$, par des booléens, c'est-à-dire une fonction $\mathcal{I} : \mathcal{V} \mapsto \{vrai, faux\}$. On dira alors que la formule est vraie si son calcul sur l'arithmétique booléenne donne un résultat égal à $vrai$, et qu'elle est fausse s'il est égal à $faux$. Notons que ce calcul s'effectue avec une complexité linéaire en fonction de la taille de la formule.

[3] Le terme *satisfiabilité* est largement répandu dans la littérature (française, spécialisée) bien qu'il soit dérivé de l'anglais *satisfiability*. L'équivalent politiquement correct fondé sur *satisfaire*, mais moins usité, est *satisfaisabilité*.

Définition 2.1 (Formule propositionnelle satisfiable) *Une formule propositionnelle F est* satisfiable *si et seulement si il existe une interprétation $\mathcal{I}$ pour laquelle elle est* vraie. *On notera alors $\mathcal{I} \models F$.*

Le problème de satisfiabilité consiste à déterminer si une formule propositionnelle est satisfiable. C'est le premier problème a avoir été démontré NP-complet — donc de complexité exponentielle — par COOK [Cook 71] en transformant une machine de TURING non-déterministe quelconque en une formule propositionnelle (voir [Garey 79], ouvrage de référence sur la théorie des problèmes NP-complets). Plus précisément, le problème original, appelé SAT, est plus structuré et pose la question de la satisfiabilité d'une formule propositionnelle en *forme normale conjonctive* (*CNF* en anglais), c'est-à-dire sous la forme d'une conjonction de *clauses*, chaque clause étant une disjonction de *littéraux*, i.e. de variables éventuellement niées : $(p \vee q) \wedge (p \vee \neg q) \wedge \neg p$ est une formule CNF sur l'ensemble de variables $\{p, q\}$ et elle n'est pas satisfiable.

Problème 2.1 (SAT)
Instance : *Soient $\mathcal{V}$ un ensemble de variables et $\mathcal{C}$ un ensemble de clauses sur $\mathcal{V}$.*
Question : *Existe-t-il une interprétation qui satisfait $\mathcal{C}$?*

L'ensemble de clauses $\mathcal{C}$ s'interprète comme la conjonction de ses éléments $\bigwedge_{c \in \mathcal{C}} c$; il faut donc construire une valuation de $\mathcal{V}$ telle que pour chaque clause de $\mathcal{C}$, au moins l'un de ses littéraux soit égal à *vrai*. Si une interprétation $\mathcal{I}$ rend vraie la clause $c \in \mathcal{C}$, on notera également $\mathcal{I} \models c$.

Le problème SAT peut être modélisé comme un CSP : $\mathcal{V}$ est l'ensemble des variables qui ont toutes $\{vrai, faux\}$ pour domaine, et $\mathcal{C}$ est l'ensemble des contraintes, chaque clause c étant vue comme une équation dans l'arithmétique booléenne $c = vrai$. Pour obtenir une définition conforme à celle des CSP sur les domaines finis (2.2), on pourra facilement transformer cette expression en intention des contraintes (construites avec des opérateurs sur les booléens) vers une forme en extension, en calculant la « table de vérité » de chaque clause et en ne retenant que les lignes dont le résultat est *vrai*.

La résolution des problèmes SAT est un domaine de recherche très actif — deux sessions lui sont encore consacrées cette année lors de la principale conférence de PPC, CP'2002.

2.1.4 CSP sur les domaines finis

Nous reprenons ici la définition usuelle des CSP sur les domaines finis telle qu'on peut la trouver dans [Van Hentenryck 95, Schiex 92] :

Définition 2.2 (CSP sur les domaines finis) *Un* problème de satisfaction de contraintes (CSP) sur des domaines finis *est défini par un triplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$:*

- *$\mathcal{X} = \{x_1, \ldots, x_n\}$ est l'ensemble des* variables.
- *$\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ est l'ensemble des* domaines finis, *i.e. $\forall i \in [1, n]$ $\mathcal{D}_i$ est un ensemble de cardinal fini $d_i = |\mathcal{D}_i|$. Le domaine $\mathcal{D}_i$ de la variable x_i est l'ensemble des valeurs qu'elle peut prendre.*

- $\mathcal{C} = \{c_1, \ldots, c_m\}$ *est l'ensemble des* contraintes. *Une contrainte* $c_i = (\mathcal{X}_i, \mathcal{R}_i)$ *est définie par l'ensemble des variables* $\mathcal{X}_i \subseteq \mathcal{X}$ *sur lesquelles elle porte et par la relation* $\mathcal{R}_i$ *qui est définie en* extension *par l'ensemble des valeurs que peuvent prendre simultanément les variables de* $\mathcal{X}_i$ *:*
  - $\mathcal{X}_i = \{x_{i_1}, \ldots, x_{i_{k_i}}\}$. *Le cardinal* $k_i = |\mathcal{X}_i|$ *est appelé* arité *de la contrainte.*
  - $\mathcal{R}_i \subset \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_{k_i}}$ *: un sous-ensemble du produit cartésien des domaines de chaque variable de* $\mathcal{X}_i$.

Comme les $\mathcal{D}_i$ sont de cardinal fini, $\forall c_i \in \mathcal{C}$ le produit cartésien $\mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_{k_i}}$ est fini et $\mathcal{R}_i$ est donc également de cardinal fini.

Par exemple, la formule logique $(p \vee q) \wedge (p \vee \neg q) \wedge \neg p$ peut être modélisée par le CSP suivant :

Exemple 2.2 (Modélisation d'une formule logique en CSP)

$$\left\{ \begin{array}{lcl} \mathcal{X} & = & \{p, q\} \\ \mathcal{D} & = & \{\{vrai, faux\}, \{vrai, faux\}\} \\ \mathcal{C} & = & \left\{ \begin{array}{l} (\{p,q\}, \{(faux, vrai), (vrai, faux), (vrai, vrai)\}), \\ (\{p,q\}, \{(faux, faux), (vrai, vrai), (vrai, faux)\}), \\ (\{p\}, \{(faux)\}) \end{array} \right\} \end{array} \right.$$

Chaque clause de la formule correspond à une contrainte de $\mathcal{C}$; les deux premières sont binaires (d'arité deux) et la dernière unaire.

Pour estimer l'effort nécessaire pour résoudre un CSP particulier, on peut utiliser les indicateurs numériques suivants :

- La *dimension* : l'entier n égal au nombre de variables.
- La *taille de l'espace de recherche* : c'est le cardinal du produit cartésien de tous les domaines $\prod_{i \in [1,n]} d_i$, que l'on majorera souvent par $d_{\max}^n$, avec $d_{\max} = \max_{i \in [1,n]} d_i$ la taille du plus grand des domaines.
- La *densité* des contraintes : $\frac{|\mathcal{C}|}{n(n-1)/2}$. Elle n'est classiquement définie que sur les CSP à contraintes binaires dont tous les $\mathcal{X}_i$ sont distincts, donc pour lesquels le nombre maximal de contraintes est $\frac{n(n-1)}{2}$.
- La *dureté* d'une contrainte : $1 - \frac{|\mathcal{R}_i|}{|\mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_{k_i}}|}$. Elle indique la proportion de l'espace de recherche interdit par la contrainte quand on le projette sur les variables de $\mathcal{X}_i$.

Ces indicateurs sont surtout utilisés dans le cadre des CSP binaires aléatoires pour analyser les performances d'algorithmes de résolution [Grant 96].

L'exemple 2.2 est ainsi de dimension 2, avec un espace de recherche de taille 4, et la dureté de ses trois contraintes sont respectivement 1/4, 1/4 et 1/2. Sa densité est en revanche mal définie car les deux premières contraintes portent sur le même ensemble de variables et la troisième est unaire. Cependant, on peut toujours supprimer une contrainte unaire en remplaçant le domaine $\mathcal{D}_i$ de la variable concernée par $\mathcal{R}_i$ et en veillant à retirer des contraintes les tuples utilisant des valeurs caduques. On peut également « fusionner » des contraintes binaires portant sur les mêmes variables x_i et x_j en calculant l'intersection de $\mathcal{R}_i$ et $\mathcal{R}_j$. On obtient alors le nouveau CSP équivalent :

Exemple 2.3 (CSP binaire équivalent à l'exemple 2.2)

$$\begin{cases} \mathcal{X} &= \{p, q\} \\ \mathcal{D}' &= \{\{faux\}, \{vrai, faux\}\} \\ \mathcal{C}' &= \{(\{p, q\}, \emptyset)\} \end{cases}$$

Ce CSP n'a trivialement pas de solution.

Réciproquement, on peut considérer les domaines des variables comme des contraintes unaires et prendre un même domaine plus large (qui contient au moins l'union des domaines) pour toutes les variables. Soit un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, on obtient alors le CSP $\mathcal{P}' = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$ équivalent :

$$\begin{cases} \mathcal{X}' &= \mathcal{X} \\ \mathcal{D}' &\supseteq \bigcup_{i \in [1,n]} \mathcal{D}_i \\ \mathcal{C}' &= C \cup \{(\{x_i\}, \mathcal{D}_i),\ \forall i \in [1, n]\} \end{cases}$$

La définition 2.2 n'induit donc pas une représentation unique d'un CSP. L'équivalence des CSP est abordée section 2.2.1.

2.1.5 Contraintes en extension et contraintes en intention

La définition 2.2 utilise une expression en *extension* des contraintes du problème, c'est-à-dire la liste exhaustive des combinaisons de valeurs autorisées. De nombreux algorithmes d'approximation de *consistance* (AC-$\{[1, 7], 2000\}$, cf. section 3.1.4), ont été mis au point et perfectionnés pour exploiter efficacement ce type de représentation dans le cas des contraintes binaires[4] (d'arité deux). Elle est bien adaptée à l'étude des CSP aléatoires sur les domaines finis qui sont générés avec des degrés de difficulté (taille, densité, dureté) fixés, justement obtenus en construisant des ensembles de valeurs autorisées de cardinal adéquat. De même, certaines applications industrielles comme les problèmes de configuration font naturellement appel à des contraintes exprimées en extension.

Pourtant une grande partie des contraintes utilisées pour modéliser des problèmes applicatifs « structurés » sont exprimées à l'aide de :

- formules analytiques telles que des équations (inéquations, diséquations) arithmétiques dans lesquelles les variables sont combinées à l'aide d'opérateurs arithmétiques : e.g. la diséquation $x^3 + y^3 \neq 3kxy$ empêche le point du plan (x, y) d'être placé sur le folium de DESCARTES ;
- équations avec des opérateurs sur des ensembles : e.g. $|s_1| + |s_2| = |s| \ \wedge \ s_1 \cap s_2 = \emptyset$ assure que (s_1, s_2) est une partition de s ;
- opérateurs logiques, réifications : e.g. les disjonctions de l'exemple 2.1 ;
- indexation, minimum de séquences de variables (contraintes « symboliques ») ;

[4] On peut toutefois transformer une contrainte d'arité plus grande en contraintes binaires. Voir section 2.1.1.

– plus généralement, spécifications de haut niveau proches du langage naturel : e.g. « toutes ces variables sont différentes », « ces deux séquences de variables sont des permutations l'une de l'autre ».

On dira que de telles contraintes sont exprimées en *intention* : contrairement à la définition générale dont sont dotées les contraintes en extension, une sémantique particulière leur est associée. Elles correspondent à des sous-problèmes structurés récurrents dans les CSP. On dispose en général d'algorithmes *ad hoc* plus efficaces pour les traiter que ceux destinés aux contraintes en extension qui n'ont pas de structure spécifique — c'est la forme la plus générale d'une contrainte.

Toutefois, dans le cadre des domaines finis, il est toujours possible de transformer une contrainte en intention en un ensemble fini de combinaisons autorisées : pour une contrainte c_i, il suffit de construire l'ensemble des substitutions pour lesquelles la contrainte est satisfaite en énumérant les tuples de $\mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_{k_i}}$. Mais la taille de cet ensemble croît exponentiellement avec l'arité de la contrainte : si la taille maximale des domaines est d et l'arité k, il faut en pire cas un ensemble de taille d^k pour décrire la contrainte, ce qui peut se révéler impraticable (efficacité en temps et en espace) pour la conception d'une contrainte au sein d'un solveur. Si la dureté de la contrainte est très forte, on peut néanmoins envisager cette transformation ; de même si la dureté est très faible et en utilisant la définition « complémentaire » des contraintes où ce sont les tuples *interdits* qui sont spécifiés.

On pourra noter les contraintes en intention en omettant l'ensemble de variables sur lesquelles elles portent car elles apparaissent en général comme un prédicat fonction de leurs variables.

Langage des contraintes Ces contraintes en intention sont en général disponibles en tant que primitives dans les solveurs PPC ainsi que des connecteurs logiques pour les combiner. Le langage constitué de ces termes composés de variables, contraintes primitives et connecteurs logiques offre une expressivité inégalée parmi les diverses technologies d'optimisation combinatoire, ce qui permet de modéliser rapidement des problèmes complexes et de natures très diverses. Les problèmes de trafic aérien étudiés dans la partie III sont ainsi décrits en tant que CSP à l'aide de ces contraintes de haut niveau.

Fonction de satisfaction Pour éviter de préciser les types de contraintes utilisés pour la définition d'un CSP, on associe à chaque contrainte une *fonction de satisfaction* qui indique si la contrainte est vérifiée pour une valuation des variables impliquées (ce qui correspond à la notion de « contrainte définie par un prédicat » de [Bessière 97]). Pour une contrainte en extension, cette fonction teste simplement l'appartenance de la valuation à $\mathcal{R}_i$:

Définition 2.3 (Fonction de satisfaction) *Soit un CSP sur des domaines finis* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$*, à chaque contrainte* $c_i \in \mathcal{C}$ *est associée une* fonction booléenne de satisfaction $\widetilde{c_i} : \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_{k_i}} \mapsto \{vrai, faux\}$ *définie par :*

$$\widetilde{c_i}(v_{i_1}, \ldots, v_{i_{k_i}}) = \begin{cases} vrai & si\ (v_{i_1}, \ldots, v_{i_{k_i}}) \in \mathcal{R}_i \\ faux & sinon \end{cases}$$

Pour les autre types de contraintes, on supposera que l'on dispose d'une telle fonction sans se préoccuper de la manière dont elle est calculée. On substituera parfois cette fonction à l'ensemble des tuples valides dans la définition d'une contrainte :

$$c_i = (\mathcal{X}_i, \widetilde{c}_i)$$

2.1.6 Instanciation et satisfaction de contraintes

Comme pour le problème SAT (cf. 2.1.3) où la sémantique d'une formule propositionnelle s'obtient en substituant une valeur booléenne à chaque variable logique, puis en remplaçant chaque connecteur par sa fonction associée dans l'arithmétique booléenne, un CSP s'interprète à l'aide d'une valuation de toutes ses variables, appelée *instanciation totale*, puis en remplaçant les contraintes c par leur fonction de satisfaction associée $\widetilde{c}$.

Cependant, chaque contrainte c est définie sur un sous-ensemble des variables $\mathcal{X}_c \subseteq \mathcal{X}$ et on pourra donc appliquer la fonction $\widetilde{c}$ à une *instanciation partielle* des variables si elle spécifie une valeur pour chaque variable de c. Les algorithmes de recherche *constructive* manipulent de telles instanciations partielles en cherchant à les étendre (cf. section 3.1).

Définition 2.4 (Instanciation partielle) *Soient un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, une* instanciation partielle *$\mathcal{I}$ est définie par un couple $\mathcal{I} = (\mathcal{X}_\mathcal{I}, \mathcal{V})$:*

- *$\mathcal{X}_\mathcal{I} = \{x_{\mathcal{I}_1}, \ldots, x_{\mathcal{I}_k}\} \subseteq \mathcal{X}$ est le sous-ensemble des variables sur lequel porte l'instanciation ;*
- *$\mathcal{V} = (v_{\mathcal{I}_1}, \ldots, v_{\mathcal{I}_k}) \in \mathcal{D}_{\mathcal{I}_1} \times \cdots \times \mathcal{D}_{\mathcal{I}_k}$ est un tuple de valeurs prises dans les domaines des variables à instancier.*

On utilisera la notation fonctionnelle $\mathcal{I}(x_i) = v_i$ pour désigner la valeur de substitution d'une variable par l'instanciation $\mathcal{I}$. Et on notera abusivement $\widetilde{c}_i(\mathcal{I})$ l'application de la fonction de satisfaction d'une contrainte c à la projection d'une instanciation $\mathcal{I}$ sur ses variables si $\mathcal{X}_i \subseteq \mathcal{X}_\mathcal{I}$. On pourra également définir une instanciation partielle comme un ensemble de couples $\mathcal{I} = \{(x_{\mathcal{I}_1}, v_{\mathcal{I}_1}), \ldots, (x_{\mathcal{I}_k}, v_{\mathcal{I}_k})\}$ (et non pas un couple d'ensembles de même cardinal) là où cette notation simplifie la compréhension.

Une solution potentielle d'un CSP est une valuation de toutes les variables du problème appelé *instanciation totale* :

Définition 2.5 (Instanciation totale) *Soit un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, une* instanciation totale *$\mathcal{I} = (\mathcal{X}_\mathcal{I}, \mathcal{V})$ est une instanciation partielle pour laquelle $\mathcal{X}_\mathcal{I} = \mathcal{X}$.*

Résoudre un CSP, c'est construire une instanciation qui respecte chaque contrainte du problème. Localement, ces instanciations doivent donc *satisfaire* chaque contrainte du problème pour qu'une solution puisse être obtenue :

Définition 2.6 (Satisfaction de contrainte) *Soit un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ et une contrainte $c_i = (\mathcal{X}_i, \widetilde{c}_i) \in \mathcal{C}$. On dit qu'une instanciation partielle $\mathcal{I} = (\mathcal{X}_\mathcal{I}, \mathcal{V})$ telle que $\mathcal{X}_i \subseteq \mathcal{X}_\mathcal{I}$:*

- satisfait *la contrainte c_i si et seulement si $\widetilde{c_i}(\mathcal{I}) = vrai$, et on notera $\mathcal{I} \models c_i$ (comme pour un problème de satisfiabilité, cf. section 2.1.3)* ;
- viole *la contrainte c_i si et seulement si $\widetilde{c_i}(\mathcal{I}) = faux$.*

Mais pour qu'une instanciation partielle puisse faire partie de la solution d'un CSP, il faut que toutes les contraintes concernées soient satisfaites ; on dit alors que l'instanciation est *consistante*.

Définition 2.7 (Instanciation consistante) *Soit un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, une instanciation $\mathcal{I} = (\mathcal{X}_\mathcal{I}, \mathcal{V})$ est* consistante *si et seulement si :*

$$\forall c_i = (\mathcal{X}_i, \widetilde{c_i}) \in \mathcal{C} \text{ telle que } \mathcal{X}_i \subseteq \mathcal{X}_\mathcal{I},\ \mathcal{I} \models c_i$$

Une solution d'un CSP se caractérise alors simplement :

Définition 2.8 (Solution d'un CSP) *Soit un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, une* solution *$S$ de $\mathcal{P}$ est une instanciation totale et consistante. On notera alors $S \models \mathcal{P}$. On appellera $\mathcal{S}_\mathcal{P}$ l'ensemble des solutions de $\mathcal{P}$.*

On dit qu'un CSP est *consistant* s'il a au moins une solution. Vérifier qu'une solution est consistante est un problème facile (il suffit de vérifier la consistance de l'instanciation pour chaque contrainte), alors que déterminer si un CSP est consistant ou non est problème NP-complet.

Notons encore qu'on peut identifier un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ à *une* contrainte $c = (\mathcal{X}, \mathcal{R})$ équivalente à la conjonction de toutes ses contraintes et dont la relation est l'ensemble de ses solutions $\mathcal{R} = \mathcal{S}_\mathcal{P}$.

2.1.7 Degré de violation

Les algorithmes de Recherche Locale ou de satisfaction de contraintes *partielle* (PCSP, cf. section 2.3.1) manipulent des instanciations totales ou partielles non consistantes. Il est utile dans ce cadre de caractériser ces instanciations en calculant le *degré de violation*[5] du CSP, c'est-à-dire le nombre de contraintes violées par une instanciation :

$$\overline{\delta}(\mathcal{I}) = |\{c \in \mathcal{C} \text{ tel que } \mathcal{X}_c \subseteq \mathcal{X}_\mathcal{I} \wedge \widetilde{c}(\mathcal{I}) = faux\}|$$

ou son ratio avec le nombre total de contraintes.

Un CSP est résolu par une instanciation totale $\mathcal{I}$ pour laquelle $\overline{\delta}(\mathcal{I}) = 0$. Pour des instances sur-contraintes, i.e. qui n'ont pas de solution, une variante du CSP appelée *Max CSP* consiste à trouver une solution qui minimise $\overline{\delta}$. Ce problème est un cas particulier du cadre plus général des CSP valués abordés section 2.3.1.

[5] Inversement, on peut calculer la notion contraire de *degré de satisfaction*.

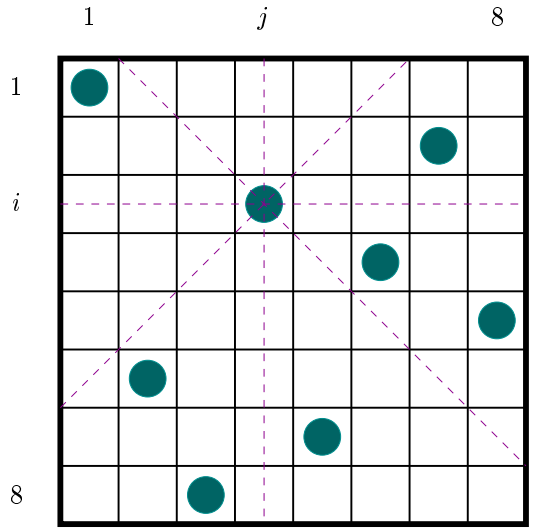


FIG. 2.1 – Une solution pour le problème des 8 reines.

2.2 Modélisation

Lors de la modélisation d'un problème en CSP, on doit choisir comment traduire des spécifications exprimées avec le langage naturel en un triplet de variables de décisions, dont l'instanciation totale caractérise une solution, de domaines et de contraintes. Des variables auxiliaires, qui n'interviennent pas dans l'expression des solutions, sont parfois nécessaires pour transformer la structure du problème en contraintes, notamment dans le cadre de la PPC où les programmes s'écrivent avec un nombre de contraintes primitives limité. Toujours avec la PPC, les solveurs fournissent parfois des variables et contraintes sur les booléens ou les ensembles, outre les entiers. Certains problèmes se modélisent plus naturellement ou « efficacement » (la recherche de solution est plus rapide) suivant le choix du domaine.

Parmi les exemples les plus classiques, le problème des huit (ou n) reines se prête à plusieurs modélisations possibles.

Problème 2.2 (Les n reines) *Soit un entier $n \geq 4$, placer n reines sur un échiquier de $n \times n$ cases sans qu'aucune reine n'en attaque une autre.*

Il faut donc modéliser avec des variables un échiquier muni de n reines et avec des contraintes le fait que deux reines ne peuvent se trouver sur la même ligne, colonne ou diagonale (de pente positive ou négative). La figure 2.1 montre une solution pour le problème à 8 reines.

Pour traduire ce problème en CSP, on peut par exemple :

1. Associer une variable booléenne (codée par une variable entière dans $[0, 1]$) à chaque

case de l'échiquier qui indique si une reine s'y trouve ou non :

$$\mathcal{P}_1 \begin{cases} \mathcal{X} = \{x_{i,j}, \forall (i,j) \in [1,n]^2\} \\ \mathcal{D} = \{\mathcal{D}_{i,j} = [0,1], \forall (i,j) \in [1,n]^2\} \\ \mathcal{C} = \begin{array}{ll} & \{\sum_{(i,j)\in[1,n]^2} x_{i,j} = n\} \quad \text{n reines} \\ \cup & \{\sum_{i=1}^{n} x_{i,j} \leq 1, \forall j \in [1,n]\} \quad \text{colonnes} \\ \cup & \{\sum_{j=1}^{n} x_{i,j} \leq 1, \forall i \in [1,n]\} \quad \text{lignes} \\ \cup & \{\sum_{(i,j)\in[1,n]^2, i-j=k} x_{i,j} \leq 1, \forall k \in [0,n-2]\} \quad \text{diagonales positives} \\ \cup & \{\sum_{(i,j)\in[1,n]^2, i+j=k} x_{i,j} \leq 1, \forall k \in [0,n-2]\} \quad \text{diagonales négatives} \end{array} \end{cases}$$

On utilise donc n^2 variables avec des domaines de taille 2, ce qui correspond à un espace de recherche de taille 2^{n^2}. On pose le nombre de contraintes suivant :

$$\underbrace{1}_{n \text{ reines}} + \underbrace{2n}_{\text{colonnes, lignes}} + \underbrace{2(2n-3)}_{\text{diagonales}} = 6n - 5$$

2. Associer deux variables entières à chaque reine qui représentent sa ligne et sa colonne :

$$\mathcal{P}_2 \begin{cases} \mathcal{X} = \{x_i, \forall i \in [1,n]\} \cup \{y_i, \forall i \in [1,n]\} \\ \mathcal{D} = \{\mathcal{D}_{x_i} = [1,n], \forall i \in [1,n]\} \cup \{\mathcal{D}_{y_i} = [1,n], \forall i \in [1,n]\} \\ \mathcal{C} = \begin{array}{ll} & \{x_i \neq x_j, \forall 1 \leq i < j \leq n\} \quad \text{colonnes} \\ \cup & \{y_i \neq y_j, \forall 1 \leq i < j \leq n\} \quad \text{lignes} \\ \cup & \{x_i - y_i \neq x_j - y_j, \forall 1 \leq i < j \leq n\} \quad \text{diagonales positives} \\ \cup & \{x_i + y_i \neq x_j + y_j, \forall 1 \leq i < j \leq n\} \quad \text{diagonales négatives} \end{array} \end{cases}$$

Avec ce modèle, on n'a pas besoin de contrainte pour indiquer que l'on doit placer n reines, c'est le nombre de variables utilisées qui rend compte de cette partie de la structure du problème. On dénombre $2n$ variables de domaine de taille n, ce qui correspond à un espace de recherche de taille n^{2n}. Chacun des quatre groupes de contraintes contient $\frac{n(n-1)}{2}$ contraintes, donc ce modèle nécessite $2n(n-1)$ contraintes.

Ce modèle présente l'inconvénient d'avoir de nombreuses symétries : les couples $(x_i, y_i), \forall i \in [1,n]\}$ correspondant à une reine sont tous équivalents, i.e. la solution (l'échiquier) est identique quelle que soit la permutation sur ces couples. On cherchera donc la solution dans un espace $n!$ fois trop grand. Pour supprimer ces symétries, il suffit par exemple d'ordonner les reines de haut en bas car il n'y a qu'une reine par ligne : $x_i < x_{i+1}, \forall i \in [1, n-1]$, ce qui rajoute $n-1$ contraintes.

Cette technique destinée à supprimer les symétries suggère de n'utiliser que les colonnes des reines comme variables car il ne peut y avoir qu'une reine par ligne, ce que fait le modèle suivant. Avec $\mathcal{P}_2$ enrichi par les contraintes de symétrie, les lignes sont fixées : $x_i = i, \forall i \in [1,n]$.

TAB. 2.1 – Mensurations de trois modèles pour le problème des 8 reines.

Modèle	$\lvert\mathcal{X}\rvert$	$\lvert\mathcal{D}_i\rvert$	$\lvert$espace$\rvert$	$\lvert\mathcal{C}\rvert$
$\mathcal{P}_1$	64	2	$1,8\ 10^{19}$	43
$\mathcal{P}_2$	16	8	$2,8\ 10^{14}$	112
$\mathcal{P}_3$	8	8	$1,7\ 10^{7}$	84

3. Associer une variable entière à chaque ligne de l'échiquier pour représenter la colonne de la reine qui s'y trouve :

$$\mathcal{P}_3 \begin{cases} \mathcal{X} = \{x_i, \forall i \in [1,n]\} \\ \mathcal{D} = \{\mathcal{D}_i = [1,n], \forall i \in [1,n]\} \\ \\ \mathcal{C} = \begin{array}{ll} & \{x_i \neq x_j, \forall 1 \leq i < j \leq n\} \quad \text{colonnes} \\ \cup & \{x_i - i \neq x_j - j, \forall 1 \leq i < j \leq n\} \quad \text{diagonales positives} \\ \cup & \{x_i + i \neq x_j + j, \forall 1 \leq i < j \leq n\} \quad \text{diagonales négatives} \end{array} \end{cases}$$

$\mathcal{P}_3$ n'utilise que n variables pour un espace de recherche de taille n^n et seulement $\frac{3n(n-1)}{2}$ contraintes car on modélise des propriétés du problème directement dans la structure du CSP.

On peut bien sûr concevoir de nombreuses autres modélisations.

Le tableau 2.1 résume les tailles de ces différents modèles avec 8 reines. Le dernier est celui qui semble le plus facile à résoudre car son espace de recherche est le plus petit. Suivant la technique de recherche utilisée, la nature des contraintes, qui sont traitées plus ou moins efficacement, peut influencer également le choix du modèle.

2.2.1 Équivalence de CSP

Pour obtenir $\mathcal{P}_3$, on s'est servi d'une propriété du problème : *il y a une reine et une seule par ligne*. On pourrait s'en servir pour rendre les contraintes du modèle de $\mathcal{P}_\infty$ plus « dures » en remplaçant les inéquations sur les lignes par des équations :

$$\sum_{j=1}^{n} x_{i,j} = 1, \forall i \in [1,n]$$

Cette information est une contrainte *impliquée* par le problème, elle ne modifie pas l'ensemble de ses solutions. On dit de telles contraintes qu'elles sont *redondantes* et que $\mathcal{P}_1$ et son raffinement, qui sont deux CSP portant sur les mêmes variables et ayant mêmes solutions, sont *équivalents* :

Définition 2.9 (CSP équivalents) *Soient deux CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ et $\mathcal{P}' = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$ définis sur le même ensemble de variables, on dit que $\mathcal{P}$ et $\mathcal{P}'$ sont* équivalents *si et seulement si $\mathcal{S}_\mathcal{P} = \mathcal{S}_{\mathcal{P}'}$. On note alors $\mathcal{P} \equiv \mathcal{P}'$.*

Sur le problème des reines, on peut remarquer également que chaque colonne contient exactement une reine et remplacer les inéquations concernées par des équations. Le problème obtenu est encore équivalent à $\mathcal{P}_1$.

De même, on pourra ôter une valeur au domaine d'une variable d'un CSP si celle-ci ne fait partie d'aucune solution[6]. Soit le CSP suivant :

$$\mathcal{P} \left\{ \begin{array}{rcl} \mathcal{X} & = & \{x, y\} \\ \mathcal{D} & = & \{[1,3],[1,3]\} \\ \mathcal{C} & = & \{x < y\} \end{array} \right.$$

on peut déduire de la contrainte $x < y$ que x est différent de 3 et y de 1. Le CSP $\mathcal{P}'$ obtenu en filtrant les domaines de x et y lui est donc équivalent :

$$\mathcal{P} \equiv \mathcal{P}' \left\{ \begin{array}{rcl} \mathcal{X} & = & \{x, y\} \\ \mathcal{D} & = & \{[1,2],[2,3]\} \\ \mathcal{C} & = & \{x < y\} \end{array} \right.$$

Toutefois, cette notion d'équivalence ne peut pas s'appliquer aux CSP $\mathcal{P}_1$, $\mathcal{P}_2$ et $\mathcal{P}_3$ car ils ne sont pas définis sur les mêmes variables. Or $\mathcal{P}_1$ et $\mathcal{P}_3$ par exemple aboutissent au même ensemble de solutions concrètes (i.e. de reines placées sur un échiquier). De même, si on avait modélisé ce problème avec une seule variable sur un domaine d'ensemble d'entier (contraint à être de cardinal n) en numérotant chaque case de 1 à n^2 — ce serait bien sûr une très mauvaise idée car les contraintes seraient des plus exotiques — la relation d'équivalence 2.9 ne pourrait pas s'appliquer, alors qu'on pourrait établir un isomorphisme entre les solutions de ce modèle ensembliste et $\mathcal{P}_3$.

Pour étendre l'équivalence de CSP en prenant en compte des différences de modélisation, y compris des ensembles de variables et des structures de domaine différents, [Rossi 90] définit une notion de *réductibilité* qui rappelle celle que l'on utilise sur les problèmes de décision en complexité des algorithmes [Garey 79]. Deux fonctions doivent être spécifiées pour transformer les variables et les valeurs d'un problème vers l'autre. Deux problèmes mutuellement réductibles sont dits *équivalents* et on peut donc chercher les solutions de l'un pour obtenir celles de l'autre.

2.2.2 Symétries

Les modèles de $\mathcal{P}_1$ et $\mathcal{P}_3$ n'ajoutent pas de symétrie au problème des n reines car chaque instanciation totale représente une unique solution. En revanche, les couples de variables (x_i, y_i) du modèle $\mathcal{P}_2$ sont interchangeables, si bien que chaque solution du problème original est représentée $n!$ fois. On a intérêt à réduire le plus possible l'espace de recherche en choisissant des modèles qui induisent le moins possible de symétries, par exemple avec des variables ensemblistes comme [Barnier 02b] pour le problème des « écolières de KIRKMAN ».

[6] Pour que les contraintes en extension restent bien définies, il faut également leur ôter tous les tuples où cette valeur apparaît.

S'il en reste, on peut essayer de les supprimer à l'aide de contraintes en « ordonnant » les variables comme mentionné précédemment.

Le problème peut aussi contenir « intrinsèquement » des symétries. Pour le problème des n reines, l'échiquier est symétrique par trois rotations (d'angles $\frac{\pi}{2}$, π et $\frac{3\pi}{2}$) et quatre réflexions (horizontale, verticale et les deux diagonales). Pour supprimer la réflexion verticale, on peut restreindre avec des contraintes supplémentaires la position de la reine située sur la première ligne à la première moitié des colonnes : $x_{1,j} = 0, \forall j \in [\lceil \frac{n}{2} \rceil + 1, n]$ pour $\mathcal{P}_1$ ou $x_1 \leq \lceil \frac{n}{2} \rceil$ pour $\mathcal{P}_3$. Avec le modèle de $\mathcal{P}_3$, on peut aussi supprimer facilement la réflexion horizontale en ajoutant la contrainte $x_1 < x_n$.

De nombreux travaux de recherche en PPC ont pour objet d'abstraire et de rendre efficace la suppression des symétries (une session est consacrée au sujet à la conférence CP'2002) pour limiter la portion de l'espace de recherche explorée et éventuellement obtenir toutes les solutions « uniques » (non-symétriques les unes des autres). [Gent 00] par exemple présente une technique générique qui produit dynamiquement de nouvelles contraintes pour supprimer des symétries identifiées par un isomorphisme entre deux solutions quelconques.

Les techniques de Recherche Locale peuvent également profiter de suppressions de symétries simples à exprimer comme la réflexion verticale pour les problèmes $\mathcal{P}_1$ et $\mathcal{P}_3$ (avec une réduction de domaine). Mais elles n'intègrent que difficilement la suppression de symétries en la modélisant par des contraintes car on cherche en général à minimiser le nombre de contraintes violées, donc on ne réduirait pas la taille de l'espace et on risquerait d'handicaper la recherche en perturbant l'évaluation des solutions. D'autre part, la RL étant incomplète, elle n'est pas utilisée pour obtenir l'ensemble des solutions d'un CSP, ce qui réduit l'enjeu des suppressions de symétries.

2.2.3 Réification

La plupart des solveurs de PPC permettent de modéliser des problèmes à l'aide de contraintes *réifiées*. La réification d'une contrainte c introduit une nouvelle variable booléenne (codée par un entier) contrainte à être égale à 1 si c est satisfaite et à 0 si elle est violée, mais c n'est pas ajoutée à l'ensemble des contraintes. On peut ensuite utiliser ces nouvelles variables pour poser d'autres contraintes (appelée parfois *méta-contraintes*). La possibilité de réifier une contrainte est en général limitée au cas des contraintes arithmétiques mais le mécanisme est général et cette extension du langage des contraintes d'un solveur renforce considérablement son pouvoir d'expression.

Par exemple, on peut écrire très facilement une contrainte de « cardinalité » qui compte le nombre de variables d'un ensemble $\{x_1, \ldots, x_k\}$ égales à une tierce variable v :

$$\text{card}(c, v, \{x_1, \ldots, x_k\}) \Leftrightarrow c = \sum_{i=1}^{k} b(x_i = v)$$

en appelant $b(c)$ la réification de la contrainte c.

Les réifications permettent également d'écrire simplement des contraintes de suppression de symétrie comme l'ordre lexicographique de couples de variables :

$$x_i < x_{i+1} \ \vee_b \ (x_i = x_{i+1} \ \wedge_b \ y_i < y_{i+1})$$

On utilise dans cette dernière expression des opérateurs booléens spéciaux $x \vee_b y$ et $x \wedge_b y$ pour désigner respectivement $b(x) + b(y) \geq 1$ et $b(x) + b(y) = 2$ (en gardant le codage avec des entiers) ou les contraintes booléennes *réifiables* équivalentes.

Enfin, on peut aussi utiliser des réifications pour modéliser des CSP avec contraintes optionnelles ou des PCSP, en essayant de maximiser le nombre de contraintes satisfaites. Mais les contraintes réifiées de la PPC sont souvent peu efficaces dans les solveurs classiques et le cadre standard des CSP n'est pas bien adapté pour modéliser cette classe de problème. Des extensions du formalisme des CSP ont été conçues pour pouvoir traiter ces problèmes (cf. section 2.3.1).

2.3 Extensions

Le cadre classique de la définition (2.2) des CSP décrit un réseau *statique* de contraintes *dures* (i.e. une solution du CSP *doit* les satisfaire) exprimées en *extension*. Dans la section précédente, la spécification des contraintes a été enrichie grâce aux contraintes en intention — qui rendent compte de l'utilisation des contraintes *primitives*, éventuellement *globales*, des solveurs PPC et des algorithmes spécifiques qui les traitent — et à la réification au sein d'un *langage* de contraintes. Mais certains problèmes réels nécessitent la modélisation de *préférences* entre diverses solutions ou la prise en compte de l'évolution d'un problème dans le temps. Nous évoquons dans les sections suivantes des extensions du CSP qui tentent de répondre à ces problèmes.

2.3.1 CSP valués

Un CSP classique impose qu'une solution respecte toutes les contraintes, et quand une instance est sur-contrainte, l'ensemble des solutions est vide. Dans le cadre de problèmes industriels, certaines contraintes doivent être respectées impérativement car elles sont intrinsèques à la structure du problème, mais on souhaite en général obtenir une solution « aussi bonne que possible », quitte à *relâcher* des contraintes de moindre importance, c'est-à-dire à accepter que certaines d'entre elles soient violées. Le nombre ou l'importance des contraintes violées influent alors sur la caractérisation de la solution optimale.

Avec un CSP orthodoxe, on peut choisir d'ignorer toutes les contraintes « optionnelles » mais le nombre de solutions risque d'être très grand, certaines d'entre elles se révélant sans intérêt pratique. Au contraire, imposer que toutes les contraintes soient dures peut aboutir à l'absence de solution. La PPC, en introduisant la réification de contraintes, peut prendre en compte de manière limitée des CSP enrichis de contraintes optionnelles (simples, e.g. arithmétiques) associées au coût (supposé entier) de leur relaxation r_i :

$$\mathcal{C}' = \{(c'_i, r_i), \forall i \in [1, m']\}$$

Le coût de la violation des contraintes s'exprimera par exemple additivement :

$$\sum_{i=1}^{m'} b(c'_i) r_i$$

et on cherchera à le minimiser. Si le problème est déjà un problème d'optimisation, il faudra choisir comment hiérarchiser l'objectif et les relaxations.

[Borning 87] présente les *hiérarchies de contraintes* et [Freuder 89] propose le formalisme des CSP *partiels* (PCSP) pour modéliser de tels problèmes ; ils adaptent également les algorithmes de la PPC pour les résoudre. [Schiex 95] présente le cadre élégant des CSP *valués* (VCSP) qui paramètre la relaxation des contraintes par une structure d'ensemble ordonné (les coûts des relaxation) muni d'une opération binaire[7] (l'addition dans le paragraphe précédent) et une fonction qui associe chaque contrainte à un élément de cet ensemble[8]. L'instanciation de ce formalisme par la structure adéquate permet d'obtenir le CSP standard et la plupart des extensions autorisant des contraintes optionnelles. Des algorithmes génériques fondés sur ceux de la PPC traditionnelle sont également décrits.

Les techniques de RL font évoluer en général des instanciations totales qui ne respectent pas toutes les contraintes et se servent du nombre de contraintes violées pour guider la recherche : dans le cadre classique des CSP, une solution est obtenue si ce nombre est égal à 0 (cf. section 2.1.7). Ce processus est bien adapté aux problèmes pour lesquelles on veut pouvoir solliciter une solution à tout moment, même si elle ne satisfait pas toutes les contraintes. On peut également raffiner ce modèle pour résoudre des problèmes VCSP en spécifiant les critères d'optimalité dans la fonction de coût.

2.3.2 CSP dynamiques

Certains problèmes industriels surviennent dans un environnement dynamique : des paramètres du problème original sont modifiés et on désire déterminer une solution qui respecte le nouveau problème à partir d'une solution précédente ou au cours de la recherche. Par exemple, on peut vouloir ajouter ou retirer des tâches à un système de *scheduling* au fur et à mesure que de nouvelles informations deviennent disponibles ou modifier l'état des ressources pour tenir compte de pannes. Dans une application graphique interactive, la scène doit évoluer pour respecter les contraintes posées sur les objets qu'elle contient en tenant compte des modifications effectuées par l'utilisateur.

Les CSP *dynamiques* (DCSP) formalisent ce type de problème en considérant une suite de CSP dont deux problèmes successifs diffèrent par l'ajout ou le retrait de contraintes, éventuellement unaires (modifications de domaines). On souhaite alors construire une nouvelle solution plus efficacement qu'en réitérant une recherche classique. Pour certains problèmes, on veut également que la nouvelle solution soit proche de la dernière solution calculée (« robustesse » de la solution).

[7] Cette structure est un « monoïde commutatif totalement ordonné » muni d'un élément minimum et d'un élément maximum. L'opérateur binaire doit satisfaire des propriétés décrites dans [Schiex 95].

[8] On peut aussi associer un coût à chaque tuple de chaque contrainte, mais le principe reste le même.

Les solveurs PPC classiques peuvent en général ajouter des contraintes dynamiquement lors de la résolution d'un CSP — c'est d'ailleurs souvent l'opération de base lors de la recherche — mais l'opération de retrait est contraire à la propriété de monotonie, essentielle pour ces solveurs. Cependant, si une solution a déjà été obtenue, le retrait de contraintes permet de la conserver, alors que l'ajout la rend éventuellement inconsistante. [Verfaillie 95] présentent différentes techniques pour résoudre efficacement les DCSP grâce à la mémorisation et la réutilisation de solutions pour guider l'heuristique d'algorithmes de PPC ou de RL et/ou la mémorisation et la réutilisation de contraintes en PPC pour identifier et conserver les déductions toujours valides dans le nouveau problème lors du retrait de contraintes.

Chapitre 3

Résolution des CSP

La méthode exacte la plus naïve pour résoudre un CSP est d'énumérer toutes les combinaisons de valeurs possibles (instanciations totales) jusqu'à ce que l'on en trouve une qui soit consistante. Si aucune solution n'a été trouvée après la vérification de chaque instanciation, on obtient la preuve que le problème est inconsistant. Mais cette approche conduit naturellement à une complexité exponentielle qui la rend impraticable sur des CSP de grande taille. Les techniques de résolution *systématiques* fondées sur ce principe construisent une solution par l'extension d'une instanciation partielle ; le choix des extensions (i.e. l'affectation d'une valeur à une variable non instanciée) est remis en cause quand une inconsistance est détectée — c'est le *retour arrière* ou *backtrack*. Ces algorithmes tentent d'améliorer l'efficacité de la recherche en exploitant les contraintes des CSP pour *inférer* des réductions de l'espace de recherche, ainsi que les informations découvertes depuis le début de la recherche. De telles méthodes de résolution sont élégamment intégrés dans le cadre de la *Programmation Par Contraintes* qui constitue un langage de modélisation de CSP et de spécification de stratégie de recherche.

Mais quelques soient les raffinements apportés, ces méthodes complètes restent exponentielles pour un CSP quelconque et peuvent donc être trop lentes pour résoudre des problèmes de grande taille. Dans ce cas, une méthode approchée de *Recherche Locale*, ou *méta-heuristique*, peut être utilisée en sacrifiant la complétude de la recherche. Ce type de recherche pratique des modifications *locales* successives à partir d'une instanciation *totale* initiale : un *voisinage* est défini autour de chaque instanciation totale et une *heuristique* est responsable de la sélection du prochain voisin pour tenter de se « diriger » vers les régions de l'espace de recherche qui contiennent des solutions. Ces instanciations totales sont éventuellement inconsistantes et une solution est obtenue lorsque plus aucune contrainte n'est violée. De telles techniques peuvent donc également fournir des « solutions » approchées pour les problèmes sur-contraints et lorsque le temps de résolution est limité.

L'hybridation de ces deux paradigmes très différents pour profiter de leurs avantages respectifs semblent une voie de recherche prometteuse, mais les concepts qu'ils manipulent sont difficiles à concilier. De nombreuses combinaisons, souvent dédiées à des problèmes particuliers, ont été proposées mais il n'existe pas encore de cadre élégant qui puisse unifier

les deux techniques de manière générique[1].

Nous présentons d'abord dans la section suivante les techniques de recherche systématique les plus utilisées pour résoudre les CSP en terminant par le paradigme de la Programmation Par Contraintes, un outil qui rencontre un succès commercial croissant dans le monde industriel. Les méthodes de Recherche Locale font ensuite l'objet de la section 3.2 et la section 3.3 explore les opportunités d'hybridation des deux approches. Enfin, nous passons en revue section 3.4 quelques systèmes existants de Programmation Par Contraintes et de Recherche Locale ainsi que les langages de modélisation dont l'ambition est de permettre d'écrire des algorithmes hybrides.

3.1 Recherche systématique

3.1.1 Générer et tester

La méthode la plus naïve pour chercher une solution d'un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est d'énumérer tous les tuples de $\mathcal{D}$ en vérifiant à chaque fois si les instanciations totales correspondantes sont consistantes. On peut vérifier chaque contrainte avec une complexité polynomiale (i.e. la vérification est un problème « facile ») si toutes ses variables sont instanciées mais il faut éventuellement parcourir tout l'espace de recherche avant de trouver une solution, et c'est même nécessaire si on souhaite obtenir un optimum ou l'ensemble des solutions du problème. Cet espace a la taille de $\mathcal{D}$, et la complexité de l'algorithme décrit, appelé « *générer et tester* » (GT), est donc exponentielle avec le nombre de variable — on doit vérifier de l'ordre de $\mathcal{O}(md^n_{\max})$ contraintes.

Pour parcourir cet espace systématiquement, on structure la recherche sous la forme d'un arbre dont chaque nœud correspond à une instanciation partielle et ses fils à l'extension de cette instanciation avec une nouvelle variable, chaque fils correspondant à une valeur de son domaine. L'algorithme GT parcourt cet arbre en profondeur d'abord[2], avec une complexité spatiale en $\mathcal{O}(n)$.

L'algorithme maintient $\mathcal{V}$, l'ensemble des variables qui ne sont pas encore instanciées, et $\mathcal{I}$, l'instanciation partielle courante qui porte sur les autres variables ($\mathcal{X} - \mathcal{V}$). Au début de la recherche, la fonction de l'algorithme 1 est appliquée à l'ensemble des variables et l'instanciation partielle courante est vide : générer_et_tester($\mathcal{X}$, $\emptyset$). Puis à chaque appel récursif, on choisit une nouvelle variable x de $\mathcal{V}$ et on essaye de compléter $\mathcal{I}$ avec l'une des valeurs de son domaine. Quand $\mathcal{I}$ est totale ($\mathcal{V} = \emptyset$), on vérifie si elle satisfait toutes les contraintes, et si c'est le cas, on renvoie la solution. Si l'algorithme termine et qu'aucune solution n'a été renvoyée, le problème est inconsistant.

On peut facilement modifier cet algorithme pour calculer une solution optimale en continuant la recherche même si une solution est trouvée. À chaque nouvelle solution, on compare son coût à celui de la meilleure solution antérieure et on la remplace s'il est

[1] L'approche de [Laburthe 98b] est néanmoins une tentative des plus élégantes pour s'en rapprocher.

[2] Un autre type de parcours serait nécessairement plus coûteux en mémoire et aurait peu d'intérêt car les solutions se trouvent toutes à une profondeur constante (n).

Algorithme 1 – générer_et_tester($\mathcal{V}$, $\mathcal{I}$)

```
1: if V = ∅ then
2:   if ⋀_{c∈C} c̃(I) then
3:     return I {une solution}
4:   end if
5: else
6:   x ∈ V {choix d'une nouvelle variable}
7:   for all a ∈ D_x do
8:     I' := I ∪ (x, a) {choix d'une valeur}
9:     générer_et_tester(V\{x}, I')
10:  end for
11: end if
```

meilleur. L'optimum est obtenu quand l'algorithme termine, i.e. tout l'espace de recherche a été parcouru. Une variante évidente peut aussi calculer l'ensemble des solutions d'un CSP.

Ce type de recherche constitue la fondation des algorithmes systématiques de résolution des CSP. On peut généraliser la notion de nœud (d'un arbre de recherche) en considérant qu'il correspond à une *décision* imposée sur le problème pour le rendre plus simple. Pour les CSP, ces décisions peuvent être vues comme l'ajout dynamique de contrainte : l'algorithme 1 ajoute à chaque nœud la contrainte $\{x = a\}$ à $\mathcal{C}$. La recherche est complète si *toutes* les autres valeurs sont également essayées. Pour simplifier, on peut se limiter à des arbres de décisions binaires qui ajoutent à chaque nœud soit une contrainte c, soit sa négation $\neg c$ définie par l'ensemble de tuples qui rendent fausse sa fonction de satisfaction. c et $\neg c$ partitionne en deux l'espace de recherche du CSP courant (i.e. le problème original restreint par les contraintes ajoutées depuis la racine de l'arbre jusqu'au nœud considéré), si bien que les solutions obtenues dans les deux branches constituent également une partition de l'ensemble des solutions de ce CSP :

$$S_{(\mathcal{X},\mathcal{D},\mathcal{C})} = S_{(\mathcal{X},\mathcal{D},\mathcal{C}\cup\{c\})} \cup S_{(\mathcal{X},\mathcal{D},\mathcal{C}\cup\{\neg c\})} \quad \text{et} \quad S_{(\mathcal{X},\mathcal{D},\mathcal{C}\cup\{c\})} \cap S_{(\mathcal{X},\mathcal{D},\mathcal{C}\cup\{\neg c\})} = \emptyset$$

On peut alors considérer que l'algorithme 1 impose à chaque nœud soit la contrainte $\{x = a\}$ avec $a \in \mathcal{D}_x$, soit sa négation $\{x \neq a\}$, jusqu'à ce que le domaine de x soit un singleton, auquel cas une nouvelle variable doit être choisie.

Pour résoudre le CSP de l'exemple 3.1, qui n'a qu'une solution $\{(x,4),(y,1),(z,4)\}$, GT doit parcourir entièrement l'arbre de recherche de la figure 3.1 jusqu'à la branche dont les instanciations sont encadrées. Le calcul de *toutes* les solutions nécessitera un parcours exhaustif.

Exemple 3.1

$$\begin{cases} \mathcal{X} &= \{x, y, z\} \\ \mathcal{D} &= \{\{2,3,4\},\{1,2,3\},\{4,5\}\} \\ \mathcal{C} &= \{xy = z, x > y\} \end{cases}$$

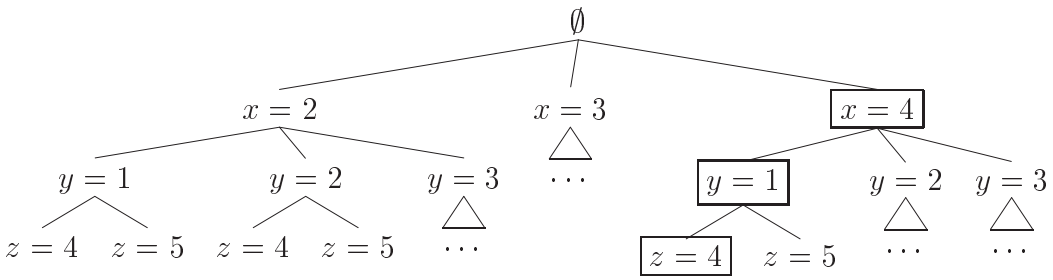


FIG. 3.1 – L'algorithme GT sur l'exemple 3.1

Cet algorithme est très coûteux en temps de calcul mais il peut incorporer de nombreuses améliorations en exploitant la structure des CSP et l'information issue des étapes précédentes de la recherche, ce qui fait l'objet des deux sections suivantes.

3.1.2 Backtrack

L'un des défauts de l'algorithme GT est de ne vérifier la satisfaction des contraintes qu'une fois l'instanciation totale obtenue, à chaque feuille de l'arbre de recherche. Or on peut tester la satisfaction d'une contrainte $c = (\mathcal{X}_c, \widetilde{c})$ dès que les variables qui la concernent sont instanciées $\mathcal{X}_c \subseteq \mathcal{X}_\mathcal{I}$. Si on détecte qu'une instanciation partielle est inconsistante car elle viole une contrainte c, on pourra alors élaguer le sous-arbre correspondant à ses extensions car toutes violeront c. Il suffit alors d'effectuer un *retour arrière* ou *backtrack* dans l'arbre jusqu'à la première instanciation antérieure qui offre une autre branche non encore explorée.

On modifie GT pour obtenir l'algorithme 2 *backtrack* (BT) en testant la consistance de l'instanciation partielle à chaque nouvelle valeur essayée (ligne 7). Pour ne pas tester plusieurs fois les contraintes déjà satisfaites, on se contente de celles qui vérifient :

$$c \in \mathcal{C} \wedge x \in \mathcal{X}_c \wedge \mathcal{X}_c \subseteq \mathcal{X}_{\mathcal{I}'}$$

c'est-à-dire celles qui n'ont pas été testées précédemment car elles concernent la variable x, inconnue dans $\mathcal{I}$ mais instanciée par $\mathcal{I}'$.

Appliqué à l'exemple 3.1, BT s'épargne le développement des sous-arbres (marquées par un carré dans la figure 3.2) qui violent la contrainte $x > y$ dès que ces deux variables sont instanciées. L'algorithme BT est donc systématiquement meilleur que GT mais sa complexité reste évidemment exponentielle en pire cas — par exemple, ils seraient équivalents si toutes les contrainte portaient sur toutes les variables et ne pouvaient être vérifiées qu'au niveau des feuilles.

Mais l'algorithme BT souffre néanmoins de plusieurs défauts parce qu'il n'exploite pas plus avant la structure des CSP et ne mémorise pas toutes les déductions obtenues depuis le début de la recherche :

– Lors d'un retour arrière, BT remet en cause la *dernière* variable instanciée (on appel parfois cet algorithme *backtrack chronologique*) alors qu'elle n'est pas forcément la

Algorithme 2 – backtrack($\mathcal{V}$, $\mathcal{I}$)

1: **if** $\mathcal{V} = \emptyset$ **then**
2: return $\mathcal{I}$ {une solution}
3: **else**
4: $x \in \mathcal{V}$ {choix d'une nouvelle variable}
5: **for all** $a \in \mathcal{D}_x$ **do**
6: $\mathcal{I}' := \mathcal{I} \cup \{(x, a)\}$ {choix d'une valeur}
7: **if** $\bigwedge_{c \in \mathcal{C} \wedge x \in \mathcal{X}_c \wedge \mathcal{X}_c \subseteq \mathcal{X}_{\mathcal{I}'}} \widetilde{c}(\mathcal{I}')$ **then**
8: backtrack($\mathcal{V} \backslash \{x\}$, $\mathcal{I}'$)
9: **end if**
10: **end for**
11: **end if**

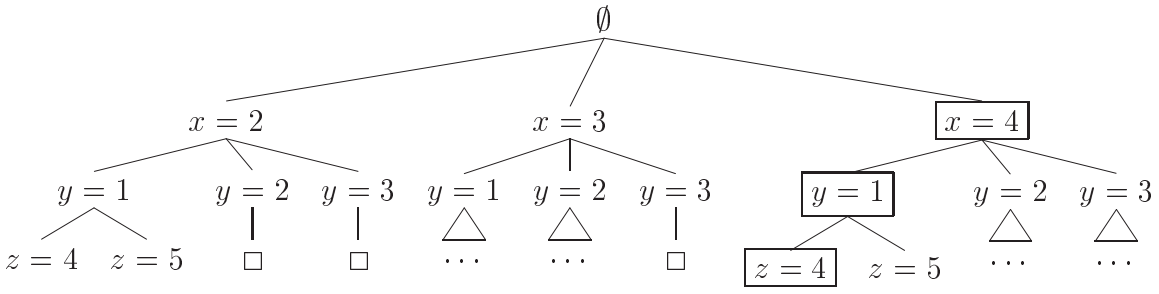


FIG. 3.2 – L'algorithme BT sur l'exemple 3.1

cause de l'échec. Les algorithmes de *backjumping* évitent cette source d'échecs en analysant la cause des inconsistances puis en effectuant un retour arrière directement vers une variable responsable de l'échec.
- Les instanciations partielles inconsistantes (appelées aussi *nogoods*) constatées au cours de la recherche doivent être redécouvertes dans les branches ultérieures de l'arbre. Les algorithmes d'*apprentissage de contraintes*[3] maintiennent les *nogoods* pertinents sous forme de contraintes ajoutées au CSP.
- BT attend que toutes les variables d'une contrainte soient instanciées avant de la vérifier. On peut en fait *inférer* des échecs bien avant en imposant un certain degré de *consistance locale* aux variables non instanciées.

Les deux premiers types d'algorithme qui concernent les variables déjà instanciées sont des techniques *rétrospectives* (*look-back*) passées en revue section 3.1.3 alors que le troisième est *prospectif* (*look-ahead*) car il infère des contraintes sur les variables non instanciées. Cette dernière technique est détaillée dans la section 3.1.4.

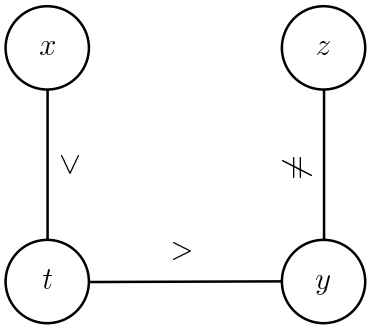
3.1.3 Backtrack « intelligent »

Backjumping

Considérons le CSP suivant :

Exemple 3.2

$$\begin{cases} \mathcal{X} &= \{x, y, z, t\} \\ \mathcal{D} &= \{\{2,3\}, \{1,2\}, \{1,2,3\}, \{2,3\}\} \\ \mathcal{C} &= \{x > t, y \neq z, y < t\} \end{cases}$$



La figure 3.3 représente une partie de l'arbre de recherche correspondant.

Sur ce problème, l'algorithme BT, après avoir échoué dans la branche la plus à gauche $\{(x,2),(y,1),(z,1)\}$ car $y \neq z$, parvient à instancier z à 2 puis échoue sur toutes les valeurs de t. Cet échec sur le domaine entier de t le conduit à *backtracker*[4] pour tenter l'instanciation $z = 3$. Mais t n'a aucune contrainte commune avec z, si bien qu'à ce nœud de l'arbre ($\{(x,2),(y,1)\}$), on ne peut obtenir de solution quelque soit la valeur de z.

Pour éviter d'explorer le sous-arbre correspondant à $\{(x,2),(y,1),(z,3)\}$ comme le fait BT (cf. figure 3.3), il faut backtracker au moins jusqu'à une variable qui ait une contrainte commune avec t, c'est-à-dire x ou y. C'est le principe général du *backjumping*, qui se

[3] On trouve dans la littérature *constraints* (ou *nogoods*) *recording* (ou *learning*).
[4] **Backtracker** *v. intr. (Anglicisme)* : Revenir en arrière.

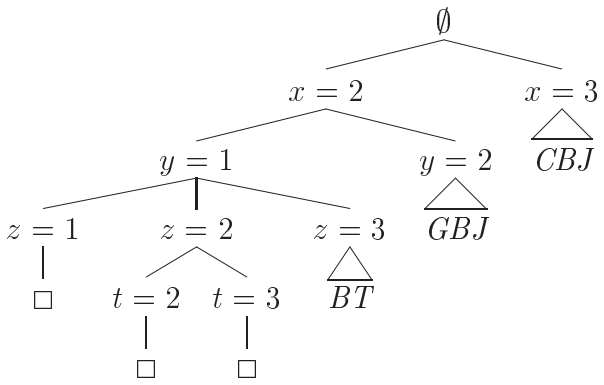


FIG. 3.3 – BT avec *backjumping* sur l'exemple 3.2.

décline en plusieurs variantes suivant la technique utilisée pour déterminer la variable sur laquelle backtracker ; nous en évoquons deux, référencées dans [Dechter 98] qui étudie leur complétude et la pertinence du choix de la variable sur laquelle backtracker.

L'algorithme *graph-based backjumping* (GBJ) utilise uniquement le graphe de contraintes pour choisir une variable. Pour être sûr de ne pas manquer de solution, on doit remettre en cause la variable instanciée le plus récemment parmi toutes les candidates, donc y dans notre cas. GBJ va alors directement explorer le sous-arbre $\{(x = 2), (y = 2)\}$ et s'épargne une partie du travail réalisé par BT (l'exploration du sous-arbre $\{(x, 2), (y, 1), (z, 3)\}$). Si $y = 1$ était déjà la dernière branche au moment du *backtrack*, on aurait continuer à « sauter » en arrière vers une variable impliquée dans une contrainte soit avec t, soit avec y.

Cependant, backtracker vers y ne résout pas l'échec car la contrainte $y < t$ n'est pas violée dans la branche $\{(x, 2), (y, 1), (z, 2)\}$. L'inconsistance est en fait due à la contrainte $x > t$, qui est la plus récente à justifier l'échec de l'instanciation de t (i.e. sa vérification indique une inconsistance). L'algorithme *conflict-directed backjumping* (CBJ) maintient ainsi pour la variable courante l'ensemble des variables instanciées en conflit avec elle et backtracke vers la plus récente en cas d'échec. Ainsi, CBJ peut directement « sauter » vers la branche $x = 3$ et économise le parcours du sous-arbre $\{(x = 2), (y = 2)\}$, puisque seule une instanciation de x différente de 2 peut satisfaire $x > t$.

Apprentissage de contraintes

On choisit à présent d'instancier les variables de l'exemple 3.2 dans un ordre différent : z, x, t puis y. La figure 3.4 montre l'arbre de recherche de BT sur ce CSP.

On peut constater sur la branche la plus à gauche correspondant à l'instanciation $\mathcal{I}_1 = \{(z, 1), (x, 2)\}$ qu'aucune des valeurs de t ne convient à cause de la contrainte $x > t$, donc $\mathcal{I}_1$ est inconsistante. On appelle *nogood* [Schiex 93] une telle instanciation partielle qui ne fait partie d'aucune solution[5]. Un nogood peut donc être vu comme une nouvelle

[5]On dit également que $\mathcal{I}_1$ est une instanciation partielle *globalement* inconsistante.

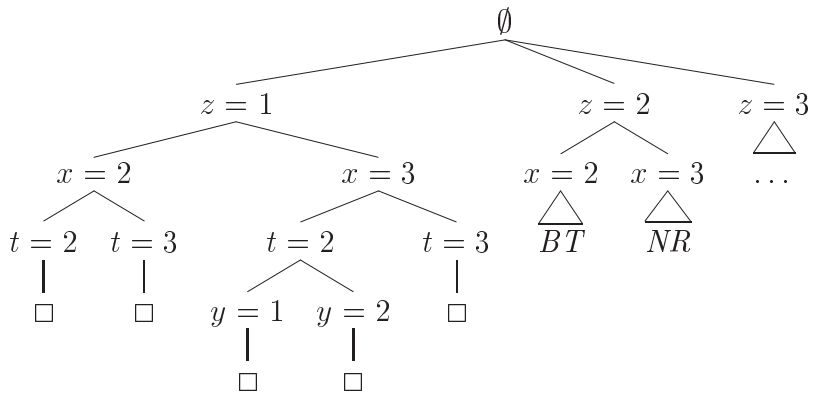


FIG. 3.4 – BT avec apprentissage de contraintes sur l'exemple 3.2.

contrainte que l'on peut ajouter globalement au problème sans modifier l'ensemble de ses solutions. Comme z n'intervient pas dans la contrainte violée, on peut limiter le nogood à $\{x = 2\}$ et on peut donc ajouter au CSP 3.2 la contrainte unaire $x \neq 2$. Or l'algorithme BT va tenter par la suite de chercher une solution dans la branche $\{z = 2, x = 2\}$ alors qu'elle ne peut pas, comme on l'a déduit précédemment, être étendue en une solution.

Pour éviter cette exploration redondante, les algorithmes de *nogoods recording* entretiennent par différentes techniques une « base de données » de nogoods au sein d'un algorithme de *backtrack* et exploitent donc les informations déduites depuis le début de la recherche pour poser de nouvelles contraintes sur le CSP. En effet, à chaque feuille de l'arbre parcouru par BT, une instanciation partielle globalement inconsistante $\mathcal{I}$ est détectée car elle viole une contrainte c et on pourrait donc ajouter $\mathcal{I}$ à l'ensemble des contraintes du CSP. Mais le stockage de ces nogoods peut prendre un espace de taille exponentielle avec le nombre de variables. Or, comme on l'a mentionné précédemment, les variables de $\mathcal{I}$ impliquées dans c suffisent à provoquer l'inconsistance, donc la projection de $\mathcal{I}$ sur $\mathcal{X}_c$ est elle-même un nogood. Cette propriété est d'autant plus intéressante que les nogoods les plus courts sont les meilleurs : si la contrainte correspondante est imposée, une plus grande partie de l'espace de recherche peut être élaguée car moins de variables instanciées sont nécessaires pour vérifier sa consistance.

De plus, les nogoods mémorisés peuvent être combinés pour en produire de nouveaux. Un nogood peut être vu comme une implication : la branche la plus à gauche de l'arbre 3.4, par exemple, correspond au nogood $\{(x, 2), (t, 2)\}$, ce qui peut également s'écrire

$$x = 2 \;\Rightarrow\; t \neq 2$$

De même la branche voisine correspond au nogood $x = 2 \;\Rightarrow\; t \neq 3$. Comme 2 et 3 sont les seules valeurs possibles de t, on en déduit que $\{(x, 2)\}$ est un nogood. BT applique en fait implicitement cette technique lors d'un retour arrière sans projeter le nogood sur les contraintes violées et en le mémorisant grâce à l'instanciation partielle courante. Plus généralement, les algorithmes de *nogoods recording* utilisent la forme de résolution suivante

pour générer de nouveaux nogoods à partir de ceux qui ont été mémorisés : soit une variable x de domaine $\mathcal{D}_x = \{v_k, \forall k \in [1, s]\}$ et s nogoods de la forme $\mathcal{I}_k \Rightarrow x \neq v_k$ alors $\bigcup_{k=1}^{s} \mathcal{I}_k$ est un nogood. Donc si un ensemble de nogoods élimine toutes les valeurs possibles d'une variable x, on peut en déduire un nouveau nogood, éventuellement plus court, dans lequel x n'apparaît pas. La contrainte $\{x \neq 2\}$ ajoutée au problème 3.2 par un algorithme doté de *nogoods recording* (NR) permet alors d'éviter l'exploration du sous-arbre $\{(z, 2), (x, 2)\}$ comme l'illustre la figure 3.4.

Les informations collectées par les algorithmes de *nogood recording* sont proches de celles utilisées pour le *backjumping* (cf. section 3.1.3), mais elles ne sont pas utilisée de la même manière[6]. De nombreux algorithmes de *backtrack* hybrides [Schiex 93, Ginsberg 93, Dechter 98] profitent donc à la fois de ces deux améliorations en limitant l'ensemble des nogoods produits pour garder une complexité temporelle et spatiale polynomiale. Ces raffinements de BT peuvent en effet être plus coûteux en temps d'exécution sur certains problèmes que l'algorithme de base s'ils ne permettent pas d'élaguer des portions suffisamment larges de l'espace de recherche ; ainsi, même si les nombres de nœuds visités et de retours arrières sont systématiquement inférieurs à celui de BT, le maintien des structures de données nécessaires peut ralentir la recherche.

Mentionnons finalement qu'une autre technique, le *backmarking* (BM), permet d'éviter des tests de consistance redondants, supposés coûteux, (mais pas d'éliminer des sous-arbres) en mémorisant leur résultat puis en le réutilisant s'il est toujours valide. L'espace requis par BM reste polynomial[7] mais cet algorithme ne permet pas de déduire de nouvelles contraintes et de réduire le nombre de nœuds visités.

3.1.4 Propagation de contrainte

L'algorithme BT et ses raffinements fondés sur des schémas rétrospectifs se contentent de vérifier la satisfaction d'une contrainte quand toutes ses variables sont instanciées. Or, plus les inconsistances sont détectées précocement, plus la portion de l'espace de recherche éliminée est importante. Les techniques *prospectives* d'amélioration de BT tentent de maintenir des propriétés de *consistance locale*, moins forte que la consistance « globale » (i.e. le problème est résolu) et de complexité polynomiale, à chaque nœud de l'arbre de recherche. Ces propriétés de consistance locale impliquent des variables qui n'ont pas été encore instanciées et éliminent des valeurs de leur domaine pour éviter des échecs *a priori*.

Consistance locale

Établir la consistance « globale » d'un CSP revient à le résoudre. On peut néanmoins définir des propriétés de consistance *locale* plus faibles qui permettent de simplifier le CSP en inférant des contraintes induites (i.e. qui ne changent pas l'ensemble des solutions) et qui peuvent être établies avec des algorithmes de complexité polynomiale.

[6] Les effets des deux techniques peuvent pourtant être identiques sur l'élagage de certains sous-arbres.

[7] BM stocke une information de taille constante pour chaque variable et pour chaque valeur de chaque variable, donc l'espace nécessaire a une taille en $\mathcal{O}(nd_{\max})$.

Node consistency Sur l'exemple 3.2, l'algorithme NR découvre la contrainte induite $x \neq 2$ dans le sous-arbre $\{(z,1)\}$ mais instancie à nouveau x à 2 dans le sous-arbre $\{(z,2)\}$ avant de découvrir que la contrainte $x \neq 2$ est violée. On pourrait donc transformer le CSP 3.2 en remplaçant $\mathcal{D}_x$ (le domaine de x) par $\mathcal{D}_x/\{2\} = \{3\}$. On obtient alors un nouveau CSP équivalent au précédent mais avec un espace de recherche moindre. C'est la technique de consistance locale la plus simple.

On dit qu'un CSP est *node consistent*[8] si toutes les contraintes unaires (d'arité 1) sont satisfaites :

Définition 3.1 (*Node consistency*) *Un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est* node consistent *si et seulement si $\forall x \in \mathcal{X}$, $\mathcal{D}_x \neq \emptyset$ et $\forall v \in \mathcal{D}_x$, $\forall c \in \mathcal{C}$ telle que $\mathcal{X}_c = \{x\}$, alors $\widetilde{c}((v)) = vrai$.*

En supposant qu'un CSP a au plus n contraintes unaires[9], un algorithme qui transforme un CSP en un problème *node consistent* aura une complexité en $\mathcal{O}(nd_{\max})$: il suffit de parcourir l'ensemble des contraintes unaires (c) et de remplacer le domaine des variables concernées (x) par $\mathcal{D}_x \cap \mathcal{R}_c$ — on suppose, ce qui a un intérêt opérationnel, que les contraintes peuvent contenir des valeurs qui ont été supprimées des domaines. Ces contraintes unaires peuvent ensuite être retirées du CSP.

Arc-consistance Sur l'exemple 3.2, les variables x et t de domaine $\{2,3\}$ sont liées par la contrainte $x > t$. Si $x = 2$, aucune valeur de t ne permet à la contrainte d'être satisfaite, donc on peut retirer 2 de $\mathcal{D}_x$ et obtenir un CSP équivalent. De même, une instanciation partielle où $t = 3$ ne peut être consistante, donc on peut retirer 3 de $\mathcal{D}_t$. Ces réductions de domaine deviennent évidentes si on exprime la contrainte en extension : $(\{x,t\}, \{(3,2)\})$. On peut ainsi supprimer du domaine d'une variable toute valeur qui ne permet pas de trouver un tuple satisfaisant une des contraintes qui la concernent et construit avec des valeurs appartenant aux domaines des autre variables impliquées dans la contrainte. On dit d'une telle valeur (qui ne peut faire partie d'aucune solution) qu'elle n'a pas de *support* dans les domaines des autre variables. On définit cette notion dans le cas des CSP binaires :

Définition 3.2 (Support d'une valeur) *Soit un CSP binaire $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ et $c = (\{x,y\}, \mathcal{R}_c) \in \mathcal{C}$ une contrainte de $\mathcal{P}$ sur les variables x et y. Une valeur $v_y \in \mathcal{D}_y$ du domaine de y est un* support *pour une valeur $v_x \in \mathcal{D}_x$ du domaine de x si et seulement si $(v_x, v_y) \in \mathcal{R}_c$, i.e. (v_x, v_y) satisfait c.*

La propriété d'*arc-consistance* [Mackworth 77] est définie sur les arcs, i.e. les contraintes, du graphe associé à un CSP binaire. Elle caractérise une forme *locale* de consistance et une *approximation* (forte) de la consistance globale du CSP.

[8] On pourrait traduire (en bon français) par *cohérence de nœud* qui fait référence au graphe de contraintes associé à un CSP (cf. section 2.1.1).

[9] On ne perd pas en généralité car on peut « fusionner » deux contraintes en extension portant sur les mêmes variables en une seule ayant comme relation leur intersection. On peut également transformer facilement toute contrainte *unaire* en intention en contrainte en extension.

Définition 3.3 (Arc-consistance) *Un arc orienté (x, y) du graphe associé à un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, et correspondant à une contrainte $c = (\{x, y\}, \mathcal{R}_c) \in \mathcal{C}$, est* arc-consistant *si et seulement si toutes les valeurs du domaine de x ont un support dans le domaine de y.*

Notons que l'arc-consistance d'un arc est *directionnelle* : si (x, y) est consistant, (y, x) ne l'est pas forcément.

On peut simplement rendre un arc (x, y) arc-consistant en éliminant de $\mathcal{D}_x$ les valeurs qui n'ont pas de support dans $\mathcal{D}_y$, et on obtient un nouveau CSP équivalent. L'algorithme 3 [Mackworth 77] rend un arc (x, y) consistant et renvoie un booléen indiquant si le domaine de x a été modifié. On note $\mathcal{R}_{(x,y)}$ la relation qui définit la contrainte sur l'arc (x, y).

Algorithme 3 – révise(x, y)

```
1: modif := faux
2: for all v_x ∈ D_x do
3:    if ∄v_y ∈ D_y telle que (v_x, v_y) ∈ R_(x,y) then
4:       D_x := D_x/{v_x}
5:       modif := vrai
6:    end if
7: end for
8: return modif
```

Un CSP binaire est *arc-consistant* si tous ses arcs orientés (i.e. ses contraintes en ordonnant leurs variables dans un sens, puis dans l'autre) sont arc-consistants. Toute instanciation partielle consistante de taille 1 pourra alors être étendue en une instanciation partielle consistante de taille 2.

Définition 3.4 (Arc-consistance d'un CSP binaire) *Un CSP binaire $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est* arc-consistant *si et seulement si $\forall c = (\{\{x, y\}, \mathcal{R}_c) \in \mathcal{C}$, les arcs (x, y) et (y, x) sont arc-consistants.*

En général, on souhaite également qu'aucun domaine ne soit vide (sinon le problème est trivialement inconsistant) ou plus généralement que le problème soit aussi *node consistent* si on accepte des contraintes unaires. Toute instanciation partielle consistante de taille 1 pourra alors être étendue avec n'importe quelle nouvelle variable.

Pour établir l'arc-consistance d'un CSP binaire en supprimant des valeurs de ses domaines, il ne suffit pas d'exécuter l'algorithme « révise » une fois pour chaque arc parce que l'élimination d'une valeur pour une variable x peut rendre caduque l'arc-consistance des arcs impliquant x et traités précédemment. Sur le CSP de l'exemple 3.2, si on commence par rendre arc-consistant les arcs (y, t) et (t, y) associés à la contrainte $y < t$, aucune valeur de leur domaine n'est supprimée car $\{(y, 1), (t, 2)\}$ et $\{(y, 2), (t, 3)\}$ sont des instanciations partielles consistantes ; si on rend ensuite les arcs de $x > t$ arc-consistants, on supprime 2 du domaine de x et 3 de celui de t, donc on doit de nouveau établir l'arc-consistance des arcs de $y < t$, ce qui élimine 2 du domaine de y. On doit donc reconsidérer les arcs déjà

traités dont l'extrémité « droite » est modifiée ultérieurement — c'est pourquoi la fonction « révise » indique si le domaine de x a changé. Finalement, la valeur 1 est éliminée du domaine de z pour rendre arc-consistants les arcs de $y \neq z$. On obtient alors un nouveau CSP équivalent :

$$\begin{cases} \mathcal{X} &= \{x, y, z, t\} \\ \mathcal{D} &= \{\{3\}, \{1\}, \{2, 3\}, \{2\}\} \\ \mathcal{C} &= \{x > t, y \neq z, y < t\} \end{cases}$$

Sur ce problème simple, l'arc-consistance a ainsi permis d'obtenir la consistance globale.

De nombreux algorithmes d'établissement de l'arc-consistance d'un CSP binaire, dont les principaux sont appelés AC-n suivant leur version ($n \in [1, 7]$), ont été proposés depuis l'émergence des techniques de consistance dans les années 70. L'un des plus simples est AC-3 [Mackworth 77], de complexité $\mathcal{O}(md^3_{\max}$ (m étant le nombre de contraintes et $d_{\max}$ la taille du plus grand domaine), qui utilise la fonction « révise » : AC3 utilise un ensemble

---

Algorithme 4 – AC3

1: $\mathcal{Q} := \bigcup_{c=(\{x,y\},\mathcal{R}_c)\in\mathcal{C}} \{(x, y), (y, x)\}$
2: **while** $\mathcal{Q} \neq \emptyset$ **do**
3: $\quad (x, y) \in \mathcal{Q}$
4: $\quad \mathcal{Q} := \mathcal{Q}/(x, y)$
5: $\quad$ **if** révise(x, y) **then**
6: $\quad\quad \mathcal{Q} := \mathcal{Q} \cup \{(z, x),\ \exists c = (\{x, z\}, \mathcal{R}_c) \in \mathcal{C},\ z \neq y\}$
7: $\quad$ **end if**
8: **end while**

---

d'arcs $\mathcal{Q}$ initialisé avec tous les arcs du CSP. Tant qu'il n'est pas vide, on lui retire un arc (x, y) que l'on rend arc-consistant grâce à l'algorithme 3 (« révise »). Si le domaine de x en a été modifié, on replace dans $\mathcal{Q}$ tous les arcs de la forme (z, x) (avec $z \neq y$) déjà traités. Cet algorithme a une complexité temporelle en $\mathcal{O}(md^3_{\max})$.

L'algorithme AC-4 [Mohr 86], de complexité temporelle optimale[10] en $\mathcal{O}(md^2_{\max})$, qui n'utilise pas la fonction « révise », maintient des compteurs pour dénombrer les supports de chaque valeur, avec une complexité spatiale élevée en $\mathcal{O}(md^2_{\max})$. AC-6 [Bessière 93] utilise le même principe mais ne maintient qu'un unique support pour chaque valeur, réduisant la complexité spatiale de l'algorithme à $\mathcal{O}(\mathcal{O}(md_{\max})$ ainsi que sa complexité temporelle en moyenne. AC-7 [Bessière 95] est fondé sur le même principe qu'AC-6 mais utilise le fait que les contraintes sont *bidirectionnelles* (i.e. l'arc (x, y) correspond à la même contrainte que l'arc (y, x)) pour améliorer l'efficacité de la recherche de support[11].

[10] En pire cas, établir l'arc-consistance d'un CSP nécessite de visiter chaque contrainte au moins une fois, et pour chaque contrainte (binaire), on peut avoir de l'ordre de $\mathcal{O}(d^2_{\max})$ couples de valeurs possibles. Donc $\mathcal{O}(md^2_{\max})$ est une borne inférieure de la complexité (en pire cas) des algorithmes AC-n.

[11] Si un support v_y a été trouvé pour une valeur v_x, alors v_x est aussi un support pour v_y.

Nous n'avons défini l'arc-consistance que pour des CSP binaires, mais [Bessière 97] adapte l'algorithme AC-7 (qui devient GAC[12]) aux cas des CSP quelconques, avec des contraintes d'arité quelconque associées à des hyper-arcs, en généralisant la définition 3.4 (on note $\pi_x(\tau)$ la projection d'un tuple τ sur la variable x) :

Définition 3.5 (Arc-consistance généralisée) *Soit un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ et une contrainte $c = (\mathcal{X}_c, \mathcal{R}_c) \in \mathcal{C}$. On dit que :*

- *un tuple $\tau \in \mathcal{R}_c$ est* valide *si et seulement si $\forall x \in \mathcal{X}_c$, $\pi_x(\tau) \in \mathcal{D}_x$;*
- *une valeur $v_x \in \mathcal{D}_x$ d'une variable $x \in \mathcal{X}_c$ est* consistante avec c *si et seulement si $\exists \tau \in \mathcal{R}_c$ tel que $\pi_x(\tau) = v_x$ et que τ soit valide ;*
- *la contrainte c est* arc-consistante *si et seulement si $\forall x \in \mathcal{X}_c$, $\mathcal{D}_x \neq \emptyset$ et $\forall v_x \in \mathcal{D}_x$, v_x est consistante avec c.*

Le problème $\mathcal{P}$ est arc-consistant *si et seulement si toutes ses contraintes sont arc-consistantes.*

Les algorithmes d'arc-consistances ont été prolifiquement étudiés et il en existe des adaptations pour résoudre les CSP dynamiques (cf. section 2.3.2) [Bessière 91] ou de complexité moindre pour des classes particulières de CSP et de contraintes [Kumar 92]. Enfin, de nombreuses approximations de l'arc-consistance ont été définies .

k-consistance Après avoir établi l'arc-consistance du CSP 3.2, on obtient un CSP satisfiable et même globalement consistant. Mais dans le cas général, on obtient un problème plus petit qui nécessite tout de même d'effectuer une recherche de solution. Par exemple, le CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1, 2\}\}, \{x \neq y, x \neq z, y \neq z\})$ n'est pas modifié par l'application d'un algorithme AC bien qu'il ne soit pas satisfiable.

On peut définir des propriétés de consistance « intermédiaires » entre l'arc-consistance et la consistance globale pour tenter de réduire encore plus l'espace de recherche en inférant des contraintes induites d'arité plus grande que celles, unaires, liées à l'établissement de l'arc-consistance. Nous avions remarqué précédemment que l'arc-consistance (pour un CSP binaire) équivalait à pouvoir étendre n'importe quelle instanciation partielle consistante de taille 1 avec toute autre nouvelle variable. La *k-consistance* généralise cette propriété :

Définition 3.6 (k-consistance) *Un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est* k-consistant *si et seulement si $\forall \mathcal{I} = (\mathcal{X}_\mathcal{I} = \{x_1, \ldots, x_{k-1}\}, \mathcal{V}_\mathcal{I})$, une instanciation partielle consistante de $k-1$ variables, et $\forall x \in \mathcal{X}/\mathcal{X}_\mathcal{I}$, une variable qui n'est pas instanciée par $\mathcal{I}$, alors $\exists v \in \mathcal{D}_x$ telle que $\mathcal{I} \cup \{(x, v)\}$ soit consistante.*

Un CSP est fortement k-consistant *si et seulement s'il est i-consistant $\forall i \in [1, k]$.*

La *node consistency* et l'arc-consistance correspondent ainsi respectivement à la 1-consistance et à la 2-consistance pour des CSP binaires. D'autre part, un CSP qui est fortement n-consistant est globalement consistant car un peut étendre successivement n'importe quelle instanciation partielle jusqu'à une instanciation totale. Cependant, les algorithmes

[12]GAC (*Generalized Arc Consistency*) peut également établir l'arc-consistance pour des contraintes exprimées par un prédicat $(\mathcal{X}_c, \widetilde{c})$.

qui établissent la k-consistance pour $k > 2$ sont en général trop coûteux et l'établissement de la n-consistance forte est bien sûr de complexité exponentielle en fonction de n. D'autres formes de consistance locale (plus fortes ou plus faibles que l'arc-consistance) ont également été définies et évaluées [Debruyne 97] pour améliorer l'efficacité des solveurs quand l'arc-consistance est trop coûteuse ou trop aveugle.

Backtrack et inférence

Comme l'établissement de l'arc-consistance (et même de la k-consistance forte pour $k < n$) ne suffit pas à obtenir une solution dans le cas général, il est nécessaire de lui faire succéder une phase de *recherche* pour résoudre un CSP. Le principe général utilisé par la plupart des algorithmes de résolution de CSP, notamment en PPC, est d'utiliser ces techniques prospectives (*look-ahead*) de consistance locale au sein d'un algorithme de *backtrack*, éventuellement raffiné par des techniques rétrospectives (*look-back*) :

1. *Propagation de contraintes* : à chaque nœud de l'arbre de recherche exploré par BT, le CSP, restreint par une nouvelle affectation, est simplifié en inférant des contraintes induites grâce à des algorithmes d'arc-consistance (ou d'une approximation de l'arc-consistance) qui *filtrent* les domaines.
2. *Élagage* : si un domaine devient vide, le sous-CSP correspondant au nœud n'est pas consistant, donc il est élagué et un retour arrière est effectué en rétablissant les domaines modifiés ;
3. *Exploration* : si tous les domaines sont réduits à un singleton, une solution est obtenue, sinon un nouveau nœud est exploré (i.e. une nouvelle variable est instanciée).

On peut d'ailleurs considérer BT comme une amélioration de GT auquel on ajoute l'établissement de l'arc-consistance sur les contraintes concernant des variables instanciées. De nombreuses variantes de l'hybridation de BT avec des formes de consistance locale existent entre le *backtrack* simple et l'établissement de l'arc-consistance du CSP entier à chaque nœud, appelé *Maintaining Arc Consistency* (MAC). Pour un CSP binaire, en supposant que les variables sont sélectionnées dans l'ordre croissant de leur indice ($i \in [1, n]$) et que la variable qui vient d'être instanciée est x_i (variable courante), ces algorithmes appliquent à chaque nœud la fonction « révise » aux arcs du CSP de la manière suivant [Schiex 92] :

- *Backtrack* (BT) : l'arc-consistance est établie entre la variable courante et les variables instanciées, i.e. révise(x_i, x_k) avec $1 \leq k < i$.
- *Forward Checking* (FC) : l'arc-consistance est établie pour les arcs qui lient x_i aux variables non-instanciées, i.e. révise(x_k, x_i) avec $i < k \leq n$.
- *Partial Look-ahead* (PL) : l'arc-consistance est établie également pour une partie des arcs impliquant des variables non-instanciées, i.e. révise(x_j, x_k) avec $i \leq j < k \leq n$.
- *Full Look-ahead* (FL) : l'arc-consistance est établie également pour tous les arcs impliquant des variables non-instanciées, i.e. révise(x_j, x_k) avec $i \leq j \neq k \leq n$.
- *Really Full Look-ahead* (RFL) : l'arc-consistance complète est établie pour le CSP. Une seule itération de « révise » sur tous les arcs ne suffit donc pas (cf. section 3.1.4),

contrairement aux précédents algorithmes qui ne considèrent les arcs qu'une unique fois. On peut utiliser par exemple un algorithme AC-n.

Plus les contraintes sont propagées, moins le nombre de nœuds explorés sera grand, mais l'efficacité globale peut être moindre car le traitement à chaque nœud est plus coûteux. Le choix de la meilleure technique n'est pas systématique et dépend fortement du CSP à résoudre ainsi que des autres sophistications apportées au *backtrack* (*look-back scheme*, cf. section 3.1.3, et heuristiques, cf. la section suivante) ; cette question est examinée dans la suite.

3.1.5 Heuristiques d'instanciation

Dans de la description des algorithmes de *backtrack* des sections précédentes, la stratégie de sélection de la prochaine variable à instancier n'est pas précisée (p.ex. ligne 4 de l'algorithme 2 : $x \in \mathcal{V}$). Or cette stratégie influe sur l'ordre de vérification des contraintes, la puissance de filtrage de la propagation de contraintes et l'utilité des techniques de *look-back*, ce qui a un impact très important sur le nombre de nœuds explorés. De même, l'algorithme 2 ne spécifie pas la manière de choisir la valeur d'instanciation d'une variable (ligne 5) : $a \in \mathcal{D}_x$. Cet ordre conditionne lui aussi l'efficacité de la recherche d'une solution ou d'un optimum[13].

Ordre d'instanciation des variables

Si on utilise un algorithme tel que MAC pour résoudre le problème suivant :

Exemple 3.3

$$\begin{cases} \mathcal{X} &= \{x, y, z, t\} \\ \mathcal{D} &= \{[1,2], [1,2], [1,2], [1,100]\} \\ \mathcal{C} &= \{x \neq y, x \neq z, y \neq z\} \end{cases}$$

l'établissement initial de l'arc-consistance ne réduit aucun domaine (ce CSP est déjà arc-consistant). Si on commence par instancier x à 1, MAC va supprimer cette valeur du domaine de y et z, puis l'établissement de l'arc-consistance de $y \neq z$ va rendre vide le domaine de y (ou de z) et va donc échouer. La branche $x = 2$ conduit de manière similaire à un échec et l'inconsistance du CSP est alors prouvée en ayant visité deux nœuds, comme l'illustre l'arbre gauche de la figure 3.5. Le résultat est identique si on comme par instancier y ou z.

En revanche, si on commence à instancier t, aucun autre domaine n'est réduit et la démonstration de l'échec décrite précédemment doit être effectuée. Il faudra alors prouver cet échec pour chacune des 99 autres valeurs possibles de t avant d'obtenir l'inconsistance

[13] Si on recherche toutes les solutions d'un CSP, toutes les valeurs doivent être systématiquement essayées. Cependant, l'ordre de sélection des valeurs peut intervenir sur l'efficacité des techniques de *nogoods recording*.

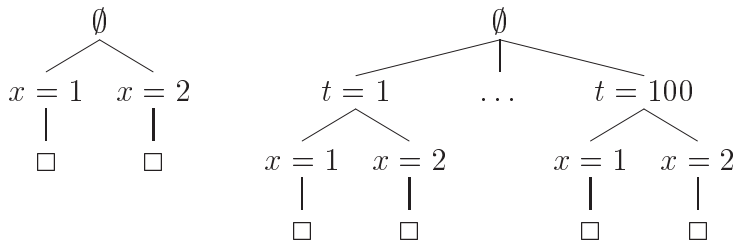


FIG. 3.5 – L'algorithme MAC avec deux ordres d'instanciation de variables différents sur l'exemple 3.3.

du CSP, ce qui prend 100 fois plus de temps qu'avec tout autre ordre d'instanciation[14] (cf. figure 3.5).

On remarque dans l'exemple précédent qu'aucune contrainte ne porte sur t et donc son instanciation ne permet pas de simplifier le CSP (hormis le domaine de z lui-même), i.e. la propagation des contraintes est sans effet. Une des heuristiques d'instanciation des variables les plus efficaces est de commencer par instancier les variables « les plus contraintes » du CSP pour détecter des échecs plus tôt dans l'arbre de recherche et donc en élaguer de plus grande partie : c'est le *first fail principle*. Plusieurs interprétation de ce principe peuvent être données ; les ordres d'instanciation les plus courants sont de choisir d'abord la variable qui a le plus petit domaine (`dom`[15]) ou qui est impliquée dans le plus grand nombre de contraintes. Ces stratégies peuvent être *statiques*, i.e. l'ordre d'instanciation est identique pour toutes les branches de l'arbre, ou *dynamiques*, i.e. que le critère de sélection dépend de l'état de la recherche (i.e. du nœud courant) et l'ordre pourra donc différer suivant la branche parcourue.

Les stratégies génériques les plus efficaces sont en général dynamiques (*Dynamic Variable Ordering*, DVO) et dotées de plusieurs critères pour départager les ex æquo : par exemple choisir la variable de plus petit domaine *et* en cas d'égalité, celle qui correspond au nœud de plus grand degré dans le graphe de contraintes original [Frost 95] ou encore celle qui est liée par une contrainte avec le plus grand nombre de variables non instanciées (heuristique de BRÉLAZ [Smith 99a], conçue à l'origine pour le coloriage de graphes). De nombreuses variantes sont possibles et des critères sophistiqués intégrant de l'expertise peuvent être efficaces pour des problèmes particulier. Cependant, il faut veiller à l'efficacité du maintien de ces critères pour les stratégies dynamiques, par exemple avec des techniques incrémentales (voir section 4.3.4).

Enfin, certains algorithmes tentent de transformer un CSP quelconque vers une classe de problèmes pour laquelle on connaît des algorithmes polynomiaux. Par exemple, un CSP dont le graphe associé est un arbre (i.e. il ne contient pas de cycle) peut être résolu sans *backtrack* grâce à l'établissement de l'arc-consistance et à un ordonnancement des variables adéquat (*min width ordering*). La stratégie d'instanciation consiste alors à

[14] Sauf si on utilise des techniques d'apprentissage de contraintes par exemple.
[15] Cette stratégie est parfois appelée *Minimum Remaining Values* (MRV).

trouver un ensemble de nœuds (le plus petit possible) dont l'élimination du graphe de contraintes supprime tous les cycles[16] et à instancier les variables correspondantes en premier [Dechter 86]. Si une instanciation partielle consistante est trouvée pour cet ensemble, une solution peut être trouvée par MAC sans retour arrière.

Ordre de sélection des valeurs

La stratégie de sélection des valeurs suscite moins d'intérêt et de travaux que celle des variables, car pour montrer qu'un nœud de l'arbre de recherche mène à un échec, il faut en explorer toutes les branches, donc toutes les valeurs possibles pour la variable à instancier. L'ordonnancement des valeurs est donc indifférent pour prouver l'inconsistance d'un CSP ou toutes ses solutions. Cependant, il peut avoir une influence importante sur l'efficacité de la recherche d'une seule solution : si une heuristique parfaite pouvait être utilisée, on pourrait trouver une solution sans retour arrière.

Le principe des stratégie de sélection de valeurs (*Look-ahead Value Ordering*, LVO) est en général de choisir celle qui a le plus de chance de mener à une solution. En effet, comme on l'a remarqué précédemment, l'ordre est indifférent si un échec survient, donc on peut choisir la valeur qui a le plus de support avec les variables non instanciées[17] ou le moins de valeurs en conflit [Frost 95] (stratégie *min-conflicts*). Ces heuristiques peuvent être, comme celles de sélection de variables, statiques ou dynamiques. Les mieux informées sont bien sûr les dynamiques car elles exploitent l'état de la recherche mais peuvent être trop coûteuses à maintenir. Enfin, encore plus que celles des variables, ces stratégies peuvent bénéficier de l'expertise dans un domaine particulier de CSP.

3.1.6 Choix de la combinaison de techniques

Différentes améliorations de l'algorithme *backtrack* ont été présentées dans les trois sections précédentes, classées suivant l'étape de l'algorithme dans laquelle elles interviennent. De nombreuses combinaisons sont possibles qui laissent le choix :

1. De la *puissance du filtrage* : ce sont les techniques de *look-ahead* destinées à simplifier le problème *avant* de prendre une décision non-déterministe, c'est-à-dire de développer un nœud de l'arbre de recherche. Elles utilisent directement la structure du CSP pour inférer des contraintes induites.
2. Du mécanisme de *retour arrière* : ce sont les techniques de *look-back* qui remettent en cause les choix menant à un échec en indiquant une variable instanciée *pertinente* dont la valeur doit changer. Nous avons également classé le *nogoods recording* dans cette catégorie bien qu'il ne soit pas conçu pour désigner la variable à reconsidérer mais à conserver des contraintes induites par un état de la recherche dans des branches parallèles, car il utilise des techniques communes en exploitant les échecs lors de la recherche plutôt que la structure du CSP lui-même.

[16] Un tel ensemble est appelé *cycle cutset*.

[17] Remarquons que ce critère est déjà maintenu par AC-4.

3. De la *stratégie d'instanciation des variables* : elle doit indiquer avec quelle variable non-instanciée étendre la solution partielle courante. Ce sont des *heuristiques* guidées le plus souvent par le *first fail principle* afin de résoudre les parties les plus « contraintes » du problème en premier.
4. De la *tactique de sélection des valeurs* : après avoir choisi la variable à instancier, il faut décider avec quelle valeur le faire. Cette étape correspond à la phase non-déterministe de la recherche et son importance est considérée moindre en général car, pour un CSP sans solution par exemple, il faut parcourir toute la largeur de l'arbre avant de prouver qu'il est insatisfiable. De même pour chaque nœud correspondant à un échec, donc les heuristiques utilisées tentent de choisir une valeur qui mènera le plus probablement à une solution.

Le choix des différents composants est un compromis entre l'effort calculatoire nécessaire pour maintenir les structures de données qu'ils utilisent et le gain qu'ils apportent à la recherche en réduisant le nombre de nœuds explorés ou de vérifications de contrainte. De nombreux travaux de synthèse ont été publiés pour comparer expérimentalement les différents hybrides de BT et identifier les algorithmes les plus performants. La palme du meilleur algorithme reste un sujet de recherche actif et polémique qui divise les communautés scientifiques CSP et PPC.

En effet, [Kumar 92] (qui rapportent les résultats d'autres études de la communauté CSP) et [Bacchus 95, Dechter 98] estiment que le degré de filtrage doit être limité et désigne FC comme le meilleur compromis, notamment lorsqu'il est combiné avec CBJ ou une DVO. Mais les problèmes testés pour établir ces résultats étaient souvent des problèmes relativement faciles comme celui des n-reines (cf. section 2.2), du « zèbre[18] » (p.ex. [Prosser 93]) ou des problèmes aléatoires de petite taille.

Au contraire, [Sabin 94] ou [Bessière 96], d'obédience PPC, montrent que ces résultats doivent être nuancés et que MAC (intégré dans un BT chronologique standard) associé à une DVO (et éventuellement une LVO) surpasse largement FC-CBJ sur les problèmes (aléatoires) les plus « difficiles » (cf. [Smith 95]), notamment quand la taille des domaines augmente et que la densité de contraintes est faible. De plus, si FC est largement plus efficace quand on lui adjoint du *backjumping*, [Bessière 96] met en évidence son peu d'intérêt dans le cas de MAC utilisé avec une bonne DVO : l'ordonnancement des variables selon la taille croissante de leur domaine conduit en général à considérer en premier les variables ayant le plus de conflits avec les variables déjà instanciées, de telle sorte que BJ (et même CBJ) n'effectue pas de grand saut en arrière et qu'un *backtrack* chronologique remonte souvent à la variable pertinente.

De manière générale, l'utilisation d'heuristiques *dynamiques* (DVO) surpassent largement celles qui sont statiques (SVO), bien qu'elles soient plus coûteuses à maintenir. Un consensus désigne la stratégie qui sélectionne d'abord la variable de plus petit domaine (`dom`) comme la base d'une DVO efficace [Bacchus 95]. La plupart des études destinées à comparer ces heuristiques concluent également que les stratégies les plus performantes ajoutent à `dom` des critères exploitant la topologie du graphe de contraintes

[18] Le problème du zèbre est un puzzle logique attribué à Lewis Carroll.

[Bessière 96, Smith 99a]. D'autre part, les heuristiques efficaces de sélection de valeurs (LVO), beaucoup moins étudiées et utilisées, exploitent essentiellement la stratégie *min-conflicts* [Frost 95, Bessière 96] comme gradient indiquant les valuations les plus prometteuses.

Pour améliorer la compréhension de ces nombreuses variantes des études théoriques ont également été menées. [Kondrak 95] prouve ainsi la correction de BJ, CBJ, BM, FC et leurs hybrides, et les hiérarchise suivant l'ensemble de nœuds visités et le nombre de vérifications de contraintes ; cependant, l'effet des DVO, dont l'influence est largement reconnue, n'est pas pris en compte. Dans un cadre plus fédérateur et tourné vers l'implémentation, [Bacchus 00] se fonde sur la notion de *nogoods* et caractérise les différents hybrides suivant la manière dont ils les découvrent, les utilisent et les maintiennent.

La comparaison des divers algorithmes de *backtrack* est donc un sujet délicat. Un décalage existe entre la communauté PPC dont les outils, qui ont vocation à être utilisés pour résoudre des problèmes industriels strucurés, privilégient les techniques de consistance en choisissant systématiquement MAC (au sein d'un *backtrack* chronologique) combiné avec des stratégies d'instanciation qui exploitent la structure du problème, et la communauté CSP qui établit souvent ses résultats sur des problèmes aléatoires ou académiques en favorisant FC et les *look-back schemes* (notamment FC-CBJ). Quoiqu'il en soit, il est maintenant admis que toutes les améliorations de BT ne sont pas orthogonales et que leurs effets ne s'aditionnent pas simplement, lorsqu'ils ne sont pas inutilement redondants (par exemple, CBJ et `dom` en utilisant FC ou surtout MAC). D'autre part, le compromis idéal entre recherche et inférence dépend des caractéristiques du problème (nombre de variables, taille des domaines, topologie du graphe de contraintes, densité, dureté...), et, de manière duale, la notion de problèmes *exceptionnellement* difficiles est relative à l'algorithme utilisé [Smith 95].

3.1.7 Programmation Par Contraintes

Pour résoudre un CSP binaire dont les contraintes sont exprimées en extension, on peut implémenter l'un des algorithmes décrits précédemment avec un langage de programmation (ou s'en procurer une version existante) en représentant le problème à l'aide de structures de données adéquates — par exemple des listes de couples pour les contraintes et des listes simples pour les domaines. Il faut ensuite indiquer l'ordre de sélection des variables et celui des valeurs ; si l'implémentation a prévu plusieurs heuristiques, on pourra en sélectionner une en passant un paramètre à l'algorithme. Cette approche est celle qui est (était ?) utilisée le plus souvent pour résoudre des problèmes en programmation mathématique (linéaire, non-linéaire, mixte...).

Mais la modélisation d'applications industrielles complexes dans un cadre si strict dénature souvent la structure du problème, voire la rend impossible à exprimer. En effet, la spécification de tels problèmes est en générale plus « directe » et rapide à concevoir avec des contraintes en *intention*, souvent d'arité quelconque, qu'il est parfois impossible d'exprimer en extension (cf. section 2.1.1 et 2.1.5). Il faudrait alors enrichir considérablement la variété des structures de données utilisées par notre algorithme pour pouvoir

représenter de tels CSP. Comme on connaît en général pour ces contraintes en intention des algorithmes spécifiques et efficaces de consistance locale qui exploitent leur sémantique particulière, il faudrait également pouvoir intégrer à notre algorithme des procédures spécifiques à chaque contrainte. Finalement, ces problèmes réels peuvent souvent profiter de l'expertise liée à leur domaine pour guider la recherche de solution avec des heuristiques *ad hoc* ; leur implémentation nécessiterait de modifier le code de notre algorithme à chaque fois.

Pour réduire le temps de développement des applications de programmation mathématique en préservant le plus possible l'efficacité d'implémentations *ad hoc*, des langages de modélisation tel que AMPL [Fourer 93] ont été conçus. Il s'agit d'une couche d'abstraction destinée à séparer les algorithmes de résolution de la spécification du problème. Les problèmes sont alors représentés par des termes d'un *langage* (donc des *programmes*) dédié à la spécification concise et souple (générique) de leur structure. Un solveur (intégré ou indépendant du système) se charge ensuite de résoudre le résultat de la compilation de ces programmes. Ces langages sont devenus d'autant plus intéressants que les techniques de résolution élargissaient le cadre des problèmes qu'elles pouvaient résoudre (e.g. la programmation mixte) et nécessitaient plus d'expressivité pour décrire les problèmes.

Les CSP sur les domaines finis sont un cadre très souple qui permet d'exprimer des problèmes d'optimisation combinatoire dans des domaines divers. Notamment dans le cadre d'applications réelles, un tel langage de modélisation semble nécessaire pour disposer d'outils assez flexibles et expressifs permettant de développer rapidement et efficacement des programmes de résolution de CSP aussi variés. La Programmation Par Contraintes (PPC) répond, entre autres, à ce besoin. Elle constitue un formalisme générique et cohérent qui permet de spécifier un CSP et une stratégie de recherche puis de résoudre le problème. La PPC sépare clairement ces deux étapes (spécification et résolution) par une architecture à deux niveaux :

1. un « solveur » de contraintes (*constraint store*) (en général incomplet) qui maintient un ensemble de contraintes (et de variables) et permet d'inférer des propriétés du système grâce à des opérations basiques comme l'ajout de contraintes, le test de satisfaction et la propagation (contraintes induites) ;
2. un *langage* (de programmation) qui permet de spécifier déclarativement les variables et les contraintes du problème ainsi que le contrôle du *constraint store* à l'aide de termes logiques (*buts* de recherche non-déterministes).

Cette structure résulte en un paradigme générique (vis-à-vis du *constraint store* dont les opérations sont abstraites) pour générer, manipuler et vérifier des contraintes de manière très expressive tout en profitant d'algorithmes efficaces pour différents types de contraintes et de domaines. Les systèmes de PPC permettent à l'utilisateur une modélisation souple et rapide qu'il est très simple de raffiner et de modifier, ainsi qu'un contrôle de haut niveau de la recherche de solutions. Ils permettent également à leur concepteurs d'intégrer facilement de nouveaux algorithmes dédiés à des contraintes spécifiques. Enfin, la généricité de l'approche PPC se concrétise même par la possibilité pour les *utilisateurs* d'enrichir eux-mêmes la collection de contraintes primitives au sein de certains systèmes en fournissant au sein

du langage des constructions pour abstraire le fonctionnement interne du *constraint store* ; cette « extensibilité » ouvre des perspectives très vastes quant aux domaines d'application de la PPC.

L'architecture particulière de la PPC est héritée des systèmes de programmation logique pour laquelle les programmes sont des ensembles de clauses logiques et leur exécution une recherche de preuves (ou « solutions ») non-déterministe contrôlée par *backtrack*. Les systèmes Prolog peuvent en fait être vus comme des solveurs de contraintes d'égalité (syntaxique) sur un domaine spécifique, les termes rationnels ou finis. Pour en améliorer l'expressivité, ce principe de démonstration automatique a été généralisé, en utilisant la même architecture, à des structure mathématiques plus riches que les termes rationnels. La recherche de preuves utilise alors des techniques hybrides de résolution logique et de résolution de contraintes dans ces structures mathématiques. Cette forme originelle de PPC est appelée la Programmation Logique par Contraintes (PLC) [Jaffar 86]. Le cadre générique de la PLC, paramétré par un domaine de contraintes, est, comme celui de la programmation logique, doté de sémantiques (opérationnelles, logiques, algébriques) cohérentes qui en assurent les fondations [Jaffar 98].

Les premiers systèmes de PPC étaient tous construits à partir d'un système Prolog dont la syntaxe déclarative convient à la spécification de CSP et le contrôle non-déterministe à la recherche systématique par *backtrack*. Mais certains aspects du développement d'applications industrielles comme la modularité, le traitement des données, l'interface avec d'autres systèmes ou l'écriture d'algorithmes impératifs spécifiques de propagation de contraintes sont parfois des tâches difficiles en programmation logique et le résultat manque parfois d'efficacité — les systèmes de PLC utilisent d'ailleurs souvent des routines de bas niveau codées en C. L'architecture de la PPC est en fait également adaptée à la conception de systèmes fondés sur d'autres types de langages hôtes comme C++ (ILOG Solver) [Puget 94] ou Lisp (PECOS) [Puget 92]. Ces systèmes, présentés sous la forme d'une librairie, introduisent alors des constructions spécifiques pour retenir la syntaxe déclarative de Prolog lors de la spécification des contraintes et fournir des structures de contrôle non-déterministes pour la recherche de solutions. Enfin, certains systèmes sont eux-mêmes des compilateurs (éventuellement vers un autre langage cible) pour des langages qui étendent l'architecture classique de la PLC, comme la PLC *concurrente* [Saraswat 93] implémentée dans le système Mozart [Mozart 02].

La PPC apparaît maintenant comme une plate-forme cohérente de modélisation et de résolution de problèmes d'optimisation combinatoire qui permet d'intégrer élégamment des algorithmes très divers de l'Intelligence Artificielle et de la Recherche Opérationnelle, tout en rendant la spécification du problème et le contrôle de la recherche abstraits grâce à un langage déclaratif. Eugene FREUDER [Freuder 96] résume avec enthousiasme le processus de résolution que propose la PPC (traduction[19] libre) :

> *La programmation par contraintes constitue l'un des paradigmes les plus proches du Saint Graal de la programmation que l'informatique ait jamais réalisé : l'uti-*

[19] « Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it. »

lisateur spécifie le problème, et l'ordinateur le résout.

Domaines

La PPC est un paradigme générique qui permet d'appliquer la même technologie à différents domaines, c'est-à-dire des structures mathématiques dotées d'une théorie équationnelle. Dans le cadre de la Recherche Opérationnelle, les problèmes considérés peuvent être continus, discrets ou mixtes. Pour résoudre ces problèmes, les systèmes de PPC existants implémentent des solveurs sur différents types de domaines.

Le plus important dans le cadre de l'optimisation combinatoire est celui des domaines finis dans lequel les variables prennent leur valeur dans des sous-ensembles d'un intervalle des entiers. La plupart des systèmes actuels propose ce type de contraintes et nous avons déjà détaillé une partie des techniques de propagation, notamment dans le cas des contraintes en extension. C'est cette instance de la PPC que nous utiliserons dans la suite mais la plupart des concepts présentés peuvent se généraliser aux autres domaines.

L'autre domaine prépondérant est celui des réels, représentés par des intervalles de nombre flottants (ou des rationnels en précision arbitraire). Pour les problèmes linéaires, des algorithmes complets, tel le Simplex, sont en général utilisés. Pour les problèmes non-linéaires, des techniques algébriques peuvent être utilisés dans certains cas (polynômes) mais en général des méthodes plus robustes sont choisies comme l'arithmétique des intervalles pour propager les contraintes, combinées avec des algorithmes de recherche [Sam-Haroud 95]. Enfin, certains solveurs attendent pour les propager que les équations deviennent linéaires grâce à l'instanciation d'une partie de leurs variables.

D'autres domaines moins bien représentés dans les systèmes courants ont également été implémentés (hormis les termes rationnels qui sont le domaine originel de la programmation logique et donc disponibles dans tous les systèmes Prolog) :

- Les booléens, qui sont souvent traités comme un cas particulier des domaines finis.
- Les ensembles finis (d'entiers) dont les contraintes sont propagées par des algorithmes [Gervet 97] du même type que ceux des domaines finis, mais la représentation des domaines ensemblistes est spécifique (compacte, cf. section 5.3 — une représentation en extension serait exponentielle avec le cardinal). Ces domaines permettent de modéliser plus « naturellement » certains problèmes combinatoire, et notamment d'éviter d'introduire des symétries de permutations (e.g. [Barnier 02b] sur le problème des écolières de Kirkman).
- Il existe dans certains systèmes de PPC des contraintes sur des domaines plus exotiques (e.g. les listes dans Prolog IV).

Les systèmes de PLC fondés sur Prolog (ainsi que les compilateurs d'extensions de Prolog) utilisent en général des constructions syntaxiques spécifiques pour déclarer des variables à domaines alors que les librairies pour des langages non logiques offrent des fonctions ou des classes. Par exemple, les expressions suivantes :

ECL^{i}PS^e	ILOG Solver	FaCiLe
`D :: 0..9`	`IlcIntVar D(m, 0, 9)`	`let d = Fd.interval 0 9`

créent une variable à domaine fini de domaine $[0, 9]$. Les domaines finis sont en général représentés pour le solveur soit en extension avec un tableau de booléens (bits), soit avec un ensemble (e.g. une liste) d'intervalles, soit de manière plus compacte en ne conservant que les bornes (les « trous » peuvent alors être exprimées par des contraintes unaires).

Primitives et langage de contraintes

Outre la possibilité d'exprimer une contrainte en extension (dans le cas des domaines finis), les langages des systèmes PPC fournissent sur chaque domaine un ensemble de contraintes primitives — certaines d'entre elles sont mixtes, e.g. la contrainte de cardinalité entre une variable entière et une variable ensembliste. Pour les domaines finis, les solveurs disposent en général de contraintes arithmétiques linéaires et non-linéaires avec les opérateurs classiques :

$$+,\ -,\ \times,\ /,\ \texttt{mod},\ \texttt{abs}$$

qui peuvent se combiner en des expressions de complexité arbitraire pour spécifier des relations arithmétiques :

$$=,\ \neq,\ \leq,\ \geq,\ <,\ >$$

Une contrainte globale spécifique se charge souvent d'améliorer la propagation des expressions linéaires et les termes non-linéaires sont remplacés par des variables intermédiaires (voir section 5.5.6). Les contraintes sont ainsi « normalisées » pour s'adapter au jeu de contraintes primitives du solveur.

Des contraintes *symboliques* qui permettent une grande souplesse de modélisation ont été introduites précocement dans les systèmes de PPC :

- indexation d'un « tableau » de variables par une variable — souvent utilisée pour modéliser la quantification existentielle sur une variable ($\exists x \in \mathcal{X}' \subseteq \mathcal{X}$ telle que...) ;
- élément minimum ou maximum d'un tableau de variables ainsi que son indice.

Des contraintes globales correspondant à des sous-problèmes récurrents en optimisation combinatoire sont proposées par certains solveurs. Ce sont des contraintes génériques qui portent sur un nombre quelconque de variables (comme la contrainte de somme généralisée mentionnée précédemment) et qui permettent des propagations et des tests de consistance efficaces en raisonnant *globalement* sur le sous-problème. Elles font appel à des techniques *ad hoc* issues de la RO et adaptées au caractère incrémental de la recherche constructive. Par exemple, la contrainte très populaire « tous différents » sur k variables est équivalente à $\frac{k(k-1)}{2}$ diséquations binaires mais certaines implémentations sont beaucoup plus performantes (voir section 5.5.7) car elles sont complètes : elles utilisent un algorithme polynomial de mariage (*matching*) maximal dans un graphe bipartie pour en construire une solution.

Les contraintes des systèmes de PPC peuvent être *réifiées*, c'est-à-dire transformées en nouvelles variables booléennes qui indiquent si elles sont satisfaites ou violées. En effet, il est possible de déterminer pour certaines contraintes (essentiellement arithmétiques) si elle est induite par le CSP (ou violée) avant l'instanciation de ses variables. Ces contraintes réifiées peuvent alors se combiner avec des opérateurs logiques ou des relations arithmétiques pour produire des *méta-contraintes* (l'étape de réification étant éventuellement transparente pour l'utilisateur) :

$$\vee,\ \wedge,\ \neg,\ \oplus,\ \Rightarrow,\ \Leftrightarrow$$

Cette possibilité ajoute considérablement à l'expressivité des systèmes de PPC (on peut par exemple optimiser la somme de contraintes réifiées pour traiter les CSP partiels 2.3.1) ainsi qu'à leur efficacité (disjonction constructive).

Les contraintes primitives (d'arité non fixée dans le cas des contraintes globales) associées à leurs variables peuvent donc être vues comme les atomes d'un langage dont les termes se construisent par induction à l'aide d'opérateurs logiques sur les réifications. Quand ce langage ne suffit plus pour modéliser un problème particulier, certains systèmes permettent à l'utilisateur d'écrire ses propres contraintes dotées d'algorithmes de propagation spécifiques

Propagation sur les domaines finis

Les contraintes arithmétiques des systèmes de PPC sur les domaines finis ne sont pas, en général, équipées d'algorithmes de propagation assurant l'arc-consistance car elle est trop coûteuse : les domaines peuvent être de grande taille et donc longs à parcourir exhaustivement, et s'ils correspondent à des intervalles (comme c'est souvent le cas au début de la recherche), les contraintes linéaires sont arc-consistantes en ne modifiant que leurs bornes, opération qui peut se calculer en temps constant (ou linéaire pour les sommes généralisées). Des approximations de l'arc-consistance sont donc plus « rentables », comme la $2B$-consistance [Lhomme 93] qui caractérise l'arc-consistance limitée aux bornes d'un intervalle.

Les autres contraintes assurent des degrés de consistance local variés suivant la complexité des algorithmes capables de les réaliser, et certains systèmes de PPC permettent même de les paramétrer pour les adapter à des problèmes particuliers pour trouver le meilleur compromis entre la réduction de l'espace de recherche et le coût des calculs.

Ces propagations sont presque systématiquement intégrées au sein d'un algorithme de type MAC, qui n'assure pas exactement l'arc-consistance mais un degré de consistance locale composite suivant le type de chaque contrainte, en réexaminant les contraintes dès qu'une de leurs variables a été modifiée — seulement si les *bornes* ont été modifiées dans le cas de la $2B$-consistance — jusqu'au point fixe ou jusqu'à ce qu'un domaine soit vide. Cette algorithme est piloté par un *backtrack* chronologique simple.

Buts de recherche

Les systèmes de PPC permettent de contrôler la recherche de solutions avec un langage de *buts* déclaratif hérité de la programmation logique et qui spécifie des structures de contrôle non-déterministes grâce à des disjonctions ou points de choix. Dans le cas de la PLC, ce langage est donc le langage hôte lui-même, mais les librairies de PPC destinées à des langages déterministes doivent fournir des constructions ou des fonctions spécifiques pour l'implémenter ainsi qu'un environnement d'exécution (une pile).

Dans le cadre de la PPC, les buts de recherche sont construits avec des buts primitifs spécifiques du système, telle que l'instanciation non-déterministe d'une variable, des buts définis par l'utilisateur et les connecteurs logiques $\wedge$ qui représente opérationnellement une séquence de buts et $\vee$ qui créent un point de choix.

Dans les applications réelles, les points de choix traditionnels de BT qui consistent à instancier une variable sont trop limités pour concevoir des heuristiques guidées par l'expertise du domaine. Pour résoudre un problème de *scheduling* par exemple, il est inefficace d'essayer successivement chaque valeur des variables temporelles. On préférerait simplement ordonnancer ces tâches (un décision en rapport avec la sémantique du problème) ou explorer les domaines de manière dichotomique. La PPC généralise ainsi les nœuds de BT en leur associant des décisions sous la forme de contraintes ajoutées aux CSP qui ne sont pas limitées au choix de la valeur d'une variable. En outre, les buts peuvent s'écrire récursivement (ce qui rend les heuristiques dynamiques très concises) e.g. :

$$dichotomic(start) :- \\ (start \leq \frac{start_{\min}+start_{\max}}{2} \ \vee \ start > \frac{start_{\min}+start_{\max}}{2}) \vee dichotomic(start).$$

La recherche coïncide alors avec la résolution d'un terme du langage des buts exécuté en profondeur d'abord par défaut, comme dans les systèmes Prolog. Si une inconsistance est détectée, le but courant échoue et on revient sur la dernière alternative.

3.2 Recherche locale

3.2.1 Principe général

Alors que la plupart des études sur la résolution des CSP se concentraient sur les améliorations des techniques d'inférence et de recherche systématique par *backtracking*, des algorithmes de *Recherche Locale* (RL) ou *recherche stochastique* particulièrement efficaces sur des instances « difficiles » (notamment du problème SAT, cf. section 2.1.3) ont été proposés au début des années 90[20] [Selman 92, Minton 94]. Les algorithmes systématiques se heurtent à leur complexité exponentielle en pire cas, contrepartie de leur complétude, quand les problèmes sont de trop grande taille : si on fixe une limite de temps de calcul, l'algorithme peut terminer sans rendre de résultat même si le problème est consistant. Pour

[20] Cependant, des algorithmes de RL sont utilisés en RO depuis les années 60 [Lin 65].

résoudre ces problèmes qui résistent aux algorithmes systématiques, on choisit d'abandonner l'exactitude de la recherche pour pouvoir parcourir plus librement l'espace de recherche en étant guidé par une méta-heuristique, c'est-à-dire une heuristique générique d'optimisation inspirée d'une métaphore (par exemple géographique, *hill climbing* ou biologique, *algorithmes évolutionnistes*). Ces techniques de RL sont initialisées avec une instanciation totale éventuellement non-consistante puis la modifient itérativement en suivant des gradients locaux pour se diriger vers une solution consistante ou un optimum :

1. Un *voisinage* est défini autour de l'instanciation totale courante : il représente l'ensemble des instanciations totales obtenues en altérant « localement » la solution courante, par exemple en changeant la valeur de l'une de ses variables.
2. Une *heuristique de sélection* indique l'un des voisins qui devient la solution courante, par exemple celui qui permet de faire diminuer le plus le nombre de contraintes violées ou le coût de la solution (pour un problème de minimisation).
3. Si une instanciation totale consistante est obtenue, la solution est renvoyée. Sinon (ou si on cherche un optimum) un *critère d'arrêt* est testé pour limiter le temps de calcul, par exemple un nombre d'itérations maximal.

Pour un voisinage donné, si on se contente de choisir systématiquement le voisin qui réduit le plus le nombre de contraintes violées (qu'on appellera *coût* de la solution), on peut se retrouver bloqué dans un optimum local dont tous les voisins dégradent le coût. Une stratégie est donc utilisée pour s'en « échapper », par exemple en autorisant des transitions défavorables (i.e. choix d'un voisin de coût supérieur) ou en recommençant la recherche avec une nouvelle solution choisie aléatoirement ; certains algorithmes utilisent également les solutions précédentes (*recherche taboue*) ou maintiennent un ensemble de solutions (*algorithmes génétiques*) pour informer leur stratégie.

Les algorithmes de RL ne sont pas complets car ils ne maintiennent pas de représentation de la partie *déjà* explorée de l'espace de recherche — les algorithmes systématiques le font en structurant l'espace sous la forme d'un arbre — et donc ne peuvent garantir que tout l'espace de recherche sera exploré ni qu'une même solution ne sera pas visitée plusieurs fois. Leur efficacité est donc empirique et dépend de l'instance considérée, mais ils ont l'avantage de pouvoir résoudre les PCSP (problèmes sur-contraints, cf. section 2.3.1) car ils manipulent des solutions éventuellement non-consistantes, et fournir une solution (éventuellement non-consistante) à tout moment (algorithme *anytime*) car l'état courant est une instanciation *totale*.

L'algorithme 5 présente un cadre générique qui schématise la plupart des algorithmes de RL. nb_essais recherches successives sont tentées tant qu'une condition globale sur les solutions déjà trouvées est respectée. Chaque recherche consiste en $nb\_iterations$ itérations d'un déplacement local dans le voisinage de la solution courante s tant qu'une condition « locale » à cette recherche est satisfaite (e.g. la recherche peut être arrêtée si l'évaluation de la solution ne s'améliore plus pendant un certain nombre d'itérations). À chaque étape, on teste si s est consistante, auquel cas la solution est renvoyée. Sinon un voisin v de s est choisi et si v respecte un certain critère, par exemple v n'a pas été considéré depuis un certain nombre d'itérations (*recherche taboue*), il remplace la solution courante. Au

bout de *nb_iterations* itérations, une nouvelle solution initiale est générée et la recherche recommence — on peut utiliser par exemple un algorithme glouton pour construire les solutions initiales. S'il s'agit d'un problème d'optimisation, il suffit de modifier l'algorithme pour mémoriser la solution courante si son coût est inférieur à celui de la meilleure solution trouvée au lieu de la renvoyer (lignes 4 et 5).

Algorithme 5 – RL

```
1:  s := initialise()
2:  for i = 1 to nb_essais while condition_globale do
3:     for j = 1 to nb_iterations while condition_locale do
4:        if consistante(s) then
5:           return s
6:        end if
7:        v := choisit(voisinage(s))
8:        if acceptable(v) then
9:           s := v
10:       end if
11:    end for
12:    s := reinitialise(s)
13: end for
14: return s
```

3.2.2 Guider la recherche

Différentes stratégies sont utilisées pour guider la recherche vers des solutions qui violent le moins possible de contraintes. Elles définissent le voisinage de la solution courante, en général l'ensemble des instanciations qui diffèrent de la solution courante par la valeur d'une unique variable (la taille de cet ensemble est $n(d-1)$ si les domaines ont même cardinal d), et la manière de sélectionner l'un des voisins :

- *Hill climbing* : le voisin qui minimisent le nombre de contraintes violées est choisi. C'est par exemple l'heuristique de GSAT [Selman 92] destinée à résoudre des problèmes SAT (cf. section 2.1.3) — dans ce cas, la taille du voisinage est n.
- *Min-conflicts* [Minton 94] : pour éviter d'avoir à explorer exhaustivement le voisinage comme avec le *hill climbing*, une variable en *conflit* (i.e. impliquée dans une contrainte violée) est choisie aléatoirement et instanciée avec la valeur qui minimise le nombre de conflits. Cette heuristique tente ainsi de « réparer » une instanciation non-consistante.
- *Algorithmes génétiques* (AG) [Goldberg 89] : contrairement aux autres algorithmes de RL, les AG maintiennent une « population » de solutions et utilisent un opérateur de « recombinaison ». Cet opérateur (le *croisement*) construit une nouvelle instanciation (ou deux) à partir de deux solutions de la population en prenant une partie des valeurs de l'une et le reste dans l'autre, ce qui correspond à la métaphore géné-

tique de la reproduction sexuée ; cette descendance a éventuellement une meilleure évaluation que leurs géniteurs. Ce type de transitions « non locales » et l'utilisation simultanée d'un ensemble de solutions fait sortir les AG du cadre de la RL, bien qu'ils en soient une extension. Notons que, de manière analogue au croisement d'un AG, [Selman 93] propose de combiner les solutions des deux dernières tentatives de GSAT (en conservant les valeurs communes) pour obtenir une meilleure réinitialisation (cf. ligne 12 de l'algorithme 5) qu'une instanciation aléatoire.

Bien d'autres techniques sont envisageables comme l'exploration de voisinages plus grands (où les valeurs de plusieurs variables sont changées) ou plus « sémantiques » (comme le voisinage k-opt pour le TSP [Lin 73]).

3.2.3 S'échapper des minima locaux

Si le meilleur voisin est systématiquement sélectionné, un algorithme de RL risque (fortement) de se retrouver bloquée avec une solution courante dont tout le voisinage dégrade le coût, i.e. un optimum local. De nombreuses techniques ont été proposées pour éviter ce comportement en acceptant des transitions défavorables avec une certaine probabilité, ce qui confère à la RL son caractère « stochastique » :

- *Multi-start* : comme le suggère l'algorithme 5, la recherche redémarre à partir d'une nouvelle solution initiale (e.g. générée aléatoirement) une fois qu'un certain nombre de transitions a été réalisé.
- *Random walk* : la définition du voisinage et/ou la sélection de la transition sont « randomisées » pour accepter des voisins qui n'améliorent pas le coût de la solution courante. Par exemple, [Selman 94] présente une variante de GSAT dans laquelle la transition standard est appliquée avec une certaine probabilité p, sinon une variable en conflit est choisie aléatoirement et sa valeur changée, même si cette transition dégrade le coût de la solution courante. Une méta-heuristique très populaire, le *recuit simulé* [Kirkpatrick 83], utilise une technique similaire en choisissant un voisin aléatoirement puis en acceptant systématiquement la transition si elle est favorable et avec une certaine probabilité (dépendante d'un paramètre global appelé *température*, par analogie avec le procédé métallurgique de *recuit*) si elle est défavorable.
- *Algorithmes génétiques* : les AG gardent leur population de solutions diversifiée pour éviter d'être bloqués dans un optimum local. Les solutions sont gardées d'une génération à l'autre en les sélectionnant avec une probabilité inversement proportionnelle à leur coût (de nombreuses variantes existent [Alliot 96]).
- *Recherche taboue* [Glover 89] : pour empêcher la recherche d'effectuer des séquences de transitions qui « bouclent » et de rester bloquée dans un optimum local, une liste de longueur fixée des dernières solutions visitées est maintenue et un voisin n'est accepté que s'il n'en fait pas partie. Dans HSAT, une variante de GSAT proposée par [Gent 93], une « mémoire » est également utilisée, mais pour éviter de modifier à nouveau une variable qui l'a été récemment entre des voisins ex æquo.
- *Pondération des contraintes* : en constatant que, sur certains problèmes, différentes tentatives de GSAT se concluaient par les mêmes clauses non satisfaites, [Selman 93]

modifie l'algorithme pour donner plus d'importance, lors de la sélection du meilleur voisin, à ces clauses. Ainsi, après chaque nouvelle tentative, le poids (initialisé à 1) des clauses non satisfaites est incrémenté pour accentuer leur importance dans le nombre de conflits associé à un voisin. Cette technique est également à la base de la méta-heuristique *Guided Local Search* (GLS) [Tsang 99] et utilisée par [Eiben 95] au sein d'un AG. On peut la considérer comme l'apprentissage ou l'adaptation automatique d'une heuristique qui indique la difficulté de résoudre chaque clause.

Là encore, la variété de ces techniques heuristiques, dont seules les principales sont recensées ici, n'a de limite que l'imagination de leur auteurs. Par ailleurs, la plupart d'entre elles nécessitent le réglage de nouveaux paramètres qui viennent s'ajouter à la liste, souvent déjà longue, de ceux qui contrôlent le fonctionnement d'un algorithme de RL.

3.2.4 Problèmes d'optimisation

Les méthodes de RL considèrent un CSP (standard) comme un problème d'optimisation en calculant le coût d'une solution à partir du nombre de contraintes violées (cf. problème Max CSP, section 2.1.7). Cependant, quand on souhaite trouver la solution consistante optimale d'un CSP vis-à-vis d'un certain critère, il faut intégrer le degré de satisfaction *et* le critère dans l'évaluation d'une instanciation, ce qui pose des problèmes de convergence similaires à ceux des algorithmes d'optimisation *multi-critères* [Talbi 99]. Au contraire, les algorithmes systématiques qui intègrent un Branch & Bound au sein d'un algorithme *back-track* ne recherchent l'optimum que parmi les instanciations consistantes, mais ne peuvent pas fournir de « solutions » (approchées) si le problème est sur-contraint.

Certaines approches, comme celles des AG de [Eiben 95], divisent l'ensemble des contraintes en deux groupes : le premier contient les contraintes qui doivent être satisfaites par toutes les instanciations manipulées et les opérateurs de l'AG (croisement et mutation qui définissent une notion équivalente au voisinage) doivent être adaptés pour ne générer que des instanciations valides, et le second est constitué des autres contraintes du CSP qui sont incorporées à la fonction de coût. Par exemple, des opérateurs qui préservent le fait qu'une instanciation représente une permutation ont été utilisés au sein d'AG pour résoudre des instances du TSP. Cette contrainte peut donc être rangée dans la première catégorie, mais dans le cas général, de tels opérateurs sont très complexes à réaliser.

3.2.5 « Programmation par recherche locale » ?

Contrairement aux algorithmes systématiques de résolution des CSP, la RL ne jouit pas de solides fondations théoriques[21] et consiste plutôt en une collection de techniques issues de métaphores intuitives. Les études qui ont pour ambition d'unifier les différents types d'algorithmes de RL et d'améliorer leur compréhension ne sont que très récentes. [Yagiura 99] propose d'adopter un cadre générique similaire à celui de l'algorithme 5 (appelé

[21] Des théorèmes de convergence statistique ont été démontrés pour quelques uns des algorithmes de RL, mais sans réelle utilité en pratique (e.g. [Cerf 94] pour les AG).

random multi-start local search) pour comparer les comportement des divers algorithmes et de leurs raffinements. Plus générale et originale, l'approche de [Taillard 98] utilise le concept de *Adaptive Memory Programming* pour fédérer recherche taboue, AG, recuit simulé et même colonie de fourmis ; par exemple, la population d'un AG est ainsi vue comme la « mémoire » de la recherche passée, au même titre qu'une liste taboue. Ce formalisme permet également de décrire et mieux comprendre le fonctionnement d'hybrides de ces méta-heuristiques.

De même, les outils destinés à abstraire les mécanismes élémentaires de la résolution par RL ne sont apparus que très récemment et sont encore loin de l'élégance et de la généricité de la PPC pour modéliser et résoudre les CSP ainsi que pour intégrer les techniques spécifiques à ce type de recherche. Localizer [Michel 97], dont est extrait l'algorithme générique 5, est une des tentatives les plus remarquables en ce sens : un langage de spécification d'algorithmes de RL et de ses structures de données permet d'exprimer de manière souple et concise la résolution d'un CSP, et le système maintient automatiquement et efficacement les structures de données lors de la recherche. Des librairies de classes génériques ont également été développées (LOCAL++ [Schaerf 00], HotSpot [Fink 98], SCOOP [SCOOP 98]) mais avec un niveau d'abstraction moindre que celui de Localizer.

3.3 Hybridations

3.3.1 Les divergences des deux paradigmes sont-elles conciliables ?

Recherche systématique et recherche locale sont deux méthodes très différentes pour résoudre les CSP. La première est complète mais de complexité exponentielle en pire cas alors que la seconde est inexacte mais robuste car elle peut toujours fournir une solution « approchée ». Certains problèmes peuvent donc exclure l'alternative : si une preuve d'inconsistance est nécessaire, la RL ne peut pas convenir, mais si le problème est de trop grande taille et qu'une solution, même partielle, est nécessaire, la recherche par *backtrack* sera trop coûteuse en temps de calcul. D'autres problèmes sont résolus plus ou moins efficacement suivant leur structure par l'une ou l'autre méthode : la PPC par exemple exploite activement les contraintes des CSP, ce qui réduit efficacement l'espace de recherche des problèmes très structurés mais la recherche arborescente rend l'optimisation difficile car l'ordre d'exploration est trop rigide (on ne peut pas se déplacer « horizontalement » dans l'arbre de recherche), alors que la RL est mieux adaptée ; au contraire, la RL est avant tout une technique d'optimisation et peut parcourir plus librement l'espace de recherche, mais des « solutions » de faible coût (pour un problème d'optimisation) peuvent être inutilisables si elles violent certaines contraintes. Ces caractéristiques doivent être prises en compte lors du choix d'un algorithme pour résoudre un CSP, mais, comme l'illustre la vive polémique du *panel* de IJCAI'95 [Freuder 95], il n'y a guère de consensus dans la communauté scientifique pour préconiser une approche plutôt qu'une autre.

Pourtant, comme le suggère Eugene FREUDER en introduction de ce même *panel*[22] :

[22] « Rather than competing, can we cooperate ? »

Au lieu de faire la compétition, peut-on coopérer ?

Il faudrait en effet pouvoir profiter des avantages des deux méthodes pour résoudre efficacement des problèmes ambitieux, très structurés et de grande taille. Mais les difficultés conceptuelles d'algorithmes hybrides sont importantes :

- BT manipule des instanciations *partielles* alors que la RL consiste à modifier successivement des instanciations *totales*. Comment faire cohabiter ces deux types d'états ?
- Les heuristiques de la RL peuvent suivre librement des « gradients locaux » pertinents calculés à partir de l'état courant, mais pour déterminer le meilleur voisin, ils doivent considérer de nombreuses solutions, éventuellement inconsistantes ou plus mauvaises que l'état courant. Au contraire, les heuristiques des algorithmes de BT sont mal informées à proximité de la racine de l'arbre de recherche (quand très peu de variables sont instanciées), mais de nombreux choix sont élagués par inférence, puis efficaces près des feuilles mais la recherche est alors confinée dans une très petite partie de l'espace de recherche. Comment profiter des heuristiques et de la liberté de déplacement de la RL dans une recherche systématique ? Peut-on améliorer l'exploration des voisinages de la RL avec des techniques d'inférence ou considérer des voisinages plus ambitieux ?
- La RL permet d'obtenir une solution, éventuellement approchée, à tout moment alors que la recherche systématique peut être complètement inutilisable si un temps de réponse trop court est nécessaire. Peut-on obtenir des solutions à tout moment au sein d'une recherche systématique et comment relaxer des contraintes pour des instances sur-contraintes ?
- Finalement, comment incorporer les structures de données et de contrôle nécessaires aux deux paradigmes dans un outil commun pour expérimenter des hybridations ?

3.3.2 Tentatives d'unification

Malgré des divergences fondamentales, on peut identifier certaines analogies entre les deux paradigmes.

Par exemple, la recherche systématique peut être vue comme l'exploration d'un voisinage dégénéré qui englobe tout l'espace de recherche, défini par la modification d'un nombre quelconque de variables par rapport à une instanciation totale donnée. On pourrait utiliser une recherche systématique pour explorer de « grands voisinages » (mais plus petits que l'espace de recherche entier) à chaque transition d'un algorithme de RL.

Certains algorithmes de *backtrack* plus flexibles que BT peuvent être vus comme l'utilisation d'une technique de RL sur l'espace des *instanciations partielles arc-consistantes* (voir section 3.3.3). Des restrictions peuvent être nécessaires sur le « voisinage » exploré et la complétude est parfois sacrifiée [Prestwich 01]. On peut voir également ce type d'hybrides comme une RL sur l'espace des *chemins de décision* menant à une solution (plus générale que la notion précédente) [Jussien 99].

Une analogie particulièrement élégante concerne les concepts de *point de choix* des algorithmes systématiques et de *voisinage* de la RL. Ce sont des structures locales à un

état donné de la recherche, éventuellement vides, qui définissent l'ensemble des transitions qui peuvent lui succéder. Les deux techniques explorent ces structures locales par une succession de transition jusqu'à ce qu'un voisinage (ou point de choix) vide soit rencontré ; la RL peut alors redémarrer une recherche (*multi-start*) ou recourir à un voisinage plus large, alors que les algorithmes systématiques effectuent un retour arrière. [Laburthe 98a], à partir de cette analogie, considère la recherche d'une solution comme une séquence d'états et un algorithme de recherche comme le *contrôle* des transitions entre états fondé sur les points de choix ; un langage de spécification d'algorithmes hybrides fondé sur ces notions et qui sépare la définition des points de choix et de leur contrôle y est présenté (voir section 3.4.3).

3.3.3 Intégration

Depuis l'utilisation des algorithmes de RL pour résoudre des CSP (au début des années 90), de nombreuses études ont tenté de déterminer des éléments susceptibles de rendre compte de leur efficacité, puis de les incorporer au sein d'hybrides, construits à partir d'algorithmes systématiques et plus efficaces que l'une seule des deux techniques. Le succès croissant des techniques de consistance locale a bien sûr suscité la démarche inverse, c'est-à-dire l'intégration de *look-ahead* et de recherche par *backtrack* dans des algorithmes de RL. Au demeurant, certains hybrides peuvent être classés dans l'une ou l'autre catégorie suivant la perspective adoptée. Nous passons en revue dans les sections suivantes les principales combinaisons des deux paradigmes.

Backtrack flexible

L'une des faiblesses du *backtrack* est son manque de liberté de déplacement dans l'arbre de recherche. Si la solution ne se trouve pas dans la partie « la plus à gauche » (i.e. visitée en premier) de l'arbre et que le *look-ahead* est inefficace pour l'élaguer, il faut un temps de calcul exponentiel pour la découvrir et on ne peut pas résoudre les instances de grande taille. Cette situation risque d'autant plus de se produire que les heuristiques d'instanciation des variables sont « aveugles » au début de la recherche (e.g. tous les domaines sont identiques dans un problèmes de coloriage de graphe) et sont susceptibles d'indiquer une mauvaise décision : une erreur de l'heuristique sur les premières variables coûtera cher en temps de calcul car un sous-arbre très vaste devra être exploré avant qu'elle soit reconsidérée.

Des algorithmes hybrides ont été proposés à partir de BT pour disposer de plus de flexibilité qu'avec le retour arrière chronologique (« classique ») sur le choix de la variable (ou *décision*) à remettre en cause. Ils peuvent être décrits comme des algorithmes de RL sur l'espace des *instanciations partielles* ou des *chemins de décision*. La complétude est parfois sacrifiée pour privilégier l'efficacité de l'exploration.

Backtrack dynamique Les algorithmes *Dynamic Backtracking* (DB) et *Partial Order Backtracking* (PDB) [Ginsberg 93] utilisent des nogoods pour backtracker sur une variable en conflit sans remettre en cause les valeurs des variables intermédiaires. On peut considérer

que la structure de l'arbre de recherche est modifiée dynamiquement et que la variable désigné pour le retour arrière est replacée dans l'ensemble des variables non-instanciées. PDB permet une plus grande liberté dans le choix de cette variable et laisse la possibilité de choisir des heuristiques équivalentes à celles de la RL (e.g. GSAT)

Une extension de ces algorithmes, *Path-Repair*, est présentée dans [Jussien 99]. *Path-Repair* rend générique le mécanisme de gestion des nogoods pour utiliser d'autres heuristiques fondées sur le voisinage du *chemin de décisions* correspondant à l'état de la recherche. Une heuristique de type *recherche taboue*, qui rend la recherche incomplète, est utilisée pour résoudre des problèmes de *scheduling*.

Avec des techniques de CBJ pour maintenir l'arc-consistance, [Prestwich 01] propose une technique similaire, l'algorithme *Incomplete Dynamic Backtracking* (IDB), dans lequel une ou plusieurs variables instanciées choisies aléatoirement ou avec une heuristique sont remises en cause à chaque *backtrack*. Cette liberté de mouvement est bien sûr gagnée au détriment de la complétude. IDB est décrit comme un algorithme de RL (appelé parfois Constrained Local Search) qui explore l'espace des affectations partielles en minimisant le nombre de variables non instanciées.

Remise en cause du parcours en profondeur d'abord Plusieurs algorithmes on été proposés pour explorer l'arbre de recherche dans un ordre différent de la classique *depth first search*. En effet, les solutions d'un CSP ne sont pas forcément réparties uniformément dans l'arbre de recherche et ces algorithmes permettent d'en « sonder » successivement différentes parties non consécutives (selon l'ordre standard), en abandonnant éventuellement la complétude de la recherche.

- *Iterative sampling* [Langley 92] : cette technique non systématique construit une solution en choisissant aléatoirement variables et/ou valeurs et dès qu'un échec est rencontré, elle backtracke jusqu'à la racine de l'arbre et recommence.
- *Iterative broadening* [Ginsberg 90] : le nombre de branches explorées à chaque nœuds est limité par une constante que l'on incrémente si aucune solution n'est trouvée. La recherche est ainsi mieux répartie dans l'arbre.
- *Bounded Backtrack Search* (BBS) [Harvey 95a] : le nombre de retours arrières est limité par une constante et la recherche redémarre de la racine en changeant d'heuristique.
- *Limited Discrepancy Search* (LDS) [Harvey 95b] : cet algorithme tente de réparer les erreurs de l'heuristique en parcourant l'arbre de recherche en commençant par la suivre strictement, puis en examinant tous les chemins qui en divergent une fois, puis deux fois etc. Cet algorithme est complet car tout l'arbre est parcouru lors de la dernière itération correspondant au chemin qui divergent n (la hauteur de l'arbre) fois de l'heuristique, i.e. la branche la plus à droite de l'arbre. LDS peut être considérée comme des RL successives qui explorent un voisinage de taille croissante de la solution de *référence* (i.e. la plus à gauche).
- *Climbing Discrepancy Search* (CBS) [Milano 02] : cette technique prolonge une idée de [Harvey 95b] qui propose, pour des problèmes d'optimisation, d'utiliser la meilleure

solution trouvée comme heuristique de référence.

Voisinage exploré par backtrack

La RL est efficace car elle permet de suivre le gradient local au voisinage de la solution courante. Or ce gradient est d'autant plus pertinent que le voisinage considéré est grand. C'est pourquoi des voisinages de plus en plus ambitieux ont été élaborés au sein des techniques de RL, mais ces grands voisinages sont coûteux à explorer exhaustivement. Comme l'un des points faibles de la RL pour les CSP est le peu d'exploitation des contraintes, on peut utiliser des algorithmes systématiques pour *filtrer* un voisinage et ne considérer que des voisins consistants ou qui améliorent le coût.

[Pesant 96] modélise ainsi le voisinage d'une solution avec la PPC (i.e. chaque solution du programme est un voisin valide) pour des problèmes de VRP (*Vehicle Routing Problem*). De même, dans le cadre de problèmes de VRP, [Shaw 98] décrit la *Large Neighbourhood Search* (LNS) en utilisant une recherche LDS dans le programme en contraintes, et [Bent 01] obtient de nouveaux résultats sur le *benchmark* SOLOMON en utilisant d'abord un *recuit simulé* pour minimiser le nombre de véhicules utilisés et favoriser les solutions sur lesquelles la LNS employée est efficace. Avec un autre type de RL, les *algorithmes génétiques*, [Rousseau 00] résout des problèmes d'emploi du temps, en utilisant la PPC pour « réparer » les croisement inconsistants et générer la population initiale.

Dans un cadre assez différent, [Minton 94] utilise l'heuristique de réparation *min-conflicts* (cf. section 3.2.2) au sein d'une RL rendue complète en intégrant le choix de la valeur de réparation dans un *backtrack*. BT cherche ainsi une séquence d'au plus n réparations (une par variable) qui mène à une solution à partir d'une instanciation totale inconsistante initiale (par exemple générée avec un algorithme glouton). L'efficacité de la méthode est attribuée à la pertinence de l'information procurée par cette instanciation totale.

Mentionnons également l'algorithme présenté dans [Barnier 98] qui hybride Algorithmes Génétiques et PPC : la population de l'AG est constituée de sous-ensembles de l'espace de recherche et chaque individu est évalué en calculant une solution sur le sous-CSP correspondant avec un programme en contraintes. En faisant varier la taille des sous-ensembles, on passe « continûment » d'une résolution par un AG « pur » (chaque « sous-domaine » est un singleton, i.e. les individus sont des instanciations totales) à une résolution par PPC « pure » (chaque individu correspond au problème entier). Cette hybridation est générique mais le choix de la taille des sous-ensembles et des opérateurs génétiques est délicat et très dépendant du problème pour assurer une « bonne » convergence de l'algorithme.

Élagage par recherche locale

Réciproquement, un algorithme de RL peut être utilisé pour détecter l'inconsistance d'un sous-arbre dans un algorithme de *backtrack*. [Sellmann 02] effectue ainsi une recherche de cliques dans les nœuds proches d'une feuille pour déterminer si le sous-arbre correspondant est inconsistant sur un problème analogue à celui des écolières de Kirk-

man. [Focacci 02] utilise également ce type d'hybridation pour élaguer des branches sous-optimales lors de la résolution d'un problème de TSP avec fenêtres de temps.

Post-optimisation

Les solutions sous-optimales obtenues par un algorithme systématique peuvent être améliorées *après coup* avec une méthode de RL pour obtenir de meilleures bornes supérieures ou une meilleure solution quand le temps est limité. On peut également utiliser une instanciation partielle et la compléter en utilisant un algorithme de *backtrack* incomplet (et rapide) comme proposé dans [Caseau 94] pour résoudre des problèmes de *scheduling*.

3.4 Systèmes

Malgré la jeunesse du domaine, de nombreux systèmes de PPC puissants ont été (et sont toujours) développés. Certains permettent d'utiliser différents types de solveurs (e.g. domaines finis et solveur linéaire avec ILOG Solver [Solver 99]) voire de paradigmes (e.g. recherche systématique ou *repair* avec ECL^{i}PS^e [ECL^{i}PS^e 01]). La plupart d'entre eux sont des extensions de Prolog (e.g. Prolog III [Colmerauer 90]), appelées systèmes de *Programmation Logique par Contraintes* (PLC), mais il existe également des compilateurs pour d'autres paradigmes de langage de programmation (e.g. la programmation concurrente avec Mozart [Mozart 02]) et des librairies pour des compilateurs de langage non logique (e.g. Lisp pour PECOS [Puget 92]). Tous les systèmes que nous considérerons sont fondés sur les concepts détaillés dans la section 3.1.7 et permettent de spécifier des CSP et des buts de recherche de manière déclarative, caractéristique héritée de la PLC, mais certains d'entre eux offrent des abstractions pour contrôler le parcours de l'arbre de recherche et utiliser d'autres stratégies que la *depth first search* classique [Choi 01b]. Ces systèmes de PPC peuvent être vus comme des langages de modélisation qui permettent le développement sûr et rapide d'algorithmes de résolution de problèmes d'optimisation combinatoire.

En revanche, les outils destinés à abstraire les mécanismes élémentaires de la résolution par RL ne sont apparus que très récemment et sont encore loin de l'élégance et de la généricité de la PPC pour modéliser et résoudre les CSP ainsi que pour intégrer les techniques spécifiques à ce type de recherche. Localizer [Michel 97] est le système le plus proche d'un langage de modélisation qui abstrait le contrôle et le maintien des structures de données d'algorithmes de RL. Des librairies de classes génériques existent également pour faciliter l'implémentation de techniques de RL (e.g. LOCAL++ [Schaerf 00]), mais sans le niveau d'abstraction nécessaire pour les classer dans les langages de modélisation.

Depuis quelques années, des langages de modélisation destinés à séparer la description du problème du paradigme de recherche utilisé pour le résoudre ont été proposés. OPL [Van Hentenryck 99] permet ainsi de spécifier un CSP et de le résoudre soit avec des techniques de PPC, soit avec des algorithmes de programmation mathématiques. Une autre démarche encore plus ambitieuse est celle de SaLSA [Laburthe 98a] qui propose de spécifier des algorithmes de recherche, aussi bien systématique que locale, à l'aide d'un langage qui

sépare la description des points de choix de celle du contrôle.

3.4.1 Langages de Programmation Logique par Contraintes

Les langages dédiés à la résolution des CSP proposent de spécifier à l'aide de *termes* les différents composants du problème et de la méthode de résolution. Ils utilisent un compilateur spécifique (éventuellement vers un langage cible) qui peut réaliser des optimisations *ad hoc* liées au contrôle de la recherche (e.g. gestion de la mémoire). Cependant, l'intégration de programmes de recherche au sein d'applications plus vastes développées avec un langage généraliste peut être difficile.

L'architecture particulière de la PPC décrite dans la section 3.1.7 est héritée des systèmes de programmation logique pour laquelle les programmes sont des ensembles de clauses logiques et leur exécution une recherche de preuves (ou « solutions ») non-déterministe contrôlée par *backtrack*. Les systèmes Prolog (« purs ») peuvent en fait être vus comme des solveurs de contraintes d'égalité (syntaxique) sur un domaine spécifique, les termes rationnels ou finis (ou encore *domaine de Herbrand*). Pour en améliorer l'expressivité, ce principe de démonstration automatique a été généralisé, en utilisant la même architecture, à des structure mathématiques plus riches que les termes rationnels. La recherche de preuves utilise alors des techniques hybrides de résolution logique et de résolution de contraintes dans ces structures mathématiques, faisant appel à des algorithmes de RO et d'Intelligence Artificielle. Cette forme originelle de PPC est appelée la Programmation Logique par Contraintes (PLC) [Jaffar 86]. Le cadre générique de la PLC, paramétré par un domaine de contraintes, est, comme celui de la programmation logique, doté de sémantiques (opérationnelles, logiques, algébriques) cohérentes qui en assurent les fondations [Jaffar 86].

Le système Prolog II développé par Alain Colmerauer [Colmerauer 82] est considéré comme le premier langage de PLC car il était doté de contraintes de diséquations sur les termes rationnels résolues en *co-routines*. Les langages de PPC se sont, depuis, considérablement étoffés en intégrant des solveurs sur des domaines variés. Prolog IV [Colmerauer 96] par exemple permet d'exprimer des CSP sur les domaines finis[23], rationnels, réels et booléens et résout les contraintes numériques (continues) avec des techniques de programmation linéaire et d'arithmétique des intervalles. D'autres langages de PLC, commerciaux ou libres, sont apparus concurremment :

- CHIP [Dincbas 88] est également l'un des systèmes pionniers de l'application industrielle de la PLC, qui a introduit les contraintes arithmétiques, symboliques et globales sur les domaines finis. Il dispose de solveurs sur les rationnels et les booléens.
- ECL^{i}PS^e [ECL^{i}PS^e 01] est un système dédié aux applications industrielles qui comportent de nombreuses extensions de la PLC classique. Des solveurs sur les domaines finis (similaires à ceux de CHIP), les ensembles, les rationnels et les réels sont dis-

[23] Ils ne sont toutefois pas traités par les techniques spécifiques exposées dans la section 3.1 mais comme des réels contraints à prendre une valeur entière avec le solveur arithmétique sur les intervalles.

ponibles, ainsi que des interfaces pour utiliser des librairies (externes) de Simplex. Mais ECL^{i}PS^e permet également de « réparer » des solutions inconsistantes avec des techniques de RL telles que *min-conflicts* (cf. section 3.2.2) en associant une valeur provisoire (*tentative value*) aux variables logiques. Des contraintes peuvent également être définies par l'utilisateur en utilisant une interface de haut niveau (*Constraint Handling Rules* (CHR)).

- CLP($\mathcal{R}$) [Jaffar 92] est une implémentation sur les réels (flottants) du cadre générique CLP($\mathcal{X}$) établi par [Jaffar 86]. Un Simplex résout les contraintes linéaires alors que le traitement des contraintes non-linéaires est différés (tant qu'elles ne sont pas linéarisées par l'instanciation d'un nombre suffisant de variables).
- GNU Prolog (et CLP(FD)) [Diaz 02] est un système *Open Source* muni d'un solveur sur les domaines finis. Il permet à l'utilisateur d'écrire ses propres contraintes à l'aide d'une seule primitive utilisée pour toutes les contraintes [Codognet 96] (de manière équivalente aux CHR).

Bien d'autres systèmes existent, et certains langages notamment sont fondés sur des extensions du cadre classique de la PLC :

- Mozart [Mozart 02] est un langage de PLC *concurrente* [Saraswat 93] qui offre aussi certaines caractéristiques. de la programmation fonctionnelle, orientée objet et distribuée.
- Mercury [Somogyi 95] et Gödel [Hill 94] sont des systèmes de PLC fortement typés. Plus récemment, [Fages 01] a proposé le formalisme de la PLC typée (TCLP).
- CHR [Früwirth 98] est un langage de règles avec gardes dédié à l'écriture de solveurs.

Les versions récentes de plusieurs de ces systèmes pallient à certaines faiblesses des premiers langages de programmation logique et disposent d'interfaces pour communiquer avec des langages généralistes populaires et des librairies d'interface graphique, ce qui permet de développer des applications complexes autonomes ou qui s'intègrent à des projets plus larges.

3.4.2 Librairies

Au lieu d'écrire un compilateur de Prolog (ou de modifier un compilateur existant), un système de PPC peut être réalisé sous la forme d'une *librairie* en utilisant un langage hôte fondé sur un paradigme différent de la programmation logique. Mais l'élégante adéquation entre Prolog et la résolution systématique des CSP disparaît. Ces librairies implémentent donc un modèle d'exécution de programmes en contraintes en ne retenant de la CLP que les composants liés au domaine d'intérêt (la résolution de problèmes d'optimisation combinatoire réels) : syntaxe déclarative, recherche non-déterministe et buts de recherche — elles ne possèdent pas en général de solveur sur les termes rationnels. Ces librairies doivent donc fournir des fonctions ou des opérateurs pour créer variables et contraintes, ainsi que pour spécifier des stratégies de recherche. Suivant le langage hôte choisi, certaines faiblesses des langages de programmation logique peuvent être évitées naturellement (efficacité, typage, ordre supérieur, structures de données, traitement des données, intégration dans des applications industrielles etc.) :

- PECOS [Puget 92] est une librairie écrite en Lisp.
- ILOG Solver [Solver 99] est le successeur de PECOS écrit en C++. Il se sert des classes de C++ pour implémenter variables, contraintes et buts de recherche et permettre à l'utilisateur de les étendre. Il incorpore un grand nombre de contraintes globales et dédiées à des applications spécifiques (e.g. ILOG Scheduler) sur les domaines finis, les ensembles et les réels. ILOG Solver propose également des librairies hybrides utilisant des algorithmes de RL (ILOG Dispatcher) et [De Backer 99] décrit un cadre général pour les implémenter. Ce système commercial est actuellement leader du marché et le plus performant en temps d'exécution.
- JCL [Torrens 97] est une librairie écrite en Java pour résoudre des CSP binaires sur les domaines finis.
- CHOCO [Laburthe 00] est une librairie pour les domaines finis écrite en Claire [Caseau 96], un langage « multi-paradigmes » (logique, fonctionnel, objet, impératif) adapté au non-déterminisme.

Des librairies destinées à la RL ont récemment été développées. La RL n'a pas donné naissance à un cadre formel simple et unifié[24] comme la PPC mais rassemble plutôt une collection de techniques similaires ; ces librairies n'ont donc pas de fondation théorique rigoureuse comme les systèmes de PPC. Elles permettent néanmoins de s'affranchir des parties fastidieuses de l'écriture des algorithmes de RL en choisissant une structure d'algorithme supposée être assez générique pour fédérer un nombre suffisant de techniques. Les librairies LOCAL++ [Schaerf 00] et EASYLOCAL++ [Gaspero 00], écrites en C++, sont des collections de classes virtuelles qui servent de cadre à la conception d'algorithmes de *hill climbing*, recuit simulé et recherche taboue. Localizer++ [Michel 01a] est une version sous forme de librairie pour C++ de Localizer, un langage de modélisation d'algorithme de RL mentionné dans la section suivante.

3.4.3 Langages de modélisation

Les langages de modélisation permettent de spécifier des problèmes et éventuellement des algorithmes pour les résoudre avec des constructions de haut niveau proches de celles de leur description abstraite (telle qu'on peut la trouver dans une publication scientifique par exemple) comme des quantificateurs, des manipulations d'ensembles etc. Ils permettent de séparer clairement les composants orthogonaux impliqués dans la résolution des problèmes d'optimisation et automatisent certains traitements, ce qui réduit les efforts de développement et favorise les expérimentations.

Nous passons en revue trois langages de modélisation destinés respectivement à la PPC (et la programmation mathématiques), à la RL et à des algorithmes hybrides :

- OPL (Optimization Programming Langage) [Van Hentenryck 99] est une tentative d'unification des langages de modélisation pour la programmation mathématique tel que AMPL [Fourer 93] et des langages de PPC, ainsi que de leurs techniques de

[24] Des études récentes comme [Taillard 98] tentent d'y remédier mais aucune implémentation n'a encore été réalisée.

résolution. OPL permet de décrire des CSP avec des notations algébriques de haut niveau, des stratégies de recherche comme en PPC et la manière de parcourir l'arbre de recherche. En outre, différents solveurs peuvent être utilisés sur le même modèle et un langage de « scripts » permet de combiner plusieurs recherches.

- Localizer [Michel 97] est un langage de modélisation d'algorithmes de RL. Il automatise la maintenance incrémentale (et efficace) de structures de données (appelées *invariants*) reliées par des dépendances fonctionnelles (voisinages, critères de sélection) et propose un cadre générique (cf. algorithme 5) pour décrire la méta-heuristique elle-même. Localizer est, à notre connaissance, le seul langage de ce type.
- SaLSA (Specification Language for Search Algorithms) [Laburthe 98a] est un langage de description d'algorithmes de recherche hybride. La recherche d'une solution, aussi bien en RL qu'en PPC, est vue comme une séquence d'états correspondant à des *points de choix*, par exemple l'instanciation d'une variable à l'une de ses valeurs possibles pour la PPC et le choix d'un voisin pour la RL. SaLSA sépare la description de ces points de choix et le contrôle de la recherche, c'est-à-dire la stratégie d'exploration des points de choix. Des opérateurs permettent de combiner des points de choix et de ne visiter qu'une partie de leurs branches (cf. section 3.3.3) : les algorithmes sont alors spécifiés avec des termes paramétrés par leur « gloutonnerie ». Les différents types de propriétés à maintenir (consistance locale, invariants) sont pris en charge par des solveurs spécifiques. SaLSA est très élégant mais seule une partie très restreinte du langage a été implémentée.

Deuxième partie

FaCiLe

Chapitre 4

Une librairie de contraintes fonctionnelle

FaCiLe est une librairie *Open Source*[1], simple et concise de programmation par contraintes entièrement écrite avec Objective Caml [Leroy 00], un langage de programmation fonctionnel de la famille de ML[2]. FaCiLe implémente la plupart des contraintes usuelles disponibles dans les systèmes de PPC les plus populaires (Prolog [Colmerauer 90], ILOG Solver [Solver 99], CHIP [Dincbas 88], ECL^{i}PS^e [ECL^{i}PS^e 01]...) pour les domaines finis et les ensembles finis[3], un langage de buts destiné à la recherche de solutions et l'optimisation, ainsi qu'un module d'*invariants* permettant de maintenir des critères de choix lors de la recherche. Notre librairie permet également de construire des contraintes, buts et invariants définis par l'utilisateur ou en combinant des primitives à l'aide d'une interface simple et souple qui fait largement appel au niveau d'abstraction élevé permis par le langage (fonctions d'ordre supérieur). FaCiLe a été conçu en évitant délibérément l'approche « encore un nouveau langage » pour profiter des caractéristiques d'OCaml telles que le typage, les clôtures, le système de module, le récupérateur de mémoire, l'efficacité et la portabilité. Le prototypage et l'expérimentation s'en trouvent facilités : la modélisation, le traitement des données et l'interface se conjuguent au sein d'un même et puissant langage qui garantit un niveau de sécurité élevé.

Les motivations de FaCiLe sont en fait très proches de celles du système CHOCO[4] [Laburthe 00] qui a été développé à peu près à la même période pour le langage Claire :

FaCiLe est
- petite : a peu près 4500 lignes de code ;
- extensible : briques de bases, généricité (foncteurs) ;

[1] Un logiciel *Open Source* est un programme assorti d'une licence qui garantit la liberté de le redistribuer gratuitement ou commercialement, l'accès gratuit à son code source et la possibilité de le modifier. Un tiers est ainsi autorisé à en corriger les erreurs et à le faire évoluer pour répondre à ses besoins.

[2] `cm.bell-labs.com/cm/cs/what/smlnj/sml.html`

[3] Des variables et contraintes sur les intervalles de flottants sont également prévues, mais pas encore implémentées.

[4] `www.choco-constraints.net`

- efficace : presque assez ;
- simple : comme son nom l'indique ;
- gratuite : licence LGPL, code source disponible.

4.1 Introduction

4.1.1 Le langage

Lors de la conception de FaCiLe, nous avons tout d'abord délibérément écarté l'approche « encore un autre (plus ou moins) nouveau langage de plus » : notre propos n'est pas de réinventer la roue et devoir refaire le travail et les compromis des concepteurs de langages de programmation, mais bien de réaliser un système de PPC simple et expressif. Notre projet diffère ainsi de systèmes tels que Oz [Smolka 98], Claire [Caseau 96] ou Mercury [Somogyi 95] qui sont des compilateurs. Le choix d'une *librairie* s'est donc naturellement imposé pour satisfaire nos préoccupations d'efficacité, portabilité et disponibilité.

Un second choix doit ensuite être fait pour le langage hôte de notre librairie, entre les langages de « bas niveau » impératifs très proches du fonctionnement du matériel et les langages de « haut niveau » à vocation applicative qui permettent de se concentrer sur l'architecture du système. Dans la première catégorie se trouve l'inévitable et très populaire C++ pour lequel ILOG Solver [Solver 99], le logiciel commercial qui connaît le plus grand succès, a été développé. L'avantage principal d'un tel langage — hormis son omniprésence — réside dans sa rapidité d'exécution, mais nous trouvons que les contreparties à payer sont inabordables : la gestion mémoire est manuelle et donc fastidieuse et source de nombreuses erreurs difficiles à détecter et corriger, le typage est débile (au sens premier du terme évidemment), les structures de données sont pauvres ainsi que la modularité, le code est verbeux et peu expressif etc. Dans la seconde catégorie, la programmation logique et la programmation fonctionnelle sont les candidats de choix.

La programmation logique est historiquement le langage de prédilection de la PPC : les contraintes sont naturellement des prédicats et la recherche non-déterministe de solution (preuve) est inhérente à Prolog. Un système de programmation logique basique, qui peut être vu comme un solveur de contraintes sur les termes de l'univers de Herbrand, est alors enrichi avec des techniques spécifiques (co-routines, attribut de variable, algorithmes d'inférence...) pour gérer les contraintes sur des structures plus complexes (p.ex. domaines finis). Pourtant, si cette élégante adéquation est pratiquable pour l'exhibition de petits exemples, elle est trop coûteuse dans un cadre industriel pour des problèmes de taille réelle : le traitement des données, le codage d'algorithmes de filtrage efficace pour des contraintes globales (d'ailleurs souvent écrits en C), l'affichage des résultats etc. sont des tâches délicates dans le cadre de la programmation logique. Le manque de typage et de vérifications à la compilation dans les systèmes Prolog classiques (hormis quelques systèmes récents comme Mercury [Somogyi 95], Gödel [Hill 94] ou TCLP [Fages 01]) dissuadent également de les utiliser dans le développement de gros projets pour des raisons de sécurité et de robustesse.

La programmation fonctionnelle n'offre pas la sémantique logique adéquate pour exprimer des contraintes, mais les langages de la famille de ML évitent les écueils des langages de bas niveau en proposant une sémantique forte, la manipulation de termes d'ordre supérieur et de clôtures, la gestion de mémoire automatique, et ceux que partagent les systèmes Prolog communs car ils permettent la modularité avec généricité (*foncteurs*), un typage fort et polymorphe. De plus, l'implémentation OCaml est très efficace en temps d'exécution et en occupation mémoire [Bagley 01, McClain 99, Morrisett 00], portable[5], *Open Source* et fournit des librairies bien documentées. Ces caractéristiques puissantes nous ont conduits à utiliser OCaml à la fois comme langage source pour écrire des programmes en contraintes *et* pour l'implémentation elle-même de la librairie. Quelques systèmes de PPC ont déjà été conçus au-dessus de langage fonctionnel tel que Lisp comme le séminal et orienté objet PECOS [Puget 92] (ancêtre de Solver) ou SCREAMER [Siskind 93]. Notre approche est plus proche du système présenté dans [Henz 99] qui utilise l'ordre supérieur et les exceptions de ML pour permettre de construire des moteurs de recherche avec backtrack programmables.

4.2 Un avant-goût de FaCiLe

FaCiLe est composée d'une vingtaine de modules qui structure la librairie et son utilisation : `Domain` pour la manipulation des domaines, `Fd` pour les variables, `Arith` pour les expressions arithmétiques, `Cstr` pour les contraintes, `Goals` pour les buts de recherche et leur contrôle etc. Le type des objets manipulés dans chacun de ces modules est nommé t, et la fonction principale permettant d'en créer des instances est `create`. Nous nous référerons à ces différents objets avec leur notation qualifiée par le nom du module : `Module.objet`.

4.2.1 Séquence magique avec FaCiLe

La programmation avec FaCiLe se conforme au schéma classique de la PPC : des variables à domaine sont créées, puis des contraintes les utilisant directement ou par l'intermédiaire d'expressions arithmétiques sont posées, et un but de recherche est finalement résolu pour obtenir une solution. Le programme 4.1 résout (naïvement) le problème classique de la *séquence magique* :

Problème 4.1 (Séquence magique) *Une séquence magique de taille n est une séquence de n entiers $X = (x_0, x_1, \ldots, x_{n-1})$ telle que le nombre d'occurrences de 0 soit x_0, le nombre d'occurrences de 1 soit x_1 etc, c'est-à-dire :*

$$\forall i \in [0, n-1] \quad |\{x \in X \text{ tel que } x = i\}| = x_i$$

Programme 4.1 (Séquence magique avec FaCiLe)

[5] OCaml est porté sur la plupart des plate-formes courantes : Linux, UNIXes, MacOS, Windows... / x86, Alpha, Sparc, PPC, HPPA, IA64... Voir `caml.inria.fr/ocaml/portability.html`.

```
let xs = Fd.array n 0 (n-1)
let is_equal_to i x = fd2e x =~~ i2e i
Array.iteri
  (fun i xi ->
    let cardi = Arith.sum Array.map (is_equal_to i) xs) in
    Cstr.post (fd2e xi =~ cardi))
  xs;
Goals.solve (Goals.Array.forall Goals.indomain xs)
```

Variables et contraintes Tout d'abord, un tableau `xs` de n variables ayant comme domaine $[0, n-1]$ est créé avec la fonction `Fd.array`. Une fonction (`is_equal_to`) est ensuite définie pour construire une contrainte d'égalité réifiée entre une variable `x` et un entier `i`, à l'aide de l'opérateur `=~~` : cette fonction renvoie une expression arithmétique booléenne qui sera égale à 1 si l'égalité est vérifiée et à 0 sinon. Une itération sur `xs` est alors effectuée pour poser la famille de contraintes qui spécifie le nombre d'occurrences d'un entier donné dans le tableau : il s'agit d'un simple appel à l'itérateur `Array.iteri` de la librairie standard d'OCaml qui applique séquentiellement une fonction à chaque indice (`i`) et élément (`xi`) du tableau. Donc, pour chaque indice `i` de `xs`, le nombre d'éléments égal à `i` est d'abord compté à l'aide d'un second itérateur `Array.map` qui applique la fonction (`is_equal_to`) pour transformer `xs` en un tableau d'expressions booléennes associées aux contraintes réifiées $[x_0 = i; \ldots; x_j = i; \ldots; x_{n-1} = i]$; la somme de ses éléments est calculée par l'expression renvoyée par la primitive de FaCiLe `Arith.sum`. On contraint ensuite ce cardinal à être égal à `xi` en posant une contrainte avec la fonction de FaCiLe `Cstr.post`.

Typage et surcharge La discipline de typage stricte d'OCaml ne permet pas la surcharge d'opérateurs ou fonctions ni les conversions implicites. Des opérateurs spéciaux et des fonctions de conversions font donc partie de FaCiLe, comme le montre l'exemple 4.1 :

- Les opérateurs et contraintes arithmétiques sont suffixés par `~` : `+~`, `%~`, `=~`, `<~`...
- Les contraintes réifiées par `~~` : `=~~`, `<>~~`...
- Conversion d'une variable à domaine fini en expression arithmétique : `fd2e`.
- Une constante (*integer*) en expression : `i2e`.

Cette syntaxe particulière peut paraître lourde à l'utilisateur d'autres systèmes de PPC peu typés où les opérateurs sont surchargés et les conversions implicites. Dans de tels langages, la signification d'une fonction dépend de son contexte d'utilisation selon un système de règles sophistiqué prétendûment intuitif, qui peut même aller jusqu'à masquer la création de variables — pour les contraintes réifiées en général — alors que c'est une opération très coûteuse. En contrepartie, dans FaCiLe, la signification des expressions arithmétiques est spécifiée simplement et de manière cohérente : le résultat correspond exactement à ce qui est attendu et les erreurs sont détectées à la compilation[6].

[6]Néanmoins, un des sujets actifs de recherche sur OCaml est l'ajoût d'une forme de surcharge cohérente et sûre (autant que faire se peut...) au sein du système de type — un prototype est déjà disponible `pauillac.inria.fr/~furuse/generics/`. C'est en fait la combinaison de surcharges *et* de conversions

Recherche de solution La ligne finale du programme 4.1 est une itération conjonctive de la primitive d'étiquetage (i.e. un but) `Goals.indomain`. Un nouvel itérateur polymorphe est utilisé, `Goals.Array.forall`, dont le type est :

$$(\alpha \rightarrow Goals.t) \rightarrow \alpha\ array \rightarrow Goals.t$$

α est une variable de type et `Goals.t` et le type des buts. L'itérateur n'a donc besoin que d'une fonction qui s'applique à des éléments du tableau et qui renvoie un but pour construire la conjonction des buts correspondant dans l'ordre croissant des indices tu tableau. Un paramètre optionnel (qui n'est pas montré ici) de type $\alpha\ array \rightarrow int$ qui renvoie l'indice de l'élément courant à sélectionner peut également être fournit à `Goals.Array.forall` comme heuristique pour modifier cet ordre. Notons aussi que FaCiLe est doté du but `Goals.Array.labeling` de type $Fd.t\ array \rightarrow Goals.t$ (`Fd.t` étant le type des variables à domaine fini) équivalent à `Goals.Array.forall Goals.indomain`.

La recherche de solution est enfin effectuée en appliquant le but obtenu à la fonction `Goals.solve`, dont le type est $Goals.t \rightarrow bool$, et qui renvoie `true` si le but réussit et `false` s'il échoue.

Cet exemple basique n'est sans doute guère plus court que s'il avait été écrit avec un langage *ad hoc*, mais sa concision est manifeste pour un système fondée sur une librairie. La manipulation de fonctions en tant qu'objet quelconque rendue possible par un langage tel que OCaml et le style de programmation par itérateur qui applique un traitement uniforme à une structure de données permettent de définir et composer des fonctions de manière simple et concise. En outre, l'inférence et la vérification des types assure la correction du code sans surcharge verbeuse.

4.2.2 Profil d'un programme en FaCiLe

Plus généralement que dans l'exemple 4.1 où la concision du code est privilégiée, le profil typique d'un programme utilisant FaCiLe — après la partie préliminaire d'acquisition et de traitement des données — est le suivant :

1. Création de domaines :

   e.g. `Domain.create` : $int\ list \rightarrow Domain.t$

2. Création de variables :

   e.g. `Fd.create` : $Domain.t \rightarrow Fd.t$

3. Création d'expressions arithmétiques :

   e.g. `Arith.sum_fd` : $Fd.t\ array \rightarrow Arith.t$

4. Création de contraintes arithmétiques :

   e.g. `=~`, `<~` : $Arith.t \rightarrow Arith.t \rightarrow Cstr.t$

implicites qui est la source d'erreurs la plus dangereuse. FaCiLe pourra donc être munie d'une interface plus légère et versatile pour les utilisateurs peu scrupuleux.

et globales :

e.g. `Alldiff.cstr` : $Fd.t\ array \rightarrow Cstr.t$

5. Pose des contraintes :

   e.g. `Cstr.post` : $Cstr.t \rightarrow unit$

6. Création de buts :

   e.g. `Goals.Array.labeling` : $Fd.t\ array \rightarrow Goals.t$

7. Recherche :

   e.g. `Goals.solve` : $Goals.t \rightarrow bool$

4.3 Fonctionnalités

4.3.1 Domaines et variables

Pour que leur utilisation soit sûre, les domaines et les variables de FaCiLe ont un type abstrait et ne peuvent être manipulés que par l'intermédiaire de fonctions qui en assurent l'intégrité. Les fonctions de création, par exemple, ne permettent pas que ces objets de base ne soient pas initialisés, ce qui est une source d'erreurs fréquentes avec les systèmes de PPC dont le langage hôte est impératif.

Afin de pouvoir les partager sans risque et faciliter ainsi la gestion de leur modification avec le backtrack, les domaines de FaCiLe sont fonctionnels, c'est-à-dire qu'ils ne sont pas modifiables « en place » et que les fonctions qui les traitent renvoient toutes un « nouveau » domaine (qui partage éventuellement une partie de la structure du domaine d'origine). FaCiLe propose actuellement deux types de domaines (et donc deux types de variables) : les entiers et les ensembles d'entiers.

Les variables possèdent un *attribut* lorsqu'elles sont inconnues (pas encore instanciées) auquel il faut accéder pour obtenir leur domaine ou la liste des contraintes qui leur sont attachées. Elles le perdent quand elles deviennent instanciées et ne contiennent plus alors que la valeur d'instanciation. La distinction entre ces deux états est assurée par le typage. Bien que le type des variables soit abstrait (`Fd.t` et `SetFd.t`), la fonction `Fd.value` : $t \rightarrow$ *concrete* permet d'en obtenir une pseudo-représentation concrète avec un type « somme » : $type\ concrete = Val\ of\ elt \mid Unk\ of\ attr$. Le style de programmation par *pattern-matching* associé au typage assure la distinction entre l'état *instanciée* et l'état *inconnue* avec sûreté et facilité :

```
match Fd.value x with
  Val v -> fv v
| Unk attr -> fa attr
```

Les deux types de variables `Fd.t` et `SetFd.t` possèdent une interface pratiquement identique. Outre les diverse fonctions d'accès aux informations contenues dans une variable, deux fonctions permettent de réduire son domaine :

- `unify` : $t \rightarrow elt \rightarrow unit$
  Instanciation avec un élément du domaine.
- `refine` : $t \rightarrow domain \rightarrow unit$
  Réduction avec un domaine inclus dans le domaine courant de la variable.

Ces réductions de domaine entraînent le déclenchement d'*événements* auxquels les contraintes vont être attachées pour être réveillées :

- **on_refine** : modification du domaine ;
- **on_min** : modification de la borne inférieure ;
- **on_max** : modification de la borne supérieure ;
- **on_subst** : instanciation.

Un échec survient si le domaine d'une variable est réduit à l'ensemble vide.

4.3.2 Contraintes

Les contraintes de FaCiLe ont un type abstrait (`Cstr.t`) et peuvent être manipulées (par exemple combinées avec des opérateurs logiques) avant d'être posées (fonction `Cstr.post` : $Cstr.t \rightarrow unit$), c'est-à-dire d'être ajoutées à l'ensemble des contraintes actives.

Contraintes arithmétiques FaCiLe assure avec le typage la distinction entre variables et expressions arithmétiques pour rendre impossible (détection à la compilation) des opérations telles que le raffinage d'une expression ou la manipulation de son hypothétique domaine.

Les contraintes arithmétiques linéaires et non-linéaires sont construites à l'aide d'un opérateur (équation, diséquation ou inéquation stricte ou large) et de deux *expressions arithmétiques*. Ni les variables ni les entiers (natifs) ne sont des expressions arithmétiques qui possèdent leur propre type abstrait `Arith.t` ; des fonctions de conversion doivent être utilisées (`fd2e` et `i2e`), ce qui évite toute confusion. Réciproquement, une expression peut-être transformée en variable (`e2fd`). Les expressions peuvent être combinées pour en former de plus complexes avant de construire des contraintes. L'utilisateur dispose des opérateurs suivants pour écrire une expression :

- `+~`, `-~`, `*~`, `/~` ;
- `**~` exponentiation à une puissance entière ;
- `%~` modulo ;
- `Arith.abs` valeur absolue.

ainsi que des sommes et produits généralisés et du produit scalaire sur des tableaux de variables ou d'expressions.

Les contraintes arithmétiques peuvent également être réifiées et combinées avec les opérateurs logiques suivants :

- `&&~~` : conjonction ;
- `||~~` : disjonction ;
- `=>~~` : implication ;
- `<=>~~` : équivalence ;
- `xor` : ou exclusif ;

– `not` : négation.

Contraintes globales Des contraintes basiques sur les tableaux de variables permettent d'obtenir leur minimum, leur maximum (ou l'index de la variable correspondante) ou de les indexer par une variable logique.

Des contraintes globales qui permettent de modéliser efficacement des sous-problèmes récurrents en optimisation combinatoire sont également disponibles dans la librairie :

– `Alldiff.cstr` : « tous différents » ;
– `Gcc.cstr` : cardinalité généralisée [Régin 96] ;
– `Sorting.cstr` : tri [Bleuzen-Guernalec 97].

Par exemple, la contrainte de cardinalité généralisée peut être employée pour résoudre le problème de la séquence magique plus rapidement qu'avec le programme 4.1 en remplaçant les contraintes réifiées par :

```
let cards_values = Array.mapi (fun i xi -> (xi, i)) xs
Cstr.post (Gcc.cstr xs cards_values);
```

Pour pouvoir s'adapter à la spécifité du problème à résoudre, le degré de filtrage des contraintes « tous différents » et de cardinalité peut être paramétré avec des arguments optionnels. Le compromis entre coût (en temps de calcul) et puissance du filtrage peut ainsi être choisi finement.

Contraintes sur les ensembles Les variables d'ensembles sont utiles pour modéliser naturellement certains problèmes très symétriques tels que celui des écolières de Kirkman [Barnier 02b]. Les contraintes ensemblistes portant sur des variables d'ensembles exclusivement permettent de manipuler l'union, l'intersection, d'assurer l'inclusion ou la disjonction et d'ordonner des ensembles ; d'autres utilisent des variables entières pour pouvoir accéder au cardinal, au minimum ou garantir l'appartenance d'un élément.

Deux contraintes globales viennent compléter le pouvoir d'expression des variables d'ensembles :

– `all_disjoint` : disjonction généralisée ;
– `at_most_one` : les variables d'un tableau ont un cardinal fixé et au plus un élément en commun deux à deux.

Contraintes utilisateur Pour que FaCiLe puisse être efficace sur des problèmes réels, il faut pouvoir introduire des algorithmes de filtrage dédiés à un problème particulier et subvenir au manque d'expressivité des contraintes arithmétiques. La librairie offre donc à l'utilisateur la possibilité de définir ses propres contraintes par l'intermédiaire d'une interface de haut niveau, aussi générale que celle destinée au développement, en définissant les fonctions suivantes :

– `update` : propagation. Un identificateur (entier) peut être passé en paramètre pour identifier la cause du réveil. La fonction doit renvoyer un booléen pour indiquer si elle est résolue ou non.

- `delay` : événements de réveil. La contrainte peut être réveillée sur l'instanciation, la modification du domaine, du minimum ou du maximum d'une variable, accompagné optionnellement de l'identificateur transmis à `update`.
- `check` (optionnelle) : test de satisfaction pour la réification. La fonction doit renvoyer un booléen ou lever l'exception `DontKnow`.
- `not` (optionnelle) : négation de la contrainte pour la réification.
- `init` (optionnelle) : initialisation. Cette fonction est la première appelée quand la contrainte est posée.

Ces fonctions doivent être passées en arguments de la fonction de création de contraintes `Cstr.create`.

Les fonctions optionnelles `check` et `not` doivent être fournies simultanément pour que la contrainte soit réifiable, sinon une exception est levée quand l'utilisateur tente de la réifiée. D'autres arguments optionnels peuvent également être passés à la création d'une contrainte : sa priorité (parmi `immediate`, `normal` et `later`), son nom (une chaîne de caractère) et une fonction d'affichage.

Événements utilisateur FaCiLe est doté d'un mécanisme d'événements génériques (de type abstrait `Cstr.event`) qui permet à l'utilisateur d'en définir de nouveaux (outre les quatre[7] déjà mentionnés qui sont déclenchés par le raffinage des variables) pour y attacher des objets réactifs, i.e. des contraintes. L'abstraction de la gestion des contrainte constitue un support pour maintenir des structures de données avec l'état du système ou observer ses propriétés et son comportement.

FaCiLe exporte à cette fin les fonctions suivantes :

- `new_event` : crée un nouvel événement ;
- `register` : attache une contrainte à un événement (éventuellement avec un identificateur entier) ;
- `schedule` : déclenche un événement ;
- `registered` : permet d'obtenir la liste des contraintes attachées à un événement.

4.3.3 Recherche et langage des buts

FaCiLe contrôle la recherche de solutions avec des *buts* explorés traditionnellement en profondeur d'abord. Le langage de buts de FaCiLe offre une expressivité proche des buts de Prolog et une sémantique claire grâce aux fonctionnalités d'ordre supérieur autorisées par ML. Des buts de recherche complexes peuvent ainsi être exprimés simplement par composition ou itération des buts d'exploration et d'étiquetage prédéfinis de FaCiLe.

Il est cependant indispensable pour résoudre des problèmes réels de pouvoir définir des buts spécifiques, impossible à exprimer à l'aide de primitives quelque soit la taille de la collection de buts proposée par un système de PPC. FaCiLe permet à l'utilisateur de

[7] Les événements primitifs de FaCiLe sont en fait au nombre de huit, car les événements des variables entières et d'ensembles sont distincts.

spécifier simplement et de manière très concise de nouveaux buts éventuellement récursifs et de partager des données localement en passant la continuation du calcul en paramètre.

Les buts de FaCiLe sont simplement des clôtures évaluées paresseusement lors de leur exécution par un interpréteur (`Goals.solve`) et leur type est abstrait (`Cstr.t`) pour en garantir l'intégrité.

Conjunction et disjonction Les buts peuvent être combinés grâce à des opérateurs de conjonction (`&&~`) et de disjonction (`||~`) qui ont pour éléments neutres respectifs `Goals.success` et `Goals.fail`. Par exemple, en utilisant le but prédéfini d'instanciation non-déterministe d'une variable, `Goals.indomain`, on peut facilement écrire la boucle d'énumération disjonctive d'une variable :

```
let enum x = (Goals.indomain x &&~ Goals.fail) ||~ Goals.success
```

Itérateurs La programmation fonctionnelle permet de composer des fonctions en utilisant des *itérateur*. Un itérateur est associé à un type de données et définit un contrôle « par défaut » pour traiter cette donnée en la parcourant. En imitant les itérateurs de la librairie standard d'OCaml, FaCiLe fournit des itérateurs polymorphes conjonctifs et disjonctifs sur les tableaux et les listes (sous-modules `Goals.Array` et `Goals.List`) compatibles avec le contrôle des buts. Par exemple, l'itérateur conjonctif sur un tableau :

$$\texttt{Goals.Array.forall} : (\alpha \to \mathit{Goals.t}) \to \alpha\ \mathit{array} \to \mathit{Goals.t}$$

applique uniformément un but sur tous ses éléments par conjonction :

$$\texttt{Goals.Array.forall g [|x1; ...; xn|]} \Leftrightarrow \texttt{g x1 \&\&~ ... \&\&~ g xn}$$

L'étiquetage d'un tableau de variables, par exemple, est la simple itération du but d'instanciation d'une variable. Ce but s'écrit simplement comme une composition grâce à l'application partielle :

```
let labeling_array = Goals.Array.forall Goals.indomain
```

et le type de cette fonction est donc :

$$\mathit{Fd.t\ array} \to \mathit{Goals.t}$$

Comme une matrice n'est qu'un tableau de tableaux, il suffit simplement de composer la fonction précédente avec l'itérateur pour obtenir un itérateur sur les matrices :

```
let labeling_matrix = Goals.Array.forall Goals.labeling_array
```

de type :

$$\mathit{Fd.t\ array\ array} \to \mathit{Goals.t}$$

Buts utilisateur FaCiLe permet à l'utilisateur de définir ses propres buts à l'aide de trois primitives :

- `Goals.atomic` : $(unit \to unit) \to Goals.t$
  Cette fonction permet d'écrire des buts qui ne réalisent que des effets de bord. Par exemple un but d'affichage d'une variable peut s'écrire :
  ```
  let print v = Goals.atomic (fun () -> Fd.fprint stdout v)
  ```
- `Goals.create` : $(\alpha \to Goals.t) \to \alpha \to Goals.t$
  Cette fonction est la plus générale. Elle prend en argument une fonction qui renvoie un nouveau but et lui passe également un argument. Elle permet également d'écrire des buts récursifs comme l'itération disjonctive sur une liste[8] :
  ```
  let rec iter_disj goal list =
    Goals.create
      (function
          [] -> Goals.fail
        | x :: xs -> goal x ||~ iter_disj goal xs)
      list
  ```
  de type :
  $$(\alpha \to Goals.t) \to \alpha\ list \to Goals.t$$
- `Goals.create_rec` : $(Goals.t \to Goals.t) \to Goals.t$
  Cette dernière fonction de création de buts est destinée au cas particulier des buts récursifs à argument constant : l'argument du but passé en argument est le but lui-même. De tels buts pourraient bien sûr s'écrire à l'aide de la fonction précédente `Goals.create`, mais FaCiLe optimise la représentation de ce type de but quand ils sont créés avec cette fonction spécifique. L'instanciation non-déterministe d'une variable à l'aide d'une fonction choisissant une valeur dans son domaine peut ainsi s'écrire :
  ```
  let instantiate choose var =
    Goals.create_rec
      (fun self ->
        match Fd.value var with
          Val _ -> Goals.success
        | Unk attr ->
            let dom = Var.Attr.dom attr in
            let x = choose dom in
            let left = Goals.atomic (fun () -> Fd.subst var x)
            and right =
              Goals.atomic
                (fun () -> Fd.refine var (Domain.remove x dom)) in
            left ||~ (right &&~ self))
  ```
  et son type est :
  $$(Domain.t \to Fd.elt) \to Fd.t \to Goals.t$$

[8] Cette fonction est déjà prédéfinie par `Goals.List.exists`.

Si la variable est instanciée, le but renvoyé est `Goals.success`, sinon la fonction `choose` choisit une valeur `x` dans le domaine de la variable, puis on définit les deux buts de raffinage correspondant au point de choix : la première (`left`) instancie la variable à `x` et la seconde (`right`) retire `x` du domaine de la variable. Finalement, le but renvoyé effectue soit `left`, soit la conjonction de `right` et du but lui-même.

Continuation Pour implémenter en Prolog des buts de recherche complexes qui s'échangent des données, il suffit simplement de partager des variables logiques entre les arguments des buts et l'unification se charge du reste. Ainsi, les paramètres *out* du premier but sont pris comme des paramètres *in* du second dans le schéma suivant (syntaxe à la Prolog) :

$$g_1(P_{in}, P_{out}),\ g_2(P_{out}) \ldots$$

Avec FaCiLe, les variables à domaines pourraient jouer un rôle similaire, mais les entiers (ou les ensembles d'entiers) ne sont pas des structures de données douées d'un grand pouvoir d'expression. On peut également utiliser les références backtrackables polymorphes (cf. section 4.3.4) disponibles dans FaCiLe comme variables globales partagées par les buts, mais le contrôle non-déterministe de l'exécution des buts rend ce style de programmation difficile à maîtriser.

Cependant, comme ML permet de manipuler des fonctions comme n'importe quelle autre donnée, on peut passer en argument d'un but une continuation (d'ordre supérieure, i.e. une fonction) qui pourra prendre elle-même en paramètre le résultat d'un calcul quelconque effectué dans ce but :

```
let g1 = fun pin cont →
  Goals.create
    (fun () →
       let pout = ... in
       cont pout)
    () in
g1 pin g2
```

L'encodage de buts de recherche complexes avec des continuations s'intègre ainsi naturellement avec le style fonctionnel et atteint un pouvoir d'expression proche de celui des buts de Prolog. Pour illustrer cette expressivité, l'exemple suivant implémente le but Prolog classique `delete` et son utilisation en passant son résultat (un élément et le reste de la liste) à un but pris en paramètre :

Prolog

```
delete([X | Xs], X, Xs).
delete([X | Xs], Y, [X | Ys]):- delete(Xs, Y, Ys).

:- delete(L, X, Xs), cont(X, Xs).
```

FaCiLe

```
let rec delete = fun l cont ->
  Goals.create
    (function
        [] -> Goals.fail
      | x::xs ->
          cont x xs ||~ delete xs (fun y ys -> cont y (x::ys)))
    l

delete l g
```

La liste `l` est explorée disjonctivement en appliquant la continuation `cont` à un élément et au reste de la liste — et l'ordre de la liste est préservé en replaçant l'élément sélectionné en tête de liste.

Les idiomes de Prolog ne s'écrivent évidemment pas de manière plus concise ou élégante avec FaCiLe, mais la puissance d'expression procurée par l'ordre supérieur permet d'obtenir un langage de buts de recherche souple doté d'une sémantique cohérente. Comparées aux macros qui permettent de définir des buts dans ILOG Solver, et qui souffrent d'une sémantique purement syntaxique, ou à l'écriture de nouveaux objets verbeuse et fastidieuse, les briques de base du module des buts de FaCiLe offrent une solution bien plus élégante et expressive.

4.3.4 Mécanismes génériques

Comme mentionné dans la section 4.3.2, FaCiLe dispose d'événements génériques que l'utilisateur peut utiliser pour implémenter un contrôle réactif similaire à celui des contraintes. FaCiLe offre également d'autres mécanismes génériques qui s'intègrent naturellement dans un système d'optimisation sous contraintes : les références backtrackables et les invariants.

Ces deux types d'objets ont la particularité d'être polymorphes, c'est-à-dire que leur généricité réside dans le fait que n'importe quelle structure de données peut leur être passé en paramètre.

Réferences backtrackables

Les références backtrackables polymorphes, fournies par le module de la pile, permettent de faire « vivre » des structures de données mutables avec le backtrack : elles abstraient

la maintenance des structures de données lors de la recherche. L'utilisateur peut utiliser ces références pour écrire des algorithmes plus simplement car il n'a pas à se soucier de maintenir ou recalculer ces données lors des backtracks. Elles peuvent notamment servir pour écrire des contraintes globales qui nécessitent de telles structures sophistiquées.

Leur type est abstrait :

$$type\ \alpha\ ref = \alpha\ t$$

et leur module de définition exportent trois fonctions principales pour les manipuler :

- `ref` : $\alpha \rightarrow \alpha\ ref$, création ;
- `set` : $\alpha\ ref \rightarrow \alpha \rightarrow unit$, mise-à-jour ;
- `get` : $\alpha\ ref \rightarrow \alpha$, déréférencement.

Invariants

FaCiLe propose encore un niveau d'abstraction supplémentaire pour maintenir des structures de données liées entre elles par des dépendances fonctionnelles. Inspirées des invariants de Localizer [Michel 97], un langage de modélisation pour la recherche locale, les références invariantes polymorphes de FaCiLe propagent automatiquement les modifications subies vers les autres références qui en dépendent et « vivent » avec le backtrack. Ce mécanisme s'apparente à celui des contraintes, mais avec des dépendances *directionnelles* et des variables instanciées de n'importe quel type.

L'utilisateur peut s'en servir comme dans Localizer pour intégrer des procédures de recherche locale dans sa recherche PPC. Des références invariantes natives liées aux variables à domaine sont également disponibles pour pouvoir, par exemple, maintenir facilement des critères de sélection de variables ou de valeur. FaCiLe propose un mécanisme générique qui prend en argument n'importe quelle fonction pour maintenir un invariant, et quelques invariants spécifiques maintenues efficacement (somme et produit généralisés, indexation et minimum d'un tableau).

Par exemple, le critère pour l'heuristique `dom/deg` décrite dans [Bessière 96] se calcule très simplement à l'aide de références invariantes :

```
let float_div = Invariant.binary (fun x y -> float x /. float y)

let best vars =
  let ratios =
    Array.map
      (fun v ->
        let dom = Invariant.Fd.size v
        and deg = Invariant.Fd.constraints_number v in
        float_div dom deg)
      vars in
  Invariant.Array.argmin ratios
```

Le première fonction transforme en fonction invariante la division flottante à partir d'entiers

et la seconde s'en sert pour renvoyer la référence invariante égale à l'index du plus petit ratio. Il suffit ensuite de l'intégrer dans un but récursif d'exploration d'un tableau.

4.4 Expressivité

FaCiLe et ML apportent une combinaison unique parmi les solveurs de contraintes pour écrire des buts de recherche, et plus généralement des programmes en contraintes. Les itérateurs qui abstraient le contrôle par défaut sur les structures de données jouent le rôle d'opérateurs d'agrégat et de quantificateurs. L'ordre supérieur et l'application partielle permettent de composer les fonctions comme avec des notations mathématiques mais dans un cadre rigoureux dénué d'ambiguïté. Le polymorphisme offre la généricité et constitue une couche d'abstraction supplémentaire dans un système où l'écriture de code complexe est très sûre grâce à l'analyse réalisée par le typage.

Ainsi, nous pensons que FaCiLe+ML constitue un environnement de programmation par contraintes à l'expressivité proche de celui d'un langage de modélisation (itérateurs, abstraction, ordre supérieur, polymorphisme, inférence de types), mais qui permet de traiter des problèmes réels grâce au réalisme (traitement des données, interface, efficacité du compilateur, environnement) et à la sûreté (typage strict, gestion automatique de la mémoire) du langage hôte. FaCiLe partage donc certaines motivations d'OPL++ [Michel 01b] qui estime que les constructions de C++ permettent de se débarrasser de la couche de modélisation OPL.

4.5 Perspectives

FaCiLe est une librairie essentiellement destinée à la recherche et l'enseignement, et son ambition est plus de fournir des briques de base simples et cohérentes pour pouvoir étendre la librairie et la comprendre que de tendre vers l'exhaustivité des domaines de contraintes pour concurrencer les systèmes commerciaux. Toutefois, le développement futur de FaCiLe intégrera de nouvelles contraintes dédiées telles que les contraintes de *scheduling*.

Les perspectives de développement s'orientent plutôt vers la coopération de solveurs (p.ex. une contrainte globale utilisant `lp_solve`[9]) et le support pour des paradigmes d'optimisation combinatoire différents tels que la recherche locale (techniques de « réparation » de solutions avec des variables mixtes PPC/RL).

L'extension la plus notable en cours est la réalisation d'une couche de modélisation inspirée d'OPL [Van Hentenryck 99] au-dessus de FaCiLe [Slama 02], qui s'avère concise et simple à écrire grâce aux constructions de haut niveau offertes par le langage.

[9] `www.netlib.org/ampl/solvers/lpsolve`

Chapitre 5

Implémentation de FaCiLe

L'implémentation de FaCiLe a été dans un premier temps aussi naïve que possible pour que la librairie soit petite et facile à maintenir. Elle est constituée d'environ 4500 lignes de code (auxquelles viennent s'ajouter 400 lignes d'interface (signatures) documentées exhaustivement), entièrement écrit en Objective Caml, y compris les fonctions de bas niveau comme la manipulation des domaines ou l'inférence sur les contraintes arithmétiques. FaCiLe est ainsi concise et robuste, mais son efficacité en temps d'exécution peut être moindre que celle d'un système codé en C/C++ ou finement optimisé.

5.1 Architecture

FaCiLe est constituée d'une hiérarchie d'une douzaine de modules principaux[1] qui la structurent. Aux 4500 lignes de code de la librairie s'ajoutent a peu près 400 lignes de signatures qui constituent l'interface et qui sont documentées exhaustivement ; cette interface permet également d'abstraire les types de données de FaCiLe et de préserver ainsi l'intégrité des objets manipulés et l'orthogonalité de l'implémentation vis-à-vis de l'interface.

5.1.1 Modularité

La structuration de FaCiLe en modules permet d'en hiérarchiser clairement les différents composants en suivant les dépendances entre unités de compilation. La figure 5.1 présente les principaux modules et leur dépendance. La partie du haut regroupe les fonctionnalités et structures de bas niveau telles que la pile et les domaines (*Domain*) qui ne dépendent d'aucun autre module. Le module de la pile (*Stack*) qui implémente la continuation d'échec fournit des « références backtrackables » polymorphes, i.e. des structures mutables qui « vivent » avec le backtrack (cf. section 5.2.2), aux autres composants de FaCiLe et à l'utilisateur, et permet de contrôler le backtrack lors de la résolution des buts.

[1] Quelques modules auxiliaires fournissent des fonctions utiles au débuggage ou absentes de la librairie standard d'OCaml qui ne sont pas spécifiques à un système de programmation par contraintes.

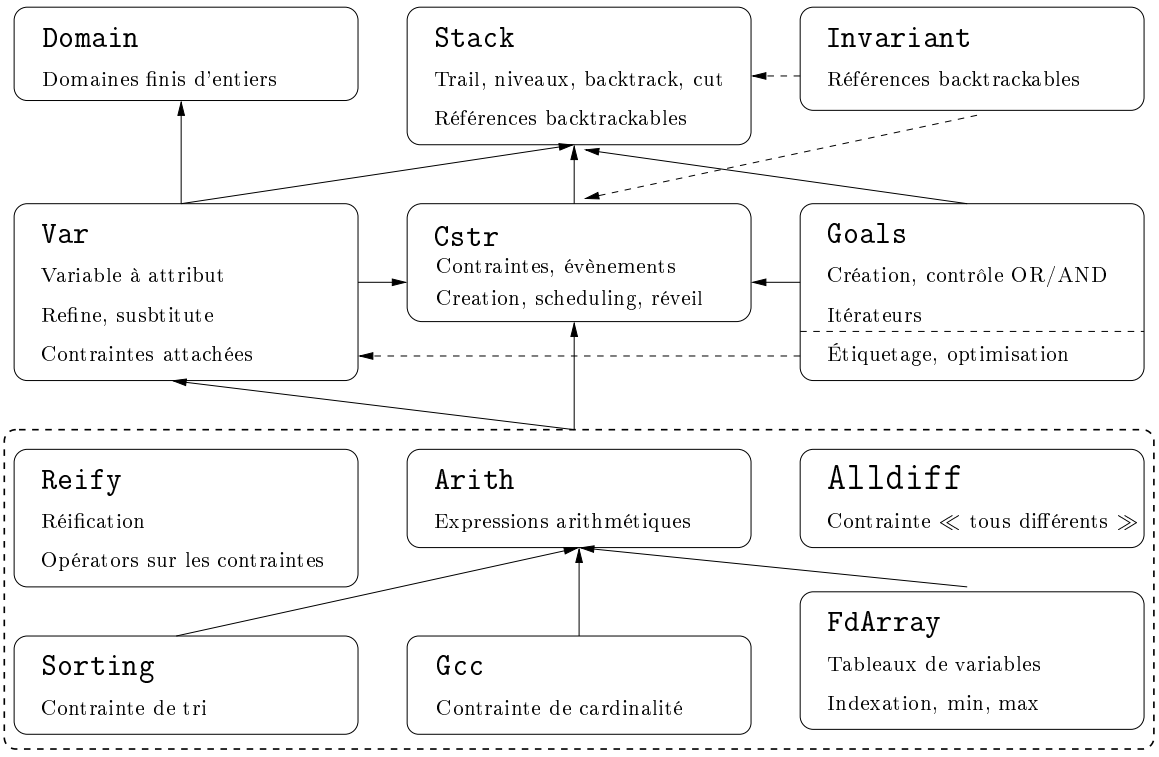


FIG. 5.1 – Architecture de FaCiLe

Viennent ensuite les briques de bases qui fournissent l'essentiel de l'interface avec l'utilisateur : les variables et leur raffinage (`Var`), la structure et la gestion des contraintes (`Cstr`) ainsi que celles des buts (`Goals`). Le module des contraintes ne fait appel qu'à la pile afin de disposer de références backtrackables utilisées dans la structure des contraintes et la gestion des événements. Les variables utilisent naturellement les domaines ainsi que la pile pour disposer de références backtrackables sur ces derniers ; la structure et le raffinage des variables dépend également du module de gestion des contraintes pour attacher celles-ci aux variables et déclencher leur réveil (mécanismes d'événements). Le troisième des modules principaux, dédiés aux buts de recherche, implémente la continuation de succès et utilise la pile pour réaliser les disjonctions. Ce module qui effectue la résolution des buts (recherche en profondeur d'abord à la Prolog) supervise le contrôle de la pile en récupérant les échecs puis en appelant le backtrack ; il utilise ensuite le mécanisme d'événements des contraintes pour déclencher un événement spécifique à chaque backtrack. Enfin, les buts d'étiquetage dépendent naturellement des variables, ainsi que les buts d'optimisation qui nécessitent également le module des contraintes (pour contraindre le coût dynamiquement).

Les autres modules de FaCiLe correspondent à l'implémentation de contraintes spécifiques et utilisent donc tous le module des contraintes et celui des variables :

- les contraintes arithmétiques (`Arith`) ;
- la réification de contraintes et les opérateurs logiques sur les contraintes (`Reify`) ;
- les contraintes globales « tous différents » (`Alldiff`), de cardinalité généralisée (`Gcc`), de tri (`Sorting`) et d'indexation dans un tableau (`FdArray`) — ces trois dernières contraintes dépendent également du module des contraintes arithmétiques pour poser des égalités entre variables.

Le module des invariants (`Invariant`, en haut à droite dans la figure 5.1), que l'on n'a pas encore évoqué car il est orthogonal au paradigme de la programmation par contrainte, implémente des références backtrackables reliées par des équations fonctionnelles (directionnelles, contrairement aux contraintes). Il dépend donc du module de la pile pour les références backtrackables et il utilise le mécanisme d'événements des contraintes pour les mettre à jour. Il dépend en fait également des variables (mais le lien n'apparaît pas sur la figure 5.1 pour des raisons de clarté) car il propose des primitives pour obtenir des invariants associés aux différentes caractéristiques des variables (minimum, maximum, taille etc.), afin de pouvoir, par exemple, maintenir facilement et efficacement des critères sur l'état des variables du problème pendant la recherche.

Notons encore que les modules qui implémentent les variables et les contraintes sur les ensembles ne figurent pas sur le schéma pour ne pas le surcharger, car leur place au sein de l'architecture de FaCiLe est similaire à celles des modules sur les variables entières.

5.1.2 Foncteurisation des variables

Les langages de la famille de ML tels qu'OCaml (et Haskell, SML) possèdent un système de module puissant doté de *foncteurs*. Un module regroupe au sein d'une structure une

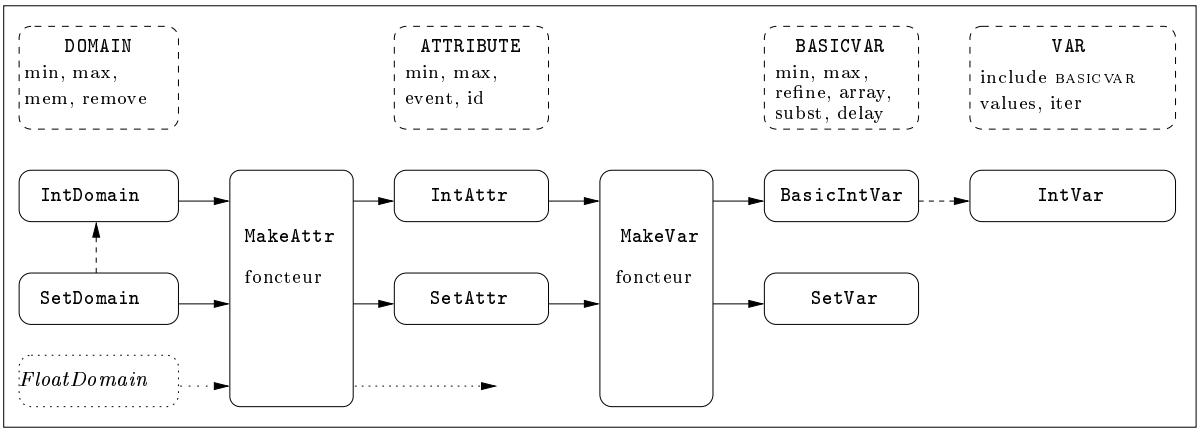


FIG. 5.2 – Foncteurisation des variables

collection de types, de données et de fonctions[2] ; il est associé à une signature qui représente son type et permet de restreindre son interface pour abstraire les types qu'il exporte et/ou en cacher d'autres. Un foncteur est une fonction des modules vers les modules qui permet de prendre en argument un module d'un type donné[3] et d'utiliser les données et les fonctions qu'il exporte ; la signature du résultat peut elle aussi être restreinte pour abstraire l'implémentation du foncteur. Les foncteurs permettent ainsi une forme de généricité de masse et d'abstraction dotée d'un typage statique très sûr. Cependant, un tel système de type est moins souple que le paradigme de la programmation orientée objet (que propose également OCaml) qui permet de définir incrémentalement une structure par héritage et redéfinition des méthodes héritées.

FaCiLe utilise des foncteurs pour implémenter différents types de variables logiques, actuellement des variables entières et des variables d'ensembles, mais la généricité du mécanisme permet d'implémenter n'importe quel type de variable nécessitant la même signature. Le principe de cette implémentation est illustré par la figure 5.2 où les foncteurs sont représentés par les grandes boîtes verticales, leurs arguments et le résultat de leur application, qui sont des modules, par les petites boîtes horizontales et leur type apparaissent en haut dans les boîtes en pointillés ; les flèches illustrent l'application des foncteurs à leurs arguments.

Le premier foncteur utilisé, `MakeAttr`, prend en argument un module de type `DOMAIN` qui doit fournir dans son interface les type (abstraits) des domaines et de leurs éléments, ainsi que les fonctions d'accès au domaine (*min*, *max* etc.), et renvoie un module de type `ATTRIBUTE` qui contient par exemple le type des événements manipulé par l'utilisateur et une fonction permettant de connaître le nombre de contraintes attachées à un attribut. Suivant le même principe, un deuxième foncteur `MakeVar` est appliqué au résultat du

[2]Les modules d'OCaml peuvent en fait contenir n'importe quel objet tel que modules, foncteurs, types de modules ou foncteurs (signatures) et classes.

[3]La signature du module effectivement pris en argument par le foncteur peut être plus générale que celle fournit par le type du foncteur (polymorphisme, données et fonctions supplémentaires).

premier pour construire un module de type `BASICVAR` offrant une interface riche pour manipuler les variables logiques. FaCiLe applique ainsi successivement aux modules qui implémentent les domaines entiers (`IntDomain`) et les domaines d'ensembles (`SetDomain`) le foncteur `MakeAttr`, ce qui génère les modules `IntAttr` et `SetAttr`, puis le foncteur `MakeVar` pour construire les modules `BasicIntVar` et `SetVar`. Ce mécanisme permet de factoriser le code des attributs et des variables en gardant un niveau d'abstraction élevé, car la signature du résultat des foncteurs est elle-même restreinte dans l'interface fournit à l'utilisateur. Il pourrait également servir à implémenter d'autres types de variables comme le suggère dans la figure 5.2 l'hypothétique module de domaines sur les flottants `FloatDomain`.

Cependant, certaines fonctions utiles pour des variables entières n'ont pas forcément d'intérêt pour les autres types de variables, comme l'énumération des valeurs de son domaine. C'est pourquoi le module `BasicIntVar` est enrichi par le module `IntVar` qui réalise une simple inclusion du premier en y ajoutant ces fonctions.

5.1.3 Interface

Uniformité

Les modules de FaCiLe sont en premier lieu des unités conceptuelles, c'est-à-dire qu'ils définissent essentiellement une notion à laquelle est associé un type et les opérations sur les objets de ce type. Autant que possible, l'interface de ces modules est uniforme : le type principal est nommé `t`, la fonction de création `create` ou encore l'affichage `fprint`. La notation qualifiée par le nom du module permet d'éliminer les conflits de nommage. Ce principe est étendu à des sous-modules des modules principaux qui structurent plus profondément encore FaCiLe en fournissant toujours une interface uniforme (e.g. variables à domaines entiers / variables d'ensembles, itérateurs sur les listes / les tableaux etc.).

Abstraction

Comme mentionné dans la section 5.1.2, les signatures des modules, qui constituent l'interface, permettent de restreindre les objets et fonctions exportées ainsi que d'abstraire les types des données. L'abstraction combinée au typage assurent que les données sont créées et manipulées en préservant leur intégrité. Par exemple, la création d'une variable avec un domaine vide génère immédiatement un échec car l'utilisateur n'a pas directement accès à la représentation concrète des variables et doit passer par la fonction `create` qui effectue des vérifications (sur des propriétés invariantes des données). Également grâce au typage et au polymorphisme, le module des invariants ne permet à l'utilisateur de modifier que les références générées par la fonction `create`, car celles qui sont le résultat d'une fonction doivent être mises-à-jour automatiquement (lorsque les références arguments de la fonction ont elles-mêmes été changées). Enfin, l'abstraction des types permet de modifier facilement l'implémentation des objets de manière transparente pour l'utilisateur, et donc sans que les programmes écrits avec la librairie n'aient besoin d'être modifiés.

Empaquetage

FaCiLe, telle qu'elle est présentée à l'utilisateur, est en fait empaquetée dans un « super-module » qui permet de séparer finement la signature interne destinée au développement de la signature externe fournit à l'utilisateur. Cette approche permet également de mieux appréhender les conflits de nommage si d'autres librairies sont utilisées simultanément. D'autre part, pour que l'utilisation de librairie reste souple, FaCiLe est dotée d'un module qui regroupe les fonctions les plus couramment utilisées et les opérateurs infixes (arithmétiques et logiques), de telle manière que l'ouverture de ce module permet d'utiliser directement les idiomes les plus fréquents (i.e. sans utiliser la notation préfixée par le nom du module).

Arguments optionnels

L'utilisation d'une librairie de PPC est parfois assez proche de celle d'un langage de modélisation, et l'utilisateur n'est pas forcément un spécialiste ou désire tout simplement aller vite pendant la phase de prototypage. Pour obtenir un certain degré de souplesse dans l'écriture d'un programme, FaCiLe fait donc généreusement appel aux arguments optionnels permis par OCaml. Ainsi, le degré de filtrage de certaines contraintes globales peut-être finement paramétré, mais FaCiLe fournit des arguments par défaut pour permettre à l'utilisateur de raffiner son code par étapes successives et se concentrer sur le modèle lors de la phase d'écriture. De même, l'identification explicite des objets (par une chaîne de caractères) est facultative et FaCiLe génère automatiquement un nom si nécessaire.

5.2 La pile

La recherche non-déterministe est implémentée dans FaCiLe de façon classique à l'aide d'une pile. Le module `Stack` fournit une pile (unique[4]) globale et abstraite : l'implémentation est cachée à l'utilisateur qui n'y accède qu'à l'aide de fonctions fournies dans l'interface du module. Les opérations principales sur la pile exportées par le module sont la sauvegarde de point de choix, le backtrack (ou retour arrière) et la coupure. Lors de l'exécution d'un but, le module `Goals` assure la continuation de succès à l'aide d'une liste de buts, et la pile gère la continuation d'échec : elle sauvegarde les points de choix et effectue les backtracks grâce au *trailing* de clôtures (fonctions accompagnées de leur environnement) chargées de défaire les modifications subies par les données.

Cette pile est structurée en *niveaux* : un niveau correspond à l'empilement d'un point de choix et contient la continuation d'échec implémentée avec une liste de buts ainsi qu'un lien vers le niveau précédent utilisé quand le niveau est *coupé*. Entre ces niveaux, la pile contient une traînée ou *trail* qui correspond aux modifications à défaire. Cette traînée

[4]Le module de la pile n'est pas réentrant à l'heure actuelle. Cependant, le mécanisme de foncteurisation illustrée dans la section 5.1.2 devrait permettre d'en créer un nombre quelconque, ce qui serait comparable à de multiples *managers* avec la librairie ILOG Solver, mais avec un typage en assurant une utilisation correcte.

est implémentée comme une liste de clôtures, ce qui fournit un mécanisme très général : n'importe quelle fonction peut être « sauvegardée ». Cette implémentation peut paraître trop coûteuse car la plupart des traînées sont de simples rétablissements de pointeurs mais le *profiling* (temporel) de l'exécution d'un programme en FaCiLe ne montre pas de pénalités liées à cette technique. En revanche, ce type de *trailing* est assez gourmand en mémoire, bien qu'une partie des modifications de domaines (qui sont fonctionnels) permet de les partager.

Cette pile est implémentée avec les types concrets suivants :

$$\textit{type}\ \text{continuation} = \left|\begin{array}{l}\text{Alive}\quad \textit{of}\quad \alpha\ \text{list}\\ \text{Coupé}\end{array}\right.$$

$$\textit{type}\ \text{pile} = \left|\begin{array}{lll}\text{Vide}\\ \text{Niveau} & \textit{of} & \text{niveau}\\ \text{Trail} & \textit{of} & \text{undo} * \text{pile}\end{array}\right. \qquad \textit{and}\ \text{niveau} = \left\{\begin{array}{ll}\text{date :} & \text{date}\\ \textit{mutable}\ \text{ou :} & \text{continuation}\\ \text{dépile :} & \text{pile}\\ \text{précédent :} & \text{pile}\end{array}\right\}$$

où *undo* est le type unit → unit qui correspond à une clôture sans paramètre et qui ne réalise que des effets de bord, et *date* est un entier (abstrait) qui sert au *time-stamping*. Le type *continuation* correspondant au champ *ou* est un type « somme[5] » constitué soit d'une liste polymorphe utilisée dans FaCiLe avec des buts (le constructeur est alors *Alive*) soit la constante *Coupé* indiquant que le point de choix a été supprimé. Une pile est donc soit vide (à l'initialisation), soit un niveau correspondant à un point de choix, soit une traînée qui contient la fonction appelée pour défaire les modifications lors d'un backtrack. Le champ *dépile* d'un niveau contient simplement la pile au moment où le point de choix a été empilé et permet de la rétablir après un backtrack. Enfin, le champ *précédent* indique le niveau précédent pour pouvoir y accéder directement si le niveau courant a été coupé. Le champ *ou* est le seul modifiable (*mutable*) car sa valeur doit être changée à *Coupé* si le niveau a été coupé.

5.2.1 Point de choix et backtrack

Quand un point de choix est empilé, un niveau est créé avec une nouvelle date et on sauvegarde dans le champ *ou* la liste de buts correspondant à la continuation d'échec ; le champ *dépile* est alors lié au niveau ou à la traînée située au sommet de la pile et *précédent* pointe sur le dernier niveau rencontré (disponible par l'intermédiaire d'une variable globale). La figure 5.3 illustre ce fonctionnement lors de l'empilement d'un point de choix correspondant

[5]Dans le système de type de ML, on appelle *produit* un type correspondant à un tuple, e.g. :

```
type data = int * float * string
```

et *somme* (ou *variant* en anglais) un type dont les valeurs peuvent prendre des formes différentes, e.g. :

```
type arbre = Feuille of data | Noeud of int * arbre
```

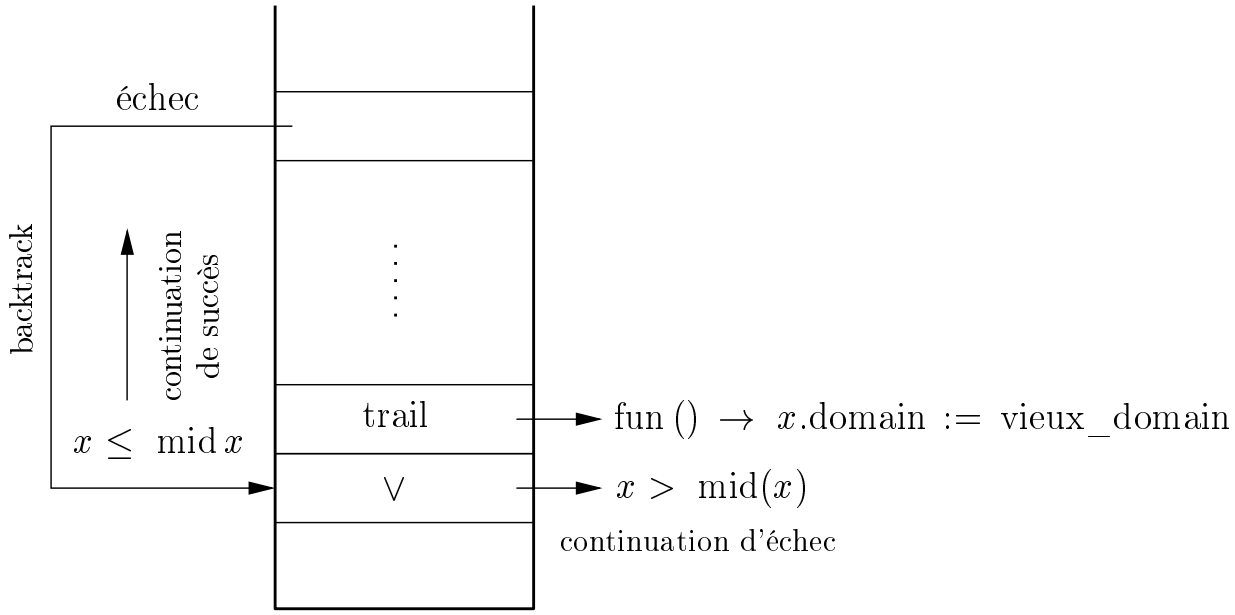


FIG. 5.3 – Empilement d'un point de choix lors de l'exécution d'un but d'exploration dichotomique

à l'exécution d'un but d'exploration dichotomique du domaine d'une variable :

$$\text{dichotomic}(x) :- (x \leq \text{mid}(x) \vee x > \text{mid}(x)) \wedge \text{dichotomic}(x)$$

$\text{mid}(x)$ étant défini par $\frac{\max(x)+\min(x)}{2}$.

Les raffinages de variables suscitées par la contrainte $x \leq \text{mid}(x)$ sont stockées dans la pile au dessus du niveau : sur le schéma, seule la clôture permettant de rétablir le domaine de x est montrée. Quand un échec survient, le contrôle de l'exécution des buts déclenche le backtrack qui appelle successivement les clôtures traînées en sautant les éventuels points de choix coupés. Quand le précédent point de choix est atteint[6], la pile est rétablie et la continuation d'échec est retournée. Le contrôle des buts remplace alors sa continuation de succès courante par la continuation d'échec et relance l'exécution des buts.

5.2.2 Références polymorphes backtrackables

Le module de la pile fournit des *références backtrackables* polymorphes exportées avec un type abstrait (type α ref) ainsi que les fonctions de création, affectation et déréférencement. L'affectation utilise la pile pour traîner la modification avec une clôture rétablissant simplement son ancienne valeur comme l'illustre la figure 5.3 dans le cas d'une référence

[6] On suppose qu'aucun autre niveau n'a été créé entre celui du but d'exploration dichotomique et l'échec qui déclenche le backtrack.

sur le domaine d'une variable. Ce mécanisme très général permet de faire « vivre » avec le backtrack n'importe quelle structure de donnée fonctionnelle (i.e. qui ne peut pas être modifiée en place) dans un programme utilisant FaCiLe.

Pour limiter le nombre de modifications traînées, l'affectation utilise la technique du time-stamping [Aggoun 90]. L'affectation marque ces références avec la date du niveau courant, et lors de la prochaine affectation, cette date est comparée avec celle du niveau actuel : si elle est identique ou postérieure, i.e. aucun nouveau point de choix n'a été empilé (ou il a été coupé), la modification n'a pas besoin d'être traînée.

5.3 Domaines

Les domaines finis de FaCiLe sont des listes d'intervalles fonctionnelles qui peuvent être partagées. Toutes les fonctions de manipulation des domaines sont entièrement écrites en OCaml, contrairement à la plupart des solveurs fondés sur un langage de haut niveau (p.ex. Prolog) qui appellent des routines écrites en C/C++ car leurs performances en rapidité d'exécution sont trop faibles.

Des listes d'intervalles Les domaines de FaCiLe sont implémentés par des listes d'intervalles pour avoir une représentation économique en mémoire, contrairement à la structure de tableau de bits souvent utilisée dans les systèmes de PPC. Cette représentation présente cependant l'inconvénient d'avoir une complexité linéaire en pire cas pour les opérations d'appartenance ou de retrait de valeur. Une structure d'ensemble avec des complexités logarithmiques pour ces opérations aurait également pu être choisie, mais les performances en pratique de la représentation par liste étaient supérieures pour notre jeu de tests. En effet, cette structure est très légère — les constantes multiplicatives sont faibles dans l'expression de leur coût en temps de calcul — et en adéquation avec les techniques de consistance arithmétique dont la plupart sont des approximations de l'arc consistance sur les *bornes* des domaines, ce qui les réduit la plupart du temps à des intervalles. De plus, le maximum d'un domaine et son cardinal sont maintenus à chaque manipulation pour accélérer l'accès à ces informations qui serait sinon systématiquement linéaire en temps.

Une structure de données fonctionnelle Cette représentation par listes au sein d'un langage fonctionnel nous a naturellement conduit à utiliser une structure elle aussi fonctionnelle. Les domaines de FaCiLe ne peuvent donc pas être modifiés « en place » et les fonctions qui les manipulent renvoient sytématiquement un nouveau domaine. Cette propriété permet de simplifier le contrôle du *trailing*, c'est-à-dire la technique utilisée par FaCiLe pour implémenter le backtrack lors de la recherche de solution (cf. section 51), en garantissant l'intégrité des données. D'autre part, de telles structures fonctionnelles peuvent être partagées, ce qui rend certaines opérations de réduction peu coûteuses en mémoire comme l'illustre la figure 5.4 : le domaine d_1 est réduit à d'_1 en lui supprimant les valeurs inférieures à 1.

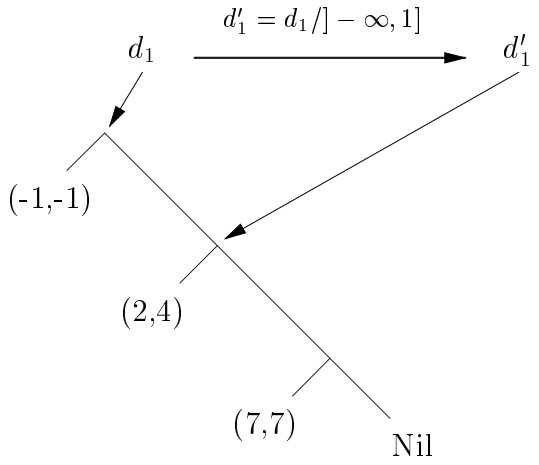


FIG. 5.4 – Partage des domaines

De même, plusieurs variables peuvent partager un même domaine lors de leur création. Leur modification ultérieure suscitera la recopie du domaine si elle est nécessaire. Par exemple, dans la figure 5.5, deux variables v_1 et v_2 partagent le même domaine initial $\{0; 4; 5; 6; 10\}$ jusqu'à l'ajout de la contrainte $v_1 < v_2$; un nouveau domaine est alors créé pour v_1 tandis que v_2 est toujours associé à une partie du domaine initial.

Domaine d'ensembles Les domaines d'ensembles sont inspirés de la librairie Conjunto [Gervet 97] disponible dans le système ECL^{i}PS^e. Les domaines sont représentés par leur plus grande borne inférieure ou *glb* (*greatest lower bound*) et leur plus petite borne supérieure ou *lub* (*lowest upper bound*) au sens de l'inclusion, qui correspondent à un treillis d'ensembles. La figure 5.6, où les arcs descendants sont les inclusions, représente l'ensemble (le treillis) des éléments contenus dans le domaine $sd = [\{1, 2\}, \{1, 2, 3, 4, 5\}]$ défini par :

$$s \in sd \Rightarrow s \supseteq \{1, 2\} \text{ et } s \subseteq \{1, 2, 3, 4, 5\}$$

Avec cette structure de données pour les domaines d'ensembles, seuls des treillis ou « intervalles » complets peuvent être représentés. Les « trous », qui apparaissent si des valeurs du treillis strictement supérieures au *glb* ou strictement inférieures au *lub* (au sens de l'inclusion) n'appartiennent pas au domaine initial d'une variable, ne seront donc gérés que par l'intermédiaire de contraintes unaires sur les variables.

Les bornes de ces domaines sont implémentées à l'aide des domaines entiers de FaCiLe qui se sont avérés plus performants en pratique qu'une structure d'ensemble avec des opérations de complexité théorique logarithmique.

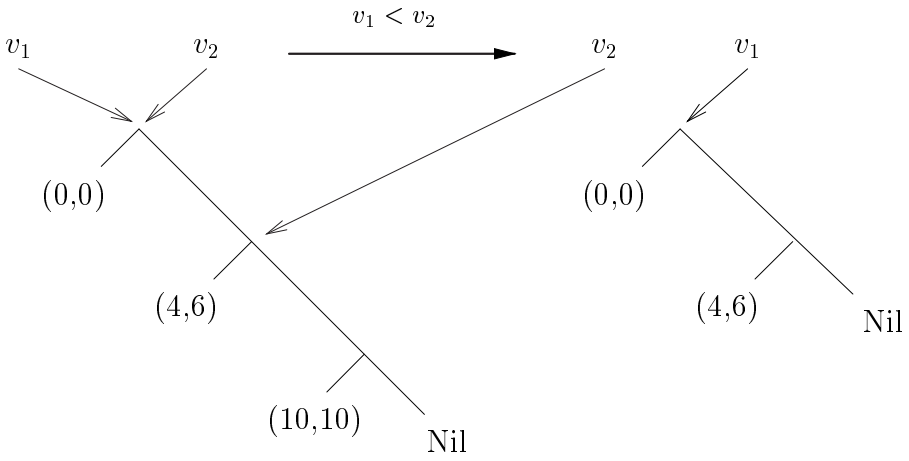


FIG. 5.5 – Partage de domaines par les variables

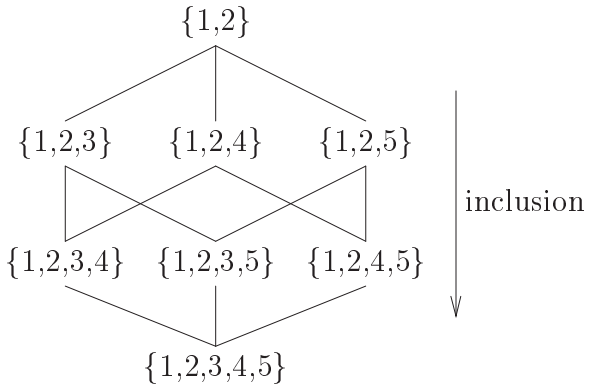


FIG. 5.6 – Treillis d'un domaine d'ensembles

5.4 Variables à attribut

Les variables de FaCiLe sont des références backtrackables (cf. section 5.2.2) sur une structure qui peut se trouver dans deux états mutuellement exclusifs :

- soit *instanciée* à une valeur qui est un élément de son domaine initial ;
- soit *inconnue* avec un domaine (de taille supérieure à 1) et des contraintes attachées.

On appelle *attribut* la structure associée à une variable inconnue. Cette représentation des variables est inspirée de [Le Huitouze 90] et de celle du système ECL^{i}PS^e [ECL^{i}PS^e 01]. Dans notre implémentation, l'attribut d'une variable contient toutes les données nécessaires à son traitement lorsqu'elle est inconnue :

- une référence backtrackable sur le domaine ;
- des références backtrackables sur les listes de contraintes (à réveiller) associées à chaque événement déclenché par les réductions du domaine ;
- un identifiant (entier) unique ;
- un nom défini par l'utilisateur (chaîne de caractères).

Quand une variable devient instanciée, elle perd son attribut et donc les données contenues dans celui-ci. Le glaneur de cellules peut ainsi récupérer l'attribut si aucun point de choix ne précède l'instanciation.

Ces deux états mutuellement exclusifs sont implémentés par un type somme :

```
type variable = Unknown of attribute | Value of element
```

Les erreurs éventuelles de traitement — considérer qu'une variable est instanciée alors qu'elle est encore inconnue — sont ainsi systématiquement détectées à la compilation.

5.5 Contraintes

Les contraintes de FaCiLe sont des structures créées à l'aide de primitives ou entièrement définies par l'utilisateur et que l'on peut manipuler et combiner (dans le cas de réifications) avant de les « poser », c'est-à-dire de les rendre actives — i.e. capables de réduire les domaines des variables impliquées. Elles consistent essentiellement en une collection de fonctions :

- **init** : première fonction appelée à la pose de la contrainte ;
- **delay** : enregistrement auprès des événements ;
- **update** : filtrage des domaines et détection d'échec (i.e. d'inconsistance) suite à un événement ;
- **check** : vérification de la satisfaction ou de la violation de la contrainte (sans réduction de domaine) ;
- **not** : négation de la contrainte pour la réification ;
- **fprint** : affichage.

Les contraintes contiennent également des données telles que :

- leur **priorité** parmi trois différentes (*immédiate*, *normale*, *tardive*) ;
- leur **état** (booléen) : résolue ou non ;

- leur **activité** (booléen) : déjà placée dans une file de réveil ou non ;
- leur **nom** (chaîne de caractères) et un **identificateur** unique (entier généré automatiquement).

Lors de la création d'une contrainte, son *état* et son *activité* sont initialisés à *faux* (non-résolue et en attente).

5.5.1 Pose des contraintes

Lorsqu'une contrainte est posée, la fonction *init* est tout d'abord appelée pour initialiser les structures de données internes de la contrainte ainsi que pour effectuer les réductions de domaine initiales (si nécessaires) en appelant éventuellement *update*. Si la contrainte n'est pas déjà résolue à ce stade, sa fonction *delay* est appelée pour l'enregistrer auprès des événements spécifiés : à chaque variable sur laquelle la contrainte est attachée, on associe une liste d'événements (cf. section 5.5.3) sur lesquels elle doit être réveillée, e.g. $([\min;\max], v)$ si elle doit être réveillée sur la modification du minimum ou du maximum de la variable v.

Optionnellement, un identificateur (entier) peut également être associé à une variable attachée, par exemple son indice dans un tableaux de variables $([\text{subst}], i, v_i)$ (l' événement est ici associé à l'instanciation de la variable v_i) ; cet identificateur, qui sert à fournir de l'information sur l'événement qui a suscité le réveil, est passé en argument de la fonction *update* lors de chaque réveil.

Par ailleurs, la pose d'une contrainte enregistre celle-ci dans le « tas » des contraintes non-résolues. Ce tas est un ensemble de pointeurs « faibles » sur les contraintes, c'est-à-dire que ces pointeurs ne sont pas des racines d'accès pour le récupérateur de mémoire. Ainsi, ce dernier peut récupérer l'espace occupé par une contrainte dès qu'elle n'est plus attachée à aucune variable, par exemple après un backtrack si elle est posée dynamiquement ou quand les variables sur lesquelles elle porte sont instanciées et perdent leur attribut.

Ce tas permet par exemple de détecter s'il reste des contraintes non-résolues après le calcul d'une solution pour laquelle les variables auxiliaires n'auraient pas toutes été instanciées (*floundering*). Le tas de contraintes et la pile constituent un *état global* du système : une recherche pourrait être interrompue puis reprise si cet état était stocké au moment de l'interruption — cette fonctionnalité n'a pas été encore implantée dans FaCiLe.

5.5.2 Réveil des contraintes

Quand le domaine d'une variable est réduit, les contraintes non-résolues attachées aux événements déclenchés sont placées dans l'une des trois files d'attente de FaCiLe suivant leur priorité. On vérifie cependant que les contraintes n'avaient pas déjà été réveillées (introduites dans une file) avec le même identificateur d'événement pour éviter de multiples réveils par la même cause. Une fois placée dans la file d'attente, la contrainte est marquée (son champ *activité* est positionné à *vrai*) pour indiquer qu'elle est déjà réveillée.

Les files d'attente sont ensuite vidées l'une après l'autre dans l'ordre croissant de leur priorité. Quand une contrainte est retirée de sa file d'attente, sa fonction *update* est appelée

en lui passant en argument l'identificateur si elle est toujours non-résolue. Cette fonction filtre les domaines des variables et renvoie un booléen indiquant si la contrainte a été résolue ou non, ce qui permet de mettre à jour l'état de la contrainte. Enfin, l'activité de la contrainte est remise à « en attente ».

Les variables dont le domaine a été réduit par une contrainte lors de cette phase déclenchent des événements qui placent de nouveau des contraintes dans les files d'attente. C'est la file non-vide de priorité la plus importante qui devient alors dynamiquement la file courante à vider. Le point fixe est atteint lorsque toutes les files sont vides.

5.5.3 Événements génériques

Le mécanisme d'événements utilisé pour associer les contraintes aux raffinages des variables est en fait générique. L'utilisateur peut en définir de nouveaux pour y attacher des « objets réactifs » implémentés par des contraintes qui perdent alors leur sémantique classique dans le cadre de la PPC car n'importe quelle fonction peut être appelée par l'*update* d'une contrainte.

Un événement (type abstrait `event` dans l'interface de FaCiLe) est implémenté par une référence backtrackable sur une liste de contraintes (chacune pouvant être associée à un identificateur qui sera passé au *callback*, i.e. la fonction `update`, de la contrainte). Cette liste de contraintes qui peut être modifiée dynamiquement (et qui est rétablie avec le backtrack) correspond aux objets réactifs à réveiller quand l'événement est déclenché. Le déclenchement d'un événement parcourt simplement la liste de contraintes pour les placer dans la file d'attente de priorité adéquate si elles ne s'y trouvent pas déjà (*activité* de la contrainte) et si elle n'ont pas été désactivées (*état* de la contrainte).

5.5.4 Réification

Les contraintes de FaCiLe peuvent être réifiées, c'est-à-dire qu'une variable logique booléenne (une variable standard de domaine $\{0, 1\}$) est contrainte à prendre la valeur 1 si et seulement si la contrainte est satisfaite (et 0 si et seulement si elle est violée). Les contraintes réifiées ne sont pas posées mais passées en argument de la contrainte primitive de réification ou d'un opérateur logique.

La fonction *check* de la contrainte réifiée est alors utilisée par la contrainte de réification pour vérifier sa satisfaction sans réduire de domaines ni signaler d'échec. *check* doit renvoyer un booléen qui indique si la contrainte est satisfaite ou violée et lever l'exception *DontKnow* si elle ne peut pas le déterminer. La variable booléenne est instanciée à 1 si *check* renvoie *vrai* et à 0 si elle renvoie *faux*. Réciproquement, si la variable booléenne est instanciée à 1, la contrainte est alors posée, et si elle est instanciée à 0, c'est la négation de la contrainte, obtenue grâce à la fonction *not*, qui est posée.

La contrainte de réification se sert également de la fonction *delay* de la contrainte réifiée pour spécifier ses conditions de réveil, et optionnellement de la fonction *delay* de la négation de la contrainte : si les deux ensembles de conditions de réveil sont différents, la réification

sera réveillée plus fréquemment mais la violation éventuelle de la contrainte réifiée pourra être détectée plus tôt.

5.5.5 Arguments optionnels

Les fonctions destinées à la réification, *check* et *not*, sont des arguments optionnels de la création de contraintes. En effet, les réifications portent essentiellement sur des contraintes arithmétiques pour la modélisation de problèmes d'optimisation, et il n'y aurait guère d'intérêt à réifier certaines contraintes comme « tous différents » ou la contrainte de cardinalité globale, ni à écrire la négation de ces contraintes. Une contrainte peut donc être créée en omettant les fonctions *check* et *not* qui sont alors remplacées par la levée d'une exception (« erreur fatale ») déclenchée si on essaye de réifier la contrainte.

De même, les fonctions et données suivantes sont des arguments optionnels de la création de contrainte :

- *init* appelle *update* par défaut pour filtrer les domaines à la pose de la contrainte ;
- la *priorité* par défaut est *normale* ;
- le *nom* par défaut est *anonymous* ;
- *fprint* affiche par défaut le nom de la contrainte.

Seules les fonctions *update* et *delay* sont donc nécessaires pour créer une contrainte.

5.5.6 Contraintes arithmétiques

Contraintes linéaires et non-linéaires

Bien que cela soit transparent pour l'utilisateur, FaCiLe possède deux types de contraintes arithmétiques de conception très différentes :

- Les équations, inéquations et diséquations sur des sommes globales :

$$\sum_{i=1}^{k} c_i v_i \quad \begin{matrix} \leq \\ = \\ \neq \end{matrix} \quad c$$

  où les v_i sont des variables, et c et les c_i des constantes.
- Les contraintes binaires ou ternaires d'égalité avec un opérateur non-linéaire :

$$\begin{aligned} z &= x \text{ op } y \quad \text{avec op} \in \{\times, /, \text{mod}\} \\ z &= |x| \\ z &= x^e \end{aligned}$$

  avec x, y et z des variables et e une constante positive.

L'interface du module arithmétique ne permet en fait d'accéder qu'aux contraintes linéaires. Dès qu'un opérateur non-linéaire apparaît dans une expression, une variable intermédiaire interne est créée, et c'est cette dernière qui sera utilisée au sein de la contrainte de somme globale. Ainsi, la contrainte

$$4y + 10x^2t \geq 2900 \tag{5.1}$$

génère deux variables intermédiaires v_1 et v_2 et pose les contraintes suivantes :

$$\begin{cases} v_1 = x^2 & \text{exponentiation} \\ v_2 = v_1 * t & \text{produit} \\ 4y + v_2 \geq 2900 & \text{somme globale} \end{cases}$$

De même, les opérandes des contraintes non-linéaires doivent être des variables. Si des expressions « non-terminales » (i.e. qui ne sont pas des variables) apparaissent dans une contrainte non-linéaire, des variables intermédiaires sont également créées. Ainsi, une contrainte linéaire dans laquelle apparaît l'expression

$$exp_1 \text{ op } exp_2$$

où op est un opérateur non-linéaire et exp_1, exp_2 des expressions non-terminales, génère trois variables v_1, v_2, v_3 et les contraintes

$$\begin{cases} v_1 = exp_1 \\ v_2 = exp_2 \\ v_3 = v_1 \text{ op } v_2 \end{cases}$$

v_3 remplacera alors l'expression dans la contrainte linéaire.

Normalisation des expressions arithmétiques

Au moment où une contrainte arithmétique est créée par l'utilisation d'un opérateur de relation (in/dis/équation), les expressions en opérandes sont « normalisées » pour les simplifier. Ainsi, les variables peuvent être disséminées au sein des expressions sans entraîner de perte d'efficacité en temps, en mémoire et en puissance de filtrage. Par exemple, l'inégalité suivante :

$$3y + 2xt \times 5x + y \geq 2900$$

générera la même contrainte que la contrainte 5.1. FaCiLe simplifie donc les expressions en « factorisant » les variables identiques qui apparaissent dans les sommes et les produits :

$$\sum_{i=1}^{k} c_i v_i \quad \Leftrightarrow \quad \sum_{i \in \phi([1,k])} \left(\sum_{j \in [1,k] \,|\, v_j = v_i} c_j \right) v_i$$

$$\prod_{i=1}^{k} v_i^{e_i} \quad \Leftrightarrow \quad \prod_{i \in \phi([1,k])} v_i^{\sum_{j \in [1,k] \,|\, v_j = v_i} e_j}$$

ϕ étant une fonction qui calcule un sous-ensemble (de cardinal $|\{v_1, \ldots, v_k\}|$) des indices $[1, k]$ tel que chaque variable n'apparaisse qu'une seule fois. Les variables identiques sont donc regroupées en introduisant si possible de nouveaux coefficients ou des contraintes d'exponentiation au lieu de produits simples.

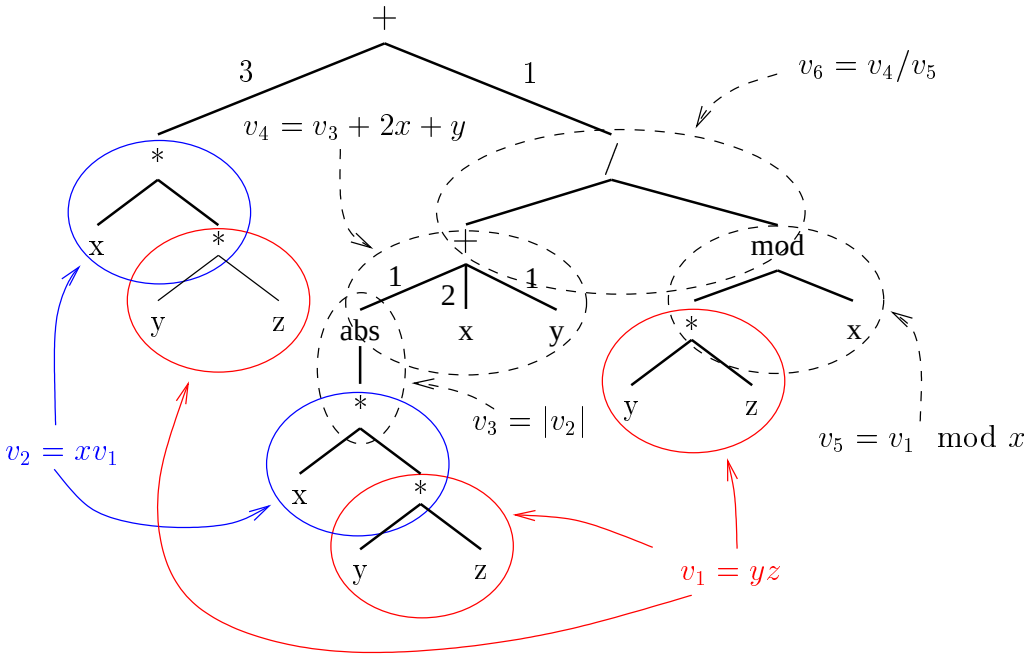


FIG. 5.7 – Réutilisation des variables intermédiaires

Réutilisation des variables intermédiaires

Pour éviter de générer plusieurs fois la même variable intermédiaire, FaCiLe vérifie si elle a déjà été créée et le cas échéant la réutilise pour améliorer l'efficacité en temps et en mémoire des contraintes arithmétiques. Par exemple, à la création de la contrainte

$$3xyz + \frac{|xyz| + 2x + y}{yz \mod x} = 128$$

FaCiLe ne génère que 6 variables intermédiaires au lieu de 9, comme l'illustre la figure 5.7 où l'expression arithmétique est représentée sous forme d'arbre dont les nœuds sont étiquetés par des opérateurs et les feuilles par des variables (de plus, les arêtes filles des sommes globales sont étiquetées avec le coefficient associé à chaque sous-expression). La variable v_1 est ainsi utilisée trois fois et la variable v_2 deux fois. Les autres variables intermédiaires correspondent à des sous-expressions qui n'apparaissent qu'une seule fois et ne sont donc pas réutilisées.

Détection automatique des contraintes linéaires booléennes

Quand une contrainte linéaire est créée, FaCiLe détecte automatiquement si des parties de l'expression sont des sommes de booléens. Les expressions linéaires sont transformées en

$$\sum_i a_i \sum_{j_i} b_{j_i} \; + \; \sum_i c_i v_i$$

où les b_{j_i} sont des variables booléennes et les v_i des variables non-booléennes, puis tous les $a_i \sum_{j_i} b_{j_i}$ dont le nombre de termes est suffisamment important sont remplacés par une variable intermédiaire et traités spécifiquement avec des contraintes plus efficaces que la contrainte générale sur les expressions linéaires.

Dépassement des entiers

Les entiers standards d'OCaml sont codés sur 31 bits (ou 63 bits sur les processeurs à 64 bits) et toutes les opérations de l'arithmétique entière sont donc calculées modulo 2^{31} (ou 2^{63}) sans lever d'exception ou échouer si le plus petit ou le plus grand entier représentable est dépassé. Or, même si leurs solutions restent dans l'intervalle des entiers représentables, certaines contraintes arithmétiques nécessitent des calculs qui dépassent ces bornes, notamment à la pose de la contrainte alors que les domaines n'ont pas encore été réduits.

FaCiLe permet d'effectuer un contrôle optionnel du résultat de chaque opération arithmétique pour détecter ces dépassements et lever une exception le cas échéant. En effet, FaCiLe redéfinit les opérations arithmétiques avec des assertions qui peuvent disparaître en utilisant une option du compilateur. Le MAKEFILE fourni avec la distribution standard de FaCiLe propose par défaut de laisser le contrôle de dépassement pour la compilation en *bytecode* où les performances sont au second plan et de le supprimer lorsque la compilation s'effectue en code natif optimisé. Par exemple, le calcul de la borne supérieure de l'expression x^7 avec $x \in [1, 20]$, i.e. 20^7, est strictement supérieur au plus grand entier représentable (sur une machine 32 bits).

Pour éviter ces dépassements d'entiers, la contrainte de somme globale effectue une réduction précoce des domaines dans le cas où tous les termes sont de même signe *avant* d'évaluer les bornes de l'expression. Ainsi, le programme de *séquence magique* (cf. problème 4.1) inclus dans la distribution standard de FaCiLe permet de trouver une solution pour une taille supérieure à 4500 (si la mémoire du système est suffisante[7]), une taille bien supérieure à celles permises par les systèmes testés dans [Fernández 00].

5.5.7 Contraintes globales

Tous différents

La diséquation généralisée (« tous différents ») remplace $\frac{n(n-1)}{2}$ contraintes entre chaque paire de variables d'un tableau $[v_1, \ldots, v_n]$:

$$\text{alldiff}\,[v_1, \ldots, v_n] \quad \Leftrightarrow \quad \{v_i \neq v_j,\ 1 \leq i < j \leq n\}$$

FaCiLe propose deux variantes de cette contrainte globale que l'on sélectionne grâce à un argument optionnel :

[7] La séquence magique de taille 4500 a été obtenue avec un PC sous Linux possédant 512 MO de RAM.

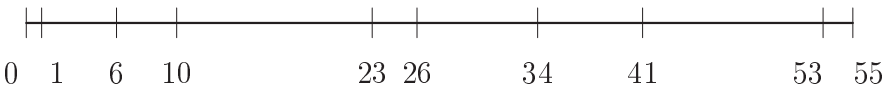


FIG. 5.8 – Règle de Golomb optimale à dix graduations.

- Une version « paresseuse » qui attend l'instanciation de chaque variable pour éliminer sa valeur des domaines des autres variables. Le filtrage est le même qu'avec les $\frac{n(n-1)}{2}$ contraintes binaires mais on ne pose qu'une seule contrainte réveillée par n événements différents (*subst*) associés à l'index de la variable correspondante passé en paramètre à *update*.
- Une version globale qui repose sur un algorithme de mariage dans le graphe bipartie des variables et des éléments de l'union de leur domaine pour détecter des échecs plus précocement que la version locale, ceci avec une complexité en $\mathcal{O}(n^{\frac{5}{2}})$ [Hopcroft 73]. Cette contrainte pose également la contrainte de la version simple, et peut se paramétrer pour être réveillée sur n'importe lequel des événements associés aux réductions de domaine, en pratique soit sur l'instanciation des variables (*subst*), soit sur leur modification (*refine*).

Le tableau 5.1 compare les performances des différents paramétrages de la contrainte « tous différents » sur le problème 5.1 de la règle de GOLOMB optimale (*Optimal Golomb Ruler*) dont les seules contraintes sont des diséquations :

Problème 5.1 (Règle de Golomb) *Une règle de Golomb de taille n est un ensemble de n entiers ou graduations $a_1 < \cdots < a_n$ tel que toutes les distances $a_j - a_i$, $1 \leq i < j \leq n$ entre deux graduations distinctes soient différentes. La règle de Golomb optimale est la règle de longueur minimale.*

Ce problème très difficile, qui a des applications en télécommunication, en radio-astronomie ou encore en cristallographie, n'a pu être résolu optimalement que jusqu'à 23 graduations à l'heure actuelle, les derniers résultats ayant été obtenus grâce à des systèmes qui distribuent les calculs sur des milliers de machines connectées à l'Internet[8]. Voir [Dollas 98] pour plus de détails sur les algorithmes utilisés pour résoudre le problème de Golomb, et [Smith 99b] pour les modélisations en programmation par contraintes.

La modèle utilisé ici pour obtenir les résultats du tableau 5.1 casse des symétries évidentes en posant $a_1 = 0$ (translation de la règle) et en ordonnant la « première » et la « dernière » distance $a_2 - a_1 < a_n - a_{n-1}$ (symétrie par rapport au centre de la règle). La contrainte « tous différents » est ensuite posée sur les $\frac{n(n-1)}{2}$ distances, ce qui équivaut à poser de l'ordre de $\mathcal{O}(n^4)$ diséquations. On cherche enfin la solution optimale qui minimise a_n à l'aide d'un but d'optimisation standard et une heuristique dynamique qui sélectionne la graduation qui a le plus petit domaine et le plus grand nombre de contraintes en cas d'égalité. La figure 5.8 montre une règle de Golomb optimale à dix graduations.

[8] `www.distributed.net/ogr`

TAB. 5.1 – Performances de la contrainte « tous différents » sur le problème de Golomb

taille	refine		subst		basique		$\mathcal{O}(n^4)$ $\neq$	
	CPU	bt	CPU	bt	CPU	bt	CPU	bt
5	0.00	10	0.00	13	0.00	15	0.00	15
6	0.02	34	0.02	38	0.02	55	0.02	55
7	0.04	227	0.04	240	0.07	323	0.13	324
8	0.40	1416	0.52	1537	0.61	2128	1.68	2124
9	3.87	8383	4.17	9667	4.85	13964	17.82	13956
10	24.79	34121	27.70	44098	39.60	79117	169.24	79132

Le tableau 5.1 présente le temps de calcul (en secondes) et le nombre de backtracks nécessaires pour obtenir la preuve d'optimalité du problème de Golomb de 5 à 10 graduations, sur un PC sous Linux à 1,7 GHz. Les deux premières paires de colonnes correspondent à la version sophistiquée de la contrainte « tous différents » réveillée respectivement sur modification (*refine*) ou sur instanciation (*subst*) de l'une des distances. La troisième paire de colonnes présente la version basique de la contrainte, et la dernière paire, le modèle équivalent avec des diséquations binaires. On peut noter que les réductions inférées par la contrainte basique sont identiques à celles réalisées par les diséquations binaires, mais beaucoup plus efficaces en temps de calcul car seule une contrainte réveillée sur $\mathcal{O}(n^2)$ événements est posée au lieu de $\mathcal{O}(n^4)$ diséquations. La version sophistiquée est en revanche beaucoup plus efficace en puissance de filtrage et le surcoût en temps dû à l'algorithme de mariage est largement compensé par la détection plus précoce des inconsistances. On constate également que la contrainte reste efficace quand on choisit le maximum de propagation en la réveillant à chaque modification, ce qui correspond au meilleur temps de calcul obtenu.

Cardinalité globale

FaCiLe fournit une contrainte de « cardinalité globale » (*Global Cardinality Constraint*, appelée aussi *distribute* dans ILOG Solver) [Régin 96]. Cette contrainte prend en argument un ensemble de variables $\{v_1, \ldots, v_n\}$ et un ensemble de couples $\{(c_1, val_1), \ldots, (c_m, val_m)\}$ où c_i est une variable correspondant au nombre d'occurrences de val_i dans $\{v_1, \ldots, v_n\}$:

$$\forall i \in [1, m] \quad c_i = |\{v_j = val_i,\ j \in [1, n]\}|$$

Cette contrainte utilise un algorithme de flot dans le graphe orienté des variables et des valeurs, les cardinaux correspondant à la capacité d'arêtes particulières. Un argument optionnel permet de régler la sophistication de l'algorithme pour augmenter sa rapidité au détriment du filtrage :

- *Basic* teste la consistance mais n'effectue pas de réduction de domaine ;
- *Medium* filtre en plus les variables ;

Taille	High	Medium	Réifiée
200	0.25	0.23	0.90
400	1.32	1.24	4.69
600	3.50	3.20	12.80
800	6.32	6.11	33.68
1600	30.28	28.07	—
3200	145.98	136.71	—
4500	308.47	283.66	—

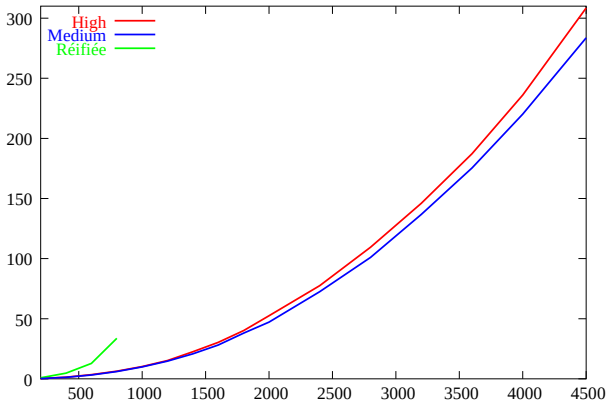


FIG. 5.9 – Performances de la contrainte de cardinalité globale sur le problème de séquence magique

– *High* (valeur par défaut) réduit aussi les cardinaux.

La contrainte de cardinalité globale (*gcc*) permet de modéliser élégamment et efficacement le problème de séquence magique 4.1. En effet, la spécification d'une séquence magique correspond exactement à cette contrainte appliquée aux variables de la séquence x_i et à l'ensemble de couples cardinal/valeur (x_i, i) :

$$\text{gcc}([x_1, \ldots, x_n], [(x_1, 1), \ldots, (x_n, n)])$$

Le tableau 5.9 résume les performances en temps de calcul (sur un PC sous Linux à 700 MHz) de la contrainte de cardinalité globale sur ce problème. En fonction de la taille de la séquence qui figure dans la première colonne, la deuxième colonne (*High*) correspond à la contrainte de cardinalité paramétrée avec le maximum de filtrage, la troisième (Medium) avec l'option intermédiaire (*Medium*) et la dernière à la version par contraintes réifiées du programme 4.1 ; notons également que ces temps ont été obtenus en ajoutant la contrainte redondante $\sum_{i=0}^{n-1} ix_i = n$ et en sélectionnant la variable de plus petit domaine en premier lors de la recherche.

L'option *Basic*, qui n'effectue pas de filtrage, n'y figure pas car elle n'est pas efficace (plus de 350 K backtracks et 215 s pour une séquence de longueur 100). En revanche, les trois versions présentées trouvent une solution avec seulement 7 backtracks pour *High* et *Réifiée* et 8 backtracks pour *Medium*, quelque soit la taille. Les résultats manquent au-dessus de la taille 800 pour la contrainte réifiée car la consommation mémoire est trop importante : n^2 contraintes réifiées et n sommes de n termes doivent être créées. On constate que les performances de la contrainte de cardinalité sont bien supérieures et qu'il n'y a que peu de différences entre « Medium » et « High », le surcroît de filtrage apporté par cette dernière étant trop coûteux pour ce problème.

Tri

FaCiLe implémente la contrainte globale de tri décrite dans [Bleuzen-Guernalec 97]. Elle contraint deux tableaux de variables de même taille en assurant que le deuxième contienne les mêmes éléments que le premier, mais triés par ordre croissant. De plus, l'algorithme de filtrage utilisé réduit les bornes des domaines des variables de manière optimale et avec une complexité optimale en $\mathcal{O}(n \log(n))$. FaCiLe est le premier solveur à proposer cette contrainte dans sa distribution standard. Elle a été utilisée pour modéliser et résoudre le problème d'allocation de créneaux présenté au chapitre 6.

Cette contrainte, définie dans le module `Sorting` par la fonction `Sorting.cstr`, prend deux tableaux de variables en argument et optionnellement le tableau de la permutation correspondant au tri.

5.6 Recherche

5.6.1 Les buts

Comme les contraintes, les buts de FaCiLe sont implémentés avec un type abstrait et consistent essentiellement en une fonction (une clôture) qui est appelée quand le but est activé. Cette fonction peut soit ne rien renvoyer, soit renvoyer un nouveau but qui sera exécuté après le succès du but courant ; elle peut également être récursive, c'est-à-dire faire un appel au but lui-même pour construire le résultat (le but renvoyé).

Les buts de FaCiLe sont construits à partir de tels buts définis par l'utilisateur ainsi que d'opérateurs logiques binaires (conjonction et disjonction) et d'itérateurs (conjonctifs ou disjonctifs) sur des structures de données (tableaux et listes). La programmation fonctionnelle permet de composer directement ces itérateurs polymorphes pour en obtenir de nouveaux : e.g. un itérateur sur des matrices en composant l'itérateur sur les tableaux avec lui-même : $matrix = array \circ array$, ou encore `let matrix g = array (array g)` avec la syntaxe concrète de ML, `array` étant l'itérateur sur les tableaux et `g` le but à appliquer à chaque élément de la matrice..

5.6.2 Le contrôle

Le contrôle de la recherche est implémenté au-dessus de la pile en codant la continuation de succès avec *une liste de buts*. Les buts de la continuation de succès sont appelés conjonctivement l'un après l'autre, en insérant éventuellement en tête de liste le nouveau but renvoyé par le but courant. Quand la liste de but a été vidée, une exception *Succès* est levée et la recherche se termine en renvoyant un booléen positionné à *vrai*.

Les échecs sont implémentés dans FaCiLe à l'aide du mécanisme natif d'exception ; ils sont en général levés par une contrainte ou par la réduction d'une variable avec un domaine vide. Si un échec survient pendant l'exécution d'un but, le contrôle décrit précédemment est interrompu et c'est *la pile* qui implémente alors la continuation d'échec : on backtracke

jusqu'au dernier point de choix en défaisant les modifications traînées et on rétablit la continuation de succès, i.e. une nouvelle liste de buts, sauvegardée dans ce niveau.

Lors de chaque backtrack, une fonction optionnelle définie par l'utilisateur est appelée avec le nombre de backtracks depuis le début de la recherche comme argument. De plus, un événement (implémenté avec le même mécanisme que ceux déclenchés par les réductions de domaine des variables) est déclenché pour pouvoir réveiller des « contraintes » (ou plutôt des « objets réactifs ») aux points de choix ; cette fonctionnalité est utilisée par l'un des buts d'optimisation de FaCiLe pour maintenir la borne supérieure du coût qui peut avoir été rétablie lors d'un backtrack.

Ce langage de buts de recherche et le contrôle associé sont hérités de la programmation logique et ont été conservés tels quels pour des raisons de simplicité et d'efficacité. Cependant, la recherche standard en profondeur d'abord peut être facilement modifiée. Par exemple, pour implémenter une stratégie d'exploration telle que la *Limited Discrepancy Search* (LDS) [Harvey 95b], il suffit de maintenir l'écart courant avec l'heuristique dans une variable entière backtrackable que l'on met à jour avec une contrainte réveillée sur l'événement associé aux points de choix FaCiLe fournit ainsi une fonction qui transforme un but standard en but LDS, ce dernier pouvant être combiné avec d'autres buts standards ou LDS.

La recherche pourrait également être rendue plus générique en modifiant le mécanisme de rétablissement de l'état du système (actuellement une pile) utilisé pour le backtrack, par exemple en copiant et en stockant l'état lors des points de choix comme dans [Choi 01a], et en veillant à partager les données pour limiter la consommation de mémoire.

5.6.3 Optimisation

FaCiLe fournit une fonction d'optimisation qui prend un but et un coût en arguments, et renvoie un nouveau but qui permet de chercher une solution optimale par *Branch & Bound* en posant dynamiquement une contrainte sur le coût à chaque fois qu'une solution est trouvée. Les buts d'optimisation peuvent être combinés pour écrire facilement des techniques de recherche sophistiquées. Le but de recherche peut également être transformé en but LDS avant d'être à nouveau transformé en but d'optimisation. En outre, l'utilisateur peut passer en argument à ces buts d'optimisation une fonction appelée à chaque solution trouvée et qui prend le coût de la solution comme argument, ce qui permet par exemple d'afficher ou stocker ce coût et la solution (à l'aide d'une clôture).

FaCiLe propose deux types d'optimisation sélectionnés grâce à un argument optionnel — nous supposerons sans perte de généralité qu'il s'agit d'une minimisation :

- **Restart** : Le premier point de choix de la recherche est mémorisé, et à chaque fois qu'une solution de meilleur coût est trouvée, on backtracke jusqu'à ce point de choix. Le coût est alors alors raffiné dynamiquement pour être strictement inférieur à celui de la dernière solution et le but de recherche est appelé de nouveau.
- **Continue** : Le premier point de choix de la recherche est également mémorisé, mais à chaque fois qu'une solution de meilleur coût est trouvée, on backtracke seulement jusqu'à qu'au dernier point de choix où le minimum du domaine du coût est stricte-

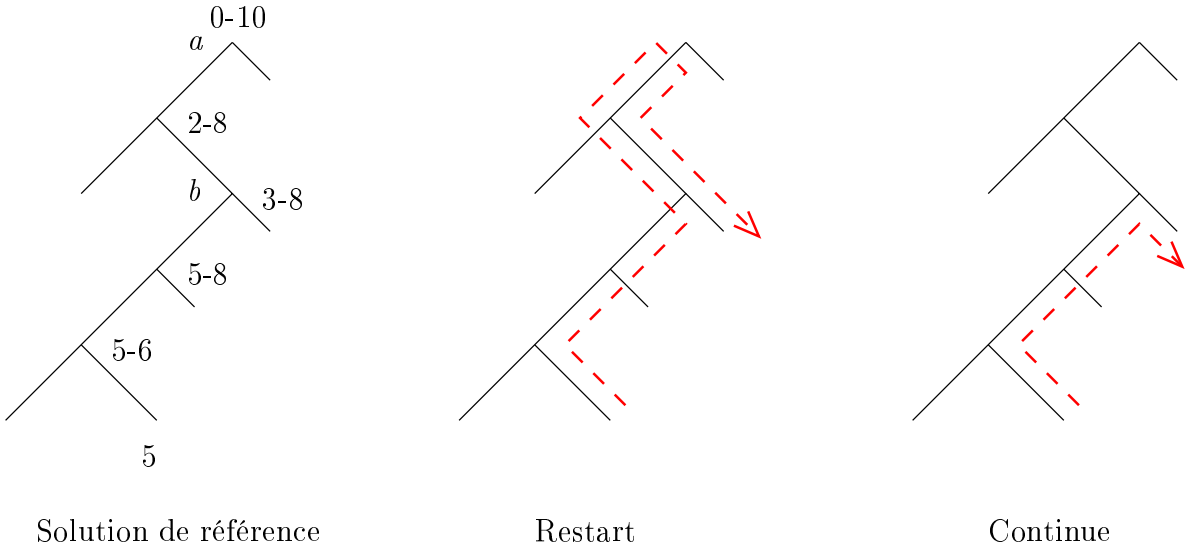


FIG. 5.10 – Minimisation : comparaison des modes *Restart* et *Continue*

ment inférieur à celui de la dernière solution, donc jusqu'à la première opportunité de faire diminuer le coût. Si on atteint le premier point de choix mémorisé avant, la recherche échoue (i.e. est terminée). Dans le cas contraire, pour maintenir le coût inférieur à celui de la dernière solution trouvée, une contrainte spéciale réveillée à chaque point de choix est posée au début de la recherche et sa borne supérieure est mise à jour à chaque fois qu'une solution est trouvée.

La figure 5.10 illustre les différences de parcours de l'arbre de recherche — c'est-à-dire les backtracks effectués et les nœuds visités — entre deux solutions successives par une minimisation en mode *Restart* d'une part et une minimisation en mode *Continue* d'autre part. Dans cet arbre de recherche, la borne inférieure du coût, dont le domaine est indiqué à chaque nœuds, est augmentée par trois fois le long du chemin qui mène à la solution de référence (la dernière trouvée). En mode *Restart*, la recherche backtracke jusqu'à la racine a de l'arbre et rejoint le nœud b après avoir revisité une partie du sous-arbre de gauche excepté les branches élaguées par la contrainte dynamique sur la borne supérieure du coût. Si la contrainte qui relie le coût aux variables de décisions du problème n'infère que peu ou pas de réductions de domaines — ce qui est en général le cas vers la racine — la recherche va devoir éventuellement revisiter une grande partie de l'arbre de recherche. En revanche, le mode *Continue* permet de ne backtracker que jusqu'au point b, c'est-à-dire la première opportunité pour suivre une branche où le coût peut être instancié à une valeur inférieure à la dernière trouvée.

Le mode *Continue* semble ainsi systématiquement plus efficace que le mode *Restart* car il visite un nombre de nœud inférieur, mais l'implémentation du premier nécessite une contrainte pour maintenir la borne supérieure du coût réveillée après chaque backtrack car elle peut être rétablie à une ancienne valeur (inférieure). Cette contrainte peut alors

elle-même susciter le réveil des autres contraintes, ce qui est parfois plus coûteux en temps de calcul que la stratégie du mode *Continue* qui raffine le coût au début de la recherche.

5.7 Invariants

Les *invariants* sont des équations fonctionnelles entre des variables *instanciées* qui doivent être maintenues quand ces variables changent de valeur. Ces équations, contrairement à celles qui définissent des contraintes, sont des dépendances *directionnelles* entre les variables : dans l'invariant $z = f(x, y)$, la valeur de z peut être calculée si x et/ou y ont été modifiés, mais pas l'inverse. Un ensemble de variables et d'invariants peut être représenté comme un graphe de contraintes (les nœuds sont les variables et les arêtes les invariants) mais orienté et sans cycle. Seules les feuilles de cet arbre, c'est-à-dire les variables qui ne sont pas le résultat d'un invariant, peuvent être modifiées par l'utilisateur, et les autres variables doivent ensuite être mises-à-jour. Comme le graphe des invariants ne contient pas de cycle, les variables peuvent être mises-à-jour en une seule passe pour chaque modification.

Dans le domaine de la bureautique, la plupart des tableurs numériques fonctionnent sur ce principe. Dans celui de la programmation par contrainte, certains systèmes graphiques précurseurs tels que Sketchpad [Sutherland 63] et ThingLab [Borning 81] utilisaient des *one-way constraints*, qui peuvent être vues comme des invariants. Plus récemment, des systèmes tels que DeltaBlue [Sannella 93] ont été conçus pour manipuler des *multi-way constraints* dont la sémantique est plus proche des contraintes classiques, mais toujours sur des variables instanciées. Mais l'inspiration des invariants de FaCiLe vient plutôt de la recherche locale et du langage de modélisation Localizer [Michel 97] où ils sont utilisés pour automatiser la mise-à-jour incrémentale des structures de données et des critères qui définissent un voisinage et une fonction de coût. Mais les invariants de Localizer sont plus généraux que ceux de FaCiLe car ils peuvent être définis récursivement — ce qui engendre des cycles dans le graphe associé — et leur mécanisme de mise-à-jour est complexe pour en garantir la complexité temporelle optimale lorsque plusieurs variables ont été modifiées (ce qui correspond à une transition dans un algorithme de recherche locale).

FaCiLe fournit des *références invariantes backtrackables* polymorphes qui permettent d'utiliser des invariants primitifs ou de spécifier la fonction qui calcule l'invariant. Les variables ainsi créées sont mises-à-jour automatiquement et ces modifications sont défaites lors des backtracks.

5.7.1 Maintenance incrémentale

Les primitives de FaCiLe calculent incrémentalement des invariants sur des tableaux de variables tels que somme et produit généralisés en diminuant la complexité temporelle

TAB. 5.2 – Les invariants primitifs de FaCiLe et leur complexité temporelle et spatiale

Invariant	Temps	Mémoire
$s = \sum_{i=1}^{n} x_i$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$p = \prod_{i=1}^{n} x_i$	$\mathcal{O}(1)$ ou $\mathcal{O}(n)$	$\mathcal{O}(1)$
$m = \min_{i\in[1,n]} x_i$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
$i = \text{argmin}_{i\in[1,n]} x_i$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

(dans ces cas linéaire) de l'opération standard :

$$s = \sum_{i=1}^{n} x_i \qquad x_j \to x'_j \qquad s' \to s - x_j + x'_j$$

$$p = \prod_{i=1}^{n} x_i \qquad x_j \to x'_j \qquad \begin{cases} \text{si } x_j \neq 0 & p' \to \frac{p x'_j}{x_j} \\ \text{si } x_j = 0 & p' \to x'_j \prod_{i\in[1,n]\setminus\{j\}} x_i \end{cases}$$

On peut donc mettre à jour ces invariants en un temps constant (linéaire à l'initialisation) quand une des variables change ($x_j \to x'_j$) sauf dans le cas où x_j était nulle (et donc le produit également). La table 5.2 donne les complexités temporelles et spatiales d'une mise-à-jour des invariants implémentés dans FaCiLe. Le minimum d'un tableau (index ou valeur) est implémenté en utilisant la librairie standard d'ensembles d'OCaml (qui se sert d'arbres binaires équilibrés).

Si l'utilisateur spécifie sa propre fonction pour obtenir un invariant, celle-ci est appelée lors de chaque mise-à-jour et donc l'invariant est entièrement recalculé.

5.7.2 Contrôle des mises-à-jour

Dans Localizer, une phase de planning précède la mise-à-jour des invariants pour garantir que chaque invariant ne sera considéré qu'une seule fois pour chaque variable apparaissant dans l'expression qui le calcule, en prenant en compte un nombre quelconque de modifications (transition entre deux solutions de la recherche locale). Ce planning détermine l'ordre dans lequel seront propagés les invariants en analysant leur dépendance. D'après [Michel 97], cette phase est statique (effectuée à l'initialisation) si les éléments des tableaux ne dépendent pas les uns des autres dans les invariants d'indexation, et doit être dynamique (recalculée après chaque transition) dans le cas contraire. L'ambition de cette technique est d'obtenir une complexité proportionnelle au nombre de modifications subies par les variables.

Les invariants de FaCiLe sont plus restrictifs (pas de définition récursive ni de cycle, mais les variables peuvent dépendre les une des autres dans les tableaux) et doivent « vivre » avec les backtracks pour pouvoir les utiliser comme critères au sein des buts. Comme FaCiLe possède des mécanismes d'événements génériques et des références backtrackables polymorphes, la solution la plus simple est de contrôler la mise-à-jour des invariants comme

des contraintes. De plus, la possibilité de passer en paramètre l'indice de la variable qui a été modifiée à la fonction de propagation d'une contrainte permet d'implémenter les opérations sur les tableaux avec les mêmes complexités que celles de Localizer.

Lors de la création d'une référence invariante, une référence backtrackable et un nouvel événement lui sont associés. Les « contraintes » (objets réactifs) qui calculent des invariants faisant intervenir cette variable s'enregistrent alors sur cet événement déclenché lors de la modification de la variable. FaCiLe fournit également des primitives liées aux caractéristiques des variables logiques et réveillées sur leurs événements standards. Pour ces dernières, un foncteur est utilisé de la même manière que dans la section 5.1.2 pour obtenir des invariants sur les différents types de variables logiques. Cette implémentation moins efficace que celle Localizer est cependant d'une très grande concision et simplicité.

5.8 Perspectives

FaCiLe est une librairie destinée essentiellement à la recherche et l'enseignement, mais avec des performances souvent meilleures que les systèmes Prolog et une robustesse comparable à celle d'ILOG Solver, le système commercial leader du marché. L'une de ses caractéristiques importantes est sa grande concision et la possibilité de l'étendre facilement à l'aide de « briques » de base et grâce à la puissance d'expression de son langage hôte.

Néanmoins, des évolutions de la librairie sont envisagées pour permettre de l'utiliser dans des domaines variés tels que le *scheduling*, le *planning* et les problèmes d'optimisation continus et mixtes. Des améliorations de plus bas niveau peuvent également être apportées pour améliorer les performances en temps et en mémoire de FaCiLe, sans toutefois sacrifier sa simplicité.

5.8.1 Domaines de variables, de contraintes et coopération de solveurs

Certains domaines typiques de la programmation par contraintes nécessitent des contraintes globales dédiées, souvent issues d'algorithmes de Recherche Opérationnelle qui ont été adaptés pour pouvoir être maintenus incrémentalement. Par exemple, pour pouvoir résoudre efficacement des problèmes de *scheduling*, qui sont l'une des applications industrielles les plus répandues de la programmation par contraintes, il faudrait ajouter à FaCiLe des techniques telles que le *edge-finding* [Carlier 89] ou les *intervalles de tâches* [Caseau 94].

D'autre part, un système de programmation par contraintes constituent un environnement adapté pour contrôler des solveurs dédiés, comme les solveurs d'équations linéaires utilisant un Simplex qui permettent, en relaxant la contrainte d'intégrité, d'obtenir de meilleures bornes sur le coût [Focacci 99]. Ce type d'approche s'intègre élégamment par l'intermédiaire de contraintes globales [Milano 00]. De même, pour résoudre des problèmes industriels mixtes continus et entiers, des variables logiques sur des domaines continus doivent être ajouter à FaCiLe (cf. section 5.1.2).

5.8.2 Optimisation de bas niveau

À plus bas niveau, des optimisations peuvent être expérimentées dans le code de FaCiLe sans avoir à bouleverser l'existant. L'architecture de la librairie a été conçue pour encapsuler les concepts essentiels et l'abstraction des structures de données permet de les modifier facilement sans avoir à réécrire les modules qui en dépendent.

La consommation de mémoire et sa récupération automatique est un des facteurs qui limite l'efficacité de FaCiLe. Le mécanisme très générique de traînée (*trailing*) de clôtures utilisée dans FaCiLe (cf. section 51) est en partie responsable de cette consommation, car le récupérateur de mémoire ne coopère pas de façon informée avec la pile (récupération des domaines entre niveaux coupés par exemple). [Choi 01a] propose d'abstraire le mécanisme de rétablissement d'état pour pouvoir expérimenter différentes stratégies (recalcule, copie paresseuse etc.). Certaines de ces techniques pourraient être plus efficaces que le trailing, notamment en veillant au partage des structures de donnée non-mutables.

La représentation des domaines par des listes d'intervalles peut également handicaper les performances lors de certaines opérations qui sont de complexité linéaire avec cette structure (e.g. appartenance). Une représentation avec des vecteurs de bits quand le domaine de la variable est suffisamment petit pourrait améliorer l'efficacité d'une partie de ces manipulations, d'autant plus que certaines opérations se réduisent à une instruction machine (intersection, union).

Troisième partie

Applications à la gestion du trafic aérien

Chapitre 6

Allocation de créneaux

L'organisme européen de régulation du trafic (CFMU) qui est chargé d'allouer les créneaux de départs des vols afin de respecter les charges admissibles des centres de contrôles en route (ATCC) utilise un algorithme glouton dotés de mauvaises propriétés concernant sa correction, l'interprétation des contraintes de capacités et l'optimisation. Un nouveau modèle en Programmation Par Contraintes a été proposé au sein de la plate-forme SHAMAN (du CENA) mais il engendre un profil de charge très irrégulier et viole les contraintes de capacités. Cette étude présente deux modèles originaux en PPC du problème d'allocation de créneaux qui accorde une place prépondérante à la charge de travail des contrôleurs : un raffinement du modèle SHAMAN et une nouvelle approche fondée sur des contraintes de *tri* maintiennent « continûment » le débit d'avions sous la capacité spécifiée. Nous établissons des équivalences entre ces modèles tout en montrant la supériorité de la solution *continue* avec tri pour les preuves d'échec et/ou d'optimalité. Les différents comportements des modèles sont mis en évidence sur des données partielles et complètes du trafic aérien français et les améliorations que pourraient apporter ces modèles au système actuel sont évoquées.

6.1 Introduction

L'encombrement de l'espace aérien est un des problèmes majeurs que les organismes de régulation du trafic aérien (ATM, *i.e.* Air Traffic Management) européen doivent résoudre. Les capacités des centres français de contrôle du trafic aérien (ATCC, *i.e.* Air Traffic Control Center) sont dépassées par la croissance constante du nombre de vols, entraînant des retards toujours plus importants. Les coûts engendrés par ces retards pénalisent aussi bien les compagnies que les passagers, à tel point que la Commission Européenne a déclaré très récemment (1erdécembre 1999) que les systèmes actuels de régulation du trafic aérien (ATFM, *i.e.* Air Traffic Flow Management) sont incapables de supporter les pointes de trafic ni de s'adapter à la croissance prévue. Les pertes financières dues aux retards sont évaluées par la Commission à plus de 5 milliards d'euros pour l'année 1999 et des améliorations drastiques doivent être entreprises pour surmonter le problème [IP 99].

L'organisme central de régulation des flux de trafic (CFMU, *i.e.* Central Flow Management Unit, situé à Bruxelles) est chargé, entre autres mesures stratégiques ou tactiques, de retarder les créneaux de décollages[1] des vols impliqués dans les secteurs surchargés. L'objectif de ces affectations de retards est de respecter les contraintes de capacité *en route*[2] fournies par chaque centre de contrôle suivant leur *schéma d'ouverture* quotidien.

Actuellement, la CFMU résout ce problème en deux étapes [CFMU 00] :

1. un outil « pré-tactique » (PREDICT) détecte les secteurs surchargés et permet à un expert de choisir des régulations en simulant leur impact ;
2. un système « tactique » (TACT) auquel est intégré le module CASA (*Computer Assisted Slot Allocation*) utilise ensuite un algorithme glouton pour allouer dynamiquement les créneaux de décollage des vols régulés selon le principe « premier arrivé, premier servi » au fur et à mesure que les plans de vol parviennent à la CFMU.

CASA calcule une liste de créneaux en divisant la durée de la période à réguler par sa capacité et tente d'attribuer chaque créneau selon l'heure d'arrivée des vols dans le secteur concerné. Quand de nouveaux plans de vol arrivent, la solution est reconsidérée et tous les créneaux déjà alloués peuvent être décalés pour respecter le principe « premier arrivé, premier servi ».

La solution obtenue par ce processus dépend de l'ordre de résolution des contraintes et aucune optimisation globale n'y est possible — excepté lors de la phase finale où les créneaux alloués à des vols annulés tardivement peuvent être réattribués. De plus, dans le cas où un vol subit des régulations multiples, le créneau est calculé uniquement en fonction du secteur qui « induit le plus grand retard », ce qui peut violer les contraintes de capacité des autres secteurs traversés par ce vol. Une étude précédente [Gotteland 00] estime que les régulations locales de la CFMU ont un impact très faible sur la surcharge globale des centres de contrôle. Mais CASA est un outil opérationnel qui doit donc se plier à des contraintes supplémentaires (dont on ne tiendra pas compte dans les modèles présentés) et qui est bien intégré dans la chaîne complexe du processus d'ATM.

Pour surmonter les inconvénients et le manque d'efficacité de CASA, le CENA a développé un module d'allocation de créneaux en Programmation Par Contraintes au sein de la plate-forme SHAMAN[3] [Plusquellec 98]. Cet outil pose des contraintes de capacité sur tous les secteurs ouverts en divisant leur durée totale en tranches contiguës de 30 min et en restreignant le nombre de vols qui entrent dans les secteurs durant chacune de ces tranches : en supposant que le secteur AIX est ouvert de 12h00 à 13h30 avec une capacité de 32 vols par heure, trois contraintes seront posées pour assurer qu'il n'entrera pas plus de 16 avions dans le secteur entre 12h00 et 12h30, puis entre 12h30 et 13h00 et enfin entre 13h00 et 13h30. Avec cette approche, la PPC apporte une vue globale des contraintes de capacité, si bien que les solutions (s'il en existe) sont consistantes (i.e. toutes les contraintes sont satisfaites).

[1] Un créneau est un intervalle de faible durée pendant lequel le vol est autorisé à décoller.

[2] Le contrôle aérien est divisé en deux grandes catégories : le contrôle *en approche* pour la gestion des décollage et des atterrissages et le contrôle *en route* pour la gestion des vols en altitude.

[3] System to Help Analysis and Monitoring of Acc resources and Air route Network

Cependant, le modèle de SHAMAN n'empêche pas des pics de trafic pouvant excéder la capacité des secteurs sur des périodes dont le début n'est pas un multiple de 30 min à partir du début de leur contrainte, et les contrôleurs auraient alors à gérer des charges de travail trop importantes. En outre, même si SHAMAN ne tente pas d'optimiser la solution obtenue, la stratégie de recherche utilisée est guidée par la minimisation de la somme totale des délais. En conséquence, les vols retardés ont une tendance chronique à se concentrer au début de chaque tranche de 30 min, ce qui génère également des pics de trafic dépassant les capacités (si on les calcule sur des périodes inférieures à 30 min).

Les contraintes de capacité ne sont donc pas très clairement définies et sont manifestement mal interprétées d'un point de vue opérationnel par les systèmes actuels d'ATFM. Il serait plus réaliste de maintenir la charge de travail des contrôleurs continûment en dessous de la capacité spécifiée. Nous proposons dans ce but plusieurs nouveaux modèles qui apportent une interprétation plus adéquates des contraintes de capacités, et produisent par conséquent des problèmes plus difficiles à satisfaire :

- Un modèle à fenêtres glissantes qui permet de lisser le profil de la charge de travail des contrôleurs. Un paramètre fait varier la « dureté » du modèle du moins contraint (SHAMAN) au plus contraint (les contraintes de capacités sont satisfaites sur n'importe quelle période d'une durée donnée).
- Un modèle fondé sur une contrainte de tri où sont déclarées de manière « continue » les contraintes de capacité sur les rangs des dates d'entrée des avions dans un secteur.

La souplesse et l'expressivité de la PPC avec FaCiLe nous ont permis de formuler et d'implémenter simplement ces nouveaux modèles, ainsi que de reproduire celui de SHAMAN, présenté à des fins de comparaison. Les expérimentations menées sur ces modèles originaux concrétisent leurs intentions : le profil du débit d'avions est beaucoup plus lisse et sans pic de surcharge. En outre, le modèle continu nous permet d'obtenir des solutions optimales ou des preuves d'échec.

Nous donnons d'abord une description précise du problème d'allocation de créneaux ainsi que des indications sur la taille et la complexité des instances auxquelles nous nous sommes intéressés. Nous présentons ensuite les différents modèles utilisés pour la résolution de ce problème, en commençant par la formulation standard pour aboutir aux modèles « continus » ; puis nous exposons les résultats obtenus avec des instances simplifiées et complètes en comparant les comportements de ces différents modèles. Nous concluons en résumant les enjeux et contributions de cette étude et donnons enfin quelques pistes à exploiter pour des travaux ultérieurs.

6.2 Description du problème d'allocation de créneaux

La régulation des flux de trafic est un filtre tactique destiné à homogénéiser les vols qui doivent traverser l'espace aérien contrôlé : il s'agit de limiter le nombre d'appareils qui pénètrent dans un secteur donné pendant un intervalle de temps donné. Ce planning est raffiné en temps réel par les contrôleurs.

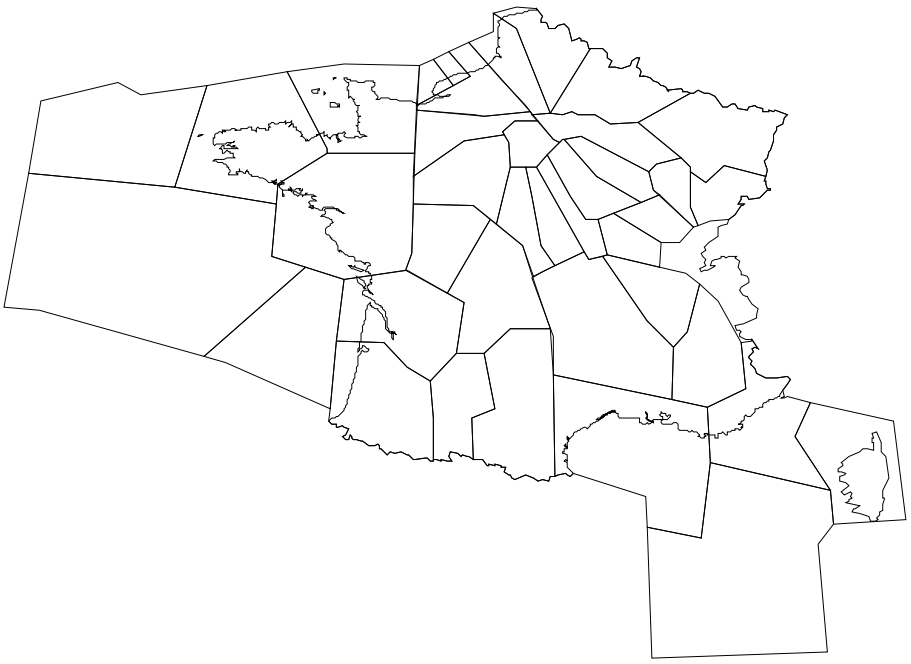


FIG. 6.1 – Sectorisation de l'espace aérien français au niveau de vol 240 (7200 m)

Les données du problème d'ATFM sont :

- Les *plan de vols* : un vol décolle d'un aéroport français donné ou entre dans l'espace aérien français à une date donnée, suit une route prédéfinie à une vitesse fixée et atterrit sur un aéroport ou sort de l'espace aérien national.
- Les *secteurs* : l'espace aérien est divisé en secteurs de contrôle traversés par les routes suivies par les aéronefs. Un secteur est un polyèdre, en général un cylindre vertical, doté d'une *capacité* exprimée par un nombre maximum de vols entrant dans le secteur pendant un intervalle de temps donné (une heure). Le découpage de l'espace aérien, c'est-à-dire le nombre et la forme des secteurs, change au cours de la journée suivant un *schéma d'ouverture* quotidien. La capacité d'un même secteur peut elle aussi changer à des heures données. Nous appellerons *secteur-période* un secteur de contrôle pendant un intervalle de temps particulier et durant laquelle la capacité est constante.

La figure 6.1 donne une idée du découpage de l'espace aérien français à 24000 ft.

Les contraintes du problème sont les diverses capacités des secteurs de contrôle. Plusieurs mesures peuvent être prises par la CFMU pour respecter ces contraintes : changer la route aérienne empruntée, retarder le départ, modifier la vitesse de l'appareil pendant le vol... Nous ne nous intéressons ici qu'aux mesures d'attente au sol : chaque vol peut-être retardé au décollage. La CFMU est chargée de résoudre ce problème quotidiennement.

La difficulté du problème réside moins dans la complexité des contraintes que dans la taille des données. Nous avons utilisé des données réelles archivées par le logiciel COURAGE utilisé par l'Aviation Civile française et résolu le problème d'ATFM pour la journée du 20 mai 1999 :

- L'espace aérien français est traversé par 7375 vols entre 0h00 et 23h59.
- Quelque 140 secteurs sont activés pendant cette journée ; les contraintes de capacités changent jusqu'à six fois pour un même secteur.
- Plus de 700 vols traversent le secteur le plus fréquenté.
- Les capacités varient de 19 à 52 vols par heure suivant le secteur-période.

Le but de l'allocation de créneaux est de réduire les retards attribués aux vols tout en respectant les contraintes de capacité des secteurs. Plusieurs critères de minimisation sont possibles : la somme totale des retards, le retard maximal, le retard moyen, etc. [Maugis 96] présente une étude complète sur ce que pourrait et devrait être la fonction de coût du problème d'allocation de créneaux et conclut en choisissant la plus simple : la somme totale des retards. C'est également le choix de [Bertsimas 95] qui utilise un modèle légèrement différent. Dans cette étude, nous avons choisi de minimiser le retard maximal mais nous nous intéressons plus aux propriétés qualitatives des solutions produites qu'à leur coût en terme de retards induits.

6.3 Quatre modèles

Nous décrivons dans cette section quatre modèles pour le problème d'affectation de créneaux. Ces modèles ne sont pas équivalents et diffèrent par l'interprétation des contraintes de capacité des secteurs de contrôle.

La description des modèles utilise les données suivantes :

- $\mathcal{S}$: l'ensemble des secteurs-périodes, chaque secteur-période s étant associé à un début $start(s)$ et une fin $end(s)$;
- $\mathcal{F}$: l'ensemble des vols ;
- t_i^s : l'heure à laquelle le vol i entre dans le secteur s si ce vol n'est pas retardé ;
- $capa^s$: la *capacité* du secteur-période s (nombre de vols entrant par heure) ;
- δ : la durée de la période unitaire pour la contrainte de capacité (en minutes).

Tous les modèles travaillent sur les variables de décision suivantes :

- D_i : retard au décollage du vol i.

Nous présentons d'abord le modèle implémenté dans SHAMAN et décrit par [Maugis 96] et [Plusquellec 98], ainsi qu'un autre modèle équivalent mais plus efficace. Plusieurs « effets » de bord surviennent avec ce modèle, ce qui conduit à des profils irréguliers et des surcharges de trafic. Deux nouveaux modèles sont ensuite introduits pour remédier aux problèmes générés par les modèles précédents.

6.3.1 Fenêtres juxtaposées

Le modèle utilisé dans SHAMAN (que nous appellerons dans la suite modèle `Standard`) discrétise les durées d'ouverture des secteurs par tranches de $\delta = 30\,min$ pour poser les contraintes de capacité et réduire la somme des délais. Ces périodes sont successives et contiguës :

– $\mathcal{P}^s = \{p_0^s, p_1^s, \ldots\}$: périodes successives de longueur δ, chacune dotée d'un début ($start$) et d'une fin (end). La première période commence avec le début du secteur-période : $start(p_0^s) = start(s)$.

On peut encore noter ces tranches comme des multiples de δ :

$$\mathcal{P}^s \quad = \quad \{[start(s), start(s)+\delta[, [start(s)+\delta, start(s)+2\delta[, \ldots\}$$

Le choix de non recouvrement des contraintes de capacité conduit à plusieurs formulations possibles ; nous en présentons deux dans les sections suivantes, l'une utilisant des variables booléenne et des contraintes locales, l'autre utilisant une contrainte globale.

Variables booléennes

Le modèle `Standard` utilise des variables auxiliaires booléennes correspondant à la présence des vols dans les périodes :

– B_{i,p_j^s} : le vol i entre dans le secteur s durant la période p_j^s.

Les contraintes correspondantes sont les suivantes : la première relie les variables auxiliaires aux variables de décision, la seconde exprime la capacité du secteur.

$$\forall s \in \mathcal{S} \;\; \forall p_j^s \in \mathcal{P}^s \qquad \begin{cases} \forall i \in \mathcal{F} \quad B_{i,p_j^s} \text{ ssi } start(p_j^s) \leq t_i^s + D_i < start(p_j^s) + \delta \\ \\ \displaystyle\sum_{i \in \mathcal{F}} B_{i,p_j^s} \leq capa^s \end{cases}$$

L'inconvénient majeur de cette formulations est le nombre important de variables auxiliaires, proportionnel au nombre de secteurs et au nombre de périodes.

$$|\{B_{i,p_j^s}\}| \quad = \quad |\mathcal{F}| \sum_{s \in \mathcal{S}} \frac{end(s) - start(s)}{\delta}$$

Pour restreindre l'usage de variables auxiliaires, la formulation suivante utilise une contrainte globale.

Contrainte globale de cardinalité

Une contrainte globale de cardinalité (désignée par `Gcc` dans la suite) est spécifiée avec les composants suivants :

– un ensemble de variables $X = \{X_1, \ldots, X_n\}$;
– un ensemble de valeurs $V = \{V_1, \ldots, V_d\}$;
– un ensemble de cardinaux $C = \{C_1, \ldots, C_d\}$.

Elle contraint le nombre d'occurrences de valeurs de V parmi X. Formellement, la contrainte $gcc(C, V, X)$ exprime :

$$\forall i \in [1, d] \quad |\{x \in X | x = V_i\}| = C_i$$

[Régin 96] a proposé un algorithme de propagation efficace et complet pour cette contrainte (implémentée dans FaCiLe).

La contrainte de cardinalité permet de reformuler aisément le modèle `Standard` pour l'allocation de créneaux. L'idée de base est de *calculer* la période pendant laquelle un vol entre dans un secteur en fonction de son retard. Or dans le modèle standard les périodes constituent une partition régulière du temps, ce qui permet de réduire ce calcul à une simple opération arithmétique : si les périodes successives de longueur δ sont numérotées $0, 1, \ldots, j, \ldots$ la période correspondant à un temps t vaut $\lfloor t/\delta \rfloor$. Une contrainte de capacité s'exprime donc en prenant $V_j = j$ et en imposant que C_j soit inférieur à la capacité de la période j.

Le modèle s'exprime en utilisant les variables et valeurs auxiliaires suivantes :
- X_i^s : index de la période de l'heure d'entrée du vol i dans le secteur-période s ;
- $V_j^s = j$: index de la $j^{\text{ème}}$période dans le secteur-période s ;
- $C_j^s \in [0, capa^s]$: nombre de vols entrant dans le secteur-période s durant la période j.

Pour chaque secteur-période, il suffit alors de lier les variables auxiliaires aux variables de décision et de contraindre les variables auxiliaires avec une contrainte de cardinalité globale :

$$\forall s \in \mathcal{S} \quad \begin{cases} \forall i \in \mathcal{F} \quad X_i^s = \dfrac{t_i^s + D_i}{\delta} \\ \\ gcc(C^s, V^s, X^s) \end{cases}$$

Les arguments pour préférer l'usage d'une contrainte globale à un ensemble de contraintes élémentaires sont nombreux :
- le modèle est plus simple à écrire et plus facile à appréhender ;
- le nombre de contraintes et de variables est réduit ;
- une propagation plus efficace peut être réalisée.

Cependant, une contrainte globale est parfois trop spécifique pour être utilisée au sein d'un problème « non pur » et nous montrerons dans le cas présent la trop grande rigidité de cette solution qui interdit de la raffiner.

Inconvénients

Les deux modèles précédemment décrits ont l'inconvénient d'être fortement discontinus. Ils contraignent seulement la charge du secteur au début de chaque période, c'est-à-dire uniquement 48 fois par jour (pour des périodes de 30 min). Ainsi, la charge peut grimper bien au-delà de la capacité spécifiée pour des fenêtres de temps qui ne commencent pas à une date multiple de δ (à partir du début de la contrainte). La figure 6.2 illustre ce phénomène : pour un nombre total de vols qui n'excède pas la capacité ni dans la première période, ni dans la seconde, une tranche de 30 min chevauchant les deux autres peut compter jusqu'à deux fois trop de vols.

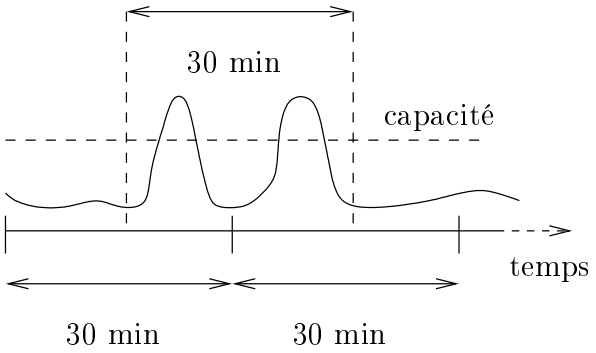


FIG. 6.2 – Surcharge entre deux périodes avec le modèle `Standard`

De plus, les contraintes de ce modèle n'empêche pas les pics de trafic pour des fenêtres de temps inférieures à δ, comme l'illustre la figure 6.3 : les deux profils correspondent (approximativement) à la même quantité de vols (supposée inférieure à la capacité sur la durée totale de la période), mais celui de gauche, autorisé par le modèle `Standard`, répartit moins bien la charge de travail pour un contrôleur que le profil de droite, beaucoup plus régulier.

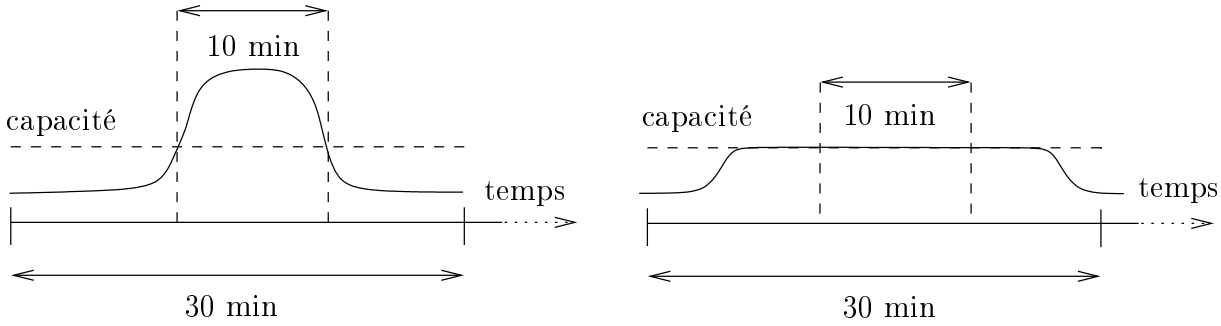


FIG. 6.3 – Le modèle `Standard` autorise des surcharges sur des fenêtres de temps courtes (inférieures à δ)

Comme l'objectif de la fonction de coût est de minimiser la somme des retards, l'effet de bord attendu de cette modélisation est la concentration de vols en début de période : les vols retardés sont programmés le plus tôt possible dans la prochaine fenêtre de temps. La figure 6.4 illustre ce comportement : la courbe en pointillés correspond à la charge d'un secteur avant régulation, et le modèle `Standard` tend à décaler la surcharge de la première période au début de la seconde. Cet effet de bord chronique confirmé par les expérimentations (voir section 6.4).

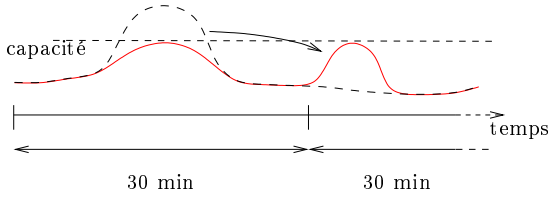


FIG. 6.4 – Pics de trafic chroniques dus à l'optimisation du modèle `Standard`

6.3.2 Modèles continus

Nous proposons deux nouvelles formulations pour tenter de lisser la charge de trafic avec une modélisation plus contraignante des contraintes de capacité. Le premier modèle est un simple raffinement du modèle `Standard`. La seconde utilise une contrainte globale de tri sur un problème dual.

Fenêtres glissantes

En conservant l'idée du modèle `Standard`, il est possible de définir un modèle « plus continu » en considérant des périodes qui se chevauchent, comme le montre la figure 6.5.

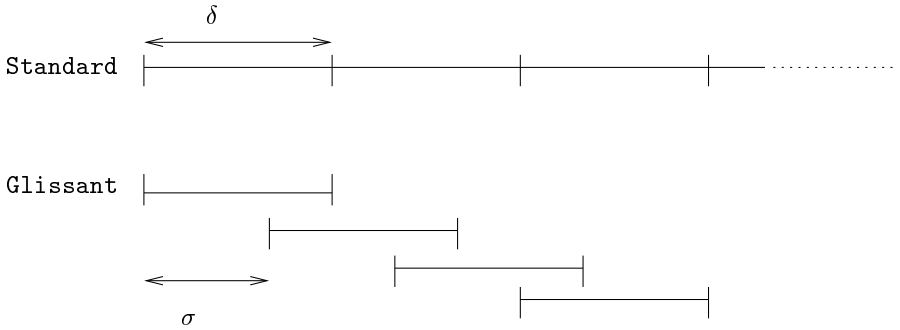


FIG. 6.5 – Modèle `Standard` et modèle `Glissant`

Nous introduisons pour cela un paramètre supplémentaire σ correspondant à l'intervalle de temps entre 2 périodes successives :

$$\mathcal{P}^s \quad = \quad \{[start(s), start(s) + \delta[, [start(s) + \sigma, start(s) + \sigma + \delta[, \ldots\}$$

Ce modèle subsume évidemment le modèle `Standard` et lui est équivalent pour $\sigma = \delta$. Il peut être mis en œuvre avec les mêmes variables booléennes auxiliaires. Malheureusement, l'astuce permettant d'utiliser la contrainte globale de cardinalité ne s'applique plus dans le cas général d'un σ quelconque : un vol est concerné par plusieurs périodes simultanément.

On s'attend à ce que le profil de la charge soit de plus en plus lisse quand σ diminue et donc que davantage de fenêtres de temps sont prises en compte.

Modèle par tri

La contrainte de tri Soit D un ensemble totalement ordonné. La contrainte de tri est la relation associée à la fonction standard de tri. Alors qu'une fonction de tri prend en argument une séquence de n éléments de D et retourne une autre séquence contenant les mêmes éléments ordonnés, la contrainte « lie » deux séquences de variables à domaine fini inclus dans D.

Formellement, la contrainte de tri sur les variables intervalle $X_1, \ldots, X_n, Y_1, \ldots, Y_n$ dans un ensemble totalement ordonné $(D, \preceq)$ exprime

$$(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \in sort$$

où

$$sort = \left\{ \begin{array}{l} (x_1, \ldots, x_n, y_1, \ldots, y_n) \in D^{2n} \text{ tel que} \\ (y_1, \ldots, y_n) \text{ est une permutation de } (x_1, \ldots, x_n) \text{ et } \forall i \leq j,\ y_i \preceq y_j \end{array} \right\}$$

Par exemple, soient :

$$X = \{[0, 13]; [6, 10]; [10, 11]; [4, 16]; [4, 6]\}$$

$$Y = \{[1, 3]; [5, 10]; [6, 9]; [11, 17]; [10, 15]\}$$

deux séquences de variables. La pose de la contrainte $sort(X, Y)$ conduit aux raffinements suivant sur les bornes des intervalles :

$$X = \{[1, 3]; [6, 9]; 11; [11, 15]; [5, 6]\}$$

$$Y = \{[1, 3]; [5, 6]; [6, 9]; 11; [11, 15]\}$$

[Bleuzen-Guernalec 97] propose un algorithme efficace de filtrage pour cette contrainte. Il est remarquable que la complexité de cet algorithme complet de consistance (i.e. la propagation effectuées sur les bornes des intervalles est maximale) ait une complexité optimale en $\mathcal{O}(n \log n)$. L'algorithme est décrit en six étapes impliquant des structures de données relativement complexes (arbre binaire balancé, ...) et des tris standards. Le raffinement n'est fait que sur les bornes des domaines des variables.

Un modèle dual Pour ce problème, nous utilisons une contrainte de tri par secteur-période. Cette contrainte est posée sur des variables auxiliaires correspondant aux heures d'entrée dans le secteur et à ces mêmes heures ordonnées :

- T_i^s : heure d'entrée du vol i dans le secteur-période s ;
- S_j^s : heure d'entrée du $j^{\text{ème}}$vol entrant dans le secteur-période s.

La figure 6.6 montre un tableau de 4 vols et le domaine de leur date d'entrée respective (l'unité de temps est 5 min), puis le tableau des variables triées (après propagation) relié au tableau d'origine par les permutations possibles.

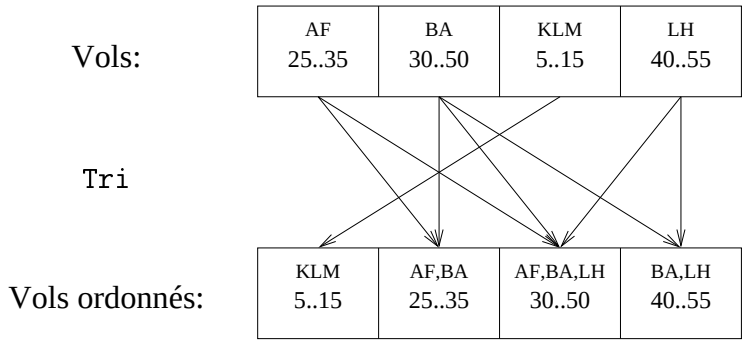


FIG. 6.6 – Permutations entre un tableau de vols et ces mêmes vols *triés*

Les contraintes de tri lient les variables de décision avec les variables auxiliaires et les contraintes de capacité sont imposées sur ces dernières avec un modèle « dual » du modèle `Standard`. Les vols qui entrent dans le secteur ne sont plus « comptés » comme précédemment, mais seuls les *rangs* des dates d'entrée sont considérés : si les rangs de deux vols sont trop proches, on contraint leurs dates à être suffisamment séparées dans le temps. Ainsi, deux vols de dates d'entrée S_i et S_j dans la séquence ordonnée tels que $i + capa \leq j$ doivent être distants d'au moins δ minutes (c.f. figure 6.7) :

$$\forall s \in \mathcal{S} \quad \begin{cases} \forall i \in \mathcal{F} \quad T_i^s = t_i^s + D_i \\ sort(T^s, S^s) \\ \forall i \in [1, |\mathcal{F}| - capa^s] \quad S_i^s + \delta \leq S_{i+capa^s}^s \end{cases}$$

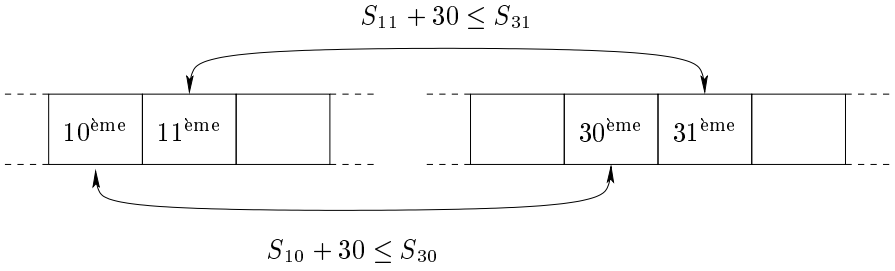


FIG. 6.7 – Contrainte de capacité appliquée aux vols ordonnés ($\delta = 30$, $capa = 20$)

La contrainte de capacité n'est pas tout à fait correcte car elle est posée même pour des paires de vols dont l'un entre avant le début ou après la fin du secteur-période. Elle doit donc être relaxée dans ce cas ; cela peut être réalisé simplement à l'aide de variables

booléennes :

$$\forall s \in \mathcal{S} \quad \begin{cases} \forall i \in \mathcal{F} \quad B_i^s \text{ ssi } start(s) \leq S_i^s < end(s) \\ \\ \forall i \in [1, |\mathcal{F}| - capa^s] \quad B_i^s \wedge B_{i+capa^s}^s \Longrightarrow S_i^s + \delta \leq S_{i+capa^s}^s \end{cases}$$

La variable booléenne B_i^s est la réification de la contrainte exprimant que le secteur-période s est concerné par le vol i. Si pour une paire de vols, un des deux vols est en dehors de la période, la contrainte de capacité n'est pas posée.

L'utilisation d'une contrainte de tri a déjà été proposée par [Zhou 96] pour résoudre le problème de *job-shop*. La contrainte d'utilisation unique d'une machine (de capacité unitaire) a un instant donné est alors facilement exprimée en contraignant la $j^{\text{ème}}$tâche à finir avant le début de la $(j+1)^{\text{ème}}$. Notre formulation diffère donc de celle-ci par l'expression supplémentaire d'une capacité à travers les contraintes de précédence.

Ce modèle par `Tri` assure que les contraintes de capacité sont respectées « continûment » sur chaque secteur-période. Le modèle `Glissant` doit lui être équivalent lorsque le paramètre σ est réduit à l'unité de temps, c'est-à-dire que toutes les fenêtres de temps de durée δ sont contraintes.

6.4 Expérimentations

Nous présentons dans cette partie les résultats obtenus pour les différents modèles appliqués à des données réelles (journée du 20 mai 1999) et implémentés avec FaCiLe.

La recherche utilise la stratégie implantée dans SHAMAN pour les modèles `Standard` et `Gcc` : les secteurs-périodes *a priori* les plus surchargés sont sélectionnés d'abord, et les vols les plus tardifs qui participent à cette surcharge sont repoussés après le secteur-période considéré. La solution obtenue n'est pas optimisée. Pour les modèles à fenêtres glissantes et par tri, les variables de retard (D_i) sont étiquetées par date de décollage croissante.

Les tests ont été menés avec une précision ϵ (unité de temps) de 5 min et un retard maximum de 60 min, exceptés ceux où sont précisées des valeurs différentes.

Secteur unique

Afin d'analyser les solutions produites par les différents modèles, nous nous sommes intéressés à une instance extrêmement simple du problème avec un seul secteur et aucune variation de capacité. Nous avons choisi le secteur le plus chargé (644 vols) du trafic réel de la journée du 20 mai 1999 dans l'espace aérien français. La capacité horaire a été fixée à 40. Les vols concernés sont attendus dans le secteur entre 0h52 et 23h39.

Les premières expérimentations ont visé à montrer l'équivalence et l'ordre entre les différents modèles ; $\texttt{M1} \preceq \texttt{M2}$ signifie que `M2` est plus contraint que `M1`, i.e. que la solution pour `M2` est une solution pour `M1`, et $\texttt{M1} \equiv \texttt{M2}$ si et seulement si $\texttt{M1} \preceq \texttt{M2} \wedge \texttt{M2} \preceq \texttt{M1}$:

$$\begin{aligned} \texttt{Standard} &\equiv \texttt{Gcc} \\ \texttt{Standard} &\preceq \texttt{Tri} \\ \texttt{Standard} &\equiv \texttt{Glissant} \text{ avec } \sigma = \delta \\ \texttt{Glissant} &\preceq \texttt{Tri} \\ \texttt{Glissant} \text{ avec } \sigma = \epsilon &\equiv \texttt{Tri} \end{aligned}$$

Les résultats numériques du tableau 6.1 confirment cet ordre : un modèle plus contraint produit une solution de plus grand coût. On constate que la plupart des vols pour cet exemple ne sont pas ou très peu retardés (nombres des deux dernières colonnes à comparer au nombre total de vols, 644). Le retard moyen pour cet exemple est inférieur à la précision du calcul (5 min) pour toutes les solutions.

TAB. 6.1 – Retards induits par la régulation : influence du modèle et des paramètres

Modèle	δ	σ	$\sum D_i$	$\lvert\{D_i = 0\}\rvert$	$\lvert\{D_i \leq 15\}\rvert$
`Standard, Gcc`	60		690	602	624
`Standard, Gcc`	30		1960	532	595
`Glissant`	60	30	1760	549	597
`Glissant`	60	15	2480	504	565
`Tri, Glissant`	60	ϵ	3660	467	553

Cependant, le coût de la solution est un indicateur qui donne peu d'informations sur sa qualité. Les différences essentielles des quatre modèles sont qualitatives et s'observent sur le profil de la charge de trafic. La figure 6.8 montre cette charge pour le secteur unique considéré, c'est-à-dire le nombre de vols entrant pendant les prochaines δ minutes. La courbe en pointillés correspond au trafic non régulé (horaires prévus) et la courbe continue correspond à la solution.

À première vue, le modèle `Standard` ne régule presque aucun vol et ne lisse pas la charge de contrôle. Il faut regarder finement pour vérifier que ce modèle assure seulement que la courbe passe sous la limite de capacité à chaque début de période (entouré d'un cercle), c'est-à-dire toutes les δ minutes (ici toutes les heures).

La courbe pour le modèle `Tri` montre un résultat plus probant et attendu. La charge instantanée est pratiquement égale à la capacité maximale entre 5h00 et 20h00, et aucun dépassement n'est à déploré.

La figure 6.9 montre l'influence du paramètre σ sur le modèle `Glissant`. Pour $\sigma = \delta = 60$, la solution produite est celle du modèle `Standard`. Pour un σ inférieur (ici 15 min), on observe que la charge instantanée passe sous la capacité toutes les σ minutes mais qu'elle l'excède de 10% entre-temps.

La figure 6.10 montre le biais engendré par le modèle `Standard`. Pour cette expérimentation, nous avons réduit la capacité de 10% tout en augmentant le retard maximal autorisé pour les vols (120 min) afin de contraindre un nombre plus important de vols à être retardés. La figure donne le nombre de vols présents dans le secteur à tout instant

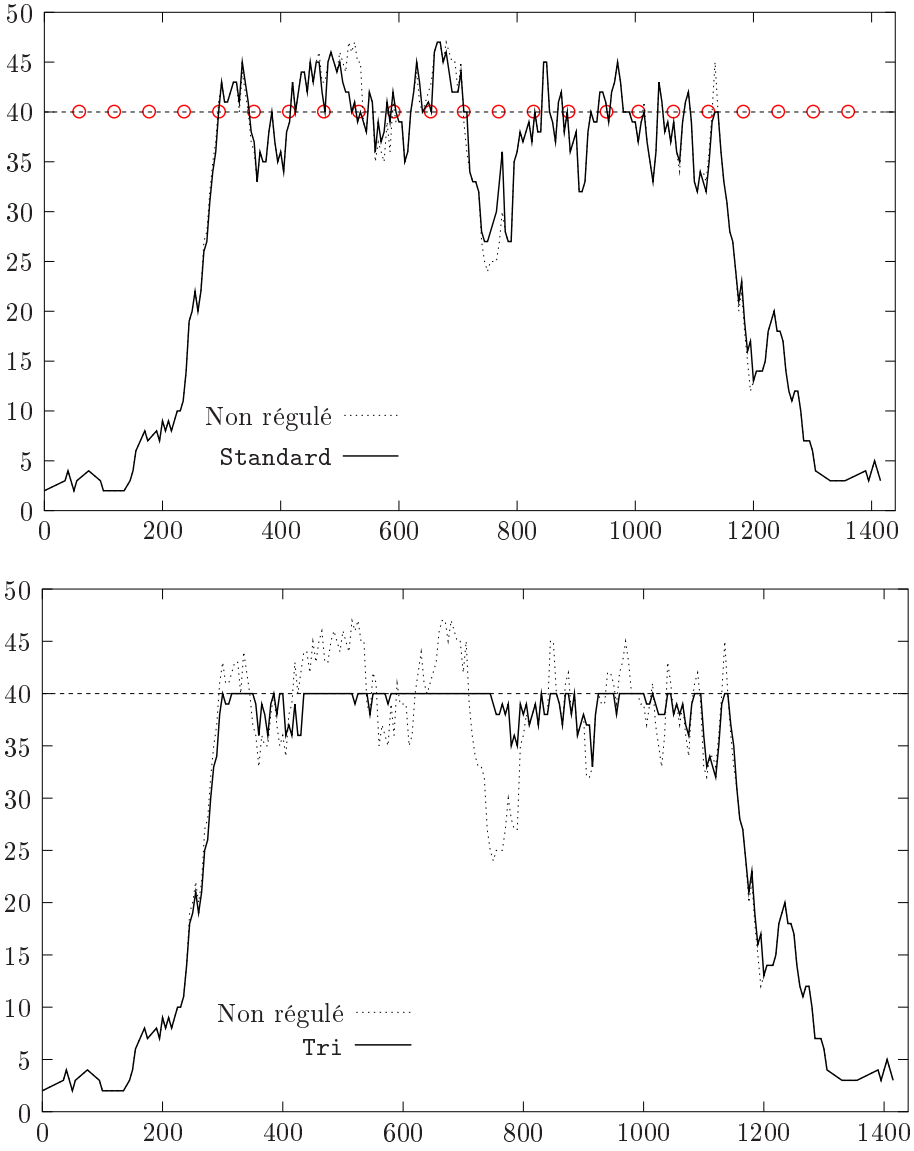


FIG. 6.8 – Régulation avec les modèles `Standard` (en haut) et `Tri` (en bas). Nombre de vols entrant dans le secteur pendant les prochaines $\delta = 60$ minutes en fonction du temps.

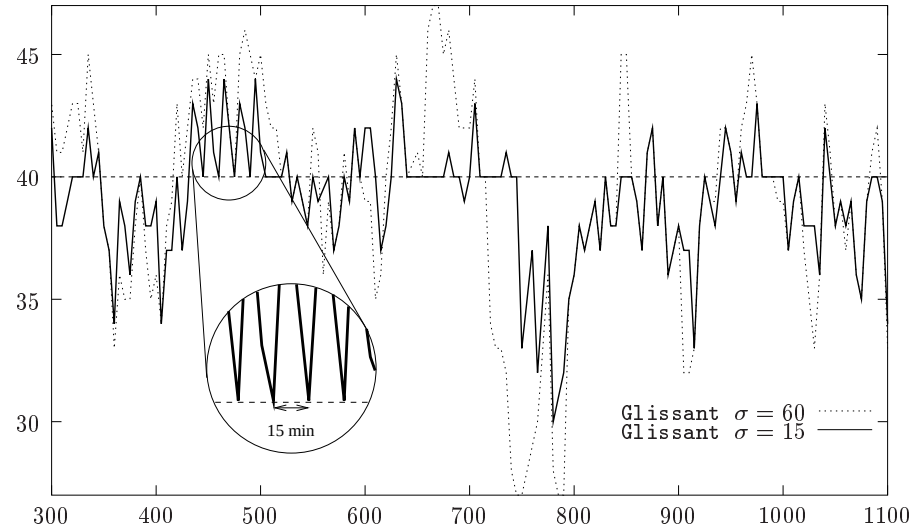


FIG. 6.9 – Régulation avec le modèle `Glissant` : influence de σ (zoom sur la période 5h-19h20). Le courbe tombe sous la capacité maximale toutes les σ min.

TAB. 6.2 – Preuve d'échec pour une instance sur-contrainte

Modèle	Capacité	$\sum D_i$
`Standard`	36	∞
`Gcc`	36	∞
`Glissant` ϵ	36	∞
`Tri`	36	Échec prouvé
`Gcc`	37	7390
`Tri`	37	19775

(on observe au passage qu'un aéronef passe peu de temps dans le secteur). Des pointes apparaissent à chaque début de période ; un vol qui doit être retardé doit passer dans la prochaine période non saturée, or une minimisation du retard correspond à faire passer le vol en début de période. La solution produite par le modèle `Tri` n'a pas cet inconvénient et assure une charge régulière.

Le tableau 6.2 indique la possibilité d'effectuer des preuves d'échec pour les différents modèles. Pour une capacité réduite de 10% (36), l'échec ne peut pas être prouvé en « temps fini » (∞ dans la table) excepté pour le modèle `Tri`. On observe également que le problème semble posséder une transition de phase car il est facile de trouver une solution pour une capacité légèrement supérieure (37).

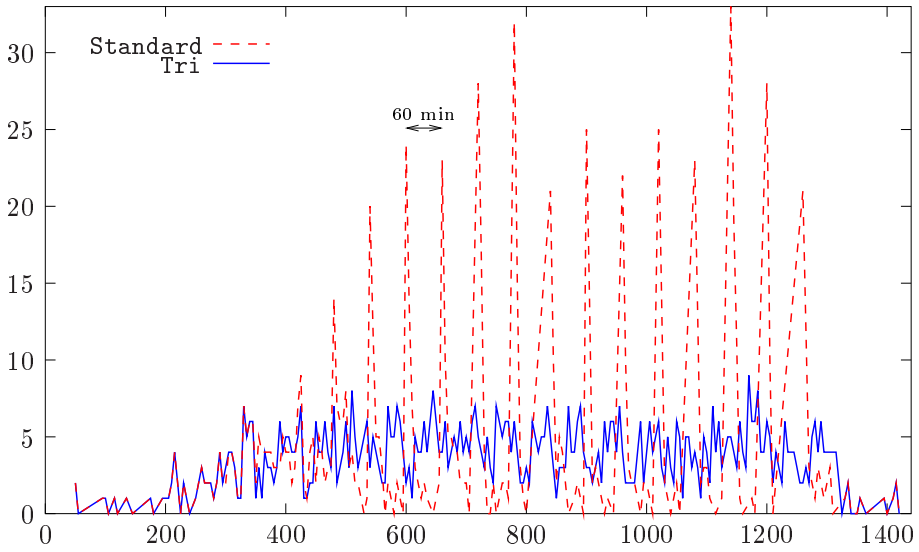


FIG. 6.10 – Nombre instantané d'avions présents dans le secteur (capacité = 36 ; retard max = 120). Le modèle `Standard` produit des pointes de trafic au début de chaque période alors que le modèle `Tri` garde une charge plus régulière.

Problème complet

Nous avons obtenu une preuve d'échec avec le modèle `Tri` pour l'instance complète munie des contraintes de capacité spécifiées. Une solution peut cependant être obtenue en autorisant des dépassements de capacité sur les secteurs de contrôle. Nous avons ainsi trouvé une solution avec une surcharge de 25%.

La figure 6.11 donne un aperçu du trafic traversant le centre de contrôle de Brest pendant la journée entière. Cette figure est divisée selon l'axe vertical en différents secteurs et le temps est représenté horizontalement. Un secteur-période est ainsi représenté par une boîte dans laquelle figure la courbe du nombre de vols qui doivent entrer dans le secteur dans les prochaines δ =60 min. Seuls trois secteurs sont ouverts dans les premières heures de la journée (en haut à gauche), puis 5 secteurs les remplacent, auxquels viennent se rajouter 4 autres secteurs quelques heures plus tard (et l'un des précédents est supprimé simultanément), etc. La régulation n'apporte pas ici les effets qualitatifs remarquables que l'on peut observer parmi les instances partielles, car le problème est bien trop complexe pour avoir des propriétés locales évidentes. D'autres centres de contrôle (Aix, Bordeaux) sont encore plus complexes (grand nombre de secteurs et de configurations possibles).

Cependant, on peut remarquer la saturation du secteur `J` entre 9h et 12h (secteur-période entouré). La figure 6.12 compare l'effet du modèle `Standard` et du modèle `Tri` sur ce même secteur-période. Ce secteur est très chargé et la régulation produite par le modèle `Standard` a peu d'effet sur la surcharge observée pour le trafic non régulé (la capacité maximale apparaît en pointillés). En revanche, le modèle `Tri` engendre un profil de charge qui réalise un plateau qui se superpose à la capacité maximale sans jamais la dépasser, ce

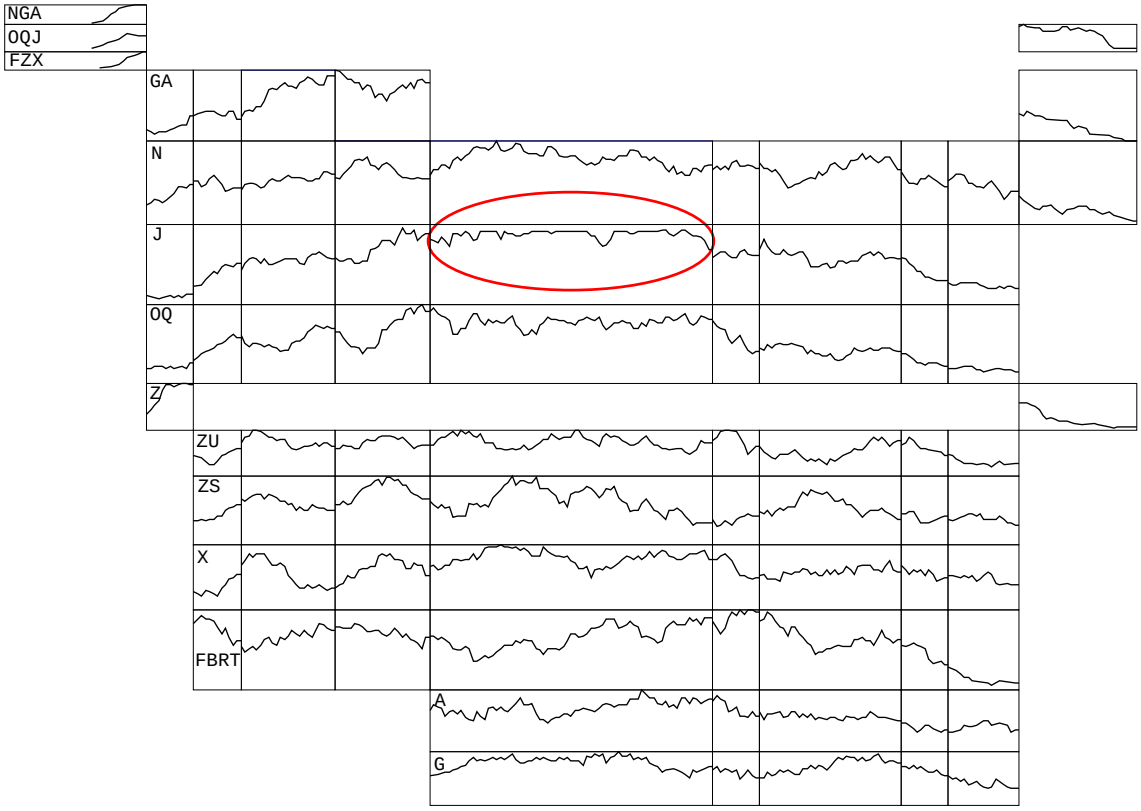


FIG. 6.11 – Espace aérien du centre de contrôle de Brest - 20 mai 1999

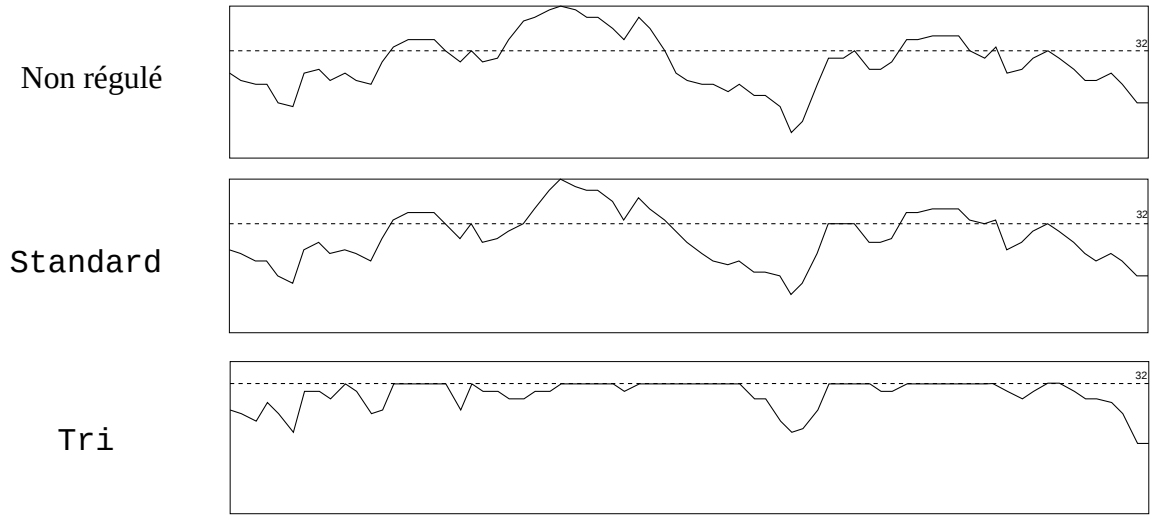


FIG. 6.12 – Centre de Brest, secteur J entre 9h et 12h

qui confirme les propriétés de lissage attendues de notre modèle.

6.5 Conclusion

L'allocation de créneaux pour l'ATFM est un problème d'optimisation combinatoire difficile mal résolu par le système qu'utilise actuellement la CFMU. Les contraintes de capacité des secteurs de contrôle ne sont pas clairement définies et la Programmation par Contraintes permet d'en modéliser facilement différentes interprétations qui accordent plus ou moins d'importance à la régularité de la charge de travail des contrôleurs. Le modèle à fenêtres de temps contiguës fournit ainsi des solutions peu réalistes dans lesquelles apparaissent des pics de trafic qui amènent probablement les Centres de Contrôle à sous-estimer leur capacité réelle. En revanche, les modèles à fenêtres glissantes (de pas minimal ϵ) ou par contrainte de tri assurent une charge de travail régulière en produisant des problèmes beaucoup plus contraints menant à des solutions plus coûteuses (en terme de somme des retards attribués). Le modèle dual par tri nécessite moins de paramètres de réglages que le modèle à fenêtres de temps et procure des preuves d'échecs efficaces quand le problème est sur-contraint, avec des temps de calcul comparables.

La grande taille des données du trafic aérien - nombre de vols et taille des domaines - rend le problème d'allocation de créneaux difficile à optimiser suivant le critère de la somme totale des délais attribués, mais des solutions optimales sont obtenues avec le retard maximal comme critère de minimisation. D'autre part, l'intégration dans la fonction de coût de divers facteurs tels que la régularité du trafic (qui est prise en compte dans la modélisation du problème par notre approche) est une nécessité opérationnelle qui pénaliserait fortement les modèles qui minimisent la somme des retards.

Chapitre 7

Schéma d'ouverture des centres de contrôle

Pour adapter la capacité des Centre de Contrôle en route au trafic aérien prévu, des plannings, appelés *schémas d'ouverture déposés*, sont élaborés empiriquement par des experts. Chaque centre est découpé en secteurs élémentaires associés à une position de contrôle et dotés d'une certaine capacité, et ils peuvent être regroupés selon des configurations prédéfinies pour diminuer le nombre de positions ouvertes, au détriment de la capacité globale. Nous avons implémenté le modèle de [Gianazza 02] en Programmation Par Contraintes pour résoudre ce problème de partitionnement en minimisant les surcharges de trafic, puis nous proposons de l'enrichir en tenant compte de la complexité des transitions entre configurations successives pour rendre les solutions plus « réalistes » d'un point de vue opérationnel. Le programme concis écrit avec FaCiLe s'avère bien plus efficace qu'un Branch & Bound *ad hoc* et le second modèle, plus contraint, génère des solutions plus régulières que le premier. Néanmoins, on peut émettre des réserves sur l'intérêt opérationnel d'optimiser les schémas d'ouverture déposés sur la base des capacités déclarés car les schémas d'ouverture *réalisés* en diffèrent radicalement.

7.1 Introduction

La première étape du processus d'ATFM consiste à élaborer le planning de chaque Centre de Contrôle un jour ou deux à l'avance. Les prévisions de trafic calculées à l'aide des plans de vol déposés permettent d'ajuster la configuration des secteurs de contrôle suivant les ressources disponibles (nombre de contrôleurs présents) durant la journée et de prévoir d'éventuelles surcharges ; le cas échéant, la CFMU régule les flux de trafic incriminés en retardant les avions pour respecter les contraintes de capacité.

Ces « schémas d'ouverture » des centres de contrôle sont définis par un expert (Flow Manager Position) qui dispose d'outils d'analyse pour détecter les surcharges engendrées par le trafic prévu sur une configuration particulière. Mais les choix du FMP sont empiriques car ces outils ne proposent pas de procédures d'optimisation de ce problème de

partitionnement, par exemple pour minimiser les dépassements de capacité ou le nombre de positions activées.

[Gianazza 02] propose différents algorithmes pour résoudre ce problème : un algorithme génétique, un A^* et un Branch & Bound. Ce dernier surpasse aisément l'algorithme génétique car le nombre de configurations prédéfinies n'est pas trop grand, ainsi que le A^* parce que son heuristique ne fournit pas de bonnes estimations. Cependant, ce Branch & Bound *ad hoc* n'élague l'arbre de recherche que sur la contrainte de partition et le coût de la solution, et n'est pas facile à enrichir pour prendre en compte des contraintes annexes comme la « faisabilité » des transitions entre configurations successives.

Nous reprenons en partie le modèle de [Gianazza 02] et intégrons au système utilisé un programme en FaCiLe pour comparer les performances de notre librairie au Branch & Bound *ad hoc*. Nous proposons en outre d'intégrer dans le coût de la solution des critères qui caractérisent le réalisme des transitions entre configurations successives.

Le problème des schémas d'ouverture des centres de contrôle a également été étudié par [Delahaye 95] qui propose un algorithme génétique pour le résoudre, mais sans l'appliquer à des données de taille réelle. Cependant, nous pensons que le problème est trop contraint et sa taille trop faible pour que ce type d'algorithme soit plus efficace qu'une recherche énumérative.

L'étude présentée dans [Verlhac 01] est plus proche de la nôtre. Un modèle en programmation entière mixte est utilisé pour minimiser les dépassements de capacité et le nombre de positions ouvertes. Puis des raffinements successifs modélisent les effets macroscopiques de l'allocation de créneaux par des transferts de charge de trafic dans le temps puis également dans l'espace en considérant le problème globalement pour tous les centres de contrôle. De très bon gains de performances sont calculés en comparant les délais produits par l'allocation des créneaux avant et après l'optimisation. Mais seule la variation des fréquences de changement de configuration est prise en compte dans cette étude alors que nous essayons de minimiser la complexité des transitions.

Le problème des schémas d'ouverture est détaillé dans la section 7.2 puis modélisé avec des contraintes dans la section suivante. La section 7.4 présente la stratégie de recherche utilisée pour obtenir les schémas optimaux analysés dans la section 7.5. La conclusion se garde enfin d'être hâtive quant aux gains espérés par cette optimisation en resituant le problème dans son cadre opérationnel.

7.2 Description du problème

Un Centre de Contrôle est découpé en secteurs élémentaires (par exemple 10 pour Reims et 28 pour Aix) qui correspondent à des positions de contrôle (assurées en générale par deux contrôleurs chacune). Chaque secteur est associé à une capacité exprimée en nombre d'avions par heure. Suivant le nombre de contrôleurs disponibles et les flux de trafic, ces secteurs peuvent être regroupés pour former une partition du Centre de Contrôle qui nécessite moins de positions ouvertes (une par groupe), chaque groupe étant doté d'une capacité spécifique inférieure à la somme des capacités des secteurs élémentaires qui le

TAB. 7.1 – Partitions des Centres de Contrôle français (mai 1999)

Centre	Secteurs	Groupes	Configurations	Partitions
Aix (AIX)	28	82	$20,5\,10^6$	$6\,10^{21}$
Bordeaux (BORD)	24	73	$3,1\,10^6$	$446\,10^{15}$
Brest (BRST)	18	40	$17,5\,10^3$	$682\,10^9$
Paris Est (PE)	11	24	426	$678\,10^3$
Paris Ouest (PW)	11	18	274	$678\,10^3$
Reims (REIM)	10	14	118	$116\,10^3$

composent.

Les contrôleurs peuvent ainsi régler finement leur charge de travail en regroupant ou en dégroupant dynamiquement les secteurs de contrôle qui leur incombent suivant la complexité du trafic qui les traversent. Parmi toutes les partitions possibles d'un Centre de Contrôle, seules celles qui utilisent les groupes de secteurs prédéfinis par les procédures de la Circulation Aérienne peuvent être utilisées, ce qui limite fortement le nombre de solutions possibles. Le tableau 7.1 montre la taille du problème pour les six centres français avec le nombre de configurations possibles combinant les regroupements prédéfinis et le nombre de partitions totales[1].

Les *schémas d'ouverture* quotidiens de chacun des six Centres de Contrôle français sont planifiés un jour ou deux à l'avance par un expert (Flow Manager Position) en découpant la journée par tranches d'une heure en moyenne[2]. Le FMP établit manuellement ce schéma en choisissant pour chaque tranche une configuration (partition) parmi celles qui sont prédéfinies, chaque secteur et groupe de secteurs étant associé à une capacité donnée (nombre d'avions par heure). De plus, le nombre de positions ouvertes (secteurs ou groupes de secteurs) doit rester inférieur à l'effectif des contrôleurs durant la tranche horaire considéré. Un logiciel lui permet alors de visualiser si le schéma d'ouverture respecte les charges de trafic prévues qui sont calculées en fonction des plans de vol déposés.

La figure 7.1 illustre la partition d'un centre de contrôle fictif (inspiré de la partie supérieure du centre de Paris-Est) comptant 5 secteurs et 4 regroupements prédéfinis. La configuration de gauche (ASUP, UTJ) ne demande que 2 positions ouvertes alors que celle de droite (UP, UTS, ARJ) en nécessite 3 et devra donc être éventuellement éliminée si trop peu de contrôleurs sont disponibles dans la période considérée.

Les capacités annoncées par les schémas d'ouverture sont en fait très souvent dépassées lors des pointes de trafic. Pour limiter ces surcharges, une phase ultérieure de planning est réalisée par le Control Flow Management Unit. Cet organisme centralise plans de vols et schémas d'ouverture et régule les flux d'avions supposés les plus pénalisants en les retardant

[1] Ces partitions ne sont pas toutes réalistes : seuls des volumes convexes peuvent être exploités et beaucoup d'autres critères opérationnels doivent être pris en compte pour les concevoir.

[2] Les périodes de trafic faible et stable peuvent être définies pour une durée de quatre heures consécutives et le grain minimal est d'une demi-heure.

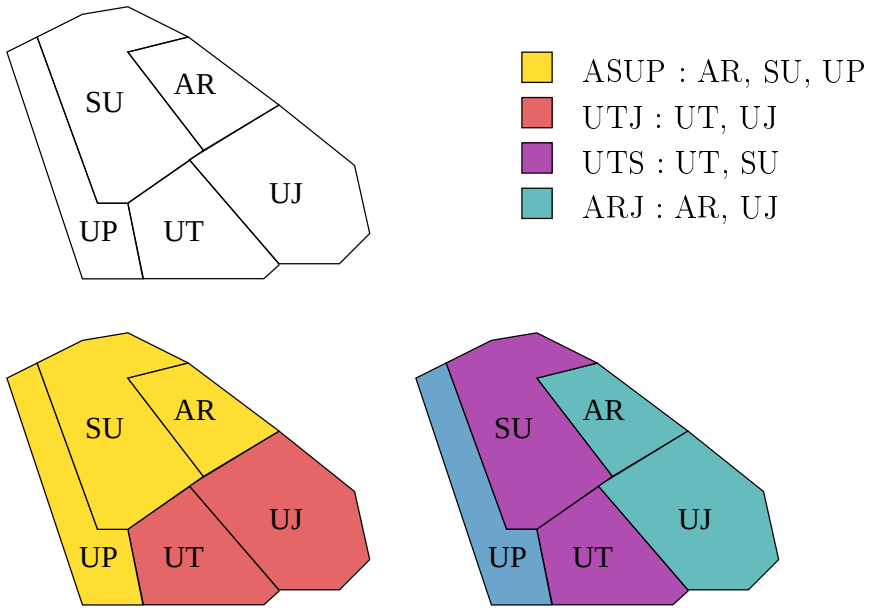


FIG. 7.1 – Deux partitions d'un centre de contrôle (fictif)

(allocation de créneaux de décollage). Cette étape est également effectuée manuellement par des experts à l'aide d'outils d'analyse qui permettent de simuler les régulations choisies et d'en visualiser les effets. Ce deuxième problème fait l'objet du chapitre 6 avec pour objectif de lisser la charge de travail des contrôleurs.

Ce processus de régulation en deux étapes répond à de nombreuses contraintes opérationnelles difficiles à modéliser et l'intervention d'experts y est donc indispensable. Cependant, les outils employés ne proposent pas de procédures d'optimisation automatiques qui pourraient faciliter grandement la tâche de l'opérateur et fournir des solutions bien meilleures. Nous proposons donc de résoudre le problème des schémas d'ouverture avec un programme en contraintes en reprenant la modélisation de [Gianazza 02] qui minimise l'écart entre charge de travail et capacité. Puis nous raffinons ce modèle pour obtenir des solutions plus réalistes vis-à-vis des transitions lors des regroupements et dégroupements de secteurs.

L'idéal serait bien sûr de résoudre les deux problèmes simultanément, mais leurs tailles sont trop importantes pour espérer obtenir l'optimum global en un temps raisonnable. Néanmoins, en combinant la recherche de solution optimale pour le problème des allocations de créneaux et celle des schémas d'ouverture, la capacité du système de contrôle entier devrait être amélioré tout en respectant les contraintes du problème.

Notons que les schémas d'ouverture *réalisés*, c'est-à-dire la trace des configurations successives des centres de contrôle lors de la journée de trafic, sont toutefois très différents des schémas d'ouverture *déposés* la veille : comme l'illustre la figure 7.2, les schémas réalisés sont en général plus dynamiques (il peut survenir une dizaine de changements de configuration en moins d'une heure) et utilisent jusqu'à deux fois moins de positions ou-

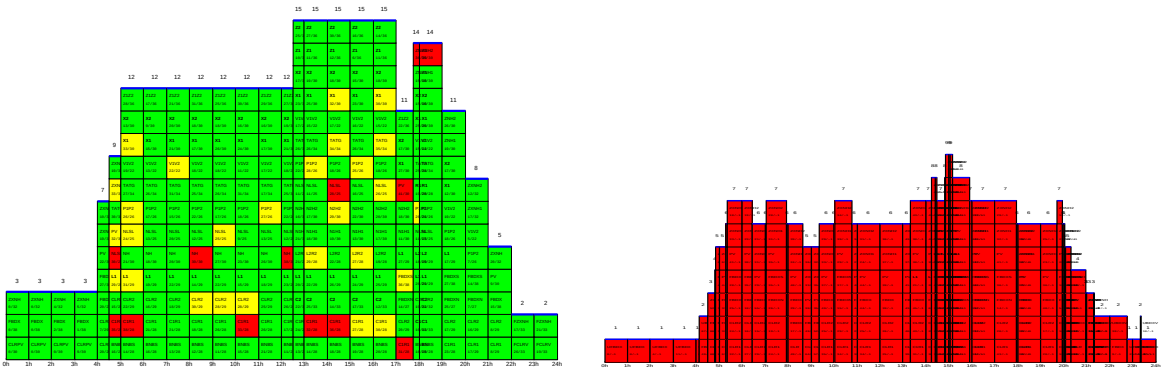


FIG. 7.2 – Schémas d'ouverture déposés (à gauche) et réalisés (à droite) pour le centre de Bordeaux (20/05/1999). L'axe horizontal correspond au temps et la hauteur donne le nombre de secteurs ouverts. Les secteurs en rouge sont surchargés.

vertes (chaque secteur pouvant être largement surchargé). De même, de nombreux aléas viennent interférer avec les créneaux de décollage alloués. Si bien que les spéculations sur les gains de capacités ou la réduction des retards observés dans cette phase de planning ne peuvent être extrapolées directement sur les performances réalisées. La sensibilité aux incertitudes des solutions optimales calculées pour l'allocation de créneaux a ainsi été étudiée par [Rivière 01a] qui conclut à leur robustesse.

7.3 Modélisation

Les données aéronautiques pour un jour particulier ainsi que les plans de vol déposés permettent de précalculer les charges de trafic, les capacités et le nombre de contrôleurs disponible pour chaque centre de contrôle de l'espace aérien français. Dans la suite, on confondra parfois *secteur* et *groupe de secteurs* quand la distinction n'est pas pertinente[3].

Le problème est résolu indépendamment pour chacun des six centres de contrôle. La journée est découpée en périodes de durées variables (multiples d'une demi-heure) selon le schéma déposé par le FMP. Pour un centre donné, on connaît donc :

- $\mathcal{S}_e$: l'ensemble des secteurs élémentaires de cardinal n_e.
- $\mathcal{S}$: l'ensemble des groupes de secteurs utilisables de cardinal n, i.e. $\forall s \in \mathcal{S},\ s \subset \mathcal{S}_e$. Notons que tous les secteurs élémentaires y sont présents : $\forall s \in \mathcal{S}_e,\ \{s\} \in \mathcal{S}$.
- $w_{s,t}$: la charge de trafic pour le secteur $s \in \mathcal{S}$ pour la période t.
- $c_{s,t}$: la capacité maximale du secteur $s \in \mathcal{S}$ pour la période t.
- p_t : le nombre de positions (secteurs) maximal autorisé pour la période t.

L'ensemble $\mathcal{S}$ des secteurs est représenté par une matrice S de $n_e \times n$ booléens qui spécifie les secteurs élémentaires contenus dans un secteur particulier. Par exemple, la matrice associée à un centre composé de quatre secteurs élémentaires $\mathcal{S}_e = \{a, b, c, d\}$ qui

[3]Un secteur est un groupe de secteurs réduit à un singleton.

peuvent être regroupés en neuf secteurs :

$$\mathcal{S} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a,b\}, \{a,c\}, \{b,d\}, \{c,d\}, \{a,b,c,d\}\}$$

sera représentée par la matrice suivante :

$$\left\{ \begin{array}{ccccccccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array} \right\}$$

Chaque période est résolue indépendamment. Pour une période donnée, la partition du centre de contrôle sera spécifiée en associant une variable de décision booléenne v_j ($j \in [1, n]$) à chaque secteur s_j qui indique s'il fait partie ou non de la solution, c'est-à-dire qui sélectionne la colonne correspondant à ce secteur si elle est égale à 1.

Pour contraindre la configuration à être une partition du centre de contrôle, le produit scalaire de chaque ligne de la matrice avec le vecteur des variables de décision doit être égale à 1 :

$$\forall i \in [1, n_e] \quad \sum_{j=1}^{n} S_{i,j} v_j = 1$$

Une contrainte redondante sur les cardinaux des secteurs retenus est systématiquement ajoutée pour améliorer l'efficacité du modèle :

$$\sum_{j=1}^{n} |s_j| v_j = n_e$$

La dernière contrainte assure que le nombre de secteurs ouverts est inférieur au nombre de positions disponibles dans la période t considérée :

$$\sum_{j=1}^{n} v_j \leq p_t$$

On aurait pu également modéliser ce problème de partition avec n_e variables ensemblistes comme le suggère [Gervet 97], car la partition qui compte le plus de secteurs ouverts correspond à prendre tous les secteurs élémentaires indépendamment :

$$\forall i \in [1, n_e] \quad \emptyset \subseteq v_i \subseteq \mathcal{S}_e$$

Une contrainte de partition est ensuite posée sur ces variables et on limite le nombre de secteurs ouverts en comptant le nombre de v_i non-vides. Néanmoins, les performances de ce modèle sont médiocres et le coût (voir la section suivante) est difficile à exprimer car notre fonction n'est pas une fonction *graduée*[4] au sens de [Gervet 97], donc la contrainte

[4]Une fonction graduée associe un coût à chaque élément d'un ensemble d'ensembles $\mathcal{S}$ et doit vérifier :

$$\forall s_1, s_2 \in \mathcal{S} \quad s_1 \subset s_2 \Rightarrow f(s_1) < f(s_2)$$

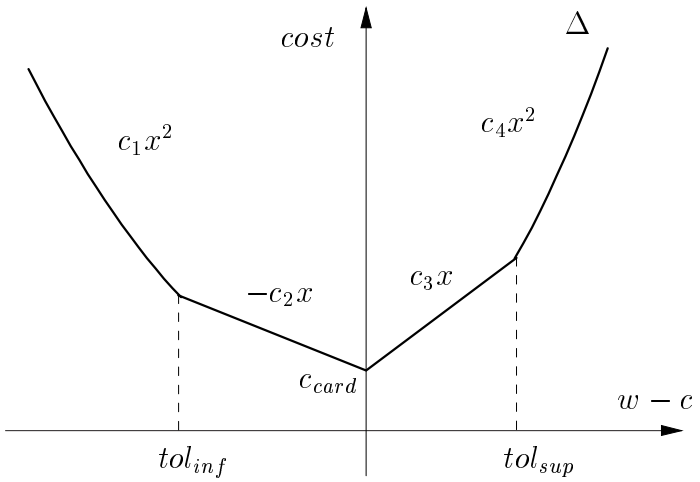


FIG. 7.3 – Coût associé à l'écart entre la charge et la capacité d'un secteur

de poids proposée par [Gervet 97] n'est pas adéquate. Conséquemment, cette modélisation n'a pas été retenue.

7.3.1 Coût d'une configuration

L'objectif, inspirés de [Gianazza 02], est de minimiser la somme des écarts entre charges et capacités ($w_s - c_s$) pour les secteurs utilisés dans la partition. Les capacités des secteurs ne sont donc pas modélisées comme des contraintes (dures) mais sont incluses dans la fonction de coût. Une fonction définie par morceaux Δ dont le profil est représenté figure 7.3 pénalise fortement les surcharges (quadratiquement) et peu les faibles écarts (linéairement) compris dans les bornes d'une « tolérance » (inférieure tol_{inf} ou supérieure tol_{sup}) autour de la capacité maximale. Quatre coefficients positifs (c_{1-4}) permettent de faire varier l'importance relative des différents types d'écarts entre charge et capacité, et des constantes adéquates (a et b) rendent la fonction continue :

$$\Delta(x) = \begin{cases} c_1x^2 + a & \text{si} & & & x & < & tol_{inf} \\ -c_2x & \text{si} & tol_{inf} & \leq & x & < & 0 \\ c_3x & \text{si} & 0 & \leq & x & \leq & tol_{sup} \\ c_4x^2 + b & \text{si} & tol_{sup} & < & x & & \end{cases}$$

D'autre part, on cherche à diriger la recherche vers des solutions qui nécessitent le moins possibles de positions ouvertes. Le nombre de secteurs utilisés est donc intégré dans le coût avec un coefficient c_{card}, ce qui revient à ajouter une constante à la fonction Δ (cf. figure 7.3). Le coût à minimiser est finalement le suivant :

$$cost = \sum_{j=1}^{n} \left(\Delta(w_j - c_j) + c_{card}\right) v_j$$

7.3.2 Coût tenant compte des transitions

Le modèle précédent résout chaque période indépendamment les unes des autres et peut donc conduire à des solutions dans lesquelles deux configurations consécutives optimales sont très différentes. Opérationnellement, les contrôleurs ne peuvent passer d'une partition à la suivante qu'en regroupant ou dégroupant un faible nombre de secteurs seulement, suivant l'évolution de la charge de trafic. Les transitions proposées par les solutions du modèle précédent peuvent ainsi être trop complexes à réaliser.

Pour obtenir des configurations moins difficiles à mettre en œuvre, on peut inclure dans la fonction de coût la « différence » entre deux configurations successives à l'aide d'un nouveau coefficient c_{diff}. Le calcul de l'optimum pour la période t dépend alors de la configuration précédente :

$$cost_t = \sum_{j=1}^{n} \left(\Delta(w_{j,t} - c_{j,t}) + c_{card}\right) v_{j,t} \;+\; c_{diff} \sum_{j=1}^{n} \delta(v_{j,t}, v_{j,t-1})$$

$$\text{avec} \quad \delta(x,y) = \begin{cases} 1 & \text{si } x \neq y \\ 0 & \text{sinon} \end{cases}$$

On pourrait également envisager de résoudre le problème globalement en considérant toutes les périodes simultanément. La fonction δ poserait alors des contraintes réifiées de différence entre deux variables $v_{j,t}$ et $v_{j,t-1}$ alors que les $v_{j,t-1}$ sont des constantes quand les périodes sont résolues successivement. Cependant, la taille du problème devient trop importante — 30 fois plus grande en moyenne — pour pouvoir le résoudre de manière optimale.

7.4 Stratégie de recherche

La stratégie de recherche utilisée pour obtenir les résultats présentés dans la section suivante effectue un étiquetage des variables booléennes de décision en les ordonnant de manière statique. Comme on connaît la contribution de chaque secteur dans le coût s'il est choisi pour faire partie de la configuration, on commence par sélectionner les secteurs de plus petit coût car on cherche à minimiser le coût global.

Mais ce critère ne tient pas compte du nombre total de secteurs ouverts que l'on cherche également à minimiser. On modifie donc cet ordre en calculant pour chaque secteur le ratio du coût partiel de chacun avec son cardinal, c'est-à-dire le nombre de secteurs élémentaires qui le constituent :

$$\frac{\Delta(w_j - c_j)}{|s_j|}$$

On choisit d'abord la variable inconnue associée à la plus petite valeur de ce critère et on l'instancie à 1 (i.e. on sélectionne le secteur). Cette stratégie s'avère plus efficace que la première mentionnée quelque soit le type de coût global utilisé.

On pourrait également commencer par sélectionner les secteurs déjà utilisés lorsqu'on optimise en tenant compte de la solution précédente pour réduire les différences entre

TAB. 7.2 – Temps de calcul des schémas optimaux en secondes (P-IV à 1,7 GHz) pour la journée du 20 mai 1999

Centre	Standard	Transitions	[Gianazza 02]
AIX	2.6	9.1	42
BORD	0.68	0.93	6.1
BRST	0.4	0.80	1.7
PE	0.02	0.1	0.07
PW	0.05	0.12	0.08
REIM	0.05	0.09	0.05

configurations successives (cf. section 7.3.2). Mais les variations de charge sont (systématiquement) très soudaines vers 4h et 20h, si bien que le nombre de secteurs ouverts augmente ou diminue drastiquement. Il faudrait donc utiliser un premier critère pour adopter cette stratégie ou conserver la précédente suivant les variations de trafic entre périodes consécutives.

7.5 Résultats

Les modèles présentés ont été implémentés avec FaCiLe et intégrés au système utilisé par [Gianazza 02] qui propose notamment une procédure d'optimisation complète par un algorithme de Branch & Bound *ad hoc*. Les résultats obtenus avec la fonction de coût standard imitant celle de [Gianazza 02] sont identiques dans la plupart des cas à ceux calculés par le Branch & Bound *ad hoc* : en effet, le profil de notre fonction est similaire mais [Gianazza 02] utilise des coefficients flottants (qui entraîneraient un dépassement des entiers (32 bits) de FaCiLe) pour établir une hiérarchie entre les diverses composantes de la fonction de coût et de faibles différences apparaissent sur certaines journées de trafic.

Les temps de calculs sont toujours très courts : de l'ordre de 2 s avec la coût standard pour le centre le plus complexe (AIX) sur les journées testées (20 à 30 fois plus rapide que la procédure *ad hoc*) et environ 10 s pour le modèle qui tient compte des transitions. Le tableau 7.2 montre les temps d'exécution sur un Pentium IV à 1,7 GHz pour chaque centre avec les plans de vol de la journée du 20 mai 1999 (les dimensions des problèmes sont celles indiquées sur la figure 7.1). L'implémentation de notre modèle avec FaCiLe est donc très efficace.

La figure 7.4 présente (de haut en bas) les schémas d'ouverture obtenus avec la fonction de coût standard (identique à celui généré pas le Branch & Bound *ad hoc*), puis en tenant compte des transitions et enfin celui conçu empiriquement par le FMP, ceci pour le centre d'Aix le 20 mai 1999. Sur cette représentation, l'axe horizontal correspond au temps et chaque boîte désigne un secteur ouvert où figurent son nom (en gras s'il s'agit d'un secteur élémentaire), sa charge et sa capacité ($w_{s,t}$ / $c_{s,t}$). Les secteurs surchargés au-delà de la

« tolérance supérieure » sont colorés en rouge, ceux dont la charge se situe entre les bornes des tolérances inférieure et supérieure sont en jaune, et enfin les secteurs sous-chargés en vert[5]. Le nombre total de secteurs ouverts est noté pour chaque période au dessus de la configuration ainsi que le nombre de positions disponibles qui est représenté par un trait (bleu si la limite est atteinte et noir sinon).

Avec le coût standard, on obtient un schéma optimal qui nécessite beaucoup moins de positions que le schéma déposé par le FMP, mais le nombre de secteurs ouverts varie fortement au cours de la journée et les configurations successives sont très différentes les unes des autres. Ce schéma serait vraisemblablement trop complexe à réaliser car chaque dégroupement et regroupement de secteurs nécessitent que les contrôleurs s'échangent des informations.

Avec le modèle tenant compte des transitions, le profil du schéma est lissé et les changement de configuration sont beaucoup moins complexes, le nombre de secteurs ouverts restant inférieur au schéma du FMP (mais souvent supérieur à ceux du premier schéma). Ce type de schéma serait donc plus réaliste à proposer au FMP comme outil d'aide à la décision.

On peut également noter que quelque soit la méthode utilisée (y compris celle du FMP), des dépassements de capacité subsistent. En effet, remplacer les termes du coût liés au surcharge par des contraintes (« dures ») résulte le plus souvent en un problème sur-contraint (et donc sans solution). Cependant, [Gianazza 02] montre que ces schémas d'ouverture optimaux, combinés avec des allocations de créneaux préalables telles que décrites dans le chapitre 6 sur un centre pour lequel tous les secteurs élémentaires sont ouverts[6], font disparaître les dépassements de capacité. De plus, [Gianazza 02] obtient un gain élevé sur la somme et le maximum des retards générés par rapport à une allocation fondée sur les schémas déposés par les FMP, et également sur les temps d'activité des positions de contrôle.

7.6 Conclusion

Nous avons utilisé FaCiLe pour substituer un programme en contraintes à un algorithme de Branch & Bound *ad hoc* qui génère des schémas d'ouverture optimaux pour les centres de contrôle français. Le modèle PPC très concis construit des schémas identiques avec des performances bien meilleures en temps de calcul (plus d'un ordre de magnitude sur les instances de plus grandes tailles). D'autre part, le modèle standard a pu être facilement modifié pour prendre en compte le réalisme des schémas générés vis-à-vis des transitions

[5] Avec un exemplaire imprimé en noir et blanc, le rouge apparaît le plus foncé, le jaune le plus clair et le vert est la teinte de gris intermédiaire.

[6] Cette configuration ne correspond pas forcément au maximum de capacité. En effet, un secteur AB, doté d'une capacité de 40 et composé des secteurs élémentaires A et B de capacité 25 chacun, peut faire passer une charge de 30 avions. S'il est dégroupé et que tout le flux de trafic traverse les deux secteurs A et B, la charge reste de 30 et la capacité de chacun est dépassé. En revanche si 15 avions traversent uniquement A et les 15 autre uniquement B, la charge reste en dessous des capacités et on obtient bien une capacité globale plus élevée.

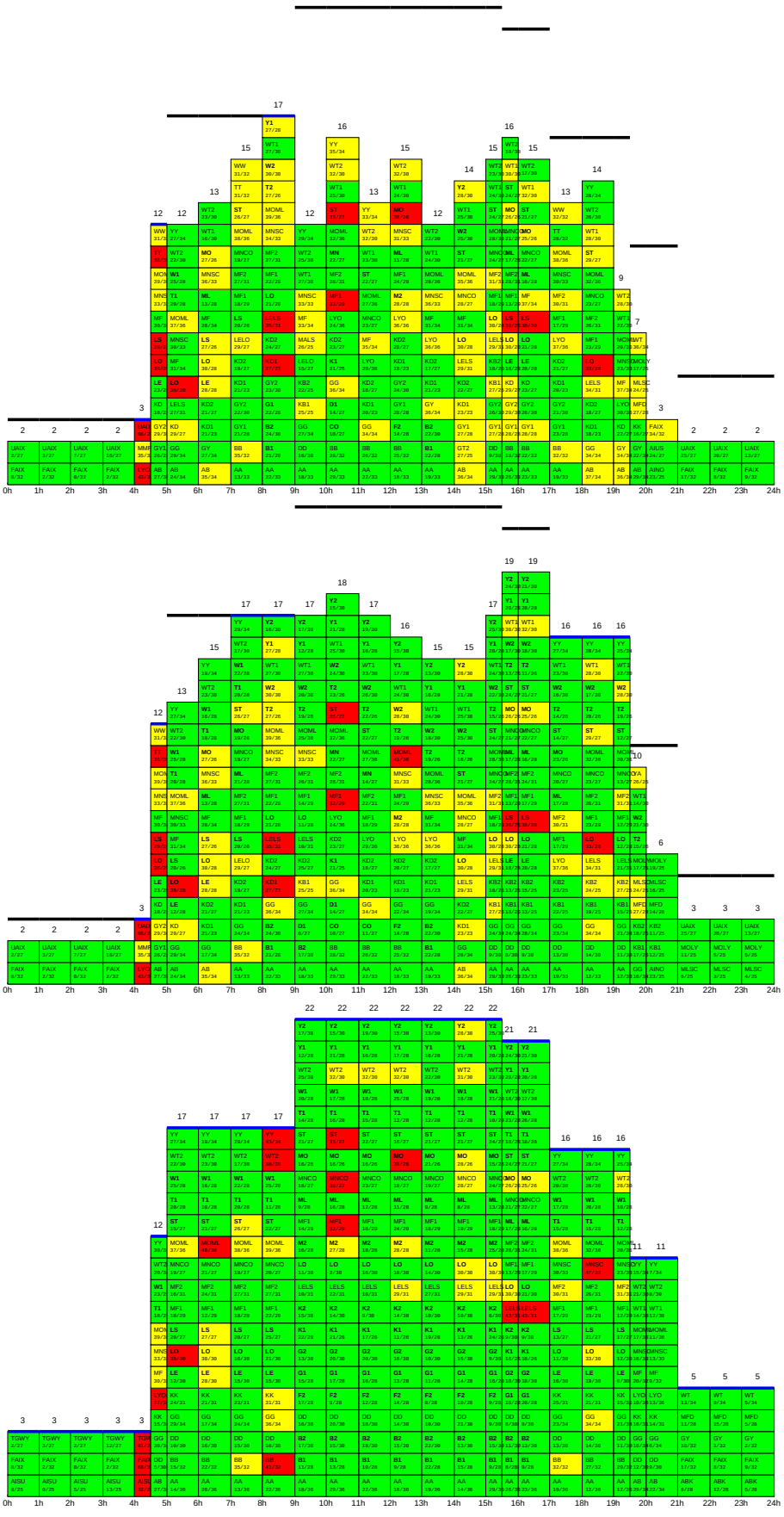


FIG. 7.4 – Schémas d'ouverture standard, avec transitions et FMP pour le centre d'Aix

entre configurations successives. Les schémas ainsi obtenus s'avèrent moins complexes et ont un profil de nombre de positions ouvertes plus lisse, mieux adapté à la gestion d'une salle de contrôle.

Cependant, ce modèle ne prend pas strictement en compte les contraintes opérationnelles des transitions entre configurations qui sont réalisés par dégroupements ou regroupements de secteurs suivant la charge de travail des contrôleurs. Mais les schémas déposés par les FMP des centres de contrôles sont également très différents de ceux qui sont observés lors de la journée de trafic : d'une part, ces derniers sont beaucoup plus dynamiques car les contrôleurs doivent s'adapter aux trafic réel qui diffère sensiblement des plans de vols déposés ; d'autre part, les données utilisées pour l'optimisation des schémas comme la charge de trafic horaire et les capacités annoncées des secteurs ne sont pas de bons indicateurs de la charge de travail des contrôleurs ni de la capacité réelle d'un secteur.

Les modèles utilisés s'efforcent de s'approcher des interprétations « déclarées » des données du trafic et de l'espace aérien mais l'observation du contrôle « en temps réel » suggère qu'elles sont peu fiables. Ainsi, la régulation du trafic par la CFMU à l'aide des schémas déposés par les FMP constitue un outil validé surtout par les résultats opérationnels du système de contrôle français. Les résultats de l'optimisation de ces processus, effectués empiriquement par des experts à l'heure actuelle, permettent donc difficilement de spéculer sur les gains effectifs de capacité et d'activité des contrôleurs qu'ils devraient apporter.

Chapitre 8

Réseau de routes directes

L'objectif de la régulation des flux de trafic aérien (ATFM : *Air Traffic Flow Management*) est d'améliorer la capacité de l'espace aérien tout en respectant les contraintes du contrôle (ATC : *Air Traffic Control*) *et* les demandes des compagnies qui cherchent à minimiser leur coût d'exploitation. Nous présentons dans la suite la conception d'un réseau de routes qui tente d'optimiser ces critères. L'idée principale est de ne considérer que des routes directes et de séparer verticalement les flux qui se croisent en leur allouant des niveaux de vol distincts. Ce problème est un coloriage de graphe que nous résolvons grâce à la programmation par contraintes et à un algorithme glouton chargé de trouver des cliques dans le graphe des contraintes afin de pouvoir poser des contraintes globales. Au cours de la recherche de solutions optimales minimisant le nombre de niveaux de vol distincts alloués, les symétries de permutation parmi les niveaux équivalents sont supprimées, et l'ordre d'instanciation des variables est guidée par les cliques découvertes durant la première étape statique. Avec une implémentation utilisant FaCiLe, l'optimalité est atteinte pour toutes les tailles de flux hormis la plus petite (flux de taille 1 qui correspondent au problème le plus grand), et le nombre de niveaux de vol obtenu reste compatible avec la structure actuelle de l'espace aérien. Néanmoins, de nombreuses autres contraintes devraient être ajoutées à ce modèle très simplifié pour obtenir un réseau de routes opérationnel, si bien que les conclusions présentées se rapporte essentiellement à la *faisabilité* du concept de séparation verticale des flux importants en route directe. Cette technique de coloriage de graphe a également été testée sur divers benchmarks et présente de bons résultats sur les instances applicatives, qui contiennent toutes des cliques de grande taille.

8.1 Introduction

Le réseau de routes français (et européen) a été construit de manière incrémentale au cours du temps par de petits ajouts et modifications locaux pour satisfaire la demande des utilisateurs et les contraintes de la régulation ATC (ainsi que d'autres types de contraintes comme l'évitement des zones militaires ou l'amélioration des moyens de navigation). Ce processus n'aboutit évidemment pas à une solution très efficace et conduit à sous-estimer

la capacité globale de l'espace aérien français, ainsi qu'à générer des déviations excessives par rapport aux routes optimales (directes).

Plusieurs études ont été et sont actuellement menées par Eurocontrol [Letrouit 98, Maugis 98] et le CENA [Mehadhebi 00] afin de trouver des solutions dans ce domaine. L'étude présentée ici concerne essentiellement la faisabilité de l'approche « idéale » suivante : ne considérer que des routes directes (i.e. un segment joignant origine et destination du vol) pour les flux d'avions d'une taille supérieure à un minimum donné[1], et séparer verticalement ceux qui s'intersectent pour éviter tout conflit, et donc leur résolution par l'ATC. Ce schéma ne prend en compte aucune autre contrainte opérationnelle, si bien que le modèle utilisé conduit à un problème de coloriage de graphe : colorier les nœuds d'un graphe avec le plus petit nombre de couleurs possibles de telle manière que deux nœuds adjacents (i.e. qui sont reliés par une arête) ne soient pas de la même couleur.

Le problème de coloriage de graphe est un problème NP-dur très classique et un sujet de recherche encore très actif, avec de nombreux domaines d'application [Trick 94]. Comme c'est un problème (NP-)difficile, notamment sur les graphes aléatoires (et donc peu structurés) de grande taille, les techniques heuristiques figurent en bonne place parmi les approches les plus performantes, telles que les algorithmes gloutons [Brélaz 79, Leighton 79] ou la recherche locale [Hertz 87, Morgenstern 86, Fleurent 94]. Cependant, les algorithmes énumératifs complets peuvent également être très efficaces [Coudert 97] pour les problèmes issus d'applications réelles car elles profitent de la structure du graphe, telle que la présence d'une clique[2] de grande taille qui fournit une borne inférieure et peut se colorier trivialement[3]. La PPC est un outil efficace et élégant pour implémenter ce type d'algorithme exact [Van Hentenryck 90] car les nœuds sont directement identifiés à des variables et les arêtes à des contraintes de différence. De plus, le Branch & Bound est l'algorithme de recherche standard dans les systèmes de PPC et la souplesse des buts de recherche facilite la conception d'heuristiques d'étiquetage. Enfin, des contraintes additionnelles peuvent facilement être ajoutées au modèle grâce au niveau d'abstraction élevé que procure la PPC.

L'idée principale de notre algorithme de coloriage est fondée sur la recherche de cliques de grande taille par un algorithme glouton dans un premier temps, afin de fournir une borne inférieure et d'alimenter des contraintes globales « tous différents » au sein d'un solveur PPC. Puis des contraintes de diséquation sont posées sur les flux restant et un solution optimale est recherchée par un algorithme standard de Branch & Bound. Pendant la recherche, les symétries entre couleurs sont éliminées en coupant les points de choix associés à l'instanciation d'un flux avec une couleur encore inutilisée. D'autre part, l'ensemble des cliques découvertes par l'algorithme glouton peut être utilisé pour guider la recherche.

La combinaison de ces techniques permet d'obtenir des preuves d'optimalité pour toutes les instances utilisées, exceptée la plus grande, avec des temps de calcul très courts. De plus, le nombre de niveaux de vol optimal pour des flux de taille significative pourrait

[1] La flexibilité maximum serait atteinte en prenant en compte tous les flux, donc avec une taille minimale d'un seul avion.

[2] Une clique est un sous-graphe complet : chaque nœud est relié à tous les autres par une arête.

[3] Tous les nœuds d'une clique doivent être coloriés de couleurs différentes car ils sont tous reliés les uns aux autres.

être utilisé dans l'espace aérien supérieur. Notre algorithme est également efficace sur des benchmarks d'applications réelles [Trick 94] et soutient bien la comparaison avec DSATUR [Brélaz 79], une méthode gloutonne classique de coloriage.

La description du problème d'allocation de niveaux de vol et sa modélisation en PPC avec des flux d'une taille minimale donnée est présentée section 8.2. La section 8.3 détaille ensuite l'utilisation de l'algorithme glouton pour trouver des cliques de manière à améliorer l'efficacité de l'inférence durant la recherche. La stratégie utilisée pour obtenir les résultats présentés section 8.5 est exposée dans la section précédente 8.4, de même que le but qui élimine dynamiquement les symétries et permet d'élaguer des parties importantes de l'arbre de recherche pendant l'optimisation. La section 8.6 discute de la pertinence de notre approche vis-à-vis de travaux similaires et se termine par les benchmarks. Les conclusions et perspectives de cette étude sont finalement présentées dans la dernière section.

8.2 Description du problème et modélisation

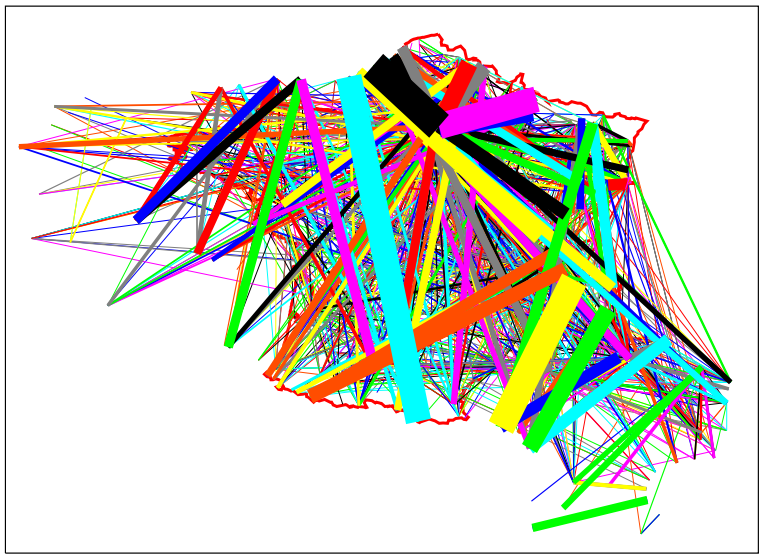


FIG. 8.1 – Routes directes (avec la taille des flux) dans l'espace aérien français.

La conception des routes dans l'espace aérien français et européen résulte d'un processus incrémental conditionné par la demande des utilisateurs et l'évolution des équipements ATC au sol, de l'avionique et des procédures de vol. La régulation des vols destinée à assurer la séparation des avions (typiquement 4-8 NM dans le plan horizontal et 1000-2000 ft verticalement) est prise en charge par les centres ATC qui sont divisés en de nombreux secteurs. La forme de ces secteurs permet aux contrôleurs de maîtriser la complexité du trafic et de résoudre les conflits qui peuvent survenir aux intersections du réseau de routes. Ce

processus ATC manque de souplesse vis-à-vis des compagnies aériennes qui souhaitent que leurs appareils dévient le moins possible de la route directe joignant origine et destination pour optimiser leurs coûts d'exploitation.

Un réseau idéal, i.e. des routes directes pour chaque flux, serait trop complexe à maîtriser pour des contrôleurs (humains) si les avions n'étaient pas séparés verticalement : les conflits qui apparaissent dans le plan horizontal formeraient une structure bien trop dense, comme le montre la figure 8.1 où la largeur de chaque routes est proportionelle au nombre de vols qui l'empruntent. L'idée principale est donc d'allouer des niveaux de vol différents aux routes qui se croisent pour éviter les conflits. Avec cette procédure, les vols qui partagent la même paire origine/destination utiliseraient séquentiellement la même route directe à une altitude fixée, distincte de celles de toutes les autres routes qui la coupent. En outre, seuls les avions dont les fenêtres de temps de vol se recouvrent sont pris en compte pour une éventuelle séparation verticale.

Notre modèle implémente cette stratégie à l'aide de la notion de *flux* : les vols qui partagent même origine et destination sont regroupés en un flux de taille égale au nombre de ces avions, et les dates de début et de fin de ce flux sont calculées en fonction des premiers et derniers vols concernés. Pour une journée donnée, les flux sont pré-calculés de sorte que les données de notre modèle sont :

- $\mathcal{F}$: l'ensemble des flux de cardinal $n = |\mathcal{F}|$;
- r_i : la route directe du flux i ;
- n_i : la taille du flux i ;
- t_i : la fenêtre de temps du flux i.

Les variables de décision sont les niveaux de vols :

$$\mathcal{L} = \{L_i, \forall i \in [1, n]\}$$

Leur domaine commun initial est $[1, n]$. Enfin, le coût à minimiser est le nombre de niveaux de vol distincts :

$$c = card(\mathcal{F})$$

dont le domaine initial est $[0, n]$.

Cependant, pour contrôler la taille du problème, l'ensemble des flux est filtré selon le paramètre n_{min} pour ne prendre en compte que les flux composés de plus de n_{min} vols. En fait, le trafic aérien se divise généralement en deux catégories principales : les vols commerciaux quotidiens qui correspondent à la plupart des flux de grande taille et qui sollicitent (et nécessitent) des niveaux de vol élevés, et les vols privés IFR[4] qui interviennent dans les petits flux et qui ne peuvent atteindre que de faibles niveaux de vols, si bien qu'ils interfèrent peu avec la première catégorie. Pour prendre en compte ce comportement opérationnel, il s'avère pertinent de ne résoudre le problème que pour des flux de plus de 5–10 avions, en considérant que les flux plus petits évoluent à une altitude inférieure au plus petit niveau de vol attribué aux flux principaux.

[4]Instrument Flight Rule : ce sont des vols pour lesquels un plan de vol doit être déposé et qui sont systématiquement contrôlés par l'ATC.

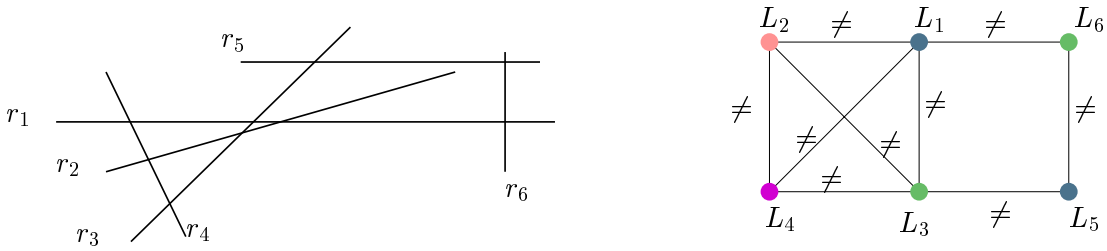


FIG. 8.2 – Transformation d'un réseau de routes aériennes en graphe de contraintes, où $\{L_1, L_2, L_3, L_4\}$ est une clique.

Le modèle en contraintes s'exprime directement : les contraintes de différence sont posées entre les niveaux des flux qui s'intersectent et dont les fenêtres de temps se chevauchent.

$$\forall i, j \quad 1 \leq i < j \leq n \quad intersecte(r_i, r_j) \wedge chevauche(t_i, t_j) \Rightarrow L_i \neq L_j$$

L'objectif est simplement de minimiser le nombre de niveaux distincts alloués, si bien que ce modèle est équivalent à un problème de coloriage de graphe.

La figure 8.2 montre un réseau de route et le graphe de contraintes correspondant obtenu en identifiant les routes à des nœuds et les intersections à des arêtes (on suppose que toutes les fenêtres de temps de chevauchent). Ce graphe contient une clique maximum de taille 4 $\{L_1, L_2, L_3, L_4\}$ et peut être coloré avec 4 couleurs.

8.3 Recherche de cliques

Le modèle précédent ne pose que des contraintes de différence « locales » entre deux flux. Cependant, le graphe induit par ces contraintes peut contenir des sous-graphes complets, c'est-à-dire des *cliques*, qui peuvent être utilisées pour poser des contraintes globales « tous différents » et produire pendant la recherche des réductions de domaines plus importantes que celles des différences binaires. Bien que les problèmes de recherche de la clique *maximum*[5] ou de toutes les cliques *maximales*[6] d'un graphe soient bien connus pour être des problèmes NP-durs, des algorithmes incomplets peuvent être utilisés pour en fournir une solution approchée, en supposant que l'heuristique choisie sera assez efficace pour la structure particulière du graphe considéré.

Nous utilisons un algorithme glouton très simple (figure 6), similaire à ceux présentés dans [Bomze 99], pour construire un ensemble de cliques en essayant d'obtenir les plus grandes possibles. À partir de chaque nœud i (numéroté de 1 à n) du graphe, une clique est construite incrémentalement avec les nœuds de sa liste d'adjacence (adj(i)) comme candidats, en commençant par le nœud dont l'ensemble des voisins à la plus grande intersection avec les autres candidats. Cet algorithme a la propriété de ne produire que des cliques maximales, de telle manière qu'aucun sous-ensemble d'une clique générée ne peut

[5] C'est-à-dire trouver la sous-graphe complet de plus grande taille dans un graphe donné.
[6] Une clique *maximale* est une clique qui n'est strictement contenue dans aucune autre clique.

être renvoyé : on ne posera donc pas de contrainte globale inutile sur une clique $\mathcal{L}' \subset \mathcal{L}$ subsumée par une autre contrainte globale sur une clique qui la contient $\mathcal{L}'' \supset \mathcal{L}'$. Néanmoins, des cliques identiques peuvent être découvertes à partir de nœuds initiaux différents, et l'ensemble de cliques sur lesquelles sont posées les contraintes globales est filtré pour supprimer les doublons et pour sélectionner les cliques de cardinal strictement supérieur à 2 — les cliques de taille 2 sont déjà traitées par les contraintes de différence.

Algorithme 6 – Algorithme glouton de recherche de cliques

1: $cliques := \emptyset$
2: **for** $i = 1$ **to** n **do**
3: $\quad cl := \{i\}$
4: $\quad candidats := \text{adj}(i)$
5: $\quad$ **while** $candidats \neq \emptyset$ **do**
6: $\quad\quad j := \text{argmax}_{k \in candidats} |candidats \cap \text{adj}(k)|$
7: $\quad\quad cl := cl \cup \{j\}$
8: $\quad\quad candidats := candidats \cap \text{adj}(j)$
9: $\quad$ **end while**
10: $\quad cliques := cliques \cup \{cl\}$
11: **end for**

On peut estimer la complexité en pire cas de cet algorithme, qui a lieu quand le graphe est complet. Pour une telle instance, la liste des candidats diminue d'un nœud à chaque passage dans la boucle `while` ; en notant c le nombre de ces candidats, on obtient :

$$\underbrace{n}_{\text{boucle } \texttt{for} \text{ (lignes 2–11)}} \times \underbrace{\sum_{c=1}^{n-1}}_{\text{boucle } \texttt{while} \text{ (lignes 5–9)}} \underbrace{c}_{\texttt{argmax} \text{ (ligne 6)}} \times \underbrace{(c+n-1)}_{\text{intersection (ligne 6)}}$$

Ce qui donne un temps de calcul en $\mathcal{O}(n^4)$ pour un graphe « dense », i.e. le nombre d'arêtes est en $\mathcal{O}(n^2)$, ce qui est une bonne estimation de la structure de notre problème pour les valeurs faibles de n. En revanche, pour les plus grandes instances, les flux ajoutés sont de petites tailles, et leur fenêtre de temps est donc très courte ; il en résulte que ces flux intersectent peu d'autres flux dans le temps, et génèrent peu de nouvelles arêtes. Pour ces graphes moins denses, la ligne 8 de l'algorithme élimine beaucoup de candidats au fur et à mesure que la clique grandit, et la complexité de l'algorithme devient plus faible. La figure 8.3 présente les temps de calcul pour la recherche de cliques en fonction du nombre de nœuds, chaque point correspondant à une valeur de n_{min} (la taille minimale des flux considérés, voir section 8.2) comprise entre 20 et 1 : la courbe a bien une croissance en $\mathcal{O}(n^4)$ pour les petites instances, et moins rapide pour les grandes.

D'autre part, même si cette étape statique est efficace (quelques secondes pour des flux à plus de 3 avions), on peut restreindre le temps de calcul en se limitant aux premières cliques seulement : dans notre implémentation, les nœuds sont au préalable triés par degré

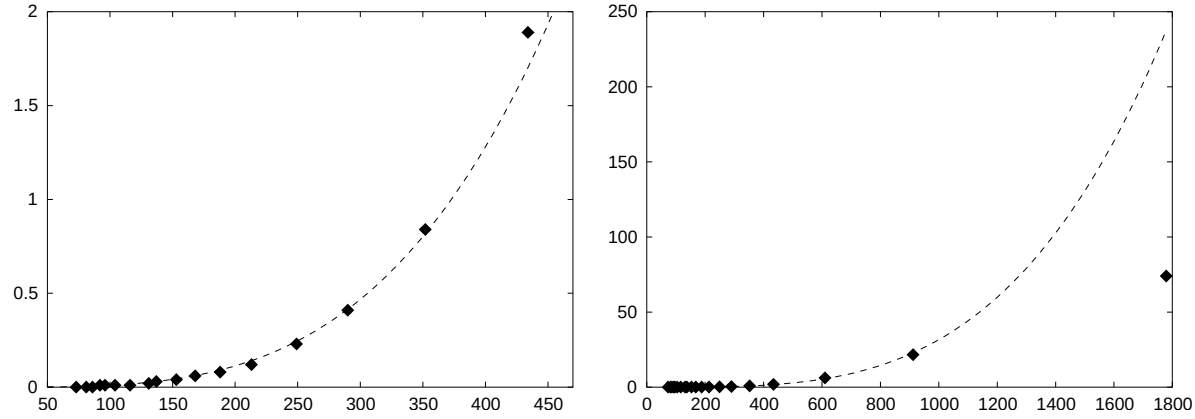


FIG. 8.3 – Temps de calcul (en secondes sur un P-III 700 MHz) de la recherche de cliques en fonction du nombre de nœuds. Complexité en $\mathcal{O}(n^4)$ (courbe en pointillés) excepté pour les plus grandes instances (à droite).

décroissant et sélectionnés dans cet ordre pour initier les cliques ; cette heuristique tend à générer d'abord les cliques les plus grosses, qui sont aussi les plus utiles pour accélérer la recherche dans le problème de coloriage. Par ailleurs, les cliques produites sont stockées dans une base de données pour chaque instance afin d'éviter de répéter ce calcul lors de l'expérimentation sur les buts de recherche.

Un pourcentage élevé (entre 50% et 90%) des nœuds de départ (ligne 3) génèrent des cliques valides (de taille supérieure à 3), notamment quand n croît, comme le montre la partie gauche de la figure 8.4 où figure le ratio entre le nombre de cliques et n en fonction de n, pour toutes les cliques découvertes (courbe supérieure) et pour toutes les cliques distinctes (courbe inférieure). On notera que le nombre de cliques redondantes est assez faible.

Pour chaque clique produite par l'algorithme glouton, on pose une contrainte globale « tous différents » qui utilise un algorithme de couplage dans les graphes biparties (*binary matching*) implémenté par une primitive de FaCiLe. Cet algorithme de filtrage global génère des réductions de domaine plus importantes que l'ensemble des $\frac{n(n-1)}{2}$ contraintes binaires de différence équivalent. Évidemment, les contraintes binaires entre variables impliquées dans une même clique ne sont pas posées pour éviter des redondances inutiles et coûteuses en temps de calcul. De plus, la taille de la plus grosse clique trouvée est utilisée comme borne inférieure pour la variable de coût. La courbe de droite de la figure 8.4 montre la taille de la plus grosse clique en fonction du nombre de nœuds : des cliques de taille significative sont découvertes, car cette borne inférieure est systématiquement proche du coût de la solution optimale pour les instances résolues (voir sections 8.5 et 8.6).

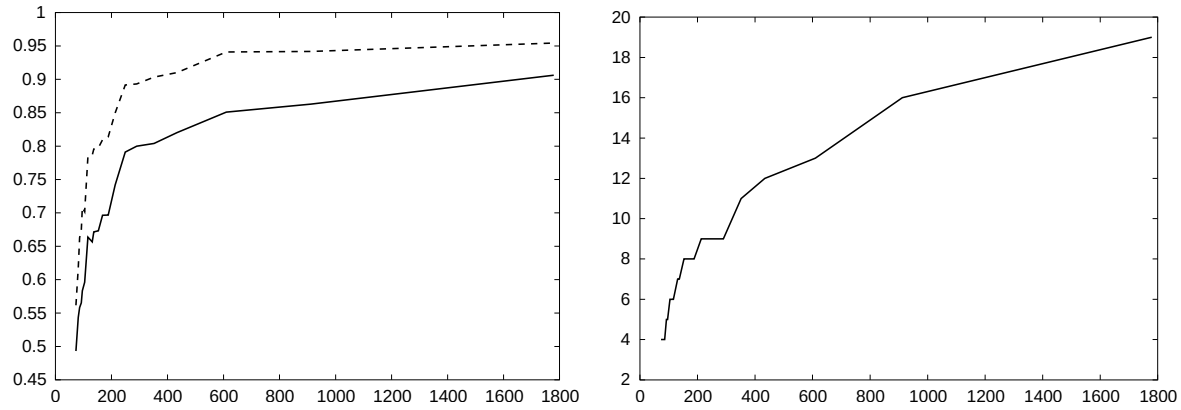


FIG. 8.4 – À gauche : ratios « *nombre de cliques* »/n et « *nombre de cliques distinctes* »/n en fonction du nombre de nœuds n. À droite : taille de la plus grande clique, en fonction de n.

8.4 Stratégie de recherche

La stratégie de recherche qui semble être la plus efficace de celles que nous avons expérimentées sur le problème de réseau de routes est guidée par les cliques découvertes durant la première étape statique de notre algorithme (cf. section précédente 8.3). L'idée principale est que les cliques de grande taille correspondent aux régions les plus contraintes du réseau de flux et donc que l'instanciation précoce de leurs variables peut produire des réductions de domaines efficaces (*first-fail principle*) en tirant partie des contraintes globales « tous différents ».

Les résultats présentés dans la section 8.5 sont obtenus avec une recherche classique en profondeur d'abord combinée à un Branch & Bound pour minimiser le nombre de niveaux de vols. Les cliques sont choisies par taille décroissante, puis toutes les variables d'une même clique sont sélectionnées avec une DVO standard par taille de domaine croissant. Cette stratégie semble assez robuste pour la plupart des instances du problème de réseau de routes.

Pendant la recherche, les symétries de permutation entre les n couleurs équivalentes (i.e. les niveaux de vols) que les flux peuvent prendre sont cassées dynamiquement. L'ensemble des couleurs déjà allouées est maintenu à chaque fois qu'un flux est instancié (par un point de choix ou par propagation), et lorsqu'on doit choisir la valeur d'instanciation d'un flux, s'il ne reste plus de couleur déjà utilisée dans son domaine, on sélectionne de manière déterministe la plus petite valeur possible. Ainsi, quand une nouvelle couleur doit être choisie, aucun point de choix n'est créé car toutes les couleurs non-utilisées sont équivalentes. Mais si l'intersection entre l'ensemble des couleurs déjà choisies et le domaine d'un flux n'est pas vide, la plus petite valeur commune est essayée en premier, si bien qu'une nouvelle couleur est allouée uniquement s'il n'y a aucun autre choix. Ce but d'instanciation garantit que toutes les parties de l'arbre de recherche qui mènent à des solutions symétriques

vis-à-vis des permutations sur les couleurs seront élaguées : pour un ordre d'instanciation des variables donné, on attribue aux flux la plus petite couleur possible. Cette élimination de symétries permet d'obtenir des preuves d'optimalité pour la plupart des des tailles de problème (cf. section suivante 8.5).

8.5 Résultats

Les résultats présentés dans cette section ont été obtenus à l'aide de toutes les techniques décrites précédemment, implémentées en OCaml avec FaCiLe :

- un algorithme glouton pour trouver des cliques distinctes (nombreuses et de grande taille) ;
- des contraintes globales « tous différents » posées sur toutes les cliques ;
- l'élimination dynamique des symétries sur les permutations parmi les couleurs ;
- une stratégie de recherche guidée par les cliques trouvées à la première étape.

Le programme a été exécuté avec la taille minimum des flux n_{min} variant de 20 à 1. Toutes les tailles ont été résolues jusqu'à la preuve d'optimalité, sauf pour le plus gros problème ($n_{min} = 1$, 1 779 flux, 165 859 intersections et un optimal situé entre 20 et 24 couleurs).

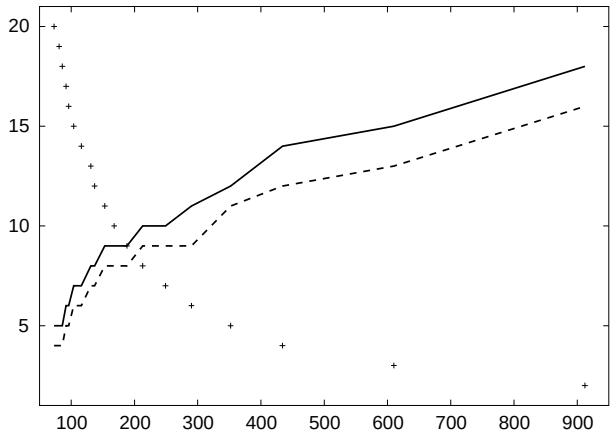


FIG. 8.5 – Nombre optimal de niveaux de vol (courbe supérieure) et taille de la plus grande clique trouvée (courbe inférieure) en fonction du nombre de flux. Les croix correspondent aux valeurs de n_{min} associées (nombre minimal d'avions par flux).

La figure 8.5 montre le nombre de niveaux de vol optimal en fonction du nombre de flux (courbe du haut) et les nombres minimaux d'avions par flux sont notés par des croix. Il ne faut pas plus de 12 niveaux de vols pour des flux à plus de 5 avions. Donc en considérant une séparation verticale de 1000 ft, les flux considérés pourraient évoluer dans l'espace aérien supérieur qui s'étend approximativement de 20000 ft and 40000 ft. Sur le même graphique, la courbe du bas (en pointillés) indique la taille de la plus grosse clique trouvée : elle fournit une borne inférieure proche de l'optimum.

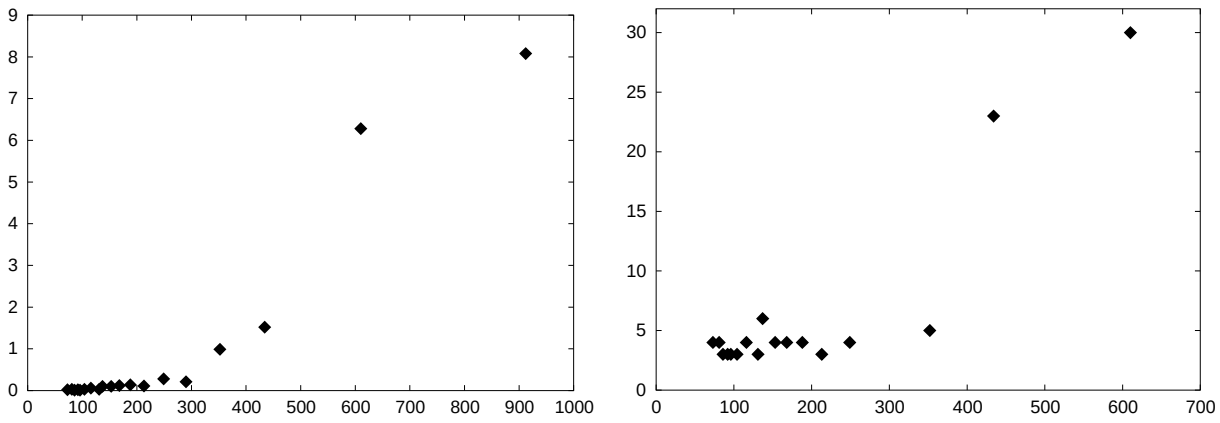


FIG. 8.6 – Temps CPU en secondes (à gauche) et nombre de backtracks (à droite) pour la preuve d'optimalité en fonction du nombre de flux.

Les temps CPU pour prouver l'optimalité (à gauche) et le nombre de backtracks pour y parvenir (à droite) sont présentés figure 8.6. Pour $n_{min} = 2$ (912 flux, 68759 intersections), la solution optimale et la preuve d'optimalité n'ont pu être obtenues qu'en contraignant plus avant la variable de coût à sa valeur optimale (18 — avec une plus grande clique de taille 16) ; cette instance n'apparaît donc pas sur le graphique des nombres de backtracks (252 backtracks en l'occurrence). D'autre part, la preuve d'optimalité pour $n_{min} = 6$ n'a pu être atteinte qu'en utilisant une autre stratégie, l'heuristique de BRÉLAZ, au sein de l'étiquetage par cliques.

8.6 Autres approches

Les deux sections suivantes compare notre algorithme avec les travaux existants sur la conception de réseaux de routes pour l'ATFM d'une part, et sur le coloriage de graphe d'autre part. La troisième présente les résultats obtenus sur des benchmarks standards de coloriage de graphe, suivis des performances d'un algorithme efficace mais approché de coloriage de graphe (DSATUR) sur notre problème de réseau de routes.

8.6.1 Conception de réseaux de routes pour l'ATFM

La nécessité d'accroître la capacité de l'espace aérien européen a débouché sur plusieurs approches en ce qui concerne la conception d'un nouveau réseau de routes.

Le projet TOSCA [Maugis 98] mené par Eurocontrol considère la possibilité d'un réseau de routes directes mais estime que la complexité des conflits engendrés serait impossible à gérer par le système ATC actuel, car le modèle n'envisage pas la séparation verticale des flux. Cette étude s'intéresse davantage aux « outils » ATFM ainsi qu'aux équipements liés à l'ATC (au sol et embarqués) et aux procédures de vol pour gagner en flexibilité et en

capacité dans les limites du réseau actuel. Pourtant, de telles mesures paraissent avoir peu de potentiel en terme d'accroissement de capacité parce qu'elles restent trop assujetties au système « sur-contraint » existant.

Une étude plus originale présentée dans [Mehadhebi 00] tente de minimiser simultanément la longueur des routes et la « complexité » du trafic dans les régions où le maillage du réseau est dense. Un diagramme de VORONOÏ est construit à partir des nœuds du réseau et un indicateur y modélise la charge de travail liée au contrôle. Un voisinage est ensuite défini sur cette structure pour optimiser le réseau par recherche locale. Notre approche est très différente car le problème de la gestion des conflits par l'ATC est évacué en assurant une séparation verticale des flux. Cependant, notre modèle de routes directes est très éloigné des procédures ATFM et ATC actuelles et son intégration opérationnelle nécessiterait un grand nombre de raffinements (cf. section 8.7).

Un concept très différent mais concerné en partie par les mêmes préoccupations est celui du « *free flight* » qui permet aux avions de choisir librement leur route (*a priori* directes, cf. section 8.1) et de résoudre dynamiquement les conflits à l'aide de divers algorithmes. [Alliot 98] présente un algorithme de *free flight* distribué qui utilise des jetons de priorité pour ordonner la résolution des conflits et les résout l'un après l'autre à l'aide d'un A^* cherchant une solution avec des manœuvres simples. Cet algorithme paraît très efficace pour gérer le trafic dans l'espace aérien supérieur, mais cette approche est, encore une fois, très différente de la nôtre car elle est ciblée sur l'ATC automatique distribué, ce qui est également éloigné du système opérationnel actuel.

Enfin, le modèle de [Letrouit 98] est similaire au nôtre mais utilise des techniques *ad hoc* de RO. Un algorithme recherche des sous-réseaux d'un réseau de routes directes dont le graphe associé est un graphe de permutation. Ces graphes particuliers admettent un algorithme polynomial pour le calcul de la clique maximum. Ces cliques sont ensuite utilisées pour guider un algorithme approché de coloriage, mais avec des performances assez faibles. La méthode proposée est intégrée dans un outil graphique interactif destiné à la conception d'un réseau de routes européen.

8.6.2 Coloriage de graphe

Notre modèle très simple se réduit à un problème de coloriage de graphe pur qui est un problème très classique, bien étudié et toujours un thème de recherche actif. Notre algorithme peut donc être comparé avec les principales approches existantes.

Comme le coloriage de graphe est un problème NP-dur, la plupart d'entre elles sont des méthodes heuristiques tels que les algorithmes gloutons (DSATUR [Brélaz 79], RLF [Leighton 79]) ou la recherche locale (recuit simulé [Morgenstern 86], recherche taboue [Hertz 87], algorithme génétique [Fleurent 94]), enrichies éventuellement de recherche énumérative (backtrack limité). La plupart des techniques génèrent le coloriage soit en attribuant séquentiellement une couleur à chaque nœud, soit en partitionnant les nœuds en stables[7] (*independent sets* en anglais) formant des classes de couleurs. Certaines d'entre

[7]Un stable est un ensemble de nœuds tel qu'aucune arête n'existe entre deux de ses éléments : soit

elles recherchent également une clique (en général la plus grande possible) pour fournir une borne inférieure et/ou initier le coloriage. Enfin, on peut voir un lointain rapport entre notre stratégie de recherche guidée par les cliques et le travail de [Culberson 92] qui mentionne l'utilisation de cliques comme second critère pour ordonner les variables ex æquo dans l'algorithme DSATUR, mais sans résultat probant. Toutefois, on peut remarquer que chercher un stable maximum ou chercher une clique maximum sont des problèmes NP-durs au même titre que le problème de coloriage, si bien que la plupart des techniques utilisées pour ces tâches préliminaires ne sont pas complètes, voire gloutonnes.

Ces algorithmes et leurs versions raffinées ou hybrides (e.g. [Dorne 98] présente une « recherche taboue génétique » initialisée avec DSATUR) sont efficaces pour trouver de « bonnes » solutions pour les graphes aléatoires de grande taille telles que les instances proposées par le challenge DIMACS[8], pour lesquelles une recherche énumérative peut avoir un coût prohibitif. Pourtant, des algorithmes complets reposant sur l'énumération et le backtracking ont également été conçus pour fournir des preuves d'optimalité ou d'échec (pour obtenir des bornes inférieures par exemple), comme les travaux précurseurs de [Christofides 71] et [Brown 72] ainsi que leurs successeurs plus sophistiqués (e.g. [Brélaz 79, Peemöller 83]). Plus récemment, [Coudert 97] a proposé un algorithme complet qui recherche d'abord une clique *maximum* puis utilise un coloriage séquentiel complet. L'auteur remarque que toutes les instances issues d'application réelles qu'il a pu trouver sont *1-parfaites*, c'est-à-dire que leur nombre chromatique (la taille de leur coloriage optimal) est égal à la taille de leur clique maximum. Ainsi, la recherche s'arrête dès qu'un coloriage de même cardinalité a été découvert. Bien que le problème d'allocation des niveaux de vol semble exhiber la même propriété[9], notre technique profite plutôt de la découvertes de cliques sous-optimales en grand nombre. Sur les graphes d'applications réelles, [Coudert 97] présente d'excellents résultats. Néanmoins, les graphes qui ne sont pas 1-parfaits dégradent considérablement les performances de l'algorithme présenté, et on rappelle également que le coût du calcul d'une clique maximum est exponentiel (on doit donc résoudre un *autre* problème NP-difficile avant de calculer le coloriage).

handicapent considérablement la recherche, et on rappelle que trouver une clique maximum a une complexité exponentielle en pire des cas.

La PPC est aussi une technique de choix pour colorier des graphes de taille « raisonnable », directement modélisés en associant les nœuds à des variables et les arêtes à des contraintes de différence. Les CSP sont d'ailleurs souvent considérés comme des graphes, éventuellement munis d'hyper-arêtes pour représenter les contraintes impliquant plus de deux variables, ce qui permet d'utiliser la théorie des graphes pour raisonner sur des problèmes en contraintes. En particulier, les stratégies classiques d'instanciation de variables les plus robustes sont inspirées par des heuristiques développées pour des problèmes de coloriage de graphe, e.g. voir [Smith 99a] à propos de l'heuristique de Brélaz [Brélaz 79].

$G = (V, E)$ un graphe, $V' \subseteq V$ est un stable si et seulement si $\forall u, v \in V',\ (u, v) \notin E$. Les nœuds d'un stable peuvent donc être coloriés de la même couleur.

[8] `ftp ://dimacs.rutgers.edu/pub/challenge`

[9] Un algorithme complet a été utilisé pour vérifier cette caractéristique et on peut ajouter son résultat à la base de données des cliques découvertes par l'algorithme glouton.

TAB. 8.1 – Benchmarks du matériel

r100.5	r200.5	r300.5	r400.5	r500.5
0.01	0.23	2.06	12.78	47.78

La recherche est en général contrôlée par un Branch & Bound pour obtenir des preuves d'optimalité ou d'échec. [Van Hentenryck 90] présente cette technique avec un ordre d'instanciation similaire à celui de BRÉLAZ et le même type d'élimination de symétries sur les couleurs que celui que nous utilisons. Cependant, cette approche ne profite pas de la présence de cliques ni de contraintes globales pour accélérer la recherche. À la différence des autre techniques mentionnées, la PPC offre un niveau d'abstraction élevé qui permet de raffiner le modèle utilisé en ajoutant d'autres types de contraintes et d'expérimenter facilement diverses heuristiques grâce à la souplesse des buts de recherche, autant de caractéristiques très intéressantes pour les perspectives de notre application (cf. section 8.7). En revanche, une de ses limitations importantes est le coût en temps de calcul de la recherche par backtrack standard sur les instances de grande taille. Plusieurs tentatives ont été expérimentées pour résoudre ce problème telles que le *Backtracking Dynamique Incomplet* (IDB) proposé par [Prestwich 01] avec de bons résultats sur les problèmes du challenge DIMACS, mais en abandonnant les preuves d'optimalité et d'échec.

8.6.3 Benchmarks

Benchmarks de coloriage de graphe

Nous avons comparé les performances de notre algorithme avec divers problèmes de benchmark tirées de [Trick 94] (qui reprend des instances du challenge DIMACS). Les résultats sont rassemblés dans les tableaux 8.2 et 8.3 où figurent le nom du problème, le nombre chromatique (si celui-ci est connu), le nombre de backtracks nécessaires pour obtenir la meilleure solution trouvée et le temps CPU correspondant, le nombre de backtracks nécessaires pour obtenir la preuve d'optimalité et le temps CPU correspondant, et enfin la taille de la plus grande clique découverte par notre algorithme. Les temps de recherche (qui exclut la lecture des données et la phase statique de la recherche gloutonne de cliques — réutilisée durant les diverses tentatives) ont été obtenus sur un P-III 700 MHz. Les performances de ce matériel sont notées avec le benchmark « machine » standard fournit par DIMACS qui exécute le programme `dfmax`[10] écrit par David JOHNSON et David APPLEGATE (une recherche complète de clique maximum) et présenté à titre indicatif dans le tableau 8.1.

Contrairement à ce qui a été présenté pour le problème de réseau de routes, diverses heuristiques, y compris celle de BRÉLAZ et `dom/deg` (cf. [Bessière 96]), souvent combinées avec la stratégie guidée par les cliques (décrite dans la section 8.4), ont été essayées, car

[10] `www.cs.sunysb.edu/~algorith/implement/dimacs/distrib/color/graph`

les performances étaient mauvaises sur certaines instances.

Le tableau 8.2 résume les résultats obtenus sur des instances d'applications réelles, soit industrielles (allocation de registres, emploi du temps) soit académiques (graphes issus du texte d'œuvres littéraires, graphes sur des cartes géographiques...). Les preuves d'optimalité sont rapidement obtenues hormis sur les problèmes des « reines[11] » pour lesquelles seules des bornes inférieures ont été trouvées pour les instances de taille supérieure à 9. Toutefois, les résultats rapportés par COUDERT dans [Coudert 97] et par KIROVSKI et POTKONJAK dans [Kirovski 98] suggèrent que ces problèmes sont très difficiles ; d'ailleurs, COUDERT ne les considère pas comme des graphes applicatifs mais comme des graphes aléatoires (ils n'ont pas non plus la propriété d'être 1-parfaits), et aucun résultat au-dessus la taille 9 n'est mentionné. Pour cette dernière instance, plus de 561.10^6 backtracks et 5 heures de calculs lui sont nécessaires, tandis que les problèmes réels ne prennent que quelques secondes.

Ces résultats ont été obtenus avec l'algorithme complet, mais nous avons également essayé d'observer séparément les contributions des contraintes globales « tous différents » et de l'étiquetage guidé par les cliques, qui sont supposés être complémentaires car leur combinaison devrait permettre des réductions de l'espace de recherche plus précoces en sélectionnant d'abord des variables impliquées dans les contraintes globales les plus importantes. De manière inattendue, pour quelques instances, la preuve d'optimalité n'a pu être atteinte qu'avec l'heuristique guidée par les cliques en l'absence de contraintes globales, ce qui montre l'intérêt de découvrir de grandes cliques.

Mais notre algorithme est moins efficace sur les graphes aléatoires qui ne présentent que des cliques de faible cardinalité (ou difficile à découvrir). La taille du problème est également un obstacle quand elle devient trop importante : le recherche est trop coûteuse en temps car le backtracking standard ne supporte pas bien la mise à l'échelle (surtout sur les problèmes peu structurés).

Performances de DSATUR sur le problème de réseau de routes

Les performances de DSATUR, un algorithme glouton classique de coloriage, sont présentées dans le tableau 8.4. Il s'agit d'une version implémentée par Joseph CULBERSON[12]. Pour chaque taille minimale de flux, le nombre de variables, de contraintes, la taille du meilleur coloriage trouvé par DSATUR et par notre algorithme sont présentés. Ces résultats sont obtenus en choisissant la meilleure heuristique parmi celles disponibles pour DSATUR. Tous les temps d'exécution sont inférieurs à 1 s sur le même système que celui utilisé pour les benchmarks précédents, mais DSATUR ne peut prouver l'optimalité de la solution et génère des coloriages de plus grande taille que ceux de notre algorithme pour des flux minimaux entre 2 et 5 avions (en gras dans le tableau 8.4). Cependant, cette implémentation de DSATUR n'est pas optimisée aussi finement que notre algorithme.

[11] Ces graphes correspondent à un échiquier de taille $n \times n$ sur lequel n ensembles de n reines — donc une reine par case — doivent être placées de telle sorte qu'aucune paire de reines appartenant au même ensemble ne soit en conflit.

[12] `www.cs.ualberta.ca/~joe/Coloring/Colorsrc`

TAB. 8.2 – Coloriage de problèmes applicatifs.

nom	χ	meilleur	bt (sol)	cpu (sol)	bt (preuve)	cpu (preuve)	clique
fpsol2.i.1	65	**65**	0	0.33	3	0.51	64
fpsol2.i.2	30	**30**	0	0.77	3	1.26	29
fpsol2.i.3	30	**30**	0	0.23	3	0.36	29
inithx.i.1	54	**54**	0	0.74	3	1.11	53
inithx.i.2	31	**31**	0	0.58	3	0.89	30
inithx.i.3	31	**31**	0	0.60	3	0.90	30
mulsol.i.1	49	**49**	0	0.08	3	0.12	48
mulsol.i.2	31	**31**	0	0.09	3	0.12	30
mulsol.i.3	31	**31**	0	0.09	3	0.12	30
zeroin.i.1	49	**49**	0	0.08	3	0.13	48
zeroin.i.2	30	**30**	0	0.06	3	0.11	29
zeroin.i.3	30	**30**	0	0.07	3	0.12	29
school1	14	**14**	27	13.23	30	13.36	13
school1_nsh	14	**14**	272	8.91	275	9.04	13
games120	9	**9**	0	0.02	3	0.02	8
anna	11	**11**	0	0.05	3	0.07	10
david	11	**11**	0	0.02	3	0.03	10
homer	13	**13**	0	0.15	3	0.16	12
huck	11	**11**	0	0.01	3	0.01	10
jean	10	**10**	0	0.00	3	0.00	9
miles250	8	**8**	0	0.03	3	0.03	7
miles500	20	**20**	0	0.03	3	0.04	19
miles750	31	**31**	0	0.07	3	0.09	30
miles1000	42	**42**	0	0.09	3	0.12	41
miles1500	73	**73**	0	0.17	3	0.17	72
queen5_5	5	**5**	0	0.00	3	0.00	4
queen6_6	7	**7**	89	0.08	115	0.10	5
queen7_7	7	**7**	122	0.12	125	0.12	6
queen8_8	9	**9**	19088	17	97964	89	7
queen9_9	9	10	77218	453	∞	∞	8
queen10_10	*LB : 10*	11	12494627	10741	∞	∞	10
queen11_11	*LB : 11*	13	1855	17	∞	∞	10
queen12_12	*LB : 12*	14	172119	187	∞	∞	11
queen13_13	*LB : 13*	17	20	0.67	∞	∞	12
queen14_14	*LB : 14*	17	4597	4.91	∞	∞	13
queen15_15	*LB : 15*	18	5094	84	∞	∞	14

TAB. 8.3 – Coloriage de problèmes aléatoires.

nom	χ	meilleur	bt (sol)	cpu (sol)	bt (preuve)	cpu (preuve)	clique
le450_5a	5	**5***	225	4.18	228	4.24	4
le450_5b	5	**5***	3564	51	3567	51	4
le450_5c	5	**5***	10	0.79	13	0.89	4
le450_5d	5	**5***	7	0.78	10	0.86	4
le450_15a	15	**15**	3905043	37976	∞	∞	14
le450_15b	15	**15**	26509	240	26512	240	14
le450_15c	15	23	4	5.96	∞	∞	14
le450_15d	15	23	2	6.04	∞	∞	14
le450_25a	25	**25**	0	0.97	3	1.09	24
le450_25b	25	**25**	0	0.95	3	1.05	24
le450_25c	25	28	0	2.00	∞	∞	24
le450_25d	25	28	2	0.96	∞	∞	24
myciel3	4	**4**	0	0.0	5	0.01	2
myciel4	5	**5**	0	0.0	50	0.06	2
myciel5	6	**6**	0	0.02	4417	11	2
myciel6	7	**7**	0	0.06	∞	∞	2
myciel7	8	**8**	0	0.24	∞	∞	2
flat1000_76_0	76	113	33	28	∞	∞	13

TAB. 8.4 – Performance de DSATUR sur le problème de réseau de routes.

n_{min}	nb var	nb arêtes	DSATUR	Cliques+PPC
10	168	2151	9	9
9	188	2720	9	9
8	213	3562	10	10
7	249	5156	10	10
6	290	7274	11	11
5	352	10581	13	**12**
4	434	16106	15	**14**
3	610	33059	16	**15**
2	912	68759	21	**18**
1	1779	165859	24	24

8.7 Conclusion et perspectives

Un modèle très simple pour gérer le trafic aérien dans un réseau de routes directes a été présenté. Ce modèle, comme celui de [Letrouit 98], évite statiquement les conflits et les contraintes de l'ATC en séparant verticalement les flux d'avions qui se croisent. Le problème d'optimisation qui en résulte est l'allocation de niveaux de vols aux flux (ensemble de vols partageant même origine et destination), et sa taille peut être adaptée en restreignant le nombre d'avions minimal dans les flux considérés. Ce problème d'allocation est équivalent à un problème de minimisation de coloriage de graphe et nous présentons une combinaison de techniques pour le résoudre.

Tout d'abord, un algorithme glouton est utilisé pour trouver des cliques *maximales* (et non *maximum*, voir section 8.3) ; cet étape statique s'avère efficace pour la structure de notre problème : un grand nombre de cliques sont découvertes et la taille de la plus grande est une bonne borne inférieure du nombre de niveaux de vol minimal. Ces cliques sont ensuite utilisées pour poser des contraintes globales (« tous différents ») dans un programme en contraintes. Enfin, une stratégie de sélection des variables guidée par l'ensemble des cliques découvertes lors de la première étape est utilisée au sein d'un Branch & Bound pour rechercher un coloriage optimal. Pendant cette recherche, les symétries de permutation entre niveaux de vol (ou couleurs) équivalents sont éliminées dynamiquement en ne posant pas de point de choix lorsqu'un flux doit être instancié à un niveau encore inutilisé.

Avec une implémentation en FaCiLe de ces techniques, nous parvenons à résoudre le problème optimalement et avec des temps de calculs de quelques secondes pour la plupart des tailles sur un jour de trafic typique, exceptée la plus grande. Les solutions optimales pour des flux de taille « opérationnelle » suggèrent que ce trafic pourrait évoluer selon ce schéma dans l'espace aérien supérieur, séparé des flux les plus petits supposés voler à des niveaux inférieurs et contrôlés de manière spécifique par l'ATC. Nous avons également montré que cette technique de coloriage est robuste sur les problèmes d'applications réelles pour lesquels de grandes cliques sont découvertes. Notre algorithme trouve également de meilleures solutions que DSATUR sur plusieurs instances significatives du problème de réseau de routes.

Cependant, cette modélisation du trafic est très simplifiée et son ambition est essentiellement une étude de faisabilité du concept de réseau de routes directes. Pour la rendre plus réaliste, beaucoup de contraintes additionnelles devraient être prises en compte. Par exemple, les performances des avions, qui ont un niveau de vol préféré pour minimiser leur coût d'exploitation — ce qui dépend essentiellement du type d'appareil et de la longueur de la route — pourrait être modélisées en restreignant les domaines des flux ou en modifiant la fonction de coût. Les manœuvres des avions dans le plan vertical à l'intérieur des zones terminales devraient également être prises en compte pour éviter des conflits avec les flux évoluant à un niveau inférieur, par exemple en découpant chaque route en plusieurs tronçons. Les paramètres qui influencent les procédures de vols actuelles (utilisation des balises, évitement des zones militaires, structure des secteurs de contrôle, parité du niveau de vol suivant le cap etc.) devraient également être considérés pour permettre l'intégration d'un nouveau réseau. En outre, les trafics aérien français et européen sont fortement couplés,

si bien que l'implémentation d'un réseau de routes directes affecterait au moins les pays limitrophes : le problème devrait donc être résolu globalement mais sa grande taille pourrait rendre les algorithmes systématiques inutilisables (trop longs) ; d'autres techniques de PPC non complètes ou de recherche locale pourraient alors se révéler plus efficaces.

Nous envisageons également de raffiner cette application et son algorithme par des techniques de recherche de cliques plus efficaces, l'élaboration d'un critère dynamique de sélection des cliques pendant la recherche ou l'utilisation de ces cliques au sein de stratégies standard. Le modèle pourrait aussi être raffiné en permettant de découper les flux dont les fenêtres de temps ne sont pas continues pour changer leur niveau de vol durant la journée.

Conclusion et perspectives

Algorithmes et technologies d'optimisation

Recherche systématique et recherche locale La résolution d'un problème d'optimisation combinatoire sous forme de CSP consiste d'abord à le modéliser en termes de variables, domaines et contraintes, puis à appliquer à ce modèle un algorithme de recherche. Les deux principaux paradigmes pour la résolution des CSP sur les domaines finis, décrits au chapitre 3, sont d'une part la recherche *systématique*, qui alterne la construction énumérative d'une instanciation partielle et l'inférence de contraintes induites par le CSP, et d'autre part la *Recherche Locale* (ou *méta-heuristique*) qui parcourt stochastiquement l'espace des instanciations totales en suivant des gradients locaux selon une heuristique pour que le nombre de contraintes violées ou le coût de la solution diminuent. La première approche est complète, ce qui permet d'obtenir des preuves d'optimalité ou d'inconsistance, mais de complexité exponentielle, ce qui limite son efficacité sur des problèmes de grande taille. Inversement, la seconde est incomplète mais peut permettre d'obtenir des solutions « approchées » (sous-optimales ou inconsistantes) sur des CSP de grande taille ou sur-contraints. Alors que les techniques de consistance locale de la recherche systématique tentent de « raisonner » *a priori* sur les contraintes du problème pour réduire au maximum l'espace de recherche destiné à être exploré exhaustivement (pour garantir la complétude), la RL exploite des informations *locales* calculées dans le voisinage d'un point de l'espace des « solutions » (éventuellement inconsistantes). Mais la structure arborescente de la recherche effectuée par les algorithmes systématiques impliquent de considérer les changements de valeur des variables dans un ordre fixé ; les erreurs des heuristiques d'instanciation sont ainsi pénalisantes si elles concernent les premières variables choisies, et c'est justement au début de la recherche que les heuristiques sont les moins bien « informées » (dans le cas de stratégies dynamiques). Au contraire, la RL peut se déplacer librement vers les parties de l'espace de recherche qui semblent prometteuses mais en sacrifiant la complétude de la recherche et en ne garantissant aucune propriété de consistance sur les instanciations considérées ; la recherche peut alors être bloquée dans un optimum local si le voisinage ne contient pas de meilleure solution que l'instanciation courante et une stratégie doit être utilisée pour s'en échapper.

Modélisation et Programmation Par Contraintes Les différences technologiques qui distinguent l'utilisation d'un algorithme de recherche systématique et celle d'un algo-

rithme de recherche locale pour résoudre un CSP ne concernent pas que la « mécanique » de la recherche mais également la modélisation du problème et les systèmes disponibles pour implémenter la recherche. En effet, la RL consiste essentiellement en une collection d'algorithmes fondés sur des métaphores différentes (recuit simulé, algorithmes génétiques, recherche taboue etc.) et qui, grossièrement, manipulent le même type d'objets et partagent une certaine dynamique. Mais les tentatives d'unification formelle des diverses méta-heuristiques ne sont que très récentes, dotées de peu de propriétés et seules de rares systèmes (e.g. Localizer [Michel 97]) sont disponibles pour faciliter l'écriture d'algorithmes et la modélisation de problèmes. En revanche, une partie des techniques de la recherche systématique a été efficacement intégrée dans le cadre de la *Programmation Par Contraintes*, un formalisme générique et cohérent pour spécifier un CSP et une stratégie de recherche et pour le résoudre. La PPC sépare clairement ces différentes étapes et permet de les exprimer de manière déclarative, ce qui lui confère un statut de langage de modélisation permettant de réaliser rapidement des prototypes et de les modifier facilement. La PPC provient de l'extension de la Programmation Logique à de nouvelles structures de domaines et jouit de sémantiques logique et opérationnelle rigoureuses. Divers systèmes, libres ou commerciaux, implémentent la PPC sur les domaines finis, mais aussi sur d'autres domaines comme les réels ou les ensembles, et certains d'entre eux ont permis de réaliser des applications industrielles avec succès. Les systèmes de PPC sont bâtis autour d'une architecture à deux niveaux séparant le solveur de contraintes lui-même du langage qui le contrôle et permet de spécifier CSP et stratégie de recherche. Ce design permet d'utiliser les systèmes de PPC pour intégrer des algorithmes de résolution de contraintes très divers, comme des techniques de Recherche Opérationnelle ou d'Intelligence Artificielle, pour améliorer les performances dans des domaines particuliers. Les langages de contraintes des systèmes de PPC se sont ainsi enrichis considérablement et permettent de modéliser des CSP avec une grande expressivité dans des domaines de plus en plus variés. Les systèmes les plus ouverts donnent accès à la partie « solveur » et permettent à l'utilisateur d'écrire ses propres contraintes et d'incorporer ainsi à la PPC des techniques spécifiques issues de l'expertise sur un problème particulier. Cette « extensibilité » est un atout majeur pour résoudre des problèmes industriels complexes.

Hybridation Recherche systématique et recherche locale sont donc des paradigmes fondés sur des technologies très différentes et qui possèdent des vertus et des faiblesses parfois complémentaires. Pour profiter des avantages des deux approches sur des problèmes difficiles et de grande taille, de nombreux algorithmes hybrides, dont les principaux sont décrits section 3.3, ont été proposés, notamment dans les domaines du *scheduling* et du *routing* (problèmes de tournées de véhicules). Certains permettent, au sein d'une recherche énumérative, de rendre plus souple le choix de la variable remise en cause lors d'un backtrack pour pouvoir suivre des gradients locaux, ou de modifier le parcours classique en profondeur d'abord pour mieux répartir l'exploration de l'arbre de recherche, éventuellement au prix de la complétude. D'autres utilisent l'une des deux approches pour résoudre des sous-problèmes intervenant au sein d'une recherche menée par l'autre : par exemple l'exploration

de voisinages avec la PPC ou la détection d'inconsistances par recherche locale dans un algorithme de *backtrack*.

Mais ces méthodes sont souvent dédiées à des problèmes particuliers et n'offrent pas de cadre assez abstrait pour implémenter de manière concise, efficace et générique des algorithmes hybrides. La PPC semble une plate-forme assez fédératrice et proche d'un langage de modélisation pour servir de support à l'intégration de paradigmes multiples. Certains systèmes offrent déjà des mécanismes pour combiner PPC et RL, par exemple en utilisant des techniques de « réparation » (cf. section 3.2.2) comme ECL^{i}PS^e ou en fournissant des abstractions pour contrôler l'exploration de l'arbre de recherche comme Mozart et ILOG Solver. Mais les difficultés conceptuelles restent nombreuses pour unifier les deux approches tout en gardant un langage de modélisation assez abstrait et efficace : manipuler à la fois instanciations partielles et totales, utiliser efficacement un même modèle avec les deux paradigmes, spécifier des stratégies complexes d'exploration de l'arbre de recherche, etc.

Langages et librairies d'optimisation La plupart des systèmes de PPC utilise le langage d'origine, Prolog, comme langage hôte, mais le traitement des données, le codage d'algorithmes de filtrage efficace pour des contraintes globales, les interfaces avec d'autres applications etc. sont des tâches délicates dans le cadre de la programmation logique. Le manque de typage et de vérifications à la compilation dans les systèmes Prolog classiques dissuadent également de les utiliser dans le développement de gros projets pour des raisons de sécurité et de robustesse. Quelques langages de PPC (e.g. Claire, Mozart, Mercury) se démarquent pour ces raisons de Prolog en ajoutant des constructions empruntées à d'autres paradigmes de langages de programmation (fonctionnelle, impérative, objet), des vérifications de types etc. mais le développement d'un nouveau langage est très lourd et les fondations théoriques de tels assemblages restent à établir. L'autre approche est d'écrire une librairie pour un langage « généraliste » (e.g. C++ pour ILOG Solver ou Lisp pour PECOS), en fournissant des constructions et des primitives pour spécifier variables, contraintes et stratégie de recherche, et en implémentant des mécanismes pour la gestion de la recherche non-déterministe.

Cependant, les systèmes de PPC qui adoptent cette dernière approche utilisent des langages peu fiables (e.g. typage et gestion mémoire de C++) et qui manquent d'abstractions (modules, ordre supérieur) ou d'efficacité en temps d'exécution (e.g. Lisp). Nous avons donc développé une librairie de PPC pour Objective Caml [Leroy 00], un langage applicatif de programmation fonctionnelle fortement typé, doté de puissantes abstractions et compilé efficacement. Les constructions fonctionnelles, les clôtures et la richesse des structures de données confèrent à ce langage des caractéristiques proches de celles d'un langage de modélisation, et il est ainsi particulièrement bien adapté à la spécification de problèmes et de procédures de recherche complexes construite à l'aide de primitives simples. Notre librairie, FaCiLe (Functional Constraint Library), a été conçue de manière à être simple, petite et « extensible » : l'utilisateur peut l'enrichir avec de nouvelles contraintes et des buts de recherche arbitraires pour incorporer des techniques dédiées à un problème particulier ;

les primitives (définies par le langage et par l'utilisateur) peuvent être ensuite facilement combinées pour obtenir des composants complexes. FaCiLe partage ainsi les préoccupations d'un projet similaire, CHOCO [Laburthe 00], conçu indépendemment et utilisant Claire [Caseau 96] comme langage hôte.

Mais notre librairie donne également accès aux mécanismes de bas niveau du solveur (évènements, réveil des contraintes, références « backtrackables ») grâce à des abstractions destinées à l'intégration de techniques issues d'autre paradigmes que la PPC. Ainsi, FaCiLe implémente le concept d'*invariants* introduits par Localizer (un langage de modélisation d'algorithmes de recherche locale) pour maintenir automatiquement des structures de données liées par des dépendances fonctionnelles. Toutefois, elle n'offre pas encore la possibilité d'utiliser des techniques de RL sur les variables logiques d'un modèle, ni d'abstraction pour contrôler l'exploration de l'arbre de recherche avec des stratégies hybrides.

Simultanément à l'élaboration de FaCiLe, des langages de modélisation plus ambitieux, destinés à séparer la description du problème du paradigme de recherche utilisé pour le résoudre, ont été développés. OPL (Optimization Programming Langage), un langage commercial conçu par [Van Hentenryck 99], permet ainsi de spécifier un CSP et de le résoudre soit avec des techniques de PPC, soit avec des algorithmes de programmation mathématiques, en utilisant des solveurs externes ; notons qu'une librairie pour C++, OPL++, équivalente à OPL hormis la syntaxe, a été développée en estimant, comme dans le cas de FaCiLe, que l'expressivité du langage hôte est suffisante pour se passer d'un langage de modélisation *ad hoc*. Une autre démarche encore plus originale est celle de SaLSA (Specification Language for Search Algorithms) [Laburthe 98a] qui propose de spécifier des algorithmes de recherche hybrides à l'aide d'un langage qui sépare la description de « points de choix », concept commun aux deux paradigmes (recherche énumérative et RL) dans ce langage, de celle du contrôle de l'exploration ; cependant, à notre connaissance, seule une partie assez restreinte de ce langage a été implémentée.

Fonctionnalités et implémentation de FaCiLe FaCiLe est une librairie *Open Source* de PPC sur les domaines finis essentiellement destinée à la recherche et l'enseignement, mais avec des performances souvent meilleures que celles des systèmes Prolog et une robustesse comparable à celle d'ILOG Solver, le système commercial leader. Elle a fait l'objet de deux publications [Barnier 01a, Barnier 01b] qui la présentent succinctement.

L'ambition de FaCiLe est plus de fournir des « briques » de base simples et cohérentes pour pouvoir l'étendre et la comprendre que de tendre vers l'exhaustivité des domaines de contraintes pour concurrencer les systèmes commerciaux. Toutefois, elle propose déjà des contraintes globales très efficaces sur les domaines finis (diséquation et cardinalité généralisées, tri) et des contraintes sur les domaines d'ensembles finis. Son développement futur prévoit de nouvelles contraintes sur les domaines finis dédiées à des grandes classes de problèmes (e.g. *scheduling* et *routing*) et l'implémentation de contraintes sur les intervalles de flottants (approximations des réels).

Les perspectives de développement s'orientent davantage vers la coopération de solveurs (e.g. avec un solveur linéaire) et le support pour la Recherche Locale. La PPC est pour

FaCiLe le cadre fédérateur destiné à accueillir des algorithmes hybrides. Le module « d'invariants backtrackables », qui utilisent les mécanismes génériques de gestion des contraintes, en constitue une première étape, et des techniques spécifiques de la RL comme la « réparation » de variables instanciées (variables logiques mixtes PPC/RL) seront proposées. D'autre part, des stratégies d'exploration plus flexibles, telle que la Limited Discrepancy Search, seront implémentées sous la forme de buts de recherche standards ; ces variantes de l'algorithme *backtrack*, comme celle qui implémente le Branch & Bound, pourront utiliser les mécanismes génériques de FaCiLe sans qu'il soit nécessaire de le modifier.

Applications

Les domaines d'application de l'optimisation combinatoire sont nombreux dans le monde industriel : *planning*, *scheduling*, *routing*, affectation de ressources, séquençage de molécules... La PPC est une technologie particulièrement appropriée car elle utilise un langage de spécifications très expressif qui permet de modéliser facilement les problèmes très variés qui conviennent au cadre générique des CSP. Dans le domaine de la gestion du trafic aérien, les problèmes d'aide à la décision posent souvent de délicates question d'interprétation et la modélisation est alors une étape cruciale de leur résolution. FaCiLe nous a permis d'expérimenter et de raffiner différents modèles pour résoudre les problèmes de l'allocation des créneaux de décollage, des schémas d'ouvertures des centres de contrôle et d'allocation de niveaux de vols dans un réseau de routes directes. Les contributions originales des deux premières études concernent essentiellement les différentes modélisations de ces problèmes en PPC destinées à rendre les solutions plus réalistes d'un point de vue opérationnel. La troisième utilise un algorithme hybride, qui n'a jamais été publié à notre connaissance, pour accélérer la recherche de solutions. Les résultats obtenus correspondent à des données réelles de journées de trafic aérien. Elles ont fait l'objet de plusieurs publications : [Barnier 00b], [Barnier 00a] et [Barnier 01d] concernent l'allocation de créneaux, et [Barnier 02a] celle des niveaux de vols.

Quelques applications de moindre importance, non mentionnées précédemment, ont été également réalisées avec notre librairie au sein du Laboratoire d'Optimisation Globale ou par des tiers.

Gestion des flux de trafic aérien Nous présentons plusieurs modèles du problème d'allocation de créneaux qui permettent de comparer les mérites de différentes interprétations de la capacité des secteurs de contrôle. Nous proposons notamment un modèle « continu » original utilisant des contraintes de tri. Alors que les formulations classiques de ce problème tendent à générer des pics de trafic périodiques, ce modèle permet de lisser le débit des flux d'avions entrant dans les secteurs de contrôle et de respecter les contraintes de capacité sur tout intervalle de temps d'une durée donnée. Nous tentons ainsi de rendre la charge de travail des contrôleurs plus régulière.

Le problème d'allocation de créneaux est déterminé par la donnée des schémas d'ouvertures de chacun des centres de contrôle, dont l'élaboration résulte en un problème de

planning et de partitionnement. Nous avons transformé le modèle de [Gianazza 02] (implémenté avec un algorithme de Branch & Bound *ad hoc*) qui optimise l'utilisation des positions de contrôle ouvertes en un modèle en PPC. Mais les solutions obtenues présentent des changements de configurations qui sont trop complexes et fréquentes. Nous proposons un raffinement original de ce modèle pour rendre plus réalistes les transitions entre configurations successives. Les performances de l'implémentation de ces modèles avec FaCiLe sont très supérieures (un ordre de magnitude pour les instances les plus grandes) à celles de l'algorithme *ad hoc* utilisé par [Gianazza 02].

Enfin, pour étudier la faisabilité d'un réseau de routes directes, nous avons calculé le nombre minimal de niveaux de vol en modélisant en PPC les flux d'avions et leur séparation comme un problème de coloriage de graphe. Pour améliorer l'efficacité du coloriage, les symétries entre les couleurs sont supprimées dynamiquement et une technique originale est présentée : des cliques sont découvertes au préalable par un algorithme glouton et sont utilisées pour poser des contraintes globales (qui détectent des inconsistances plus précocement) et pour guider la recherche. L'implémentation en FaCiLe de cet algorithme permet d'obtenir une solution optimale pour tous les flux de plus de deux avions. Elle donne également de bons résultats sur les instances réelles de benchmarks standards (de coloriage de graphe), ce qui montre la robustesse de la technique.

Autres applications FaCiLe a été utilisé avec succès dans d'autres domaines d'applications : l'optimisation de la taille d'encodeurs optiques absolus de position angulaire[13] utilisant des codes de GRAY cycliques (ce qui nécessite de nombreuses contraintes de modulo et d'indexation), la génération d'emplois du temps de candidats pour le Bureau des Concours de l'ENAC, l'anti-recouvrement d'étiquettes de plots radar d'un écran de contrôle aérien... FaCiLe a aussi permis à des utilisateurs externes d'optimiser du micro-code pour le processeur vectoriel de la Playstation II et d'évaluer les performances de programmes générés automatiquement pour le solveur Prolog SICStus. Le développement de FaCiLe a ainsi pu bénéficier des remarques et des comptes-rendus de programmeurs issus de milieux industriels différents.

[13] Ce type de composant est utilisé dans des capteurs électroniques qui doivent pouvoir être redémarrés instantanément.

Bibliographie

[Aggoun 90] A. Aggoun et N. Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. *Séminaire de Programmation Logique de Trégastel*. France, 1990. CNET. *cité page(s)* 117

[Alliot 96] Jean-Marc Alliot. *Techniques d'optimisation stochastique appliquées aux problèmes du contrôle aérien*. Thèse d'habilitation, Université de Toulouse Paul Sabatier, 1996. *cité page(s)* 78

[Alliot 98] Jean-Marc Alliot et Nicolas Durand. Faces : A free flight autonomous and coordinated embarked solver. *Proceedings of the Second Air Traffic Management R & D Seminar ATM-2000*. Orlando, USA, 1998. Eurocontrol & FAA. *cité page(s)* 179

[Bacchus 95] Fahiem Bacchus et Paul van Run. Dynamic Variable Ordering in CSPs. *Proceedings of the First International Conference on Constraint Programming*. Edited by Ugo Montanari et Francesca Rossi. Springer-Verlag, 1995, p 258–275. *cité page(s)* 68

[Bacchus 98] Fahiem Bacchus et Peter van Beek. On the conversion between non-binary and binary constraint satisfaction problems. *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*. AAAI Press, 1998, p 311–318. *cité page(s)* 35

[Bacchus 00] Fahiem Bacchus. A uniform view of backtracking. Unpublished, 2000. *cité page(s)* 69

[Bagley 01] Doug Bagley. *The Great Computer Language Shootout*. Web Page, 2001. `www.bagley.org/~doug/shootout`. *cité page(s)* 95

[Barnier 98] Nicolas Barnier et Pascal Brisset. Combine & conquer : Genetic algorithm and CP for optimization. *Conference on Principles and Practice of Constraint Programming CP'98 (Poster)*. Pisa, Italie, octobre 1998. *cité page(s)* 84

[Barnier 00a] Nicolas Barnier et Pascal Brisset. Allocation de créneaux pour la régulation du trafic aérien. *Journées Francophones de Program-*

mation Logique et Programmation par Contraintes JFPLC'2000. Marseille, France, avril 2000. *cité page(s)* 18, 191

[Barnier 00b] Nicolas BARNIER et Pascal BRISSET. Slot allocation in air traffic flow management. *Practical Application of Constraint Technologies and Logic Programming PACLP'2000*. Manchester, UK, avril 2000. *cité page(s)* 18, 191

[Barnier 01a] Nicolas BARNIER et Pascal BRISSET. FaCiLe : a Functional Constraint Library. *ALP Newsletter*, mai 2001, vol 14, n°2. *cité page(s)* 18, 190

[Barnier 01b] Nicolas BARNIER et Pascal BRISSET. FaCiLe : a Functional Constraint Library. *Colloquium on Implementation of Constraint and LOgic Programming Systems CICLOPS'01 (Workshop of CP'01)*. Paphos, Cyprus, décembre 2001. *cité page(s)* 18, 190

[Barnier 01c] Nicolas BARNIER et Pascal BRISSET. *FaCiLe : a Functional Constraint Library - User's and Reference Manual - Release 1.0*. CENA, Toulouse, France, juin 2001. *cité page(s)* 18, 20

[Barnier 01d] Nicolas BARNIER, Pascal BRISSET et Thomas RIVIÈRE. Slot allocation with constraint programming : Models and results. *International Air Traffic Management R&D Seminar ATM-2001*. Santa Fe (NM), USA, décembre 2001. *cité page(s)* 18, 191

[Barnier 02a] Nicolas BARNIER et Pascal BRISSET. Graph coloring for air traffic flow management. *CPAIOR'02 : Fourth Internation Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*. Le Croisic, France, mars 2002. *cité page(s)* 18, 191

[Barnier 02b] Nicolas BARNIER et Pascal BRISSET. Solving the Kirkman's schoolgirl problem in a few seconds. *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming - CP'02*. Ithaca (NY), USA, septembre 2002. *cité page(s)* 46, 72, 100

[Bent 01] Russell BENT et Pascal VAN HENTENRYCK. *A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows*. Rapport technique CS-01-06, Brown University, septembre 2001. *cité page(s)* 30, 84

[Bertsimas 95] Dimitris BERTSIMAS et Sarah STOCK. *The Air Traffic Flow Management Problem with Enroute Capacities*. Rapport technique, MIT, octobre 1995. *cité page(s)* 143

[Bessière 91] Christian BESSIÈRE. Arc-consistency in dynamic constraint satisfaction problems. *Proceeding of the Ninth National Conference on Artificial Intelligence*. 1991, p 221–226. *cité page(s)* 63

[Bessière 93] Christian Bessière et Marie-Odile Cordier. Arc-consistency and arc-consistency again. *AAAI*. 1993. *cité page(s)* 62

[Bessière 95] Christian Bessière, Eugene C. Freuder et Jean-Charles Régin. Using inference to reduce arc consistency computation. *Proceedings of IJCAI'95*. Montréal, Canada, 1995. *cité page(s)* 62

[Bessière 96] Christian Bessière et Jean-Charles Régin. MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. *Principles and Practice of Constraint Programming*. 1996, p 61–75. *cité page(s)* 68, 69, 106, 181

[Bessière 97] Christian Bessière et Jean-Charles Régin. Arc consistency for general constraint networks : Preliminary results. *Proceedings of IJCAI'97*. Nagoya, Japan, 1997. *cité page(s)* 40, 63

[Bleuzen-Guernalec 97] Noëlle Bleuzen-Guernalec et Alain Colmerauer. Narrowing a $2n$-block of sorting in $O(n \log n)$. *Third International Conference on Principles and Practice of Constraint Programming CP'97*. Springer-Verlag, 1997. *cité page(s)* 100, 130, 148

[Bomze 99] I. Bomze, M. Budinich, P. Pardalos et M. Pelillo. *The maximum clique problem. Handbook of Combinatorial Optimization*. Edited by D.-Z. Du et P. M. Pardalos, volume 4. Kluwer Academic Publishers, Boston, MA, 1999. *cité page(s)* 173

[Borning 81] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transaction on Programming Languages and Systems*. 1981, p 353–387. *cité page(s)* 133

[Borning 87] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer et M. Woolf. Constraint hierarchies. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Edited by Norman Meyrowitz. New York (NY), USA, 1987. ACM Press. *cité page(s)* 49

[Brélaz 79] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 1979, vol 22, p 251–256. *cité page(s)* 170, 171, 179, 180

[Brown 72] R. J. Brown. Chromatic scheduling and the chromatic number problem. *Management Science*, 1972, vol 19, p 451–463. *cité page(s)* 180

[Carlier 89] J. Carlier et E. Pinson. An algorithm for solving the jobshop problem. *Management Science*, 1989, vol 35, p 164–176. *cité page(s)* 135

[Carlsson 88] M. Carlsson et J. Widèn. *SICStus Prolog User's Manual*. Research Report SICS/R88007C, SICS, 1988. *cité page(s)* 29

[Caseau 94] Yves CASEAU et François LABURTHE. Improved CLP Scheduling with Task Intervals. *Proceedings of the 11th International Conference on Logic Programming, ICLP'94*. Edited by Pascal VAN HENTENRYCK. The MIT press, 1994. *cité page(s)* 85, 135

[Caseau 96] Yves CASEAU et François LABURTHE. *Introduction to the CLAIRE Programming Language*. Rapport technique, LIENS, 1996. *cité page(s)* 88, 94, 190

[Cerf 94] Raphaël CERF. *Une théorie asymptotique des algorithmes génétiques*. Thèse : Université de Montpellier, France, 1994. *cité page(s)* 79

[CFMU 00] Eurocontrol CFMU, Brussels. *Basic CFMU Handbook - General & CFMU Systems*, 6.0 edition, février 2000. *cité page(s)* 25, 140

[Choi 01a] Chiu Wo CHOI, Martin HENZ et Ka Boon NG. Components for state restoration in tree search. *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*. Edited by Toby WALSH. LNCS, Cyprus, 2001. Springer Verlag. *cité page(s)* 131, 136

[Choi 01b] Chui Wo CHOI, Martin HENZ et Ka Boon NG. A compositional framework for search. *Proceedings of CICLOPS : Colloquium on Implementation of Constraint and LOgic Programming Systems, appeared as Technical Report TR-CS-003/2001, New Mexico State University*. Edited by Enrico PONTELLI. Paphos, Cyprus, décembre 2001. *cité page(s)* 85

[Christofides 71] N. CHRISTOFIDES. An algorithm for the chromatic number of a graph. *Computer Journal*, 1971, vol 14, p 38–39. *cité page(s)* 180

[Codognet 96] Philippe CODOGNET et Daniel DIAZ. Compiling constraints in clp(FD). *Journal of Logic Programming*, 1996, vol 27, n°3, p 185–226. *cité page(s)* 87

[Colmerauer 82] Alain COLMERAUER. *Prolog-II Manuel de Référence et Modèle Théorique*. Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1982. *cité page(s)* 29, 86

[Colmerauer 90] A. COLMERAUER. An introduction to Prolog III. *CACM*, 1990, vol 33, n°7. *cité page(s)* 85, 93

[Colmerauer 96] Alain COLMERAUER. *Spécification de Prolog IV*. Rapport technique, Laboratoire d'informatique de Marseille, 1996. *cité page(s)* 86

[Cook 71] Stephen A. COOK. The complexity of theorem-proving procedures. *Proceedings of the Third Annual Symposium on Theory of Computing*. New York, USA, 1971. ACM. *cité page(s)* 37

[Coudert 97] Olivier COUDERT. Exact coloring of real-life graphs is easy. *Design Automation Conference DAC97*. Anaheim, California, juin 1997. *cité page(s)* 170, 180, 182

[CPAIOR 99] CP-AI-OR'99 : First international workshop on integration of ai and or techniques in constraint programming for combinatorial optimisation problems, Ferrara, Italy, février 1999. *cité page(s)* 30

[Culberson 92] Joseph C. CULBERSON. *Iterated Greedy Graph Coloring and the Difficulty Landscape*. Rapport technique TR 92-07, University of Alberta, Edmonton, Canada, 1992. *cité page(s)* 180

[Dantzig 49] G. DANTZIG. Programming in a linear structure. *Econometrics*, 1949, vol 17. *cité page(s)* 29

[De Backer 99] Bruno DE BACKER, Vincent FURNON et Paul SHAW. An object model for meta-heuristic search in constraint programming. *First International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems - CP-AI-OR'99*. Ferrara, Italy, février 1999. *cité page(s)* 88

[Debruyne 97] Romuald DEBRUYNE et Christian BESSIÈRE. Some practicable filtering techniques for the constraint satisfaction problem. *IJCAI*. 1997, p 412–417. *cité page(s)* 64

[Dechter 86] Rina DECHTER. Learning while searching in constraint satisfaction problems. *Proceedings of AAAI-86*. Philadelphia (PA), USA, 1986. *cité page(s)* 67

[Dechter 98] R. DECHTER et D. FROST. *Backtracking algorithms for constraint satisfaction problems – a survey*, 1998. *cité page(s)* 57, 59, 68

[Delahaye 95] Daniel DELAHAYE, Jean-Marc ALLIOT, Marc SCHOENAUER et Jean-Loup FARGES. Genetic algorithms for automatic regroupement of air traffic control sectors. *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. 1995. *cité page(s)* 158

[Diaz 02] Daniel DIAZ. *GNU Prolog*, 2002. `pauillac.inria.fr/~diaz/-gnu-prolog`. *cité page(s)* 87

[Dincbas 88] M. DINCBAS, P. Van HENTENRYCK, H. SIMONIS, A. AGGOUN et T. GRAF. The constraint logic programming language CHIP. *Int. Conf. Fifth Generation Computer Systems*. volume 1, Tokyo, Japan, 1988. *cité page(s)* 29, 86, 93

[Dollas 98] A. DOLLAS, W. RANKIN et D. MCCRACKEN. A new algorithm for Golomb ruler derivation and proof of the 19 marker ruler. *IEEE Transactions on Information Theory*, janvier 1998, vol 44, n°1, p 379–382. *cité page(s)* 127

[Dorne 98] R. DORNE et J.K. HAO. Meta-heuristics : Advances and trends in local search paradigms for optimization, chapitre 3. Tabu search for graph coloring, T-colorings and set T-colorings, p 77–92. Kluwer Academic Publishers, 1998. *cité page(s)* 180

[ECL^{i}PS^e 01] IC-Parc, Imperial College, London. *ECL^{i}PS^e User Manual (`www.icparc.ic.ac.uk/eclipse`)*, 2001. *cité page(s)* 29, 85, 86, 93, 120

[Eiben 95] A. E. Eiben et P. E. Rau. Improving the performance of GAs on a GA-hard CSP. *Proceedings of the Workshop on Studying and Solving Really Hard Problems - CP'95*. Cassis, France, 1995. *cité page(s)* 79

[Fages 01] François Fages et Emmanuel Coquery. Typing constraint logic programs. *Journal of Theory and Practice of Logic Programming TPLP*, novembre 2001, vol 1(6), p 751–777. *cité page(s)* 87, 94

[Fernández 00] Antonio J. Fernández et Patricia M. Hill. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Constraints*, juillet 2000, vol 5, n°3, p 275–301. *cité page(s)* 126

[Fink 98] A. Fink, S. Voss et D. Woodruff. Building reusable software components for heuristc search. *Operations Research Proceedings*. Edited by P. Kall et H.-J. Lüthi. Berlin, 1998. Springer. *cité page(s)* 29, 80

[Fleurent 94] C. Fleurent et J. A. Ferland. *Genetic and Hybrid Algorithm for Graph Coloring*. Rapport technique, Université de Montréal, 1994. *cité page(s)* 170, 179

[Focacci 99] Filippo Focacci, Andrea Lodi et Michela Milano. Cost-based domain filtering. *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming - CP'99*. LNCS, Alexandria, Virginia, USA, octobre 1999. Springer Verlag. *cité page(s)* 135

[Focacci 02] Filippo Focacci et Paul Shaw. Pruning sub-optimal search branches using local search. *CPAIOR'02 : Fourth Internation Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*. Le Croisic, France, mars 2002. *cité page(s)* 30, 85

[Fourer 93] Robert Fourer, David M. Gay et Brian W. Kernighan. AMPL : A modeling language for mathematical programming. The Scientific Press, San Francisco, CA, 1993. *cité page(s)* 29, 70, 88

[Freuder 89] Eugene C. Freuder et Richard J. Wallace. Partial Constraint Satisfaction. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA*. 1989, p 278–283. *cité page(s)* 49

[Freuder 95] Eugene C. Freuder, Rina Dechter, Matthew L. Ginsberg, Bart Selman et Edward P. K. Tsang. Systematic versus sto-

chastic constraint satisfaction. *IJCAI*. 1995, p 2027–2032. *cité page(s)* 80

[Freuder 96] Eugene C. Freuder. *In Pursuit of the Holy Grail. ACM Computing Surveys*, volume 28A(4). Association for Computing Machinery, 1996. *cité page(s)* 71

[Frost 95] Daniel Frost et Rina Dechter. Look-ahead Value Ordering For Constraint Satisfaction Problems. *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*. Montreal, Canada, 1995. *cité page(s)* 66, 67, 69

[Früwirth 98] T. Früwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, octobre 1998, vol 37, n°1–3, p 95–138. *cité page(s)* 87

[Garey 79] M.R. Garey et D.S. Johnson. *Computer and Intractability : A Guide to the Theory of NP-Completeness*. Freeman, 1979. *cité page(s)* 26, 37, 46

[Gaspero 00] Luca Di Gaspero et Andrea Schaerf. *EASYLOCAL++ : An object-oriented framework for flexible design of local search algorithms*, 2000. *cité page(s)* 88

[Gent 93] Ian P. Gent et Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. *National Conference on Artificial Intelligence*. 1993, p 28–33. *cité page(s)* 78

[Gent 00] Ian Gent et Barbara Smith. Symmetry breaking during search in constraint programming. *Proceedings of ECAI'2000*. Berlin, Germany, août 2000. *cité page(s)* 47

[Gervet 97] Carmen Gervet. Interval propagation to reason about sets : Definition and implementation of a practical language. *Constraints*, 1997, vol 1, n°3, p 191–244. `www.icparc.ic.ac.uk/~cg6`. *cité page(s)* 72, 118, 162, 163

[Gervet 01] Carmen Gervet. *Large Scale Combinatorial Optimization : A Methodological Viewpoint. Constraint Programming and Large Scale Discrete Optimization*. Edited by E.C. Freuder et R.J. Wallace, volume 57 of *DIMACS : Series in Discrete Mathematics and Theoretical Computer Science*, p 151–175. American Mathematical Society, DIMACS, 2001. *cité page(s)* 29

[Gianazza 02] David Gianazza et Jean-Marc Alliot. Optimization of air traffic control sector configurations using tree search methods and genetic algorithms. *21st Digital Avionics Systems Conference DASC*. Irvine (CA), USA, octobre 2002. *cité page(s)* 19, 157, 158, 160, 163, 165, 166, 192

[Ginsberg 90] Matthew L. Ginsberg et William D. Harvey. Iterative broadening. *National Conference on Artificial Intelligence*. 1990, p 216–220. *cité page(s)* 83

[Ginsberg 93] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1993, vol 1, p 25–46. *cité page(s)* 59, 82

[Glover 89] F. Glover. Tabu search - part i. *ORSA Journal on Computing*, 1989, vol 1, n°3, p 190–206. *cité page(s)* 78

[Goldberg 89] David Goldberg. *Genetic Algorithms*. Addison Wesley, 1989. *cité page(s)* 77

[Gotteland 00] Jean-Baptiste Gotteland, Philippe Kerlirzin, Serge Manchon et Christine Plusquellec. Building and evaluating a minimal regulation scheme. *3rd USA/Europe Air Traffic Management R&D Seminar*. Napoli, juin 2000. *cité page(s)* 140

[Grant 96] Stuart A. Grant et Barbara M. Smith. The phase transition behaviour of maintaining arc consistency. *European Conference on Artificial Intelligence*. 1996, p 175–179. *cité page(s)* 38

[Harvey 95a] William D. Harvey. *Nonsystematic Backtracking Search*. Thèse : CIRL, University of Oregon, 1995. *cité page(s)* 83

[Harvey 95b] William D. Harvey et Matthew L. Ginsberg. Limited discrepancy search. *Fourteenth International Joint Conference on Artificial Intelligence IJCAI'95*. Edited by Chris S. Mellish. volume 1, Montréal, Québec, Canada, août 1995. Morgan Kaufmann. *cité page(s)* 83, 131

[Henz 99] Martin Henz et Gert Smolka. Design of a finite domain constraint programming library for ML. `www.comp.nus.edu.sg/~henz/projects/rooms`, 1999. *cité page(s)* 95

[Hertz 87] A. Hertz et D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 1987, vol 39, p 345–351. *cité page(s)* 170, 179

[Hill 94] P. M. Hill et J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994. *cité page(s)* 87, 94

[Hopcroft 73] J. Hopcroft et R. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal of Computing*, 1973, vol 2, n°4, p 225–231. *cité page(s)* 127

[IP 99] Commission Européenne. *Fifteen Countries, a Single European Sky*, 1999. IP/99/924 (`europa.eu.int/comm/pr_en.htm`). *cité page(s)* 25, 139

[Jaffar 86] J. Jaffar et J.-L. Lassez. *Constraint Logic Programming.* Rapport technique 86/74, Monash University, Victoria, Australia, juin 1986. *cité page(s)* 71, 86, 87

[Jaffar 92] J. Jaffar, S. Michaylov, P. Stuckey et R. Yap. The clp(∇) language and system. *ACM Transactions on Programming Languages and Systems*, 1992, vol 14, n°3, p 339–395. *cité page(s)* 87

[Jaffar 98] Joxan Jaffar, Michael J. Maher, Kim G. Marriott et Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 1998, vol 37, p 1–46. *cité page(s)* 29, 71

[Jussien 99] Narendra Jussien et Olivier Lhomme. The path-repair algorithm. *CP99 Post-Conference Workshop on Large Scale Combinatorial Optimisation and Constraints.* Alexandria (VA), USA, octobre 1999. *cité page(s)* 81, 83

[Kirkpatrick 83] S. Kirkpatrick, C.D. Gelatt et M. Vecchi. Optimization by simulated annealing. *Science*, 1983, vol 220. *cité page(s)* 78

[Kirovski 98] Darko Kirovski et Miodrag Potkonjak. Efficient coloring of a large spectrum of graphs. *Design Automation Conference.* 1998, p 427–432. *cité page(s)* 182

[Kondrak 95] Grzegorz Kondrak et Peter van Beek. A theoretical evaluation of selected backtracking algorithms. *International Joint Conference on Artificial Intelligence IJCAI'95.* Edited by Chris Mellish. Montreal, 1995. *cité page(s)* 69

[Kumar 92] Vipin Kumar. Algorithms for constraint-satisfaction problems : A survey. *AI Magazine*, 1992, vol 13, n°1, p 32–44. *cité page(s)* 63, 68

[Laburthe 98a] Francois Laburthe et Yves Caseau. SaLSA : A language for search algorithm. *Proceedings of the Fourth Conference on Principles and Practice of Constraint Programming.* 1998. *cité page(s)* 30, 82, 85, 89, 190

[Laburthe 98b] Francois Laburthe et Yves Caseau. Salsa : A language for search algorithm. *Proceedings of the Fourth Conference on Principles and Practice of Constraint Programming.* 1998. *cité page(s)* 52

[Laburthe 00] François Laburthe. Choco : implementing a CP kernel. *TRICS Workshop - CP2000.* 2000. *cité page(s)* 20, 88, 93, 190

[Langley 92] P. Langley. Systematic and nonsystematic search strategies. *Proceedings of the First International Conference on Artificial Intelligence Planning Systems.* 1992, p 145–152. *cité page(s)* 83

[Le Huitouze 90] Serge Le Huitouze. Une nouvelle structure de données pour l'implémentation des extensions de Prolog. *Séminaire de Programmation Logique de Trégastel.* France, 1990. CNET. *cité page(s)* 120

[Leighton 79] Frank Thomson LEIGHTON. A graph colouring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 1979, vol 84, n°6, p 489–503. *cité page(s)* 170, 179

[Leroy 00] Xavier LEROY. *The Objective Caml System : User's and Reference Manual (*`http ://caml.inria.fr`*)*, 2000. *cité page(s)* 18, 20, 93, 189

[Letrouit 98] Vincent LETROUIT. *Optimisation du réseau des routes aériennes en Europe*. Thèse : Institut National Polytechnique de Grenoble, 1998. *cité page(s)* 170, 179, 185

[Lhomme 93] Olivier LHOMME. Consistency techniques for numeric CSPs. *IJCAI*. Chambéry, France, 1993. *cité page(s)* 74

[Lin 65] S. LIN. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 1965, vol 44, p 2245–2269. *cité page(s)* 75

[Lin 73] S. LIN et B. W. KERNIGHAN. An effective heuristic for the traveling salesman problem. *Operations Research*, 1973, vol 21, n°2, p 498–516. *cité page(s)* 78

[Mackworth 77] Alan K. MACKWORTH. Consistency in networks of relations. *Artificial Intelligence*, 1977, vol 8, p 99–118. *cité page(s)* 60, 61, 62

[Marsten 81] Roy E. MARSTEN. The design of the XMP linear programming library. *ACM Transactions on Mathematical Software*, décembre 1981, vol 7, n°4, p 481–497. *cité page(s)* 29

[Maugis 96] Lionnel MAUGIS. Mathematical programming for the air traffic flow management problem with en-route capacities. *in Proceedings of the 14th Triennial World Conference of the International Federation of Operational Research Societies*. juillet 1996. *cité page(s)* 143

[Maugis 98] L. MAUGIS, J.-B. GOTTELAND, R. ZANNI et P. KERLIRZIN. *TOSCA-II - WP3 : Assessment of the TMA to TMA hand-over concept*. Rapport technique TOSCA/SOF/WPR/3/03, SOFREAVIA, 1998. *cité page(s)* 170, 178

[McClain 99] David MCCLAIN. A comparison of programming languages for scientific processing. `www.azstarnet.com/~dmcclain/-LanguageStudy.html`, janvier 1999. *cité page(s)* 95

[Mehadhebi 00] Karim MEHADHEBI. A methodology for the design of a route network. *Proceedings of the Third Air Traffic Management R & D Seminar ATM-2000*. Napoli, Italy, juin 2000. Eurocontrol & FAA. *cité page(s)* 170, 179

[Michel 97] L. MICHEL et P. VAN HENTENRYCK. Localizer : A modeling language for local search. *Proceedings of the Third Conference*

on Principles and Practice of Constraint Programming. 1997. *cité page(s)* 21, 29, 80, 85, 89, 106, 133, 134, 188

[Michel 01a] Laurent MICHEL et Pascal VAN HENTENRYCK. *Localizer++ : An Open Library for Local Search*, janvier 2001. *cité page(s)* 88

[Michel 01b] Laurent MICHEL et Pascal VAN HENTENRYCK. OPL++ : A modeling layer for constraint programming libraries. *CP-AI-OR'2001*. Edited by Carmen GERVET et Mark WALLACE. Wye College (Imperial College), Ashford, Kent UK, avril 2001. IC-PARC. *cité page(s)* 107

[Milano 00] Michela MILANO, Greger OTTOSSON, Philippe REFALO et Erlendur S. THORSTEINSSON. The benefits of global constraints for the integration of constraint programming and integer programming. *Proceedings of the National Conference on Artificial Intelligence, Workshop on Integration of AI and OR Techniques for Combinatorial Optimization (AAAI-OR-00)*. AAAI, juillet 2000. *cité page(s)* 135

[Milano 02] Michela MILANO et Andrea ROLI. On the relation between complete and incomplete search : an informal discussion. *CPAIOR'02 : Fourth Internation Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*. Le Croisic, France, mars 2002. *cité page(s)* 30, 83

[Minton 94] Steven MINTON, Mark D. JOHNSTON, Andrew B. PHILIPS et Philip LAIRD. *Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. Constraint-based Reasoning*. Edited by Eugene C. FREUDER et Alan K. MACKWORTH. MIT Press, 1994. *cité page(s)* 75, 77, 84

[Mohr 86] R. MOHR et T.C. HENDERSON. Arc and path consistency revisited. *Artificial Intelligence*, 1986, vol 28, p 225–233. *cité page(s)* 62

[Morgenstern 86] Craig MORGENSTERN et Harry SHAPIRO. Chromatic number approximation using simulated annealing. *ACM Mountain Regional Meeting Proceedings*. 1986. *cité page(s)* 170, 179

[Morrisett 00] Greg MORRISETT et John REPPY. *The Third Annual ICFP Programming Contest*. Web Page, septembre 2000. `www.cs.cornell.edu/icfp`. *cité page(s)* 95

[Mozart 02] Universität des Saarlandes and Swedish Institute of Computer Science and Université catholique de Louvain. *The Mozart Programming System 1.2.4*, 2002. `www.mozart-oz.org`. *cité page(s)* 71, 85, 87

[Peemöller 83] J. PEEMÖLLER. A correction to Brélaz's modification of brown's coloring algorithm. *Communications of the ACM*, 1983, vol 26, p 595–597. *cité page(s)* 180

[Pesant 96] Gilles PESANT et Michel GENDREAU. A view of local search in constraint programming. *Second International Conference on Principles and Practice of Constraint Programming*. Edited by Eugene C. FREUDER. volume 1118 of *Lecture Notes in Computer Science*, Cambridge, Massachusetts, USA, août 1996. Springer. *cité page(s)* 30, 84

[Plusquellec 98] Christine PLUSQUELLEC et Serge MANCHON. *Description du module d'allocation de créneaux utilisant la Programmation Par Contraintes implanté dans SHAMAN*. Rapport technique, CENA, 1998. *cité page(s)* 140, 143

[Prestwich 01] Steven PRESTWICH. Local search and backtracking versus non-systematic backtracking. *Papers from the AAAI 2001 Fall Symposium on Using Uncertainty Within Computation*. North Falmouth, Cape Cod, MA, novembre 2001. *cité page(s)* 81, 83, 181

[Prosser 93] P. PROSSER. Hybrid algorithms for constraint satisfaction problems. *Computational Intellligence*, 1993, vol 9, n°3, p 268–299. *cité page(s)* 68

[Puget 92] Jean-François PUGET. PECOS : A high-level constraint programming language. *SPICIS'92*. Singapore, 1992. LNCS. *cité page(s)* 18, 29, 71, 85, 88, 95

[Puget 94] Jean-François PUGET. A C++ implementation of CLP. *Proceedings of the Second Singapore International Conference on Intelligent Systems SPICIS'94*. Singapore, novembre 1994. *cité page(s)* 71

[Régin 96] Jean-Charles RÉGIN. Generalized arc consistency for global cardinality constraint. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. 1996. *cité page(s)* 100, 128, 145

[Ridoux 95] Olivier RIDOUX. *Imagining CLP($\Lambda_{\alpha,\beta}$). Constraint Programming : Basics and Trends*. Edited by Andreas PODELSKI. Springer, 1995. *cité page(s)* 36

[Rivière 01a] Thomas RIVIÈRE. *Allocation de créneaux à la SHAMAN avec FaCiLe*. Rapport technique, CENA, 2001. *cité page(s)* 19, 161

[Rivière 01b] Thomas RIVIÈRE. *Robustesse des solutions du problème d'allocation de créneaux*. Rapport DEA : DEA IFP, INPT, 2001. *cité page(s)* 19

[Rossi 90] Francesca ROSSI, Charles PETRIE et Vasant DHAR. On the equivalence of constraint satisfaction problems. *ECAI'90 : Proceedings of the 9th European Conference on Artificial Intelligence*. Edited by Luigia Carlucci AIELLO. Stockholm, 1990. Pitman. *cité page(s)* 35, 46

[Rousseau 00] Louis-Martin Rousseau, Gilles Pesant et Michel Gendreau. A hybrid algorithm to solve a physician rostering problem. *CP-AI-OR'00*. Paderborn, Germany, mars 2000. *cité page(s)* 84

[Sabin 94] Daniel Sabin et Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94*. Edited by Alan Borning. volume 874, 1994, p 10–20. *cité page(s)* 68

[Sam-Haroud 95] D. Sam-Haroud. *Constraint consistency techniques for continuous domains*. Thèse : Swiss Federal Institute of Technology, Lausanne, Switzerland, 1995. *cité page(s)* 72

[Sannella 93] Michael Sannella, John Maloney, Bjorn N. Freeman-Benson et Alan Borning. Multi-way versus one-way constraints in user interfaces : Experience with the DeltaBlue algorithm. *Software - Practice and Experience*, 1993, vol 23, n°5, p 529–566. *cité page(s)* 133

[Saraswat 93] V.A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993. *cité page(s)* 71, 87

[Schaerf 00] Andrea Schaerf, Marco Cadoli et Maurizio Lenzerini. LOCAL++ : A C++ framework for local search algorithms. *Software - Practice and Experience*, 2000, vol 30, n°3, p 233–257. *cité page(s)* 29, 80, 85, 88

[Schiex 92] Thomas Schiex. Informatique théorique et intelligence artificielle, chapitre 14. Problèmes de satisfaction de contraintes. Cépaduès, 1992. *cité page(s)* 37, 64

[Schiex 93] Thomas Schiex et Gérard Verfaillie. Nogood recording for static and dynamic CSP. *Proceeding of the 5th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'93)*. Boston (MA), USA, novembre 1993. *cité page(s)* 57, 59

[Schiex 95] Thomas Schiex, Hélène Fargier et Gérard Verfaillie. Valued constraint satisfaction problems : Hard and easy problems. *IJCAI'95 : Proceedings of the International Joint Conference on Artificial Intelligence*. Edited by Chris Mellish. Montreal, Canada, 1995. *cité page(s)* 49

[SCOOP 98] SINTEF. *SCOOP 2.0 Reference Manual*, 1998. STF 42A98001. *cité page(s)* 29, 80

[Sellmann 02] Meinolf Sellmann et Warwick Harvey. Heuristic constraint propagation. *CPAIOR'02 : Fourth Internation Workshop on Integration of AI and OR Techniques in Constraint Programming for*

Combinatorial Optimisation Problems. Le Croisic, France, mars 2002. *cité page(s)* 30, 84

[Selman 92] Bart SELMAN, Hector J. LEVESQUE et D. MITCHELL. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*. Edited by Paul ROSENBLOOM et Peter SZOLOVITS. Menlo Park, California, 1992. AAAI Press. *cité page(s)* 75, 77

[Selman 93] Bart SELMAN et Henry KAUTZ. Domain-independent extensions to gsat : Solving large structured satisfiability problems. *Proceedings of IJCAI'93*. 1993. *cité page(s)* 78

[Selman 94] Bart SELMAN, Henry A. KAUTZ et Bram COHEN. Noise strategies for improving local search. *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*. 1994, p 337–343. *cité page(s)* 78

[Shaw 98] Paul SHAW. Using constraint programming and local search methods to solve vehicle routing problems. *Principles and Practice of Constraint Programming*. 1998, p 417–431. *cité page(s)* 84

[Siskind 93] Jeffrey Mark SISKIND et David Allen MCALLESTER. Nondeterministic Lisp as a substrate for constraint logic programming. *Proceedings of the Eleventh National Conference on Artificial Intelligence*. Edited by Richard FIKES et Wendy LEHNERT. Menlo Park, California, 1993. AAAI Press. *cité page(s)* 95

[Slama 02] Soufiene SLAMA. *Compilation d'OPL avec FaCiLe*. Rapport DEA : DEA Programmation et Systèmes, INPT, 2002. *cité page(s)* 107

[Smith 95] Barbara M. SMITH et Stuart A. GRANT. *Where the Exceptionally Hard Problems Are*. Rapport technique 95.35, University of Leeds, 1995. *cité page(s)* 68, 69

[Smith 99a] Barbara M. SMITH. *The Brélaz Heuristic and Optimal Static Orderings*. Research Report 1999.13, School of Computer Studies, University of Leeds, juin 1999. *cité page(s)* 66, 69, 180

[Smith 99b] Barbara SMITH, Kostas STERGIOU et Toby WALSH. *Modelling the Golomb Ruler Problem*. Rapport technique 1999.12, School of Computing - University of Leeds, juin 1999. *cité page(s)* 127

[Smolka 98] Gert SMOLKA. Concurrent constraint programming based on functional programming. *Programming Languages and Systems*. Edited by Chris HANKIN. Lecture Notes in Computer Science, vol. 1381, Lisbon, Portugal, 1998. Springer-Verlag. *cité page(s)* 94

[Solver 99] SOLVER. *ILOG Solver 4.4 User's Manual (*`www.ilog.fr`*)*, 1999. *cité page(s)* 29, 85, 88, 93, 94

[Somogyi 95] Zoltan SOMOGYI, Fergus HENDERSON et Thomas CONWAY. Mercury : an efficient purely declarative logic programming language. *Proceedings of the Australian Computer Science Conference*. Glenelg, Australia, février 1995. *cité page(s)* 87, 94

[Sutherland 63] I. E. SUTHERLAND. *Sketchpad : A Man-Machine Graphical Communication System*. Rapport technique 296, MIT Lincoln Laboratory, 1963. *cité page(s)* 133

[Taillard 98] E. D. TAILLARD, L.-M. GAMBARDELLA, M. GENDREAU et J.-Y. POTVIN. *Adaptive Memory Programming : A Unified View of Meta-Heuristics*. Rapport technique IDSIA-19-98, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, 1998. *cité page(s)* 29, 80, 88

[Talbi 99] El-ghazali TALBI. *Métaheuristiques pour l'optimisation combinatoire multi-objectif : État de l'art*. Rapport technique, Rapport CNET (France Télécom), octobre 1999. *cité page(s)* 79

[Torrens 97] Marc TORRENS, Rainer WEIGEL et Boi FALTINGS. *Java Constraint Library*. Working Notes of the Swiss Workshop on Collaborative and Distributed Systems, mai 1997. Lausanne, Switzerland. *cité page(s)* 29, 88

[Trick 94] Michael TRICK. *Network Resources for Coloring a Graph*. `mat.gsia.cmu.edu/COLOR/color.html`, 1994. *cité page(s)* 170, 171, 181

[Tsang 99] E.P.K. TSANG, C.J. WANG, A. DAVENPORT, C. VOUDOURIS et T.L. LAU. A family of stochastic methods for constraint satisfaction and optimization. *The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP)*. London, UK, avril 1999. *cité page(s)* 79

[Van Hentenryck 90] Pascal VAN HENTENRYCK. A logic language for combinatorial optimization. *Annals of Operations Research*, 1990, vol 21, p 247–274. *cité page(s)* 170, 181

[Van Hentenryck 95] Pascal VAN HENTENRYCK. Constraint solving for combinatorial search problems : a tutorial. *Principle and Practice of Constraint Programming CP'95*. Edited by Ugo MONTANARI et Francesca ROSSI. volume 976 of *Lecture Notes in Computer Science*, Cassis, France, septembre 1995. Springer. *cité page(s)* 37

[Van Hentenryck 99] Pascal VAN HENTENRYCK. *The OPL Optimization Programming Language*. The MIT Press, Cambridge (MA), USA, 1999. *cité page(s)* 85, 88, 107, 190

[Verfaillie 95] Gérard VERFAILLIE et Thomas SCHIEX. Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes : bilan de

quelques approches. *Revue d'Intelligence Artificielle*, 1995, vol 9, n°3, p 269–309. *cité page(s)* 50

[Verlhac 01] Céline VERLHAC et Serge MANCHON. Optimization of opening schemes. *International Air Traffic Management R&D Seminar ATM-2001*. Santa Fe (NM), USA, décembre 2001. *cité page(s)* 158

[Yagiura 99] Mutsunori YAGIURA et Toshihide IBARAKI. On metaheuristic algorithms for combinatorial optimization problems. Systems and Computers in Japan (to appear), 1999. *cité page(s)* 29, 79

[Zhou 96] Jianyang ZHOU. A constraint program for solving the job-shop problem. *Principles and Practice of Constraint Programming*. Springer-Verlag, 1996. *cité page(s)* 150