Contributions à la résolution pratique de problèmes combinatoires

—des fourmis et des graphes—

Mémoire pour l'obtention de l'Habilitation à Diriger des Recherches

de l'Université Claude Bernard Lyon 1

Christine Solnon

Présenté le 6 décembre 2005 devant le jury composé de
- Patrick Albert, Conseiller technique, Ilog S.A., Gentilly, Examinateur
- Marco Dorigo, Directeur de recherches du FNRS, IRIDIA, Bruxelles, Rapporteur
- François Fages, Directeur de recherches, INRIA, Rocquencourt, Rapporteur
- Jean-Michel Jolion, Professeur, LIRIS, INSA de Lyon, Examinateur
- Alain Mille, Professeur, LIRIS, Université Lyon 1, Examinateur
- Michel Rueher, Professeur, I3S, Sophia Antipolis, Examinateur
- Gérard Verfaillie, Maître de recherches, ONERA, Toulouse, Rapporteur

Un grand merci...

...à tous !

(C'est promis, dès que j'ai le temps je développerai des remerciements chaleureux et personnalisés à tous ceux qui m'ont permis d'en arriver là, et ils sont nombreux : tous les membres de mon « dream jury », tous les chercheurs avec qui j'ai eu l'occasion et le plaisir de travailler, les étudiants en stages de Master/DEA et en thèses bien sûr, mes amis et collègues du Nautibus, de l'IUT et d'ailleurs, ma famille, ... et lirispaj qui a effectué toutes les expérimentations présentées ici sans jamais rechigner !)

Table des matières

Première partie

Synthèse des activités

Chapitre 1

Curriculum Vitæ

Christine SOLNON
née le 11 avril 1967 à Béthune (62)
3 enfants nés en 1992, 1995 et 1998

Adresse professionnelle : LIRIS UMR 5205 CNRS
Université Lyon 1
Nautibus, 43 bd du 11 novembre
69622 Villeurbanne cedex

e-mail : `christine.solnon@liris.cnrs.fr`
URL : `http ://www710.univ-lyon1.fr/~csolnon`

Diplômes

– Juin 1989 : *D.E.A. Informatique fondamentale*
  Université de Nice - Mention Bien
– Février 1993 : *Thèse de doctorat*
  Université de Nice Sophia Antipolis - Mention très honorable
  Titre : Un système d'inférence de relations inter-types pour Prolog
  — application à la génération de modèles objets —
  Jury : Michel RUEHER (directeur), Jean-Claude BOUSSARD (président),
  Pierre DERANSART et Paul-Yves GLOESS (rapporteurs),
  Gille KAHN et Jacques MORGENSTERN (examinateurs).

Emplois

– Septembre 1989 / Septembre 1992 : *Allocataire de recherche / Monitrice*
  Université de Nice Sophia Antipolis
– Septembre 1992 / Septembre 1994 : *Attachée Temporaire d'Enseignement et de Recherche* à mi-temps
  Université de Nice Sophia Antipolis
– depuis Septembre 1994 : *Maître de Conférences*
  Université Lyon 1, département informatique de l'IUT A

Bénéficiaire de la prime d'encadrement doctoral et de recherche (PEDR) depuis septembre 2004

Chapitre 2

Activités de recherche

2.1 Environnement et orientation générale de mes recherches

J'ai préparé ma thèse de doctorat sous la direction de Michel RUEHER, au laboratoire I3S de l'Université de Nice Sophia Antipolis. J'ai soutenu cette thèse en février 1993, et je suis arrivée à l'Université Lyon 1 en septembre 1994. J'ai alors intégré le groupe « Modélisation des connaissances » animé par Jean-Marc FOUET au sein du laboratoire LISI. A la mort de Jean-Marc FOUET, en mai 2000, l'animation de ce groupe a été reprise par Alain MILLE qui venait d'arriver au LISI. Au début de l'année 2003, le laboratoire LISI a fusionné avec d'autres laboratoires Lyonnais pour former le laboratoire LIRIS, et j'ai alors intégré l'axe « Données, Documents et Connaissances » dirigé par Alain MILLE.

D'une façon générale, ma thématique de recherche s'inscrit dans le cadre des problèmes combinatoires $\mathcal{NP}$-difficiles, problèmes qui ne peuvent être résolus de façon exacte en temps polynomial (à moins que $\mathcal{P} = \mathcal{NP}$). Dans ce contexte, je me suis intéressée à différents problèmes combinatoires (problèmes de satisfaction de contraintes, de recherche de cliques maximum, de sac-à-dos, d'appariement de graphes, ...), différentes applications (analyse statique de programmes Prolog, raisonnement à partir de cas, recherche d'informations, planification, ...) et différentes approches de résolution (programmation (logique) par contraintes, programmation par ensembles-réponses, optimisation par colonies de fourmis, recherche locale, ...).

Ces travaux peuvent être regroupés en cinq thèmes que nous présentons brièvement ci-dessous, en faisant apparaître leur enchaînement chronologique et leur articulation. Une présentation plus détaillée des deux thèmes principaux, constituant le cœur de mes activités de recherche, est ensuite faite dans la section suivante.

de 1989 à 1994 : Analyse statique de programmes Prolog. Ma thèse de doctorat, préparée sous la direction de Michel RUEHER au laboratoire I3S de l'Université de Nice Sophia Antipolis, a porté sur l'analyse statique de programmes Prolog. L'objectif était de faciliter la mise au point et la compréhension d'un programme Prolog en fournissant une vue schématique de sa structure. Dans ce contexte, ma contribution a porté sur les points suivants :

- Définition d'un système d'inférence de types : ce système infère, à partir d'un programme Prolog, une collection d'équations décrivant les relations ensemblistes entre les termes du programme. Le plus petit modèle de ces équations approxime (est un sur-ensemble de) la sémantique dénotationnelle du programme et correspond au plus petit point fixe d'une abstraction α de l'opérateur de conséquence immédiate T_P, i.e., $lfp(\alpha(T_P))$.
- Définition d'un système d'inférence de relations d'héritage : ce système déduit des relations de sous-typage à partir de l'ensemble d'équations générées par le système d'inférence de types. On a introduit un raisonnement par induction structurelle pour garantir la terminaison dans le cas d'équations récursives.

– Définition d'un processus de transformation de programmes Prolog : l'opérateur d'abstraction α perd les relations de dépendance inter-arguments définies par un programme Prolog, de sorte que l'approximation calculée ($lfp(\alpha(T_P))$) peut devenir très mauvaise dans certains cas. On a donc défini un processus de transformation de programmes qui permet de propager certaines de ces relations de dépendance par des techniques d'évaluation partielle ; le programme résultant est sémantiquement équivalent, mais son approximation calculée est meilleure que celle du programme de départ.
– Implémentation de ces trois systèmes en Prolog III, langage de programmation en logique avec contraintes.

Si ces travaux s'intègrent essentiellement dans le contexte de la programmation en logique, ils ont été l'occasion d'un premier contact avec des problèmes combinatoires. En particulier, j'ai utilisé des systèmes de contraintes ensemblistes pour décrire la structure des programmes Prolog analysés, et j'ai utilisé la programmation logique par contraintes pour résoudre ces systèmes de contraintes ensemblistes.

Ce thème de recherche a fait l'objet de six publications entre 1991 et 1994 [LOPSTR-91, PLILP-92, JFPL-92, LPAR-93, LOPSTR-94, JFPL-94].

de 1994 à 1997 : Résolution de problèmes de satisfaction de contraintes par coopération de solveurs.
A mon arrivée dans l'équipe de Jean-Marc FOUET au LISI, j'ai orienté mes recherches sur la résolution de problèmes de satisfaction de contraintes, thème plus proche des activités de l'équipe que j'intégrais. Je me suis alors intéressée aux approches coopératives, faisant collaborer plusieurs algorithmes de résolution. Je me suis plus particulièrement intéressée à deux types de problèmes :

– Les problèmes « mixtes » sur les réels.
De nombreux problèmes industriels, notamment en conception mécanique, peuvent être modélisés par un ensemble de contraintes portant sur des variables dont les domaines de valeurs sont des intervalles définis sur les réels. Ces systèmes de contraintes sont généralement « mixtes » dans le sens où ils comprennent des équations et inéquations linéaires, mais aussi des équations non linéaires et souvent non polynomiales. Une approche coopérative permet d'appréhender de tels problèmes en combinant différents algorithmes de résolution, e.g., algorithme de Gauss pour les équations linéaires, algorithme du Simplexe pour les inéquations linéaires, et algorithmes de propagation d'intervalles pour les équations non linéaires.
Dans ce contexte, nous avons défini une nouvelle architecture concurrente permettant de faire coopérer un solveur linéaire fondé sur les algorithmes de Gauss et du Simplexe avec un solveur non linéaire fondé sur la propagation d'intervalles. Cette nouvelle architecture concurrente permet d'accélérer la résolution en coupant certains processus de convergence lente de l'algorithme de propagation. Nous avons également défini un solveur linéaire « actif » qui utilise l'algorithme du Simplexe pour calculer les bornes des variables par rapport aux contraintes linéaires. Ce nouveau solveur linéaire permet de résoudre de façon plus précise les problèmes dans notre approche coopérative.

Ce travail a été fait en collaboration avec Michel RUEHER, Professeur à l'Université de Nice Sophia Antipolis, et a fait l'objet de deux publications en 1996 et 1997 [Interval-96, RC-97].

– Les problèmes linéaires en nombres entiers.
De nombreux problèmes d'optimisation peuvent être modélisés sous la forme de problèmes linéaires en nombres entiers, e.g., les problèmes de planification de production ou d'allocation de ressources. Ces problèmes consistent à optimiser une fonction objectif linéaire portant sur un ensemble de variables de décision soumises à des contraintes (équations ou inéquations) linéaires. La combinatoire vient du fait que les variables de décision doivent prendre des valeurs entières. Pour résoudre ces problèmes, on utilise généralement une approche par « séparation et évaluation », qui effectue une exploration complète de l'ensemble des solutions (structuré en arbre) et utilise un solveur —généralement fondé sur l'algorithme du Simplexe— pour réduire la combinatoire.
Dans ce contexte, ma contribution a porté sur la définition d'une approche coopérative pour la résolution de problèmes linéaires en nombres entiers. L'idée est de faire coopérer un ensemble de solveurs linéaires (basés sur l'algorithme du Simplexe) pour calculer les domaines des variables sur les réels, puis de réduire ces domaines à l'intervalle entier inclus le plus proche. Cette réduction de domaines est effectuée jusqu'à

atteindre un point fixe où tous les domaines sont bornés par des entiers appartenant à l'espace des solutions sur les réels. Ce point fixe correspond à une consistance partielle —appelée LIB-consistance— qui a été formellement caractérisée.

Ce travail a fait l'objet de deux publications en 1997 [JFPLC-97, ICTAI-97]

depuis 1999 : Optimisation par colonies de fourmis. J'ai bénéficié d'un congé maternité de janvier à septembre 1998 pour la naissance de mon troisième enfant. Cette coupure a été l'occasion d'orienter mes activités de recherche dans une nouvelle direction, tout en restant dans le thème général de la résolution de problèmes combinatoires.

Les approches coopératives sur lesquelles j'ai travaillé de 1994 à 1997 sont caractérisées par un contrôle centralisé de la résolution, avec un parcours global et complet de l'espace de recherche. Les systèmes multi-agents proposent une architecture différente —flexible et décentralisée— pour la résolution coopérative de problèmes. Ainsi, je me suis intéressée aux possibilités d'utilisation de systèmes multi-agents pour la résolution de problèmes combinatoires. J'ai bénéficié pour cela d'un environnement local particulièrement actif dans le domaine des systèmes multi-agents, et j'ai notamment participé aux réunions du groupe de travail Lyonnais sur les systèmes multi-agents animé par Salima HASSAS, Maître de Conférences au LIRIS.

Je me suis alors plus particulièrement intéressée aux systèmes multi-agents réactifs du type « colonies de fourmis », et à leur application à la résolution de problèmes combinatoires. Ce thème de recherche est devenu un des deux thèmes les plus significatifs de mes activités de recherche, et est détaillé à la section 2.2.1.

Ce thème de recherche a été l'occasion de six stages de DEA ou Master recherche, dont deux se poursuivent par des thèses, et de collaborations avec quatre équipes de recherche étrangères ; il a fait l'objet de quinze publications entre 2000 et 2005 [ANTS-00, ECAI-00, JFPLC-00, CP-01, JNPC-01, TEC-02, EvoCOP-02, EvoCOP-03a, EvoCOP-03b, ROADEF-03, EvoCOP-04, Bioma-04, Francoro-04, EvoCOP-05, Heuristics-05]. Mon investissement dans ce thème de recherche se traduit également par mon appartenance au réseau d'excellence européen sur les algorithmes évolutionnistes EvoNet, et ma participation aux comités de programme de quatre conférences internationales sur ce sujet.

depuis 2002 : Appariement de graphes et similarité. L'arrivée d'Alain MILLE au LISI, en 2000, a été pour moi l'occasion de découvrir un nouveau domaine de recherche, i.e., le raisonnement à partir de cas, et une nouvelle problématique au sein de ce domaine de recherche, i.e., mesurer la similarité d'objets. Cette problématique a été plus particulièrement motivée par les travaux de Pierre-Antoine CHAMPIN, alors en thèse sous la direction d'Alain MILLE, travaux qui posaient la question de l'évaluation de la similarité d'objets de conception décrits par des graphes. Je me suis alors intéressée à ce problème, qui se ramène à une recherche d'un « meilleur » appariement entre les sommets des graphes, et qui est un problème très clairement combinatoire.

Ce thème de recherche, commencé de façon opportuniste avec l'arrivée d'Alain MILLE et Pierre-Antoine CHAMPIN au LISI, est devenu un des deux thèmes les plus significatifs de mes activités de recherche. Il est plus longuement décrit à la section 2.2.2.

Ce thème de recherche a été l'occasion de deux stages de DEA, tous deux se poursuivant par des thèses, et a fait l'objet de sept publications entre 2003 et 2005 [ICCBR-03, JNPC-03, CPAIOR-04, JFPLC-04, GbR-05, RJCIA-05, EvoCOP-05]. Il a également débouché sur le projet « Graph-I-Sim », projet financé par une ATIP « jeunes chercheurs » du CNRS et dont je suis porteuse.

depuis 2002 : Programmation par ensembles-réponses pour la planification de l'évacuation. Robert LAURINI, Professeur au LIRIS, m'a proposé en 2002 de co-encadrer la thèse de Claudia ZEPEDA CORTES, le sujet de cette thèse présentant des liens forts avec mes activités de recherche. L'objectif de cette thèse est d'étudier les capacités de la programmation par ensembles-réponses pour modéliser des situations de catastrophes naturelles et assister la définition de plans d'évacuation de la population. La programmation par ensembles-réponses, ou Answer Set Programming, est un paradigme de programmation en logique où la sémantique des programmes est définie en termes d'ensembles de littéraux —appelés ensembles-réponses— qui sont des modèles du programme. Ce paradigme de programmation permet de représenter de façon très naturelle de nombreux problèmes issus de l'intelligence artificielle, notamment lorsque les connaissances sont incomplètes, lorsque l'on veut permettre des raisonnements non monotones, ou lorsque l'on souhaite exprimer des préférences. Dans ce contexte, les contributions concernent notamment les points suivants :

- L'étude des spécificités des problèmes de planification de l'évacuation et la proposition d'un cadre générique pour modéliser et résoudre ce type de problème à l'aide de la programmation par ensembles-réponses. Ce travail a été validé sur un cas réel, à savoir l'évacuation de la population en cas d'éruption du Popocatepetl au Mexique.
- L'introduction de la notion de « contenu sémantique » (Semantic Content) d'un programme logique. Ce contenu sémantique offre un cadre unifié pour obtenir différentes sémantiques basées sur la notion d'ensembles-réponses, et notamment les ensembles-réponses généralisés minimaux et les ensembles-réponses préférés qui sont utilisés dans cette thèse pour rechercher les meilleurs plans d'évacuation en fonction de différentes préférences.
- L'étude des possibilités de la programmation par ensembles-réponses pour obtenir des plans d'évacuation alternatifs. Il s'agit de pouvoir exprimer le fait que si on préfère généralement utiliser le plan d'évacuation prédéfini, dans le cas où certains tronçons de routes de ce plan sont bloqués, on doit chercher des alternatives à ce plan prédéfini. Cette étude a porté sur deux approches : l'utilisation des règles de restauration de la consistance de CR-Prolog et l'utilisation de la double négation pour exprimer des préférences entre les ensembles-réponses, et une procédure de transformation de programmes pour calculer les meilleurs ensembles-réponses par rapport à ces préférences.

Ce travail a fait l'objet de trois publications en 2004 et 2005 [LA-NMR-04, ASP-05, MICAI-05].

2.2 Principaux thèmes de recherche

2.2.1 Optimisation par colonies de fourmis

Les fourmis sont capables de résoudre collectivement des problèmes complexes, comme trouver le plus court chemin entre deux points dans un environnement accidenté. Pour cela, elles communiquent entre elles de façon locale et indirecte, en déposant derrière elles des traces d'une hormone volatile appelée phéromone. Ces traces attirent les autres fourmis passant à proximité, selon une probabilité dépendant de la quantité de phéromone déposée. Ces fourmis renforcent elles-même ces traces selon un mécanisme de rétroaction positive qui permet à la colonie de converger sur les plus courts chemins.

Ce mécanisme de résolution collective de problèmes est à l'origine des algorithmes à base de fourmis, dont le premier a été proposé en 1992 par Marco DORIGO pour résoudre le problème du voyageur de commerce. L'idée est de représenter le problème à résoudre sous la forme de la recherche d'un meilleur chemin dans un graphe, puis d'utiliser des fourmis artificielles pour rechercher de bons chemins dans ce graphe. Le comportement des fourmis artificielles est inspiré des fourmis réelles : elles déposent des traces de phéromone sur les composants du graphe et elles choisissent leurs chemins en fonction de probabilités dépendant des traces de phéromone précédemment déposées ; ces traces sont évaporées au cours du temps. Intuitivement, cette communication indirecte fournit une information sur la qualité des chemins empruntés afin d'attirer les fourmis, dans les itérations futures, vers les zones correspondantes de l'espace de recherche. Ces caractéristiques du comportement des fourmis artificielles définissent la « méta-heuristique d'optimisation par une colonie de fourmis » ou « Ant Colony Optimization (ACO) metaheuristic ».

Dans ce contexte, notre contribution porte d'une part sur l'application de la méta-heuristique ACO à la résolution de problèmes combinatoires, et d'autre part sur l'étude et l'amélioration des mécanismes permettant aux colonies de fourmis de converger vers de bonnes solutions.

Applications de la méta-heuristique ACO. Nous avons proposé de nouveaux algorithmes à base de fourmis pour les problèmes combinatoires suivants :

- Les problèmes de satisfiabilité de formules booléennes (SAT), dont le but est de trouver une valuation de variables logiques satisfaisant un ensemble de formules booléennes sous forme clausale.

  Ce travail est décrit dans [ANTS-00].

  Il a été effectué en collaboration avec Simone PIMONT, *Maître de Conférences au LIRIS, et a en partie fait l'objet du stage de DEA de Jean* DUONG.

- Les problèmes de satisfaction de contraintes, dont le but est d'affecter des valeurs à des variables soumises à des contraintes.

  Ce travail est décrit dans [CP-01, TEC-02].

- Les problèmes de recherche de permutations sous contraintes, dont le but est de rechercher une permutation d'un ensemble de valeurs satisfaisant un certain nombre de contraintes.

  Ce travail est décrit dans [JFPLC-00, ECAI-00].

- Le problème d'ordonnancement de voitures, dont le but est de séquencer des voitures sur une chaine d'assemblage en respectant des contraintes de capacité liées aux options à installer sur les voitures ; il s'agit là d'un cas particulier du problème de recherche de permutations sous contraintes pour lequel nous avons spécialisé l'algorithme décrit dans [ECAI-00]. Nous avons alors comparé les performances de ACO avec celles de l'algorithme de recherche locale développé à SAP pour ce problème.

  Ce travail est décrit dans [EvoCOP-03-a].

  Il a été effectué en collaboration avec Jens GOTTLIEB *et Markus* PUCHTA, *de la société SAP (Allemagne).*

  J'ai par ailleurs été invitée par les éditeurs du numéro spécial de la revue EJOR (European Journal of Operational Research) sur le problème d'ordonnancement de voitures, Christian Artigues, Van-Dat Cung et Alain Nguyen, à rédiger un article de synthèse sur ce problème et les différentes approches utilisées pour le résoudre, en introduction au numéro spécial à paraître en 2006.

- Les problèmes de recherche de cliques maximum dans des graphes, dont le but est de rechercher dans un graphe le plus grand sous-ensemble de sommets tous connectés deux à deux. Nous avons proposé et comparé deux algorithmes ACO, basés sur deux stratégies phéromonales différentes.

  Ce travail est décrit dans [EvoCOP-03-b, ROADEF-03, Heuristics-05].

  Il a été fait en collaboration avec Serge FENET, *Maître de Conférences au LIRIS.*

- Les problèmes de sac-à-dos multidimensionnels, dont le but est de sélectionner un sous-ensemble d'objets maximisant une fonction profit tout en respectant des contraintes de capacités, la fonction profit et les contraintes étant linéaires.

  Ce travail est décrit dans [Bioma-04, Francoro-04].

  Il a été fait en collaboration avec Khaled GHEDIRA, *Professeur à l'IEG de Tunis (Tunisie), et a fait l'objet du stage de Master d'Inès* ALAYA. *Ce stage se poursuit actuellement par une thèse de doctorat qui a débuté en janvier 2005.*

- Les problèmes d'appariement de graphes, dont le but est de trouver l'appariement de sommets entre deux graphes qui maximise leur similarité.

Ce travail est décrit dans [EvoCOP-05].

Il a été fait en collaboration avec Khaled GHEDIRA*, Professeur à l'IEG de Tunis (Tunisie), et a fait l'objet du stage de Master d'Olfa* SAMMOUD*. Ce stage se poursuit actuellement par une thèse de doctorat qui a débuté en septembre 2005.*

Plusieurs de ces problèmes appartiennent à la classe plus générale des problèmes de recherche de sous-ensembles, i.e., les problèmes de satisfaction de contraintes, de recherche de cliques maximum, de sac-à-dos multidimensionnels et d'appariement de graphes. Nous avons généralisé les différents algorithmes ACO que nous avons proposé pour ces différents problèmes en un algorithme générique pour la résolution de ces problèmes de recherche de sous-ensembles.

Ce travail est décrit dans [Particle-Swarm-Optimization-06].

Il a été fait en collaboration avec Derek BRIDGE*, Professeur à l'université de Cork (Irlande).*

Etude et amélioration de la méta-heuristique ACO. La méta-heuristique ACO utilise un mécanisme de rétroaction —via les dépots de phéromone— pour faire converger la colonie de fourmis vers de « bonnes » solutions. Il s'agit d'un mécanisme complexe que nous avons eu l'occasion d'étudier à travers les différentes applications auxquelles nous nous sommes intéressés. Nous avons alors proposé différentes approches pour améliorer les performances d'algorithmes ACO :

– Nous avons proposé d'utiliser des techniques de fouille de données et de classification pour extraire, à partir des traces de phéromone, des informations permettant de doper les performances d'algorithmes utilisant la méta-heuristique ACO.

  Ce travail est décrit dans [JNPC-01].

  Il a fait l'objet du stage de DEA d'Arnaud STUBER.

– Nous avons étudié les possibilités d'utilisation des algorithmes par estimation de distributions (EDA), pour doper les performances de ACO.

  Ce point a fait l'objet du stage de DEA de Alain-Pierre MANINE.

– J'ai défini une nouvelle technique d'initialisation des traces de phéromone, par échantillonnage de l'espace de recherche, permettant d'accélérer la convergence des algorithmes.

  Ce travail est décrit dans [EvoCOP-02].

Nous avons également mené une étude comparative sur les problèmes de satisfaction de contraintes, où nous avons comparé les performances de la méta-heuristique ACO avec les algorithmes génétiques d'une part et la programmation par contraintes d'autre part. Nous avons utilisé à cette occasion la notion de « ratio de ré-échantillonnage » pour expliquer les différences de performance des algorithmes considérés.

Ce travail est décrit dans [EvoCOP-04].

Il a été fait en collaboration avec Jano VAN HEMERT*, chercheur au CWI à Amsterdam (Hollande).*

2.2.2 Appariements de graphes et similarité

Evaluer la similarité de deux objets est un problème qui se pose dans de nombreuses applications informatiques, un corollaire de ce problème étant de retrouver, parmi un ensemble d'objets, l'objet le plus similaire à un autre. Citons par exemple la recherche d'informations, où il s'agit de trouver des documents similaires à un document donné, la reconnaissance d'images, où l'on confronte une image à des modèles, ou encore le raisonnement à partir de cas, où il s'agit de trouver un problème déjà résolu ressemblant à un nouveau problème à résoudre dans le but d'adapter la solution du problème déjà résolu au nouveau problème à résoudre.

Lorsque les objets sont décrits par des graphes, il s'agit d'évaluer la similarité de graphes, problème qui se ramène à chercher un « meilleur » appariement des sommets des graphes, i.e., une mise en correspondance des sommets des graphes permettant de retrouver un maximum de caractéristiques communes aux deux graphes. Sur ce thème, notre contribution a porté sur la définition d'une nouvelle mesure de similarité, la conception d'algorithmes pour calculer cette mesure, ainsi que la définition d'une contrainte globale pour les problèmes d'isomorphisme de graphes.

Définition d'une nouvelle mesure de similarité. Nous avons défini un modèle de graphes orientés multi-étiquetés permettant de représenter des objets structurés, et nous avons défini le problème du calcul de la similarité entre deux graphes multi-étiquetés comme la recherche d'un meilleur appariement entre leurs sommets. L'originalité de cette mesure vient du fait qu'elle est basée sur un appariement multivoque —autorisant un sommet d'un graphe à être apparié à plusieurs sommets de l'autre graphe— ce qui permet notamment de comparer des graphes décrivant des objets à des niveaux de granularité différents.

> *Ce travail est décrit dans [ICCBR-03].*
>
> *Il a été fait en collaboration avec Pierre-Antoine* CHAMPIN *pendant sa thèse de doctorat qu'il a effectuée sous la direction d'Alain* MILLE.

Cette mesure est générique dans le sens où elle est paramétrée par deux fonctions qui permettent d'intégrer des connaissances de similarité propres à l'application considérée. Nous avons montré qu'elle est plus générale que d'autres approches souvent utilisées pour comparer et mettre en correspondance deux graphes, comme par exemple la recherche d'un isomorphisme entre deux graphes pour tester leur équivalence, ou bien la recherche d'un isomorphisme de sous-graphe pour tester l'inclusion d'un graphe dans un autre, ou encore la recherche d'un plus grand sous-graphe commun (ou le calcul de la « distance d'édition entre deux graphes ») pour mesurer l'intersection de deux graphes.

> *Ce travail est décrit dans [GbR-05] et [RJCIA-05].*
>
> *Il constitue une partie du travail de thèse de Sébastien* SORLIN.

Conception d'algorithmes pour mesurer la similarité de graphes. Pour mesurer la similarité de graphes à l'aide de notre mesure de similarité, il s'agit de trouver le « meilleur » appariement (celui qui maximise la similarité), un appariement étant un ensemble de couples de sommets. Ce problème est clairement combinatoire ($\mathcal{NP}$-difficile). Nous avons étudié et comparé différentes approches pour la résolution de ce problème :

- Une approche par « séparation et évaluation », qui effectue une exploration complète de l'ensemble des appariements possibles (structuré en treillis) et utilise des fonctions d'approximation pour réduire la combinatoire.
- Une approche gloutonne, qui construit incrémentalement un appariement en choisissant à chaque itération le « meilleur » couple de sommets à ajouter à l'appariement en cours de construction. Cette approche permet de trouver rapidement (en temps quadratique par rapport au nombre de couples de sommets) des solutions de qualité « honorable ».

  > *Ces deux premiers algorithmes sont décrits dans [JNPC-03].*
  >
  > *Ils ont été conçus en collaboration avec Pierre-Antoine* CHAMPIN *pendant sa thèse de doctorat qu'il a effectuée sous la direction de Alain* MILLE*, et ils ont été implémentés à l'occasion du stage de Maîtrise de Sébastien* SORLIN.

- Une approche par recherche locale taboue réactive qui, partant d'un appariement initial généré par l'algorithme glouton, explore l'espace des appariements par « mouvements » successifs, chaque mouvement consistant en l'ajout ou la suppression d'un couple de sommets. Afin de diversifier la recherche, on utilise une liste taboue pour mémoriser et interdire les derniers mouvements effectués. La longueur de cette liste taboue est auto-régulée en fonction du besoin de diversification de façon réactive.

  > *Ce travail est décrit dans [GbR-05] et [RJCIA-05].*
  >
  > *Il a fait l'objet du stage de DEA de Sébastien* SORLIN.

– Une approche par colonie de fourmis, où les fourmis construisent des appariements selon une stratégie gloutonne, la phéromone étant utilisée pour les guider vers les zones les plus prometteuses de l'espace de recherche. On rejoint sur ce point ma première thématique de recherche développée en 2.2.1.

Ce travail est décrit dans [EvoCOP-05].
Il a été fait en collaboration avec Khaled GHEDIRA, *Professeur à l'IEG de Tunis (Tunisie), et a fait l'objet du stage de Master d'Olfa* SAMMOUD. *Ce stage se poursuit actuellement par une thèse de doctorat qui a débuté en septembre 2005.*

Définition d'une contrainte globale pour les problèmes d'isomorphisme de graphes. L'isomorphisme de graphes permet de vérifier si deux graphes sont structurellement équivalents, et donc de tester si deux objets sont identiques (abstraction faite des informations perdues lors de la modélisation de l'objet en graphe). Ce problème peut très facilement être modélisé en un problème de satisfaction de contraintes puis être résolu par un solveur de contraintes. Toutefois, sur ce type de problèmes, la programmation par contraintes est bien moins efficace que les algorithmes dédiés qui sont capables de tirer partie de la sémantique globale du problème pour réduire la combinatoire.

Aussi avons nous introduit une nouvelle contrainte globale dédiée au problème de l'isomorphisme de graphes, permettant d'exprimer le fait que deux graphes doivent être isomorphes. Nous avons alors proposé un algorithme de filtrage associé à cette contrainte. Celui-ci exploite les arêtes du graphe de façon globale afin de réduire le domaine des variables. Nous avons aussi montré que cette contrainte globale est décomposable en un ensemble de « contraintes de distance » propageant mieux les réductions des domaines que les « contraintes d'arêtes » habituellement utilisées pour ce problème.

Ce travail est décrit dans [CPAIOR-04] et [JFPC-04].
Il constitue une partie de la thèse de Sébastien SORLIN.

2.3 Perspectives de recherche

Mes perspectives de recherche s'intègrent essentiellement dans le cadre du projet « Graph-I-Sim ». Ce projet, dont je suis porteuse, a débuté en mai 2005 et est financé par une Action Thématique et Incitative sur Programmes (ATIP) « Jeunes chercheurs » du département STIC du CNRS.

Ce projet se place dans le prolongement de mes activités de recherche de ces cinq dernières années, faisant le lien entre l'optimisation par colonies de fourmis et la mesure de similarité de graphes : il s'agit d'appliquer notre mesure de similarité de graphes au domaine de la recherche de documents, et de développer de nouveaux algorithmes ACO pour cela. On s'intéresse plus précisément à des documents représentés à l'aide de graphes RDF, et au problème de rechercher dans une base de documents les documents les plus similaires à un document requête donné. Dans ce contexte, les principaux points que nous souhaitons explorer sont décrits ci-après.

Conception d'un nouvel algorithme pour mesurer la similarité de graphes. Les deux algorithmes que nous avons proposés dans [GbR-05] et [EvoCOP-05], respectivement basés sur une approche taboue réactive et une approche par colonies de fourmis, ont des performances très complémentaires. Nous avons comparé ces deux algorithmes sur un ensemble de problèmes d'isomorphisme de sous-graphe ainsi que sur un ensemble de problèmes de mesure de similarité de graphes RDF générés aléatoirement. Ces expérimentations ont montré que certains problèmes difficilement résolus par un algorithme le sont facilement par l'autre, et inversement.

Ainsi, un premier objectif est de proposer un nouvel algorithme intégrant les deux approches. Plus précisément, il s'agira de concevoir un algorithme ACO utilisant des techniques de recherche locale pour améliorer les appariements successivement construits par les fourmis.

Intégration interactive et dynamique de connaissances de similarité. Les deux algorithmes proposés dans [GbR-05] et [EvoCOP-05] sont paramétrés par des connaissances de similarité, permettant par exemple d'associer des poids à différents critères comme la taille, la forme, la texture ou la couleur dans le cas de graphes décrivant des images. Cependant, lorsque ces paramètres sont modifiés, ces algorithmes doivent être relancés sans utiliser les résultats précédemment trouvés. Un objectif est de concevoir un algorithme capable d'intégrer dynamiquement (au cours de la résolution puisqu'il s'agit d'une approche « anytime ») des connaissances de similarité exprimées par l'utilisateur en fonction de la pertinence des premiers résultats trouvés, e.g., diminuer le poids des caractéristiques liées à la couleur si les premiers résultats retournés mettent trop l'accent sur la couleur.

Notons que la méta-heuristique ACO est particulièrement bien adaptée à la résolution de tels problèmes dynamiques : l'expérience passée accumulée par la colonie de fourmis est « compilée » sous la forme de traces de phéromones ; lorsque certaines données du problème changent, on peut modifier les traces relatives aux données modifiées tout en conservant les autres.

Il n'existe pas actuellement de système de recherche d'informations permettant d'intégrer dynamiquement des connaissances de similarité. Ce point nous semble important pour que le système puisse retourner des documents pertinents, dans la mesure où la notion de pertinence est généralement subjective et dépend des objectifs de l'utilisateur pour une requête donnée.

> *Ces deux premiers points (algorithme hybridant recherche locale et colonies de fourmis, et intégration interactive et dynamique de connaissances de similarité) constituent le point de départ de la thèse d'Olfa* SAMMOUD*, thèse qui a commencé en septembre 2005.*

Validation sur la problématique de la recherche d'images par l'exemple. Il s'agit tout d'abord de proposer une approche pour générer des graphes RDF à partir d'images. Pour identifier des régions dans les images, on se propose d'utiliser l'approche par groupements perceptuels proposée par Nicolas ZLATOFF qui effectue actuellement une thèse au LIRIS sous la direction de Bruno TELLEZ et Attila BASKURT. Ces régions constitueront les sommets du graphe RDF ; les relations topologiques entre ces régions seront exprimées par des arcs. Pour étiqueter les sommets du graphe, on pourra utiliser des informations de bas-niveau ainsi que les critères perceptuels ayant contribué à la formation des régions.

Les images étant décrites par des graphes RDF, on pourra alors utiliser notre algorithme de mesure de similarité de graphes RDF pour rechercher les images les plus similaires à une image requête. Un point original de notre mesure est qu'elle permet non seulement de quantifier les différences entre deux graphes, mais aussi d'identifier leurs points communs et différences, et donc d'expliquer à l'utilisateur les réponses trouvées par le système. Ces explications pourront être utilisées par l'utilisateur pour ajouter ou modifier des connaissances de similarité s'il juge les résultats peu pertinents.

> *Ce point constituera le dernier volet de la thèse de Sébastien* SORLIN*, dont la soutenance est prévue fin 2006. Ce travail sera fait en collaboration avec Pierre-Antoine* CHAMPIN*, Bruno* TELLEZ *et Nicolas* ZLATOFF.

2.4 Encadrements

Encadrements de thèses

Claudia ZEPEDA CORTES : *Evacuation planning using Answer Set Programming*
Date de début : Septembre 2002
Soutenance prévue : 12 Décembre 2005
Thèse en co-tutelle avec le Mexique
Publications : [LA-NMR-04, ASP-05, MICAI-05]
Co-encadrement avec Robert LAURINI, Professeur au LIRIS, David SOL MARTINEZ et Mauricio OSORIO, Professeurs à Universidad de las Américas-Puebla au Mexique

Sébastien SORLIN : *Mesurer la similarité de graphes*
Date de début : Septembre 2003
Soutenance prévue : 2006
Financement : allocation MENRT
Publications : [JNPC-03, CPAIOR-04, JFPLC-04, GbR05, RJCIA-05]
Encadrement à 100%

Inès ALAYA : *Optimisation par colonies de fourmis pour les problèmes de recherche de sous-ensembles*
Date de début : Janvier 2005
Soutenance prévue : 2008
Financement : Tunisien
Publications : [Francoro-04, BIOMA-04]
Co-encadrement avec Khaled GHÉDIRA, Professeur à l'IEG de Tunis

Olfa SAMMOUD : *Optimisation par colonies de fourmis pour l'appariement de graphes*
Date de début : Septembre 2005
Soutenance prévue : 2008
Financement : Tunisien
Publications : [EvoCOP-05]
Co-encadrement avec Khaled GHÉDIRA, Professeur à l'IEG de Tunis

Encadrements de stages de DEA et de Master

Dinh-Toan N'GUYEN : *Analyse et conception d'une bibliothèque pour l'optimisation par colonies de fourmis*
Mars 2005 à Septembre 2005
Master Recherche de l'Université Lyon 1
Co-encadrement avec Patrick ALBERT (Ilog, Gentilly)

Olfa SAMMOUD : *Mesurer la similarité de graphes avec des colonies de fourmis*
Mars 2003 à Mars 2004
Master Recherche de l'ISG de Tunis
Co-encadrement avec Khaled GHEDIRA, Professeur à l'ISG de Tunis

Inès ALAYA : *Application de la méta-heuristique ACO au problème du sac-à-dos multidimensionnel*
Mars 2003 à Mars 2004
Master Recherche de l'ISG de Tunis
Co-encadrement avec Khaled GHEDIRA, Professeur à l'ISG de Tunis

Sébastien SORLIN : *Appariement interactif de graphes typés*
Février à Juillet 2003
DEA DISIC de l'Université Lyon 1

Alain-Pierre Manine : *Dopage des algorithmes de fourmis par extraction de connaissances*
Février à Juillet 2002
DEA DISIC de l'Université Lyon 1

Arnaud Stuber : *Dopage des algorithmes à base de fourmis par des techniques de fouille de données*
Février à Juillet 2001
DEA ECD de l'Université Lyon 1
Lauréat du prix du meilleur DEA de l'association des « Amis de l'Université de Lyon » dans la catégorie « sciences de la matière et de l'information »

Jean Duong : *Une approche collective pour la résolution des problèmes de satisfaction de contraintes*
Février à Juillet 1999
DEA DISIC de l'Université Lyon 1
Co-encadrement avec Simone Pimont, Maître de Conférences au LIRIS

2.5 Animation de la recherche

Collaborations internationales

- Derek Bridge, Professeur à Cork, Irlande : depuis 2005 ; un chapitre de livre co-signé.
- Khaled Ghédira, Professeur au laboratoire SOIE, Tunis, Tunisie : depuis 2003 ; co-encadrement de deux stages de Master, qui se poursuivent maintenant par des thèses de doctorat ; trois publications co-signées.
- Jens Gottlieb et Markus Puchta, Chercheurs à SAP, Walldorf, Allemagne : en 2002/2003 ; une publication co-signée.
- Jano van Hemert, Chercheur au CWI, Amsterdam, Hollande : en 2003/2004 ; une publication co-signée.
- David Sol et Mauricio Osorio, Professeurs à Universidad de las Américas-Puebla, Puebla, Mexique : depuis 2002 ; co-encadrement d'une thèse ; trois publications co-signées.

Collaborations nationales

- Patrick Albert, Chercheur à Ilog, Gentilly : depuis 2004 ; co-encadrement d'un stage de Master Recherche ; dépot d'un dossier pour un projet RNTL commun en Juin 2005.
- Pierre-Antoine Champin, Maître de Conférences au LIRIS : depuis 2002 ; deux publications co-signées ; membre du projet Graph-I-Sim
- Serge Fenet, Maître de Conférences au LIRIS : en 2002/2003 ; trois publications co-signées
- Bruno Tellez, Maître de Conférences au LIRIS : depuis 2005 ; membre du projet Graph-I-Sim

Groupes de travail et Associations

- AFPC (Association Française pour la Programmation par Contraintes ; anciennement AFPLC)
  Membre élue au conseil d'administration depuis 1999
  Membre du bureau (déléguée aux manifestations) depuis 2000
- Membre du réseau d'excellence européen « EvoNet » depuis 2001
- Participations aux groupes de travail « Contraintes / RO » (ASRO du CNRS) et « Indexation et Recherche d'Informations » (GdR-PRC I3 / GT 3.5)

Comités de programme de conférences et édition de numéros spéciaux de revues

- Présidente du comité de programme des JFPC 2005
  (Journées Francophones de Programmation par Contraintes)
- Membre du comité de programme des JFPC 2006
  (Journées Francophones de Programmation par Contraintes)
- Membre du comité de programme des JFPLC en 2000, 2002, 2003 et 2004
  (Journées Francophones de Programmation Logique et par Contraintes)
- Membre du comité de programme de EvoCOP en 2003, 2004, 2005 et 2006
  (European Conference on Evolutionary Computation in Combinatorial Optimization)
- Membre du comité de programme de GECCO en 2005 et 2006
  (Genetic and Evolutionary Computation Conference)
- Membre du comité de programme de ANTS en 2006
  (International Workshop on Ant Colony Optimization and Swarm Intelligence)
- Membre du comité de programme de EvoPhD 2006
  (First European Graduate Student Workshop on Evolutionary Computation)
- Co-éditeur, avec Narendra JUSSIEN et François FAGES, d'un numéro spécial de la revue RAIRO Operations Research sur le thème de « la recherche opérationnelle et les contraintes », à paraître en 2006

Comités d'organisation de conférences

- ECAI 2002 (European Conference on Artificial Intelligence) : chargée de la communication
- RJCIA 2000 (Rencontres des Jeunes Chercheurs en Intelligence Artificielle) : chargée des subventions
- JFPLC 1999 (Journées Francophones de Programmation Logique et par Contraintes) : chargée de l'organisation de la journée industrielle

Organisation de séminaires

- Responsable des séminaires du laboratoire LISI de 1996 à 2001
- Responsable des séminaires du DEA ECD (Extraction de Connaissances à partir des Données) de 2000 à 2002

2.6 Publications

Revues internationales

[Heuristics-05] **Christine Solnon** et Serge Fenet : *A study of ACO capabilities for solving the maximum clique problem*, à paraître dans le « journal of heuristics » (accepté pour publication en Aout 2005), Springer

[TEC-02] **Christine Solnon** : *Ants Can Solve Constraint Satisfaction Problems*, dans « IEEE Transactions on Evolutionary Computation », Vol.6, N°4, pages 347-357, Août 2002

[RC-97] Michel Rueher et **Christine Solnon** : *Concurrent Cooperating Solvers over Reals*, dans « International Journal of Reliable Computing », Kluwer Academic Publishers, Vol.3, N°3, pages 325-333, Octobre 1997

Chapitre de livre

[Particle-Swarm-Optimization-06] **Christine Solnon** et Derek Bridge : *An Ant Colony Optimization Meta-Heuristic for Subset Selection Problems*, chapitre du livre « System Engineering using Particle Swarm

Optimization », édité par Nadia Nedjah et Luiza Mourelle et publié par Nova Science, New York, à paraître en 2006

Conférences et workshops internationaux avec comités de programme

[MICAI-05] Claudia Zepeda, Mauricio Osorio, **Christine Solnon** et David Sol : *An application of preferences using Answer Set Programing : Evacuation planning*, à paraître dans la session posters de « 4th Mexican International Conference on Artificial Intelligence (MICAI 2005) », journal « Research on Computing Science », 10 pages, Monterrey, Mexique, Novembre 2005

[GbR-05] Sébastien Sorlin et **Christine Solnon** : *Reactive Tabu Search for Measuring Graph Similarity*, dans les actes de « 5th IAPR Workshop on Graph-based Representations in Pattern Recognition (GbR 2005) », pages 172-182, LNCS N°3434, Springer, Poitiers, France, Avril 2005

[EvoCOP-05] Olfa Sammoud, **Christine Solnon** et Khaled Ghédira : *Ant Algorithm for the Graph Matching Problem*, dans les actes de « 5th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2005) », pages 213-223, LNCS N°3448, Springer, Lausanne, Suisse, Avril 2005

[ASP-05] Claudia Zepeda, Mauricio Osorio, Juan Carlos Nieves, **Christine Solnon** et David Sol : *Applications of preferences using Answer Set Programming*, dans les actes de « Answer Set Programming : Advances in Theory and Implementation (ASP 2005) », Bath, Grande Bretagne, Juillet 2005

[Bioma-04] Inès Alaya, **Christine Solnon** et Khaled Ghédira : *Ant algorithm for the multi-dimensional knapsack problem*, dans les actes de « International Conference on Bioinspired Optimization Methods and their Applications » (BIOMA 2004), pages 63-72, Ljubljana, Slovénie, Octobre 2004

[LA-NMR-04] Claudia Zepeda, **Christine Solnon** et David Sol : *Planning Operation : An Extension of a Geographical Information System*, dans les actes de « Latin-American Workshop on Non-Monotonic Reasoning (LA-NMR 2004) », Vol. 92 of CEUR workshop proceedings edited by M. Osorio and A. Provetti, Mexico, Mexique, Avril 2004

[CPAIOR-04] Sébastien Sorlin et **Christine Solnon** : *A global constraint for graph isomorphism problems*, dans les actes de « 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004) », pages 287-301, LNCS N°3011, Springer, Nice, France, Avril 2004

[EvoCOP-04] Jano van Hemert et **Christine Solnon** : *A Study into Ant Colony Optimization, Evolutionary Computation and Constraint Programming on Binary Constraint Satisfaction Problems*, dans les actes de « 4th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2004) », pages 114-123, LNCS N°3004, Springer, Coimbra, Portugal, Avril 2004

[ICCBR-03] Pierre-Antoine Champin et **Christine Solnon** : *Measuring the similarity of labeled graphs*, dans « 5th International Conference on Case-Based Reasoning (ICCBR 2003) », pages 80-95, LNAI N°2689, Springer, Trondheim, Norvège, Juillet 2003

[EvoCOP-03-a] Jens Gottlieb, Markus Puchta et **Christine Solnon** : *A Study of Greedy, Local Search and ACO Approaches for Car Sequencing Problems*, dans « Applications of Evolutionary Computing (EvoCOP 2003) », pages 246-257, LNCS N°2611, Springer, Essex, Grande Bretagne, Avril 2003

[EvoCOP-03-b] Serge Fenet et **Christine Solnon** : *Searching for Maximum Cliques with Ant Colony Optimization*, dans « Applications of Evolutionary Computing (EvoCOP 2003) », pages 236-245, LNCS N°2611, Springer, Essex, Grande Bretagne, Avril 2003

[EvoCOP-02] **Christine Solnon** : *Boosting ACO with a Preprocessing step*, dans « Applications of Evolutionary Computing (EvoCOP 2002) », pages 163-172, LNCS N°2279, Springer, Kinsale, Irlande, Avril 2002

[CP-01] **Christine Solnon** : *Boosting Local Search with Artificial Ants*, dans « 7th International Conference on Principles and Practice of Constraint Programming (CP 2001) », article court, pages 620-624, LNCS N°2239, Springer, Paphos, Chypre, Novembre 2001

[ECAI-00] **Christine Solnon** : *Solving Permutation Constraint Satisfaction Problems with Artificial Ants*, dans « 14th European Conference on Artificial Intelligence (ECAI 2000) », pages 118-122, IOS Press, Berlin, Allemagne, Aout 2000

[ANTS-00] Simone Pimont et **Christine Solnon** : *A Generic Ant Algorithm for Solving Constraint Satisfaction Problems*, dans « 2nd International Workshop on Ant Algorithms (ANTS 2000) », pages 100-108, Bruxelles, Belgique, Septembre 2000

[ICTAI-97] **Christine Solnon** : *Cooperation of LP Solvers for Solving MILPs*, dans les actes de « International Conference on Tools for Artificial Intelligence (ICTAI 97) », pages 240-247, IEEE Computer Society Press, Newport Beach, Etats Unis d'Amérique, Novembre 1997

[Interval-96] Michel Rueher et **Christine Solnon** : *Concurrent Cooperating Solvers over Reals* , dans les actes de « International Conference on Interval Methods and Computer Aided Proofs (Interval 1996) », pages 97-98, Wuerzburg, Allemagne, Septembre 1996

[LOPSTR-94] **Christine Solnon** et Michel Rueher : *Propagation of inter-argument dependencies in tuple-distributive type inference systems*, dans les actes de « International Conference on Logic Program Synthesis and Transformation (LOPSTR 1994) », pages 199-214, LNCS N°883, Springer, Pise, Italie, Septembre 1994

[LPAR-93] **Christine Solnon** et Michel Rueher : *Extracting inheritance hierarchies from Prolog programs : a system based on the inference of type relations*, dans les actes de « International Conference on Logic Programming and Automated Reasoning (LPAR 1993) », pages 309-320, LNCS N°698, Springer, Saint Petersbourg, Russie, Juillet 1993

[PLILP-92] **Christine Solnon** et Michel Rueher : *Inference of inheritance relationships from Prolog programs : a system developped with PrologIII*, dans les actes de « International Symposium on Programming Languages and Logic Programming (PLILP 1992) », article court, pages 489-490, LNCS N°631, Springer, Leuven, Belgique, Aout 1992

[LOPSTR-91] **Christine Solnon** et Michel Rueher : *Using a Prolog prototype for designing an object-oriented scheme*, dans les actes de « International Workshop on Logic Program Synthesis and Transformation (LOPSTR 1991) », pages 300-317, Springer, Manchester, Grande Bretagne, Juin 1991

Exposé invité dans un workshop international avec actes

[TeachLP-04] **Christine Solnon** : *An on-line course on constraint programming*, dans « first International Workshop on Teaching Logic Programming (TeachLP 2004) », Linköping Electronic Conference Proceedings, Issue No. 12, pages 11-17, Saint Malo, France, Septembre 2004

Conférences nationales avec comités de programme

[RJCIA-05] Sébastien Sorlin et **Christine Solnon** : *Similarité de graphes : une mesure générique et un algorithme tabou réactif*, dans les actes des « 7ème Rencontres Jeunes Chercheurs en intelligence articifielle (RJCIA 2005) », pages 253-266, Presses Universitaires de Grenoble, Nice, Juin 2005

[JFPLC-04] Sébastien Sorlin et **Christine Solnon** : *Une contrainte globale pour le problème d'isomorphisme de graphes*, dans les actes des « 13èmes Journées Francophones de Programmation en Logique avec Contraintes (JFPLC 2004) », Editions Hermes, pages 91-107, Angers, Juin 2004

[Francoro-04] Inès Alaya, **Christine Solnon** et Khaled Ghédira : *Des fourmis pour le sac-à-dos multidimensionnel*, dans les actes de « 4èmes journées francophones de recherche opérationnelle (Francoro 2004 », pages 25-26, Lausanne, Suisse, Aout 2004

[JFPLC-03] **Christine Solnon** : *Un cours « électronique » sur la programmation par contraintes*, dans « Programmation en logique avec contraintes (JFPLC 2003) », article court, Numéro spécial de la Revue des Sciences et Technologies de l'Information, Editions Hermès, pages 77-80, Amiens, Juin 2003

[JNPC-03] Sébastien Sorlin, Pierre-Antoine Champin et **Christine Solnon** : *Mesurer la similarité de graphes étiquetés*, dans « 9èmes Journées Nationales sur la résolution Pratique de problèmes NP-Complets (JNPC 2003) », pages 325-339, Amiens, Juin 2003

[ROADEF-03] Serge Fenet et **Christine Solnon** : *Recherche de cliques maximum avec la méta-heuristique ACO*, dans « 5ème congrès de la société française de recherche opérationnelle et d'aide à la décision (ROADEF 2003) », pages 159-160, Avignon, Février 2003

[JNPC-01] Arnaud Stuber et **Christine Solnon** : *Dopage des algorithmes à base de fourmis par des techniques de fouille de données*, dans « 7èmes Journées Nationales sur la résolution Pratique de problèmes NP-Complets (JNPC 2001) », pages 271-282, Toulouse, Juin 2001

[JFPLC-00] **Christine Solnon** : *Ant-P-solveur : un solveur de contraintes à base de fourmis artificielles*, dans « 9èmes Journées Francophones sur la Programmation Logique et par Contraintes (JFPLC 2000) », Editions Hermès, pages 189-204, Marseille, Juin 2000

[JFPLC-97] **Christine Solnon** : *Coopération de solveurs linéaires sur les réels pour la résolution de problèmes linéaires sur les entiers*, dans « 6èmes Journées Francophones sur la Programmation Logique et par Contraintes (JFPLC 1997) », Editions Hermès, pages 211-225, Orléans, Juin 2000

[JFPL-94] **Christine Solnon** et Michel Rueher : *Inférence de contraintes ensemblistes à partir de programmes Prolog — application à la déduction de relations d'héritage —*, dans les actes de « 3èmes Journées Francophones sur la Programmation Logique (JFPC 1994) », Bordeaux, Juin 1994

[JFPL-92] **Christine Solnon** et Michel Rueher : *Du prototype Prolog vers un modèle objet : une approche basée sur l'inférence des relations inter-types*, dans les actes de « 1ères Journées Francophones sur la Programmation Logique (JFPC 1992) », Lille, Juin 1992

Chapitre 3

Activités d'enseignement et d'administration

3.1 Enseignements effectués au département informatique de l'IUT A

Le tableau ci-dessous récapitule les enseignements, en heures équivalent TD, que j'ai effectués au département informatique de l'IUT A depuis mon arrivée en septembre 1994. On notera des services réduits en 94/95, 97/98 et 98/99 du fait de la naissance de mon deuxième enfant en mai 1995, de la naissance de mon troisième enfant en mars 1998, et d'une année à temps partiel (à 80%) en 98/99.

Matière	94/95	95/96	96/97	97/98	98/99	99/00	00/01	01/02	02/03	03/04	04/05
Analyse et conception des systèmes d'information	60 h	60 h	60 h								
Ateliers de génie logiciel	64 h	64 h	32 h								
Prolog	40 h	40 h	40 h		20 h	20 h	20 h	20 h			
Intelligence artificielle	20 h	20 h	20 h		20 h	20 h					
Théorie des langages		48 h	48 h	48 h	48 h	48 h	48 h	48 h	24 h	24 h	
Algorithmique de base				68 h	68 h	68 h	60 h	60 h	60 h	60 h	60 h
Théorie des graphes						36 h	36 h	32 h	32 h	32 h	32 h
Structures de données							40 h	40 h	80 h	36 h	36 h
Programmation système										44 h	44 h
Système Unix											20 h
Total	184 h	232 h	200 h	116 h	156 h	192 h	204 h	200 h	196 h	196 h	192 h

Les enseignements au département informatique de l'IUT A sont organisés par groupes de 25 à 30 étudiants ; l'enseignant qui a la responsabilité d'un groupe pour une matière assure la totalité des cours, travaux dirigés et travaux pratiques pour ce groupe. Le contenu et l'organisation des différents cours que j'ai assurés depuis 1994 sont donnés ci dessous.

Analyse et Conception des Systèmes d'Information

- Volume horaire et organisation :
  - de 94 à 97 : 1 groupe de 1ère année - Cours et TD - 60 heures
- Contenu :
  - Introduction aux systèmes d'information
  - Etude du modèle de données Relationnel
  - Etude du modèle de données « Entités - Associations »

Ateliers de Génie Logiciel
- Volume horaire et organisation :
  - de 94 à 96 : 2 groupes de 2ème année - Cours et TP - 2*32 heures
  - de 96 à 97 : 1 groupe de 2ème année - Cours et TP - 32 heures
- Contenu :
  - Présentation générale du génie logiciel et des ateliers de génie logiciel
  - Présentation et utilisation de deux AGLs : TRAMIS et WinDev
  - Application sur un projet
- Support de cours en ligne : `http ://www710.univ-lyon1.fr/~csolnon/agl.html`

Prolog
- Volume horaire et organisation :
  - de 94 à 97 : 2 groupes de 2ème année - Cours, TD et TP - 2*20 heures
  - de 98 à 02 : 1 groupe de 2ème année - Cours, TD et TP - 20 heures
- Contenu :
  - Introduction à la logique et aux mécanismes de résolution
  - Présentation de Prolog (syntaxe et sémantique)
  - Application sur 2 TP
- Support de cours en ligne : `http ://www710.univ-lyon1.fr/~csolnon/prolog.html`

Intelligence Artificielle
- Volume horaire et organisation :
  - de 94 à 97 et de 98 à 00 : 1 groupe de 2ème année - Cours et TP - 20 heures (module optionnel)
- Contenu :
  - Présentation générale de l'intelligence artificielle
  - Problèmes de planification : présentation et techniques de résolution
  - Problèmes de satisfaction de contraintes : présentation et techniques de résolution
  - Application sur un projet
- Support de cours en ligne : `http ://www710.univ-lyon1.fr/~csolnon/contraintes.html`

Théorie des langages
- Volume horaire et organisation :
  - de 95 à 02 : 2 groupes de 2ème année - Cours et TD - 2*24 heures
  - de 02 à 04 : 1 groupe de 2ème année - Cours et TD - 24 heures
- Contenu :
  - Introduction aux notions de langages, grammaires et automates
  - Etude des grammaires régulières et automates finis
  - Etude des grammaires hors-contexte et automates à pile
- Support de cours : `http ://www710.univ-lyon1.fr/~csolnon/langages.pdf`

Algorithmique de base
- Volume horaire et organisation :
  - de 97 à 05 : 1 groupe de 1ère année - Cours et TD - 60 heures
- Contenu :
  - Introduction à la notion d'algorithme
  - Présentation des différents types d'enchainements algorithmiques
  - Etude des principaux algorithmes sur les tableaux et les fichiers séquentiels

Théorie des graphes
- Volume horaire et organisation :
  - de 99 à 05 : 1 groupe de 2ème année - Cours et TD - 32 heures
- Contenu :
  - Introduction à la théorie des graphes : définitions, cheminements et connexités, arbres et arborescences, graphes planaires
  - Algorithmes pour les graphes : parcours de graphes, plus courts chemins, arbres couvrants minimaux, réseaux de transport, planification de projets par les réseaux
- Support de cours : `http ://www710.univ-lyon1.fr/~csolnon/graphes.ps`

Structures de données
- Volume horaire et organisation :
  - de 00 à 02 : 1 groupe de 1ère année - Cours et TD - 40 heures
  - de 02 à 03 : 2 groupes de 1ère année - Cours et TD - 2*40 heures
  - de 03 à 05 : 1 groupe de 1ère année - Cours et TD - 36 heures
- Contenu :
  - Introduction aux notions de type abstrait de données et de structuration orientée objets
  - Notions de chaînage et d'allocation dynamique de mémoire
  - Etude des piles, files, arbres binaires de recherche, tables de hachage et graphes.

Programmation Système
- Volume horaire et organisation :
  - de 03 à 05 : 1 groupe de 2ème année - Cours et TP - 44 heures
- Contenu :
  - Notion de processus ; primitives systèmes pour la gestion des processus
  - Communication inter-processus : signaux, tubes, segments de mémoire partagée, sémaphores
  - Les processus légers (threads)
  - Communication inter-processus entre machines distantes (sockets)
  - Mise-en-œuvre sur 4 TP

Système Unix
- Volume horaire et organisation :
  - de 04 à 05 : 1 groupe de 2ème année - Cours et TP - 20 heures
- Contenu :
  - Présentation générale du système d'exploitation Unix
  - Le système de gestion de fichiers d'Unix
  - Etude du langage de script Shell
  - Mise-en-œuvre sur 2 TP

3.2 Autres enseignements

Cours pour la e-miage. La e-miage (miage électronique) est une formation à distance : les étudiants inscrits à cette formation la suivent depuis leur domicile, via un ordinateur personnel connecté à Internet. Ils sont par ailleurs suivis par un tuteur qui répond à leurs questions et corrige leurs exercices. J'ai conçu pour cette formation, en 2003, un cours sur la « programmation par contraintes ». Ce cours est composé des sept sessions suivantes, chaque session étant prévue pour durer une heure.

Session 1 (cours) : notion de problème de satisfaction de contraintes (CSP) ; illustration sur deux exemples

Session 2 (exercices) : formulation de cinq problèmes différents sous la forme de CSP

Session 3 (cours) : algorithmes pour la résolution de CSP

Session 4 (travaux pratiques) : implémentation en Prolog des algorithmes pour la résolution de CSP

Session 5 (cours) : présentation du langage de programmation par contraintes Gnu-Prolog

Sessions 6 et 7 (travaux pratiques) : résolution à l'aide de Gnu-Prolog des problèmes de la session 2

Ce cours peut être consulté à l'adresse
`http ://www710.univ-lyon1.fr/∼csolnon/PPC.html`

Ce cours est également décrit dans deux articles de recherche : [JFPLC-04] et [TeachLP-04]

Cours en Master Recherche. J'ai fait en 2004/2005 et 2005/2006 un cours de 20 heures sur *la résolution de problèmes combinatoires* pour le Master Recherche en Informatique de Lyon. Le plan de ce cours est le suivant :

- Introduction à la notion de complexité (complexité algorithmique et complexité systémique) et présentation de problèmes combinatoires
- Résolution par des approches complètes : structuration de l'espace de recherche en arbre et en treillis
- Résolution par des approches incomplètes : algorithmes gloutons, recherche locale, méta-heuristiques ; notions de paysage de recherche et dualité intensification/diversification
- Optimisation par colonies de fourmis : principes généraux de la méta-heuristique ACO et applications.

Les transparents de ce cours sont disponibles sur
`http ://mastria.insa-lyon.fr/CR/Fiche-Web-Solnon.html`.

3.3 Responsabilités pédagogiques

- depuis 1998 : responsable pédagogique des enseignements d'algorithmique et programmation au département informatique de l'IUT A
- de 1998 à 2004 : responsable pédagogique des enseignements d'architecture des ordinateurs, réseaux et systèmes d'exploitation au département informatique de l'IUT A
- Membre du conseil du département Informatique de l'IUT A depuis 1994
- Encadrements de stages : depuis mon arrivée au département Informatique de l'IUT A, j'ai participé au suivi des stages en entreprise des étudiants en fin de deuxième année, en encadrant de l'ordre de 5 stages par an.

3.4 Commissions de spécialistes

- Membre suppléante élue à la commission de spécialistes de la 27^{eme} section de l'Université Lyon I de Mars 1998 à Mars 2005
- Membre titulaire nommée à la commission de spécialistes de la 27^{eme} section de l'Université de Clermont-Ferrand I de Mars 1998 à Mars 2005

Deuxième partie

Contributions à la résolution pratique de problèmes combinatoires —des fourmis et des graphes—

Chapitre 4

Introduction

On présente dans ce chapitre le contexte général dans lequel se situent nos travaux. On définit tout d'abord en 4.1 ce que l'on entend par « problème combinatoire », et on donne quelque exemples de ces problèmes en 4.2. La section 4.3 fait ensuite un panorama des principales approches pour résoudre en pratique ces problèmes. Enfin, la section 4.4 situe par rapport à ce contexte les travaux présentés dans les deux chapitres suivants.

4.1 Quelques caractéristiques des problèmes combinatoires

On qualifie généralement de « combinatoires » les problèmes dont la résolution se heurte à une explosion du nombre de combinaisons à explorer. C'est le cas par exemple lorsque l'on cherche à concevoir un emploi du temps : s'il y a peu de cours à planifier, le nombre de combinaisons à explorer est faible et le problème sera très rapidement résolu ; cependant, l'ajout de quelques cours seulement peut augmenter considérablement le nombre de combinaisons à explorer de sorte que le temps de résolution devient excessivement long.

Cette notion de problème combinatoire est formellement caractérisée par la théorie de la complexité qui propose une classification des problèmes en fonction de la complexité de leur résolution, et nous introduisons tout d'abord les classes de problèmes qui nous intéressent plus particulièrement. Nous introduisons ensuite les notions de transition de phase et de paysage de recherche qui permettent de caractériser la difficulté « en pratique » des différentes instances d'un même problème combinatoire.

4.1.1 Complexité théorique d'un problème

On entend ici par « complexité d'un problème » une estimation du nombre d'instructions à exécuter pour résoudre les instances de ce problème, cette estimation étant un ordre de grandeur par rapport à la taille de l'instance. Il s'agit là d'une estimation dans le pire des cas dans le sens où la complexité d'un problème est définie en considérant son instance la plus difficile. Les travaux théoriques dans ce domaine ont permis d'identifier différentes classes de problèmes en fonction de la complexité de leur résolution [Pap94]. Il existe en fait un très grand nombre de classes différentes[1], et on se limitera ici à une présentation succincte nous permettant de caractériser formellement la notion de problème combinatoire.

[1] Le « zoo des complexité », que l'on peut consulter sur `http ://www.complexityzoo.com/`, recense pas moins de 442 classes de complexités différentes.

Problèmes de décision. Les classes de complexité ont été introduites pour les problèmes de décision, c'est-à-dire les problèmes posant une question dont la réponse est « oui » ou « non ». Pour ces problèmes, on définit notamment les deux classes $\mathcal{P}$ et $\mathcal{NP}$:

– La classe $\mathcal{P}$ contient l'ensemble des problèmes polynomiaux, i.e., pouvant être résolus par un algorithme de complexité polynomiale. Cette classe caractérise l'ensemble des problèmes que l'on peut résoudre « efficacement ».
– la classe $\mathcal{NP}$ contient l'ensemble des problèmes polynomiaux non déterministes, i.e., pouvant être résolus par un algorithme de complexité polynomiale pour une machine non déterministe (que l'on peut voir comme une machine capable d'exécuter en parallèle un nombre fini d'alternatives). Intuitivement, cela signifie que la résolution des problèmes de $\mathcal{NP}$ peut nécessiter l'examen d'un grand nombre (éventuellement exponentiel) de cas, mais que l'examen de chaque cas doit pouvoir être fait en temps polynomial.

Les relations d'inclusion entre les classes $\mathcal{P}$ et $\mathcal{NP}$ sont à l'origine d'une très célèbre conjecture que l'on peut résumer par « $\mathcal{P} \neq \mathcal{NP}$ ». En effet, si la classe $\mathcal{P}$ est manifestement inclue dans la classe $\mathcal{NP}$, la relation inverse n'a jamais été ni montrée ni infirmée.

Cependant, certains problèmes de $\mathcal{NP}$ apparaissent plus difficiles à résoudre dans le sens où l'on ne trouve pas d'algorithme polynomial pour les résoudre avec une machine déterministe. Les problèmes les plus difficiles de $\mathcal{NP}$ définissent la classe des *problèmes $\mathcal{NP}$-complets* : un problème de $\mathcal{NP}$ est $\mathcal{NP}$-complet s'il est au moins aussi difficile à résoudre que n'importe quel autre problème de $\mathcal{NP}$, i.e., si n'importe quel autre problème de $\mathcal{NP}$ peut être transformé en ce problème par une procédure polynomiale.

Cette équivalence par transformation polynomiale entre problèmes $\mathcal{NP}$-complets implique une propriété fort intéressante : si l'on trouvait un jour un algorithme polynomial pour un de ces problèmes (n'importe lequel) on pourrait en déduire des algorithmes polynomiaux pour tous les autres problèmes, et on pourrait alors conclure que $\mathcal{P} = \mathcal{NP}$. La question de savoir si un tel algorithme existe a été posée en 1971 par Stephen Cook... et n'a toujours pas reçu de réponse.

Ainsi, les problèmes $\mathcal{NP}$-complets sont des problèmes combinatoires dans le sens où leur résolution implique l'examen d'un nombre exponentiel de cas. Notons que cette classe des problèmes $\mathcal{NP}$-complets contient un très grand nombre de problèmes (dont quelques-uns sont décrits dans la section 4.2). Pour autant, tous les problèmes combinatoires n'appartiennent pas à cette classe. En effet, pour qu'un problème soit $\mathcal{NP}$-complet, il faut qu'il soit dans la classe $\mathcal{NP}$, i.e., que l'examen de chaque cas puisse être réalisé efficacement, par une procédure polynomiale. Si on enlève cette contrainte d'appartenance à la classe $\mathcal{NP}$, on obtient la classe plus générale des *problèmes $\mathcal{NP}$-difficiles*, contenant l'ensemble des problèmes qui sont « au moins aussi difficiles » que n'importe quel problème de $\mathcal{NP}$, sans nécessairement appartenir à $\mathcal{NP}$.

Problèmes d'optimisation. Le but d'un problème d'optimisation est de trouver une solution maximisant (resp. minimisant) une fonction objectif donnée. A chaque problème d'optimisation on peut associer un problème de décision dont le but est de déterminer s'il existe une solution pour laquelle la fonction objectif soit supérieure (resp. inférieure) ou égale à une valeur donnée. La complexité d'un problème d'optimisation est liée à celle du problème de décision qui lui est associé. En particulier, si le problème de décision est $\mathcal{NP}$-complet, alors le problème d'optimisation est dit $\mathcal{NP}$-difficile.

4.1.2 ... et en pratique ?

La théorie de la complexité nous dit que si un problème est $\mathcal{NP}$-difficile, alors il est illusoire de chercher un algorithme polynomial pour ce problème (à moins que $\mathcal{P} = \mathcal{NP}$). Toutefois, ces travaux sont basés sur une évaluation de la complexité dans le pire des cas, car la difficulté d'un problème est définie par rapport à son instance la plus difficile. En pratique, si on sait qu'on ne pourra pas résoudre en temps polynomial *toutes* les instances d'un problème $\mathcal{NP}$-difficile, il apparait que certaines instances sont beaucoup plus faciles que d'autres et peuvent être résolues très rapidement.

Considérons par exemple un problème de conception d'emploi du temps, consistant à affecter des créneaux horaires à des cours en respectant des contraintes d'exclusion —exprimant par exemple le fait que certains cours ne doivent pas avoir lieu en même temps. Il s'agit là d'un problème $\mathcal{NP}$-complet classique se ramenant à un problème de coloriage de graphes. Pour autant, la résolution pratique de ce problème peut s'avérer plus ou moins difficile d'une instance à l'autre, même si l'on considère des instances de même taille, i.e., ayant toutes le même nombre de cours et de créneaux horaires. Typiquement, les instances ayant très peu de contraintes d'exclusion sont faciles à résoudre car elles admettent beaucoup de solutions ; les instances ayant beaucoup de contraintes sont aussi facilement traitables car on arrive vite à montrer qu'elles n'admettent pas de solution ; les instances intermédiaires —ayant trop de contraintes pour que l'on trouve facilement une solution, mais pas assez pour que l'on montre facilement qu'elles n'ont pas de solution— sont généralement beaucoup plus difficiles à résoudre.

De façon plus générale, la difficulté des instances d'un problème de décision apparait souvent liée à un phénomène de « transition de phases » que nous introduisons dans le paragraphe suivant. Nous introduisons ensuite la notion de « paysage de recherche » qui permet de caractériser la difficulté des instances de problèmes d'optimisation.

Notion de transition de phases. De nombreux problèmes de décision $\mathcal{NP}$-complets peuvent être caractérisés par un paramètre de contrôle de telle sorte que l'espace des instances du problème comporte deux grandes régions : la région sous-contrainte où quasiment toutes les instances admettent un grand nombre de solutions, et la région sur-contrainte où quasiment toutes les instances sont insolubles. Très souvent, la transition entre ces deux régions est brusque dans le sens où une très petite variation du paramètre de contrôle sépare les deux régions. Ce phénomène est appelé « transition de phases » par analogie avec la transition de phases en physique qui désigne un changement des propriétés d'un système (e.g., la fusion) provoquée par la variation d'un paramètre extérieur particulier (e.g., la température) lorsqu'il atteint une valeur seuil.

Une caractéristique fort intéressante de ce phénomène est que les instances les plus difficiles à résoudre se trouvent dans la zone de transition [CKT91, Hog96, MPSW98], là où les instances ne sont ni trivialement solubles ni trivialement insolubles. Il s'agit là d'un pic de difficulté dans le sens où une toute petite variation du paramètre de contrôle permet de passer d'instances vraiment très faciles à résoudre à des instances vraiment très difficiles. Notons également que ce pic de difficulté est indépendant du type d'approche de résolution considérée [Dav95, CFG+96].

De nombreux travaux, à la fois théoriques et expérimentaux, se sont intéressés à la localisation de ce pic par rapport au paramètre de contrôle. En particulier, [GMPW96] introduit la notion de « degré de contrainte » (« constrainedness ») d'une classe de problèmes, noté κ, et montre que lorsque κ est proche de 1, les instances sont critiques et appartiennent à la région autour de la transition de phase. Ces travaux sont particulièrement utiles pour la communauté de chercheurs s'intéressant à la résolution pratique de problèmes combinatoires. Ils permettent en particulier de se concentrer sur les instances vraiment difficiles, qui sont utilisées en pratique pour valider et comparer les approches de résolution. Ces connaissances peuvent également être utilisées comme des heuristiques pour résoudre plus efficacement les problèmes $\mathcal{NP}$-complets [Hog98].

Notons cependant que ces travaux se basent généralement sur l'hypothèse d'une génération aléatoire et uniforme des instances par rapport au paramètre de contrôle, de telle sorte que les contraintes sont relativement uniformément réparties sur l'ensemble des variables du problème. Cette hypothèse n'est pas toujours réaliste et les problèmes réels sont souvent plus structurés, exhibant des concentrations irrégulières de contraintes.

Notion de paysage de recherche. La notion de transition de phases n'a de sens que pour les problèmes de décision binaires, pour lesquels on peut partitionner les instances en deux sous-ensembles en fonction de leur réponse. Pour les problèmes d'optimisation, on peut caractériser la difficulté d'une instance d'un problème d'optimisation par différents indicateurs.

Considérons par exemple une instance d'un problème d'optimisation définie par le couple (S, f) tel que S est l'ensemble des configurations candidates et $f : S \rightarrow \mathbb{R}$ est une fonction associant un coût réel à chaque configuration, l'objectif étant de trouver une configuration $s^* \in S$ telle que $f(s^*)$ soit maximal. Un premier indicateur de la difficulté de cette instance est la densité d'états [HR96] qui donne la fréquence d'un coût c par rapport au nombre total de configurations : a priori, plus la densité d'un coût est faible, et plus il sera difficile de trouver une configuration avec ce coût.

Ce premier indicateur, indépendant de l'approche de résolution considérée, ne tient pas compte de la façon d'explorer les configurations. Or, on constate en pratique que cet aspect est déterminant : s'il y a une seule configuration optimale, mais que l'on dispose d'heuristiques simples pour se diriger vers elle dans l'espace des configurations, alors l'instance sera probablement plus facilement résolue que s'il y a plusieurs configurations optimales, mais qu'elles sont « cachées ».

Ainsi, on caractérise souvent la difficulté d'une instance en fonction de la topologie de son paysage de recherche [Sta95, FRPT99]. Cette topologie est définie par rapport à une relation de voisinage $v \subseteq S \times S$ qui dépend de l'approche considérée pour la résolution, ou plus précisément de la façon d'explorer l'ensemble des configurations : deux configurations s_i et s_j sont voisines si l'algorithme peut « explorer » s_j en une étape de résolution à partir de s_i. Cette relation de voisinage permet de structurer l'ensemble des configurations en un graphe $G = (S, v)$, que l'on peut représenter comme un « paysage » en définissant l'altitude d'un sommet $s_i \in S$ par la valeur de la fonction objectif $f(s_i)$. La topologie du paysage de recherche d'une instance par rapport à un voisinage donne des indications sur la difficulté de sa résolution par un algorithme explorant les configurations selon ce voisinage. En particulier, un paysage « rugueux », comportant de nombreux optima locaux (des sommets dont tous les voisins ont un coût inférieur), est généralement plus difficile qu'un paysage comportant peu d'optima. Un autre indicateur est la corrélation entre le coût d'un sommet et sa proximité à une solution optimale, l'instance étant généralement d'autant plus facilement résolue que cette corrélation est forte [JF95, MF99].

4.2 Exemples de problèmes combinatoires

De très nombreux problèmes réels sont $\mathcal{NP}$-complets ou $\mathcal{NP}$-difficiles, e.g., pour n'en citer que quelques uns, les problèmes de conception d'emplois du temps, d'équilibrage de chaines de montage, de routage de véhicules, d'affectation de fréquences, de découpe ou d'emballage. Ainsi, [GJ79] décrit près de trois cents problèmes $\mathcal{NP}$-complets ; le « compendium of $\mathcal{NP}$ optimization problems[2] » recense plus de deux cents problèmes d'optimisation $\mathcal{NP}$-difficiles, et la bibliothèque de problèmes en recherche opérationnelle « OR library[3] » en recense plus de cent.

On présente ici deux classes de problèmes combinatoires qui nous ont plus particulièrement intéressés, à savoir les problèmes de satisfaction de contraintes et les problèmes d'appariement de graphes.

[2] http ://www.nada.kth.se/∼viggo/problemlist/compendium.html
[3] http ://people.brunel.ac.uk/ mastjjb/jeb/info.html

4.2.1 Problèmes de satisfaction de contraintes

Les contraintes font partie de notre vie quotidienne, qu'il s'agisse par exemple de faire en emploi du temps, de ranger des pièces de formes diverses dans une boîte rigide, ou encore de planifier le trafic aérien pour que tous les avions puissent décoller et atterrir sans se percuter. La notion de « Problème de Satisfaction de Contraintes » (CSP) désigne l'ensemble de ces problèmes, définis par des contraintes, et consistant à chercher une solution les respectant.

De façon plus formelle, un CSP [Tsa93] est défini par un triplet (X, D, C) tel que
- X est un ensemble fini de variables (représentant les inconnues du problème) ;
- D est une fonction qui associe à chaque variable $x_i \in X$ son domaine $D(x_i)$, c'est-à-dire l'ensemble des valeurs que peut prendre x_i ;
- C est un ensemble fini de contraintes. Chaque contrainte $c_j \in C$ est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.

De façon générale, résoudre un CSP consiste à affecter des valeurs aux variables de telle sorte que toutes les contraintes soient satisfaites. De façon plus précise :
- On appelle *affectation* un ensemble de couples $\langle x_i, v_i \rangle$ tels que x_i est une variable du CSP et v_i est une valeur appartenant au domaine $D(x_i)$ de cette variable.
- Une affectation est dite *totale* si elle instancie toutes les variables du problème ; elle est dite *partielle* si elle n'en instancie qu'une partie.
- Une affectation A *viole* une contrainte c_j si toutes les variables de c_j sont instanciées dans A, et si la relation définie par c_j n'est pas vérifiée pour les valeurs des variables de c_j définies dans A.
- Une affectation (totale ou partielle) est *consistante* si elle ne viole aucune contrainte, et *inconsistante* si elle viole une ou plusieurs contraintes.
- Une *solution* est une affectation totale consistante, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte.

Notons que suivant les applications, « résoudre un CSP » peut signifier différentes choses : on peut chercher, par exemple, une solution (n'importe laquelle), ou bien toutes les solutions, ou bien une solution qui optimise une fonction objectif donnée, ou encore une affectation totale qui satisfait le plus grand nombre de contraintes (on parle alors de max-CSP) [FW92], ou qui minimise une somme pondérée des contraintes violées (on parle alors de CSP valué ou VCSP) [SFV95]. Ces différents types de CSPs ont été généralisés dans le cadre des CSPs basés sur des semi-anneaux [BMR97].

Tous les CSPs ne sont pas des problèmes combinatoires, et leur complexité théorique dépend de la nature des contraintes et des domaines des variables. Dans certains cas le problème peut être polynomial, par exemple lorsque les contraintes sont des équations ou inéquations linéaires et les domaines des variables sont continus. Dans d'autre cas, le problème peut être indécidable, par exemple lorsque les contraintes sont des fonctions mathématiques quelconques et les domaines des variables sont continus. Bien souvent, le problème est $\mathcal{NP}$-complet (lorsqu'il s'agit d'un problème de satisfaction) ou $\mathcal{NP}$-difficile (lorsqu'il s'agit d'un problème d'optimisation). En particulier, les CSPs sur les domaines finis (dont les domaines des variables sont des ensembles finis de valeurs) sont combinatoires dans le cas général.

Le cadre très général des CSPs permet de modéliser de nombreux problèmes réels (voir par exemple la bibliothèque CSPLib [GW99] qui recense 45 problèmes modélisés sous la forme de CSPs). On décrit dans les deux paragraphes suivants deux CSPs sur les domaines finis qui sont très souvent utilisés comme jeux d'essai pour évaluer la performance d'algorithmes de résolution, et sur lesquels nous avons plus particulièrement travaillé.

CSPs binaires aléatoires. Les CSPs binaires contiennent uniquement des contraintes binaires, i.e., chaque contrainte porte sur exactement deux variables. Ces CSPs peuvent être générés aléatoirement, ce qui peut sembler quelque peu futile, mais permet en pratique de tester et comparer facilement des algorithmes de résolution.

Une classe de CSPs binaires générés aléatoirement est caractérisée par un quadruplet $< n, m, p_1, p_2 >$ où
- n est le nombre de variables du CSP,
- m est la taille du domaine des variables,
- $p_1 \in [0, 1]$ détermine la connectivité du problème, c'est-à-dire le nombre de contraintes,
- $p_2 \in [0, 1]$ détermine la dureté des contraintes, c'est-à-dire le nombre de paires de valeurs incompatibles pour chaque contrainte.

Etant donnée une classe $< n, m, p_1, p_2 >$, on peut considérer différents modèles de génération, dépendant notamment de si l'on considère p_1 et p_2 comme des probabilités ou des ratios [MPSW98].

Une caractéristique très intéressante de cette classe de CSPs est que la région de la transition de phase a été clairement localisée [CFG+96] : les instances les plus difficiles (et donc celles qui vont plus particulièrement nous intéresser) sont celles pour lesquelles $\frac{n-1}{2} p_1 \log_m(\frac{1}{1-p_2}) = 1$. Notons toutefois que [AKK+97] a montré que ce résultat n'est pas valable pour les instances telles que $p_2 \geq 1/m$ lorsque le nombre de variables n tend vers l'infini : dans ce cas, la probabilité qu'il existe une variable surcontrainte —pour laquelle toutes les valeurs sont inconsistantes— tend vers 1 de sorte que quasiment toutes les instances sont insatisfiables. Dans ce cas, il n'y a plus qu'une seule région, et donc plus de transition de phase. Cependant, [MPSW98] montre que ce phénomène ne commence à apparaître que pour les instances ayant plus de 500 variables (lorsque le domaine des variables est de taille 10). Les instances auquelles nous nous sommes intéressés ayant toujours moins de 500 variables, la localisation de la région de la transition de phase proposée par [CFG+96] reste pertinente.

Problème d'ordonnancement de voitures. Ce problème consiste à séquencer des voitures le long d'une chaîne de montage pour installer des options sur les voitures (e.g., toit ouvrant ou air conditionné). Chaque option est installée par une station différente conçue pour traiter au plus un certain pourcentage de voitures passant le long de la chaîne. Par conséquent, les voitures demandant une même option doivent être espacées de telle sorte que la capacité des stations ne soit jamais dépassée. Plus formellement, on définira une instance de ce problème par un tuple (V, O, p, q, r) tel que
- $V = \{v_1, .., v_n\}$ est l'ensemble des voitures à produire,
- $O = \{o_1, .., o_m\}$ est l'ensemble des différentes options possibles,
- $p : O \rightarrow \mathbb{N}$ et $q : O \rightarrow \mathbb{N}$ sont deux fonctions qui définissent la contrainte de capacité associée à chaque option $o_i \in O$, i.e., pour chaque séquence de $q(o_i)$ voitures consécutives sur la ligne d'assemblage, au plus $p(o_i)$ peuvent demander l'option o_i,
- $r : V \times O \rightarrow \{0, 1\}$ est la fonction qui définit les options requises, i.e., pour chaque voiture $v_i \in V$ et pour chaque option $o_j \in O$, si o_j doit être installée sur v_i, alors $r(v_i, o_j) = 1$ sinon $r(v_i, o_j) = 0$.

Il s'agit alors de chercher un ordonnancement des voitures qui satisfait les contraintes de capacité, ou le cas échéant qui en viole le moins possible. Ce problème est $\mathcal{NP}$-difficile [Kis04] et est très facilement modélisable sous la forme d'un problème de satisfaction de contraintes [DSvH88, GW99, Tsa93].

Une variante de ce problème, faisant intervenir, en plus des contraintes de capacité liées aux options, des contraintes liées à la couleur des voitures, a fait l'objet du challenge ROADEF en 2005 [NC05].

4.2.2 Problèmes d'appariement de graphes

Les graphes sont des structures de données utilisées notamment pour modéliser des objets en termes de composants (appelés sommets), et de relations binaires entre composants (appelées arcs). De façon plus formelle, un graphe est défini par un couple $G = (V, E)$ tel que
- V est un ensemble fini de sommets,
- $E \subseteq V \times V$ est un ensemble d'arcs.

Les problèmes d'appariement de graphes consistent à mettre en correspondance les sommets de deux graphes, l'objectif étant généralement de comparer les objets modélisés par les graphes. On introduit ici les quatre

problèmes d'appariement les plus connus :
- l'isomorphisme de graphes, qui permet de vérifier que deux graphes sont structurellement identiques ;
- l'isomorphisme de sous-graphes, qui permet de vérifier qu'un graphe est « inclus » dans un autre ;
- le plus grand sous-graphe commun, qui permet d'identifier la plus grande partie commune à deux graphes ;
- la distance d'édition de graphes, qui permet d'identifier le plus petit nombre d'opérations à effectuer pour transformer un graphe en un autre.

Isomorphisme de graphes. Un graphe $G = (V, E)$ est isomorphe à un graphe $G' = (V', E')$ s'il existe une bijection $\phi : V \to V'$ qui préserve les arcs, i.e., $\forall (v_1, v_2) \in V^2, (v_1, v_2) \in E \Leftrightarrow (\phi(v_1), \phi(v_2)) \in E'$.

La complexité théorique du problème de l'isomorphisme de graphes n'est pas précisément établie : le problème est clairement dans $\mathcal{NP}$ mais on ne sait pas s'il est dans $\mathcal{P}$ ou s'il est $\mathcal{NP}$-complet [For96]. La classe des problèmes *isomorphiques-complets* a donc été définie, et on conjecture que cette classe se situe entre la classe P et la classe des problèmes $\mathcal{NP}$-complets. Cependant, il a été montré que pour certains types de graphes (e.g., les graphes planaires [HW74], les arbres [AHU74], ou les graphes à degré borné [Luk82]), le problème devient polynomial.

Isomorphisme de sous-graphes (partiel). Un graphe $G = (V, E)$ est un sous-graphe isomorphe d'un graphe $G' = (V', E')$ s'il existe une injection $\phi : V \to V'$ qui préserve les arcs de G, i.e., $\forall (v_1, v_2) \in V^2, (v_1, v_2) \in E \Leftrightarrow (\phi(v_1), \phi(v_2)) \in E'$.

Notons que l'isomorphisme de sous-graphes impose non seulement de retrouver tous les arcs de G dans G', mais aussi que tout couple de sommets de G non reliés par un arc soit apparié à un couple de sommets de G' également non reliés par un arc. L'isomorphisme de sous-graphe *partiel* relâche cette deuxième condition, i.e., il s'agit de trouver une injection $\phi : V \to V'$ telle que $\forall (v_1, v_2) \in V^2, (v_1, v_2) \in E \Rightarrow (\phi(v_1), \phi(v_2)) \in E'$.

Le problème de décider si un graphe est isomorphe à un sous-graphe (partiel) d'un autre graphe, est un problème $\mathcal{NP}$-complet [GJ79].

Plus grand sous-graphe (partiel) commun. Le plus grand sous-graphe commun de deux graphes $G = (V, E)$ et $G' = (V', E')$ est le plus « grand » graphe qui soit un sous-graphe isomorphe de G et G', la taille du sous-graphe commun pouvant être définie par le nombre de sommets et/ou le nombre d'arcs. Cette définition est naturellement étendue à la notion de plus grand sous-graphe partiel commun en recherchant le plus grand sous-graphe partiel isomorphe à G et G'.

Le problème de la recherche du plus grand sous-graphe (partiel) commun est un problème $\mathcal{NP}$-difficile.

Distance d'édition de graphes. La distance d'édition entre deux graphes G et G' est le coût minimal pour transformer G en G'. Dans le cas général, on considère des graphes étiquetés, i.e., les sommets et les arcs du graphe sont associés à des étiquettes. Pour cette transformation, on dipose de six opérations élémentaires : l'insertion, la suppression et le réétiquetage de sommets et d'arcs. Chaque opération a un coût. L'ensemble d'opérations le moins coûteux permettant de transformer G en un graphe isomorphe à G' définit la distance d'édition entre G et G'. La distance d'édition est en fait une généralisation de la notion de plus grand sous-graphe commun [Bun97] : si l'on considère des graphes non étiquetés, et si l'on définit le coût des opérations d'insertion et de suppression comme étant égal à 1, alors la distance d'édition donne l'ensemble des sommets et arcs qui n'appartiennent pas au plus grand sous-graphe commun.

Le problème du calcul de la distance d'édition de graphes est un problème $\mathcal{NP}$-difficile.

Formulation de problèmes d'appariements de graphes sous la forme de CSPs. Un même problème peut généralement être modélisé de différentes façons, et les problèmes d'appariement de graphes peuvent être formulés sous la forme de problèmes de satisfaction de contraintes [Rég95]. Ainsi, trouver un isomorphisme entre deux graphes $G = (V, E)$ et $G' = (V', E')$ se ramène à résoudre le CSP (X, D, C) tel que :

- X associe une variable x_u à chaque sommet $u \in V$, *i.e.*, $X = \{x_u / u \in V\}$,
- le domaine de chaque variable x_u est l'ensemble des sommets de G' ayant le même nombre d'arcs entrants et le même nombre d'arcs sortants que u, *i.e.*,

$$D(x_u) = \{u' \in V' \ / \quad \begin{array}{l} |\{(u,v) \in E\}| = |\{(u',v') \in E'\}| \text{ et} \\ |\{(v,u) \in E\}| = |\{(v',u') \in E'\}|\} \end{array}$$

- C contient une contrainte binaire $C_{edge}(x_u, x_v)$ entre chaque paire de variables (x_u, x_v). Cette contrainte exprime le fait que les sommets de G' affectés à x_u et x_v doivent être connectés par un arc si et seulement si les sommets correspondants u et v sont connectés par un arc, *i.e.*,

$$\begin{array}{ll} \text{si } (u,v) \in E, & C_{edge}(x_u, x_v) = E' \\ \text{sinon} & C_{edge}(x_u, x_v) = \{(u',v') \in V'^2 \mid u' \neq v' \text{ et } (u',v') \notin E'\} \end{array}$$

De façon très similaire, on peut modéliser un problème d'isomorphisme de sous-graphe sous la forme d'un CSP, tandis que les problèmes de recherche de plus grands sous-graphes communs et de calcul de distances d'édition, qui sont des problèmes d'optimisation et non plus de satisfaction, peuvent être modélisés sous la forme de CSPs valués.

4.3 Résolution pratique de problèmes combinatoires

De très nombreux problèmes réels devant être résolus quotidiennement sont combinatoires et il s'agit en pratique de trouver des solutions à ces problèmes. La théorie de la complexité nous disant que l'on ne pourra pas —a priori— trouver un algorithme à la fois rapide, correct et complet, on traite ces problèmes en faisant délibérément l'impasse sur un ou plusieurs de ces critères : l'objectif est d'avoir un comportement « acceptable » en pratique, i.e., d'être capable de trouver une solution de qualité suffisante en un temps raisonnable pour les instances à résoudre.

4.3.1 Approches complètes

Pour résoudre un problème combinatoire, on peut explorer l'ensemble des configurations de façon exhaustive et systématique, et tenter de réduire la combinatoire en utilisant des techniques d'« élagage » —visant à éliminer le plus tôt possible des sous-ensembles de configurations en prouvant qu'ils ne contiennent pas de solution— et des heuristiques de choix —visant à se diriger le plus rapidement possible vers les meilleures configurations. Ces approches sont complètes dans le sens où elles sont toujours capables de trouver la solution optimale ou, le cas échéant, de prouver l'absence de solution ; en revanche, le temps de résolution dépend de l'efficacité des techniques d'élagage et des heuristiques considérées, et est exponentiel dans le pire des cas.

Un point clé de ces approches réside dans la façon de structurer l'ensemble des configurations, l'objectif étant de pouvoir éliminer des sous-ensembles de configurations de façon globale, sans avoir à examiner les configurations qu'ils contiennent une par une.

Structuration en arbre. Pour explorer l'ensemble des configurations, on peut le découper en sous-ensembles de plus en plus petits. On construit pour cela un « arbre de recherche » dont la racine correspond à l'ensemble de toutes les configurations à examiner et dont les nœuds correspondent à des sous-ensembles de configurations de plus en plus petits au fur et à mesure que l'on descend dans l'arbre. Plus précisément, les sous-ensembles de configurations associés aux fils d'un nœud constituent une partition de l'ensemble des configurations associées au nœud. Cet arbre est généralement construit selon une stratégie par « séparation et évaluation » (« branch and bound »), en partant de la racine et en itérant sur les opérations suivantes :

– sélectionner une feuille de l'arbre selon une heuristique de sélection donnée (e.g., choisir la dernière feuille créée pour une exploration « en profondeur d'abord », ou choisir la feuille la plus « prometteuse » pour une exploration « du meilleur d'abord ») ;
– séparer (partitionner) l'ensemble des configurations associé à une feuille sélectionnée en plusieurs sous-ensembles (e.g., partitionner l'ensemble des configurations en autant de sous-ensembles qu'il y a de valeurs possibles pour une variable) ;
– évaluer le sous-ensemble de configurations associé à un noeud, afin de déterminer s'il peut contenir une solution (auquel cas le nœud correspondant devra être développé ultérieurement), ou s'il ne contient assurément aucune solution (auquel cas le nœud correspondant peut être « élagué »).

La fonction d'évaluation est déterminante pour la viabilité de l'approche en pratique : cette fonction doit pouvoir déterminer efficacement (i.e., plus rapidement qu'en énumérant toutes les configurations de l'ensemble) qu'un ensemble de configurations ne peut pas contenir de solution.

Pour les problèmes d'optimisation, la fonction d'évaluation retourne une estimation du coût de la meilleure solution de l'ensemble de sorte que si cette estimation est moins bonne que la meilleure solution trouvée jusqu'ici, alors on peut couper le nœud correspondant. Cette estimation du coût est généralement obtenue en relâchant certaines contraintes du problème de sorte que le problème « affaibli » ne soit plus combinatoire et puisse être évalué en temps polynomial. Par exemple, pour résoudre un problème linéaire en nombres entiers (consistant à trouver les valeurs entières à affecter à un ensemble de variables de décision optimisant une fonction objectif linéaire et satisfaisant un certain nombre de contraintes linéaires) selon une stratégie par « séparation et évaluation », on estime généralement le coût d'un nœud de l'arbre de recherche en relâchant les contraintes d'intégralité sur les variables, le problème sur les réels pouvant être résolu en temps faiblement polynomial [Chv83].

Pour les problèmes de décision, la fonction d'évaluation doit être capable de détecter le fait qu'un ensemble ne contient pas de configuration solution. Cette évaluation est souvent réalisée en « filtrant » l'ensemble des configurations, i.e., en éliminant des parties dont on infère qu'elles ne peuvent pas contenir de solution. Ainsi, une stratégie classique pour résoudre des problèmes de satisfaction de contraintes consiste à exploiter (« propager ») les contraintes du problème pour éliminer du domaine des variables les valeurs violant les contraintes, filtrant ainsi l'ensemble des configurations candidates. Le résultat de cette étape de filtrage vérifie généralement une certaine consistance partielle, e.g., la consistance de noeud, d'arc ou de chemin [Tsa93]. L'intégration de ces techniques de propagation de contraintes filtrant les domaines des variables à l'intérieur d'une exploration arborescente est à la base de nombreux « solveurs » de contraintes qui, intégrés dans des langages de programmation, constituent le paradigme de la « programmation par contraintes » [Fag96, MS98].

Structuration en treillis. Lorsque les configurations à explorer sont des sous-ensembles d'un ensemble donné, on peut structurer l'ensemble des configurations en un treillis et adopter une stratégie d'exploration par niveaux (levelwise search) [MT97]. Un exemple célèbre est l'algorithme « a priori » [AS94] utilisé pour calculer des ensembles fréquents maximaux ; ce principe peut également être utilisé pour générer toutes les solutions d'un problème de satisfaction de contraintes [Fre78].

De façon assez générale, cette stratégie d'exploration permet de trouver parmi un ensemble initial d'objets tous les plus grands sous-ensembles satisfaisant certaines contraintes : chaque niveau i du treillis contient l'ensemble des sous-ensembles comportant i éléments ; le treillis est construit en partant du niveau 1, et en construisant chaque niveau i à partir du niveau $i - 1$. Comme pour les approches par séparation et évaluation, l'efficacité de l'approche en pratique dépend de la capacité à élaguer le treillis, et de la possibilité d'utiliser activement les contraintes du problème pour examiner le moins de configurations possible [BJ05]. En particulier, les contraintes anti-monotones (telles que lorsqu'un ensemble vérifie les contraintes, alors tous ses sous-ensembles les vérifient aussi) permettent d'élaguer de façon drastique le treillis, et de traiter de très gros volumes de données en pratique.

4.3.2 Approches incomplètes

Les approches complètes permettent de résoudre en pratique de nombreux problèmes combinatoires, comme en témoigne le succès industriel de la programmation par contraintes. Cependant, les techniques d'élagage ne permettent pas toujours de réduire suffisamment la combinatoire, et certaines instances de problèmes ne peuvent être résolues en un temps acceptable par ces approches exhaustives.

Une alternative consiste alors à explorer l'ensemble des configurations de façon opportuniste, en utilisant des heuristiques pour se guider vers les zones de l'espace des configurations qui semblent plus prometteuses. Ces approches sont incomplètes dans le sens où elles n'explorent délibérément qu'une partie de l'espace des configurations. Par conséquent, elles peuvent ne pas trouver la solution optimale dans certains cas, et encore moins prouver l'optimalité des solutions trouvées ; en contrepartie, le temps de résolution est généralement faiblement polynomial.

On distingue essentiellement deux types d'approches heuristiques :
- Les approches par voisinage explorent l'espace des configurations en le structurant en termes de voisinage ; ces approches sont qualifiées de « instance-based » dans [ZBMD04] dans le sens où les nouvelles configurations sont créées à partir de configurations existantes.
- Les approches constructives génèrent de nouvelles configurations de façon incrémentale en ajoutant itérativement des composants de solution aux configurations en cours de construction ; ces approches sont qualifiées de « model-based » dans [ZBMD04] dans le sens où elles utilisent un modèle, généralement basé sur un mécanisme stochastique adaptatif, pour construire les configurations.

Approches par voisinage. L'idée est de structurer l'ensemble des configurations en terme de voisinage, et d'explorer cet ensemble en partant d'une ou plusieurs configurations initiales et en choisissant à chaque itération une ou plusieurs configurations voisines de une ou plusieurs configurations courantes. Les approches par voisinage les plus connues sont les algorithmes génétiques et la recherche locale.

Dans le cas des algorithmes génétiques [Hol75], on génère une population initiale de configurations que l'on fait évoluer en créant à chaque génération de nouvelles configurations à partir de la population de configurations courante. Pour ces algorithmes, les opérateurs de voisinage sont généralement le croisement —qui permet de générer deux nouvelles configurations en combinant les composants de deux configurations données— et la mutation —qui permet de générer une nouvelle configuration en modifiant aléatoirement quelques composants d'une configuration donnée. Il existe différentes variantes dépendant notamment de la politique de sélection des configurations à croiser et de la politique de remplacement.

Dans le cas de la recherche locale, on génère une configuration initiale, et l'on explore ensuite l'espace des configurations de proche en proche en sélectionnant à chaque itération une configuration voisine de la configuration courante, l'opérateur de voisinage consistant généralement en une modification « élémentaire » de la configuration courante. Il existe différentes variantes dépendant de la politique de sélection de la prochaine configuration parmi l'ensemble des configurations voisines. On peut par exemple sélectionner à chaque itération le meilleur voisin [SLM92], i.e., celui qui améliore le plus la fonction objectif, ou bien le premier voisin trouvé qui améliore la fonction objectif. De telles stratégies « gloutonnes » risquent fort d'être rapidement piégées dans des optima locaux, i.e., sur des configurations dont tous les voisins sont moins bons. Pour s'échapper de ces optima locaux, on peut considérer différentes méta-heuristiques, e.g., pour n'en citer que quelques unes :

le recuit simulé [AK89] —qui autorise de sélectionner des voisins de moins bonne qualité selon une probabilité qui décroit avec le temps— la recherche taboue [GL93] —qui empêche de boucler sur un petit nombre de configurations autour des optima locaux en mémorisant les derniers mouvements effectués dans une liste taboue et en interdisant les mouvements inverses à ces derniers mouvements— ou encore la recherche à voisinage variable [MH97] —qui change d'opérateur de voisinage lorsque la configuration courante est un optimum local par rapport au voisinage courant. Notons également que ce processus de recherche locale peut être répété plusieurs fois à partir de configurations initiales différentes, comme par exemple dans l'approche « iterated local search » [LMS02] où plusieurs recherches locales sont effectuées en séquence, la configuation initiale de chaque recherche locale étant une perturbation de la configuration finale de la recherche locale précédente.

Approches constructives. L'idée est de construire une ou plusieurs configurations de façon incrémentale, i.e., en partant d'une configuration « vide », et en ajoutant des composants de configuration jusqu'à obtenir une configuration complète. Les composants de configuration à ajouter à la configuration en cours de construction sont choisis en fonction d'une heuristique qui estime localement la qualité de ce composant. Il existe différentes stratégies pour choisir les composants à ajouter à chaque itération en fonction de l'heuristique, les plus connues étant les stratégies gloutonnes et gloutonnes aléatoires, la méta-heuristique d'optimisation par colonies de fourmis et les algorithmes par estimation de distribution.

Les algorithmes gloutons (greedy) construisent une configuration en choisissant à chaque itération un composant de solution pour lequel l'heuristique est maximale. Ces algorithmes permettent de construire des configurations très rapidement, les choix effectués n'étant jamais remis en cause. La qualité de la configuration construite dépend de l'heuristique.

Les algorithmes gloutons aléatoires (greedy randomized) construisent plusieurs configurations et adoptent également une stratégie gloutonne pour le choix des composants à ajouter aux configurations en cours de construction, mais ils introduisent un peu d'aléatoire afin de diversifier les configurations construites. On peut par exemple choisir aléatoirement le prochain composant parmi les k meilleurs ou bien parmi ceux qui sont à moins de α pour cent du meilleur composant [FR89]. Une autre possibilité consiste à choisir le prochain composant en fonction de probabilités dépendant de la qualité des différents composants [JS01].

Les algorithmes à base de fourmis —dont le principe général a donné naissance à la méta-heuristique d'optimisation par colonies de fourmis ACO [DD99, DCG99, DS04]— construisent également des configurations de façon incrémentale, mais choisissent le prochain composant en fonction d'une règle de transition probabiliste qui dépend généralement de deux facteurs : un facteur heuristique —similaire à celui utilisé par les algorithmes gloutons aléatoires— et un facteur phéromonal —fonction des « traces de phéromone » déposées autour du composant. Ces traces de phéromone représentent l'expérience passée de la colonie de fourmis concernant le choix de ce composant et sont régulièrement mises-à-jour en fonction des dernières configurations calculées.

Les algorithmes par estimation de distribution EDA [LL01] font évoluer une population de configurations P et un modèle probabiliste de génération de configurations M en itèrant sur les trois étapes suivantes : (1) construction d'un ensemble de configurations S à partir du modèle probabiliste courant M, (2) mise-à-jour de la population P en remplacant les moins bonnes configurations de P par les meilleures configurations de S, (3) mise-à-jour du modèle probabiliste M en fonction de P. Différents types de modèles probabilistes, plus ou moins simples, peuvent être utilisés pour construire les configurations, e.g., PBIL [Bal94] —qui se base sur la probabilité d'apparition de chaque composant sans tenir compte d'éventuelles relations de dépendances entre les composants— ou BOA [PGCP99] —qui utilise un réseau bayésien où les relations de dépendance entre composants sont représentées par les arcs d'un graphe auxquels est associée une distribution de probabilités conditionnelles.

Il existe un parallèle assez fort entre ACO et EDA [ZBMD04] : ces deux approches utilisent un modèle probabiliste pour générer les configurations, ce modèle évoluant en fonction des configurations construites dans un processus itératif « d'apprentissage ». Ces algorithmes peuvent être qualifiés d'évolutionnistes dans le sens où le modèle probabiliste générant les configurations évolue au cours du temps.

Intensification versus diversification de la recherche. Pour ces différentes approches heuristiques, explorant l'espace des configurations de façon opportuniste, un point critique dans la mise au point de l'algorithme consiste à trouver un compromis entre deux tendances duales :

– il s'agit d'une part d'intensifier l'effort de recherche vers les zones les plus « prometteuses » de l'espace des configurations, i.e., aux alentours des meilleures configurations trouvées ;
– il s'agit par ailleurs de diversifier l'effort de recherche de façon à être capable de découvrir de nouvelles zones contenant (potentiellement) de meilleures solutions.

La façon d'intensifier/diversifier l'effort de recherche dépend de l'approche considérée et se fait en modifiant certains paramétres de l'algorithme. Pour les approches par voisinage, l'intensification de la recherche se fait en favorisant l'exploration des « meilleurs » voisins. Par exemple, pour les algorithmes génétiques, on augmente la probabilité de croisement des meilleures configurations de la population et/ou on diminue la probabilité de mutation ; pour la recherche locale taboue, on diminue la longueur de la liste taboue ; pour le recuit simulé, on diminue la température pour diminuer la probabilité de choisir des « mauvais » voisins. Pour les approches par construction, l'intensification de la recherche se fait en augmentant la probabilité de choisir les « meilleurs » composants de solution. Par exemple, pour les algorithmes gloutons aléatoires, on augmente la probabilité de choisir les meilleurs composants de solution en diminuant le nombre k ou le ratio α ; pour la méta-heuristique ACO, on augmente la probabilité de choisir les composants de solution ayant appartenu aux meilleurs configurations trouvées en augmentant l'influence de la phéromone.

En général, plus on intensifie la recherche d'un algorithme en l'incitant à explorer les configurations proches des meilleures configurations trouvées, et plus il « converge » rapidement, trouvant de meilleures solutions plus tôt. Cependant, si l'on intensifie trop la recherche, l'algorithme risque fort de « stagner » autour d'optima locaux, concentrant son effort de recherche sur une petite zone autour d'une assez bonne configuration, sans plus être capable de découvrir de nouvelles configurations. Notons ici que le bon équilibre entre intensification et diversification dépend clairement du temps de calcul dont on dispose pour résoudre le problème : plus ce temps est petit et plus on a intéret à favoriser l'intensification pour converger rapidement, quitte à converger vers des configurations de moins bonne qualité.

Le bon équilibre entre intensification et diversification dépend également de l'instance à résoudre, ou plus particulièrement de la topologie de son paysage de recherche. En particulier, plus la corrélation entre la qualité d'une configuration et sa distance à la solution optimale est forte et plus on a intéret à intensifier la recherche. Différentes approches ont proposé d'adapter dynamiquement le paramétrage au cours de la résolution, en fonction de l'instance à résoudre. Par exemple, l'approche taboue réactive de [BP01] ajuste automatiquement la longueur de la liste taboue, tandis que l'approche locale de [NTG04] adapte automatiquement le nombre de voisins considérés à chaque mouvement.

4.3.3 ... et les autres approches

Cette présentation des approches pour la résolution pratique de problèmes combinatoires n'est certainement pas exhaustive, cette classe de problèmes ayant stimulé l'imagination d'un très grand nombre de chercheurs. En particulier, la « classification » proposée —distinguant approches complètes et incomplètes, structuration en arbre et en treillis, exploration par voisinage et par construction— n'est clairement pas exclusive, et de nombreuses approches hybrides ont été proposées, combinant les principes de plusieurs approches pour donner naissance à de nouveaux algorithmes. On peut par exemple combiner approches complètes et recherche locale [JL02, PV04, EGN05], algorithmes génétiques et recherche locale [Mos89], approches constructives et recherche locale [PR02].

Une alternative aux approches complètes et incomplètes pour la résolution de problèmes d'optimisation combinatoire, consiste à résoudre le problème en s'autorisant délibérément une marge d'erreur. En particulier, les algorithmes de « ρ-approximation » [Shm95] sont des algorithmes de complexité polynomiale qui calculent une configuration dont la qualité par rapport à la qualité de la configuration optimale est bornée par ρ. Les problèmes d'optimisation $\mathcal{NP}$-difficiles ne sont pas tous équivalents en termes « d'approximabilité » : certains comme le problème du « bin-packing » peuvent être approximés avec un facteur ρ quelconque, la complexité en temps de l'algorithme d'approximation augmentant lorsque le facteur d'erreur diminue ; d'autres comme le problème de la clique maximum ne peuvent pas être approximés avec un facteur constant, ni même polynomial (à moins que $\mathcal{P} = \mathcal{NP}$).

4.4 Organisation du mémoire

La suite de ce mémoire présente nos contributions à la résolution pratique de quelques problèmes combinatoires : le chapitre 5 présente nos travaux concernant la méta-heuristique d'optimisation par colonies de fourmis ACO et son application à la résolution de différents problèmes combinatoires ; le chapitre 6 présente nos travaux concernant un problème combinatoire difficile, à savoir le problème de l'évaluation de la similarité de deux graphes, et sa résolution par différentes approches.

Optimisation par colonies de fourmis. On décrit tout d'abord en 5.1 les principes généraux de la méta-heuristique ACO, inspirée du comportement des colonies de fourmis dans la nature, ainsi que l'algorithme $\mathcal{MMAS}$ dont on s'est plus particulièrement inspiré.

On s'intéresse alors en 5.2 au problème d'ordonnancement de voitures décrit en 4.2 et on introduit deux algorithmes ACO pour ce problème. Le premier algorithme, initialement présenté dans [Sol00] puis amélioré dans [GPS03], utilise la phéromone pour identifier des sous-séquences de voitures « prometteuses ». Le deuxième algorithme, qui n'a pas encore été publié[4], utilise la phéromone pour identifier les voitures « critiques », et nous montrons qu'il obtient sur de nombreuses instances de bien meilleurs résultats que les heuristiques gloutonnes généralement utilisées pour cela. On montre ensuite que ces deux algorithmes ACO peuvent être facilement combinés, améliorant ainsi les performances de chacun. Nous comparons notamment les performances de cet algorithme ACO « double » avec celles de l'approche par recherche locale qui a gagné le Challenge ROADEF 2005, et nous montrons que notre approche obtient de meilleurs résultats sur une majorité d'instances.

On s'intéresse ensuite en 5.3 à la classe des problèmes de sélection de sous-ensembles, dont l'objectif est de trouver, parmi un ensemble donné d'objets, un sous-ensemble satisfaisant un certain nombre de contraintes et/ou optimisant une fonction objectif donnée. Un très grand nombre de problèmes combinatoires relèvent de cette classe de problèmes, comme par exemple les problèmes de satisfaction de contraintes, de recherche de cliques maximum, de sac-à-dos multi-dimensionnel et d'appariement de graphes, et nous avons proposé différents algorithmes ACO pour ces problèmes [Sol02a, FS03, ASG04, SSG05, SF05]. Nous présentons ici ces différents algorithmes dans un cadre unifié, en définissant pour cette classe de problèmes un algorithme ACO générique, paramétré par les caractéristiques du problème de sélection de sous-ensembles à résoudre. On propose et compare pour cet algorithme deux stratégies phéromonales différentes, la première consistant à déposer la phéromone sur les objets sélectionnés et la seconde consistant à déposer la phéromone sur les paires d'objets sélectionnés.

Nous nous intéressons enfin dans la section 5.4 aux mécanismes ACO permettant de diversifier et intensifier la recherche et à l'influence des paramètres de ACO sur le processus de résolution. Nous introduisons pour

[4] Un article pour le numéro spécial de la revue « European Journal of Operations Research » sur le Challenge ROADEF 2005 est en cours de soumission.

cela deux mesures de diversification, i.e., le taux de ré-échantillonnage et le taux de similarité, et nous étudions l'influence des différents paramètres des algorithmes ACO sur la diversification de la recherche par rapport à ces deux indicateurs. Cette étude a été en partie publiée dans [SF05]. Nous décrivons enfin une procédure de pré-traitement que nous avons introduite dans [Sol02b] et qui permet d'accélérer la convergence d'un algorithme ACO en séparant la phase d'exploration —pendant laquelle la phéromone est délibérément ignorée— de la phase d'intensification —pendant laquelle la phéromone est utilisée pour guider la recherche.

Appariements de graphes et similarité. On introduit tout d'abord en 6.1 quelques points qui nous semblent importants lorsque l'on s'attaque au problème d'évaluer la similarité d'objets.

La section 6.2 décrit ensuite la mesure de similarité de graphes que nous avons introduite dans [CS03]. L'originalité de cette mesure vient du fait qu'elle est basée sur un appariement multivoque —autorisant un sommet d'un graphe à être apparié à plusieurs sommets de l'autre graphe— ce qui permet notamment de comparer des graphes décrivant des objets à des niveaux de granularité différents. Cette mesure est générique, dans le sens où elle est paramétrée par deux fonctions qui permettent d'intégrer des connaissances de similarité propres à l'application considérée, et on montre qu'elle est plus générale que d'autres approches souvent utilisées pour comparer et mettre en correspondance deux graphes, comme par exemple les appariements de graphes présentés en 4.2.2, mais aussi d'autres appariements multivoques introduits plus récemment. Cette comparaison de notre mesure avec d'autres appariements existants a été publiée dans [SS05].

On introduit ensuite en 6.3 différents algorithmes que nous avons proposés pour mettre en œuvre cette mesure : un algorithme glouton initialement introduit dans [CS03], un algorithme par recherche taboue et un algorithme par recherche réactive proposés dans [SS05], et un algorithme basé sur la méta-heuristique ACO proposé dans [SSG05]. On compare expérimentalement ces différents algorithmes.

On s'intéresse enfin en 6.4 à la résolution pratique du problème d'isomorphisme de graphes à l'aide de la programmation par contraintes. En effet, les problèmes d'appariements de graphes en général, et le problème de l'isomorphisme de graphes en particulier, peuvent aisément être modélisés sous la forme de CSPs, comme nous l'avons montré en 4.2.2. Cependant, la sémantique du problème original est alors décomposée en un ensemble de contraintes binaires, de sorte que le CSP apparait beaucoup plus difficile à résoudre que le problème original. Pour permettre une résolution plus efficace du problème de l'isomorphisme de graphes par la programmation par contraintes, on introduit une nouvelle contrainte globale et un algorithme de filtrage associé. Ce travail a été publié dans [SS04].

Chapitre 5

Optimisation par colonies de fourmis

5.1 Introduction

5.1.1 Des fourmis naturelles à la méta-heuristique d'optimisation par colonies de fourmis

Les fourmis sont capables de résoudre collectivement des problèmes complexes, comme trouver le plus court chemin entre deux points dans un environnement accidenté. Pour cela, elles communiquent entre elles de façon locale et indirecte, grâce à une hormone volatile, appelée phéromone : au cours de sa progression, une fourmi laisse derrière elle une trace de phéromone qui augmente la probabilité que d'autres fourmis passant à proximité choisissent le même chemin [DAGP90, DS04].

Ce mécanisme de résolution collective de problèmes est à l'origine des algorithmes à base de fourmis artificielles. Ces algorithmes ont été initialement proposés dans [Dor92, DMC96], comme une approche multi-agents pour résoudre des problèmes d'optimisation combinatoire. L'idée est de représenter le problème à résoudre sous la forme de la recherche d'un meilleur chemin dans un graphe, appelé graphe de construction, puis d'utiliser des fourmis artificielles pour rechercher de bons chemins dans ce graphe. Le comportement des fourmis artificielles est inspiré des fourmis réelles : elles déposent des traces de phéromone sur les composants du graphe de construction et elles choisissent leurs chemins relativement aux traces de phéromone précédemment déposées ; ces traces sont évaporées au cours du temps.

Intuitivement, cette communication indirecte via l'environnement —connue sous le nom de stigmergie— fournit une information sur la qualité des chemins empruntés afin d'attirer les fourmis et d'intensifier la recherche dans les itérations futures vers les zones correspondantes de l'espace de recherche. Ce mécanisme d'intensification est combiné à un mécanisme de diversification, essentiellement basé sur la nature aléatoire des décisions prises par les fourmis, qui garantit une bonne exploration de l'espace de recherche. Le compromis entre intensification et diversification peut être obtenu en modifiant les valeurs des paramètres déterminant l'importance de la phéromone.

Les fourmis artificielles ont aussi d'autres caractéristiques, qui ne trouvent pas leur équivalent dans la nature. En particulier, elles peuvent avoir une mémoire, qui leur permet de garder une trace de leurs actions passées. Dans la plupart des cas, les fourmis ne déposent une trace de phéromone qu'après avoir effectué un chemin complet, et non de façon incrémentale au fur et à mesure de leur progression. Enfin, la probabilité pour une fourmi artificielle de choisir un arc ne dépend généralement pas uniquement des traces de phéromone, mais aussi d'heuristiques dépendantes du problème permettant d'évaluer localement la qualité du chemin.

Ces caractéristiques du comportement des fourmis artificielles définissent la « méta-heuristique d'optimisation

Procédure $\mathcal{MAX}-\mathcal{MIN}$ ***Ant System***
Initialiser les traces de phéromone à τ_{max}
Répéter
Chaque fourmi construit une solution en exploitant les traces de phéromone
Optionnellement : améliorer les solutions construites par recherche locale
Evaporer les traces de phéromone en les multipliant par un facteur de persistance ρ
Récompenser les meilleures solutions en ajoutant de la phéromone sur leurs composants phéromonaux
si une trace de phéromone est inférieure à τ_{min} **alors** la mettre à τ_{min}
si une trace de phéromone est supérieure à τ_{max} **alors** la mettre à τ_{max}
Jusqu'à ce qu'un nombre maximal de cycles soit atteint **ou** une solution de qualité acceptable ait été trouvée
retourner la meilleure solution trouvée

FIG. 5.1 – Cadre algorithmique général du $\mathcal{MAX}-\mathcal{MIN}$ *Ant System*

par une colonie de fourmis » ou « Ant Colony Optimization (ACO) metaheuristic » [DD99, DCG99, DS04]. Cette méta-heuristique a permis de résoudre différents problèmes d'optimisation combinatoire, comme par exemple le problème du voyageur de commerce [DG97], le problème d'affectation quadratique [GTD99] ou le problème de routage de véhicules [BHS99].

5.1.2 Le $\mathcal{MAX}-\mathcal{MIN}$ *Ant System*

Le premier algorithme à base de fourmis, appelé *Ant System*, a été proposé par Marco Dorigo en 1992 [Dor92, DMC96], et ses performances ont été initialement illustrées sur le problème du voyageur de commerce. Malgré des premiers résultats encourageants, montrant que les performances de *Ant System* sont comparables à celles d'autres approches « généralistes » comme les algorithmes génétiques, l'algorithme *Ant System* n'est pas compétitif avec des approches spécialisées, développées spécialement pour le problème du voyageur de commerce. Ainsi, différentes améliorations ont été apportées à l'algorithme initial, donnant naissance à différentes variantes de *Ant System*, comme par exemple *ACS* (*Ant Colony System*) [DG97] et $\mathcal{MMAS}$ ($\mathcal{MAX}-\mathcal{MIN}$ *Ant System*) [SH00] qui obtiennent en pratique des résultats vraiment compétitifs, parfois meilleurs que les approches les plus performantes.

Nos travaux sur l'optimisation par colonies de fourmis se sont essentiellement inspirés du schéma algorithmique du $\mathcal{MMAS}$ décrit dans la figure 5.1. Conformément à la méta-heuristique ACO, à chaque cycle de l'algorithme chaque fourmi construit une solution. Ces solutions peuvent éventuellement être améliorées en appliquant une procédure de recherche locale. Les traces de phéromone sont ensuite mises-à-jour : chaque trace est « évaporée » en la multipliant par un facteur de persistance ρ compris entre 0 et 1 ; une certaine quantité de phéromone, généralement proportionnelle à la qualité de la solution, est ensuite ajoutée sur les composants des meilleures solutions (les meilleures solutions construites lors du dernier cycle ou les meilleures solutions construites depuis le début de l'exécution).

La construction d'une solution par une fourmi dépend de l'application considérée, mais suit toujours le même schéma général et se fait de façon incrémentale selon un principe glouton probabiliste : partant d'une solution initialement vide, la fourmi ajoute incrémentalement des composants de solution choisis selon une règle de transition probabiliste. En général, cette règle de transition définit la probabilité de choisir un composant en fonction de deux facteurs : un facteur phéromonal —qui traduit l'expérience passée de la colonie concernant le choix de ce composant— et un facteur heuristique —qui évalue localement la qualité du composant. L'importance respective de ces deux facteurs dans la probabilité de transition est généralement modulée par deux paramètres α et β.

Une caractéristique essentielle du $\mathcal{MMAS}$ est qu'il impose des bornes maximales et minimales τ_{min} et τ_{max} aux traces de phéromone. L'objectif est d'éviter une stagnation prématurée de la recherche, i.e., une situation où les traces de phéromone sont tellement marquées que les fourmis construisent les mêmes solutions à chaque itération et ne sont plus capables de trouver de nouvelles solutions. En effet, en imposant des bornes sur les quantités maximales et minimales de phéromone, on garantit que la différence relative entre deux traces ne peut devenir trop importante, et donc que la probabilité de choisir un sommet ne peut devenir trop petite. De plus, les traces de phéromone sont initialisées à la borne maximale τ_{max} au début de la résolution. Par conséquent, lors des premiers cycles de la recherche la différence relative entre deux traces est modérée (après i cycles, cette différence est bornée par un ratio de ρ^i où ρ est le facteur de persistance de la phéromone et a une valeur généralement très proche de 1), de sorte que l'exploration est accentuée.

5.1.3 Organisation du chapitre

On décrit en 5.2 une application de la méta-heuristique ACO au problème d'ordonnancement de voitures. Il s'agit là d'une application « classique » dans le sens où elle se formule très naturellement en un problème de recherche d'un meilleur chemin dans un graphe. Un point original de nos recherches sur ce problème est que nous avons proposé deux structures phéromonales complémentaires, la première étant utilisée pour « apprendre » les bons chemins, et la seconde étant utilisée pour « identifier » les sommets les plus « critiques ». La combinaison de ces deux structures nous a permis d'obtenir des résultats très compétitifs.

On décrit en 5.3 une application de la méta-heuristique ACO à la classe des problèmes de sélection de sous-ensembles optimaux. Il s'agit là de présenter dans un cadre unifié différents algorithmes ACO appliqués à différents problèmes se ramenant tous à la recherche d'un sous-ensemble d'objets optimal par rapport à des contraintes et/ou une fonction objectif. Contrairement au problème d'ordonnancement de voitures ou à celui du voyageur de commerce, ces problèmes ne sont pas des problèmes de recherche de chemins, où l'ordre dans lequel les sommets sont visités a un sens, mais des problèmes de sélection, où l'ordre dans lequel les sommets sont choisis n'est pas significatif. Nous décrivons et comparons deux structures phéromonales différentes pour ces problèmes de sélection.

On étudie enfin en 5.4 l'influence des paramètres des algorithmes ACO sur le processus de résolution. Cette étude est essentiellement menée sur un algorithme ACO appliqué à la recherche de cliques maximums, mais nous avons constaté des comportements très similaires pour tous les autres algorithmes ACO que nous avons développés. Nous introduisons pour cette étude deux indicateurs de diversification, i.e., le taux de ré-échantillonnage et le taux de similarité, qui nous permettent de quantifier et qualifier la capacité d'exploration et d'intensification de l'algorithme ACO. On montre en particulier que lorsque l'algorithme est bien paramétré, il ne construit (quasiment) jamais deux fois une même solution, faisant ainsi preuve d'une bonne exploration, mais intensifie progressivement ses efforts sur une zone plus restreinte de l'espace de recherche, les solutions calculées partageant de plus en plus de composants au fur et à mesure de la recherche. On introduit enfin une procédure de prétraitement qui permet d'accélérer la phase d'exploration d'un algorithme ACO.

5.2 Ordonnancement de voitures par colonies de fourmis

5.2.1 Notations

On s'intéresse ici à une application de la méta-heuristique ACO pour la résolution du problème d'ordonnancement de voitures décrit en 4.2. Etant donnée une instance de problème (C, O, p, q, r), on considère les notations suivantes :

– une *séquence*, notée $\pi =< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$, est une suite de voitures appartenant à C ;

- l'*ensemble de toutes les séquences* que l'on peut construire à partir d'un ensemble de voitures C est noté Π_C ;
- la *longueur* d'une séquence π, notée $|\pi|$, est le nombre de voitures qu'elle contient ;
- la *concatenation* de 2 séquences π_1 et π_2, noté $\pi_1 \cdot \pi_2$, est la séquence composée des voitures de π_1 suivies des voitures de π_2 ;
- une séquence π_f est un *facteur* d'une autre séquence π, noté $\pi_f \sqsubseteq \pi$, s'il existe deux autres séquences éventuellement vides π_1 et π_2 telles que $\pi = \pi_1 \cdot \pi_f \cdot \pi_2$;
- le *nombre de voitures qui demandent une option* o_i dans une séquence π (resp. dans un ensemble S de voitures) est noté $r(\pi, o_i)$ (resp. $r(S, o_i)$) et est défini par $r(\pi, o_i) = \sum_{<c_l>\sqsubseteq\pi} r(c_l, o_i)$ (resp. $r(S, o_i) = \sum_{c_l \in S} r(c_l, o_i)$) ;
- le *coût* d'une séquence π est le nombre de contraintes de capacité violées, i.e.,

$$\begin{aligned} cout(\pi) &= \sum_{o_i \in O} \sum_{\substack{\pi_k \sqsubseteq \pi \text{ tel que} \\ |\pi_k| = q(o_i)}} violation(\pi_k, o_i) \\ \text{où } violation(\pi_k, o_i) &= \begin{cases} 0 & \text{si } r(\pi_k, o_i) \leq p(o_i); \\ 1 & \text{sinon.} \end{cases} \end{aligned}$$

- la *classe* d'une voiture est définie par l'ensemble des options qui doivent être installées sur cette voiture, i.e.,

$$\forall c_i \in C, classOf(c_i) = \{o_k \in O / r(c_i, o_k) = 1\}$$

5.2.2 Un premier algorithme ACO pour identifier les « bonnes » séquences de voitures

Résoudre une instance (C, O, p, q, r) du problème d'ordonnancement de voitures se ramène à rechercher une permutation des voitures de C satisfaisant les contraintes de capacité. Ce problème se formule très naturellement en un problème de recherche d'un meilleur chemin hamiltonien dans un graphe. Il s'agit là d'une application classique d'ACO, dans la lignée du problème du voyageur de commerce, pour laquelle on dépose la phéromone sur les arcs empruntés par les fourmis afin d'identifier les séquences de sommets « prometteuses ». On décrit ici un premier algorithme ACO basé sur ce principe. Cet algorithme suit le schéma général du $\mathcal{MMAS}$ rappelé dans la Figure 5.1 ; les quatres paragraphes suivants décrivent respectivement le graphe de construction considéré, la procédure de construction d'une séquence par une fourmi, l'heuristique locale utilisée par les fourmis et la procédure de dépot de phéromone.

Graphe de construction. Le graphe de construction associé à une instance de problème (C, O, p, q, r) est le graphe complet orienté associant un sommet à chaque voiture $c_i \in C$. Les fourmis communiquent en déposant des traces de phéromone sur les arcs de ce graphe. La quantité de phéromone sur un arc (c_i, c_j) est notée $\tau(c_i, c_j)$ et représente l'expérience passée de la colonie concernant le fait de séquencer la voiture c_j juste derrière la voiture c_i.

Construction d'une séquence par une fourmi. L'algorithme de construction d'une séquence de voitures π par une fourmi est décrit dans la figure 5.2. La première voiture de la séquence est choisie selon une probabilité qui dépend d'une heuristique η décrite dans le paragraphe suivant. On ajoute ensuite itérativement des voitures en fin de séquence jusqu'à ce que toutes les voitures aient été séquencées. A chaque itération, l'ensemble *cand* des voitures pouvant être ajoutées en fin de séquence est limité à l'ensemble des voitures qui introduisent le plus petit nombre de nouvelles violations (ligne 4). Afin d'éliminer les symétries dues au fait que deux voitures peuvent demander les mêmes options, on restreint également l'ensemble des candidats aux voitures appartenant à des classes de voitures différentes, ne gardant pour chaque classe de voitures que celle ayant le plus petit

Procédure construire-séquence
Entrées : une instance (C, O, p, q, r) du problème d'ordonnancement de voitures
une fonction heuristique $\eta : C \times \Pi_C \rightarrow \mathbb{R}^+$
une matrice phéromonale : $\tau : C \times C \rightarrow \mathbb{R}^+$
deux paramètres à valeurs numériques : α et β
Sortie : une séquence π contenant chaque voiture de C une et une seule fois
1- choisir une première voiture $c_i \in C$ selon la probabilité $p_{c_i} = \frac{[\eta(c_i,<>)]^\beta}{\sum_{c_k \in C}[\eta(c_k,<>)]^\beta}$
2- $\pi \leftarrow < c_i >$
3- **tant que** $\mid \pi \mid \leq \mid C \mid$ **faire**
4- $cand \leftarrow \{c_k \in C - \pi$ tel que $\forall c_j \in C - \pi, cout(\pi. < c_k >) \leq cout(\pi. < c_j >)$ et
5- $(classOf(c_k) = classOf(c_j)) \Rightarrow (k \leq j) \}$
6- choisir $c_j \in cand$ selon la probabilité $p_{c_j} = \frac{[\tau(c_i,c_j)]^\alpha \cdot [\eta(c_j,\pi)]^\beta}{\sum_{c_k \in cand}[\tau(c_i,c_k)]^\alpha \cdot [\eta(c_k,\pi)]^\beta}$
7- $\pi \leftarrow \pi \cdot < c_j >$
8- $c_i \leftarrow c_j$
9- **fin tant**
10- **retourne** π

FIG. 5.2 – Algorithme de construction d'une séquence de voitures par une foumi

numéro (ligne 5). Etant donné l'ensemble de voitures candidates *cand*, la prochaine voiture à séquencer, c_j, est choisie selon une probabilité qui dépend (1) de la quantité de phéromone sur l'arc reliant la dernière voiture séquencée c_i à la voiture candidate c_j et (2) de l'heuristique η. L'importance relative de ces deux facteurs peut être réglée par l'intermédiaire des deux paramètres α et β.

Fonction heuristique η. Nous avons introduit et comparé dans [GPS03] cinq définitions différentes pour la fonction heuristique η. Ces définitions sont toutes liées aux taux d'utilisation des options [Smi96] qui traduisent le fait qu'une option est d'autant plus critique qu'elle est demandée par beaucoup de voitures par rapport à sa capacité.

La fonction heuristique qui s'est montrée la plus performante en moyenne est basée sur la somme des taux d'utilisation des options demandées par la voiture candidate, i.e.,

$$\eta(c_j, \pi) = \sum_{o_i \in O} r(c_j, o_i) \cdot \mathrm{dynUtilRate}(o_i, C - \pi)$$

où $\mathrm{dynUtilRate}(o_i, \pi)$ est le taux d'utilisation de l'option o_i pour les voitures restant à produire, i.e.,

$$\mathrm{dynUtilRate}(o_i, \pi) = \frac{[r(C, o_i) - r(\pi, o_i)] \cdot q(o_i)}{p(o_i) \cdot [\mid C \mid - \mid \pi \mid]}$$

Nous avons évalué cette heuristique en exécutant l'algorithme de la figure 5.2 en « neutralisant » l'action de la phéromone, i.e., en mettant le paramètre α à 0 de sorte que la probabilité de choix d'une voiture ne dépend plus que de l'heuristique. Les expérimentations nous ont montré qu'un tel algorithme glouton probabiliste peut résoudre en quelques constructions seulement toutes les instances du jeu d'essai généré par J. Lee [LLW98] : ce jeu d'essai contient 70 instances satisfiables, chaque instance ayant 200 voitures et 5 options. L'algorithme glouton probabiliste est capable de résoudre toutes ces instances en quelques exécutions seulement : 17 constructions en moyenne (106 constructions pour l'instance la plus difficile, et 1 pour les instances les plus faciles) sont nécessaires pour résoudre chaque instance ; le temps d'exécution pour construire une séquence étant inférieur à 1 millième de seconde.

Dépot de phéromone. A la fin de chaque cycle de l'algorithme $\mathcal{MMAS}$, lorsque chaque fourmi a construit une solution et après avoir évaporé la phéromone, des traces de phéromone sont déposées sur les solutions construites afin d'attirer les fourmis suivantes vers les zones correspondantes de l'espace de recherche.

On adopte ici une stratégie élitiste, où seules les meilleures fourmis du cycle —celles qui ont construit une séquence de coût minimal par rapport à l'ensemble des séquences construites pendant le cycle— déposent de la phéromone. Pour ces meilleures fourmis, on récompense les arcs correspondant à des couples de voitures visitées consécutivement ; la quantité de phéromone déposée est inversement proportionnelle au nombre de contraintes violées.

De façon plus précise, si l'ensemble des séquences construites pendant le cycle est S_π, alors la procédure de dépot de phéromone est la suivante :

pour chaque séquence $\pi_i \in S_\pi$ **faire**
  si $\forall \pi_k \in S_\pi, cout(\pi_k) \geq cout(\pi_i)$ **alors**
    pour chaque couple de voitures consécutives $< c_j, c_k > \sqsubseteq \pi_i$ **faire**
      $\tau(c_j, c_k) \leftarrow \tau(c_j, c_k) + 1/cout(\pi_i)$

5.2.3 Un deuxième algorithme ACO pour identifier les voitures « critiques »

La fonction heuristique η de notre premier algorithme ACO a pour but de favoriser la sélection de voitures « critiques », i.e., de voitures sur lesquelles doivent être installées des options ayant de forts taux d'utilisation de sorte qu'elles peuvent difficilement être séquencées sans violer des contraintes de capacité. Nous proposons maintenant d'utiliser la méta-heuristique ACO pour identifier ces voitures critiques en fonction des expériences passées.

Structure phéromonale. Les traces de phéromone sont déposées sur les classes de voitures, une classe de voitures regroupant l'ensemble des voitures demandant un même sous-ensemble d'options : étant donné un ensemble C de voitures, l'ensemble des classes de voitures est

$$classes(C) = \{classOf(c_i)/c_i \in C\}$$

Etant donnée une classe de voitures $cc \in classes(C)$, on note $\tau(cc)$ la quantité de phéromone déposée sur elle. Intuitivement, cette quantité de phéromone représente l'expérience passée de la colonie concernant la difficulté de séquencer les voitures de cette classe.

L'algorithme ACO que nous proposons ici n'entre pas dans le cadre du $\mathcal{MMAS}$ tel qu'il est présenté en 5.1 : nous n'introduisons qu'une borne minimale τ_{min} (garantissant que la probabilité de choisir une voiture ne sera jamais nulle), et pas de borne maximale τ_{max}. De plus, les traces de phéromone sont initialisées à la borne minimale τ_{min} au début de la recherche, l'objectif étant d'identifier le plus rapidement possible les classes de voiture critiques.

Construction d'une séquence par une fourmi. Les fourmis construisent incrémentalement des séquences, suivant en cela l'algorithme de la figure 5.2 mais en considérant une autre définition en ce qui concerne la probabilité de choisir une voiture c_j parmi l'ensemble des voitures candidates *cand* (ligne 6). Cette probabilité dépend maintenant de la quantité de phéromone déposée sur la classe de la voiture c_j, l'objectif étant de favoriser la sélection des voitures les plus critiques :

$$p_{c_j} = \frac{[\tau(classOf(c_j))]^\gamma}{\sum_{c_k \in cand}[\tau(classOf(c_k))]^\gamma}$$

Le paramètre γ est introduit pour régler le degré d'influence de la phéromone sur le choix des sommets.

Mise-à-jour de la phéromone. Les fourmis déposent des traces de phéromone pendant la construction des séquences : à chaque fois que plus aucune voiture ne peut être séquencée sans introduire de nouvelles violations de contraintes, une trace de phéromone est ajoutée sur chaque classe de voiture dont au moins une voiture reste à séquencer, indiquant ainsi que les voitures de ces classes devraient être séquencées en priorité. La quantité de phéromone déposée est égale au nombre de nouvelles violations de contraintes introduites par les voitures de la classe. Plus précisément, on modifie l'algorithme de la figure 5.2 en ajoutant entre les lignes 5 et 6 les trois lignes suivantes :

si $\forall c_i \in cand,\ cout(\pi. < c_i >) - cout(\pi) > 0$ **alors**
  pour chaque classe de voitures $cc \in classes(C - \pi)$ **faire**
    $\tau(cc) \leftarrow \tau(cc) + cout(\pi. < c_k >) - cout(\pi)$ où c_k est une voiture de la classe cc

Notons que ces dépôts phéromonaux ont lieu pendant la construction d'une séquence, et non une fois que toutes les fourmis ont construit une séquence comme cela est souvent le cas dans les algorithmes ACO. En effet, les quantités de phéromone déposées ne dépendent pas de la qualité globale des séquences construites durant le cycle, mais de l'évaluation locale de la voiture par rapport à la séquence en cours de construction. Notons également que toutes les fourmis déposent de la phéromone, et non uniquement les meilleures du cycle.

Enfin, les dépots phéromonaux ayant lieu à chaque construction d'une séquence, l'évaporation a également lieu à la fin de la construction de chaque séquence.

5.2.4 Un algorithme ACO « double »

Les structures phéromonales utilisées dans les deux algorithmes ACO introduits précédemment sont complémentaires : la première vise à identifier des sous-séquences de voitures prometteuses, la seconde vise à identifier des classes de voitures critiques. On peut donc les combiner très facilement. L'algorithme résultant utilise deux structures phéromonales différentes :

– Les fourmis déposent de la phéromone sur les couples de voitures $(c_i, c_j) \in C \times C$, la quantité de phéromone $\tau(c_i, c_j)$ représentant l'expérience passée de la colonie concernant le fait de séquencer la voiture c_j juste derrière la voiture c_i.
  Pour cette structure phéromonale, les quantités de phéromone sont mises-à-jour à la fin de chaque cycle, une fois que chaque fourmi a construit une séquence, et seules les meilleures fourmis déposent de la phéromone.
– Les fourmis déposent également de la phéromone sur les classes de voiture $cc \in Classes(C)$, la quantité de phéromone $\tau(cc)$ représentant l'expérience passée de la colonie concernant la difficulté de séquencer les voitures de cette classe sans violer de contraintes.
  Pour cette structure phéromonale, la phéromone est déposée par chaque fourmi pendant qu'elle construit une séquence, et la phase d'évaporation a lieu à la fin de chaque construction d'une séquence par une fourmi.

L'algorithme de construction d'une séquence par une fourmi est celui de la figure 5.2, mais en considérant une autre définition en ce qui concerne la probabilité de choisir une voiture c_j parmi l'ensemble des voitures candidates *cand* (ligne 6). Cette probabilité dépend maintenant de deux facteurs phéromonaux différents, le premier évaluant l'intéret de séquencer c_j juste derrière la dernière voiture séquencée, et le second évaluant la difficulté de placer c_j. De façon plus précise :

$$p_{c_j} = \frac{[\tau(c_i, c_j)]^\alpha \cdot [\tau(classOf(c_j))]^\gamma}{\sum_{c_k \in cand} [\tau(c_i, c_k)]^\alpha \cdot [\tau(classOf(c_k))]^\gamma}$$

5.2.5 Comparaison expérimentale des différents algorithmes ACO

Jeux d'essais. Les instances du jeu d'essai généré par J. Lee [LLW98] sont trivialement résolues par les différents algorithmes ACO que nous venons de décrire en moins d'un centième de secondes en moyenne. Nous considérons donc ici un nouveau jeu d'essai contenant des instances plus difficiles générées par Laurent Perron et décrites dans [PS04]. Les instances de ce jeu d'essai ont toutes $|O| = 8$ options et $|Classes(C)| = 20$ classes de voitures différentes par instance ; les contraintes de capacité sur les options, définies par les fonctions p et q, sont générées aléatoirement en respectant les contraintes suivantes : $\forall o_i \in O, 1 \leq p(o_i) \leq 3$ et $p < q \leq p + 2$. Toutes ces instances admettent une solution ne violant aucune contrainte.

Ces instances sont regroupées en trois groupes en fonction du nombre de voitures $|C|$ à séquencer : le premier groupe contient 32 instances de 100 voitures, le deuxième 21 instances de 300 voitures et le troisième 29 instances de 500 voitures. Pour chacun de ces trois groupes, nous avons séparé l'ensemble des instances en deux sous-groupes en fonction de leur « difficulté » : lorsque tous les algorithmes considérés ont réussi à résoudre l'instance, on considère qu'elle est « facile », sinon on considère qu'elle est « difficile ». Ainsi, il y a 23 (resp. 14 et 13) instances à 100 (resp. 300 et 500) voitures qui sont faciles et 9 (resp. 7 et 16) instances à 100 (resp. 300 et 500) voitures qui sont difficiles.

Conditions expérimentales. Les différents algorithmes considérés et leur paramétrage sont les suivants (on trouvera une discussion plus détaillée sur l'influence des paramètres sur le processus de résolution en 5.4) :

- *ACO-seq* correspond au premier algorithme ACO, pour lequel la phéromone est déposée sur les séquences de voitures. Pour cet algorithme, on a fixé le poids du facteur phéromonal α à 2, le poids du facteur heuristique β à 6, le taux de persistance ρ à 0.99, et les bornes minimales τ_{min} et τ_{max} à 0.01 et 4 respectivement.
- *Glouton* correspond au premier algorithme aussi, mais pour lequel le facteur phéromonal est délibérément ignoré en mettant son poids α à 0, de sorte que l'algorithme se comporte comme un algorithme glouton probabiliste, utilisant comme heuristique gloutonne η. Pour cet algorithme, on a fixé le poids du facteur heuristique β à 6.
- *ACO-classes* correspond au deuxième algorithme ACO, pour lequel la phéromone est déposée sur les classes de voitures. Pour cet algorithme, on a fixé le poids du facteur phéromonal γ à 6, le taux de persistance ρ à 0.99, et la borne minimale τ_{min} à 1.
- *ACO-seq+classes* correspond à l'algorithme ACO double, combinant les deux structures phéromonales. Pour la première structure phéromonale, on a fixé le poids du facteur phéromonal α à 2, le taux de persistance ρ à 0.99, et les bornes minimales τ_{min} et τ_{max} à 0.01 et 4 respectivement. Pour la deuxième structure phéromonale, on a fixé le poids du facteur phéromonal γ à 6, le taux de persistance ρ à 0.99, et la borne minimale τ_{min} à 1.

Chaque algorithme considéré construit au maximum 150000 séquences par exécution : pour *ACO-seq* et *ACO-seq+classes* on a fixé le nombre maximum de cycles à 5000 et le nombre de fourmis à 30 ; pour *Glouton* et *ACO-classes* on a fixé le nombre maximum de cycles à 150000, une seule séquence étant construite à chaque cycle (notons que pour *Glouton*, nous avons supprimé les opérations concernant la gestion de la phéromone, i.e., le calcul des facteurs phéromonaux, les dépôts phéromonaux et l'évaporation).

Les algorithmes ont été implémentés en C et ont été exécutés sur un Pentium 4 cadencé à 2GHz. Comme ils sont non déterministes, nous les avons exécutés 50 fois pour chaque instance considérée.

Comparaison sur les instances faciles. Pour les instances faciles, toutes les exécutions de chacun des quatre algorithmes considérés ont trouvé une solution. Ces instances ne sont donc pas très discriminantes pour nos algorithmes. On constate cependant des différences en ce qui concerne le nombre moyen de séquences construites et le temps CPU correspondant pour trouver une solution : *Glouton* construit en moyenne 2696 séquences en 0.96 secondes ; *ACO-seq* en construit 602 en 0.24 secondes ; *ACO-classes* en construit 593 en 0.13 secondes et

TAB. 5.1 – Résultats expérimentaux de *Glouton*, *ACO-classes*, *ACO-seq* et *ACO-seq+classes* sur les instances difficiles. Chaque ligne donne le numéro de l'instance suivi des résultats obtenus par chacun des quatre algorithmes : le pourcentage d'exécutions ayant trouvé une solution, le temps CPU moyen en secondes (suivi de l'écart-type entre parenthèses) mis pour trouver une solution et le nombre de séquences construites divisé par 30 (ce nombre correspond au nombre de cycles effectués par *ACO-seq* et *ACO-seq+classes* puisque ces deux algorithmes construisent 30 séquences par cycle).

	Glouton			*ACO-classes*			*ACO-seq*			*ACO-seq+classes*		
Nom	%	temps	$\frac{\#\,seq}{30}$	%	temps	$\frac{\#\,seq}{30}$	%	temps	$\frac{\#\,seq}{30}$	%	temps	$\frac{\#\,seq}{30}$
100-11	16	9.1(5.6)	2331	100	3.2 (3.2)	909	100	2.9(1.2)	649	100	0.7 (0.3)	176
100-16	0			12	7.8 (3.6)	2347	0			100	1.5 (0.6)	420
100-19	0			0			0			84	8.9 (4.3)	2339
100-21	2	15.3(0.0)	4178	6	11.9 (4.2)	3489	44	9.0(4.8)	2097	98	2.7 (2.5)	730
100-60	0			86	4.3 (3.6)	1312	0			100	0.9 (0.3)	246
100-82	10	7.7(4.6)	2163	54	7.2 (3.3)	2330	64	2.3(3.8)	571	100	2.0 (2.0)	608
100-85	0			10	5.0 (4.1)	1649	0			24	9.5 (4.1)	2798
100-86	0			0			0			52	6.5 (4.8)	1795
100-98	0			0			4	17.6(4.2)	4276	2	26.5 (0.0)	4809
300-08	38	20.5(12.0)	2133	100	0.2 (0.2)	20	100	1.8(0.5)	152	100	0.1 (0.1)	12
300-25	78	19.8(15.0)	1940	100	0.6 (0.6)	65	100	2.0(0.4)	163	100	0.3 (0.3)	30
300-37	0			100	4.0 (3.8)	447	100	4.2(3.2)	349	100	1.5 (0.7)	142
300-40	0			0			0			82	16.0 (11.5)	1482
300-45	0			78	19.6 (13.1)	2181	72	8.6(10.0)	687	100	2.9 (1.1)	280
300-80	8	26.2(14.5)	2432	100	0.1 (0.1)	11	100	2.8(0.5)	218	100	0.1 (0.1)	8
300-81	0			0			0			100	14.7 (10.3)	1388
500-07	0			0			0			100	15.3 (2.9)	593
500-13	0			0			0			54	38.2 (27.2)	1536
500-27	0			0			0			24	42.5 (21.2)	1614
500-28	0			0			0			100	15.7 (2.8)	593
500-30	0			100	0.0 (0.0)	2	90	13.4(4.1)	549	100	0.0 (0.0)	2
500-34	0			0			0			24	43.7 (24.5)	1758
500-38	0			100	0.5 (0.6)	27	0			100	0.7 (0.7)	26
500-52	100	11.4(11.4)	698	0			100	0.8(0.3)	36	100	11.3 (1.7)	484
500-53	0			100	5.6 (7.0)	356	96	14.5(20.9)	570	100	1.7 (1.3)	72
500-62	0			0			100	9.7(1.8)	421	100	8.4 (0.7)	350
500-64	98	20.5(19.3)	1169	100	0.0 (0.0)	1	100	1.6(0.4)	62	100	0.0 (0.0)	1
500-65	32	37.9(24.8)	2519	100	0.1 (0.1)	9	100	1.9(0.5)	89	100	0.1 (0.1)	8
500-68	0			0			12	55.2(28.8)	2072	100	16.5 (5.9)	673
500-77	0			100	0.2 (0.2)	11	100	7.4(0.6)	297	100	0.2 (0.2)	10
500-88	0			18	37.1 (15.0)	2570	22	55.5(36.9)	2097	22	18.5 (15.2)	891
500-93	0			0			38	25.5(24.8)	1067	100	9.1 (1.1)	367

ACO-seq+classes en construit 206 en 0.07 secondes.

Comparaison sur les instances difficiles. Le tableau 5.1 donne les résultats obtenus par chacun des quatre algorithmes pour les instances difficiles. Comparons tout d'abord les performances de *Glouton* et *ACO-classes*, qui construisent tous deux les séquences en utilisant une heuristique gloutonne estimant la « difficulté » des classes de voitures, mais qui diffèrent sur la façon de faire cette estimation : *Glouton* se base sur la somme des taux d'utilisation des options demandées par les classes, tandis que *ACO-classes* « apprend » cela en fonction de l'expérience passée. On constate qu'à l'exception (notable) de l'instance 500-52, *ACO-classes* a un pourcentage de réussite nettement supérieur à *Glouton*, et est également bien plus rapide en général : les deux algorithmes mettent à peu près le même temps pour construire une séquence, mais *ACO-classes* en construit généralement sensiblement moins que *Glouton* pour trouver une solution.

Lorsque l'on compare les performances de *Glouton* et *ACO-classes* avec celles de *ACO-seq* et *ACO-seq+classes* respectivement, on remarque que l'intégration de la structure phéromonale pour « apprendre » les bonnes séquences de voitures améliore toujours les performances : *ACO-seq* (resp. *ACO-seq+classes*) a été capable de résoudre 11 (resp. 14) instances de plus que *Glouton* (resp. *ACO-classes*) ; le nombre de séquences construites pour trouver une solution, et donc les temps d'exécutions, étant également nettement plus petits. Ainsi, l'algorithme combinant les deux structures phéromonales —*ACO-seq+classes*— a été capable de résoudre chaque instance au moins une fois sur les 50 exécutions considérées.

Notons par ailleurs que pour des instances de même taille, les temps d'exécution d'*ACO-seq+classes* peuvent varier considérablement : certaines instances (comme 500-64 ou 500-65) sont en fait résolues par la deuxième structure phéromonale (visant à « apprendre » les classes critiques) en quelques cycles seulement, tandis que d'autres (comme 500-13 ou 500-27) ont besoin de plusieurs centaines de cycles avant que la première structure phéromonale (visant à « apprendre » les bonnes séquences) ne permette à l'algorithme de converger.

5.2.6 Comparaison avec d'autres approches

Approches considérées. De très nombreuses approches différentes ont été proposées pour ce problème d'ordonnancement de voitures, qui est un problème de référence pour évaluer l'efficacité de nouvelles approches de résolution. Les approches complètes, basées sur la programmation par contraintes [DSvH88, RP97] ou la programmation linéaire en nombres entiers [GGP04] sont généralement limitées à des instances de petites tailles et ne permettent pas, par exemple, de résoudre en un temps raisonnable certaines instances du jeu d'essai généré par J. Lee [LLW98], alors que nos différents algorithmes ACO les résolvent toutes en moins d'un centième de seconde.

De nombreuses approches heuristiques ont également été proposées, la plupart d'entre elles étant basées sur une exploration par recherche locale, e.g., pour n'en citer que quelques unes, [DT99, LLW98, MH02, GPS03, NTG04, PS04, EGN05]. On compare maintenant le meilleur de nos quatre algorithmes, *ACO-seq+classes*, avec deux de ces approches par recherche locale qui se sont avérées particulièrement performantes, i.e., *IDWalk* et *PV*.

IDWalk [NTG04] est une nouvelle méta-heuristique basée sur la recherche locale qui s'est avérée plus performante que d'autres méta-heuristiques basées sur la recherche locale —comme la recherche taboue et le recuit simulé— notamment pour le problème d'ordonnancement de voitures. En plus du nombre maximum S de mouvements autorisés, *IDWalk* a un seul paramètre Max qui donne le nombre maximum de voisins considérés à chaque mouvement. A chaque itération, l'algorithme choisit le premier voisin non détériorant, et si tous les Max voisins détériorent la configuration courante, l'algorithme choisit celui qui détériore le moins la configuration. Le paramètre Max est réglé automatiquement en effectuant au début de la recherche quelques marches courtes (de 20000 mouvements) avec différentes valeurs possibles, puis en lançant une marche longue avec la valeur du paramètre réglé. Ce paramétrage est également ajusté à plusieurs reprises au cours de la résolution lorsque la longueur de la marche atteint certains seuils. Dans le cas du problème d'ordonnancement de voitures, le voisinage considéré est un échange entre deux voitures appartenant à des classes différentes dont une participe à un conflit ; la voiture participant à un conflit est choisie selon une probabilité proportionnelle au nombre de contraintes qu'elle viole. La séquence initiale de voitures à partir de laquelle la recherche locale est effectuée est une permutation aléatoire de l'ensemble des voitures à produire.

PV (pour « Petit Voisinage ») [EGN05] est l'algorithme qui a remporté le challenge ROADEF [NC05], challenge qui a vu s'affronter 27 équipes de chercheurs du monde entier pour la résolution d'un problème d'or-

TAB. 5.2 – Résultats expérimentaux de *IDWalk*, *ACO-seq+classes* et *PV* sur les instances faciles. Pour chaque algorithme, on donne le temps CPU moyen en secondes (suivi de l'écart-type entre parenthèses) pour trouver une solution ; pour *IDWalk* et *PV* on donne le nombre de centaines de mouvements effectués pour trouver une solution tandis que pour *ACO-seq+classes* on donne le nombre de séquences construites.

	IDWalk		*ACO-seq+classes*		*PV*	
	Temps	$\frac{\#\text{mvt}}{100}$	Temps	# seq	Temps	$\frac{\#\text{mvt}}{100}$
25 instances à 100 voitures	2.2 (1.4)	282	0.2 (0.1)	1692	1.1 (0.7)	82
13 instances à 300 voitures	11.2 (4.5)	2907	0.2 (0.1)	421	2.4 (0.8)	193
17 instances à 500 voitures	21.4 (9.2)	5278	4.5 (1.0)	5482	5.3 (2.0)	350

donnancement de voitures posé par Renault. Ce problème intègre, en plus des contraintes de ratio liées aux capacités des stations de montage, des contraintes liées aux couleurs visant à minimiser la consommation de solvant. Le problème d'ordonnancement de voitures « classique » auquel nous nous sommes intéressés est donc un cas particulier du problème du challenge, et les algorithmes conçus pour le challenge peuvent être utilisés pour résoudre les instances classiques. L'algorithme qui a remporté le challenge s'inspire de l'algorithme par recherche locale de [GPS03] : partant d'une séquence initiale calculée à l'aide d'un algorithme glouton similaire à *Glouton*, l'algorithme choisit à chaque itération le premier voisin de la configuration courante qui ne détériore pas son coût. Le voisinage considéré est défini par un ensemble de cinq transformations possibles (échange de 2 voitures, insertion d'une voiture en avant ou en arrière, miroir et mélange aléatoire), la probabilité de choisir chacune de ces transformations étant respectivement de 0.6, 0.13, 0.13, 0.13 et 0.01.

Jeux d'essais. Nous considérons les instances générées par Laurent Perron [PS04] comme en 5.2.5, et nous séparons également ces instances en deux groupes en fonction de leur « difficulté », mais on considère pour cela les taux de réussite des trois algorithmes comparés *IDWalk*, *ACO-seq+classes* et *PV* : on considère qu'une instance est facile lorsque les taux de réussite sont de 100% pour les trois algorithmes. Ainsi, il y a 25 (resp. 13 et 17) instances à 100 (resp. 300 et 500) voitures qui sont faciles et 7 (resp. 8 et 12) instances à 100 (resp. 300 et 500) voitures qui sont difficiles.

Conditions expérimentales. *ACO-seq+classes* est exécuté sur un Pentium 4 cadencé à 2GHz. Il est paramétré comme en 5.2.5 de sorte qu'il construit au maximum $5000 * 30$ séquences, ce qui prend en moyenne 22, 53 et 123 secondes pour les instances ayant respectivement 100, 300 et 500 voitures. L'algorithme a été exécuté 50 fois sur chaque instance.

IDWalk [NTG04] est exécuté sur une machine cadencée à 2.2GHz et chaque exécution a été limitée à 30, 60 et 130 secondes pour les instances ayant respectivement 100, 300 et 500 voitures. L'algorithme a été exécuté 13 fois sur chaque instance.

PV [EGN05] est exécuté sur une machine cadencée à 3GHz et chaque exécution a été limitée à 20, 40 et 90 secondes pour les instances ayant respectivement 100, 300 et 500 voitures. L'algorithme a été exécuté 13 fois sur chaque instance.

Comparaison sur les instances faciles. Le tableau 5.2 donne les résultats obtenus par chacun des trois algorithmes sur les instances faciles, i.e., les instances pour lesquelles toutes les exécutions de tous les algorithmes ont trouvé une solution. Pour ces instances, on constate que *ACO-seq+classes* est plus rapide que *PV* qui est lui-même plus rapide que *IDWalk*, mais pour les instances à 500 voitures, les temps de *PV* se rapprochent de ceux de *ACO-seq+classes*.

TAB. 5.3 – Résultats expérimentaux de *IDWalk*, *ACO-seq+classes* et *PV* sur les instances difficiles. Chaque ligne donne le numéro de l'instance, suivi pour chaque algorithme du pourcentage d'exécutions ayant trouvé une solution, du temps CPU moyen en secondes (suivi de l'écart-type entre parenthèses) pour trouver une solution ; pour *IDWalk* et *PV* on donne le nombre de centaines de mouvements effectués pour trouver une solution tandis que pour *ACO-seq+classes* on donne le nombre de séquences construites.

	IDWalk			*ACO-seq+classes*			*PV*		
	%	Temps	$\frac{\#mvt}{100}$	%	Temps	# seq	%	Temps	$\frac{\#mvt}{100}$
100-19	0			84	8.9 (4.3)	70186	15	17.5 (0.7)	475
100-21	77	22.1 (8.5)	6146	98	2.7 (2.5)	21902	92	7.2 (5.3)	256
100-60	0			100	0.9 (0.3)	7395	15	14.1 (4.3)	315
100-85	0			24	9.5 (4.1)	83962	54	8.5 (4.9)	671
100-86	77	17.5 (8.9)	4093	52	6.5 (4.8)	53867	92	6.5 (4.4)	216
100-98	0			2	26.5 (0.0)	144280	0		
100-99	77	16.3 (5.7)	9037	100	0.0 (0.0)	345	100	3.3 (3.0)	230
300-08	0			100	0.1 (0.1)	380	100	21.0 (9.0)	1847
300-25	92	14.5 (3.7)	4813	100	0.3 (0.3)	924	100	3.6 (1.6)	370
300-36	23	39.2 (10.2)	17290	100	0.0 (0.0)	5	100	4.7 (2.9)	717
300-40	0			82	16.0 (11.5)	44475	38	30.1 (6.9)	1353
300-45	77	26.4 (16.0)	7657	100	2.9 (1.1)	8422	85	13.3 (9.1)	633
300-47	54	35.7 (21.9)	10800	100	0.0 (0.0)	12	100	6.8 (3.0)	880
300-78	92	20.3 (11.6)	5383	100	0.3 (0.2)	727	100	4.7 (1.8)	316
300-81	0			100	14.7 (10.3)	41643	69	24.2 (6.6)	2175
500-08	92	47.8 (27.9)	26520	100	0.1 (0.0)	66	100	6.6 (1.8)	420
500-13	38	97.5 (17.4)	35310	54	38.2 (27.2)	46089	100	13.5 (6.1)	818
500-27	100	26.3 (16.8)	9122	24	42.5 (21.2)	48438	100	11.0 (2.9)	1099
500-30	62	54.3 (37.8)	8346	100	0.0 (0.0)	64	100	10.7 (3.9)	528
500-34	100	18.0 (2.7)	2604	24	43.7 (24.5)	52770	100	9.3 (3.9)	470
500-44	8	66.9 (0.0)	23440	100	0.1 (0.1)	107	100	10.8 (7.2)	869
500-53	0			100	1.7 (1.3)	2174	100	30.2 (13.8)	2951
500-56	85	36.1 (23.8)	11660	100	1.1 (1.1)	1344	100	19.7 (5.5)	602
500-65	0			100	0.1 (0.1)	249	100	25.7 (9.3)	977
500-74	69	70.6 (31.2)	30240	100	0.4 (0.4)	432	100	8.2 (1.9)	669
500-77	92	69.5 (24.3)	16140	100	0.2 (0.2)	326	100	9.6 (3.4)	712
500-88	100	26.3 (10.5)	6945	22	18.5 (15.2)	26741	100	10.8 (3.5)	826

Quand on compare les algorithmes à base de recherche locale sur le nombre de mouvements effectués, on remarque que *PV* fait moins de mouvements que *IDWalk*, ce qui s'explique probablement d'une part par le fait que *PV* démarre sa recherche à partir d'une solution générée par un algorithme glouton alors que *IDWalk* la démarre d'une solution générée aléatoirement, et d'autre part par le fait que les mouvements de *PV* sont plus sophistiqués que ceux de *IDWalk* (qui ne considère que l'échange de deux voitures).

Comparaison sur les instances difficiles. Le tableau 5.3 donne les résultats obtenus par chacun des trois algorithmes sur les instances difficiles. Pour ces instances, on constate qu'il n'y a pas une méthode meilleure que les autres sur toutes les instances. En général, *ACO-seq+classes* et *PV* sont meilleurs que *IDWalk* mais *IDWalk* est meilleur que *ACO-seq+classes* pour trois instances à 500 voitures (500-27, 500-34 et 500-88).

Lorsque l'on compare les performances de *ACO-seq+classes* avec celles de *PV*, on constate que pour 7 (resp. 6) instances *ACO-seq+classes* a un meilleur (resp. moins bon) taux de réussite que *PV*. En revanche, *ACO-seq+classes* est souvent beaucoup plus rapide que *PV*, d'autant plus que la machine sur laquelle *PV* a été exécuté est plus puissante.

On constate cependant des différences entre ACO d'une part et les deux approches par recherche locale d'autre part en ce qui concerne le passage à l'échelle. En effet, pour *IDWalk* et *PV* la complexité en temps d'un mouvement ne dépend pas du nombre de voitures à séquencer : chaque mouvement est évalué « localement » en considérant à chaque fois les voitures adjacentes aux voitures concernées par le mouvement de sorte que la complexité d'un mouvement ne dépend que de la fonction q définissant le nombre de voitures sur lesquelles on doit vérifier la contrainte de capacité. Ainsi, *IDWalk* (resp. *PV*) font en moyenne 34481, 33757 et 33183 (resp. 3788, 7648 et 6587) mouvements par seconde pour les instances ayant respectivement 100, 300 et 500 voitures. En revanche, la complexité en temps de *ACO-seq+classes* dépend à la fois du nombre de voitures à séquencer et du nombre de classes de voitures différentes : s'il y a $|C| = n$ voitures à séquencer et $|classes(C)| = k$ classes de voitures différentes, alors la complexité de la construction d'une séquence par une fourmi est en $\mathcal{O}(n \cdot k)$, la complexité du dépot phéromonal est en $\mathcal{O}(n)$ et la complexité de l'évaporation est en $\mathcal{O}(n^2)$. Ainsi, *ACO-seq+classes* construit en moyenne 6944, 2816 et 1220 séquences par seconde pour les instances ayant respectivement 100, 300 et 500 voitures. Cette progression laisse penser que pour des instances ayant un plus grand nombre de voitures, *IDWalk* et *PV* pourraient devenir plus rapides qu'*ACO-seq+classes*.

Ces résultats doivent également être modulés par le fait qu'*IDWalk* est une méta-heuristique généraliste qui n'a pas été spécialement conçue pour résoudre le problème d'ordonnancement de voitures et qui gagnerait probablement en performances si elle intégrait des heuristiques propres au problème considéré, notamment pour générer la configuration initiale à partir de laquelle la recherche locale est effectuée (ce que fait *PV*). De même, *PV* a été mis au point pour résoudre les instances du Challenge ROADEF et les résultats donnés ici ont été obtenus avec le même paramétrage (en ce qui concerne les probabilités de transformation notamment) que pour le challenge. Par ailleurs, la limite de temps de 20 secondes pour les instances à 100 voitures apparait trop courte pour cette approche, et si on augmente le temps maximum d'exécution à 180 secondes, alors *PV* a un taux de réussite de 100% pour les instances 100-21, 100-85 et 100-86 tandis que les taux de réussite de 100-19 et 100-60 passent à 85% et 23% respectivement. En revanche, l'instance 100-98 n'est toujours pas résolue, par aucune des 13 exécutions.

Une conclusion naturelle de ces résultats expérimentaux est que, ACO et recherche locale ayant des performances complémentaires, il serait certainement intéressant de les hybrider : les fourmis construisent des solutions en exploitant la phéromone, et la recherche locale améliore la qualité des solutions construites par les fourmis. De fait, une telle hybridation entre ACO et recherche locale s'est avérée particulièrement fructueuse pour de nombreux problèmes d'optimisation combinatoire [DG97, SH00, SF05].

Enfin, même si cela est rarement pris en compte pour comparer des approches, on peut souligner la simplicité de l'algorithme ACO, qui est très facile à implémenter : le code C correspondant à *ACO-seq+classes* comporte moins de 300 lignes tout compris. Cela n'est généralement pas le cas pour les approches par recherche locale qui doivent mettre en œuvre des structures de données évoluées afin d'évaluer le plus incrémentalement possible le coût de chaque mouvement.

5.3 Sélection de sous-ensembles par colonies de fourmis

5.3.1 Motivations

Beaucoup de problèmes résolus par la méta-heuristique ACO sont des problèmes d'ordonnancement qui se modélisent assez naturellement sous la forme de recherche de meilleurs chemins hamiltoniens dans un graphe

—où les fourmis doivent visiter chaque sommet du graphe— e.g., les problèmes de voyageur de commerce [DMC96], d'affectation quadratique [SH00], de routage de véhicules [BHS99] ou d'ordonnancement de voitures [GPS03]. Pour résoudre de tels problèmes de recherche de meilleurs chemins hamiltoniens avec la méta-heuristique ACO, on associe généralement une trace de phéromone $\tau(i, j)$ à chaque arc (i, j) du graphe. Cette trace de phéromone représente l'expérience de la colonie concernant l'intéret de visiter le sommet j juste après le sommet i, et elle est utilisée pour guider les fourmis dans leur phase de construction de chemins.

Cependant, de nombreux problèmes combinatoires se ramènent à un problème de sélection plutôt que d'ordonnancement : étant donné un ensemble d'objets S, le but de ces problèmes est de trouver un sous-ensemble de S qui satisfait certaines propriétés et/ou optimise une fonction objectif donnée. Nous appellerons ces problèmes des problèmes de sélection de sous-ensembles, abrégé par SS-problèmes. Quelques exemples classiques de SS-problèmes sont, par exemple, le problème de la clique maximum, le problème du sac-à-dos multidimensionnel, le problème de la satisfiabilité de formules booléennes, et le problème de satisfaction de contraintes.

Pour ces problèmes, les solutions sont des sous-ensembles d'objets, et l'ordre dans lequel les objets sont sélectionnés n'est pas significatif. Par conséquent, cela n'a pas beaucoup de sens de les modéliser comme des problèmes de recherche de chemins et de déposer des traces de phéromone sur des couples de sommets visités consécutivement. Deux structures phéromonales différentes peuvent être considérées pour résoudre un SS-problème à l'aide de la méta-heuristique ACO ;

- On peut associer une trace de phéromone $\tau(i)$ à chaque objet $i \in S$, de telle sorte que $\tau(i)$ représente l'expérience passée de la colonie concernant l'intéret de sélectionner l'objet i. Cette structure phéromonale a été expérimentée sur différents SS-problèmes, comme par exemple des problèmes de sac-à-dos multidimensionnels [LM99], de recouvrement d'ensembles [HRTB00], de satisfaction de contraintes [TB05], de cliques maximums [SF05], et de recherche de k-arbres de poids minimum [Blu02].
- On peut associer une trace de phéromone $\tau(i, j)$ à chaque paire d'objets différents $(i, j) \in S \times S$, de telle sorte que $\tau(i, j)$ représente l'expérience passée de la colonie concernant l'intéret de sélectionner les objets i et j dans un même sous-ensemble. Cette structure phéromonale a été expérimentée sur différents SS-problèmes : des problèmes de recherche de cliques maximums [FS03], des problèmes de sac-à-dos multidimensionnels [ASG04], des problèmes de satisfaction de contraintes [Sol02a], des problèmes de recherche de k-arbres de poids minimum [Blu02], et des problèmes d'appariement de graphes [SSG05].

Notre but ici est de présenter ces différents algorithmes dans un cadre unifié, et nous introduisons donc un algorithme ACO générique pour les SS-problèmes. Cet algorithme est paramétré d'une part par les caractéristiques du SS-problème à résoudre, et d'autre part par la structure phéromonale considérée —déterminant si les fourmis déposent la phéromone sur des objets ou sur des paires d'objets.

5.3.2 Problèmes de sélection de sous-ensembles

Les problèmes de sélection de sous-ensembles (SS-problèmes) ont pour but de trouver un sous-ensemble consistant et optimal d'objets. Plus formellement, un SS-problème est défini par un triplet $(S, S_{consistant}, f)$ tel que

- S est un ensemble d'objets ;
- $S_{consistant} \subseteq \mathcal{P}(S)$ est l'ensemble de tous les sous-ensembles de S qui sont consistants ;
  Afin de pouvoir construire chaque sous-ensemble de $S_{consistant}$ de façon incrémentale (en partant de l'ensemble vide et en ajoutant à chaque itération un objet choisi parmi l'ensemble des objets consistants avec l'ensemble en cours de construction), on impose la contrainte suivante : pour chaque sous-ensemble consistant non vide $S' \in S_{consistant}$, il doit exister au moins un objet $o_i \in S'$ tel que $S' - \{o_i\}$ est aussi consistant.
- $f : S_{consistant} \rightarrow I\!R$ est la fonction objectif qui associe un coût $f(S')$ à chaque sous-ensemble d'objets consistant $S' \in S_{consistant}$.

Le but d'un SS-problème $(S, S_{consistant}, f)$ est de trouver $S^* \in S_{consistant}$ tel que $f(S^*)$ soit maximal.

Les paragraphes suivants décrivent un certain nombre de SS-problèmes selon ce formalisme $(S, S_{consistant}, f)$.

Le problème de la clique maximum a pour but de trouver le plus grand sous-ensemble de sommets connectés deux à deux dans un graphe :
- S contient l'ensemble des sommets du graphe ;
- $S_{consistant}$ contient toutes les cliques du graphe, i.e., tout ensemble $S' \subseteq S$ tel que chaque paire de sommets distincts de S' soit connectée par une arête du graphe ;
- f est la fonction cardinalité.

Le problème du sac-à-dos multidimensionnel a pour but de trouver un sous-ensemble d'objets qui maximise un profit total tout en satisfaisant un certain nombre de contraintes de capacité :
- S contient l'ensemble des objets ;
- $S_{consistant}$ contient tous les sous-ensembles de S qui respectent les contraintes de capacité, i.e.,

$$S_{consistant} = \{S' \subseteq S \mid \forall i \in 1..m, \sum_{j \in S'} r_{ij} \leq b_i\}$$

  où m est le nombre de ressources, r_{ij} est la quantité de ressource i consommée par l'objet j, et b_i est la quantité de ressource i disponible au total ;
- f évalue le profit total, i.e., $\forall S' \in S_{consistant}, f(S') = \sum_{j \in S'} p_j$ où p_j est le profit associé à l'objet j.

Le problème de la maximisation de la satisfiabilité de formules booléennes (Max-SAT) a pour but de trouver une affectation d'un ensemble de variables booléennes satisfaisant un plus grand nombre de formules booléennes, i.e.,
- S contient l'ensemble des variables booléennes ;
- $S_{consistant}$ contient tous les sous-ensembles de S, i.e., $S_{consistant} = \mathcal{P}(S)$;
- f évalue le nombre de formules satisfaites, i.e., $\forall S' \in S_{consistant}, f(S')$ est le nombre de formules booléennes qui sont satisfaites lorsque les variables de S' sont affectées à vrai et toutes les autres à faux.

Le problème de la satisfiabilité de formules booléennes (SAT) est un cas particulier de MaxSAT dont le but est de trouver une affectation qui satisfait toutes les formules. Ce problème peut être modélisé comme un SS-problème en définissant f comme étant la fonction qui retourne 1 si toutes les formules sont satisfaites et 0 sinon.

Les problèmes de satisfaction maximale de contraintes (Max-CSP) ont pour but de trouver une affectation de valeurs à des variables qui satisfait un maximum de contraintes, i.e.,
- S contient chaque étiquette $\langle X_i, v_i \rangle$ associant une variable X_i à une valeur v_i appartenant au domaine de X_i ;
- $S_{consistant}$ contient tous les sous-ensembles de S qui n'ont pas deux étiquettes différentes pour une même variable, i.e.,

$$S_{consistant} = \{S' \subseteq S \mid \forall (\langle X_i, v_i \rangle, \langle X_j, v_j \rangle) \in S' \times S', X_i = X_j \Rightarrow v_i = v_j\}$$

- f évalue le nombre de contraintes satisfaites, i.e., $\forall S' \in S_{consistant}, f(S')$ est le nombre de contraintes telles que chaque variable impliquée dans la contrainte est affectée à une valeur par une étiquette de S' et la contrainte est satisfaite par cette affectation.

Comme pour le problème SAT, les problèmes de satisfaction de contraintes (CSPs) sont des cas particuliers de MaxCSPs et peuvent être facilement modélisés sous la forme de SS-problèmes.

Le problème de la couverture minimale de sommets a pour but de trouver le plus petit ensemble de sommets d'un graphe qui contienne au moins un sommet de chaque arête du graphe. Comme il s'agit là d'un problème de minimisation, on considère en fait le problème dual consistant à chercher le plus grand ensemble de sommets tel qu'aucune arête n'ait ses deux sommets dans l'ensemble, i.e.,
- S contient l'ensemble des sommets du graphe ;
- $S_{consistant}$ contient tout sous-ensemble $S' \subseteq S$ tel que pour toute arête (i,j) du graphe, $i \notin S'$ ou $j \notin S'$;
- f est la fonction cardinalité.

Le problème de la recherche du plus grand sous-graphe commun a pour but de rechercher le sous-graphe commun à deux graphes ayant le plus grand nombre de sommets, i.e.,
- S contient tout couple de sommets (i,j) tel que i est un sommet du premier graphe et j est un sommet du deuxième graphe ;
- $S_{consistant}$ contient tous les sous-ensembles définissant un sous-graphe commun aux deux graphes, i.e., n'appariant pas un même sommet d'un graphe à plusieurs sommets différents de l'autre graphe, et permettant de retrouver les arcs des deux graphes :

$$S_{consistant} = \{S' \subseteq S / \quad \begin{array}{l} \forall((i,j),(k,l)) \in S'^2, i = k \Leftrightarrow j = l \text{ et} \\ (i,k) \text{ est une arête du premier graphe ssi } (j,l) \text{ est une arête du second graphe } \} \end{array}$$

- f est la fonction de cardinalité.

De façon assez similaire, on peut définir les autres problèmes d'appariement de graphes introduits en 4.2 sous la forme de SS-problème.

Le problème du k-arbre de poids minimum est une généralisation du problème de la recherche d'arbres couvrants minimaux, et a pour but, étant donné un graphe pondéré, de trouver le sous-arbre ayant exactement k arêtes et étant de poids minimum, i.e.,
- S contient l'ensemble des arêtes du graphe ;
- $S_{consistant}$ contient tout sous-ensemble $S' \subseteq S$ tel que $|S'| \leq k$ et S' est un arbre ;
- $f(S') = 0$ si $|S'| \neq k$, et $f(S') = \sum_{(i,j)\in S'} weight(i,j)$ sinon.

5.3.3 Un algorithme ACO générique pour les problèmes de sélection de sous-ensembles

L'algorithme ACO générique, appelé *Ant-SS*, est paramétré par :
- le SS-problème à résoudre, décrit par un triplet $(S, S_{consistant}, f)$;
  des instanciations de cet algorithme pour résoudre des problèmes de cliques maximums, de sac-à-dos et de satisfaction de contraintes sont décrites en 5.3.5.
- une stratégie phéromonale qui détermine l'ensemble des composants phéromonaux sur lesquels les fourmis déposent des traces de phéromone, et la façon d'exploiter et de renforcer ces traces ;
  deux stratégies phéromonales différentes sont décrites en 5.3.4.
- un ensemble de paramètres numériques, dont le rôle et l'initialisation sont discutés en 5.4.

L'algorithme suit le cadre algorithmique général du $\mathcal{MAX} - \mathcal{MIN}$ *Ant System* rappelé dans la figure 5.1, et nous décrivons dans les deux paragraphes suivants la procédure utilisée par les fourmis pour construire une solution et la procédure de dépôt de phéromone.

Construction d'une solution par une fourmi. La figure 5.3 décrit la procédure suivie par les fourmis pour construire un sous-ensemble. Le premier objet est choisi aléatoirement ; les objets suivants sont choisis au sein de l'ensemble *Candidats* —contenant l'ensemble des objets « consistants » par rapport à l'ensemble des objets

Procédure construire-sous-ensemble
Entrée : un SS-problème $(S, S_{consistant}, f)$ et une fonction heuristique associée $\eta_{factor} : S \times \mathcal{P}(S) \rightarrow \mathbb{R}^+$;
une stratégie phéromonale et un facteur phéromonal associé $\tau_{factor} : S \times \mathcal{P}(S) \rightarrow \mathbb{R}^+$;
deux paramètres à valeurs numériques : α et β
Sortie : un sous-ensemble consistant d'objets $S' \in S_{consistant}$

1. Choisir aléatoirement le premier objet $o_i \in S$
2. $S_k \leftarrow \{o_i\}$
3. $Candidats \leftarrow \{o_j \in S \mid S_k \cup \{o_j\} \in S_{consistant}\}$
4. **Tant que** $Candidats \neq \emptyset$ **faire**
5. Choisir un objet $o_i \in Candidats$ avec la probabilité
6. $p_{o_i} = \frac{[\tau_{factor}(o_i,S_k)]^\alpha \cdot [\eta_{factor}(o_i,S_k)]^\beta}{\sum_{o_j \in Candidats}[\tau_{factor}(o_j,S_k)]^\alpha \cdot [\eta_{factor}(o_j,S_k)]^\beta}$
7. $S_k \leftarrow S_k \cup \{o_i\}$
8. Enlever o_i de $Candidats$
9. Enlever de $Candidats$ chaque objet o_j tel que $S_k \cup \{o_j\} \notin S_{consistant}$
10. **Fin tant que**

FIG. 5.3 – Algorithme de construction d'un sous-ensemble s par une fourmi

qui ont déjà été sélectionnés par la fourmi— en fonction d'une règle de transition probabiliste. Plus précisément, la probabilité p_{o_i} de sélectioner un objet $o_i \in Candidats$ pour une fourmi k ayant déjà sélectionné un sous-ensemble d'objets S_k dépend de deux facteurs :

- Le *facteur phéromonal* $\tau_{factor}(o_i, S_k)$ évalue l'expérience de la colonie en ce qui concerne l'ajout de l'objet o_i au sous-ensemble S_k sur la base des traces phéromonales qui ont été déposées précédemment sur les composants phéromonaux. La définition de ce facteur dépend de la stratégie phéromonale, comme décrit en 5.3.4.
- Le *facteur heuristique* $\eta_{factor}(o_i, S_k)$ évalue l'intéret de l'objet o_i en fonction d'informations locales à la fourmi, i.e., en fonction de la solution qu'elle a construite jusqu'ici S_k. La définition de ce facteur dépend du problème $(S, S_{consistant}, f)$ et plusieurs exemples sont donnés en 5.3.5.

Conformément à la méta-heuristique ACO, α et β sont deux paramètres qui déterminent l'importance relative de ces deux facteurs.

Une fois que chaque fourmi a construit une solution, certaines des solutions construites peuvent être éventuellement améliorées en utilisant une forme de recherche locale dépendante du problème : dans certains cas, la recherche locale est appliquée à toutes les solutions ; dans d'autres elle n'est appliquée qu'à la meilleure solution du cycle.

Dépot de phéromone. On adopte ici une stratégie élitiste, où seulement les meilleures fourmis du cycle, i.e., celles ayant trouvé les meilleures solutions du cycle, déposent de la phéromone. La quantité de phéromone déposée par ces meilleures fourmis dépend de la qualité des solutions qu'elles ont construites : elle est inversement proportionnelle à l'écart de qualité entre la solution construite et la meilleure solution construite depuis le début de l'exécution S_{best}. Plus précisément, si l'ensemble des solutions construites pendant le dernier cycle est $\{S_1, \ldots, S_{nbAnts}\}$ alors la procédure de dépôt de phéromone est la suivante :

pour toute solution $S_i \in \{S_1, \ldots, S_{nbAnts}\}$ **faire**
  si $\forall S_k \in \{S_1, \ldots, S_{nbAnts}\}, f(S_i) \geq f(S_k)$ **alors**
    pour chaque composant phéromonal c de S_i **faire**
      $\tau(c) \leftarrow \tau(c) + \frac{1}{1+f(S_{\text{best}})-f(S_i)}$

L'ensemble des composants phéromonaux associés à la solution S_i dépend de la stratégie phéromonale considérée et est décrit en 5.3.4.

5.3.4 Instanciations de l'algorithme par rapport à deux stratégies phéromonales

L'algorithme générique *Ant-SS* est paramétré par une stratégie phéromonal et nous décrivons maintenant les deux instanciations de cet algorithme : *Ant-SS(Vertex)* —où la phéromone est déposée sur les objets— et *Ant-SS(Clique)* —où la phéromone est déposée sur des paires d'objets. Pour chacune de ces deux stratégies phéromonales, nous définissons :
- l'ensemble des composants phéromonaux associés à un SS-problème $(S, S_{consistant}, f)$, i.e., l'ensemble des composants sur lesquels des traces de phéromone peuvent être déposées ;
- l'ensemble des composants phéromonaux associés à une solution $S_k \in S_{consistant}$, i.e., l'ensemble des composants phéromonaux sur lesquels de la phéromone est effectivement déposée lorsque la solution S_k est récompensée ;
- le facteur phéromonal $\tau_{factor}(o_i, S_k)$ associé à un objet o_i et à une solution partielle S_k, facteur phéromonal qui est utilisé dans la règle de transition probabiliste.

La stratégie phéromonale « Vertex ». Dans un algorithme ACO, les fourmis déposent de la phéromone sur les composants des meilleures solutions construites afin d'attirer les autres fourmis vers les zones correspondantes de l'espace de recherche. Pour les problèmes de recherche de sous-ensembles, les solutions construites par les fourmis sont des sous-ensembles d'objets, et l'ordre dans lequel les objets ont été sélectionnés n'est pas significatif. Ainsi, une première stratégie phéromonale pour ce type de problèmes consiste à déposer la phéromone sur les objets sélectionnés.
- L'ensemble des composants phéromonaux contient tous les objets de S. Intuitivement, la quantité de phéromone se trouvant sur un composant phéromonal $o_i \in S$ —$\tau(o_i)$— représente l'expérience passée de la colonie concernant l'intéret de sélectionner l'objet o_i quand on construit une solution.
- L'ensemble des composants phéromonaux associés à une solution S_k est l'ensemble des objets $o_i \in S_k$.
- Le facteur phéromonal dans la règle de transition probabiliste correspond à la quantité de phéromone se trouvant sur l'objet candidat, i.e., $\tau_{factor}(o_i, S_k) = \tau(o_i)$.

La stratégie phéromonale « Clique ». La stratégie phéromonale précédente suppose implicitement que la « désirabilité » d'un objet est relativement indépendante des objets déjà sélectionnés, ce qui n'est pas toujours le cas. Considérons par exemple un problème de satisfaction de contraintes contenant deux variables x et y pouvant prendre pour valeur 0 ou 1, de telle sorte que l'ensemble initial d'objets S contient les étiquettes $\langle x, 0\rangle$, $\langle x, 1\rangle$, $\langle y, 0\rangle$ et $\langle y, 1\rangle$. Supposons maintenant que ce problème contient la contrainte $x \neq y$. Dans ce cas, la « désirabilité » de sélectionner une étiquette pour y, à savoir $\langle y, 0\rangle$ ou $\langle y, 1\rangle$, dépend de l'étiquette déjà sélectionnée pour x (et inversement). Ici, les traces de phéromone pourraient être utilisées pour « apprendre » que $(\langle x, 0\rangle, \langle y, 1\rangle)$ et $(\langle x, 1\rangle, \langle y, 0\rangle)$ sont des paires d'étiquettes qui vont bien ensemble tandis que $(\langle x, 0\rangle, \langle y, 0\rangle)$ et $(\langle x, 1\rangle, \langle y, 1\rangle)$ sont des paires d'étiquettes qui sont moins intéressantes.

Ainsi, une seconde stratégie phéromonale pour les problèmes de sélection de sous-ensembles consiste à déposer de la phéromone sur des paires d'objets.
- L'ensemble des composants phéromonaux contient toutes les paires d'objets différents $(o_i, o_j) \in S \times S$. Intuitivement, la quantité de phéromone se trouvant sur un composant phéromonal $(o_i, o_j) \in S \times S$ —$\tau(o_i, o_j)$— représente l'expérience passée de la colonie concernant l'intéret de sélectionner les objets o_i et o_j dans une même solution.
- L'ensemble des composants phéromonaux associés à une solution S_k est l'ensemble des paires d'objets différents $(o_i, o_j) \in S_k \times S_k$. Autrement dit, la clique d'arêtes associée à S_k est récompensée.

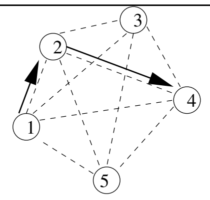
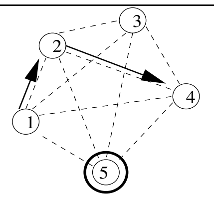
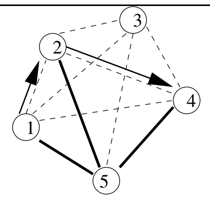
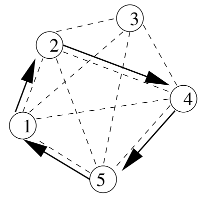
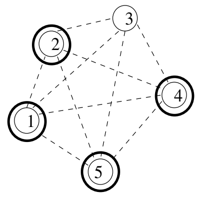
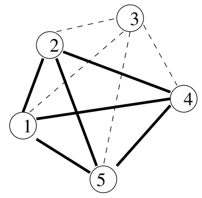
	Chemin de la fourmi	*Ant-SS(Vertex)*	*Ant-SS(Clique)*
Facteur phéromonal d'un sommet par rapport à un début de parcours	 $S_k = \{1, 2, 4\}$ **Fig. 5.4a.**	 $\tau_{factor}(5, S_k) = \tau(5)$ **Fig. 5.4b.**	 $\tau_{factor}(5, S_k) =$ $\tau(1,5) + \tau(2,5) + \tau(4,5)$ **Fig. 5.4c.**
Composants phéromonaux récompensés par rapport à un parcours complet	 **Fig. 5.4d.**	 **Fig. 5.4e.**	 **Fig. 5.4f.**

FIG. 5.4 – Stratégies phéromonales

– Le facteur phéromonal dans la règle de transition probabiliste dépend de la quantité de phéromone se trouvant entre l'objet candidat o_i et l'ensemble des objets déjà sélectionnés dans S_k, i.e.,

$$\tau_{factor}(o_i, S_k) = \sum_{o_j \in S_k} \tau(o_i, o_j)$$

Notons que ce facteur phéromonal peut être calculé de façon incrémentale : une fois que le premier objet o_i a été choisi aléatoirement, pour chaque objet candidat o_j, le facteur phéromonal $\tau_{factor}(o_j, S_k)$ est initialisé à $\tau(o_i, o_j)$; ensuite, à chaque fois qu'un nouvel objet o_l est ajouté à S_k, pour chaque objet candidat o_j, le facteur phéromonal $\tau_{factor}(o_j, S_k)$ est incrémenté de $\tau(o_l, o_j)$.

Comparaison des deux stratégies. Les deux stratégies phéromonales ont été introduites indépendamment d'un graphe de construction et du parcours d'une fourmi dans ce graphe. Nous illustrons et comparons maintenant les deux stratégies selon ce point de vue. Dans les deux cas, le graphe de construction est un graphe complet non-orienté associant un sommet à chaque objet, et le résultat d'un parcours d'une fourmi dans ce graphe est l'ensemble des sommets par lesquels elle est passée. La figure 5.4a représente un début de parcours pour une fourmi ayant successivement sélectionné les sommets 1, 2 et 4. Dans la figure 5.4b et 5.4c, on représente en gras les composants phéromonaux respectivement considérés par la fourmi dans *Ant-SS(Vertex)* et *Ant-SS(Clique)* pour définir sa probabilité de choisir l'objet 5 : alors que le facteur phéromonal dans *Ant-SS(Vertex)* ne dépend que de la quantité de phéromone se trouvant sur le sommet candidat, le fateur phéromonal dans *Ant-SS(Clique)* dépend de toutes les traces de phéromone déposées entre l'objet candidat et l'ensemble courant d'objets sélectionnés.

La figure 5.4d représente ensuite le parcours complet d'une fourmi, l'ensemble d'objets sélectionnés correspondant étant $\{1, 2, 4, 5\}$. Dans les figures 5.4e et 5.4f, on représente en gras les composants phéromonaux sur lesquels la phéromone sera respectivement déposée par *Ant-SS(Vertex)* et *Ant-SS(Clique)*. Notons que pour *Ant-SS(Clique)* toutes les arêtes appartenant à la clique des sommets visités sont récompensées (et non seulement celles qui ont été « traversées » lors de la construction de la solution).

En termes de complexité algorithmique, *Ant-SS(Vertex)* et *Ant-SS(Clique)* exécutent à peu près le même nombre d'opérations pour construire une solution (algorithme de la figure 5.3). La seule différence réside dans le calcul des facteurs phéromonaux et, comme nous l'avons souligné précédemment, dans *Ant-SS(Clique)* les facteurs phéromonaux peuvent être calculés de façon incrémentale : à chaque fois qu'un objet est sélectionné le facteur phéromonal de chaque objet candidat peut être mis-à-jour par une simple addition, de sorte que les complexités en temps des deux instanciations sont les mêmes (dans la mesure où les objets appartenant à l'ensemble *Candidats* doivent être énumérés à chaque itération afin de déterminer s'ils sont encore candidats à l'itération suivante).

Cependant, pour mettre à jour les traces de phéromone *Ant-SS(Vertex)* et *Ant-SS(Clique)* ne font pas le même nombre d'opérations : dans *Ant-SS(Vertex)*, l'étape d'évaporation est en $\mathcal{O}(\mid S \mid)$ et la récompense d'un sous-ensemble S_k est en $\mathcal{O}(\mid S_k \mid)$, tandis que dans *Ant-SS(Clique)*, l'étape d'évaporation est en $\mathcal{O}(\mid S \mid^2)$ et la récompense d'un sous-ensemble S_k est en $\mathcal{O}(\mid S_k \mid^2)$.

5.3.5 Instanciations de l'algorithme par rapport à différents problèmes

Pour résoudre un problème particulier de sélection de sous-ensembles avec *Ant-SS*, il s'agit essentiellement de définir le facteur heuristique $\eta_{factor}(o_i, S_k)$, qui évalue localement l'intéret de l'objet o_i par rapport à la solution courante S_k et qui est utilisé dans la règle de transition probabiliste. Optionnellement, une procédure de recherche locale peut être définie afin d'améliorer les solutions construites.

On illustre ici la généricité de *Ant-SS* à travers trois problèmes de sélection de sous-ensembles différents : le problème de la clique maximum, le problème du sac-à-dos et les problèmes de satisfaction de contraintes.

Le problème de la clique maximum

Facteur heuristique. Curieusement, pour le problème de la clique maximum il s'est avéré qu'il est généralement préférable de ne pas utiliser de fateur heuristique, i.e., $\eta_{factor}(o_i, S_k) = 1$ de telle sorte que les fourmis choisissent les objets sur la base du facteur phéromonal uniquement.

En effet, les heuristiques classiques pour ce problème consistent à favoriser les sommets ayant un degré important dans le graphe « résiduel », i.e., le sous-graphe induit par l'ensemble des sommets candidats, la motivation étant que plus le sommet sélectionné a un degré important, plus il y aura de candidats pour entrer dans la clique à l'itération suivante. Quand on n'utilise pas la phéromone (dans un algorithme de construction incrémentale gloutonne classique par exemple), cette heuristique permet clairement d'obtenir des cliques plus grandes qu'un choix aléatoire [BP01]. Cependant, quand on utilise un mécanisme de rétroaction phéromonal, on obtient de plus grandes cliques en fin d'exécution lorsque l'on n'utilise pas cette heuristique. De fait, des expérimentations sur différentes instances de ce problème nous ont montré que l'utilisation de l'heuristique a pour conséquence d'augmenter très fortement le taux de ré-échantillonnage, i.e., le pourcentage de solutions qui sont re-calculées. Cela montre que l'heuristique attire trop fortement la recherche autour d'un ensemble de cliques localement optimales, de sorte que la recherche n'est plus assez diversifiée pour trouver des cliques plus grandes.

Recherche locale. Différentes procédures de recherche locale peuvent être considérées. Nous avons plus particulièrement utilisé la procédure basée sur un échange $(2, 1)$ utilisée dans *GRASP* [APR99] : étant donnée une solution (clique) S_k, chaque mouvement consiste à rechercher trois objets (sommets) o_i, o_j et o_k tels que (i) $o_i \in S_k$; (ii) $o_j \notin S_k$ et $o_l \notin S_k$; et (iii) o_j et o_l sont adjacents à chaque sommet de $S_k - \{o_i\}$. On remplace alors o_i par o_j et o_l, augmentant ainsi de un la taille de la clique. Ces mouvements sont itérativement répétés jusqu'à obtenir une clique localement optimale, i.e., ne pouvant plus être améliorée par un tel échange $(2, 1)$.

Cette procédure de recherche locale est appliquée sur la meilleure clique du cycle seulement. En effet, nous avons constaté que si on l'applique à toutes les cliques construites pendant le cycle, la qualité de la solution finale n'est pas vraiment améliorée tandis que le temps d'exécution est significativement augmenté.

Le problème du sac-à-dos multidimensionnel

Facteur heuristique. Le même facteur heuristique a été utilisé dans les deux algorithmes ACO proposés pour ce problème [LM99, ASG04] : soit $c_{S_k}(i) = \sum_{j \in S_k} r_{i,j}$ la quantité de ressource i consommée par les objets appartenant à la solution S_k, et soit $d_{S_k}(i) = b_i - c_{S_k}(i)$ la capacité restante de ressource i ; le ratio suivant

$$h_{S_k}(j) = \sum_{i=1}^{m} \frac{r_{ij}}{d_{S_k}(i)}$$

représente la dureté de l'objet j, i.e., la quantité de ressources consommées par j par rapport à leur capacité restante. Plus ce ratio est faible et meilleur est l'objet j. On définit alors le facteur heuristique de l'objet j en incorporant son profit :

$$\eta_{factor}(j, S_k) = \frac{p_j}{h_{S_k}(j)}$$

Recherche locale. Pour le moment, il n'y a pas de résultats sur l'utilisation de procédures de recherche locale dans des algorithmes ACO pour ce problème. Nous envisageons de tester cela prochainement, en s'inspirant de la procédure de recherche locale proposée par [GK96].

Problèmes de satisfaction maximale de contraintes

Facteur heuristique. Le facteur heuristique utilisé dans [Sol02a] est déterminé en deux étapes :

1. on sélectionne tout d'abord une variable X_j parmi l'ensemble des variables non instanciées, i.e., $\{X_i \mid \langle X_i, v_i \rangle \in Candidats\}$, selon une heuristique de choix donnée ;
2. on calcule ensuite les facteurs heuristiques : le facteur heuristique des étiquettes $\langle X_j, v_j \rangle$ correspondant à la variable sélectionnée est inversement proportionnel au nombre de nouvelles contraintes violées par l'instanciation de X_j par v_j, tandis que le facteur heuristique des autres étiquettes (et donc leur probabilité de choix) est mis à zéro, i.e.,

$$\begin{array}{ll} \eta_{factor}(\langle X_i, v_i \rangle, S_k) = \frac{1}{1+cout(\{\langle X_i, v_i \rangle\} \cup S_k) - cout(S_k)} & \text{si } X_i = X_j \\ \eta_{factor}(\langle X_i, v_i \rangle, S_k) = 0 & \text{si } X_i \neq X_j \end{array}$$

   où $cout(S)$ est le nombre de contraintes violées par l'affectation S.

Cette procédure en deux temps est motivée par le fait qu'il peut y avoir beaucoup de candidats : s'il y a n variables non instanciées, chacune pouvant prendre m valeurs, alors il y a nm candidats. En sélectionnant d'abord une variable, on limite volontairement le nombre de candidats aux m valeurs possibles pour cette variable. Il y a par ailleurs une bonne raison de choisir la prochaine variable à instancier indépendamment de la phéromone dans le sens où il existe des heuristiques qui se sont avérées performantes quant à l'ordre d'instanciation des variables. L'heuristique que nous avons retenue est celle du « plus petit domaine d'abord » qui sélectionne à chaque itération la variable dont le domaine contient le plus petit nombre de valeurs consistantes avec l'affectation partielle en cours de construction.

Recherche locale. La procédure de recherche locale que nous avons considérée dans [Sol02a] est basée sur l'heuristique « min-conflicts » de [MJPL92]. A chaque mouvement, on choisit aléatoirement une variable impliquée dans une ou plusieurs contraintes violées et on affecte cette variable avec la valeur qui minimise le nombre de conflits. Cette procédure de réparation est arrêtée lorsque le nombre de contraintes violées n'est pas amélioré pendant un certain nombre de mouvements.

5.3.6 Résultats expérimentaux

Des résultats expérimentaux comparant les deux stratégies phéromonales de *Ant-SS(Vertex)* et *Ant-SS(Clique)* ont été publiés dans [SF05] pour le problème de la clique maximum, dans [ASG04] pour le problème du sac-à-dos multidimensionnel et dans [TB05] pour les problèmes de satisfaction de contraintes ; des résultats expérimentaux concernant *Ant-SS(Clique)* seulement ont été publiés dans [SSG05] pour des problèmes d'appariement de graphes et dans [Sol02a] pour des problèmes de satisfaction de contraintes. Nous présentons ici quelques-uns de ces résultats sur le problème de la clique maximum et les problèmes de satisfaction de contraintes.

Instances considérées. Pour le problème de la clique maximum, on décrit les résultats obtenus sur six graphes de la famille $Cn.9$ du challenge DIMACS[1]. Ces graphes ont respectivement 125, 250, 500, 1000 et 2000 sommets, et leurs plus grandes cliques connues ont respectivement 34, 44, 57, 68 et 78 sommets. Ce premier ensemble d'instances nous permet notamment d'illustrer le passage à l'échelle de *Ant-SS*.

Pour les problèmes de satisfaction de contraintes, on décrit les résultats obtenus sur quatre ensembles d'instances générées aléatoirement selon le modèle A décrit dans [MPSW98]. On ne considère ici que des instances satisfiables, dont on sait qu'elles admettent une solution. Chacun des quatre ensembles contient vingt instances différentes, chacune ayant 100 variables, des domaines de taille 8, et une connectivité égale à 0.14 (i.e., la probabilité p_1 de poser une contrainte entre deux variables est égale à 0.14). Les quatre ensembles d'instances ont été générés avec différentes valeurs pour la probabilité p_2 —déterminant la dureté de chaque contrainte en terme du nombre de couples de valeurs enlevés pour chaque contrainte. Le but est d'illustrer le comportement de *Ant-SS* dans la région de la transition de phase. Ainsi, les quatre ensembles d'instances ont été générés en fixant p_2 respectivement à 0.2, 0.23, 0.26 et 0.29 de telle sorte que le taux de difficulté κ soit respectivement égal à 0.74, 0.87, 1.00 et 1.14, la transition de phase étant située autour du point pour lequel $\kappa = 1$.

Conditions expérimentales. Pour toutes les instances, nous avons fixé le nombre de fourmis *nbAnts* à 30, α à 1, ρ à 0.99 et τ_{min} à 0.01. Pour les instances du problème de la clique maximum, nous avons fixé τ_{max} à 6 et β à 0 ; pour les instances du problème de satisfaction de contraintes, nous avons fixé τ_{max} à 4 et β à 10.

Pour les instances du problème de la clique maximum, chaque algorithme a été exécuté 50 fois sur chacune des six instances ; pour les instances du problème de satisfaction de contraintes, chaque algorithme a été exécuté 5 fois sur chacune des vingt instances des quatre ensembles d'instances (ce qui fait un total de cent exécutions par ensemble).

Toutes les exécutions ont été faites sur un Pentium 4 cadencé à 2GHz.

Comparaison de la qualité des solutions. Les deux premières colonnes du tableau 5.4 donnent la qualité des solutions trouvées par *Ant-SS(Vertex)* et *Ant-SS(Clique)* lorsque l'on n'utilise pas de recherche locale pour améliorer les solutions construites par les fourmis. Sur les instances considérées ici Ant-SS(Clique) est généralement meilleur que *Ant-SS(Vertex)* : les deux variantes sont capables de trouver les solutions optimales aux instances « faciles » (i.e., le plus petit graphe C125.9 pour la clique maximum, et les instances suffisamment éloignées de la région de la transition de phases pour les problèmes de satisfaction de contraintes). Cependant, pour les instances plus difficiles (i.e., quand on augmente la taille du graphe pour la clique maximum, ou quand on se rapproche de la région de la transition de phases pour les problèmes de satisfaction de contraintes), *Ant-SS(Clique)* obtient toujours de meilleurs résultats que *Ant-SS(Vertex)*.

[1] http ://dimacs.rutgers.edu/

TAB. 5.4 – Résultats expérimentaux : comparaison de la qualité des solutions. Chaque ligne donne la qualité des solutions obtenues avec *Ant-SS(Vertex)* et *Ant-SS(Clique)*, sans recherche locale puis avec recherche locale. Pour le problème de la clique maximum, on donne la taille moyenne des cliques trouvées suivie entre parenthèses de la taille de la plus grande clique trouvée pour les 50 exécutions ; pour les problèmes de satisfaction de contraintes, on donne le pourcentage d'exécutions qui ont trouvé une solution.

Problème de la clique maximum

		Sans recherche locale		Avec recherche locale	
Graphe	$\omega(G)$	*Ant-SS(Vertex)*	*Ant-SS(Clique)*	*Ant-SS(Vertex)*	*Ant-SS(Clique)*
C125.9	34	34.0 (34)	34.0 (34)	34.0 (34)	34.0 (34)
C250.9	44	43.9 (44)	44.0 (44)	44.0 (44)	44.0 (44)
C500.9	≥ 57	55.2 (56)	55.6 (57)	55.3 (56)	55.9 (57)
C1000.9	≥ 68	65.3 (67)	66.0 (67)	65.7 (67)	66.2 (68)
C2000.9	≥ 78	73.4 (76)	74.1 (76)	74.5 (77)	74.3 (78)

Problèmes de satisfaction de contraintes

		Sans recherche locale		Avec recherche locale	
p_t	κ	*Ant-SS(Vertex)*	*Ant-SS(Clique)*	*Ant-SS(Vertex)*	*Ant-SS(Clique)*
0.20	0.74	100%	100%	100%	100%
0.23	0.87	45%	93%	91%	100%
0.26	1.00	89%	97%	99%	100%
0.29	1.14	100%	100%	100%	100%

Les deux dernières colonnes du tableau 5.4 donnent la qualité des solutions trouvées lorsque l'on applique une procédure de recherche locale pour améliorer les solutions construites par les fourmis. On constate ici que l'intégration de la recherche locale dans *Ant-SS* améliore effectivement la qualité des solutions, pour les deux stratégies phéromonales.

Comparaison du temps CPU. Le tableau 5.5 montre que le nombre de cycles —et par conséquent le temps CPU— nécessaires pour trouver la meilleure solution dépend de la taille du problème et de sa difficulté. Par exemple, Ant-SS(Vertex) exécute respectivement 60, 359, 722, 1219 et 1770 cycles en moyenne pour résoudre les instances du problème de la clique maximum qui ont respectivement 125, 250, 500, 1000 et 2000 sommets respectivement. De la même façon, pour les problèmes de satisfaction de contraintes le nombre de cycles augmente lorsque l'on se rapproche de la région de la transition de phases.

Ce tableau montre aussi que Ant-SS(Vertex) converge en moins de cycles que Ant-SS(Clique) pour les problèmes de cliques maximums tandis qu'il fait quasiment toujours plus de cycles pour les problèmes de satisfaction de contraintes. Cependant, comme un cycle de Ant-SS(Clique) est beaucoup plus long à exécuter qu'un cycle de Ant-SS(Vertex), Ant-SS(Vertex) converge toujours plus rapidement que Ant-SS(Clique).

Notons également que le nombre de cycles est toujours diminué quand la recherche locale est utilisée pour améliorer les solutions construites par les fourmis. Cependant, comme cette étape de recherche locale prend du temps, le temps CPU global n'est pas toujours diminué.

Temps CPU vs qualité de solution. Les tableaux 5.4 et 5.5 montrent que pour choisir entre *Ant-SS(Vertex)* et *Ant-SS(Clique)*, il s'agit de prendre en compte le temps CPU disponible pour le processus de résolution. En effet, *Ant-SS(Clique)* trouve généralement de meilleures solutions que *Ant-SS(Vertex)* à la fin de son processus de résolution, mais il est aussi plus long à converger. Par conséquent, si l'on doit trouver une solution rapide-

TAB. 5.5 – Résultats expérimentaux : comparaison du temps CPU. Chaque ligne donne le nombre moyen de cycles et le temps CPU (en secondes) utilisés pour trouver la meilleure solution.

Problème de la clique maximum

	Sans recherche locale				Avec recherche locale			
	Ant-SS(Vertex)		*Ant-SS(Clique)*		*Ant-SS(Vertex)*		*Ant-SS(Clique)*	
Graphe	Cycles	Temps	Cycles	Temps	Cycles	Temps	Cycles	Temps
C125.9	60	0.1	126	0.2	14	0.0	23	0.0
C250.9	359	0.8	473	1.7	172	0.5	239	1.0
C500.9	722	3.8	923	8.9	477	4.6	671	8.6
C1000.9	1219	13.2	2359	55.0	832	23.4	1242	49.8
C2000.9	1770	41.3	3268	214.4	1427	112.4	2067	238.7

Problèmes de satisfaction de contraintes

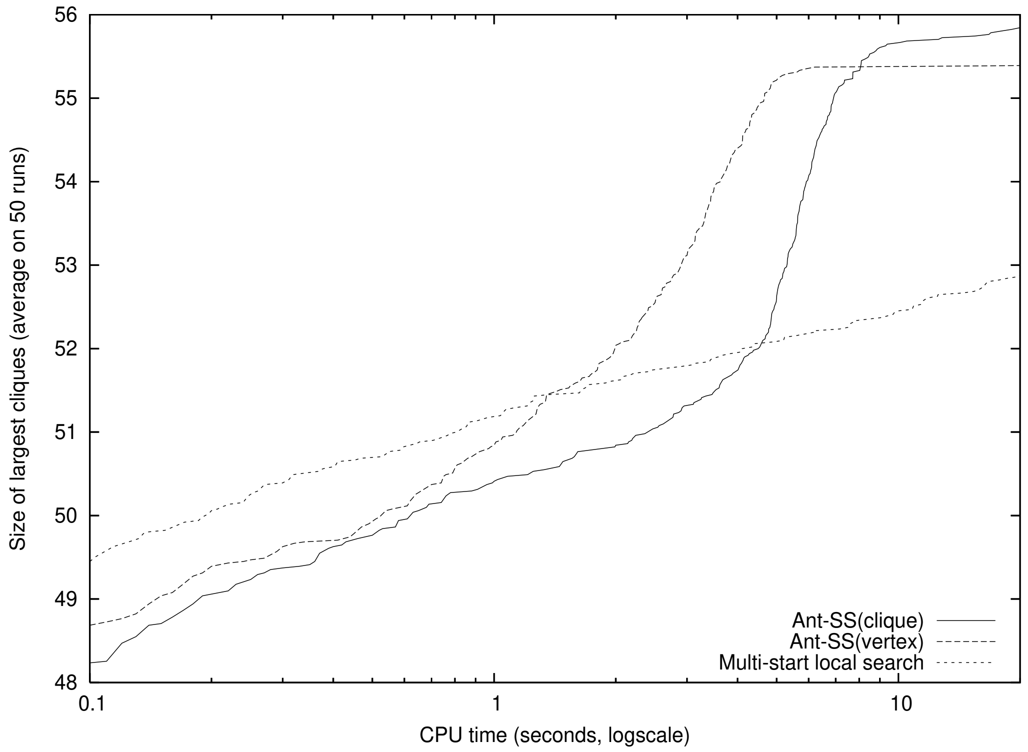
	Sans recherche locale				Avec recherche locale			
	Ant-SS(Vertex)		*Ant-SS(Clique)*		*Ant-SS(Vertex)*		*Ant-SS(Clique)*	
p_t	Cycles	Temps	Cycles	Temps	Cycles	Temps	Cycles	Temps
0.20	90	1.2	63	4.8	2	0.0	2	0.0
0.23	1084	19.5	849	78.2	472	30.8	230	31.6
0.26	1019	18.1	737	66.1	365	26.9	260	38.7
0.29	449	7.8	516	45.2	27	1.8	38	5.4

ment, il vaut mieux utiliser *Ant-SS(Vertex)*, tandis que si l'on dispose de plus de temps, il vaut mieux utiliser *Ant-SS(Clique)*. Cela est illustré dans la figure 5.5 sur le problème de la recherche d'une clique maximum pour le graphe `C500.9`. Cette figure trace l'évolution de la taille de la plus grande clique trouvée (en moyenne sur 50 exécutions) en fonction du temps CPU. On constate sur cette figure que pour des temps CPU inférieurs à 8 secondes, *Ant-SS(Vertex)* trouve de plus grandes cliques que *Ant-SS(Clique)*, tandis que pour des temps CPU plus importants *Ant-SS(Clique)* trouve de plus grandes cliques que *Ant-SS(Vertex)*.

Cette figure compare aussi *Ant-SS* avec une procédure de recherche locale « multi-start » appelée *multi-start LS*. Cette procédure de recherche locale itère sur les deux étapes suivantes : (1) construction aléatoire d'une clique maximale, et (2) amélioration de cette clique par la procédure de recherche locale décrite en 5.3.5. On remarque que pour des temps CPU inférieurs à 1 seconde, *multi-start LS* trouve de meilleures solutions que *Ant-SS*. En effet, *Ant-SS* prend du temps pour gérer la phéromone (la déposer, l'exploiter et l'évaporer) tandis que cette phéromone ne commence à influencer les fourmis qu'après quelques centaines de cycles. Par conséquent *Ant-SS(Vertex)* (resp. *Ant-SS(Clique)*) devient meilleur que *multi-start LS* seulement après une seconde (resp. cinq secondes) de temps CPU.

Comparaison avec d'autres approches pour la résolution de CSPs. Le tableau 5.4 montre que pour les CSPs binaires toutes les exécutions de *Ant-SS(Clique)*, lorsqu'il est combiné avec une procédure de recherche locale, ont réussi à trouver une solution, même pour les instances les plus difficiles. On donne dans [vHS04] plus de résultats expérimentaux et on compare les performances de *Ant-SS(Clique)* avec celles d'un algorithme génétique —l'algorithme *Glass-Box* [Mar97] qui s'est avéré le plus performant dans l'étude comparative de [CEv03]— et celles d'une approche complète —l'algorithme de [Pro93] qui combine un processus de vérification en avant (forward checking) avec des techniques de sauts en arrière dirigés par les conflits (conflict directed backjumping). On montre dans [vHS04] que *Ant-SS(Clique)* est nettement meilleur que *Glass-Box* —qui a beaucoup de mal à trouver des solutions pour les instances proches de la transition de phases. On montre également que si l'approche complète est plus rapide pour les petites instances (moins de 40 variables), ses temps d'exécution progressent de façon exponentielle lorsque l'on augmente le nombre de variables tandis

FIG. 5.5 – Comparaison des stratégies « clique » et « vertex » et d'une procédure de recherche locale « multi-start » : chaque courbe trace l'évolution de la taille de la plus grande clique trouvée (en moyenne sur 50 exécutions) au cours du temps.



que les temps d'exécution de *Ant-SS(Clique)* progressent de façon polynomiale, de sorte que *Ant-SS(Clique)* devient plus rapide dès lors que le nombre de variables devient supérieur à 40.

Ainsi, *Ant-SS(Clique)* apparait comme une approche robuste pour résoudre les CSPs binaires aléatoires dans le sens où ses taux de réussite sont très proches de 100%, même pour les instances les plus difficiles. Notons cependant que si les temps d'exécution de *Ant-SS(Clique)* progressent plus doucement que ceux des approches complètes lorsque l'on augmente la taille des instances, ils restent tout de même assez élevés et, de façon générale, ils sont plus importants que ceux des approches par recherche locale taboue comme celle, e.g., de [GH97].

Comparaison avec d'autres approches pour la recherche de cliques maximums. Pour les problèmes de recherche de cliques maximums, on compare dans [SF05] les performances de *Ant-SS* avec celles de trois approches parmi les plus performantes : une approche taboue réactive [BP01], une approche gloutonne adaptative [GLC04] et une approche combinant un algorithme génétique avec une procédure de recherche locale [Mar02]. On montre que *Ant-SS(Clique)* obtient des résultats très compétitifs : il est clairement meilleur que l'approche génétique ; il obtient des résultats comparables à ceux de l'approche gloutonne adaptative —meilleurs sur certaines instances et moins bons sur d'autres— ; il est légèrement moins bon que l'approche taboue réactive —qui est la meilleure approche, à notre connaissance, pour ce problème.

5.4 Intensification versus diversification de la recherche ACO

5.4.1 Influence des paramètres sur le processus de résolution

Quand on résoud un problème d'optimisation combinatoire avec une approche heuristique, il s'agit de trouver un bon compromis entre deux objectifs relativement duaux. Il s'agit d'une part d'intensifier la recherche autour des zones de l'espace de recherche les plus prometteuses, i.e., autour des meilleures solutions trouvées. Il s'agit par ailleurs de diversifier la recherche et de favoriser l'exploration afin de découvrir de nouvelles zones de l'espace de recherche dont on espère qu'elles contiendront de meilleures solutions. Le comportement des fourmis par rapport à cette dualité intensification/diversification peut être influencé en modifiant les valeurs des paramètres numériques.

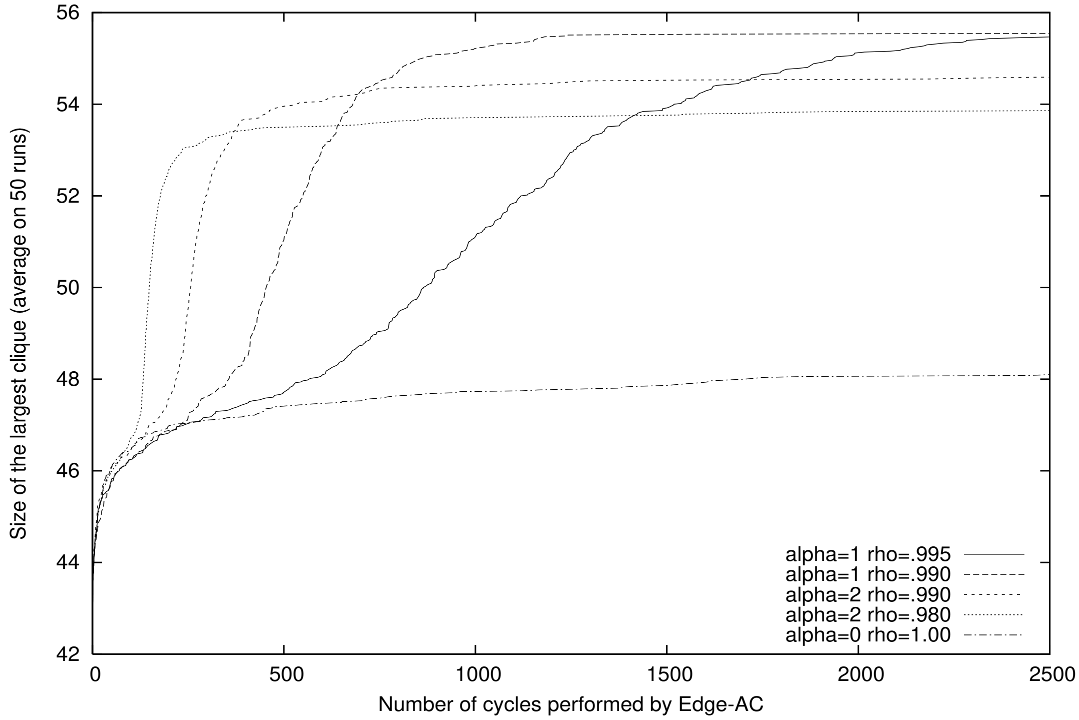
Influence de τ_{min} et τ_{max}. Ces deux paramètres ont été introduits dans le $\mathcal{MMAS}$ afin de limiter les différences relatives entre les traces de phéromone et limiter l'intensification.

Pour les différents algorithmes ACO que nous avons proposés dans [Sol02a, GPS03, FS03, ASG04, SSG05, SF05], la borne minimale τ_{min} a été fixée à 0.01. Cependant, la valeur de la borne maximale τ_{max} dépend du problème : [SH00] montre que τ_{max} doit avoir une valeur proche de $\delta_{avg}/(1-\rho)$ où δ_{avg} est la quantité moyenne de phéromone déposée sur un composant phéromonal à chaque cycle. Cette valeur varie clairement d'un problème à l'autre. Ainsi, τ_{max} a été fixé à 6 pour les problèmes de recherche de cliques maximum [FS03] et de sac-à-dos multidimensionnels [ASG04] tandis qu'il a été fixé à 4 pour les problèmes de satisfaction de contraintes [Sol02a], d'appariement de graphes [SSG05] et d'ordonnancement de voitures [GPS03].

Influence de α et ρ. Ces deux paramètres ont une influence prépondérante sur le processus de résolution. En effet, la diversification peut être accentuée soit en diminuant la valeur du poids du facteur phéromonal α —de telle sorte que les fourmis deviennent moins sensibles aux traces de phéromone et donc à l'expérience passée de la colonie— ou en augmentant la valeur du taux de persistance phéromonale ρ —de telle sorte que la phéromone s'évapore plus doucement.

Quand on augmente les capacités d'exploration des fourmis de cette façon, l'algorithme trouve généralement de meilleures solutions, mais en contrepartie il met plus de temps pour trouver ces solutions. Ainsi, l'initialisation de ces deux paramètres dépend essentiellement du temps dont on dispose pour la résolution : si le temps est limité, il vaut mieux choisir des valeurs qui favorisent une convergence rapide, comme $\alpha \in \{3, 4\}$ et $\rho < 0.99$; si en revanche le temps n'est pas limité, il vaut mieux choisir des valeurs qui favorisent l'exploration, comme $\alpha \in \{1, 2\}$ et $\rho \geq 0.99$.

Cette dualité a été observée sur tous les problèmes que nous avons résolus avec ACO. Nous l'illustrons dans la figure 5.6 sur l'instance `C500.9` du problème de recherche de cliques maximums (instance relativement difficile, pour laquelle *Ant-SS* a du mal à trouver la clique maximum qui a 57 sommets). Sur cette figure, on remarque qu'en début de résolution, aussi bien pour *Ant-SS(Clique)* que pour *Ant-SS(Vertex)*, les fourmis trouvent de meilleures solutions avec des valeurs de α et ρ favorisant l'intensification (comme par exemple $\alpha = 2$ et $\rho = 0.98$). En revanche, après un millier de cycles, les fourmis trouvent de meilleures solutions avec des valeurs de α et ρ favorisant l'exploration (comme par exemple $\alpha = 1$ et $\rho = 0.99$). On constate également que quand $\alpha = 0$ et $\rho = 1$, les meilleures cliques construites sont beaucoup plus petites : dans ce cas, la phéromone est complètement ignorée et le processus de résolution se comporte de façon purement aléatoire de telle sorte qu'après quelques centaines de cycles la taille de la meilleure clique n'augmente quasiment plus, et atteint péniblement 48 sommets. L'écart entre cette courbe et les autres permet de quantifier l'apport de la phéromone dans le processus de résolution.



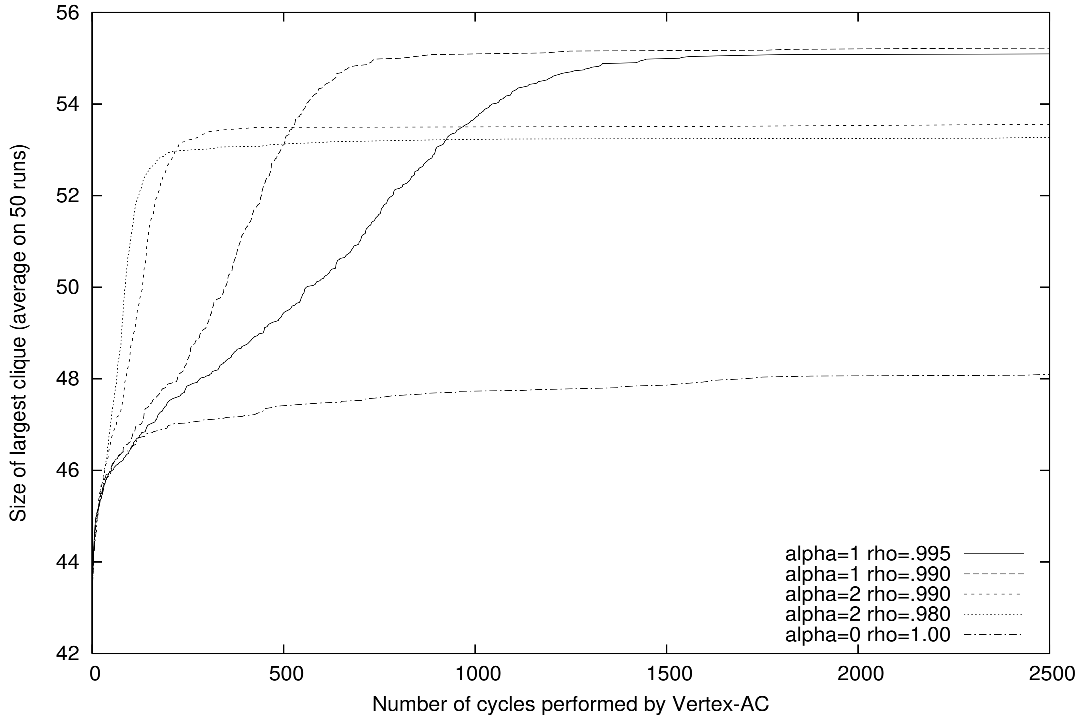


FIG. 5.6 – Influence de la phéromone sur le processus de résolution de *Ant-SS(Clique)* (courbes du haut) et *Ant-SS(Vertex)* (courbes du bas) pour le problème de la recherche d'une clique maximum dans le graphe `C500.9` : chaque courbe trace l'évolution de la taille de la meilleure clique construite (moyenne sur 50 exécutions) quand le nombre de cycles augmente, pour des valeurs données de α et ρ. Les autres paramètres ont été fixés à $nbAnts = 30$, $\tau_{min} = 0.01$, et $\tau_{max} = 6$.

Influence de $nbAnts$. Pour accentuer la diversification et éviter les phénomènes de convergence prématurée sur des optima locaux, on peut aussi augmenter le nombre de fourmis de sorte que plus de combinaisons sont explorées à chaque cycle. Pour tous les résultats expérimentaux que nous avons présentés précédemment, que ce soit pour le problème d'ordonnancement de voitures, les CSPs ou la recherche de cliques maximums, le nombre de fourmis a été fixé à 30, et nous avons constaté expérimentalement des comportements similaires par rapport à ce paramètre : lorsque l'on diminue le nombre de fourmis, la qualité des solutions est généralement dégradée ; si en revanche on l'augmente, le temps d'exécution est généralement augmenté tandis que la qualité des solutions finales n'est pas vraiment meilleure.

Notons qu'il s'agit là de résultats moyens sur des instances de tailles très différentes. Lorsque l'on s'intéresse aux résultats pour chaque instance séparément, on constate que le nombre optimal de fourmis dépend de la taille de l'instance à résoudre. Il serait intéressant à ce niveau d'avoir une approche analytique pour fixer ce paramètre, en fonction d'une heuristique dépendante du problème. Par exemple, pour le problème du voyageur de commerce, le nombre de fourmis est généralement égal au nombre de sommets du graphe.

Influence de β. Ce paramètre détermine la sensibilité des fourmis au facteur heuristique —dépendant du problème— dans la règle de transition probabiliste. L'importance de ce facteur dépend très clairement du problème considéré et par conséquent, la valeur de β est différente d'un problème à un autre.

Par exemple, β a été fixé à 0 pour le problème de la clique maximum dans [FS03, SF05] car aucune heuristique valable n'a été trouvée ; il a été fixé à 1 pour le problème du k-arbre de poids minimum dans [Blu02] ; il a été fixé à 5 pour le problème du sac-à-dos multidimensionnel à la fois dans [ASG04] et [LM99] ; il a été fixé à 6 pour le problème d'ordonnacement de voitures dans [GPS03] et il a été fixé à 10 pour les problèmes de satisfaction de contraintes dans [Sol02a]. Pour le problème d'appariement de graphes [SSG05], deux facteurs heuristiques différents sont combinés : un premier qui évalue le bénéfice immédiat apporté par l'objet candidat, et le second qui anticipe son bénéfice potentiel. Ces deux facteurs heuristiques sont pondérés par deux paramètres différents : $\beta_1 = 8$ pour le bénéfice immédiat et $\beta_2 = 3$ pour le bénéfice potentiel.

5.4.2 Evaluation de l'effort de diversification/intensification pendant une exécution

Pour évaluer le degré d'intensification de la recherche lors de l'exécution d'un algorithme ACO, on peut mesurer la distribution des quantités de phéromone déposées sur le graphe de construction [DG97] : si lors du choix d'un composant de solution, la majorité des candidats ont un facteur phéromonal très faible et seulement quelques candidats ont un facteur phéromonal important, les fourmis choisiront quasiment toujours les mêmes composants, qui seront de nouveau renforcés dans un processus auto-catalytique. Notons cependant que l'introduction des bornes τ_{min} et τ_{max} dans le $\mathcal{MMAS}$ permet de limiter ce phénomène de stagnation.

Une autre possibilité pour évaluer le degré d'intensification de la recherche consiste à mesurer la diversité des solutions calculées pendant une exécution. De nombreuses mesures de diversité ont été introduites pour les approches évolutionnistes car le maintien de la diversité dans la population est un point clé pour empêcher une convergence prématurée de la résolution. Les mesures de diversité les plus courantes sont notamment le nombre de valeurs de « fitness » différentes, le nombre d'individus structurellement différents, et la distance moyenne entre les individus de la population [BGK02].

Pour mesurer l'effort de diversification dans les algorithmes ACO que nous avons développés, nous avons utilisé deux indicateurs : le taux de ré-échantillonnage et le taux de similarité. Cette étude est menée ici sur l'algorithme *Ant-SS* appliqué à la résolution du problème de la recherche de cliques maximums, mais nous avons observé des comportements très similaires pour tous les autres algorithmes ACO que nous avons proposés.

Taux de ré-échantillonnage

Cette mesure a été utilisée, par exemple, dans [vHB02, vHS04], afin de fournir des informations sur la capacité d'un algorithme heuristique à « échantillonner » l'espace de recherche : si on définit *nbDiff* comme étant le nombre de solutions différentes générées par un algorithme pendant une de ses exécutions, et *nbTot* comme étant le nombre total de solutions générées, alors le taux de ré-échantillonnage est défini par $(nbTot - nbDiff)/nbTot$. Un taux proche de to 0 correspond à une recherche « efficace », i.e., les solutions générées sont généralement différentes les unes des autres, tandis qu'un taux proche de 1 indique une stagnation du processus de recherche autour d'un petit nombre de solutions qui sont reconstruites régulièrement.

TAB. 5.6 – Evolution du taux de ré-échantillonnage de *Ant-SS(Clique)* et *Ant-SS(Vertex)* pour le problème de recherche d'une clique maximum dans le graphe `C500.9`. Chaque ligne donne successivement les valeurs de α et ρ, et les taux de ré-échantillonnage après 500, 1000, 1500, 2000, et 2500 cycles (en moyenne sur 50 exécutions). Les autres paramètres ont été fixés à $nbAnts = 30$, $\tau_{min} = 0.01$, $\tau_{max} = 6$.

Taux de ré-échantillonnage de *Ant-SS(Clique)*

Nombre de cycles :	500	1000	1500	2000	2500
$\alpha = 1, \rho = 0.995$	0.00	0.00	0.00	0.00	0.00
$\alpha = 1, \rho = 0.99$	0.00	0.00	0.00	0.00	0.00
$\alpha = 2, \rho = 0.99$	0.00	0.04	0.06	0.07	0.07
$\alpha = 2, \rho = 0.98$	0.06	0.10	0.12	0.13	0.13

Taux de ré-échantillonnage de *Ant-SS(Vertex)*

Nombre de cycles :	500	1000	1500	2000	2500
$\alpha = 1, \rho = 0.995$	0.00	0.00	0.02	0.13	0.25
$\alpha = 1, \rho = 0.99$	0.00	0.07	0.26	0.39	0.48
$\alpha = 2, \rho = 0.99$	0.38	0.68	0.78	0.84	0.87
$\alpha = 2, \rho = 0.98$	0.58	0.78	0.85	0.88	0.91

Le tableau 5.6 donne un aperçu des performances de *Ant-SS* en termes de ré-échantillonnage. On constate que *Ant-SS(Vertex)* est moins « efficace » que *Ant-SS(Clique)* pour explorer l'espace de recherche car il reconstruit souvent des cliques qu'il avait déjà construites auparavant. Par exemple, quand on fixe α à 1 et ρ à 0.99, 7% des cliques construites par *Ant-SS(Vertex)* pendant les mille premiers cycles avaient déjà été construites avant. Ce taux de ré-échantillonnage augmente ensuite très rapidement pour atteindre 48% au bout de 2500 cycles, i.e., près de la moitié des cliques construites avaient déjà été construites auparavant. En comparaison, *Ant-SS(Clique)* ne recalcule quasiment jamais deux fois une même clique de sorte qu'il explore effectivement deux fois plus d'états dans l'espace de recherche.

Taux de similarité

Le taux de ré-échantillonnage nous permet de quantifier la taille de l'espace exploré, et nous montre par exemple que *Ant-SS(Clique)* a une plus grande capacité d'exploration que *Ant-SS(Vertex)* pour l'instance `C500.9` du problème de la clique maximum. Cependant, le taux de ré-échantillonnage ne donne aucune information sur la distribution des états explorés dans l'espace de recherche. De fait, une recherche purement aléatoire ne recalcule quasiment jamais deux fois la même solution mais, comme elle n'intensifie pas non plus la recherche autour des zones les plus prometteuses, elle n'est généralement pas capable de trouver de bonnes solutions. Afin de fournir une vue complémentaire sur les capacités des algorithmes ACO en termes de diversification et intensification, on peut également calculer le taux de similarité qui quantifie le degré de similarité des solutions construites, i.e., d'intensification de la recherche. Ce taux de similarité correspond à la « pair-wise population diversity measure » introduite pour les algorithmes génétiques, e.g., dans [SBC01, MJ01].

De façon générale, le taux de similarité d'une suite de solutions S correspond au ratio entre la taille moyenne de l'intersection de chaque paire de solutions de S par rapport à la taille moyenne des solutions de S. Un taux de similarité égal à 1 correspond à une suite de solutions toutes identiques tandis qu'un taux de similarité à 0 correspond à une suite dont toutes les paires de solutions ont une intersection vide.

La figure 5.7 trace l'évolution du taux de similarité de l'ensemble des cliques construites tous les 50 cycles, et montre ainsi l'évolution de la répartition de l'ensemble des cliques calculées au sein de l'espace des cliques, pour le problème de la recherche d'une clique maximum dans le graphe `C500.9`. Considérons par exemple la courbe traçant l'évolution du taux de similarité pour *Ant-SS(Clique)* lorsque $\alpha = 1$ et $\rho = 0.99$. La similarité passe de moins de 10% au début du processus de résolution à 45% après un millier de cycles. Cela montre que les fourmis se focalisent progressivement sur une partie de l'espace de recherche, de telle sorte que deux cliques construites après 1000 cycles partagent près de la moitié de leurs sommets en moyenne. Si l'on ajoute à cela le fait que le taux de ré-échantillonnage est nul, on peut conclure que dans ce cas *Ant-SS(Clique)* a trouvé un bon compromis entre la diversification —étant donné qu'il ne recalcule jamais deux fois une même solution— et l'intensification —étant donné que la similarité des solutions calculées augmente.

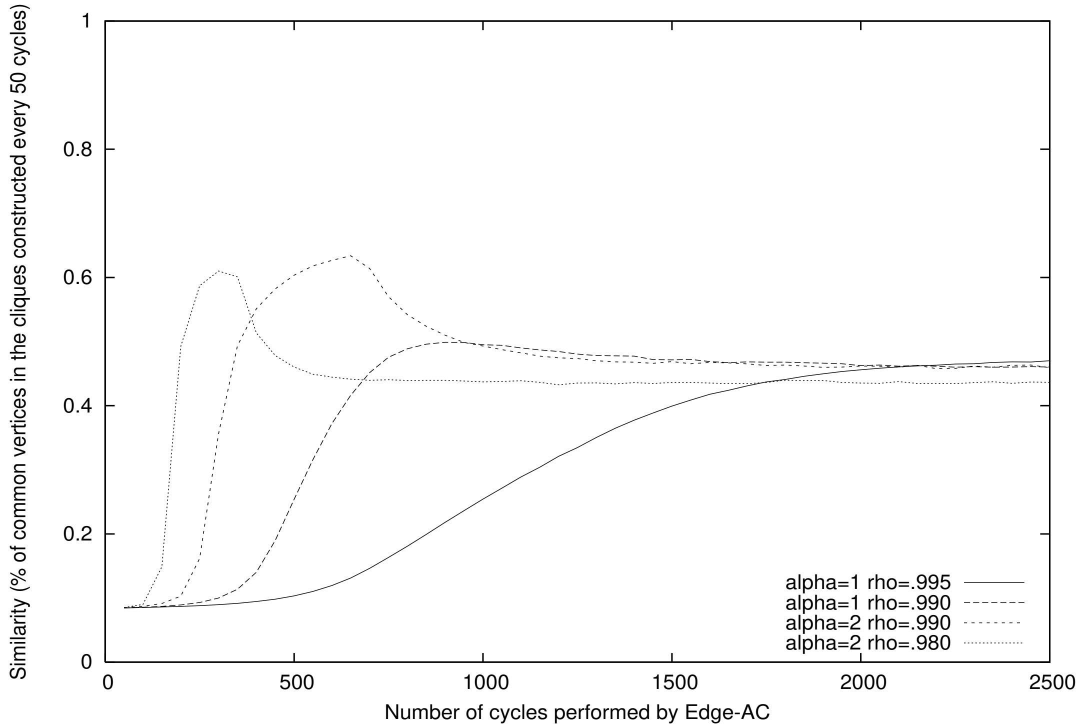
Les courbes de la figure 5.7 nous montrent également que lorsque α augmente ou ρ diminue, le taux de similarité augmente à la fois plus tôt et plus fortement. Cependant, à la fois pour *Ant-SS(Clique)* et *Ant-SS(Vertex)*, le taux de similarité des différentes exécutions converge en fin de résolution vers une même valeur, quelles que soient les valeurs de α et ρ : après deux mille cycles, les cliques calculées par *Ant-SS(Clique)* partagent environ 45% de leurs sommets tandis que celles calculées par *Ant-SS(Vertex)* partagent environ 90% de leurs sommets.

La différence de diversification entre *Ant-SS(Clique)* et *Ant-SS(Vertex)* peut s'expliquer par les choix faits au sujet de leurs composants phéromonaux. Considérons par exemple le problème de la recherche d'une clique maximum dans le graphe `C500.9`. Ce graphe a 500 sommets de sorte que *Ant-SS(Vertex)* peut déposer de la phéromone sur 500 composants phéromonaux tandis que *Ant-SS(Clique)* peut en déposer sur $500 \cdot 499/2 = 124750$ composants. Considérons maintenant les deux cliques qui sont récompensées à la fin des deux premiers cycles d'une exécution de *Ant-SS*. Pour les deux algorithmes *Ant-SS(Vertex)* et *Ant-SS(Clique)*, ces deux cliques contiennent 44 sommets en moyenne (voir la figure 5.6) et leur taux de similarité est inférieur à 10% (voir la figure 5.7) de sorte qu'elles partagent en moyenne 4 sommets. Sous cette hypothèse, après les deux premiers cycles de *Ant-SS(Vertex)*, 4 sommets —soit 1% des composants phéromonaux— ont été récompensés deux fois, et 80 sommets —soit 16% des composants phéromonaux— ont été récompensés une fois. A titre de comparaison, et sous les mêmes hypothèses, après les deux premiers cycles de *Ant-SS(Clique)*, 6 arêtes —soit moins de 0.005% des composants phéromonaux— ont été récompensées deux fois, et 940 arêtes —soit moins de 0.8% des composants phéromonaux— ont été récompensées une fois. Cela explique pourquoi la recherche est plus diversifiée avec *Ant-SS(Clique)* qu'avec *Ant-SS(Vertex)*, et également pourquoi *Ant-SS(Clique)* a besoin de plus de cycles que *Ant-SS(Vertex)* pour converger.

Vers un algorithme ACO réactif

Ces taux de ré-échantillonnage et de similarité sont très utiles lors de la phase de mise au point des algorithmes ACO pour fixer les valeurs des paramètres α et ρ qui déterminent l'influence de la phéromone sur la résolution. En effet, leur évolution au cours du processus de résolution permet de détecter les problèmes de diversification ou intensification excessives : lorsque le taux de ré-échantillonnage augmente, cela signifie que la recherche est trop intensifiée et que le processus de résolution stagne ; lorsqu'au contraire le taux de similarité n'augmente pas, cela signifie que la recherche est trop diversifiée et n'est plus guidée par la phéromone.

Ces deux taux peuvent être calculés incrémentalement lors de la construction de solutions sans que les temps



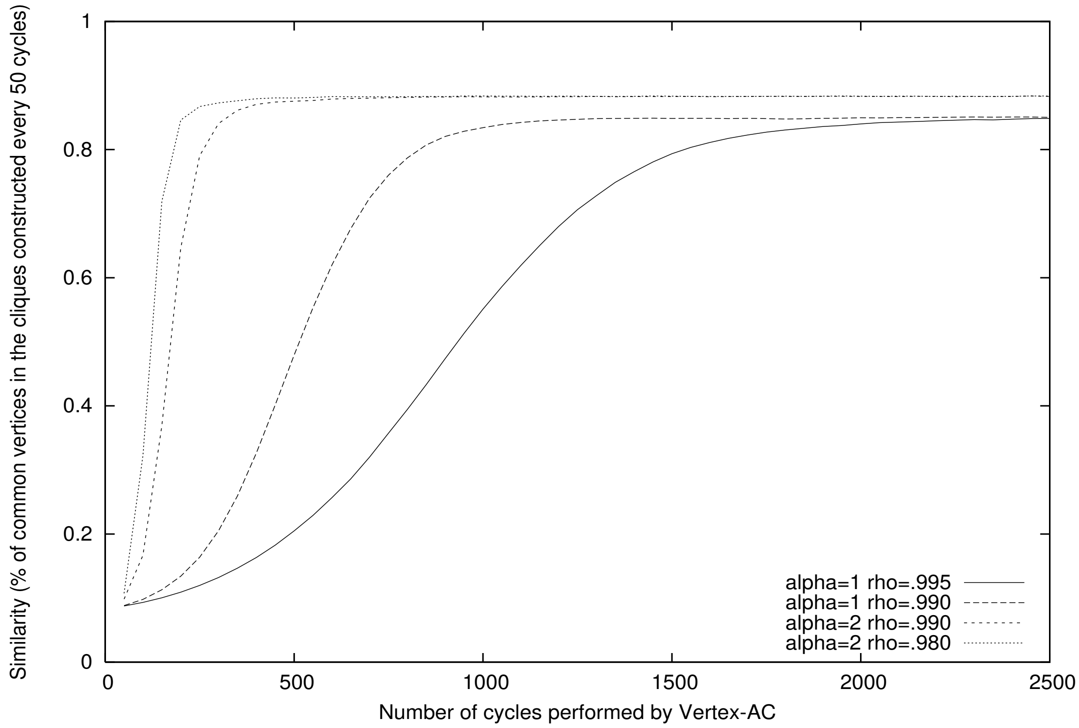


FIG. 5.7 – Evolution du taux de similarité pour *Ant-SS(Clique)* (courbes du haut) et *Ant-SS(Vertex)* (courbes du bas) pour le problème de la recherche d'une clique maximum dans le graphe `C500.9` : chaque courbe donne le taux de similarité de l'ensemble des cliques construites tous les 50 cycles (en moyenne sur 50 exécutions), pour différentes valeurs de α et ρ. Les autres paramètres ont été fixés à $nbAnts = 30$, $\tau_{min} = 0.01$, $\tau_{max} = 6$.

d'exécution ne soient augmentés de façon sensible :

– Afin de pouvoir calculer le taux de ré-échantillonnage pendant une exécution sans pénaliser les temps d'exécution, on calcule ce taux en utilisant une table de hachage. Pour chaque solution construite, on calcule deux clés de hachage différentes, et cela de façon incrémentale pendant la construction de la solution. La première clé donne l'indice d'entrée dans la table, la deuxième clé étant stockée dans la table (au lieu de stocker la solution, ce qui prendrait à la fois trop de mémoire et trop de temps pour ensuite comparer les solutions). Cette technique permet de détecter les conflits (correspondant aux cas où on re-calcule une solution déjà stockée dans la table) avec un taux d'erreur très réduit, correspondant à la probabilité que deux solutions différentes aient exactement les mêmes valeurs pour les deux clés de hachage. Notons que ce taux d'erreur est « pessimiste », i.e., le taux de ré-échantillonnage calculé ne peut être que supérieur ou égal au véritable taux de ré-échantillonnage.
– Le taux de similarité d'un ensemble de n solutions de taille m peut être évalué de façon naïve en calculant l'intersection de chaque paire de solutions, l'algorithme résultant ayant une complexité en $\mathcal{O}(n^2m)$.
  Une façon plus efficace d'évaluer ce taux de similarité consiste à maintenir un tableau `freq` tel que, pour chaque composant de solution v_i, $\texttt{freq}[\texttt{i}]$ soit égal au nombre de solutions contenant le composant v_i. Dans ce cas, le taux de similarité d'un ensemble de solutions S,
  $$\frac{\sum_{v_i \in comp}(\texttt{freq}[\texttt{i}] \cdot (\texttt{freq}[\texttt{i}] - 1))}{(\mid S \mid -1) \cdot \sum_{s_k \in S} \mid s_k \mid}$$
  où *comp* est l'ensemble des composants de solution de S, i.e.,
  $$comp = \{v_i / \exists s_k \in S, v_i \text{ est un composant de } S_k\}$$
  Ce calcul peut être facilement fait incrémentalement pendant la construction des solutions (voir [MJ01] pour plus de détails).

Ainsi, l'utilisation de ces taux pourrait être systématisée, et on pourrait concevoir un algorithme ACO réactif qui serait capable d'adapter dynamiquement ses paramètres au cours de la résolution en fonction de l'évolution des taux de ré-échantillonnage et de similarité, d'une façon un peu similaire à l'approche taboue réactive proposée dans [BP01].

5.4.3 Accélération de la phase d'exploration d'un algorithme ACO

La figure 5.6 montre que les meilleures solutions sont trouvées, en fin d'exécution, lorsque l'on choisit des valeurs pour les paramètres α et ρ qui favorisent l'exploration, comme par exemple, $\alpha = 1$ et $\rho = 0.99$. Cependant, avec un tel paramétrage la phéromone ne commence à réellement influencer les fourmis qu'après une étape d'exploration de plusieurs centaines de cycles. Notons que cette phase d'exploration en début de résolution, nécessaire pour éviter une convergence prématurée vers des solutions sous-optimales, est exacerbée dans le $\mathcal{MMAS}$ par le fait que les traces de phéromone sont initialisées à la borne maximale τ_{max} de sorte que durant les premiers cycles toutes les traces de phéromone ont quasiment la même valeur.

Cependant, si l'influence de la phéromone est délibérément limitée au début de la recherche, gérer la phéromone prend du temps. Cela est d'autant plus vrai pour *Ant-SS(Clique)* : étant donné un SS-problème $(S, S_{consistant}, f)$, chaque étape d'évaporation nécessite $\mathcal{O}(|S|^2)$ opérations, et la récompense d'une solution $S_k \in S_{consistant}$ nécessite $\mathcal{O}(|S_k|^2)$ opérations.

Partant du constat que pour *Ant-SS(Clique)* la gestion de la phéromone est coûteuse en temps alors qu'elle ne commence à influencer la recherche qu'après plusieurs centaines de cycles, nous avons proposé dans [Sol02b] d'introduire une étape de pré-traitement. L'idée générale est la suivante :

1. On collecte dans un premier temps un nombre significatif d'optima locaux, en construisant des solutions sans utiliser la phéromone (i.e., en mettant α à 0) et en améliorant ces solutions par recherche locale. Ces solutions constituent un échantillonnage de l'espace des solutions $S_{consistant}$.

2. On sélectionne dans cet échantillon les n_{Best} meilleures solutions et on les utilise pour initialiser les traces de phéromone.
3. On continue alors la recherche avec *Ant-SS(Clique)* en utilisant les traces de phéromone pour intensifier l'effort vers les zones les plus prometteuses.

On appelle *SampleSet* l'ensemble des optima locaux collectés durant la première étape, et $BestOf(n_{Best}, SampleSet)$ les n_{Best} meilleures solutions de *SampleSet*. Pour déterminer le nombre d'optima locaux qui doivent être collectés dans *SampleSet*, il s'agit de trouver un compromis entre calculer un grand nombre de solutions —qui seront plus représentatives de l'espace de recherche— et calculer un petit nombre de solutions —qui seront plus vite calculées. En pratique, les expérimentations nous ont montré que plus un problème est difficile, et plus il faut calculer d'optima locaux. Par conséquent, au lieu de calculer un nombre fixe de solutions dans *SampleSet*, on fixe uniquement n_{Best} —le nombre de solutions qui sont sélectionnées dans *SampleSet* pour initialiser les traces de phéromone— et on introduit une limite ϵ sur le taux d'amélioration de ces n_{Best} meilleures solutions, i.e., on arrête de construire de nouvelles solutions quand le coût moyen de $BestOf(n_{Best}, SampleSet)$ n'a pas été amélioré de plus de $\epsilon\%$ depuis les n_{Best} dernières solutions calculées.

Plus précisément, la procédure pour construire *SampleSet* est la suivante :

construire n_{Best} solutions et les stocker dans *SampleSet*
répéter
  $OldCost \leftarrow \sum_{S \in BestOf(n_{Best}, SampleSet)} f(S)$
  calculer n_{Best} solutions de plus et les ajouter à *SampleSet*
  $NewCost \leftarrow \sum_{S \in BestOf(n_{Best}, SampleSet)} f(S)$
jusqu'à ce que $NewCost/OldCost > 1 - \epsilon$ **ou** une solution optimale a été trouvée

A la suite de cette étape d'échantillonnage, si la solution optimale n'a pas été trouvée, l'ensemble $BestOf(n_{Best}, SampleSet)$ —qui contient les n_{Best} meilleures solutions de *SampleSet*— est utilisé pour initialiser les traces de phéromone : la quantité de phéromone déposée sur chaque paire d'objets $(o_i, o_j) \in S \times S$ est définie par

$$\tau(o_i, o_j) = \sum_{S_k \in BestOf(n_{Best}, SampleSet) \text{ tel que } \{o_i, o_j\} \subseteq S_k} \frac{1}{1 + f_{best} - f(S_k)}$$

où f_{best} est le coût de la meilleure solution de $BestOf(n_{Best}, SampleSet)$.

Nous avons évalué expérimentalement cette étape de prétraitement pour les instantiations de *Ant-SS(Clique)* pour la résolution de CSPs binaires dans [Sol02b] et pour la recherche de cliques maximums dans [FS03]. Pour ces deux SS-problèmes, nous avons montré que cette étape de prétraitement accélère la résolution (les temps de calcul sont divisés par deux en moyenne), tandis que la qualité des solutions finales est sensiblement la même.

Chapitre 6

Appariements de graphes et Similarités

6.1 Introduction

6.1.1 De la nécessité d'évaluer la similarité d'objets

Evaluer la similarité de deux objets est un problème qui se pose dans de nombreuses applications informatiques, un corollaire de ce problème étant de retrouver, parmi un ensemble d'objets, l'objet le plus similaire à un autre. Citons entre autres :
- la recherche d'informations, où il s'agit par exemple de trouver des documents similaires à un document requête donné ;
- la classification, où il s'agit de regrouper des documents similaires dans des classes ;
- le raisonnement à partir de cas, où il s'agit de trouver un problème déjà résolu ressemblant à un nouveau problème à résoudre dans le but d'adapter la solution du premier au second ;
- la reconnaissance d'images, où l'on confronte une image à son modèle ;
- le suivi d'objets dans une scène vidéo, où l'on recherche les différences entre deux images successives, ces différences étant interprétées comme des mouvements d'objets ;
- la biochimie, où l'on recherche des similitudes entre des protéines ou des molécules.

Pour ces applications, il s'agit de disposer d'une mesure de similarité à la fois *souple*, *quantitative*, *qualitative* et *personnalisable*. Pour illustrer ces différents points, considérons les deux objets de la figure 6.1, tirés d'une application en CAO [Cha02]. Ces deux objets sont similaires : les poutres a, b, c et d de l'objet 1 jouent le même rôle que les poutres 1, 2, 3 et 4 de l'objet 2, tandis que les murs e et f correspondent au mur 5. Ces deux objets, bien que similaires, ne sont toutefois pas identiques : les poutres n'ont pas la même forme (en I à gauche et en U à droite) et le nombre de murs est différent (le mur 5 à droite joue seul le rôle des murs e et f à gauche). Cet exemple nous permet d'introduire les trois points suivants, qui nous semblent tout particulièrement importants pour mesurer la similarité d'objets.

1. Pour mesurer la similarité de deux objets, il est nécessaire d'établir une correspondance entre leurs composants. Plus précisément, il s'agit de trouver le « meilleur » appariement possible, i.e., celui qui met en correspondance les composants les plus similaires, la similarité des composants étant fonction des caractéristiques qu'ils ont en commun. Par exemple, les composants a et 1 partagent les caractéristiques « être une poutre », « être à coté de » une autre poutre et « être sur » un mur, tandis qu'ils ne partagent pas la caractéristique de forme.

   Un point important pour introduire une certaine *souplesse* dans la mesure de similarité est que cet appariement ne doit pas nécessairement être univalent : il doit être possible d'associer un composant à plusieurs autres. Ainsi, le mur 5 de l'objet 2 joue seul le rôle des deux murs e et f de l'objet 1. De façon plus générale, les objets comparés peuvent être décrits à des niveaux de granularité différents, de sorte

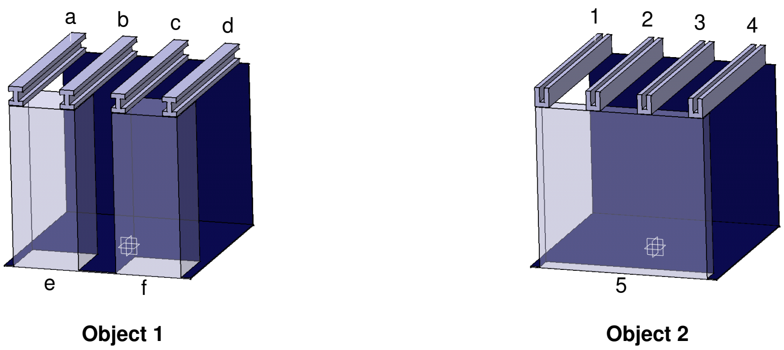


FIG. 6.1 – Deux objets de conception similaires

qu'un ensemble de composants d'un objet doit pouvoir être associé à un seul composant de l'autre objet.

2. Etant donné un tel appariement, on peut *quantifier* la similarité des deux objets par rapport à cet appariement, i.e., évaluer la quantité de caractéristiques qu'ils ont en commun par rapport au nombre total de caractéristiques. Il est également important de pouvoir *qualifier* la similarité des deux objets, i.e., expliciter en quoi ils sont similaires et en quoi ils sont différents, en identifiant les caractéristiques qu'ils ont en commun et celles qu'ils ne partagent pas.

   Par exemple, dans le contexte de la recherche d'informations, cet aspect permet d'expliquer à l'utilisateur les différences entre une requête et une réponse. Cette évaluation qualitative des différences entre deux objets est alors semblable à la notion d'explications proposée dans [GJ01].

3. Enfin, l'importance relative des différentes caractéristiques ne doit pas être imposée, mais doit être un paramètre de la mesure permettant de la *personnaliser* en fonction des objectifs de l'utilisateur et du domaine d'application considéré : on doit par exemple pouvoir spécifier que partager la caractéristique « être une poutre » est plus important que partager une caractéristique de forme. Dans le contexte de la recherche d'informations, il est également important que ces connaissances de similarité puissent être ajoutées ou modifiées de façon interactive et dynamique, afin de pouvoir améliorer la pertinence des résultats retournés et les personnaliser en fonction de l'utilisateur.

6.1.2 Similarité d'objets et appariements de graphes

Dans beaucoup d'applications, les objets dont on veut évaluer la similarité peuvent être décrits par des graphes : les sommets d'un graphe représentent les composantes d'un objet tandis que les arcs décrivent les relations binaires entre ces différentes composantes. Ainsi, les graphes ont été utilisés, par exemple, pour représenter des molécules ([GAW97]), des protéines ([Aku95]) ou des images ([BYV00], [HW02]). Les graphes sont aussi couramment utilisés en ingénierie, e.g., en conception assistée par ordinateur ([Cha02]), ou pour l'ingénierie logicielle ([GJ01]). Notons que ces graphes sont généralement typés ou étiquetés afin de caractériser les différentes composantes de l'objet et leurs relations.

Pour évaluer la similarité d'objets décrits par des graphes, il s'agit alors de mettre en correspondance les sommets des graphes de façon à retrouver le plus grand nombre de caractéristiques communes aux graphes. Nous avons introduit en 4.2.2 un certain nombre d'appariements de graphes couramment utilisés pour comparer des graphes : l'isomorphisme de graphes, qui permet de vérifier que deux graphes sont identiques, l'isomorphisme de sous-graphes, qui permet de vérifier qu'un graphe est inclus dans un autre, et la recherche d'un plus grand sous-graphe commun, qui permet d'évaluer l'intersection de deux graphes. Ces différents types d'appariement de graphes ne sont pas complètement satisfaisants pour mesurer la similarité d'objets selon nos critères décrits précédemment. En particulier, ils ne permettent pas d'apparier un sommet d'un graphe avec un ensemble de sommets de l'autre graphe, et ils ne permettent pas de personnaliser facilement la mesure aux objectifs de l'utilisateur.

6.1.3 Organisation du chapitre

On s'intéresse dans ce chapitre au problème de mesurer la similarité d'objets décrits par des graphes, et nous introduisons en 6.2 une nouvelle mesure pour évaluer la similarité de graphes. Cette mesure permet d'identifier et expliquer les différences et points communs entre deux graphes. Elle permet également de comparer des graphes décrivant des objets à différents niveaux de granularité car elle est basée sur des appariements multivoques (permettant d'associer un sommet d'un graphe à un ensemble de sommets de l'autre graphe). Enfin, cette mesure est générique et elle est paramétrée par deux fonctions qui permettent d'intégrer des connaissances spécifiques à l'application. Nous montrons que ces deux fonctions permettent de retrouver les différents types d'appariement de graphes présentés en 4.2.2, ainsi que d'autres mesures de similarité proposées plus récemment.

Nous nous intéressons ensuite en 6.3 au calcul de cette mesure, et nous proposons et comparons quatre algorithmes différents :
- un algorithme glouton, qui construit incrémentalement un appariement en choisissant à chaque itération le « meilleur » couple de sommets à ajouter à l'appariement en cours de construction ;
- un algorithme tabou qui, partant d'un appariement initial généré par l'algorithme glouton, explore l'espace des appariements en ajoutant ou supprimant itérativement des couples de sommets, une liste taboue étant utilisée pour mémoriser et interdire les derniers mouvements effectués ;
- un algorithme réactif qui reprend le principe de l'algorithme tabou mais qui auto-régule la longueur de la liste taboue en fonction du besoin de diversification de la recherche ;
- un algorithme à base de colonies de fourmis, où les fourmis construisent des appariements selon une stratégie gloutonne, la phéromone étant utilisée pour les guider vers les zones les plus prometteuses de l'espace de recherche.

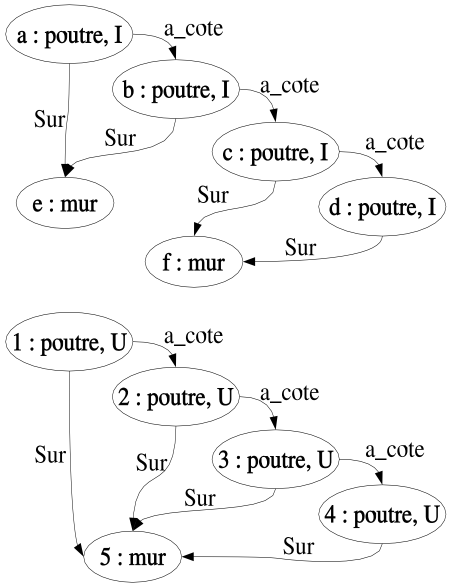
Notons que la mesure étant générique, ces algorithmes peuvent être utilisés pour résoudre différents types de problèmes d'appariement de graphes.

Enfin, la section 6.4 s'intéresse à la résolution d'un problème d'appariement de graphes particulier, à savoir le problème de l'isomorphisme de graphes. Ce problème, qui est bien résolu par des approches dédiées tirant partie d'invariants structurels pour élaguer très efficacement la recherche, devient en revanche très difficile à résoudre lorsqu'on le transforme en un problème de satisfaction de contraintes pour utiliser la programmation par contraintes car la transformation décompose la sémantique globale du problème initial en un ensemble de contraintes binaires. Afin de résoudre efficacement les problèmes d'isomorphisme de graphes avec la programmation par contraintes, nous introduisons une contrainte globale dédiée à ce problème et un algorithme de filtrage associé.

6.2 Une nouvelle mesure de similarité

6.2.1 Appariements multivoques de graphes étiquetés

Définition (graphe étiqueté). Un graphe étiqueté est un graphe orienté auquel on a associé à chacun des sommets et chacun des arcs un ensemble non vide d'étiquettes. Plus formellement, étant donné L_V un ensemble fini d'étiquettes de sommets et L_E un ensemble fini d'étiquettes d'arcs, un graphe étiqueté est défini par un triplet $G = \langle V, r_V, r_E \rangle$ tel que :
- V est un ensemble fini de sommets,



$$
\begin{aligned}
G_1 = \langle\ V_1 =& \{\, a, b, c, d, e, f \,\} \\
r_{V_1} =& \{\, (a, poutre), (b, poutre), (c, poutre), (d, poutre), \\
& (a, I), (b, I), (c, I), (d, I), \\
& (e, mur), (f, mur) \,\} \\
r_{E_1} =& \{\, (a, b, a\_cote), (b, c, a\_cote), (c, d, a\_cote), \\
& (a, e, sur), (b, e, sur), (c, f, sur), (d, f, sur) \,\}\ \rangle
\end{aligned}
$$

$$
\begin{aligned}
G_2 = \langle\ V_2 =& \{\, 1, 2, 3, 4, 5 \,\} \\
r_{V_2} =& \{\, (1, poutre), (2, poutre), (3, poutre), (4, poutre), \\
& (1, U), (2, U), (3, U), (4, U), \\
& (5, mur) \,\} \\
r_{E_2} =& \{\, (1, 2, a\_cote), (2, 3, a\_cote), (3, 4, a\_cote), \\
& (1, 5, sur), (2, 5, sur), (3, 5, sur), (4, 5, sur) \,\}\ \rangle
\end{aligned}
$$

FIG. 6.2 – Les graphes étiquetés $G1$ et $G2$ décrivant les objets 1 et 2 de la figure 6.1 : à gauche, leur représentation graphique, à droite leur représentation dans le formalisme présenté en 6.2.1.

- $r_V \subseteq V \times L_V$ est la relation associant sommets et étiquettes, i.e., r_V est l'ensemble des couples (v_i, l) tels que le sommet v_i est étiqueté par l,
- $r_E \subseteq V \times V \times L_E$ est la relation associant arcs et étiquettes, i.e., r_E est l'ensemble des triplets (v_i, v_j, l) tels que l'arc (v_i, v_j) est étiqueté par l. On suppose ici que chaque arc a au moins une étiquette de sorte que l'ensemble des arcs peut être défini par $E = \{(v_i, v_j) | \exists l, (v_i, v_j, l) \in r_E\}$.

Nous appelons les tuples de r_V les caractéristiques des sommets de G et ceux de r_E les caractéristiques des arcs de G. On définit le descripteur d'un graphe $G = \langle V, r_V, r_E \rangle$ comme l'ensemble de toutes ses caractéristiques de sommet et d'arc : $descr(G) = r_V \cup r_E$. Cet ensemble décrit entièrement le graphe et sera utilisé pour mesurer la similarité de deux graphes.

Exemple. Considérons les deux objets de la figure 6.1. Afin de représenter ces objets avec des graphes étiquetés, définissons tout d'abord les ensembles d'étiquettes des sommets et des arcs :

$$
\begin{aligned}
L_V &= \{\, poutre, I, U, mur \,\} \\
L_E &= \{\, sur, a\_cote \,\}
\end{aligned}
$$

Etant donnés ces ensembles d'étiquettes, les deux objets peuvent alors être représentés par les deux graphes étiquetés de la figure 6.2. Nous constatons que les sommets a, b, c et d possèdent les deux étiquettes $poutre$ et I tandis que les sommets 1, 2, 3 et 4 possèdent les étiquettes $poutre$ et U. Cela nous permet d'exprimer le fait que les objets correspondant à ces sommets partagent la propriété d'être une poutre bien qu'ils aient une forme différente.

De façon plus générale, le fait que les arcs et les sommets puissent avoir plusieurs étiquettes peut être utilisé pour exprimer des relations d'héritage en termes d'inclusion des ensembles d'étiquettes. La similarité de deux sommets ou arcs sera alors calculée par rapport à leurs étiquettes communes, correspondant à leurs ancêtres communs dans la hiérarchie d'héritage.

Définition (appariement multivoque). Un appariement multivoque entre deux graphes $G_1 = \langle V_1, r_{V_1}, r_{E_1} \rangle$ et $G_2 = \langle V_2, r_{V_2}, r_{E_2} \rangle$, tels que $V_1 \cap V_2 = \emptyset$, est une relation $m \subseteq V_1 \times V_2$. Un tel appariement associe à chaque sommet d'un graphe 0, 1 ou plusieurs sommets de l'autre graphe. Par extension, nous noterons $m(v)$ l'ensemble des sommets associés au sommet v dans l'appariement m, *i.e.*,

$$\begin{array}{ll} \forall v_1 \in V_1, & m(v_1) \doteq \{v_2 \in V_2 |\ (v_1, v_2) \in m\} \\ \forall v_2 \in V_2, & m(v_2) \doteq \{v_1 \in V_1 |\ (v_1, v_2) \in m\} \end{array}$$

Exemple. Pour les graphes étiquetés de la figure 6.2, nous pouvons définir les deux appariements suivants :

$$\begin{array}{rl} m_A = & \{(a,1),(b,2),(c,3),(d,4),(e,5),(f,5)\} \\ m_B = & \{(a,1),(b,2),(a,3),(b,4)),(e,5)\} \end{array}$$

L'appariement m_A met respectivement en correspondance les sommets a, b, c, et d aux sommets 1, 2, 3 et 4 ainsi que les deux sommets e et f au sommet 5 (*i.e.*, $m_A(5) = \{e, f\}$). L'appariement m_B associe les sommets 1 et 3 au sommet a, les sommets 2 et 4 au sommet b et le sommet e au sommet 5. Dans cet appariement, aucun sommet n'a été mis en relation avec c, d et f (*i.e.*, $m_B(c) = m_B(d) = m_B(f) = \emptyset$).

6.2.2 Similarité basée sur des appariements multivoques

Caractéristiques communes à deux graphes relativement à un appariement. Afin de mesurer la similarité entre deux objets, il est courant et assez intuitif de comparer la quantité de caractéristiques communes à ces deux objets par rapport à l'ensemble de toutes leurs caractéristiques [Tve77, Lin98]. Un graphe étiqueté G étant décrit par l'ensemble $descr(G)$ des caractéristiques de ses sommets et arcs, la similarité de deux graphes $G_1 = \langle V_1, r_{V_1}, r_{E_1} \rangle$ et $G_2 = \langle V_2, r_{V_2}, r_{E_2} \rangle$ dépend des caractéristiques communes des ensembles $descr(G_1)$ et $descr(G_2)$. Cependant, étant donné que $V_1 \cap V_2 = \emptyset$, l'intersection de ces deux ensembles de descripteurs sera toujours vide. Nous devons donc calculer cette intersection par rapport à un appariement m associant les sommets de G_1 à ceux de G_2, i.e.,

$$\begin{array}{rcl} descr(G_1) \sqcap_m descr(G_2) & \doteq & \{(v,l) \in r_{V_1} |\ \exists v' \in m(v),\ (v',l) \in r_{V_2}\} \\ & \cup & \{(v,l) \in r_{V_2} |\ \exists v' \in m(v),\ (v',l) \in r_{V_1}\} \\ & \cup & \{(v_i,v_j,l) \in r_{E_1} |\ \exists v'_i \in m(v_i), \exists v'_j \in m(v_j)\, (v'_i,v'_j,l) \in r_{E_2}\} \\ & \cup & \{(v_i,v_j,l) \in r_{E_2} |\ \exists v'_i \in m(v_i), \exists v'_j \in m(v_j)\, (v'_i,v'_j,l) \in r_{E_1}\} \end{array}$$

Cet ensemble contient toutes les caractéristiques des éléments de G_1 (resp. G_2) retrouvées au moins une fois dans G_2 (resp. G_1) via l'appariement m.

Exemple. Considérons les deux graphes étiquetés et leurs ensembles de descripteurs de la figure 6.2. L'intersection de leurs caractéristiques communes par rapport aux appariements m_A et m_B proposés précédemment sont :

$$\begin{array}{rcl} descr(G_1) \sqcap_{m_A} descr(G_2) & = & descr(G_1) \cup descr(G_2) \\ & & -\{\,(a,I),(b,I),(c,I),(d,I),(1,U),(2,U),(3,U),(4,U)\,\} \\ descr(G_1) \sqcap_{m_B} descr(G_2) & = & descr(G_1) \cup descr(G_2) \\ & & -\{\,(a,I),(b,I),(c,I),(d,I),(1,U),(2,U),(3,U),(4,U), \\ & & \quad (f,mur),(c,poutre),(d,poutre), \\ & & \quad (b,c,a\_cote),(c,d,a\_cote),(c,f,sur),(d,f,sur),(2,3,a\_cote)\,\} \end{array}$$

Similarité de deux graphes relativement à un appariement. Nous pouvons alors définir la similarité entre G_1 et G_2 en fonction de leurs caractéristiques communes par rapport à l'union de leurs caractéristiques :

$$sim1_m(G_1, G_2) \quad = \quad \frac{f(descr(G_1) \sqcap_m descr(G_2))}{f(descr(G_1) \cup descr(G_2))} \tag{6.1}$$

où f est une fonction positive croissante et monotone par rapport à l'inclusion, et telle que $f(\emptyset) = 0$. Cette fonction permet de pondérer l'importance des différentes caractéristiques.

Notons que cette similarité est toujours comprise entre 0 (quand $descr(G_1) \sqcap_m descr(G_2) = \emptyset$) et 1 (quand $descr(G_1) \sqcap_m descr(G_2) = descr(G_1) \cup descr(G_2)$) et qu'elle dépend de l'appariement m considéré : plus m apparie de composants (sommets ou arcs) ayant des étiquettes communes, et plus la similarité se rapproche de 1.

Cette première définition de la similarité n'est pas entièrement satisfaisante dans le sens où elle ne tient pas compte du fait qu'un sommet peut être « éclaté », i.e., apparié à plusieurs sommets. Par exemple, pour les deux graphes étiquetés de la figure 6.2, l'appariement m_A apparie le sommet 5 avec les deux sommets e et f. Il s'agit de prendre en compte ces éclatements de sommets dans notre mesure. Pour cela, nous introduisons la fonction $splits$ qui retourne l'ensemble des sommets « éclatés » avec l'ensemble des sommets auxquels ils sont reliés :

$$splits(m) \doteq \{(v, s_v) |\ v \in V_1 \cup V_2,\ s_v = m(v),\ |m(v)| \geq 2\}$$

et nous définissons un opérateur d'inclusion sur ces ensembles de sommets éclatés :

$$splits(m1) \sqsubseteq splits(m2) \dot{\Leftrightarrow} \forall(v, s_v) \in splits(m_1),\ \exists(v, s'_v) \in splits(m_2),\ s_v \subseteq s'_v$$

Par exemple, les appariements m_A et m_B précédents génèrent les splits suivants : $splits(m_A) = \{\ (5, \{e, f\})\ \}$ et $splits(m_B) = \{\ (a, \{1, 3\}),\ (b, \{2, 4\})\ \}$.

Nous pouvons alors modifier l'équation (6.1) afin qu'elle prenne en compte les éclatements de sommets :

$$sim_m(G_1, G_2) \quad = \quad \frac{f(descr(G_1) \sqcap_m descr(G_2)) - g(splits(m))}{f(descr(G_1) \cup descr(G_2))} \tag{6.2}$$

où la fonction g possède les mêmes propriétés que f : elle est positive et croissante par rapport à la relation d'inclusion $\sqsubseteq$ et $g(\emptyset) = 0$. Dès lors, la similarité ne peut que décroître lorsque le nombre d'éclatements de sommets dans l'appariement m croît.

Similarité de deux graphes. Nous avons défini la similarité de deux graphes par rapport à un appariement entre leurs sommets. Maintenant, nous pouvons définir la *similarité* de deux graphes G_1 et G_2 comme la plus grande similarité par rapport à tous les appariements possibles :

$$sim(G_1, G_2) \quad = \quad \max_{m \subseteq V_1 \times V_2} \frac{f(descr(G_1) \sqcap_m descr(G_2)) - g(splits(m))}{f(descr(G_1) \cup descr(G_2))} \tag{6.3}$$

Notons que, si le degré de similarité sim_m obtenu pour un appariement m ne contenant pas de sommet éclaté est toujours compris entre 0 et 1, sim_m peut devenir négatif lorsque beaucoup de sommets sont éclatés : le poids de la fonction $splits$, défini par g, peut alors être plus grand que le poids des caractéristiques communes, défini par f. Cependant, dans tous les cas, la similarité maximum sera toujours comprise entre 0 et 1 car la similarité obtenue par l'appariement vide est nulle.

Notons aussi que c'est le choix des fonctions f et g qui permet de déterminer le meilleur appariement. f et g seront donc choisies en fonction du domaine d'application et du type d'appariement recherché ; cet aspect est illustré dans les trois sections suivantes.

6.2.3 Connaissances de similarité

La mesure de similarité définie par l'équation 6.3 est générique dans le sens où elle est paramétrée par les deux fonctions f et g : ces deux fonctions définissent l'importance relative des caractéristiques les unes par rapport aux autres (*e.g.*, rendre le concept *« être une poutre »* plus important que le concept *« avoir une forme en U »*). Le choix de f et g permet donc l'introduction de connaissances de similarité propres au domaine de l'application considérée. Ces fonctions sont souvent définies comme une somme pondérée. On introduit pour cela les trois fonctions de pondérations suivantes, définissant les poids des caractéristiques de sommets et d'arcs et les poids des éclatements :

$$\begin{aligned} poids_V &: V \times L_V \rightarrow \mathbb{R}^+ \\ poids_E &: V \times V \times L_E \rightarrow \mathbb{R}^+ \\ poids_g &: V \times \wp(V) \rightarrow \mathbb{R}^+ \end{aligned}$$

On définit alors les fonctions f et g par :

$$\begin{aligned} f(F) &= \sum_{(v,l)\in F} poids_V(v,l) + \sum_{(v_1,v_2,l)\in F} poids_E(v_1,v_2,l) \\ g(S) &= \sum_{(v,s_v)\in S} poids_g(v,s_v) \end{aligned}$$

Sans connaissances particulières du domaine, les fonctions de pondération $poids_V$ et $poids_E$ peuvent retourner une valeur constante, ce qui revient à considérer que toutes les caractéristiques sont également importantes. La fonction $poids_g(v,s_v)$ peut retourner une valeur proportionnelle à la cardinalité de s_v.

A contrario, si des connaissances contextuelles sont disponibles, il est possible d'associer un poids particulier à chacune des caractéristiques et à chacun des éclatements de sommets. Par exemple, le fait d'être une poutre est peut être plus important que le fait d'avoir une forme en U ; il est peut être plus génant d'éclater le composant 5 que le composant 1 ...

6.2.4 Comparaison avec les problèmes d'appariement de graphes « classiques »

Les fonctions f et g peuvent être définies de telle sorte que notre mesure de similarité permette la résolution des différents problèmes d'appariement de graphes introduits en 4.2.2. Ces appariements sont souvent définis pour des graphes non-étiquetés. Nous supposerons donc qu'un graphe non-étiqueté est un cas particulier de graphe étiqueté où tous les sommets (resp. arcs) ont la même étiquette l_v (resp. l_e).

Isomorphisme de graphes. Si les fonctions f et g de la formule 6.3 sont définies comme la fonction de cardinalité, alors $sim(G,G')$=1 si et seulement si il existe un appariement m tel que $descr(G) \sqcap_m descr(G') = descr(G) \cup descr(G')$ (*i.e.*, m permet de retrouver toutes les caractéristiques de G et G') et $splits(m) = \emptyset$, (*i.e.*, m est univoque). Autrement dit, $sim(G,G') = 1$ si et seulement si G et G' sont isomorphes.

Isomorphisme de sous-graphes partiel. Si la fonction g de la formule 6.3 est la fonction cardinalité et si f est une fonction ne comptant que les caractéristiques de G, *i.e.*, f est définie comme une somme pondérée où le poids des caractéristiques de G (resp G') est 1 (resp 0), alors $sim(G,G') = 1$ si et seulement si il existe un appariement m tel que $descr(G) \subseteq descr(G) \sqcap_m descr(G')$ (*i.e.*, m permet de retrouver toutes les caractéristiques de G) et $splits(m) = \emptyset$ (*i.e.*, m est univoque). Autrement dit, $sim(G,G') = 1$ si et seulement si il existe un isomorphisme de sous-graphes partiel de G dans G'.

Isomorphisme de sous-graphes. Par rapport au problème d'isomorphisme de sous-graphes partiels, il s'agit d'ajouter le fait que tous les couples de sommets de G qui ne sont pas reliés par un arc doivent être appariés à des sommets de G' qui ne sont également pas reliés par un arc. Pour vérifier cette condition, il faut ajouter à tous les couples de sommets non reliés par un arc une étiquette "pas-un-arc" puis s'assurer que le meilleur appariement retrouve ces étiquettes. Plus formellement, étant donné un graphe $G = (V, E)$, nous définissons le graphe étiqueté $G_{label} = (V, r_V, r_E)$ tel que $r_V = \{(v, l_v)|v \in V\}$ et $r_E = \{(u, v, l_e)|(u, v) \in E\} \cup \{(u, v, l_{notE})|(u, v) \in V \times V - E\}$. Si les fonctions f et g sont définies comme pour le problème de l'isomorphisme de sous-graphes partiel, $sim(G_{label}, G'_{label}) = 1$ si et seulement si il existe un isomorphisme de sous-graphes de G dans G'.

Plus grand sous-graphe partiel commun. Définissons la fonction f de la formule 6.3 comme une fonction de cardinalité et la fonction g telle que tout éclatement de sommet soit interdit (*i.e.*, $g(S) = +\infty$ si $S \neq \emptyset$ et $g(\emptyset) = 0$). L'appariement m qui maximise la formule 6.2 est alors l'appariement permettant de retrouver un maximum de caractéristiques de sommets et d'arcs tout en interdisant les éclatements de sommets. Cet appariement m correspond donc à un plus grand sous-graphe partiel commun.

Plus grand sous-graphe commun. En s'inspirant du problème de l'isomorphisme de sous-graphes non partiel, il est possible de résoudre le problème de la recherche du plus grand sous-graphe non partiel commun à deux graphes. Pour cela, il faut définir la fonction g de telle façon que les éclatements de sommets soient interdits (*i.e.*, $g(S) = +\infty$ si $S \neq \emptyset$ et $g(\emptyset) = 0$). La fonction f doit « vérifier » que seuls des couples de sommets de même types (*i.e.*, reliés ou pas à un arc) sont appariés entre eux. Autrement dit, la fonction f doit retourner une valeur nulle s'il existe un couple de sommets (u, v) de G (resp. de G') apparié à un couple de sommets (u', v') de G' (resp. de G) tel que la relation entre u et v (relié ou pas par un arc) n'est pas la même que la relation entre u' et v'. Dans tous les autres cas, f doit retourner une valeur proportionnelle au nombre de sommets appariés.

Distance d'édition de graphes. Si les graphes ne sont pas étiquetés (ou plus précisément si tous les sommets et arcs ont la même étiquette), on constate que, étant donné un appariement m, l'ensemble des caractéristiques de sommets et d'arcs contenues dans $descr(G) - (descr(G) \sqcap_m descr(G'))$ (resp. dans $descr(G') - (descr(G) \sqcap_m descr(G')))$ correspond à l'ensemble des opérations de suppression (resp. d'insertion) d'arcs et de sommets nécessaires pour rendre le graphe G isomorphe au graphe G'. Choisissons g de telle façon que les éclatements de sommets soient interdits et f comme une fonction de pondération où les poids sont définis en fonction des coûts des transformations élémentaires d'ajout et de suppression d'arcs et de sommets. L'appariement m qui maximise la formule 6.2 définit l'ensemble des opérations élémentaires qui minimise la distance d'édition.

Si nous considérons maintenant la distance d'édition appliquée à des graphes mono-étiquetés (*i.e.*, des graphes où chaque sommet et chaque arc possède une seule étiquette), les opérations de substitution (moins coûteuses qu'une suppression suivie d'une insertion) peuvent s'appliquer. Cependant, lorsque les sommets (resp. les arcs) sont mono-étiquetés, les informations contenues dans l'ensemble $descr(G) \sqcap_m descr(G')$ des caractéristiques communes ne permettent pas de différencier le cas où deux sommets (resp. arcs) n'ayant pas la même étiquette sont appariés du cas où ces deux sommets (resp. arcs) ne sont pas appariés (dans les deux cas, aucune des deux étiquettes n'est retrouvée). Afin de détecter les substitutions, il est nécessaire d'ajouter à tous les sommets (resp. tous les arcs) une étiquette commune l_v (resp. l_e) en plus de leur étiquette d'origine. Ces étiquettes permettent alors de savoir si un sommet (resp. un arc) est apparié ou non. Si l'étiquette l_v d'un sommet u (resp. l'étiquette l_e d'un arc (u, v)) n'appartient pas à $descr(G) \sqcap_m descr(G')$, alors le sommet u (resp. l'arc (u, v)) a été supprimé ou inséré. A contrario, lorsque cette étiquette l_v (resp. l_e) est retrouvée, soit il y a eu substitution – et l'étiquette d'origine de u (resp. (u, v)) n'est pas retrouvée – soit il n'y a pas eu d'opération de transformation sur u (resp.

(u, v)). La fonction de pondération f peut être choisie de telle sorte que les poids reproduisent fidèlement les coûts de tous les types de transformation élémentaire.

6.2.5 Comparaison avec d'autres appariements de graphes multivoques

Deux articles récents ont introduit des mesures de similarité proches de la notre dans le sens où elles sont basées sur des appariements multivoques (autorisant la mise en correspondance d'un sommet d'un graphe avec un ensemble de sommets de l'autre graphe) : [AFB03] a proposé une distance d'édition étendue et [BRB04] a proposé une mesure de similarité basée sur un appariement non bijectif. Nous montrons ici que l'on peut définir les fonctions f et g de telle sorte que l'on retrouve ces mesures.

Distance d'édition étendue. Afin de comparer des images sur- et sous- segmentées, [AFB03] propose une distance d'édition de graphe étendue autorisant deux nouvelles opérations par rapport à la distance d'édition classique : l'éclatement de sommets —reliant un sommet de G à plusieurs sommets de G'— et la fusion de sommets —reliant plusieurs sommets de G à un même sommet de G'.

Etant donné un appariement m, l'ensemble des couples $(v, s_v) \in splits(m)$ tels que $v \in G$ correspond aux opérations d'éclatement de sommets et l'ensemble des couples $(v', s_{v'})$ tels que $v' \in G'$ correspond aux fusions de sommets. Dés lors, si la fonction g est définie comme une fonction de pondération où les poids correspondent aux coûts d'éclatement et de fusion de sommets et si la fonction f et les graphes sont définis comme pour la distance d'édition non étendue, alors l'appariement qui maximise la formule 6.2 permet de calculer la distance d'édition étendue telle que définie dans [AFB03].

Similarité basée sur un appariement non bijectif. Dans [BRB04], l'appariement de graphes est utilisé pour comparer des images segmentées du cerveau à un modèle du cerveau. Le modèle, d'aspect schématique, est « correctement » segmenté alors que les images, bruitées, sont généralement sur-segmentées. Il n'existe donc pas de bijection entre les régions du modèle et les régions de l'image. Les auteurs utilisent une mesure de similarité entre deux images basée sur des appariements multivoques où un sommet du schéma peut être apparié à plusieurs sommets de l'image. Plus précisément, étant donné un graphe modèle $G = (V, E)$ et un graphe image $G' = (V', E')$, les appariements autorisés sont définis par une fonction $\phi : V \rightarrow \wp(V')$ associant à chaque sommet de V un ensemble non vide de sommets de V' et telle que (i) chaque sommet de V' est associé à exactement un sommet de V, (ii) quelques couples de sommets (v, v') jugés trop différents sont interdits (*i.e.*, $v' \notin \phi(v)$) et (iii), le sous-graphe de G' induit par chaque ensemble $\phi(v)$ doit être connexe (afin de ne fusionner que des régions adjacentes). Un poids (éventuellement négatif) $s^v(v_i, v'_i)$ (resp. $s^e(e_i, e'_i)$) est associé à chaque couple de sommets $(v_i, v'_i) \in V \times V'$ (resp. à chaque couple d'arcs $(e_i, e'_i) \in E \times E'$). L'objectif est de trouver un appariement respectant les contraintes et maximisant la somme des poids des couples de sommets et d'arcs appariés.

Les fonctions f et g peuvent être définies de telle façon que l'appariement m qui maximise la fonction 6.3 corresponde au meilleur appariement au sens de [BRB04]. On ne détaille pas ici la définition de f et g qui est un peu fastidieuse, l'ensemble des contraintes à prendre en compte étant assez important. Plus de détails peuvent être trouvés dans [SS05].

Discussion. Les mesures de similarité de graphes proposées dans [AFB03] et [BRB04] sont basées sur des appariement multivoques et ont été toutes deux introduites pour des applications en reconnaissance d'images afin de prendre en compte les problèmes de sur-segmentation : les appariements multivoques permettent d'apparier une région d'une image sous-segmentée à plusieurs régions d'une image sur-segmentée.

Ces deux mesures sont spécifiques aux problèmes pour lesquelles elles ont été définies. Par exemple, [BRB04] cherche à comparer une image réelle avec sa représentation schématique. Dans ce contexte, une région de l'image réelle doit être appariée à une et une seule région du schéma alors qu'une région du schéma peut correspondre à plusieurs régions de l'image réelle. La mesure de similarité proposée et les algorithmes de résolution ont été construits autour de ces contraintes spécifiques et sont difficilement adaptables à un autre contexte.

A contrario, la mesure de similarité que nous proposons est générique et les fonctions f et g permettent d'exprimer les contraintes et les préférences propres à un problème donné (*e.g.* la recherche d'un appariement univoque ou multivoque, l'évaluation de la qualité d'un appariement...). Un algorithme permettant de calculer cette similarité peut donc être utilisé dans de nombreuses applications sans effort d'adaptation. En contrepartie de cette généricité, cet algorithme sera sans doute moins efficace que des algorithmes dédiés capables d'exploiter les connaissances dépendantes du domaine d'application pour accélérer la recherche du meilleur appariement.

6.3 Algorithmes pour mesurer la similarité de graphes

Etant donnés deux graphes multi-étiquetés G_1 et G_2, on considère maintenant le problème de calculer leur similarité maximum, i.e., trouver l'appariement m qui maximise l'équation 6.2. Notons que le dénominateur $f(descr(G_1) \cup descr(G_2))$ de cette équation ne dépend pas d'un appariement : il est introduit afin de normaliser le degré de similarité entre zéro et un. Il est donc suffisant, pour déterminer la similarité maximum entre deux graphes, de trouver l'appariement m qui maximise la fonction

$$score(m) = f(descr(G_1) \sqcap_m descr(G_2)) - g(splits(m))$$

Ce problème est fortement combinatoire : il est plus général que le problème $\mathcal{NP}$-difficile de recherche de plus grand sous-graphe commun.

Ce problème peut être formulé sous la forme d'un problème de satisfaction de contraintes valué (VCSP) [SFV95] : on associe une variable à chaque sommet de G_1, son domaine étant l'ensemble des parties de l'ensemble des sommets de G_2 ; les contraintes expriment le fait que chaque caractéristique de G_1 et de G_2 doit appartenir à l'ensemble des caractéristiques communes aux deux graphes $descr(G_1) \sqcap_m descr(G_2)$; la valuation des contraintes est définie par les fonctions de pondération. Cependant, dans cette modélisation, le domaine de chaque variable comporte $2^{|V_2|}$ éléments de sorte que la recherche risque fort de ne pas terminer en un temps « raisonnable ».

L'espace de recherche du problème étant composé de tous les appariements possibles des deux graphes, c'est-à-dire de tous les sous-ensembles de $V_1 \times V_2$, il peut être structuré en treillis par rapport à la relation d'inclusion. Ce treillis pourrait être exploré de façon complète grâce à une approche de type « séparation et évaluation », mais il faudrait pour cela disposer d'une bonne fonction d'évaluation permettant, à partir d'un nœud du treillis de trouver une borne maximale de la qualité de tous ses nœuds descendants. Malheureusement, la fonction $score$ n'est pas monotone par rapport à l'inclusion, i.e., le score d'un appariement peut diminuer ou augmenter quand on lui ajoute un nouveau couple de sommets. Cela vient du fait que la fonction $score$ est définie comme une différence de deux fonctions toutes deux croissantes lors de l'ajout d'un nouveau couple.

Ainsi, ce problème très général d'appariement de graphes apparait difficile à résoudre par une approche complète, et on propose dans cette partie quatre algorithmes incomplets : un algorithme glouton, un algorithme tabou, un algorithme réactif, et un algorithme ACO.

fonction Glouton(G_1=$\langle V_1, r_{V_1}, r_{E_1}\rangle$, G_2=$\langle V_2, r_{V_2}, r_{E_2}\rangle$)
retourne un appariement $m \subseteq V_1{\times}V_2$
  $m \leftarrow \emptyset$
  itérer
    $cand \leftarrow \{(u_1, u_2) \in V_1{\times}V_2 - m \mid score(m \cup \{(u_1, u_2)\})$ est maximal$\}$
  sortir si $\forall(u_1, u_2) \in cand'$, $score(m \cup \{(u_1, u_2)\}) \leq score(m)$
    $cand' \leftarrow \{(u_1, u_2) \in cand \mid f(look\_ahead(u_1, u_2))$ est maximal$\}$
      où $look\_ahead(u_1, u_2)$=$\{(u_1, v_1, l) \in r_{E_1} \mid \exists v_2 \in V_2, (u_2, v_2, l) \in r_{E_2}\}$
$\cup\{(u_2, v_2, l) \in r_{E_2} \mid \exists v_1 \in V_1, (u_1, v_1, l) \in r_{E_1}\}$
$\cup\{(v_1, u_1, l) \in r_{E_1} \mid \exists v_2 \in V_2, (v_2, u_2, l) \in r_{E_2}\}$
$\cup\{(v_2, u_2, l) \in r_{E_2} \mid \exists v_1 \in V_1, (v_1, u_1, l) \in r_{E_1}\}$
$-descr(G_1) \sqcap_{m\cup\{(u_1,u_2)\}} descr(G_2)$
    choisir aléatoirement un couple (u_1, u_2) dans $cand'$
    $m \leftarrow m \cup \{(u_1, u_2)\}$
  fin itérer
  retourner m

FIG. 6.3 – Recherche gloutonne d'un appariement

6.3.1 Algorithme glouton

La figure 6.3 présente un algorithme glouton utilisant une heuristique simple pour construire rapidement un appariement m de « bonne » qualité. L'algorithme démarre avec un appariement m vide et ajoute, à chaque itération, un couple de sommets dans m selon un principe glouton. On choisit le couple à insérer parmi l'ensemble $cand$ des couples qui maximisent la fonction $score$. Cet ensemble comportant généralement plus d'un candidat, on départage les ex-æquo en anticipant le potentiel des candidats : on calcule l'ensemble $look\_ahead$ des propriétés d'arcs (départs ou arrivées d'arcs) partagées par les deux sommets du couple candidat et n'appartenant pas à $descr(G_1) \sqcap_m descr(G_2)$, et on choisit aléatoirement le prochain sommet à insérer parmi ceux qui maximisent $f(look\_ahead)$. Cet algorithme glouton s'arrête lorsque plus aucun couple n'améliore la fonction score.

Cet algorithme glouton est de complexité polynomiale en $\mathcal{O}((|V_1| \cdot |V_2|)^2)$, sous réserve que le calcul des fonctions f et g s'effectue en temps linéaire. En contrepartie de cette faible complexité, cet algorithme n'est pas complet et ne garantit pas l'optimalité de son résultat. N'étant pas déterministe, cet algorithme peut être exécuté plusieurs fois afin de conserver le meilleur appariement trouvé.

6.3.2 Recherche taboue

L'algorithme glouton retourne un appariement « localement optimal » dans le sens où il ne peut pas être amélioré en ajoutant ou supprimant un seul couple de sommets. Il est néanmoins possible de l'améliorer en ajoutant et en supprimant plusieurs couples de sommets. Nous décrivons maintenant un algorithme de recherche locale qui tente d'améliorer un appariement en explorant son voisinage. Les voisins d'un appariement m sont les appariements qui peuvent être obtenus en ajoutant ou en enlevant un couple de sommets à m, i.e.,

$$\begin{aligned}\forall m \subseteq V \times V', voisinage(m) &= \{m \cup \{(v, v')\} | (v, v') \in (V \times V') - m\} \\ &\cup \{m - \{(v, v')\} | (v, v') \in m\}\end{aligned}$$

fonction Tabou($G = \langle V, r_V, r_E\rangle, G' = \langle V', r_{V'}, r_{E'}\rangle, k, limiteQualite, maxMouv$)
retourne un appariement $m \subseteq V \times V'$
$m \leftarrow Glouton(G, G')$; $best_m \leftarrow m$; $nbMouv \leftarrow 0$
tant que $sim_{best_m}(G, G') < limiteQualite$ **et** $nbMouv < maxMouv$ **faire**
$cand \leftarrow \{m' \in voisinage(m)/sim_{m'}(G, G') > sim_{best_m}(G, G')\}$
si $cand = \emptyset$ **alors**/* *pas d'aspiration* */
$cand \leftarrow \{m' \in voisinage(m)/$le mouvement inverse (de m' à m) n'a pas été réalisé durant les k dernières itérations$\}$
fin si
$cand \leftarrow \{m' \in cand/m'$ est maximal par rapport au critère de Glouton$\}$
choisir aléatoirement $m' \in cand$
$m \leftarrow m'$; $nbMouv \leftarrow nbMouv + 1$
si $sim_m(G, G') > sim_{best_m}(G, G')$ **alors** $best_m \leftarrow m$ **fin si**
fin tant que
retourner $best_m$

FIG. 6.4 – Algorithme Tabou

A partir d'un appariement initial calculé par l'algorithme glouton, l'espace de recherche est exploré de voisin en voisin jusqu'à ce que le meilleur appariement soit obtenu (lorsque la qualité de celui-ci est connue) ou jusqu'à ce que le nombre d'itérations maximum soit atteint. A chaque itération, le prochain voisin à explorer est choisi selon la méta-heuristique taboue : on choisit à chaque fois le « meilleur » voisin (par rapport au même critère que l'algorithme glouton), et pour ne pas cantonner la recherche autour d'un maximum local en ajoutant puis retirant continuellement le même couple de sommets, une liste taboue est utilisée. Cette liste de longueur k mémorise les k derniers mouvements effectués (*i.e.*, les k derniers couples de sommets ajoutés ou supprimés) afin d'interdire un mouvement inverse à un mouvement récemment effectué (*i.e.*, ajouter/supprimer un couple de sommets récemment supprimé/ajouté). Une exception nommée « aspiration » est ajoutée : si un mouvement interdit permet d'atteindre un appariement de meilleur qualité que le meilleur appariement connu jusqu'alors, le mouvement est quand même réalisé. La figure 6.4 décrit l'algorithme tabou du calcul d'une valeur approchée de la formule 6.3.

6.3.3 Recherche réactive

La longueur k de la liste taboue est un paramètre critique et difficile à déterminer : si la liste est trop longue, la diversification de la recherche est trop forte et l'algorithme converge trop lentement ; si la liste est trop courte, l'intensification de la recherche est trop forte et l'algorithme reste autour d'un optimum local et ne trouve plus de meilleures solutions. Battiti et Protasi [BP01] résolvent ce problème grâce à la recherche taboue réactive dans laquelle la longueur de la liste taboue est adaptée dynamiquement pendant la recherche. Si un même appariement est exploré plusieurs fois, la recherche doit être diversifiée. Afin de détecter une telle redondance, la clé de hachage de chaque appariement visité est mémorisée. Quand une collision a lieu dans la table de hachage, la liste taboue est allongée afin de diversifier la recherche. A contrario, quand il n'y a pas eu de collisions durant un certain nombre de mouvements (signe que la recherche est suffisamment diversifiée) la liste taboue est raccourcie. Les clés de hachage pouvant être calculées de façon incrémentale, le coût supplémentaire de ces calculs est négligeable.

L'algorithme réactif nécessite un plus grand nombre de paramètres que sa version non réactive. Il faut en effet définir la longueur de la liste au début de la recherche, définir de combien la liste doit être allongée et de combien elle doit être réduite lors d'une mise à jour de sa longueur ainsi que définir les longueurs extrêmes (minimum et maximum) de cette liste. Des expérimentations sur l'algorithme tabou non réactif permettent de définir certains de ces paramètres : les bornes de la longueur de la liste sont obtenues en mesurant l'efficacité de tabou sur des tailles de listes extrêmes. Les pas d'incrémentation et de décrémentation de la longueur de la liste sont obtenus expérimentalement, la longueur initiale de liste est toujours fixée au minimum.

6.3.4 Algorithme ACO

Le problème de l'appariement de deux graphes $G_1 = \langle V_1, r_{V_1}, r_{E_1} \rangle$ et $G_2 = \langle V_2, r_{V_2}, r_{E_2} \rangle$ est un problème de sélection de sous-ensemble tel que nous l'avons introduit en 5.3.2, et nous pouvons donc le définir par le triplet $(S, S_{consistant}, f)$ tel que

- S contient l'ensemble des couples appariant un sommet de G_1 à un sommet de G_2, i.e., $S = V_1 \times V_2$;
- $S_{consistant} = \mathcal{P}(S)$ contient tous les sous-ensembles de S dans le cas général où les appariement multivoques sont autorisés (si l'on voulait n'autoriser que les appariements univoques, on pourrait supprimer de $S_{consistant}$ tous les sous-ensembles contenant plus d'un couple pour un même sommet) ;
- f est définie par la fonction score.

On peut alors utiliser l'algorithme ACO pour les SS-problèmes introduit en 5.3.3. On adapte cependant la probabilité de choix d'un objet o_i (ligne 6) de l'algorithme 5.3 car nous considérons deux facteurs heuristiques au lieu d'un, i.e.,

$$p_{o_i} = \frac{[\tau_{factor}(o_i, S_k)]^{\alpha} \cdot [\eta 1_{factor}(o_i, S_k)]^{\beta_1} \cdot [\eta 2_{factor}(o_i, S_k)]^{\beta_2}}{\sum_{o_j \in Candidats} [\tau_{factor}(o_j, S_k)]^{\alpha} \cdot [\eta 1_{factor}(o_j, S_k)]^{\beta_1} \cdot [\eta 2_{factor}(o_j, S_k)]^{\beta_2}}$$

où

- $\eta 1_{factor}(o_i, S_k)$ est le premier facteur heuristique et vise à favoriser les couples qui font le plus augmenter la fonction score, i.e.,
$$\eta 1_{factor}(o_i, S_k) = score(S_k \cup \{o_i\}) - score(S_k)$$
- $\eta 2_{factor}(o_i, S_k)$ est le deuxième facteur heuristique qui vise à favoriser les couples qui ont le plus de caractéristiques potentielles dans l'ensemble $look\_ahead$ défini pour l'algorithme glouton, i.e.,
$$\eta 2_{factor}(o_i, S_k) = f(look\_ahead_{S_k}(o_i))$$
- β_1 et β_2 sont deux paramètres qui déterminent l'importance relative de ces deux facteurs.

6.3.5 Résultats expérimentaux

Comparaison de *Glouton*, *Tabou* et *Réactif*

Glouton, *Tabou* et *Réactif* ont été implémentés en C++ par Sébastien SORLIN à l'occasion de son stage de DEA. Nous avons comparé leurs performances dans [SS05] sur trois jeux d'essais différents : une suite de problèmes d'isomorphisme de graphes et de sous-graphes proposés par [FSV01], sept instances du problème d'appariement non-bijectif de graphes proposées par [BRB04], et cent instances du problème d'appariement multivoque de graphes que nous avons générées aléatoirement. Ces différents problèmes ont tous été modélisés en problèmes de mesure de similarité et ont tous été résolus par le même code. Une seule modification du code (facultative) a été réalisée pour les instances du problème de [BRB04]. Cette modification permet d'abstraire les étiquettes afin d'accélérer le calcul de la similarité par rapport à un appariement.

Pour toutes ces expérimentations, nous avons considéré un même paramétrage. Nous avons retenu pour cela le paramétrage qui donnait pour chaque approche les meilleurs résultats en moyenne : pour *Tabou*, la longueur k de la liste a été fixée à 30 ; pour *Réactif*, la longueur minimale (resp. maximale) de la liste taboue a été fixée à $lMin = 10$ (resp. $lMax = 50$), la longueur d'allongement ou de raccourcissement de la liste taboue a été fixée

TAB. 6.1 – Problèmes d'appariements non-bijectifs : comparaison de *LS+*, *Glouton*, *Tabou* et *Réactif*. Chaque ligne donne : les caractéristiques de l'instance (son nom suivi du nombre de sommets et d'arcs de chacun des deux graphes) ; les résultats obtenus par *LS+* (sim1=similarité de la meilleure solution construite pendant la première étape, sim2=similarité de la solution après la deuxième étape de recherche locale et #mvt=nombre de mouvements effectués) ; les résultats obtenus par *Glouton* (sim=similarité de la solution et #c=nombre de contraintes fortes violées par cette solution) ; les résultats obtenus par *Réactif* et *Tabou* (sim=similarité et #mvt=nombre de mouvements).

Instance			*LS+*			*Glouton*		*Réactif*		*Tabou*	
Nom	$(\|V_1\|, \|E_1\|)$	$(\|V_2\|, \|E_2\|)$	sim1	sim2	#mvt	sim.	#c	sim.	#mvt	sim.	#mvt
GM-5	(10,15)	(30,39)	.5168	.5474	19	.5484	2	.5481	3677	.5463	13875
GM-5a	(10,15)	(30,41)	.4981	.5435	13	.5495	0	.5529	22119	.5519	14955
GM-6	(12,42)	(95,1434)	.4122	.4248	320	.4213	0	.4213	0	.4213	0
GM-7	(14,27)	(28,63)	.6182	.6319	12	.6287	3	.6333	17542	.6342	1122
GM-8	(30,39)	(100,297)	.4978	.5186	118	.5190	0	.5210	1223	.5195	1579
GM-8a	(30,42)	(100,297)	.5014	.5222	120	.5225	0	.5245	1207	.5231	1479
GM-9	(50,88)	(250,1681)	.5049	.5187	520	.5197	0	.5199	21345	.5198	61

à $lDiff = 15$ et le nombre de mouvements sans collision dans la table avant de raccourcir la liste taboue a été fixé à $freq = 1000$.

D'une façon générale, il ressort de ces expérimentations que *Glouton* trouve très rapidement des solutions d'assez bonne qualité, et qu'il est assez souvent capable de résoudre les instances de problèmes « faciles ». En revanche, pour les instances plus difficiles, les solutions trouvées par *Glouton* sont nettement moins bonnes que celles trouvées par *Tabou* et *Réactif*. Lorsque l'on compare *Tabou* et *Réactif* on constate que *Réactif* trouve quasiment toujours de meilleures solutions que *Tabou*, les temps d'exécution de ces deux approches étant comparables en moyenne. Lorsque l'on compare les résultats obtenus par *Tabou* et *Réactif* avec différentes valeurs pour les paramètres, on constate en général qu'une variation (même légère) du paramètre k de *Tabou* influence énormément les résultats et que la valeur optimale de k varie d'une instance à l'autre d'un même problème. A contrario, les paramètres de *Réactif* sont assez souvent plus « robustes » dans le sens où de petites variations de ces paramètres ne changent pas significativement les résultats.

Nous ne reportons pas ici tous les résultats expérimentaux de [SS05], mais uniquement ceux sur le problème d'appariement non-bijectif de graphes proposé par [BRB04] et décrit en 6.2.5. Pour ce problème, nous pouvons comparer nos algorithmes avec celui proposé dans [BRB04]. Cet algorithme, appelé *LS+*, procède en deux étapes : dans une première étape, il construit de façon gloutonne un ensemble de 500 solutions pour ne garder que celles qui satisfont toutes les contraintes « dures » (imposant notamment que chaque sommet du schéma soit apparié à un ensemble non vide de sommets appartenant à un sous-graphe connexe) ; dans une deuxième étape, il optimise la meilleure solution trouvée pendant la première étape en effectuant une recherche locale dans l'espace des solutions satisfaisant toutes les contraintes dures selon une stratégie du « premier voisin améliorant » ; la recherche locale est arrétée lorsqu'elle arrive sur un optimum local.

Notons que l'exécution de *Glouton*, *Tabou* et *Réactif* sur les instances de ce problème est complètement déterministe : les poids manipulés par ces instances sont des réels et il n'y a jamais d'ex-æquo. Aucun couple n'est donc choisi aléatoirement lors de la résolution de ces instances par nos algorithmes.

Le tableau 6.1 résume les résultats obtenus par les différents algorithmes comparés. Pour *LS+*, la colonne « sim1 » donne la similarité de la meilleure solution trouvée à l'issue de la première étape de construction gloutonne. Lorsque l'on compare la qualité de ces solutions avec celles construites par *Glouton*, on constate qu'elles sont de moins bonne qualité pour cinq instances sur les sept considérées. Cependant, *Glouton* n'interdit

TAB. 6.2 – Problèmes d'appariements non-bijectifs : impact des paramètres sur *Tabou* et *Réactif*. Chaque ligne donne le numéro de l'instance suivi des résultats obtenus par *Réactif* et *Tabou* lorsqu'il sont paramétrés de façon standard (i.e., $lMin = 10$, $lMax = 50$, $lDiff = 15$ et $freq = 1000$ pour *Réactif* et $k = 30$ pour *Tabou*) puis lorsque leur paramétrage a été optimisé pour l'instance. Pour les résultats avec paramétrage standard, on donne la similarité ; pour les résultats avec paramétrage optimal, on donne la similarité, le nombre de mouvements et les valeurs des paramètres considérées.

Nom	*Réactif*	*Réactif* avec paramétrage optimal			*Tabou*	*Tabou* avec paramétrage optimal		
	sim.	sim.	#mvt	*lMin.lMax.lDiff.freq*	sim.	sim.	#mvt	k
GM-5	.5481	.5548	28358	(10.50.15.750)	.5463	.5463	13875	(30)
GM-5a	.5529	.5597	4195	(10.50.15.750)	.5519	.5558	16013	(20)
GM-6	.4213	.4213	0	(Tous)	.4213	.4213	0	(Tous)
GM-7	.6333	.6354	6744	(10.50.15.500)	.6342	.6344	5228	(35)
GM-8	.5210	.5212	24557	(10.50.10.500)	.5195	.5204	27523	(45)
GM-8a	.5245	.5248	16423	(10.50.10.1000)	.5231	.5240	4259	(50)
GM-9	.5199	.5202	26593	(15.100.20.750)	.5198	.5198	16539	(50)

pas la violation des contraintes « dures », qui sont simplement prises en compte en leur donnant un poids négatif « dissuasif » de sorte que pour deux instances (GM-5 et GM-7) la solution retournée par *Glouton* viole quelques-unes de ces contraintes (et n'est donc pas une solution au sens défini par [BRB04]).

Les colonnes « sim2 » et « #mvt » de *LS+* donnent alors la similarité des solutions obtenues après l'étape de recherche locale, et le nombre de mouvements effectués pour cela. Lorsque l'on compare ces résultats avec ceux de *Réactif* et *Tabou*, qui effectuent une recherche locale à partir de la solution trouvée par *Glouton*, on constate que les solutions trouvées par *Réactif* et *Tabou* sont souvent meilleurs que celles trouvées par *LS+*, mais qu'ils effectuent aussi plus de mouvements. En effet, la recherche locale de *LS+* s'arrête au premier optimum local trouvé tandis que *Réactif* et *Tabou* peuvent ponctuellement dégrader la solution courante pour s'échapper des optima locaux, et trouver de meilleures solutions. Une exception notable est constituée par l'instance GM-6, pour laquelle la solution trouvée par *glouton* n'est améliorée ni par *Tabou* ni par *Réactif*.

Il est difficile de comparer précisément les temps de résolution de *LS+* avec ceux de *Réactif* et *Tabou* car ils n'ont pas été exécutés sur une même machine. Il apparait cependant que ces temps sont « comparables ». Par exemple, pour l'instance 9, la première étape de *LS+* prend 5 secondes et la deuxième 63 secondes sur un Pentium 2 à 450 MHz, tandis que *Réactif* et *Tabou* prennent respectivement 81 secondes et 5 secondes sur un Pentium 4 à 2 GHz.

Le tableau 6.2 donne ensuite les résultats obtenus par *Réactif* et *Tabou* lorsque l'on optimise leur paramétrage pour chaque instance : par défaut, *Réactif* a été exécuté avec $lMin = 10$, $lMax = 50$, $lDiff = 15$ et $freq = 1000$, et *Tabou* avec $k = 30$; on a testé d'autres valeurs pour ces différents paramètres, et on donne pour chaque instance les valeurs des paramètres qui ont permis d'obtenir les meilleures solutions. Ce tableau nous montre qu'on peut améliorer les résultats de *Tabou* en optimisant la longueur de la liste pour chaque instance, et on remarque que d'une instance à l'autre, la longueur optimale de la liste varie de 20 à 50. De même, on remarque que l'on peut améliorer les résultats de *Réactif* en optimisant les valeurs des paramètres pour chaque instance : si les valeurs optimales pour *lMin* et *lMax* sont relativement stables d'une instance à l'autre, on constate que les valeurs optimales pour *lDiff* et surtout *freq* sont beaucoup plus variables. La réactivité de l'algorithme montre ici quelques limites, que l'on peut tout de même relativiser dans la mesure où *Réactif* avec un paramétrage standard obtient souvent de meilleurs résultats que *LS+* ainsi que *Tabou* (y compris lorsque *Tabou* est optimalement paramétré).

Comparaison de *Réactif* et *Ant-SS(Clique)*

Ant-SS(Clique) appliqué à la résolution du problème d'appariement multivoque de graphes a été programmé en C++ par Olfa SAMMOUD à l'occasion de son stage de Master. Nous avons publié dans [SSG05] une première série de résultats comparant expérimentalement *Réactif* et *Ant-SS(Clique)*. Sur les problèmes d'isomorphisme de sous-graphes, les deux approches ont montré des performances complémentaires. En particulier, *Ant-SS(Clique)* apparait plus « robuste » dans le sens où ses résultats d'une exécution à l'autre pour une même instance sont relativement stables alors que les résultats d'une exécution à l'autre de *Réactif* sur une même instance peuvent être très différents : pour une majorité d'instances, le pourcentage de réussite de *Ant-SS(Clique)* a été supérieur à celui de *Réactif*. En revanche, *Réactif* a été capable de résoudre certaines instances qu'*Ant-SS(Clique)* n'a pas pu résoudre. Comme pour les autres problèmes de recherche de sous-ensembles sur lesquels nous avons travaillé (problèmes de satisfaction de contraintes, de recherche de cliques maximums et de sac-à-dos multidimensionnels), on constate que les temps d'exécution de *Ant-SS(Clique)* sont généralement plus longs que ceux de *Réactif*.

Depuis la publication de ces résultats dans [SSG05], nous avons effectué de nouvelles expérimentations afin de comparer les deux stratégies phéromonales *Ant-SS(Clique)* et *Ant-SS(Vertex)* présentées en 5.3. Curieusement, ces premières expérimentations nous ont montré que pour ce problème d'appariement multivoque de graphes, *Ant-SS(Vertex)* trouve d'aussi bons résultats que *Ant-SS(Clique)* (alors que pour les autres problèmes de sélection de sous-ensembles, *Ant-SS(Vertex)* trouve très souvent de moins bons résultats que *Ant-SS(Clique)* en fin d'exécution). Comme *Ant-SS(Vertex)* est clairement plus rapide que *Ant-SS(Clique)*, cela permet à *Ant-SS(Vertex)* de devenir compétitif en terme de temps d'exécution avec *Réactif*.

Nous avons également amélioré l'algorithme *Ant-SS* en intégrant une procédure de recherche locale « simple ». Les premières expérimentations ont montré que cette hybridation permet d'obtenir de meilleurs résultats. Nous avons comparé *Réactif* avec cette version optimisée de *Ant-SS(Vertex)* sur quelques instances du problème d'appariement multivoque générées aléatoirement. Pour certaines instances, *Ant-SS(Vertex)* obtient de meilleurs résultats que *Réactif*, tandis que pour d'autres *Réactif* obtient de meilleurs résultats que *Ant-SS(Vertex)*. Il s'agit là d'une tendance dégagée après quelques expérimentations seulement, et nous ne pouvons pas donner de résultats sur un ensemble d'instances qui soit significatif. Nous préparons actuellement un article plus complet sur ces différents résultats.

6.4 Une contrainte globale pour le problème d'isomorphisme de graphes

6.4.1 Motivation

On s'intéresse plus particulièrement dans cette section au problème de l'isomorphisme de graphes, qui permet de vérifier si deux graphes sont structurellement identiques. Il existe de nombreux algorithmes dédiés à ce problème [Ull76, McK81, CFSV01]. Ces algorithmes sont très efficaces en pratique, même si leur complexité dans le pire des cas est exponentielle. Citons notamment *Nauty* [McK81] qui est, à notre connaissance, le solveur le plus efficace et qui peut résoudre des problèmes sur des graphes de plusieurs milliers de sommets en moins d'une seconde. L'idée de *Nauty* est de calculer, pour chaque sommet v_i d'un graphe, une étiquette unique déterminée à partir d'un ensemble d'invariants de sommets (*i.e.*, un ensemble de caractéristiques décrivant les relations de v_i avec les autres sommets du graphe) et telle que deux sommets ont la même étiquette si et seulement s'ils peuvent être appariés dans une fonction d'isomorphisme.

Si les algorithmes dédiés sont très efficaces pour résoudre des problèmes d'isomorphisme de graphes (malgré leur complexité exponentielle dans le pire des cas), ils ne permettent pas la résolution de problèmes plus généraux tels que des problèmes d'isomorphisme de graphes auxquels des contraintes ont été ajoutées. En particulier, il est fréquent que les sommets et les arcs des graphes soient étiquetés et que l'on recherche une fonction d'isomorphisme respectant certaines contraintes sur ces étiquettes.

Une alternative intéressante à ces algorithmes dédiés est d'utiliser la programmation par contraintes qui fournit un cadre générique pour la résolution des problèmes de satisfaction de contraintes. En effet, nous avons vu en 4.2.2 que les problèmes d'isomorphisme de graphes peuvent être aisément transformés en CSPs. Cependant, cette reformulation décompose la sémantique globale du problème original en un ensemble de contraintes binaires d'arcs (C_{edge}), chacune d'elles exprimant localement la présence ou l'absence d'un arc. Cela a pour conséquence de rendre la programmation par contraintes moins efficace que les algorithmes dédiés. Afin d'améliorer la résolution de ces CSPs, il est possible d'ajouter une contrainte globale *allDiff* exprimant de façon globale que les variables doivent avoir des valeurs différentes deux à deux. [Rég95] propose un algorithme de filtrage dédié à cette contrainte *allDiff* qui se base sur la théorie des couplages pour filtrer plus efficacement le domaine des variables. Cependant, même en ajoutant cette contrainte globale *allDiff*, la résolution d'un problème d'isomorphisme de graphes par la programmation par contraintes reste limitée à de petits graphes.

Aussi proposons-nous maintenant d'introduire une nouvelle contrainte globale pour modéliser les problèmes d'isomorphisme de graphes. Cette contrainte n'est pas sémantiquement globale au sens de [BH03] car elle peut être remplacée par l'ensemble des contraintes binaires présenté en 4.2.2. Cependant, elle permet de considérer les arcs des graphes de façon globale et donc de filtrer plus efficacement l'espace de recherche. Notons que cette contrainte peut être combinée avec la contrainte *allDiff* afin de filtrer encore plus de valeurs.

6.4.2 Quelques propriétés de l'isomorphisme de graphes

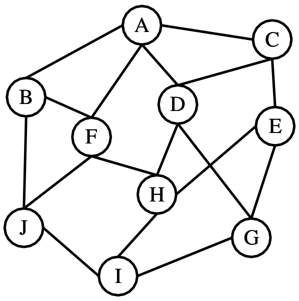
Les méthodes basées sur les invariants de sommets élaguent l'espace de recherche en utilisant le fait qu'une fonction d'isomorphisme n'associe que des sommets « similaires ». Un invariant de sommet est une étiquette $l(v)$ associée à un sommet v telle que, s'il existe une fonction d'isomorphisme appariant v et v', alors $l(v) = l(v')$ (l'inverse n'étant cependant pas toujours vrai). L'exemple le plus simple d'invariants de sommets est le degré d'un sommet : si f est une fonction d'isomorphisme entre $G = (V, E)$ et $G' = (V', E')$, alors pour chaque sommet $v \in V$, les sommets v et $f(v)$ ont le même nombre d'arcs entrants et sortants.

Nous introduisons ici quelques définitions et théorèmes utilisés par la suite pour définir un nouvel invariant de sommets basé sur une propriété de distance entre sommets proche de celle introduite dans [SD76]. Nous focaliserons notre attention sur les graphes non-orientés, i.e., des graphes avec des arêtes non-orientées (les arêtes (u, v) et (v, u) sont considérées comme identiques). L'extension de notre travail aux graphes orientés est discutée en section 6.4.5. Nous supposerons que les graphes sont connexes, i.e., que chaque sommet est accessible à partir de tous les sommets.

Définition 1. Soit un graphe $G = (V, E)$, un *chemin* entre deux sommets u et v est une suite $< v_0, v_1, ..., v_k >$ de sommets telle que $v_0 = u$, $v_k = v$ et telle que pour tout $i \in [1, k]$, $(v_{i-1}, v_i) \in E$. La longueur d'un chemin π, notée $|\pi|$, est son nombre d'arêtes, *i.e.*, k.

Définition 2. Soit un graphe $G = (V, E)$, un *plus court chemin* entre deux sommets u et v est un chemin entre u et v de longueur minimale. La *longueur d'un plus court chemin* entre u et v est notée $\delta_G(u, v)$. Nous dirons que $\delta_G(u, v)$ est la *distance* entre u et v.

La figure 6.5 donne un exemple de graphe avec pour chaque couple de sommets la distance les séparant.



$\delta_G(u,v)$	A	B	C	D	E	F	G	H	I	J
A	0	1	1	1	2	1	2	2	3	2
B	1	0	2	2	3	1	3	2	2	1
C	1	2	0	1	1	2	2	2	3	3
D	1	2	1	0	2	2	1	1	2	3
E	2	3	1	2	0	2	1	1	2	3
F	1	1	2	2	2	0	3	1	2	1
G	2	3	2	1	1	3	0	2	1	2
H	2	2	2	1	1	1	2	0	1	2
I	3	2	3	2	2	2	1	1	0	1
J	2	1	3	3	3	1	2	2	1	0

FIG. 6.5 – Un graphe $G = (V, E)$ et les distances entre chaque couple de sommets.

Théorème 1. Soient deux graphes $G = (V, E)$ et $G' = (V', E')$ tels que $|V| = |V'|$, et une fonction bijective $f : V \rightarrow V'$, les deux propositions suivantes sont équivalentes :

$$f \text{ est une fonction d'isomorphisme, } \textit{i.e.}, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E' \tag{6.4}$$

$$\forall (u, v) \in V^2, \delta_G(u, v) = \delta_{G'}(f(u), f(v)) \tag{6.5}$$

Preuve. $(6.4) \Rightarrow (6.5)$: si f est une fonction d'isomorphisme, alors (u, v) est une arête de G si et seulement si $(f(u), f(v))$ est une arête de G' donc $< v_1, v_2, ..., v_n >$ est un chemin dans G si et seulement si $< f(v_1), f(v_2), ..., f(v_n) >$ est un chemin dans G', et de sorte que $< v_1, v_2, ..., v_n >$ est un plus court chemin dans G si et seulement si $< f(v_1), f(v_2), ..., f(v_n) >$ est un plus court chemin dans G'.
$(6.5) \Rightarrow (6.4)$: Pour chaque paire de sommets $(u, v) \in V \times V$, si (u, v) est une arête de G, alors $< u, v >$ est le plus court chemin entre u et v, $\delta_G(u, v) = 1$ et donc $\delta_{G'}(f(u), f(v)) = 1$. $(f(u), f(v))$ est donc un arc de G' (et vice versa).

Le théorème 1 sera utilisé pour définir les « *contraintes de distance* » permettant la propagation des réductions de domaines lors de la résolution d'un problème d'isomorphisme de graphes par un solveur de contraintes. Nous introduisons maintenant quelques définitions utilisées par la suite pour définir une consistance partielle et un algorithme de filtrage.

Définition 3. Soit un graphe $G = (V, E)$, un sommet $u \in V$ et une distance $i \in [0, |V| - 1]$, nous notons $\Delta_G(u, i)$ l'ensemble des sommets à une distance de i du sommet u et $\#\Delta_G(u, i)$ le nombre de ces sommets, *i.e.*,

$$\Delta_G(u, i) = \{v \in V / \delta_G(u, v) = i\} \text{ et } \#\Delta_G(u, i) = |\Delta_G(u, i)|$$

Par exemple, pour le graphe G de la figure 6.5, nous avons :

$$\begin{array}{ll}
\Delta_G(A, 0) = \{A\} & \#\Delta_G(A, 0) = 1 \\
\Delta_G(A, 1) = \{B, C, D, F\} & \#\Delta_G(A, 1) = 4 \\
\Delta_G(A, 2) = \{E, G, H, J\} & \#\Delta_G(A, 2) = 4 \\
\Delta_G(A, 3) = \{I\} & \#\Delta_G(A, 3) = 1 \\
\Delta_G(A, i) = \emptyset & \#\Delta_G(A, i) = 0, \quad \forall i \geq 4
\end{array}$$

Définition 4. Soit un graphe $G = (V, E)$ et un sommet $u \in V$, nous notons $\#\Delta_G(u)$ la séquence composée de $|V|$ nombres correspondant respectivement aux nombres de sommets à une distance de 0, 1, ... ,$|V| - 1$ de u, *i.e.*,

$$\#\Delta_G(u) = < \#\Delta_G(u, 0), \#\Delta_G(u, 1), ..., \#\Delta_G(u, |V| - 1) >$$

Nous omettrons les zéros présents à la fin de la séquence.
Par exemple, les séquences des sommets du graphe G de la figure 6.5 sont :
$\#\Delta_G(A) = \#\Delta_G(D) = \#\Delta_G(F) = < 1, 4, 4, 1 >$
$\#\Delta_G(B) = \#\Delta_G(C) = \#\Delta_G(E) = \#\Delta_G(G) = \#\Delta_G(I) = \ < 1, 3, 4, 2 >$
$\#\Delta_G(H) = < 1, 4, 5 >$
$\#\Delta_G(J) = < 1, 3, 3, 3 >$

Définition 5. Soit un graphe $G = (V, E)$, nous notons $\#\Delta_G$ l'ensemble de toutes les séquences associées aux sommets de G, *i.e.*,

$$\#\Delta_G = \{s | \exists u \in V, s = \#\Delta_G(u)\}$$

Par exemple, l'ensemble de toutes les séquences présentes dans le graphe G de la figure 6.5 est :

$$\#\Delta_G = \{< 1, 3, 3, 3 >, < 1, 3, 4, 2 >, < 1, 4, 4, 1 >, < 1, 4, 5 >\}$$

Chaque séquence $\#\Delta_G(u)$ caractérise les relations en termes de distance entre le sommet u et les autres sommets de G. Lorsqu'on cherche un isomorphisme de graphes, il est possible d'utiliser ces séquences pour réduire l'espace de recherche en éliminant les appariements qui associent des sommets ayant des séquences différentes. Plusieurs sommets du même graphe peuvent cependant avoir la même séquence, ce critère n'est donc pas toujours suffisant pour élaguer efficacement l'espace de recherche. Par exemple, dans le graphe de la figure 6.5, cinq sommets ont pour séquence $< 1, 3, 4, 2 >$. La définition 6 va une étape plus loin afin de caractériser plus précisément les relations d'un sommet u avec les autres sommets du graphe.

Définition 6. Soit un sommet $u \in V$, nous notons $label_G(u)$ l'ensemble de tous les triplets (i, s, k) tels que i est une distance, s est une séquence, et k est le nombre de sommets distants de i du sommet u et dont la séquence est s, *i.e.*,

$$\begin{aligned} label_G(u) = \{(i, s, k) \ / \quad & i \in [0, |V| - 1], \\ & s \in \#\Delta_G, \text{ et} \\ & k = |\{v \in \Delta_G(u, i) / \#\Delta_G(v) = s\}|\} \end{aligned}$$

Nous ne considérerons que les triplets (i, s, k) tels que $k > 0$.
Par exemple, pour le graphe G de la figure 6.5, nous avons :

$$\begin{aligned} label_G(A) = \{ \quad & (0, < 1, 4, 4, 1 >, 1), \\ & (1, < 1, 3, 4, 2 >, 2), (1, < 1, 4, 4, 1 >, 2), \\ & (2, < 1, 3, 3, 3 >, 1), (2, < 1, 3, 4, 2 >, 2), (2, < 1, 4, 5 >, 1), \\ & (3, < 1, 3, 4, 2 >, 1)\} \end{aligned}$$

car il y a un sommet (A) distant de 0 de A et dont la séquence est $< 1, 4, 4, 1 >$, deux sommets (B et C) distants de 1 de A dont la séquence est $< 1, 3, 4, 2 >$, deux autres sommets (D et F) distants eux aussi de 1 du sommet A mais dont la séquence est $< 1, 4, 4, 1 >$, etc...

Théorème 2. Soient deux graphes $G = (V, E)$ et $G' = (V', E')$. S'il existe une fonction d'isomorphisme $f : V \to V'$ entre les deux graphes G et G', alors, pour chaque sommet $u \in V$, $label_G(u) = label_{G'}(f(u))$.

Preuve. f est une bijection et la distance entre deux sommets u et v de G est égale à la distance entre leur images par f (voir le théorème 1), *i.e.*, $\delta_G(u, v) = \delta_{G'}(f(u), f(v))$. Par conséquent, le nombre de sommets de G distants de i du sommet u est égal au nombre de sommets de G' distants de i du sommet $f(u)$, donc $\#\Delta_G(u) = \#\Delta_{G'}(f(u))$. Par conséquent, les ensembles $\#\Delta$ de séquences des deux graphes sont égaux, *i.e.*, $\#\Delta_G = \#\Delta_{G'}$. Pour chaque séquence $s \in \#\Delta_G$, et pour chaque sommet $u \in V$, le nombre de sommets distants de i du sommet u et dont la séquence est s est égal au nombre de sommets distants de i du sommet $f(u)$ ayant également pour séquence s. Nous avons donc $label_G(u) = label_{G'}(f(u))$.

La réciproque du théorème 2 n'est cependant pas toujours vraie : en effet, il peut exister deux sommets ayant le même label et tels qu'il n'existe aucune fonction d'isomorphisme les reliant l'un à l'autre.

6.4.3 Algorithmes et complexités

Nous discutons dans cette section de la complexité en temps et en espace du calcul des différentes valeurs que nous venons d'introduire. Ces complexités sont données pour un graphe $G = (V, E)$ non-orienté et connexe tel que $|V| = n$ et $|E| = p$ avec $n - 1 \leq p < n^2$.

$\delta_G(u, v)$. Toutes les définitions introduites précédemment sont basées sur les distances entre chaque paire de sommets. Pour calculer ces distances, il est nécessaire d'effectuer un parcours en largeur [CLR90] à partir de chacun des sommets de G : n parcours de complexité en $\mathcal{O}(p)$ sont donc nécessaires. La complexité en temps est donc en $\mathcal{O}(np)$. L'espace nécessaire à la mémorisation des distances est en $\mathcal{O}(n^2)$ (une distance par paire de sommets).

$\Delta_G(u, i)$, $\#\Delta_G(u, i)$ et $\#\Delta_G(u)$. Toutes ces valeurs peuvent être calculées de façon incrémentale pendant le calcul des plus courts chemins : à chaque calcul d'un nouveau $\delta_G(u, v)$, le sommet u (resp. v) est ajouté à l'ensemble $\Delta_G(v, \delta_G(u, v))$ (resp. $\Delta_G(u, \delta_G(u, v))$), les ensembles $\#\Delta_G(u, \delta_G(u, v))$ et $\#\Delta_G(v, \delta_G(u, v))$ sont incrémentés, et les séquences $\#\Delta_G(u)$ et $\#\Delta_G(v)$ sont mises à jour en incrémentant leur $\delta_G(u, v)^{\text{ième}}$ élément. Toutes ces opérations s'effectuent en temps constant. L'espace nécessaire à la mémorisation de ces informations est en $\mathcal{O}(n^2)$ (pour chaque sommet $u \in V$, les ensembles $\Delta_G(u, i)$ forment une partition des n sommets de $|V|$).

$label_G(u)$. Afin de calculer et de comparer efficacement les labels, nous commençons par trier les ensembles de séquences afin qu'un entier unique soit associé à chaque séquence. Cette opération nécessite $\mathcal{O}(n^2.log(n))$ opérations car il y a au plus n séquences différentes et que la comparaison de deux séquences est une opération en $\mathcal{O}(n)$. Une fois cette opération effectuée, le calcul des labels est réalisé en $\mathcal{O}(n^2)$ opérations (il y a n labels à calculer et chaque label contient au plus n différents triplets) et leur mémorisation nécessite une mémoire en $\mathcal{O}(n^2)$ (car chaque label contient au plus n triplets de taille constante si les séquences sont remplacées par un entier unique). Enfin, la comparaison de deux labels se fait en $\mathcal{O}(n)$ opérations, en supposant que les triplets (i, s, k) des labels soient triés.

Par conséquent, le calcul de toutes les valeurs introduites en section 3.1 pour un graphes $G = (V, E)$ nécessite $\mathcal{O}(|V|.|E| + |V|^2.log(|V|))$ opérations et une taille mémoire en $\mathcal{O}(|V|^2)$.

6.4.4 Définition de la contrainte globale

Nous introduisons ici une nouvelle contrainte globale dédiée aux problèmes d'isomorphisme de graphes. Syntaxiquement, cette contrainte est définie par la relation $gip(V, E, V', E', L)$ où
- V et V' sont deux ensembles de valeurs tels que $|V| = |V'|$,
- $E \subseteq V \times V$ est un ensemble de couples de valeurs de V,
- $E' \subseteq V' \times V'$ est un ensemble de couples de valeurs de V',

– L est un ensemble de couples qui associent une variable différente du CSP à chaque valeur de V, *i.e.*, L est un ensemble de $|V|$ couples (x_u, u) où x_u est une variable du CSP, u une valeur de V et pour toute paire de couples (x_u, u) et (x_v, v) différents de L, x_u et x_v sont des variables différentes et $u \neq v$.

Sémantiquement, la contrainte globale $gip(V, E, V', E', L)$ est consistante si et seulement s'il existe une fonction d'isomorphisme $f : V \to V'$ telle que pour chaque couple $(x_u, u) \in L$, il existe une valeur $u' \in D(x_u)$ telle que $u' = f(u)$.

Cette contrainte globale n'est pas sémantiquement globale dans le sens où elle est sémantiquement équivalente à l'ensemble des contraintes binaires décrit en 4.2.2. La contrainte gip nous permet cependant d'utiliser la sémantique globale des problèmes d'isomorphisme de graphes afin de les résoudre plus efficacement. Nous définissons tout d'abord une consistance partielle ainsi que l'algorithme de filtrage qui lui est associé. Nous décrirons ensuite comment propager les contraintes.

Label-consistance et filtrage par labels pour la contrainte gip**.** Le théorème 2 montre qu'une fonction d'isomorphisme n'associe que des sommets qui ont le même label. Nous définissons donc une consistance partielle (la *label-consistance*) pour la contrainte gip, qui garantit que pour chaque couple $(x_i, v) \in L$, le domaine de x_i ne contient que des valeurs ayant le même label que v.

Définition 7. La contrainte globale $gip(V, E, V', E', L)$ est label-consistante si et seulement si :

$$\forall (x_u, u) \in L,\ \forall u' \in D(x_u),\ label_{(V,E)}(u) = label_{(V',E')}(u')$$

Pour rendre la contrainte label-consistante, il suffit de calculer les labels de chaque sommet des deux graphes et d'enlever du domaine de chaque variable x_u associée à un sommet $u \in V$ toutes les valeurs $u' \in D(x_u)$ telles que $label_{(V,E)}(u) \neq label_{(V',E')}(u')$.

Le filtrage par labels réduit souvent très fortement les domaines des variables. Considérons par exemple le graphe G de la figure 6.5. Les trois premiers triplets des labels de chaque sommet (triés par distance, puis par séquence croissante) sont :

$$\begin{array}{ll}
label_G(A) = & \{(0, <1,4,4,1>, 1), (1, <1,3,4,2>, 2), (1, <1,4,4,1>, 2), ...\} \\
label_G(B) = & \{(0, <1,3,4,2>, 1), (1, <1,3,3,3>, 1), (1, <1,4,4,1>, 2), ...\} \\
label_G(C) = & \{(0, <1,3,4,2>, 1), (1, <1,3,4,2>, 1), (1, <1,4,4,1>, 2), ...\} \\
label_G(D) = & \{(0, <1,4,4,1>, 1), (1, <1,3,4,2>, 2), (1, <1,4,4,1>, 1), ...\} \\
label_G(E) = & \{(0, <1,3,4,2>, 1), (1, <1,3,4,2>, 2), (1, <1,4,5>, 1), ...\} \\
label_G(F) = & \{(0, <1,4,4,1>, 1), (1, <1,3,3,3>, 1), (1, <1,3,4,2>, 1), ...\} \\
label_G(G) = & \{(0, <1,3,4,2>, 1), (1, <1,3,4,2>, 2), (1, <1,4,4,1>, 1), ...\} \\
label_G(H) = & \{(0, <1,4,5>, 1), (1, <1,3,4,2>, 2), (1, <1,4,4,1>, 2), ...\} \\
label_G(I) = & \{(0, <1,3,4,2>, 1), (1, <1,3,3,3>, 1), (1, <1,3,4,2>, 1), ...\} \\
label_G(J) = & \{(0, <1,3,3,3>, 1), (1, <1,3,4,2>, 2), (1, <1,4,4,1>, 1), ...\}
\end{array}$$

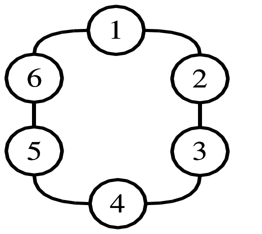
Tous les sommets de G ont des labels différents. Par conséquent, pour toute contrainte gip entre G et un autre graphe G', le filtrage par label détectera une inconsistance (si des labels de G ne sont pas présents dans G') ou réduira le domaine de chaque variable à un singleton rendant alors la consistance globale du problème facile à vérifier.

Nous pouvons comparer la consistance de labels sur la contrainte gip avec la consistance d'arcs sur le CSP défini en 4.2.2. Prenons par exemple un graphe $G' = (V', E')$ isomorphe au graphe $G = (V, E)$ de la figure 6.5, et tel que chaque sommet $u \in V$ est renommé en u' dans G'. Considérons maintenant le CSP « classique » (tel que défini en 4.2.2) associé à ces deux graphes. Dans ce CSP, le domaine $D(x_u)$ de chaque variable x_u contient tous les sommets $u' \in V'$ tels que u et u' ont le même nombre d'arcs, i.e.,

$$\begin{aligned} D(x_A) = D(x_D) = D(x_F) = D(x_H) &= \{A', D', F', H'\} \\ D(x_B) = D(x_C) = D(x_E) = D(x_G) = D(x_I) = D(x_J) &= \{B', C', E', G', I', J'\} \end{aligned}$$

Ce CSP est déja arc-consistant : le filtrage par arc consistance ne reduira donc aucun domaine. Notons aussi que sur cet exemple, ajouter une contrainte *a*llDiff ne permet pas non plus un meilleur filtrage des domaines.

Propagation de contraintes. Le filtrage par labels ne permet pas toujours de réduire le domaine de chacune des variables à un singleton. Considérons par exemple le graphe de la figure 6.6. Ce graphe comporte plusieurs symétries (il est isomorphe à n'importe quel graphe obtenu par une permutation circulaire de ses sommets). Tous les sommets sont donc associés à une même séquence et un même label. Dans ce cas, le filtrage par labels ne réduira aucun domaine.



Pour tous les sommets $u \in V$,
$\#\Delta_G(u) = <1, 2, 2, 1>$ et
$label_G(u) = \{ \; (0, <1,2,2,1>, 1), (1, <1,2,2,1>, 2), (2, <1,2,2,1>, 2), (3, <1,2,2,1>, 1)\}$

FIG. 6.6 – Un graphe $G = (V, E)$ circulaire et les séquences et labels de ses sommets

Quand le filtrage par labels ne réduit par le domaine de chaque variable à un singleton, il est nécessaire d'explorer l'espace de recherche composé de toutes les affectations possibles en construisant un arbre de recherche. A chaque nœud de cet arbre, le domaine d'une variable est découpé en plusieurs sous-domaines, puis des techniques de filtrage relatives à des consistances partielles sont utilisées pour réduire le domaine des variables. Ces techniques de filtrage utilisent les contraintes afin de propager la réduction du domaine d'une variable sur les domaines des autres variables jusqu'à ce qu'un domaine devienne vide (le nœud peut alors être coupé), ou qu'un point fixe soit atteint (soit une solution est trouvée, soit le nœud doit être une fois de plus développé).

Pour utiliser une contrainte gip pour propager les réductions de domaines, une première possibilité consisterait à utiliser l'ensemble des contraintes C_{edge} tel que défini en 4.2.2. Cependant, il est possible de tirer partie des résultats obtenus lors du filtrage par consistance de labels pour définir un ensemble de contraintes plus « fortes », i.e., des contraintes définies par un nombre plus petit (ou égal) de couples de valeurs autorisées. La propagation de ces contraintes permet alors de réduire plus fortement les domaines des variables.

L'idée est de contraindre chaque paire (x_u, x_v) de variables associée à une paire (u, v) de sommets du premier graphe à prendre leurs valeurs parmi l'ensemble des paires de sommets (u', v') du second graphe tels que la distance entre u et v est égale à la distance entre u' et v'. Comme le prouve le théorème 1, une fonction bijective entre les sommets de deux graphes est une fonction d'isomorphisme si et seulement si cette fonction préserve les distances entre chaque paire de sommets des deux graphes. La contrainte globale $gip(V, E, V', E', L)$ est donc sémantiquement équivalente à l'ensemble des « contraintes de distance » définies par : pour tout $((x_u, u), (x_v, v)) \in L \times L$ tel que $u \neq v$,

$$C_{distance}(x_u, x_v) = \{(u', v') \in V' \times V' \mid \delta_{(V,E)}(u, v) = \delta_{(V',E')}(u', v')\}$$

Nous pouvons facilement montrer que chaque contrainte binaire $C_{distance}(x_u, x_v)$ est au moins aussi forte que la contrainte binaire $C_{edge}(x_u, x_v)$ correspondante :

– si les sommets de G associés aux variables x_u et x_v sont reliés par une arête, alors $C_{distance}(x_u, x_v) = C_{edge}(x_u, x_v) = E'$,
– sinon, $C_{distance}(x_u, x_v) \subseteq C_{edge}(x_u, x_v)$ car $C_{edge}(x_u, x_v)$ contient tous les couples de sommets de G' qui ne sont pas connectés par un arc alors que $C_{distance}(x_u, x_v)$ ne contient que les couples de sommets de G' tels que la distance entre leurs sommets est égale à la distance entre les sommets de G associés à x_u et x_v.

Par conséquent, quelle que soit la consistance locale considérée, propager une contrainte $C_{distance}$ permettra de réduire au moins autant les domaines que la propagation de la contrainte C_{edge} correspondante, et, dans certains cas, réduira même plus fortement les domaines.

Considérons par exemple le graphe G de la figure 6.6 et définissons un autre graphe $G' = (V', E')$ isomorphe à G et tel que chaque sommet $u \in V$ est renommé en u' dans G'. Nous notons x_u la variable associée à un sommet $u \in V$. La contrainte d'arc entre x_1 et x_4 contient tous les couples de sommets de G' qui ne sont pas reliés par un arc, i.e.,

$$\begin{aligned} C_{edge}(x_1, x_4) = \{ \quad & (1', 3'), (1', 4'), (1', 5'), (2', 4'), (2', 5'), (2', 6'), \\ & (3', 5'), (3', 6'), (3', 1'), (4', 6'), (4', 1'), (4', 2'), \\ & (5', 1'), (5', 2'), (5', 3'), (6', 2'), (6', 3'), (6', 4')\} \end{aligned}$$

alors que la contrainte de distance entre x_1 et x_4 contient seulement les couples de sommets de G' qui sont à une distance de 3 l'un de l'autre car la distance entre les sommets 1 et 4 est égale à 3, i.e.,

$$C_{distance}(x_1, x_4) = \{(1', 4'), (2', 5'), (3', 6'), (4', 1'), (5', 2'), (6', 3')\}$$

La contrainte de distance entre x_1 et x_4 étant plus stricte que la contrainte d'arc correspondante, elle permet une meilleur propagation des réductions de domaines. Par exemple, si x_1 est assignée à $1'$, une propagation par forward-checking de la contrainte $C_{distance}(x_1, x_4)$ réduit le domaine de x_4 au singleton $\{4'\}$ alors que la propagation par forward-checking de $C_{edge}(x_1, x_4)$ ne réduit le domaine de x_4 qu'à $\{3', 4', 5'\}$, i.e., l'ensemble des sommets qui ne sont pas connectés à $1'$.

De même, lorsque la valeur $1'$ est supprimée du domaine de x_1, une propagation par consistance d'arc de la contrainte $C_{distance}(x_1, x_4)$ permet d'enlever la valeur $4'$ du domaine de x_4 alors que la même propagation de la contrainte $C_{edge}(x_1, x_4)$ ne permet d'enlever aucune valeur.

Une contrainte de distance peut être vue comme un invariant d'un couple de sommets : c'est une étiquette $l(u, v)$ affectée à un couple de sommets (u, v) telle que, s'il existe une fonction d'isomorphisme reliant les sommets (u, v) aux sommets (u', v'), alors $l(u, v) = l(u', v')$. Il est possible d'utiliser des invariants plus performants (i.e., définissant des contraintes plus strictes) que celui basé sur la distance entre deux sommets. Par exemple, il est possible de définir l'étiquette $l'(u, v)$ d'un couple (u, v) de sommets du graphe G comme un ensemble de triplets tels que $(d_u, d_v, n) \in l'(u, v)$ s'il existe n sommets dans G à une distance d_u du sommet u et à une distance d_v du sommet v. Des invariants plus sophistiqués permettent un plus fort filtrage de l'espace de recherche mais peuvent être aussi plus coûteux à calculer et à comparer.

6.4.5 Discussion

Nous avons introduit une nouvelle contrainte globale pour les problèmes d'isomorphisme de graphes. Pour une utilisation efficace de cette contrainte globale, nous avons défini une consistance partielle, la consistance de labels, et un algorithme de filtrage associé, utilisable pour réduire le domaine des variables avant la résolution du CSP. Cette consistance de labels est basée sur le calcul, pour chaque sommet u, d'une étiquette qui caractérise les relations en termes de plus courts chemins entre u et les autres sommets du graphe. Dans de nombreux cas, la consistance de labels permet à un solveur de contraintes de détecter une inconsistance ou de réduire les domaines des variables à des singletons rendant alors la consistance globale du problème facile à vérifier.

Pour les cas où la consistance de labels ne permet pas de résoudre à elle seule le problème de l'isomorphisme de graphes, nous avons défini un ensemble de contraintes de distance, sémantiquement équivalent à la contrainte globale gip, et permettant de propager les réductions de domaines. Nous avons montré que ces contraintes de distances sont plus fortes que les contraintes d'arcs traditionnellement utilisées. Ces contraintes de distance permettent alors de réduire plus fortement les domaines des variables lors d'une propagation de contraintes. Les contraintes de distance peuvent être combinées avec une contrainte globale *allDiff* pour propager encore mieux les réductions.

Comparaison avec la consistance d'arcs sur l'ensemble des contraintes binaires. Le filtrage par labels et la génération de l'ensemble des contraintes de distance peuvent être réalisés en $\mathcal{O}(np + n^2log(n))$ opérations pour les graphes composés de n sommets et p arcs ($n - 1 \leq p \leq n^2$). En comparaison, la consistance d'arcs avec AC2001 sur un CSP décrivant le problème de l'isomorphisme avec des contraintes d'arcs nécessite $\mathcal{O}(ed^2)$ opérations [BC93] où e est le nombre de contraintes, i.e., $e = n(n-1)/2$, et d est la taille du plus grand domaine, *i.e.*, $d = n$.

Etablir la consistance de labels sur notre contrainte globale est donc d'un ordre moins coûteux que la consistance d'arcs sur les contraintes binaires traditionnellement utilisées. Notons cependant que ces consistances ne sont pas comparables : pour certains graphes, tel que le graphe de la figure 6.5, la consistance de labels permet de résoudre le problème alors que AC sur les contraintes d'arcs ne réduit aucun domaine. Inversement, sur le graphe de la figure 6.6, la consistance de labels ne réduit le domaine d'aucune variable alors que la consistance d'arcs permet une réduction des domaines dès qu'une variable est affectée à une valeur.

Extension aux graphes orientés. Les définitions et les théorèmes introduits ici restent valables dans le cas des graphes orientés où les chemins doivent respecter le sens des arcs. Cependant, dans ce cas, il peut exister beaucoup de couples de sommets qui ne sont pas connectés par de tels chemins. Les séquences décrivant les sommets peuvent alors être très courtes, et, dans ce cas, le filtrage par label ne réduira pas beaucoup l'espace de recherche.

Une deuxième façon d'étendre ce travail aux graphes orientés consiste à générer le graphe non-orienté correspondant (en ignorant le sens des arcs) et à calculer les séquences et les labels sur ce graphe non-orienté. Des contraintes peuvent alors être ajoutées pour exprimer le sens des arcs.

Enfin, une dernière façon d'étendre ce travail aux graphes orientés est de considérer en parallèle différentes distances, basées chacunes sur différents chemins, *e.g.*, des chemins orientés, qui respectent le sens des arcs, des chemins non-orientés, qui ignorent le sens des arcs... Chacune de ces distances peut alors être utilisée pour calculer un label. Evidemment, si cette troisième possibilité permet un filtrage plus important des domaines, c'est aussi la plus coûteuse à réaliser.

Chapitre 7

Conclusion

Nous avons présenté dans les deux chapitres précédents nos contributions à la résolution pratique de quelques problèmes combinatoires. Nous avons tout d'abord décrit deux applications de la méta-heuristique d'optimisation par colonies de fourmis ACO. La première application concerne un problème d'ordonnancement de voitures pour lequel nous avons proposé un algorithme ACO combinant deux structures phéromonales complémentaires. Les performances de ce « double » algorithme ACO se sont avérées remarquables tant en rapidité d'exécution qu'en qualité de solution. La deuxième application concerne une classe de problèmes qui regroupe un très grand nombre de problèmes d'optimisation combinatoire, i.e., la classe des problèmes de sélection de sous-ensembles. L'algorithme ACO que nous avons proposé pour cette classe de problèmes a montré des performances honorables, trouvant très souvent les solutions optimales, mais il est généralement plus lent que des approches par recherche taboue.

Nous avons également étudié l'influence de la phéromone sur le processus de résolution d'un algorithme ACO, notamment par rapport à la dualité diversification/intensification. Cette étude, menée grâce à deux indicateurs de diversification —le taux de ré-échantillonnage et le taux de similarité— a montré que la recherche d'un algorithme ACO peut être à la fois « efficace », ne recalculant (quasiment) jamais deux fois une même solution, et « ciblée », intensifiant l'effort de recherche autour des zones les plus prometteuses. L'utilisation de ces indicateurs de diversification permet de détecter dynamiquement des problèmes de stagnation ou de diversification trop fortes. Une perspective de recherche qui nous tient à cœur concerne leur utilisation pour concevoir des algorithmes ACO « réactifs », capables d'ajuster dynamiquement leurs paramètres en fonction de l'évolution de ces deux indicateurs.

Nous nous sommes par ailleurs intéressés à une problématique ayant des applications dans de nombreux domaines, à savoir l'évaluation de la similarité d'objets. Cette problématique nous a amenés à définir un nouveau problème d'optimisation combinatoire : le problème de la recherche du meilleur appariement multivoque de deux graphes. Ce problème est plus général que les problèmes d'appariements de graphes « classiques » comme l'isomorphisme de (sous-)graphes ou la recherche de plus grands sous-graphes communs. Sa résolution est clairement un défi tant par la difficulté du problème que par l'importance de ses applications. Nous avons exploré quelques pistes pour relever ce défi. En particulier, nous avons évalué et comparé les capacités de trois méta-heuristiques, i.e., la recherche taboue, la recherche réactive et l'optimisation par colonies de fourmis, pour résoudre ce problème. Nous avons également proposé un algorithme de filtrage pour résoudre plus efficacement les problèmes d'isomorphisme de graphes à l'aide de la programmation par contraintes.

Ces premiers travaux sur la mesure de la similarité d'objets ouvrent de nombreuses perspectives de recherche. Il s'agit tout d'abord de poursuivre nos efforts pour la résolution pratique du problème de la recherche du meilleur appariement multivoque de deux graphes. La recherche locale et l'optimisation par colonies de fourmis ayant fait preuve de performances complémentaires, nous travaillons actuellement à une hybridation de ces deux approches. Il s'agit par ailleurs d'appliquer ces travaux afin de valider à la fois la pertinence de la mesure que nous avons proposée et les performances de nos algorithmes, notamment en terme de « passage à l'échelle ». Ce dernier point est à l'origine du projet « Graph-I-Sim », qui a commencé en mai 2005 et qui est financé par une ATIP « Jeunes chercheurs » du CNRS. Notre objectif dans ce projet est d'appliquer nos travaux à la recherche de documents décrits par des graphes RDF, et plus particulièrement à la recherche d'images par l'exemple.

Un point que nous souhaitons plus particulièrement explorer dans ce projet concerne l'intégration interactive et dynamique de connaissances de similarité. En effet, les algorithmes que nous avons proposés pour mesurer la similarité de graphes sont paramétrés par des fonctions qui permettent d'exprimer des connaissances de similarité dépendantes de l'application. Cependant lorsque ces paramètres sont modifiés les algorithmes doivent être relancés « depuis le début », i.e., sans utiliser les résultats précédemment trouvés. Un objectif est de concevoir un algorithme capable d'intégrer dynamiquement des connaissances de similarité exprimées par l'utilisateur en fonction de la pertinence des premiers résultats trouvés. Notons que la métaheuristique ACO est particulièrement bien adaptée à la résolution de tels problèmes dynamiques : l'expérience passée accumulée par la colonie de fourmis est « compilée » sous la forme de traces de phéromone ; lorsque certaines données du problème changent, on peut modifier les traces relatives aux données modifiées tout en conservant les autres. A notre connaissance, il n'existe pas de système de recherche d'information permettant d'intégrer dynamiquement des connaissances de similarité. Ce point nous semble important pour que le système puisse retourner des documents pertinents, dans la mesure où la notion de pertinence est généralement partiale et dépend des objectifs de l'utilisateur.

Bibliographie

[AFB03] R. Ambauen, S. Fischer, and H. Bunke. Graph Edit Distance with Node Splitting and Merging, and Its Application to Diatom Identification. In *IAPR-TC15 Workshop on Graph-based Representation in Pattern Recognition*, pages 95–106, 2003.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison Wesley, 1974.

[AK89] E.H.L. Aarts and J.H.M. Korst. *Simulated annealing and Boltzmann machines : a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Chichester, U.K., 1989.

[AKK+97] D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and C. Stamatiou. Random constraint satisfaction : a more accurate picture. In *Proceedings of Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, LNCS, pages 107–120. Springer, 1997.

[Aku95] T. Akutsu. Protein structure alignment using a graph matching technique, 1995.

[APR99] J. Abello, P.M. Pardalos, and M.G.C. Resende. On maximum clique problems in very large graphs. In J.M. Abello, editor, *External Memory Algorithms*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 119–130. American Mathematical Society, Boston, MA, USA, 1999.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 580–592, 1994.

[ASG04] I. Alaya, C. Solnon, and K. Ghédira. Ant algorithm for the multi-dimensional knapsack problem. In *Proceedings of International Conference on Bioinspired Optimization Methods and their Applications (BIOMA 2004)*, pages 63–72, 2004.

[Bal94] S. Baluja. Population-based incremental learning : A method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[BC93] C. Bessière and M.-O. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 108–113, Menlo Park, CA, USA, 1993. AAAI Press.

[BGK02] R. Burke, S. Gustafson, and G. Kendall. A survey and analysis of diversity measures in genetic programming. In *GECCO 2002 : Proceedings of the Genetic and Evolutionary Computation Conference*, pages 716–723, New York, 2002. Morgan Kaufmann Publishers.

[BH03] C. Bessière and P. Van Hentenryck. To be or not to be... a global constraint. *CP'03, Kinsale, Ireland*, pages 789–794, 2003.

[BHS99] B. Bullnheimer, R.F. Hartl, and C. Strauss. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89 :319–328, 1999.

[BJ05] J.-F. Boulicaut and B. Jeudy. Constraint-based data mining. In *The Data Mining and Knowledge Discovery Handbook*, pages 399–416. Springer, 2005.

[Blu02] C. Blum. Ant colony optimization for the edge-weighted k-cardinality tree problem. In *GECCO 2002*, pages 27–34, 2002.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2) :201–236, 1997.

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

[BRB04] M. Boeres, C. Ribeiro, and I. Bloch. A randomized heuristic for scene recognition by graph matching. In *WEA 2004*, pages 100–113, 2004.

[Bun97] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *PRL : Pattern Recognition Letters*, 18, 1997.

[BYV00] R. Baeza-Yates and G. Valiente. An image similarity measure based on graph matching. In *Proc. 7th Int. Symp. String Processing and Information Retrieval*, pages 28–38. IEEE Computer Science Press, 2000.

[CEv03] B.G.W. Craenen, A.E. Eiben, and J.I. van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 7(5) :424–444, 2003.

[CFG+96] D.A. Clark, J. Frank, I.P. Gent, E. MacIntyre, N. Tomv, and T. Walsh. Local search and the number of solutions. In *Proceedings of CP'96, LNCS 1118, Springer Verlag, Berlin, Germany*, pages 119–133, 1996.

[CFSV01] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159. Cuen, 2001.

[Cha02] P.-A. Champin. *Modéliser l'expérience pour en assister la réutilisation : de la Conception Assistée par Ordinateur au Web Sémantique*. Thèse de doctorat en informatique, Université Claude Bernard, Lyon (FR), 2002.

[Chv83] V. Chvatal. *Linear Programming*. W.H. Freeman and Co, 1983.

[CKT91] P. Cheeseman, B. Kanelfy, and W. Taylor. Where the *really* hard problems are. In *Proceedings of IJCAI'91, Morgan Kaufmann, Sydney, Australia*, pages 331–337, 1991.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CS03] P.-A. Champin and C. Solnon. Measuring the similarity of labeled graphs. In *5th International Conference on Case-Based Reasoning (ICCBR 2003)*, volume 2689 of *LNAI*, pages 80–95. Springer-Verlag, 2003.

[DAGP90] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3 :159–168, 1990.

[Dav95] A. Davenport. A comparison of complete and incomplete algorithms in the easy and hard regions. In *Proceedings of CP'95 workshop on Studying and Solving Really Hard Problems*, pages 43–51, 1995.

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

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

[DG97] M. Dorigo and L.M. Gambardella. Ant colony system : A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1) :53–66, 1997.

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

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

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

[DSvH88] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *Proceedings of ECAI-88*, pages 290–295, 1988.

[DT99] A.J. Davenport and E.P.K. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. In *Proceedings of the first international conference on the practical applications of constraint technologies and logic programming (PACLP)*, pages 345–357, 1999.

[EGN05] B. Estellon, F. Gardi, and K. Nouioua. Ordonnancement de véhicules : une approche par recherche locale à grand voisinage. In *Actes des premières Journées Francophones de Programmation par Contraintes (JFPC)*, pages 21–28, 2005.

[Fag96] F. Fages. *Programmation Logique par Contraintes*. Collection Cours de l'Ecole Polytechnique. Ed. Ellipses, 1996.

[For96] S. Fortin. The graph isomorphism problem. Technical report, Dept of Computing Science, Univ. Alberta, Edmonton, Alberta, Canada, 1996.

[FR89] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letter*, 8 :67–71, 1989.

[Fre78] E. Freuder. Synthesizing constraint expressions. *Communications ACM*, 21(11) :958–966, 1978.

[FRPT99] C. Fonlupt, D. Robilliard, P. Preux, and E. Talbi. Fitness landscape and performance of meta-heuristics, 1999.

[FS03] S. Fenet and C. Solnon. Searching for maximum cliques with ant colony optimization. In *Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2003 : EvoCOP, EvoIASP, EvoSTim*, volume 2611 of *LNCS*, pages 236–245. Springer-Verlag, 2003.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 176 –187, 2001.

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

[GAW97] E.J. Gardiner, P.J. Artymiuk, and P. Willett. Clique-detection algorithms for matching three-dimensional molecular structures. *Journal of Molecular Graphics and Modelling 15*, pages 245–253, 1997.

[GGP04] M. Gravel, C. Gagné, and W.L. Price. Review and comparison of three methods for the solution of the car-sequencing problem. *Journal of the Operational Research Society*, 2004.

[GH97] P. Galinier and J.-K. Hao. Tabu search for maximal constraint satisfaction problems. In *Proceedings of CP'97, LNCS 1330, Springer Verlag, Berlin, Germany*, pages 196–208, 1997.

[GJ79] M.R. Garey and D.S. Johnson. *A guide to the theory of NP-Completeness*. Freeman, 1979.

[GJ01] Y.-G. Guéhéneuc and N. Jussien. Quelques explications pour les patrons – Une application de la PPC avec explications pour l'identification de patrons de conception. In *JNPC'2001*, pages 111–122, 2001.

[GK96] F. Glover and G.A. Kochenberger. *Metaheuristics : The Theory and Applications*, chapter Critical Event Tabu Search for Multidimensional Zero-One Knapsack Problems, pages 407–427. Kluwer, 1996.

[GL93] F. Glover and M. Laguna. Tabu search. In C.R. Reeves, editor, *Modern Heuristics Techniques for Combinatorial Problems*, pages 70–141. Blackwell Scientific Publishing, Oxford, UK, 1993.

[GLC04] A. Grosso, M. Locatelli, and F. Della Croce. Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. *Journal of Heuristics*, 10(2) :135–152, 2004.

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

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

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

[GW99] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. Technical report, APES-09-1999, 1999. available from http ://csplib.cs.strath.ac.uk/. A shorter version appears in CP99.

[Hog96] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, pages 127–154, 1996.

[Hog98] T. Hogg. Exploiting problem structure as a search heuristic. *Intl. J. of Modern Physics C*, 9 :13–29, 1998.

[Hol75] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[HR96] T. Asselmeyer H. Rosé, W. Ebeling. Density of states - a measure of the difficulty of optimisation problems. In *Proceedings of PPSN'96*, pages 208–217. LNCS 1141, Springer Verlag, 1996.

[HRTB00] R. Hadji, M. Rahoual, E.G. Talbi, and V. Bachelet. Ant colonies for the set covering problem. In *Proceedings of ANTS 2000*, pages 63–66, 2000.

[HW74] J.E. Hopcroft and J.-K. Wong. Linear time algorithm for isomorphism of planar graphs. *Proc.* 6^{th} *Annual ACM Symp. theory of Computing*, ACM Press :172–184, 1974.

[HW02] A. Hlaoui and S. Wang. A new algorithm for graph matching with application to content-based image retrieval. *Lecture Notes in Computer Science*, 2396, 2002.

[JF95] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of International Conference on Genetic Algorithms, Morgan Kaufmann, Sydney, Australia*, pages 184–192, 1995.

[JL02] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139, 2002.

[JS01] A. Jagota and L.A. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *Journal of Heuristics*, 7(6) :565–585, 2001.

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

[Lin98] D. Lin. An Information-Theoretic Definition of Similarity. In *proceedings of ICML 1998, Fifteenth International Conference on Machine Learning*, pages 296–304. Morgan Kaufmann, 1998.

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

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

[LM99] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problem. In *Proceedings of Congress on Evolutionary Computation*, pages 1459–1464, 1999.

[LMS02] H.R. Lourenco, O. Martin, and T. Stuetzle. *Handbook of Metaheuristics*, chapter Iterated Local Search, pages 321–353. Kluwer Academic Publishers, 2002.

[Luk82] E.M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer System Science*, 25 :42–65, 1982.

[Mar97] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 330–337, 1997.

[Mar02] E. Marchiori. Genetic, iterated and multistart local search for the maximum clique problem. In *Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2002 : EvoCOP, EvoIASP, EvoSTim*, volume 2279 of *LNCS*, pages 112–121. Springer-Verlag, 2002.

[McK81] B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30 :45–87, 1981.

[MF99] P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 245–260. McGraw Hill, UK, 1999.

[MH97] N. Mladenović and P. Hansen. Variable neighborhood search. *Comps. in Oerations Research*, 24 :1097–1100, 1997.

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

[MJ01] R.W. Morrison and K.A. De Jong. Measurement of population diversity. In *5th International Conference EA 2001*, volume 2310 of *LNCS*, pages 31–41. Springer-Verlag, 2001.

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

[Mos89] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts : Towards Memetic Algorithms. Technical Report Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA, 1989.

[MPSW98] E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraints satisfaction : theory meets practice. In *CP98, LNCS 1520*, pages 325–339. Springer Verlag, Berlin, Germany, 1998.

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

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, pages 241–258, 1997.

[NC05] A. Nguyen and V.-D. Cung. Le problème du car sequencing renault et le challenge roadef'2005. In *Premières Journées Francophones de Programmation par Contraintes (JFPC 2005)*, pages 3–10, 2005.

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

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PGCP99] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. BOA : The Bayesian optimization algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, volume I, pages 525–532. Morgan Kaufmann Publishers, San Fransisco, CA, 13-17 1999.

[PR02] L. Pitsoulis and M. Resende. *Handbook of Applied Optimization*, chapter Greedy randomized adaptive search procedures, pages 168–183. Oxford University Press, 2002.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, August 1993.

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

[PV04] C. Pralet and G. Verfaillie. Travelling in the world of local searches in the sapce of partial assignments. In *Proceedings of CP-AI-OR 2004*, volume 3011 of *LNCS*, pages 240–255. Springer Verlag, 2004.

[Rég95] J.-C. Régin. *Développement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application à la Chimie Organique*. PhD thesis, Université Montpellier II, 1995.

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

[SBC01] A. Sidaner, O. Bailleux, and J.-J. Chabrier. Measuring the spatial dispersion of evolutionist search processes : application to walksat. In *5th International Conference EA 2001*, volume 2310 of *LNCS*, pages 77–90. Springer-Verlag, 2001.

[SD76] D.C. Schmidt and L.E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the Association for Computing Machinery*, 23(3) :433–445, 1976.

[SF05] C. Solnon and S. Fenet. A study of aco capabilities for solving the maximum clique problem. *à paraître dans Journal of Heuristics*, 2005.

[SFV95] T. Shiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : Hard and easy problems. In *Proceedings of IJCAI-95, MIT Press, Cambridge, MA*, pages 631–637, 1995.

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

[Shm95] D.B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 20 :355–397, 1995.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI'92, AAAI Press, Menlo Park, California*, pages 440–446, 1992.

[Smi96] B. Smith. Succeed-first or fail-first : A case study in variable and value ordering heuristics. In *third Conference on the Practical Applications of Constraint Technology PACT'97*, pages 321–330, 1996.

[Sol00] C. Solnon. Solving permutation constraint satisfaction problems with artificial ants. In *Proceedings of ECAI'2000, IOS Press, Amsterdam, The Netherlands*, pages 118–122, 2000.

[Sol02a] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.

[Sol02b] C. Solnon. Boosting ACO with a preprocessing step. In *Applications of Evolutionary Computing, Proceedings of EvoWorkshops2002 : EvoCOP, EvoIASP, EvoSTim*, volume 2279 of *LNCS*, pages 161–170. Springer-Verlag, 2002.

[SS04] S. Sorlin and C. Solnon. A global constraint for graph isomorphism problems. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004)*, volume 3011 of *LNCS*, pages 287–301. Springer-Verlag, 2004.

[SS05] S. Sorlin and C. Solnon. Reactive tabu search for measuring graph similarity. In *5th IAPR Workshop on Graph-based Representations in Pattern Recognition*, volume 3434 of *LNCS*, pages 172–182. Springer Verlag, 2005.

[SSG05] O. Sammoud, C. Solnon, and K. Ghédira. Ant algorithm for the graph matching problem. In *5th European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 3448 of *LNCS*, pages 213–223. Springer Verlag, 2005.

[Sta95] P.F. Stadler. Towards a theory of landscapes. In *Complex Systems and Binary Networks*, volume 461, pages 77–163. Springer Verlag, 1995.

[TB05] F. Tarrant and D. Bridge. When ants attack : Ant algorithms for constraint satisfaction problems. *à paraître dans Artificial Intelligence Review*, 2005.

[Tsa93] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[Tve77] A. Tversky. Features of Similarity. *Psychological Review*, 84(4) :327–352, 1977.

[Ull76] J.D. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association of Computing Machinery*, 23(1) :31–42, 1976.

[vHB02] J.I. van Hemert and T. Bäck. Measuring the searched space to guide efficiency : The principle and evidence on constraint satisfaction. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, volume 2439 of *LNCS*, pages 23–32, Berlin, 2002. Springer-Verlag.

[vHS04] J.I. van Hemert and C. Solnon. A study into ant colony optimization, evolutionary computation and constraint programming on binary constraint satisfaction problems. In *Evolutionary Computation in Combinatorial Optimization (EvoCOP 2004)*, volume 3004 of *LNCS*, pages 114–123. Springer-Verlag, 2004.

[ZBMD04] M. Zlochin, M. Birattari, N. Meuleau, and M. Dorigo. Model-based search for combinatorial optimization : A critical survey. *Annals of Operations Research*, 131 :373–395, 2004.